**Eduard Sardyka Java conspectus**

|  |  |
| --- | --- |
| **Topic 1** | **Java Platform** |

***JDK JRE JVM: purpose, differences, instruments***

**Java Platform(JP)** it is a complex of software designed to install up to operating system and to execute programmes written on Java. JP includes Java Virtual Machine, libraries, documentation. It can use additional Java libraries and servers.

Java Platform, **Standard Edition**(Java SE) - designed to use on the small firm level

Java Platform, **Enterprise Edition**(Java EE) - Java SE with special standards and documentation needed to use servers. Java EE designed to use on the enterprise level.

We have to use Java EE server and special Java class libraries to run Java EE application.

Java Platform, **Micro Edition** - designed to use on small devices.

**JDK** - Java Development Kit - free distributed by Oracle co. complect including:

* compiler from Java program code to byte-code (javac)
* standard Java classes library
* examples
* documentation
* utillits
* JRE

**JRE** - Java Runtime Environment - part of JDK, may be distributed independently

contains:

* JVM
* compiled Java class library

**JVM** - Java virtual machine (JVM) - the program that loads and executes byte-code

Purpose of JDK - to develop and support Java code, can compile and run java code

Purpose of JRE - to run bytecode (compiled classes) compiled by javac, it can not compile Java code to bytecode, it only can run byte-code.

Purpose of JVM - to load and execute byte-code classes.

IDE to develop Java code:

* Eclipse
* Idea
* NetBeans
* text editor

***How to run Java console app without IDE ? Commands, arguments, hints, variables***

We have to download JDK from Oracle site, install it and add JAVA\_HOME with name of Java directory in path system variable.

We can make Java code without IDE using only a simple text editor (Notepad++):

**class Simple {**

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

**printHello(args);**

**}**

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

**for(int i = 0; i < args.length; i++) {**

**System.out.println(args[i]);**

**}**

**System.out.println("Hello!");**

**}**

**}**

We can save it as Simple.java, than we can compile it using Win PowerShell and command calling javac.exe:

PS C:\Users\Eduard\_Sardyka\Desktop\Workspace> javac Simple.java

The  javac command makes Java class in directory C:\Users\Eduard\_Sardyka\Desktop\Workspace

with name Simple.class

If Java code has some errors javac will not compile Java code and will display error position:

PS C:\Users\Eduard\_Sardyka\Desktop\Workspace> javac Simple.java

Simple.java:10: error: ';' expected

               System.out.println("Hello!")

                                           ^

1 error

We can run Simple.class using Win PowerShell:

PS C:\Users\Eduard\_Sardyka\Desktop\Workspace> java Simple

Hello!

We can see the printed string “Hello!”.

We can give string arguments to the method main after the ‘java Simple’ command:

PS C:\Users\Eduard\_Sardyka\Desktop\Workspace> java Simple Eduard Sardyka

Eduard

Sardyka

Hello!

***Additional JDK tools***

appletviewer - запуск и отладка апплетов без использования веб-браузера,

extcheck – утилита для обнаружения jar конфликтов,

java – запуск (launcher) приложений,

javac – компилятор языка java,

javadoc – генератор API документации,

jar – создание и управление JAR (java archive) файлами,

javah – используется для написание native методов,

javap – дизассемблер из байт-кода в Java код,

policytool – GUI инструмент для управления файлами политики безопасности,

native2ascii – конвертирует текст в Unicode Latin-1,

rmic - generate stubs and skeletons for remote objects,

rmiregistry - remote object registry service,

rmid - RMI activation system daemon,

serialver - return class serialVersionUID,

javafxpackager – упаковывает JavaFX приложения для развертывания,

pack200 – преобразует JAR файлы в сжатый pack200 используя Java gzip compressor. Такие сильно сжатые JAR файлы могут быть напрямую задеплоины, сохраняя пропускную способность и уменьшая время загрузки.

unpack200 – преобразует упакованный pack200 в JAR,

jcmd - JVM Diagnostic Commands tool – отправляет диагностические команды запросов в исполняемую JVM,

jconsole – JMX-совместимый графический инструмент для мониторинга JVM. Может контролировать и локальную и удаленную JVM. Также может контролировать и управлять приложениями,

jmc - Java Mission Control (JMC) клиент содержит инструменты для мониторинга и управления приложениями без каких-либо дополнительных расходов производительности,

jvisualvm – графический инструмент, предоставляющий полную информацию о том, что происходит с приложением, пока работает JVM (Java VisualVM provides memory and CPU profiling, heap dump analysis, memory leak detection, access to MBeans, and garbage collection),

javaw - Java Web start tool,

Java service tools: schemagen, wsgen, wsimport, xjc

***\*.java \*.class files***

Имя класса в java является case-sensitive. В имени класса допускаются буквы(не обязательно английские), цифры, знак доллара $, нижнее подчеркивание (\_). Имя класса должно начинаться с буквы, знака доллара или нижнего подчеркивания. С цифры имя класса начинаться не может. Файл с расширением java пишется с использованием семантики языка Java и минимально включает слово class или interface (для интерфейса) или enum (для энама), название класса, открывающие и закрывающие фигурные скобки.

Помимо этого этот файл может содержать:

* название пакета в котором находится класс (package by.firm.app.part;)
* статические и нестатические импорты из других пакетов (import  by.firm.app.converter.StringConverter;)
* перед названием класса модификаторы (public, final, abstract),
* после названия слово extends, указывающее на наследование от какого-либо другого класса и слово implements, указывающее на то, что класс реализует методы из указанного(ых) далее интерфейса(ов),
* комментарии (/\*commented\*/, //commented)
* аннотации (@Override, @Entity, @Table, @Component),
* переменные класса и объекта примитивных и объектных типов с различными модификаторами (private, public, protected, static, final, volatile, transient) (private int i; static String str; private OtherClass obj;)
* статические и логические блоки инициализации (static { str = “Java”; } { i = 77; }),
* конструкторы с параметрами и без них с модификаторами (private, protected, public, final) (public SomeClass ( int i ) { super(); this.i = i } ),
* статические и нестатические методы с модификаторами (public, protected, private, static, final, abstract, synchronized, native) с указанием типа возвращаемого значения примитивного или объектного типа (или void - без возвращаемого значения), названия метода и списка параметров или без них и может содержать слово throws со списком возможных выбрасываемых методом исключений (public static void main(String … args) { /\* код метода \*/ }, public String getStr() { return str; })
* может включать другие классы, интерфейсы и энамы

Компилятор javac преобразует файл c расширением java в файл с расширением class, который может быть исполнен виртуальной машиной Java (JVM). JVM читает байт-код из файла с расширением class, загружает класс, интерпретирует его под используемую операционную систему и пошагово выполняет.

***\*.class file structure***

Согласно https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html

структура \*.class следующая:

ClassFile {  
    u4             magic;  
    u2             minor\_version;  
    u2             major\_version;  
    u2             constant\_pool\_count;  
    cp\_info        constant\_pool[constant\_pool\_count-1];  
    u2             access\_flags;  
    u2             this\_class;  
    u2             super\_class;  
    u2             interfaces\_count;  
    u2             interfaces[interfaces\_count];  
    u2             fields\_count;  
    field\_info     fields[fields\_count];  
    u2             methods\_count;  
    method\_info    methods[methods\_count];  
    u2             attributes\_count;  
    attribute\_info attributes[attributes\_count];  
}

1. magic: 0xCAFEBABE – первые 4 байта, уникально идентифицирует формат класс-файла.
2. minor\_version, major\_version (Version of Class File Format): 4 байта, версия формата класс файла (minor and major). Позволяет JVM проверять и идентифицировать класс-файл. Если a class у файла есть номер основной версии М. и номер вспомогательной версии м., мы обозначаем версию class формат файла как M.m.  Реализация виртуальной машины Java может поддерживать class формат файла версии v, если и только если v находится Мi.0 https://lh3.googleusercontent.com/tpGcO9TuH4f4ccofd4SjXqgzPg5lVqJ7BkZAXxe4oDMuaHHvu0uXw4eR9iKy0ZTfIXf7IkYW61yUiWXh2xoE5_IPLDshoxAvFejeWumMV6XekUBA8KVsPpxMHooY22fYiQvMk7B4v https://lh3.googleusercontent.com/miz63tMSQHojCRYesOw039-41tg4IEl1ZavrC3_aKDZTGJKPC3rgxoFJLCfMxOpnVFmsBWaZ3tPAAIxrcDM2fSPPD19793cI_pqQRYr5J4lwTlx6NrcW6BrIV4B4nY9CRSITmFn3Mj.m.
3. constant\_pool( constant\_pool\_count - количество элементов до 65535): пул для констант класса или интерфейса, включая : имена классов, имена переменных, имена интерфейсов, имена и сигнатуры методов, значения final переменных, строковые литералы и др. Константы представлены как массив определенной длины. Длина массива говорит JVM сколько констант будет ожидаться на этапе загрузки. Первый байт каждого элемента в массиве содержит тэг, определяющий тип константы.
4. аccess\_flags: 2 байта – класс/интерфейс, модификаторы доступа класса (abstract, static, etc).

|  |  |  |  |
| --- | --- | --- | --- |
| Flag Name | Значение | Описание | Где использовался |
| ACC\_PUBLIC | 0x0001 | Видимый для всех | Класс, Метод, Переменная |
| ACC\_PRIVATE | 0x0002 | Видимый только для определяемого класса | Метод, Переменная |
| ACC\_PROTECTED | 0x0004 | Видимый для подклассов | Метод, Переменная |
| ACC\_STATIC | 0x0008 | Переменная или метод статические | Метод, Переменная |
| ACC\_FINAL | 0x0010 | Нет дальнейшей подкласификации, обхода или присваивания после инициализации | Класс, Метод, Переменная |
| ACC\_SYNCHRONIZED | 0x0020 | Использует возврат в блокировке монитора | Метод |
| ACC\_VOLATILE | 0x0040 | Не может помещать в кэш | Переменная |
| ACC\_TRANSIENT | 0x0080 | Не может быть написан или прочитан постоянным объектом управления | Перемення |
| ACC\_NATIVE | 0x0100 | Реализован в других языках | Метод |
| ACC\_INTERFACE | 0x0200 | интерфейс | Класс |
| ACC\_ABSTRACT | 0x0400 | Ничего не предусматривает | Класс, Метод |

1. this\_class: 2 байта - название данного класса, т.е. его адрес в пуле констант.
2. super\_class: 2 байта - название суперкласса (так же как и this class).Если значение super\_class не равно нулю, то super\_class должен быть классом, и представляет индекс суперкласса класса в константном пуле. Если значение super\_class равно нулю, то класс должен быть java.lang.Object, и он не имеет суперкласса.
3. interfaces(interfaces\_count имеет размер 2 байта, следовательно, и интерфейсов можно определить 65535): 2 байта - интерфейсы класса - массив, состоящий из индексов на пул констант для каждого интерфейса).
4. fields(fields\_count): поля класса/интерфейса, только те, которые определены в классе или интерфейсе ЭТОГО ФАЙЛА, а не поля, которые унаследованы от суперкласса или интерфейса.
5. methods (count): методы класса (по аналогии с fields – только те методы, которые определены в ЭТОМ ФАЙЛЕ).

*Каждая из этих секций (*Methods и Fields*) имеет размер 2 байта, следовательно, и полей и методов можно определить по 65535.*

1. Attributes count: поле показывает число дополнительных атрибутов этого класса.
2. Attributes: атрибуты класса (Например: имя source файла, из которого этот файл был скомпилирован, хранится в атрибуте source code).

Для примера скомпилируем следующий класс ByteCode.java:

**public class ByteCode {**

**{**

**System.out.println("1-st logic block");**

**}**

**private int intVar = 7;**

**{**

**System.out.println("logic block #2");**

**}**

**static {**

**System.out.println("static block #1");**

**}**

**private static double doubleVar = 5.0;**

**static {**

**System.out.println("static block #2");**

**}**

**public ByteCode() {**

**intVar = 10;**

**}**

**}**

После декомпиляции посредством javap -verbose ByteCode.class получим:

Classfile /C:/Users/Eduard\_Sardyka/Desktop/Workspace/ByteCode.class

 Last modified Dec 28, 2017; size 611 bytes

 MD5 checksum 1ac8f4bbbfff37015ea45109f1677c70

 Compiled from "ByteCode.java"

public class ByteCode

 minor version: 0

 major version: 52

 flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

  #1 = Methodref          #13.#25        // java/lang/Object."<init>":()V

  #2 = Fieldref           #26.#27        // java/lang/System.out:Ljava/io/PrintStream;

  #3 = String             #28            // 1-st logic block

  #4 = Methodref          #29.#30        // java/io/PrintStream.println:(Ljava/lang/String;)V

  #5 = Fieldref           #12.#31        // ByteCode.intVar:I

  #6 = String             #32            // logic block #2

  #7 = String             #33            // static block #1

  #8 = Double             5.0d

 #10 = Fieldref           #12.#34        // ByteCode.doubleVar:D

 #11 = String             #35            // static block #2

 #12 = Class              #36            // ByteCode

 #13 = Class              #37            // java/lang/Object

 #14 = Utf8               intVar

 #15 = Utf8               I

 #16 = Utf8               doubleVar

 #17 = Utf8               D

 #18 = Utf8               <init>

 #19 = Utf8               ()V

 #20 = Utf8               Code

 #21 = Utf8               LineNumberTable

 #22 = Utf8               <clinit>

 #23 = Utf8               SourceFile

 #24 = Utf8               ByteCode.java

 #25 = NameAndType        #18:#19        // "<init>":()V

 #26 = Class              #38            // java/lang/System

 #27 = NameAndType        #39:#40        // out:Ljava/io/PrintStream;

 #28 = Utf8               1-st logic block

 #29 = Class              #41            // java/io/PrintStream

 #30 = NameAndType        #42:#43        // println:(Ljava/lang/String;)V

 #31 = NameAndType        #14:#15        // intVar:I

 #32 = Utf8               logic block #2

 #33 = Utf8               static block #1

 #34 = NameAndType        #16:#17        // doubleVar:D

 #35 = Utf8               static block #2

 #36 = Utf8               ByteCode

 #37 = Utf8               java/lang/Object

 #38 = Utf8               java/lang/System

 #39 = Utf8               out

 #40 = Utf8               Ljava/io/PrintStream;

 #41 = Utf8               java/io/PrintStream

 #42 = Utf8               println

 #43 = Utf8               (Ljava/lang/String;)V

{

 public ByteCode();

   descriptor: ()V

   flags: ACC\_PUBLIC

   Code:

     stack=2, locals=1, args\_size=1

        0: aload\_0

        1: invokespecial #1                  // Method java/lang/Object."<init>":()V

        4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;

        7: ldc           #3                  // String 1-st logic block

        9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V

       12: aload\_0

       13: bipush        7

       15: putfield      #5                  // Field intVar:I

       18: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;

       21: ldc           #6                  // String logic block #2

       23: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V

       26: aload\_0

       27: bipush        10

       29: putfield      #5                  // Field intVar:I

       32: return

     LineNumberTable:

       line 16: 0

       line 3: 4

       line 5: 12

       line 7: 18

       line 17: 26

       line 18: 32

 static {};

   descriptor: ()V

   flags: ACC\_STATIC

   Code:

     stack=2, locals=0, args\_size=0

        0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;

        3: ldc           #7                  // String static block #1

        5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V

        8: ldc2\_w        #8                  // double 5.0d

       11: putstatic     #10                 // Field doubleVar:D

       14: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;

       17: ldc           #11                 // String static block #2

       19: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V

       22: return

     LineNumberTable:

       line 10: 0

       line 12: 8

       line 14: 14

       line 15: 22

}

SourceFile: "ByteCode.java"

PS C:\Users\Eduard\_Sardyka\Desktop\Workspace>

Результат отображает такую структуру:

**public class ByteCode {**

**private int intVar;**

**private static double doubleVar;**

**static {**

**System.out.println("static block #1");**

**doubleVar = 5.0;**

**System.out.println("static block #2");**

**}**

**public ByteCode() {**

**System.out.println("1-st logic block");**

**intVar = 7;**

**System.out.println("logic block #2");**

**intVar = 10;**

**}**

**}**

Таким образом все статические блоки собрались в один, а логические блоки и инициализация полей осуществляется в конструкторе до кода написанного в самом конструкторе. При наличии нескольких конструкторов этот код продублировался бы и в других конструкторах.

|  |  |
| --- | --- |
| **Topic 2** | **Classloaders** |

***What is ClassLoader***

java.lang.ClassLoader - абстрактный класс, наследники которого реализуют загрузку классов Java. При старте JVM грузит Bootstrap classloader, который написан на языке С или C++. Он в свою очередь загружает Extension Class Loader и System Class Loader и размещает их в памяти.

***ClassLoader hierarchy and types***

**Primordial/Bootstrap Classloader** - загружаются основные классы Java из директории JAVA\_HOME/jre/lib (этот путь называют bootstrap path), включая rt.jar. Поэтому, попытка получения загрузчика у классов java.\* всегда заканчивается nullом. Загрузчик загружает только классы, найденные в пути classpath, имеющем высокий уровень доверия. Как результат этого: пропускается большая часть валидации, которая задействована для обычных классов. Управлять загрузкой базовых классов можно с помощью ключа -*Xbootclasspath*, который позволяет переопределять наборы базовых классов.

**Extension Classloader** – загружает классы из стандартных библиотек расширений Java, такие, как функции повышения безопасности. Данный загрузчик позволяет расширять возможности JVM и загружает классы из директории $JAVA\_HOME/lib/ext или в любой другой директории, которая описана в системной переменной java.ext.dirs. Это дочерний загрузчик для Bootstrap class loader.

Управлять загрузкой расширений можно с помощью системной опции java.ext.dirs.

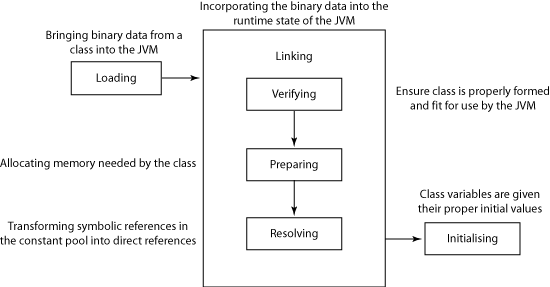
**System Classloader** - загрузчик классов по умолчанию, который загружает классы приложения из classpath - по умолчанию директория (текущая). Системный загрузчик, реализованный уже на уровне JRE. В Sun JRE — это класс sun.misc.Launcher$AppClassLoader. Этим загрузчиком загружаются классы, пути к которым указаны в переменной окружения CLASSPATH. В свою очередь это дочерний загрузчик extension class loader.

Управлять загрузкой системных классов можно с помощью ключа -classpath или системной опцией java.class.path.

**User Defined Classloaders** может использоваться для загрузки классов приложения. Данный загрузчик используется для особой реализации поведения при загрузке. Например, для повторной загрузке (reloading) классов во время выполнения или для разделения между различными группами загружаемых классов (такое обычно требуется веб-серверам, таким, как Томкат). При написании собственного загрузчика классов, необходимо учитывать все три принципа загрузчиков (делегирование, видимость и уникальность).

Каждый ClassLoader видит «свои» классы и классы «родителя». Классы «потомков», ни тем более классы «параллельных» загрузчиков, ClassLoader не видит. Более того, для JVM — это разные классы. При попытке привести один класс к другому вызовет исключение java.lang.ClassCastException, даже если у них и совпадают полные имена. Дело в том, что с точки зрения JVM уникальный идентификатор класса образует пара, состоящая из полного имени класса и загрузчика.

***Description of the process for how class is loading***



**Loading** (выделяется память для класса и создается шаблон класса без начинки) – это процесс поиска класс-файла, который представляет данный класс или интерфейс, с определенным именем и считывание его в массив байт (бинарное представление). Затем байты анализируются для подтверждения того, что они представляют объект класса Class (0xCAFEBABE) и имеют правильную major и minor версии. Также грузится класс или интерфейс, являющийся прямым суперклассом. По завершению данного процесса, создается объект класса или интерфейса из его двоичного представления в области памяти Heap. Этот объект является типом Class, который определен в пакете java.lang. Объект этого класса программист может использовать для получения информация уровня класса, например, имя класса, имя родительского класса, методы класса, переменные и так далее. Чтобы получить все эти данные необходимо вызывать метод getClass() у этого объекта. Для каждого загруженного класса (.class), создается только один объект.

Для каждого .class файла, JVM хранит следующую информацию в области методов (Method Area):

* полное имя загружаемого класса и его непосредственные родительские классы.
* определяет является ли .class файл Java классом, интерфейсом или Enum.
* определяет поля, переменные и информацию о методах и т.д.

**Linking** – процесс верификации и подготовки (verification and preparation) типа класса или интерфейса и его прямого суперкласса и суперитерфейса. Linking состоит из трех этапов: ***verifying***, ***preparing*** и опционально ***resolving***.

***Verifying*** – процесс проверки и подтверждения того, что класс или интерфейс правильно структурирован и удовлетворяет семантическим требованиям языка Java и виртуальной машины (формат и правила Java). Гарантирует корректность .class файлов. Проверяет что конкретный .class был скомпилирован и сгенерирован валидным компилятором. Если проверка не пройдена, то возникает run-time exception java.lang.VerifyError.

Верификация включает в себя проверку:

1. консистентности и правильного форматирования таблицы символов;
2. final методы и классы не переопределяются;
3. методы следуют правилам модификаторов доступа;
4. методы имеют правильное количество и типы параметров;
5. байт-код не манипулирует стеком неправильно;
6. переменные инициализируются до того, как читаются;
7. значения переменных удовлетворяют их типу.

Все эти проверки выполняются на данном этапе (верификации), поэтому не нужно производить данные проверки во время выполнения. Проверка во время связывания замедляет процесс загрузки, однако позволяет избежать необходимости многократного выполнения этих проверок при выполнении байт-кода.

***Preparing*** (выделение памяти под статику) включает в себя процесс распределения памяти для статического хранения и любый структур данных, используемых JVM (таблицы методов). Статические поля создаются и инициализируются значениями по умолчанию, однако на данном этапе не выполняются инициализаторы, статические блоки и код, так как это происходит в**initialization**. Память выделяется или в Permanent Generation (java 7) или в MetaSpace (java 8).

***Resolving*** не является обязательным этапов, который включает в себя проверку ссылок путем загрузки соответствующих классов или интерфейсов и проверки правильности ссылок. Если данный этап опущен, процесс разрешения ссылок может быть отложен до момента их использования инструкцией байт-кода. Этот процесс заменяет символические ссылки на класс (ссылки на Constant Pool) прямыми (реальные адреса объектов классов, которые уже загружены и лежат в Heap). При поиске в области методов обнаруживает зависимые сущности класса.

**Initialization** класса или интерфейса состоит в выполнении инициализирующего метода класса или интерфейса <clinit> (этот метод будет сгенерирован компилятором и будет содержать инструкции для инициализации статических полей класса (инициализаторы + статические блоки инициализации). На самом деле все статические инициализаторы при компиляции превращаются в инициализацию null и после этого следует статический блок инициализации со значением. Аналогично и для логических блоков. При этом, все статические блоки инициализации будут объединены в один). Все статические переменные инициализируются заданными значениями в коде, так же выполняются статические блоки, если такие есть. Выполнение программы происходит сверху вниз, в классе и так же от родителя к наследнику в иерархии классов.

При загрузке класса с помощью метода loadClass(String name, boolean resolve) при resolve = false (что происходит при загрузке класса самой JVM) объекта ClassLoader стадии linking и Initialization не выполняются. Инициализация будет выполнена при первом обращении к классу или к его статическому содержимому. Если использовать метод classForName, то будет выполнена и Initialization инициализации.

Конструкторы, которые будут вызваны при создании объектов класса имеют идентификатор <init>.(Методы с таким идентификатором будет сгенерированы компилятором и будет содержать в себе инструкции из определенных конструкторов, определенных в классе.) Может быть вызван только JVM.

***methods loadClass(), defineClass(), findClass()***

**LoadClass()**

public Class loadClass(String name)

Этот метод загружает класс с заданным именем. На самом деле его реализация сводится к вызову другого protected-метода:

**protected abstract Class loadClass(String name, boolean resolve)**

**throws ClassNotFoundException;**

с resolve = false; переопределение этого protected-метода – один из способов для реализации собственного загрузчика классов.

аналогичен методу **Class.forName(«Имя\_класса», true, loader)**, который динамически грузит класс.

Class.forName выполняет с классом некоторые дополнительные действия, в частности, кэширует его, обеспечивая, в отличие от прямого вызова **loadClass**, стабильную работу даже при недостаточно аккуратной реализации загрузчика loader.

Когда вызывается метод loadClass () класса loader, он имеет два способа выполнить свою работу:

1. заключается в использовании первоначального (Primordial**)** загрузчика классов путем вызова findSystemClass (). В этом случае Primordial загрузчик загрузит, свяжет и инициализирует запрошенный тип
2. loadClass() может создавать бинарные данные по-своему, затем вызывать  defineClass () и - если true передано в loadClass () параметр разрешения вызова - вызов resolveClass (). В этом случае объект загрузчика класса будет производить двоичную форму для запрошенного типа по своему собственному собственному пути, а затем использовать изначальный стандартный механизм загрузчика классов для связи и инициализации типа.

**defineClass()**

protected **final** Class defineClass(String name, byte[] b, int off, int len) throws ClassFormatError

Это как раз то самое место, где цепочка байтов – образ .class-файла (фрагмент массива b длины len по смещению off) – «чудесным образом» превращается в готовый к использованию класс. Метод **defineClass**, как и следовало ожидать, **реализован в native-коде**. Именно он помещает байт-код класса в недра виртуальной машины, где он приобретает вид, пригодный для непосредственного исполнения на конкретной аппаратной платформе, в частности, компилируется в машинный код процессора для более быстрого исполнения (так называемая технология **Just-In-Time**, сокращенно JIT-компиляция).

**findClass()**

protected Class findClass(String name) throws ClassNotFoundException;

метод который вызывается методом loadClass если класс не найден в кэше загрузчика findLoadedClass(name) и всех родительских загрузчиков.

Для **User Defined** загрузчика можно переопределять этот метод.

Он ищет класс (поток  байтов) с заданным именем (name - имя класса).

**@Override**

**public Class<?> findClass(String className) throws ClassNotFoundException {**

**try {**

**/\*\***

**\* Получaем байт-код из файла и загружаем класс в рантайм**

**\*/**

**byte b[] = fetchClassFromFS(pathtobin + className + ".class");**

**return defineClass(className, b, 0, b.length);**

**} catch (FileNotFoundException ex) {**

**return super.findClass(className);**

**} catch (IOException ex) {**

**return super.findClass(className);**

**}**

**}**

***Dynamic class loading***

Статическая загрузка класса происходит при использовании оператора "new".

Динамическая загрузка происходит "на лету" в ходе выполнения программы с помощью статического метода класса Class.forName(имя класса) или loadClass(String name).

**Driver driver = (Driver) Class.forName(driverName).newInstance();**

**Driver driver = (Driver) this.getClass().getClassLoader()**

**.loadClass(driverName).newInstance();**

Для чего нужна динамическая загрузка? Например мы не знаем какой класс нам понадобится и принимаем решение в ходе выполнения программы передавая имя класса в статический метод forName().

***Exceptions during class loading***

**ClassCastException**

Исключение может быть выброшено, если:

* Произведена попытка явна привести объект к типу, экземпляром которого он не является
* ClassLoader, который загрузил класс исходного объекта отличается от ClassLoader, который загрузил целевой класс.

При попытке явного приведения типов ClassLoader проверяет:

* Для обычных объектов(не массивов) Объект должен быть реализацией класса к которому пытаемся привести или одного из его подклассов. Если тип к которому мы пытаемся привести является интерфейсом, то класс объекта должен реализовывать этот интерфейс
* Для массивов. Класс к которому мы пытаемся привести должен быть типа массива или Object или Cloneable или Serializable

Если одно из этих правил нарушается, то выкидывается это исключение.

**ClassNotFoundException**

Исключение выкидывается на этапе загрузки класса.

Объект этого исключения создается если приложение пытается загрузить класс используя строку, представляющую его имя используя:

* forName() method in class Class
* findSystemClass method() in class ClassLoader
* loadClass() method in class ClassLoader

Однако класс файл для класса с таким именем не может быть найден. Выкидывается,когда попытка явно загрузить класс проваливается. Наличие данного класса в Java classpath не проверяется во время компиляции, поэтому даже, если такие классы не присутствуют в classpath программа будет успешно скомпилирована и свалится только во время выполнения.

**NoClassDefFoundError**

Возникает когда определенный класс присутствовал в java CLASSPATH на этапе компиляции, но недоступен на этапе выполнения. (например произошло исключение при инициализации класса в статическом блоке(**ExceptionInInitializerError**)). Ошибка выбрасывается, когда на класс, который не смог быть загружен, потом ссылаются на этапе выполнения. (При компиляции класс был, но на этапе выполнения байт код недоступен(либо файл был перемещен или удален))

**UnsatisfiedLinkError**

Может возникнуть на этапе resolving(разрешения) стадии Linking, когда программа пытается загрузить отсутствующую или неправильно расположенную библиотеку нативных методов(native library). Когда вызывается native метод, класслоадер пытается загрузить библиотеку нативных методов, в которой он определен. Если этой библиотеки нет, выкидывается исключение. Загрузка нативной библиотеки инициализируется загрузчиком классов того класса в котором произведен вызов метода System.loadLibrary().

* Для классов загруженных BootStrap класслоадером просматривается директория на которую указывает sun.boot.library.path.
* Для классов, загруженных Extension класслоадером сначала просматривается java.ext.dirs, затем sun.boot.library.path, а затем java.library.path.(соответственно директории на которые указывают эти свойства)
* Для классов, загруженных Application класслоадером просматривается директория на которую указывает sun.boot.library.path, а затем java.library.path

**ClassCircularityError**

Класс или интерфейс не могут быть загружены, так как это будут их собственные суперкласс или суперинтерфейс. Может появиться если отдельно скомпилировать классы, а потом собрать их вместе.

**ClassFormatError**

Исключение выкидывается если бинарные данные, которые представляют собой скомпилированный класс или интерфейс плохо форматированы. (Нарушена структура class файла) Выкидывается на этапе verification стадии Linking. Бинарные данные могут быть плохо форматированы, если байт код был изменен(например изменены major и minor версии) Может возникнуть, если байт код был сознательно взломан или произошла ошибка при передачи файла по сети. Единственный способ разрешения проблемы - получение корректного байт кода путем перекомпилирования.

**ExceptionInInitializerError**

Ошибка выбрасывается на стадии инициализации процесса загрузки класса.

Ошибка выкидывается, если:

* Выполнение статического блока инициализации оборваны выбрасыванием исключения E и если E не является экземпляром типа Error или одного из его подклассов, то создается новый объект типа ExceptionInInitializerError(E) с параметром E и выбрасывается вместо исключения E
* Если JVM попыталась создать новый объект типа ExceptionInInitializerError, но ей это не удалось из-за OutOfMemoryError, то выбрасывается объект OutOfMemoryError.

**VerifyError**

Выбрасывается на этапе верификации, если хорошо форматированный класс файл содержит внутренние противоречия или проблемы безопасности. (Класс файл не прошел верификацию)

***Class unloading***

If the application has no references to the type, then the type canít affect the future course of computation. The type is unreachable and can be garbage collected.

Types loaded through the primordial class loader will always be reachable and never be unloaded. Only dynamically loaded types--those loaded through class loader objects--can become unreachable and be unloaded by the virtual machine. A dynamically loaded type is unreachable if its Class instance is found to be unreachable through the normal process of garbage collecting the heap.

***How cloassloaders wind class bytecode***

1. Bootstrap Classloader загружает Extension Classloader
2. Extension Classloader загружает System ClassLoader
3. System ClassLoader загружает главный класс. Главным является класс, с которого  начинается выполнение нашей программы  (или это  может быть класс, у которого есть  метод main)
4. Возникает необходимость загрузить следующий класс. Другими словами, происходит перовое  упоминание класса в коде.
5. JVM находит текущий загрузчик о котором помнит текущий класс. Текущий класс - это класс, над которым в данный момент работает JVM.
6. Текущий ClassLoader загружает нужный класс в JVM. Текущий ClassLoader - это загрузчик, который загрузил текущий класс.
7. Загрузившийся класс запоминает этот загрузчик как текущий. Всегда можно узнать какой ClassLoader загрузил конкретный объект с помощью метода  getClassLoader().
8. Текущий ClassLoader помещает класс себе в кэш. Если во время работы программы ему понадобиться работать с этим классом, то он будет взят  из кэша, т.е. обращений к диску за этим классом, пока работает программа,  не будет.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Whence possible load classes**     - из собственной системы каталогов не отраженной в classpath (или отраженной)     - из БД     - из Internet     - можно сконструировать байт-код в памяти и работать с ним  **Сhronology call methods when loading the classes**   1. вызов loadClass  loadClass(String name, boolean resolve) -   загружает класс с заданным именем name - имя класса boolean resolve - to call resolveClass ()  2. loadClass вызывает findClass  findClass(String name) - ищет класс (поток  байтов) с заданным именем. name - имя класса  3. findClass вызывает defineClass  defineClass - final method получает массив байтов, найденный методом  findClass, и конвертирует его в экземпляр  класса Class  Рассмотрим процесс загрузки более детально. Пусть в системе исполнения встретилась декларация переменной пользовательского класс Student.  1) Системный загрузчик попытается поискать в кеше класс Student.  1.1) Если класс найден, загрузка окончена.  1.2) Если класс не найден, загрузка делегируется загрузчику расширений.  2) Загрузчик расширений попытается поискать в кеше класс Student.  2.1) Если класс найден, загрузка окончена.  2.2) Если класс не найден, загрузка делегируется базовому загрузчику.  3) Базовый загрузчик попытается поискать в кеше класс Student.  3.1) Если класс найден, загрузка окончена.  3.2) Если класс не найден, базовый загрузчик попытается его загрузить.  3.2.1) Если загрузка прошла успешно, она закончена ;)  3.2.2) Иначе управление передается загрузчику расширений.  3.3) Загрузчик расширений пытается загрузить класс.  3.3.1) Если загрузка прошла успешно, она закончена ;)  3.3.2) Иначе управление передается системному загрузчику.  3.4) Системный загрузчик пытается загрузить класс.  3.4.1) Если загрузка прошла успешно, она закончена ;)  3.4.2) Иначе генерируется исключение java.lang.ClassNotFoundException.  Если в системе присутствуют пользовательские загрузчики, они должны  а) расширять класс java.lang.ClassLoader;  б) поддерживать модель динамической загрузки.  ПРИНЦИП ВИДИМОСТИ Дочерний загрузчик классов может видеть классы, загруженные родителем, но не наоборот. Т.е. если класс был загружен Application ClassLoader и дальше попытаться явно загрузить его Extension ClassLoader, будет сгенерировано исключение java.lang.ClassNotFoundException.  ПРИНЦИП УНИКАЛЬНОСТИ Класс, загруженный Родителем, не может быть повторно загружен Ребенком.  Хотя вполне возможно написать загрузчик классов, который нарушает принципы делегирования и уникальности и сам загружает класс, так не рекомендуется делать. Класс должен быть загружен только один раз!   |  |  | | --- | --- | | **Topic 3** | **Classes** |   ***Classes (class as file vs class as type)***  Файл содержащий неоткомпилированный класс Java с именем SomeClass.java может содержать информацию не только об этом классе (class SomeClass { }), но и содержать информацию о других внешних (не public) и/или внутренних классах. Имя внутри файла должно совпадать с названием класса, т.е. в файле SomeClass.java должен быть класс или интерфейс SomeClass.  По названию файла SomeClass.java как правило нельзя установить какого типа класс(классы) в нем находятся.  Классы Java можно относить к разным типам на основе разных критериев:   * типу доступа (public, friendly-package) * абстрактный (abstract) или имеющий возможность создания * финальный (final) или имеющий возможность создания классов-наследников * обычный (внешний) класс или внутренний - находящийся внутри кода другого, возможна многократная вложенность * static - для внутренних вложенных (nested) классов или нестатический (называется inner, не имеет модификатора static) * внутренний класс находящийся внутри логического блока кода или метода и не имеющий модификатора доступа и не видимый за пределами этого блока кода - не может быть static * анонимные (anonymous) классы - создающиеся одновременно с созданием их объекта с помощью new SomeClass(){/\*здесь описывается функциональность отличная от класса-предка - SomeClass() \*/}; * по возможности изменения полей после создания объекта. Если невозможно, то immutable * по типу(с параметрами или без) и доступности конструктора класса, возможно наличие нескольких конструкторов. * классы только хранящие информацию или только оперирующие ею или смешанного типа * по наличию определенных полей и методов   Создадим файл ClassFileTest.java:  **public class ClassFileTest {**  **static {**  **class StaticBlockClass { }**  **}**  **{**  **class LogicBlockClass { }**  **}**  **ClassFileTest () {**  **class InConstructorClass { }**  **}**  **public static void main(String [] args) {**  **class StaticMethodClass { }**  **new Object() {**  **public String toString() { return "str"; }**  **};**  **}**  **public void method() {**  **class MetodClass { }**  **new Object() {**  **public String toString() { return "str"; }**  **};**  **}**  **public static class NestedClass { }**  **public class InnerClass { }**  **}**  **class OuterClassOne { }**  после компиляции (javac ClassFileTest.java) получим класс-файлы со следующими названиями:  ClassFileTest$1InConstructorClass.class  ClassFileTest$1LogicBlockClass.class  ClassFileTest$1MetodClass.class  ClassFileTest$1StaticBlockClass.class  ClassFileTest$1StaticMethodClass.class  ClassFileTest$InnerClass.class  ClassFileTest$NestedClass.class  ClassFileTest$1.class  ClassFileTest$2.class  ClassFileTest.class  OuterClassOne.class  Видно, что:   * в названии всех внутренних класс-файлов присутствует имя внешнего * в названии всех класс-файлов находящихся внутри блоков кода присутствует название внешнего + $1 + имя внутреннего; $2, $3, .. было бы если бы названия внутренних классов совпадали * в названии всех класс-файлов анонимных классов присутствует название внешнего + $ + номер анонимного класса (1, 2, … ) * в названии inner и nested класс-файлов присутствует название внешнего + $ + имя внутреннего * внешние класс-файлы имеют свое название из java-кода   ***Class syntax***  Синтаксис языка Java исторически унаследован от С++ и внешне очень схож. Основные отличия от классов C++: все функции определяются внутри классов и называются методами; невозможно создать метод, не являющийся методом класса, или объявить метод вне класса.  ***Class structure (fields, methods, blocks..)***  В начале  файла с Java кодом описывается пакет:   * название пакета в котором находится класс (package by.firm.app.part;). Пакеты представляют собой взаимно вложенные файлы (part внутри app, app внутри firm, firm внутри by, by внутри директории src проекта) * статические и нестатические импорты из других пакетов (import  by.firm.app.converter.StringConverter;)   Все описание класса (включая все поля, логические и статические блоки, конструкторы и методы)  находится между открывающей и закрывающей скобками   * перед названием класса модификаторы (public, final, abstract), * после названия слово extends, указывающее на наследование от какого-либо другого класса и слово implements, указывающее на то, что класс реализует методы из указанного(ых) далее интерфейса(ов), * комментарии (/\*commented\*/, //commented) * аннотации (@Override, @Entity, @Table, @Component), * поля переменных класса и объектов примитивных и объектных типов с различными модификаторами (private, public, protected, static, final, volatile, transient) (private int i; static String str; private OtherClass obj;) * статические и логические блоки инициализации (static { str = “Java”; } { i = 77; }), * конструкторы с параметрами и без них с модификаторами (private, protected, public, final) (public SomeClass ( int i ) { super(); this.i = i } ), * статические и нестатические методы с модификаторами (public, protected, private, static, final, abstract, synchronized, native) с указанием типа возвращаемого значения примитивного или объектного типа (или void - без возвращаемого значения), названия метода и списка параметров или без них и может содержать слово throws со списком возможных выбрасываемых методом исключений (public static void main(String … args) { /\* код метода \*/ }, public String getStr() { return str; }) * может включать другие классы и энамы   ***access modifiers***  **public, private, protected** воздействуют только на те объявления полей, мето-  дов и классов, перед которыми они стоят, а не на участок от одного до другого  спецификатора, как в С++; **элементы** **по умолчанию** не устанавливаются в private,  а **доступны для классов из данного пакета** (т. е. без модификатора доступа - **friendly, package -** так называется, но не пишется).  **public** определяет внешний (enclosing) класс, как доступный из других пакетов, класс без модификатора доступа доступен только внутри пакета.  **Поля и методы** public класса с модификатором доступа **public** доступны из других пакетов. Все поля и методы внутри интерфейса всегда public по умолчанию.  **Поля и методы** помеченные **protected** доступны только в пределах пакета и в классах-наследниках  **Поля и методы** помеченные **private** доступны только внутри класса  ***static content***  Модификатором **static** помечаются поля, методы, логические блоки которые являются принадлежащими классу в целом, а не каждому отдельно созданному объекту. Статические поля класса инициализируются при первом обращении к любому из полей или методов класса. Все статические поля при этом получают значение по умолчанию (для объектных переменных null, для примитивов: byte, short, int, long, float, double 0; char '\u0000', булевые false) а затем те которые им присваиваются в статических блоках инициализации. Помеченные static внутренние классы - называются nested.  ***Inner classes***  Одной из серьезных причин использования внутренних классов является возможность быть подклассом любого класса независимо от того, подклассом какого класса является внешний класс. Фактически при этом реализуется ограниченное множественное наследование со своими преимуществами и проблемами.  **Inner** Нестатические вложенные классы принято называть внутренними (inner). Доступ к элементам внутреннего класса возможен из внешнего только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса. Объект внутреннего класса всегда ассоциируется (скрыто хранит ссылку) с создавшим его объектом внешнего класса — так называемым внешним (enclosing) объектом. Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса. Доступ будет разрешен по имени в том числе и к полям, объявленным как private. **Внутренние** классы **не могут содержать статические атрибуты и методы**, **кроме констант** (final static). Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования.  При объявлении внутреннего класса могут использоваться модификаторы  final, abstract, private, protected, public. Внутренний класс может быть объявлен как **private**, что обеспечивает его полную невидимость вне класса-владельца. Создать объект такого класса можно только в методах и логических блоках внешнего класса. Использование protected позволяет получить доступ к внутреннему классу для класса в другом пакете, являющегося классом-наследником внутреннего класса. Если внутренний класс наследуется обычным образом (т.е. не внутри класса наследующего внешнему) другим классом (после extends указывается ИмяВнешнегоКласса.ИмяВнутреннегоКласса), то он теряет доступ к полям своего внешнего класса, в котором был объявлен.  Создание объекта внутреннего класса возможно на ссылке внешнего:  **EnclosingClass.InnerClass obj = new EnclosingClass().new InnerClass();**  **Внутренний класс** может быть объявлен также **внутри метода или логического блока** внешнего (owner) класса. Видимость такого класса регулируется областью видимости блока, в котором он объявлен. Но внутренний класс сохраняет доступ ко всем полям и методам внешнего класса, а также ко всем константам, объявленным в текущем блоке кода. Класс, объявленный внутри метода, не может быть объявлен как static, а также не может содержать статические поля и методы. Локальный класс не может иметь модификаторов доступа.  **Nested** - с модификатором **static -** вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него. Если класс вложен в интерфейс, то он становится статическим по умолчанию. Такой класс способен наследовать другие классы, реализовывать интерфейсы и являться объектом наследования для любого класса, обладающего необходимыми правами доступа. В то же время статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую имеет доступ только к статическим полям и методам внешнего класса. Для создания объекта вложенного класса объект внешнего класса создавать нет необходимости. Подкласс вложенного класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс. Статический метод вложенного класса вызывается при указании полного относительного пути к нему.  Создание объекта вложенного класса:  **EnclosingClass.NestedClass obj = new EnclosingClass.NestedClass();** т.е конструктор внешнего класса не вызывается.  **Anonymous** (безымянные) классы применяются для придания уникальной функциональности отдельно взятому экземпляру, для обработки событий, реализации блоков прослушивания, реализации интерфейсов, запуска потоков и т. д. Можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении одного-единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора new:  **SomeExistingClass obj = new SomeExistingClass(){**  **public void someExistingMethod() {**  **/\* new activity in overriden method \*/**  **}**  **public void newMethod() {/\* something new \*/ }**  **};**  Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Этот прием эффективен в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области (или одноразового) применения метода.  Конструктор анонимного класса определить невозможно.  Для перечисления объявление анонимного внутреннего класса выглядит несколько иначе, так как инициализация всех элементов происходит при первом обращении к типу. Поэтому и анонимный класс реализуется только внутри объявления типа enum.  **public enum En {**  **ONE, TWO { public int num() { return 2; } };//overridden method**  **public int num() { return 1; }**  **}**  ***Interface and Abstract Class***  Абстрактные методы размещаются в абстрактных классах (с модификатором abstract) или интерфейсах, тела у таких методов отсутствуют и должны быть реализованы в подклассах. Абстрактные методы помечаются модификатором abstract. Интерфейс (имеют модификатор interface вместо class) задает в общем поведение класса-наследника. Реализация его методов предполагается в классах-наследниках которые помечаются implements. Класс-наследник интерфейса может реализовывать методы нескольких интерфейсов и наследовать какому-либо классу. В языке Java существует два вида интерфейсов: интерфейсы, определяющие функциональность для классов посредством описания методов, но не их реализации; и интерфейсы, реализация которых автоматически придает классу определенные свойства. К последним относятся, например, интерфейсы Cloneable и Serializable, отвечающие за клонирование и сохранение объекта в информационном потоке соответственно. Данные интерфейсы указывают, что класс относится к определенной группе классов. Степень абстракции в данном случае доведен до абсолюта. В интерфейсе вообще нет никаких объявлений:  java.lang.Cloneable  java.util.EventListener  java.io.Serializable  java.rmi.Remote  В отличии от интерфейсов абстрактные классы могут иметь реализованное поведение - такие методы не помечаются abstract. При отказе от реализации некоторых методов класс-наследник абстрактного должен быть объявлен абстрактным. В интерфейсах изначально все методы являлись абстрактными (это подразумевается но не пишется - public abstract), а все поля public final static. В Java 8 в интерфейсах могут быть статические методы (с модификатором static) - они не могут быть абстрактными и дефолтные методы - имеющие дефолтную реализацию, т.е. при отсутствии в классе-наследнике переопределенного метода будет исполняться дефолтный метод интерфейса (модификатор default).    **interface InterfaceWithMain {**  **double publicStaticFinalVar = Math.random();**  **// this field is public final static!**  **static void staticMethod(int var) {//public**  **System.out.println("Interface static method");**  **}**  **default void notAbstractMethod() {//public**  **System.out.println(publicStaticFinalVar + "defaultMethod");**  **}**  **void abstractMethod();// this method is public abstract**  **static void main(String[] args) {// all methods are public!**  **System.out.println(InterfaceWithMain.publicStaticFinalVar);**  **InterfaceWithMain n1 = new NotAbstract() {**  **@Override**  **public void notAbstractMethod() {**  **//Overridden default method**  **System.out.println(**  **publicStaticFinalVar + "OverridenMethod");**  **}**  **};**  **n1.notAbstractMethod();**  **NotAbstract n2 = new NotAbstract();**  **n2.notAbstractMethod();**  **n2.abstractMethod();**  **staticMethod(7);**  **InterfaceWithMain.staticMethod(7);**  **n2.staticMethod(7);//illegal static access!**  **}**  **}**  **class NotAbstract implements InterfaceWithMain {**  **@Override**  **public void abstractMethod() {**  **System.out.println("OverridenAbstractMethod");**  **}**  **public static void staticMethod(int var) {**  **//it is not Overridden - it calls by Class's name**  **System.out.println("New static method");**  **}**    **@Override**  **public void notAbstractMethod() {**  **System.out.println(**  **publicStaticFinalVar + "OverridenNotAbstractMethod");**  **InterfaceWithMain.super.notAbstractMethod();**  **}**  **}**  ***Enums***  В отличие от статических констант, предоставляют типизированный, безопасный способ задания фиксированных наборов значений. Являются классами специального вида, не могут иметь наследников, сами в свою очередь наследуются от java.lang.Enum и реализуют java.lang.Comparable (следовательно, могут быть сортированы) и java.io.Serializable. Перечисления не могут быть абстрактными и содержать абстрактные методы (кроме случая, когда каждый объект перечисления реализовывает абстрактный метод), но могут реализовывать интерфейсы. Итератор возвращает элементы перечисления в порядке их объявления в самом enum. Каждый класс перечисления неявно содержит следующие методы:  int ordinal() – возвращает позицию элемента перечисления.  String toString()  boolean equals(Object other)  static enumType[] values() — возвращает массив, содержащий все элементы перечисления в порядке их объявления;  static <T extends Enum<T>> T valueOf(Class<T> enumType, String arg) — создает элемент перечисления, соответствующий заданному типу и значению передаваемой строки;  static enumType valueOf(String arg) — создает элемент перечисления, со- ответствующий значению передаваемой строки;  String name() — возвращает имя элемента;  int compareTo(T obj) — сравнивает элементы на больше-меньше либо равно. **СОЗДАНИЕ ОБЪЕКТОВ ПЕРЕЧИСЛЕНИЯ** Экземпляры объектов перечисления **нельзя создать** с помощью **new**. При инициализации хотя бы одного перечисления происходит инициализация всех без исключения оставшихся элементов данного перечисления. Можно использовать в switch, if, foreach, еще используют в Map/Set. Перечисление не может быть суперклассом. Поля перечисления используются для сохранения дополнительной информации, связанной с его элементом. Как и обычные классы могут реализовывать поведение, содержать вложенные классы. **Элементы перечисления по умолчанию public, static и final**. Перечисления имеют константные значения. Поэтому их можно сравнивать на == и это даже более предпочтительно, чем сравнения элементов перечисления через equals (не появится NullPointerException) Метод equals у интерфейсов реализован как у Object, т.е.(= =) Элементы перечислений удобно использовать в switch. **КОНСТРУКТОРЫ** Модификаторы доступа в конструкторе: **private**, написанный явно, или же, если **ничего не указывать** – **все** **равно** будет **private**. Все элементы должны сразу реализовывать их. Объявление методов, конструкторов и полей перечисления должно находиться только после объявления элементов перечисления, которое в этом случае должно заканчиваться точкой с запятой  **public enum EnumTest {**  **ONE(1), TWO(2), THREE(3);**  **public int num;**  **EnumTest(int num) {//default private**  **this.num = num;**  **}**  **public int getNum() {**  **return num;**  **}**  **public static void main(String[ ] args) {**  **EnumTest et = EnumTest.ONE;//to init**  **System.out.println (et);**  **switch (et) {**  **case ONE:**  **System.out.println("It is ONE: " + EnumTest.ONE.getNum());**  **break;**  **case TWO:**  **System.out.println("It is TWO: " + EnumTest.TWO.getNum());**  **break;**  **case THREE:**  **System.out.println("It is THREE: " + EnumTest.THREE.getNum());**  **break;**  **default:**  **System.out.println(" Unknown: " + et);**  **}**  **}**  **}**  АНОНИМНЫЕ КЛАССЫ ПЕРЕЧИСЛЕНИЯ Отдельные элементы перечисления могут реализовывать свое собственное поведение.    ***Constants in Java***  Константы и перечисления пишутся в верхнем регистре.  Константа может быть объявлена как поле класса, но не проинициализирована. В этом случае она должна быть проинициализирована в логическом блоке класса, или конструкторе, но только в одном из указанных мест. Значение по умолчанию константа получить не может в отличие от переменных класса. Константы могут быть объявлены в методах как локальные или как параметры метода. В обоих случаях значения таких констант изменять нельзя. Константа объекта класса - final, самого класса - final static  **public class SomeClass {**  **protected final int OBJECT\_CONST = 1;**  **public final static String CLASS\_CONST;**  **static { CLASS\_CONST = “A”; }**  **public static void method(final int PARAM\_CONST) {**  **final double METHOD\_CONST = 0.0;**  **//….**  **}**  **}**  ***Primitive Data types***  Типы данных в Java платформенно-независимые. Элементарных, или простых, типов всего восемь. Особенности примитивных типов:   * Размер примитивных типов одинаков для всех платформ; за счет этого становится возможной переносимость кода * Размер boolean не определен. Указано, что он может принимать значения true или false. * Преобразования между типом boolean и другими типами не существует. Неинициализированная явно переменная (член класса или член экземпляра класса) примитивного типа принимает значение по умолчанию в момент создания.   https://lh6.googleusercontent.com/-RfYSpIfFxmoaRRCDqQMEh4vNxV0OoBnODbm_MYxqlHTV7Bt7rgS5MJMBK841uxAZGLruv8oflOBjJdGo-QGV4v4mjCldMMUvjZFu3bv5SFJ5hn5rhuy7lKoVo3wDGCxHBTJCwhy  Литералы с плавающей точкой записываются в виде 1.75 или в экспоненциальной форме 0.123E-12 и относятся к типу double. Таким образом, действительные числа относятся к типу double. Если необходимо определить литерал типа float, то в конце литерала следует добавить символ F или f. По стандарту IEEE 754 введены понятие бесконечности +Infinity и –Infinity и значение NaN (Not a Number), которое может быть получено, например, при извлечении квадратного корня из отрицательного числа.  Символьные литералы определяются в апострофах ('a', '\n', '\141', '\u005a'). Для размещения символов используется формат Unicode, в соответствии с которым для каждого символа отводится два байта. В формате Unicode первый байт содержит код управляющего символа или национального алфавита, а второй байт соответствует стандартному ASCII коду, как в C++. Любой символ можно представить в виде '\ucode', где code представляет двухбайтовый шестнадцатеричный код символа. Java поддерживает управляющие символы, не имеющие графического изображения; '\n'– новая строка, '\r' — переход к началу, '\f' — новая страница, '\t'– табуляция, '\b' — возврат на один символ, '\uxxxx' — шестнадцатеричный символ Unicode, '\ddd'– восьмеричный символ и др.  Примитивный тип    Значение по умолчанию boolean                       false char                            '\u0000' (null) byte                            (byte)0 short                           (short)0 int                                0 long                             0L float                             0.0f double                         0.0d  **Особенности работы с переменными**В операциях присваивания нельзя присваивать переменной значение более длинного типа, в этом случае необходимо явное преобразование типа. Исключение составляют операторы инкремента «++», декремента «– –» и сокращенные операторы (+=, /= и т. д.). При явном преобразовании (тип)значение возможно усечение значения.  Преобразование примитивных типов Различают: - повышающее (разрешенное, неявное) преобразование int i = 2; long num = i; - понижающее (явное) приведение типа. int i = 777; byte b = (byte)i;//byte - диапазон значений уже чем у int Расширяющее (повышающее) преобразование: результирующий тип имеет больший диапазон значений, чем исходный тип. Сужающее (понижающее) преобразование: результирующий тип имеет меньший диапазон значений, чем исходный тип. Неявное (повышающее) преобразование: повышающее преобразование осуществляется автоматически, даже в случае потери данных. В арифметических выражениях автоматически выполняются расширяющие преобразования типа byte -> short -> int -> long -> float -> double.  Приведение типов в выражении При вычислении выражения (a @ b) аргументы a и b преобразовываются в числа, имеющие одинаковый тип:  если одно из чисел double, то в double;  иначе, если одно из чисел float, то в float;  иначе, если одно из чисел long, то в long;  иначе оба числа преобразуются в int. Арифметическое выражение над byte, short или char имеет тип int, поэтому для присвоения результата обратно в byte, short или char понадобится явное приведение типа.  При преобразовании int, long в float и float в double возможна потеря точности.  ***Methods in Java (return, params, varargs)***     * модификатор доступа(public, protected или friendly-package-без модификатора) * static - если метод класса - тогда он может оперировать только со статич полями и методами класса, для работы с объектом класса его необходимо создать в методе/ если метод не static он может обращаться ко всем другим статич и нестатическим полям и методам * synchronized - для статич метода - синхронизирован на классе и всех статич полях/ для не статич - на конкретном объекте и его полях * если в описании метода присутствует модификатор native, то это означает, что метод реализован на C++, но такой метод можно переопределить в классе-наследнике в Java-коде. * тип возвращаем значения - объектный или примитивный или void при отсутствии возвращаемого значения - в return пишется переменная или примитив такого-же или неявно приводимого типа или совместимый объект-наследник - для объектных типов. Возможно для void написание return; но это не обязательно. * при написании далее в скобках описываются имена параметров с их типами. При передаче параметров  в метод возможна передача приводимых неявно типов. Передаваемое значение копируется в параметр метода. Для объектной переменной копируется ссылка-дискриптор - указатель на объект. * далее после слова throws описываются возможные выбрасывыемые методом типы классов-исключений * в скобках - тело метода с возможными объявлениями локальных переменных и операторами языка Java. Может заканчиваться return   Возникают ситуации, когда заранее неизвестно количество передаваемых параметров в метод. В обычной ситуации пришлось бы создавать несколько перегруженных методов с разным числом параметров одного типа. Другим решением будет один метод с параметром в виде массива или коллекции, что потребует предварительной организации соответствующего объекта массива или коллекции. Начиная с версии Java 5, появилась возможность передачи в метод нефиксированного числа параметров, что позволяет отказаться от предварительного создания сложного объекта для его последующей передачи в метод. Набор объектов, переданный в такой метод, преобразуется в массив с типом и именем, которые указаны в качестве параметров метода. Например: метод printf() с переменным числом аргументов. Список параметров метода выглядит в общем случае: (Тип... args) а в случае необходимости передачи параметров других типов (vararg должен быть написан обязательно в конце!) (Тип1 t1, Тип2 t2, ТипN tn, Тип... args)  **Правила работы:** 1. Приоритет у varargs самый низкий. 2. Можно прописывать параметры через запятую, или передать сразу массив, содержащий данные того же типа. 3. Коллекцию надо преобразовать в массив. 4. Дополнительные параметры объявляются ДО объявления varargs. 5. Метод может содержать только ОДИН параметр типа varargs. 6. При передаче единичного массива при наличии при выборе между Object…arg и Type[]…arg, компилятор выберет Object. Метод с параметром Type[]...args будет вызван для единичного массива только в случае отсутствия метода с параметром Object...args. Вообще предпочтение будет отдано Type…arg. 7. При вызове метода без параметров возникает неопределенность из-за неоднозначного выбора(если несколько перегруженных varargs). Если только один vararg – метод без параметров будет его использовать. 8. Примитив…arg и Object...arg КОНФЛИКТ. Примитив всегда конфликтует с Object и классом-обёрткой. 9. Компилятор для примитивных типов предпочтет Object приведению к классу-обертке. 10. Компилятор выберет сначала производит приведение примитивных типов для поиска подходящего varargs, далее происходит автоупаковка, а уже если его нет –> приводит к Object. Приведение типов объектов классов-упаковок не происходит. Например: The method printArgCount(Long...) in the type Varargs is not applicable for the arguments (int, int).   |  |  | | --- | --- | | **Topic 4** | **Objects** |   ***What is object?***  Объект — именованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение.  Объект — обладающий именем набор данных (полей и свойств объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним.  Любой объект относится к определенному классу. В классе дается обобщенное описание некоторого набора родственных объектов. Объект — конкретный экземпляр класса.  Каждый объект в Java после создания имеет ссылку на себя - this - указатель-дискриптор данного объекта в виртуальной памяти.  ***How to create object in Java ? Different ways***  1. Ключевое слово new  **SomeObject so1 = new SomeObject();**  Может быть таким образом вызван любой конструктор (с параметрами или без).  В байткоде конвертируется в 2 вызова (первый –> new, а второй -> invokespecial (вызов конструктора)).  2. newInstance() на классе Class (рефлексивный способ создания объекта)  **SomeObject so2 = (SomeObject) Class.forName("SomeObject")**  **.newInstance();**  Или  **SomeObject so3 = SomeObject.class.newInstance();**  В байткоде метод конвертируется в вызов invokevirtual, т.е. сам метод управляет созданием объекта.  3. Class Loader и метод newInstance()  **ClassLoader cl = Test.class.getClassLoader(); //получаем ClassLoader**  **Driver driver = (Driver)cl.loadClass(driverName).newInstance();**  LoadClass aналогичен методу Class.forName, который динамически грузит класс.  В байткоде метод конвертируется в вызов invokevirtual, т.е. сам метод управляет созданием объекта.  4. newInstance() класса java.lang.reflect.Constructor (рефлексивный cпособ создания объекта)  **Constructor<Employee> constructor = Employee.class.getConstructor();**  **Employee emp3 = constructor.newInstance();**  5. Метод clone()  **Employee emp4 = (Employee) emp3.clone();**  Во время вызова метода clone() на любом объекте JVM на самом деле создает новый объект и копирует в него все содержимое предыдущего объекта. .По умолчанию происходит не глубокое копирование, что означает, что если поле класса имеет ссылочный тип, то в новом объекте оно будет ссылаться на тот же самый объект, что и поле класса, на котором было произведено клонирование. При создании объекта этим способом не вызывается конструктор. Для использования метода clone() необходимо реализовать интерфейс Cloneable (Marker interface) и определить метод clone(), иначе возникнет исключение CloneNotSupportedException. Имя метода не обязательно должно быть clone(), т.е не обязательно переопределять этот метод из класса Object, метод, создающий копию может иметь любое имя, но это не переопределение. Родительский метод clone() может быть вызван как super.clone().  Для реализации глубокого копирования необходимо правильно реализовать клонирование во всех классах, которые являются полями другого класса, или же создать их экземпляр и скопировать все значения полей в него, а затем в этом классе явно вызвать методы клонирования этих полей.  В байткоде метод конвертируется в вызов invokevirtual, т.е. сам метод управляет созданием объекта.  6. Десериализация  **ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));**  **Employee emp5 = (Employee) in.readObject();**  При десериализации объекта JVM создает отдельный объект. При десериализации для создания объекта не вызывается никакой конструктор. В байткоде метод конвертируется в вызов invokevirtual, т.е. сам метод управляет созданием объекта.  **import java.io.FileInputStream;**  **import java.io.FileOutputStream;**  **import java.io.IOException;**  **import java.io.ObjectInputStream;**  **import java.io.ObjectOutputStream;**  **import java.io.Serializable;**  **import java.lang.reflect.Constructor;**  **import java.lang.reflect.InvocationTargetException;**  **class ObjectCreation {**  **public static void main(String... args) {**  **// By using 'new' keyword**  **SomeObject so1 = new SomeObject();**  **so1.setName("First");**  **System.out.println(so1);**  **// By using Class class's newInstance() method**  **SomeObject so2 = null;**  **try {**  **so2 = (SomeObject) Class.forName("SomeObject")**  **.newInstance();**  **} catch (InstantiationException | IllegalAccessException |**  **ClassNotFoundException e) {**  **e.printStackTrace();**  **}**  **so2.setName("Second");**  **System.out.println(so2);**  **// Or we can simply do this:**  **SomeObject so3 = null;**  **try {**  **so3 = SomeObject.class.newInstance();**  **} catch (InstantiationException | IllegalAccessException e) {**  **e.printStackTrace();**  **}**  **so3.setName("Third");**  **System.out.println(so3);**  **// By using Constructor class's newInstance() method**  **Constructor<SomeObject> constructor = null;**  **SomeObject so4 = null;**  **try {**  **constructor = SomeObject.class.getConstructor();**  **so4 = constructor.newInstance();**  **} catch (InstantiationException | IllegalArgumentException |**  **InvocationTargetException | NoSuchMethodException |**  **SecurityException | IllegalAccessException e) {**  **e.printStackTrace();**  **}**  **so4.setName("Fourth");**  **System.out.println(so4);**  **// By using clone() method**  **SomeObject so5 = (SomeObject) so4.clone();**  **so5.setName("Fifth");**  **System.out.println(so5);**  **// By using Deserialization**  **// Serialization**  **ObjectOutputStream out = null;**  **try {**  **out = new ObjectOutputStream(new FileOutputStream("data.obj"));**  **out.writeObject(so5);**  **out.close();**  **} catch (IOException e) {**  **e.printStackTrace();**  **}**  **//Deserialization**  **SomeObject so6 = null;**  **ObjectInputStream in = null;**  **try {**  **in = new ObjectInputStream(new FileInputStream("data.obj"));**  **so6 = (SomeObject) in.readObject();**  **in.close();**  **} catch (IOException | ClassNotFoundException e) {**  **e.printStackTrace();**  **}**  **so6.setName("Sixth");**  **System.out.println(so6);**  **ClassLoader cl = SomeObject.class.getClassLoader();**  **//get ClassLoader**  **SomeObject so7 = null;**  **try {**  **so7 = (SomeObject)cl.loadClass("SomeObject").newInstance();**  **} catch (InstantiationException | IllegalAccessException |**  **ClassNotFoundException e) {**  **e.printStackTrace();**  **}**  **so7.setName("Seventh");**  **System.out.println(so7);**  **}**  **}**  **class SomeObject implements Cloneable, Serializable {**  **private static final long serialVersionUID = 1L;**  **private String name;**  **public SomeObject() {**  **System.out.println("SomeObject Constructor");**  **}**  **public String getName() {**  **return name;**  **}**  **public void setName(String name) {**  **this.name = name;**  **}**  **@Override**  **public int hashCode() {**  **return 31 \* ((name == null) ? 0 : name.hashCode());**  **}**  **@Override**  **public boolean equals(Object obj) {**  **if (this == obj)**  **return true;**  **if (obj == null)**  **return false;**  **if (getClass() != obj.getClass())**  **return false;**  **SomeObject other = (SomeObject) obj;**  **if (name == null) {**  **if (other.name != null)**  **return false;**  **} else if (!name.equals(other.name))**  **return false;**  **return true;**  **}**  **@Override**  **public String toString() {**  **return "SomeObject [name = " + name + ", hashcode = " + hashCode() + "]";**  **}**  **@Override**  **public Object clone() {**  **Object obj = null;**  **try {**  **obj = super.clone();**  **} catch (CloneNotSupportedException e) {**  **e.printStackTrace();**  **}**  **return obj;**  **}**  **}**  ***Object Lifecycle***  Создание объекта -> ссылка на объект доступна в коде и объект существует -> ссылок на объект в коде далее нет и он может быть уничтожен сборщиком мусора.  protected void finalize() — автоматически вызывается сборщиком мусора  (garbage collector) перед уничтожением объекта - может быть переопределен в классе для освобождения занятых объектом ресурсов.  ***class Object***  Класс Object  является суперклассом для всех классов. Ссылочная переменная типа Object может указывать на объект любого другого класса. При попытке получить  new Object().getClass().getClassLoader(); или new Object().getClass().getSuperclass();  возвращает null:  **class ObjectTest {**  **public static void main(String ... args) {**  **System.out.println(**  **new Object().getClass().getClassLoader());//null**  **System.out.println(**  **new Object().getClass().getSuperclass());//null**  **}**  **}**  ***class Object description***  Класс Object  является суперклассом для всех классов. Все остальные классы скрыто наследуют ему. Код следующего вида не вызовет ошибки компиляции:  **public class SomeClass extends Object { }**  Для любого класса **ObjectTest** не имеющего предка  **new ObjectTest().getClass().getSuperclass().getName()**  вернет java.lang.Object  ***Object class methods***  В классе Object определен набор методов, который наследуется всеми классами:  **protected Object clone()** — создает и возвращает копию вызывающего объекта. Default-реализация: происходит не глубокое копирование, что означает, что если поле класса имеет ссылочный тип, то в новом объекте оно будет ссылаться на тот же самый объект, что и поле класса, на котором было произведено клонирование. Конструктор при этом не вызывается.  Для реализации глубокого копирования необходимо правильно реализовать клонирование во всех классах, которые являются полями другого класса, а затем в этом классе явно вызвать методы клонирования этих полей.  **public boolean equals(Object ob)** — сравнивает содержимое двух объектов, предназначен для использования и переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа. В классе Object реализован так:  **public boolean equals(Object obj) {**  **return (this == obj);**  **}**  При переопределении метода equals() должны выполняться соглашения,  предусмотренные спецификацией языка Java, а именно:  • рефлексивность — объект равен самому себе;  • симметричность — если x.equals(y) возвращает значение true, то и y.equals(x)  всегда возвращает значение true;  • транзитивность — если метод equals() возвращает значение true при сравнении объектов x и y, а также y и z, то и при сравнении x и z будет возвращено значение true;  • непротиворечивость — при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;  • ненулевая ссылка при сравнении с литералом null всегда возвращает значение false  Метод тесно связан с методом hashCode(), т.к. оба работают с содержимым полей. Если содержимое двух объектов одинаковое, то и хеш-коды должны быть одинаковые.  **public Class<? extends Object> getClass()** — возвращает объект типа Class. Возвращает объект класса Class для этого объекта. Объект класса Class представляет собой метаданные об определенном классе.  **protected void finalize()** — автоматически вызывается сборщиком мусора  (garbage collector) перед уничтожением объекта, может быть переопределен в классе для освобождения занятых объектом ресурсов.  Во-первых, нет гарантии, что он будет вызван, т.к. где-то может остаться ссылка на объект.  Во-вторых, нет гарантии на то, в какое время будет вызван метод. Это связано с тем, что после того, как объект становится доступным для сборки и, если в нем переопределен метод finalize, то он не вызывается сразу, а помещается в очередь, которая обрабатывается специально созданным для этого потоком (т.е. удаление присходит в 2 этапа).  Любое исключение, выброшенное в finalize методе, игнорируется GC и не перекидывается никуда дальше.  Существует некоторый способ немного увеличить вероятность вызова finalize метода: System.runFinalization() и  Runtime.getRuntime().runFinalization()  **public int hashCode()** — вычисляет и возвращает хэш-код объекта (число,  в общем случае вычисляемое на основе значений полей объекта). Вычисление управляется следующими соглашениями:  • все одинаковые по содержанию объекты одного типа должны иметь одина-  ковые хэш-коды;  • различные по содержанию объекты одного типа могут иметь различные  хэш-коды;  • во время работы приложения значение хэш-кода объекта не изменяется,  если объект не был изменен.  Если хеш-коды разные, то и входные объекты гарантированно разные.  Если хеш-коды равны (коллизия), то входные объекты не всегда равны.  Исходный код в классе Object:  **public native int hashCode()** - дает хэш-код от дискриптора-указателя на объект в виртуальной памяти  **public String toString()** — возвращает представление объекта в виде строки. Обычно переопределяется так, чтобы возвращать информацию о полях объекта. В классе Object:  **public String toString() {**  **return getClass().getName() + "@" + Integer.toHexString(hashCode());**  **}**  **public final void notify()** - пробуждает один поток, который ждет монитор на данном объекте  **public final void notifyAll()** - пробуждает все потоки, которые ожидают монитор на данном объекте  **public final void wait()throws InterruptedException** - заставляет текущий поток ждать пока другой поток не вызовет notify() или notifyAll() на этом объекте  **public final void wait(long timeout)throws InterruptedException,**  **public final void wait(long timeout, int nanos)throws InterruptedException** - заставляет текущий поток ждать, пока либо другой поток не вызовет notify() метод или notifyAll() метод этого объекта или указанное кол-во времени не пройдет  ***Overview of JMM***  *Java Memory Model* (*JMM*) описывает поведение потоков в среде исполнения Java. Модель памяти — часть семантики языка Java, и описывает, на что может и на что не должен рассчитывать программист, разрабатывающий ПО не для конкретной Java-машины, а для Java в целом. JMM - абстракция, которая гарантирует, что один и тот же код будет выполнен одинаково на всех платформах.  Память виртуальной машины разделена на несколько частей.  https://lh6.googleusercontent.com/yAmXW9b5lqslNqK4GdowgHBJOlruzIRTikQAwf6qh6oSP7RBCYuYg5fyWWg5g9-2J4W2Saoq05xEWbiG5K6Mpr7Ebrmu8A9WfW-ywf722Zc2J8T1_VblhvCpuEaSADh_ScNReAXu  Все методы хранятся в стеке и попадают туда при вызове. Переменные в методах также имеют стековую память. Если в методе создается объект, то он помещается в кучу (heap), но его ссылка помещается в стек.  **Heap Area**  Используется всеми частями приложения.  Всякий раз при создании объекта он хранится в куче, а ссылка на него в стеке.  Объекты в куче доступны из любой точки программы.  Память в куче живет с самого начала и до конца работы программы.  Можно определить начальный и максимальный размер памяти в куче: -Xms и –Xmx (опции JVM)  Если память кучи заполнена: java.lang.OutOfMemoryError: Java Heap Space  Для кучи выделено больше памяти, чем для стека.  **Stack Area**  Используется только одним потоком исполнения программы.  Хранит только локальные переменные примитивных типов и ссылки на объекты в куче.  Стековая память не может быть доступна для других потоков.  Управление памятью: по схеме LIFO  Стековая память существует лишь какое-то время работы программы, во время которого выполняется метод.  Размер памяти: -Xss  Если память стека заполнена: java.lang.StackOverflowError  Из-за простоты распределения память (LIFO), стековая память работает намного быстрее кучи.  **Code cache** - используется для хранения нативного кода, исполняемого JVM и кода, скомпилированного с помощью JIT компилятора.  ***Stack memory***  Java Stack Memory используется для выполнения потоков. Они содержат данные конкретного метода, которые обычно коротко живущие и ссылки на другие объекты, лежащие в Heap.  Когда запускается новый поток, JVM создает новый Java Stack для потока. В нем в отдельных фреймах хранится состояние потока. JVM выполняет только два действия над стеком: push и pop frames.  Current method – метод, который в данный момент выполняется потоком.  Current frame – фрейм стека для текущего метода.  Current class – класс, в котором определен текущий метод.  Current class constant pool – пул констант текущего класса.  Во время выполнения текущего метода, JVM отслеживает текущий класс и текущий пул констант. Когда JVM сталкивается с инструкциями над данными, хранимыми в стековом фрейме, она выполняет данные операции в текущем фрейме.  Когда поток запускает Java метод, виртуальная машина создает и запихивает новый фрейм в потоковый стек. Этот новый фрейм потом становится текущим фреймом. Во время выполнения метода, он использует фрейм для хранения параметров, локальных переменных, промежуточных вычислений и др.  Метод может завершиться одним из следующих способов:   * Normal completion – если метод завершен с помощью return. * Abrupt completion – если метод завершен из-за возникшего исключения.   В обоих вариантах завершения метода JVM pops and discards (сбрасывает) стек фрейм метода. После этого фрейм для предыдущего метода становится текущим фреймом.  **Данные**  Все данные, хранимые в потоковом стеке, private для этого потока. Стек не имеет доступа и не может изменять стек другого потока. Благодаря этому, нет проблем с синхронизацией многопоточного доступа к локальным переменным. Когда поток запускает метод, локальные переменные метода размещаются в фрейме стека вызывающего потока. Только поток, вызвавший метод, может иметь доступ к локальным переменным.  Как и Method Area и Heap, Java Stack и Stack Frames не обязательно смежные в памяти.  **STACK FRAME** Stack Frame разделен на три части:  **1.** **Local Variables** Данная область организована в виде нулевого массива слов. Инструкции, использующие значения из этой области имеют индекс в данном массиве. Значения int, float, reference и returnAddress занимают одну запись в данном массиве. Значения типов byte, short и char конвертируются в int перед размещением. Значения типов long и double занимают две последовательные ячейки (entry) в массиве. Дня использования значений типа long или double инструкции получают значение индекса первой ячейки. Размер области локальных переменных зависит от нужд конкретного метода. Данный размер определяется во время компиляции и включается в данные класс файла для каждого метода. Компилятор размещает параметры в область локальных переменных и параметров методов в порядке их декларирования.  **2.Operand Stack** Является рабочим пространством на (runtime workspace) для выполнения промежуточных действий. Как и область для локальных переменных, стек операндов организован в виде нулевого массива. Но доступ к данным осуществляется не по индексам, а через push и pop. Если инструкция push значение в стек операндов, то следующая инструкция может pop и использовать значение. В стеке операндов хранятся те же типы, что и в области локальных параметров: int, long, float, double, reference и returnType. Byte, short и char конвертируются в int перед тем, как push их в стек операндов. JVM использует стек операндов в качестве рабочего места. Многие инструкции pop значения из стека операндов, производят действия над ними и push результат. Например, сложение двух integer: Первые две инструкции push int значения переменных в нулевую и первую позиции стека операндов. Далее add инструкция pop эти значения, складывает и push результат обратно в стек операндов. Четвертая инструкция pop результат, добавленный сверху в стеке операндов, и размещает его в область локальных переменных.  **3.Frame Data** В данной области хранятся данные для поддержки пула констант, нормального возвращаемого значения и отправки исключений. Многие инструкции в наборе инструкций JVM ссылаются на записи в пуле констант. Некоторые инструкции просто push константные значения типов int, long, float, double, String из пула констант в стек операндов. Некоторые инструкции используют записи из пула констант для того, чтобы ссылаться на классы или массивы для instantiate, поля для доступа или методы для вызова. Другие инструкции определяют, является ли конкретный объект потомком определенного класса или интерфейса, посредством записи в пути констант. Когда JVM сталкивается с инструкцией, которая ссылается на запись в пуле констант, она использует указатель данных фрейма на пул констант для доступа к данной информации. Ссылки на типы, поля и методы в пуле констант на данном этапе символические. Когда JVM проверяет записи в пуле констант, которые ссылаются на класс, интерфейс, поле или метод, данная ссылка все еще может быть символической. В таком случае, JVM должна разрешить ссылки в данный момент. Помимо разрешения пула констант frame data должны служить вспомогательным механизмом для JVM для обработки normal и abrupt завершения метода. Если метод завершен нормально (с с помощью return), JVM должна восстановить/возобновить фрейм стека вызванного метода. JVM должна установить pc регистр, чтобы тот указывал на инструкцию в вызванном методе, которая следует за инструкцией, вызванной для завершения метода. Если завершившийся метод вернул значение, JVM должна push это значение в стек операндов вызванного метода. Frame data должен содержать ссылку на таблицу исключений метода, которую JVM использует для обработки любого исключения, выброшенного во время выполнения метода. Когда метод выбрасывает исключение, JVM использует таблицу исключений, на которую ссылается frame data, для определения как обрабатывать исключение. Если JVM находит соответствующее условие catch в таблице исключений метода, она передает управление в начало этого условия catch. В противном случае, метод завершается abruptly. JVM использует информацию из frame data для восстановления фрейма вызванного метода. Он далее перевыбрасывает то же исключение в контекст вызванного метода. http://www.artima.com/insidejvm/ed2/jvm8.html  ***Heap memory***  Heap Area (динамически распределяемая область памяти, создаваемая при старте JVM) – место для хранения всех Objects и соответствующих им экземпляров переменных и массивов. Существует по одному Heap Area для каждой JVM. Именно по причине того, что данные области для хранения являются единичными в рамках одной JVM, память является разделяемым ресурсом, из-за чего в многопоточных системах могут возникнуть проблемы. Именно здесь работает сборщик мусора GC. Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться с любой части приложения.  В Heap выделяется место под сам объект. Количество выделенной памяти зависит от полей. Например, если 2 поля int (каждый по 32 бит), то в сумме под объект выделяется 64 бит.  JVM Heap memory физически разделена на две части: Young Generation и Old Generation.  https://lh6.googleusercontent.com/J7gLRPJ92WtloSoI382gWWVcTcRAMJN-tXUGc3C8oRto64a06KPq783DxKlcRZqWxMGyO2gwS2csf3QkB7n_ZEI1h6uvFyQlpD_JtaV4ZFulBo7MVOCq_Fe1LMRKzUKQlFZ0Ifhe  https://lh3.googleusercontent.com/Dnv0Yx9EW5EwbRR6NmBTDvjM0V2hC7a1PPLAEkOCm2Vn05t_YuHqd7-cZHyHJ6pMvI5c8aTgfrjikiVMSze6nf2-3DqcVBLXbw9fYOy3VKGOTtXX1z_rIyTjxw7uIdXbLQrQ9bAq  **Young generation** – место, куда попадают после создания все новые объекты. После того, как данная область памяти заполняется, garbage collector производит очистку памяти. Данная очистка носит название: Minor GC. <https://plumbr.eu/handbook/garbage-collection-in-java#minor-gc>  Young generation разделена на три части: Eden Memory и две Survivor Memory (S0 И S1). Ключевые особенности Young generation memory:  Большинство ново созданных объектов располагаются в Eden Memory.  После того, как Eden полностью заполняется объектами, производится Minor GC и все выжившие объекты перемещаются в одну из областей Survivor Memory.  Minor GC также проверяет выжившие объекты, находящиеся в Survivor Memory, и переносит их во вторую Survivor область. Таким образом, одна из Survivor областей всегда пустая.  Объекты, которые пережили много циклов GC, переносятся в Old Generation Memory. Обычно это производится путем установления порогового значения возраста для объектов Young Generation перед тем, как они будут выбраны для перемещения в Old Generation Memory.  **Old Generation** (Tenured) – содержит долго живущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и прочие), которые пережили несколько циклов Minor GC. Обычно сборка мусора в Old Generation Memory выполняется после его заполнения. Данная сборка мусора называется Major GC и занимает больше времени по сравнению с Minor GC.  Все GC работают в режиме “**Stop the World**”, потому что все потоки приложения останавливаются до окончания данной операции.  Ввиду того, что Young Generation содержит коротко живущие объекты, Minor GC производится очень быстро и не успевает повлиять на приложение.  Тем не менее Major GC требует намного больше времени, потому что во время Major GC проверяются все живые объекты. Работа Major GC должна быть минимизирована, потому что она может сделать приложение не ответственны за продолжительность сборки мусора. Таким образом, если необходимо построить приложение, которое должно быть высокопроизводительным, но в нем часто происходит Major GC, будет заметно проседать время.  Длительность работы сборщика мусора зависит от выбранной им стратегии. Именно поэтому важно следить и корректировать работу GC.  **JAVA HEAP MEMORY SWITCHES**  Java предоставляет множество различных параметров для установки размеров памяти и ее показателей.  VM SWITCH                       VM SWITCH DESCRIPTION  -Xms                         For setting the initial heap size when JVM starts  -Xmx                         For setting the maximum heap size.  -Xmn   For setting the size of the Young Generation, rest of the space goes for Old Generation. -XX:PermGen    For setting the initial size of the Permanent Generation memory -XX:MaxPermGen     For setting the maximum size of Perm Gen  -XX:NewRatio            For providing ratio of old/new generation sizes. The default value is 2.  -XX:SurvivorRatio   For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8.  ***What is String pool***  Memory Pools были созданы разработчиками JVM Memory для того, чтобы создать пул неизменяемых объектов, если реализация поддерживала бы их. String Pool – один из примеров Memory Pool.  Memory Pool в зависимости от JVM принадлежит Heap или Perm Gen.  String pool - коллекция ссылок на строковые объекты. Строки, являясь частью пула литералов, размещены в куче, но ссылки на них находятся в пуле литералов. Ссылка из пула литералов будет возвращена только при создании строки с помощью сокращенного синтаксиса (без использования new). Если явно указать оператор new, то это прямой приказ JVM создать новый объект и пул литералов при этом просмотрен не будет. На уже созданном строковом объекте вызов метода intern() вернет ссылку на объект из пула (если объекта нет в пуле - то занесет его в пул).  **До java 7** все строковые объекты в приложении хранились **в PermGen**, что часто вызывало ошибку OutOfMemoryError из-за переполнения этой области памяти.  **В java 7** все строковые объекты хранятся **в heap**, вместе со всеми остальными объектами. Перемещение строк из PermGen в heap позволило увеличить производительность и сократить появление исключения из-за переполнения PermGen.  ***What is PermGen, Metaspace***  **Perm Gen** содержит метаданные приложения, необходимые JVM для описания классов и методов, используемых в приложении (объекты класса Class). Perm Gen – не часть Java Heap Memory.  JVM населяет Perm Gen классами, которые используются в приложении, во время выполнения. Perm Gen также содержит Java SE библиотеки классов и методов. Объекты из Perm Gen забираются сборщиком мусора во время полной очистки full GC.  **Method Area** – **часть** пространства **Perm Gen**, используемая для хранения структуры класса (констант времени выполнения и статических переменных) и кода для методов и конструкторов. К нему имеют доступ все созданные в приложении потоки. Он создается при старте виртуальной машины и заполняется класслоадерами из байткода. Данные в method area остаются в памяти до тех пор, пока жив класслоадер, которые их туда загрузил.  В method area хранится:  Информация о классах (количество полей/методов, имя суперкласса, имена интерфейсов и так далее)  Байткод методов и конструкторов  Пул констант для каждого загруженного класса  В Java 7 Method Area хранилась в PermGen. С Java 8 method Area теперь хранится в отдельном участке нативной памяти, который называется Metaspace, который по умолчанию ограничен только объемом памяти на машине.  Note: регистр PC указывает на текущую выполняемую инструкцию (хранит адрес текущей инструкции в Method Area). Если метод, который выполняется в текущий момент определенным потоком является нативным методом, то значение регистра PC JVM является неопределенным.  Runtime Constant Pool (часть Method Area) – представление Constant Pool в классе per-class. Содержит константы класса времени выполнения и статические методы. Этот пул похож на таблицу символов для стандартного языка программирования. Другими словами, когда ссылаются на класс, метод или поле JVM ищет реальный адрес в памяти в runtime constant pool (пул констант для каждого класса).  https://lh3.googleusercontent.com/cEOP0zyPlvPVRWKU6DhcvWpL_p8FJrjIzMEs3sG84D9DE4ubS2HiB4KQdvokO8uDRNiR606eswTpCyML7lxTIkapyeWak76CtNjq33f7Zgpkh1OUuB9g_uVTCCT3ygK7cZ8qFKPT  При загрузке класса ClassLoader создает новый объект класса в PermGen.  При переполнении этой области памяти возникает исключение OutOfMemoryError. Из-за того, что данный участок памяти не мог расширяться динамически и часто переполнялся в Java 8 полностью отказались от использования PermGen в пользу Metaspace.  Metaspace:  Metaspace это замена PermGen, основное отличие которой с точки зрения Java-программистов — возможность динамически расширяться, ограниченная по умолчанию только размером нативной памяти. Параметры PermSize и MaxPermSize отныне упразднены (получив эти параметры JVM будет выдавать предупреждение о том, что они более не действуют), и вместо них вводится опциональный параметр MaxMetaspaceSize, посредством которого можно задать ограничение на размер Metaspace.  В результате максимальный Metaspace по умолчанию не ограничен ничем кроме предела объема нативной памяти. Но его можно по желанию ограничить параметром MaxMetaspaceSize, аналогичным по сути к MaxPermSize.  Предполагается, что таким образом можно будет избежать ошибки «java.lang.OutOfMemoryError: PermGen space» за счет большей гибкости динамического изменения размера Metaspace. Но, конечно, если размер Metaspace достигнет своей границы — будь то максимум объёма нативной памяти, или лимит заданный в MaxMetaspaceSize — будет выброшено аналогичное исключение: «java.lang.OutOfMemoryError: Metadata space».  Сборка мусора в Metaspace  Логи Garbage Collector-а будут сообщать также и о сборке мусора в Metaspace.  Сама сборка мусора будет происходить при достижении Metaspace размера, заданного в MaxMetaspaceSize. Когда MaxMetaspaceSize не задан, сборка мусора в Metaspace тоже осуществляется перед каждым его динамическим увеличением.  ***Garbage Collector - basic principles of work***  Garbage Collector должен делать всего две вещи:  - Обнаруживать мусор  - Очищать память от мусора  Как Garbage Collector обнаруживает мусор? Существует два подхода к обнаружению мусора:  - Reference counting  - Tracing  Reference counting Суть подхода состоит в том, что каждый объект имеет счетчик. Счетчик хранит информацию о том, сколько ссылок указывает на объект. Kогда ссылка уничтожается, счетчик уменьшается. Если значение счетчика равно нулю, - объект можно считать мусором и память можно очищать. Главным минусом такого подхода является сложность обеспечения точности счетчика. Также при таком подходе сложно выявлять циклические зависимости (когда два объекта указывают друг на друга, но ни один живой объект на них не ссылается). Это приводит к утечкам памяти. В общем, Reference counting редко используется из за недостатков. Во всяком случае HotSpot VM его не использует. По этому мы можем отложить в памяти, что такой подход есть, и продолжить дальше.  Tracing В "Tracing" главная идея состоит в мысли: "Живые объект - те до которых мы можем добраться с корневых точек (GC Root), все остальные - мусор. Все что доступно с живого объекта - также живое". Если мы представим все объекты и ссылки между ними как дерево то нам нужно пройти с корневых узлов по всем узлам. При этом узлы, до которых мы сможем добраться - не мусор, все остальные - мусор. При таком подходе легко выявить циклические зависимости, - все объекты к которым невозможно добраться с корневых точек будут считаться мусором.  HotSpot VM использует именно такой подход.  Что такое корневая точка (GC Root)? Литература говорит что существует 4 типа корневых точек:   * Локальные переменные и параметры методов * Java Потоки * Статические переменные * Ссылки из JNI   Самое простое java приложение будет иметь такие корневые точки:   * Локальные переменные внутри main метода, параметры main метода. * Поток который выполняет main. * Статические переменные класса, внутри которого находится main метод.   Как GC очищает память от мусора? 1. Copying collectors Память делится на две части "from-space" "to-space". Принцип работы такой:  Объекты аллоцируются в "from-space"  "from-space" заполняется, нужно собрать мусор  Приложение приостанавливается  Запускается сборщик мусора. Находятся живые объекты в "from-space" и копируются в "to-space"  Когда все объекты скопированы "from-space" полностью очищается  "to-space" и "from-space" меняются местами Главный плюс такого подхода в том, что объекты плотно забивают память. Минусы подхода:  Приложение должно остановится пока не пройдет полный цикл сборки мусора  В худшем случае form-space и to-space должны быть одинакового размера. Это случай, когда все объекты живые. В итоге, плюс в том, что память используется эффективно. Но при этом приложение должно прекращать свою работу на время сборки мусора. Также очень не эффективно используется память, так как в худшем случае "from-space" должен быть равен "to-space". В чистом виде такой алгоритм в HotSpot VM не используется. 2. Mark-and-sweep Алгоритм можно описать так:  Объекты аллоцируются в памяти  Нужно запустить GC  Приложение приостанавливается  Сборщик проходится по дереву объектов, помечая живые объекты  Сборщик проходится по всей памяти, находя все не отмеченные куски памяти, сохраняя их в "free list"  Когда новые объекты начинают аллоцироваться они аллоцируются в память доступную в "free list" Минусы:  Приложение не работает пока происходит сборка мусора  Время работы зависит от размеров памяти и количества объектов  Если не использовать "compacting" память будет использоваться не эффективно В чистом виде такой подход для сборки мусора в Hotspot VM тоже не используется. Какой подход используется в HotSpot VM? **Сборщики мусора HotSpot VM используют подход "Generational Garbage Collection".** Как мы увидим, этот подход позволяет использовать разные алгоритмы для разных этапов сборки мусора. Это позволяет использовать наиболее подходящий алгоритм. Было замечено, что большинство приложений удовлетворяют двум правилам (weak generational hypothesis):  Большинство аллоцированых (только что созданных) объектов быстро становятся мусором. (гипотеза о поколениях)  Существует мало связей между объектами, которые были созданы в прошлом и только что аллоцироваными объектами. Именно на эти правила опирается подход "Generational Garbage Collection". Подавляющее большинство объектов создаются на очень короткое время, они становятся ненужными практически сразу после их первого использования. Итераторы, локальные переменные методов, результаты боксинга и прочие временные объекты, которые зачастую создаются неявно, попадают именно в эту категорию. Вот тут и возникает идея разделения объектов на младшее поколение (young generation) и старшее поколение (old generation). В соответствии с этим разделением и процессы сборки мусора разделяются на малую сборку (minor GC), затрагивающую только младшее поколение, и полную сборку (full GC), которая может затрагивать оба поколения. Малые сборки выполняются достаточно часто и удаляют основную часть мертвых объектов. Полные сборки выполняются тогда, когда текущий объем выделенной программе памяти близок к исчерпанию и малой сборкой уже не обойтись. При этом разделение объектов по поколениям не просто условное, они физически размещаются в разных регионах памяти. Объекты из младшего поколения по мере выживания в сборках мусора переходят в старшее поколение. В старшем поколении объект может прожить до окончания работы приложения, либо будет удален в процессе одной из полных сборок мусора. Существуют факторы, которые могут задержать его в мире живых чуть дольше, чем нам того хотелось бы. Все мы знаем, что считать объект живым просто по факту наличия на него ссылок из других объектов нельзя. В противном случае рецепт бессмертия в JVM был бы до безобразия прост и заключался бы в наличии взаимных ссылок хотя бы у двух объектов друг на друга, а в общем случае — в наличии цикла в графе связанности объектов. При таком подходе и ограниченном объеме памяти более-менее серьезная программа долго не проработала бы, поэтому с отслеживанием циклов в графах объектов JVM справляется хорошо. Рассмотрим такую ситуацию: У нас есть молодой объект A и ссылающийся на него объект B, уже заслуживший место в старшем поколении. В какой-то момент времени оба этих объекта стали нам не нужны и мы обнулили все имеющиеся у нас ссылки на них. Очевидно, объект A можно было бы удалить в ближайшую малую сборку мусора, но для того, чтобы получить это знание, сборщику пришлось бы просмотреть всё старшее поколение и понять, что объект B ссылающийся на A, тоже является мусором, а следовательно их оба можно утилизировать. Но анализ старшего поколения не входит в план малой сборки, так как является относительно дорогой процедурой, поэтому объект А во время малой сборки будет считаться живым. Таким образом, чаще всего для целей малой сборки мусора объект считается мертвым и подлежащим утилизации, если до него невозможно добраться по ссылкам ни из объектов старшего поколения, ни из так называемых корней (roots), к каковым относятся ссылки из стеков потоков, статические члены классов и т. п. При полной сборке мусора могут анализироваться оба поколения, поэтому здесь сборщик может плясать только от корней.  **Общий подход к очистке мусора** 1. Все новосозданные объекты размещаются в Eden. Обе области Survivor пусты (в самом начале). 2. Когда Eden полностью заполняется, начинает работу minor gc. 3. Живые объекты перемещаются в первую Survivor 0 область. Мертвые (без ссылок) – удаляются во время очистки Eden. 4. Во время следующей minor gc в Eden происходит то же самое. Мертвые объекты удаляются, а живые перемещаются в Survivor 1 (вторую область). Вдобавок, объекты, размещенные в S0 во время предыдущей minor gc инкрементируют свой возраст и переносятся в S1. После этого Eden и S0 пусты и еще теперь существуют объекты разного возраста. 5. Во время следующей minor gc все повторяется (только поменяны местами S0 и S1). 6. «Повышение» - после того, как возраст объектов достиг определенного порога, они перемещаются из New Generation в Old Generation. 7. Так как minor gc продолжает размещать объекты, объекты продолжают проходить на «повышение». 8. В конечном счете запустится major gc для Old Generation, которая очистит и скомпанует место. В **HotSpot VM** реализовано **четыре** **сборщика мусора** основанных **на идее "Generational Garbage Collection":**   * **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. * **Garbage-First (G1)** — создан для постепенной замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных.   **SERIAL GC** Serial GC (он же последовательный сборщик) — младший с точки зрения заложенной в него функциональности, но старший с точки зрения продолжительности присутствия в JVM сборщик мусора. Он медленно, но верно собирал мусор еще тогда, когда многие из нас даже не подозревали о существовании языка Java. И до сих пор продолжает собирать. Так же медленно, но так же верно. А не отправился на задворки истории он потому, что не у всех программ большие кучи и не все программы работают на компьютерах с мощными многоядерными процессорами. В таких спартанских условиях он оказывается весьма кстати. Использование Serial GC включается опцией -XX:+UseSerialGC. Принципы работы При использовании данного сборщика куча разбивается на четыре региона, три из которых относятся к младшему поколению (Eden, Survivor 0 и Survivor 1), а один (Tenured) — к старшему: Среднестатистический объект начинает свою жизнь в регионе Eden (переводится как Эдем, что вполне логично). Именно сюда его помещает JVM в момент создания. Но со временем может оказаться так, что места для вновь создаваемого объекта в Eden нет, в таких случаях запускается малая сборка мусора.  minor GC Для того, чтобы "minor GC" проходил быстро, нужно что бы при нем не приходилось сканировать "old generation". Возникает вопрос: "Как выявить ссылки на объекты c "old generation" на объекты в "young generation" не сканируя "old generation"" Как мы помним, соответствуя "weak generational hypothesis" их должно быть мало, но они могут быть. Для решения этой проблемы HotSpot VM содержит структуру "card table". Память в "old generation" разбивается на карты (cards). Card table - это массив с однобайтной ячейкой, каждая ячейка массива соответствует куску памяти (карте) в "old generation". Когда в каком-то поле объекта обновляется ссылка, то в "card table" нужная карта помечается как "грязная" (для этого нужна однобайтная ячейка). В итоге при "minor GC" для выявления ссылок "old-to-new" сканируется не весь "old-generation", а только объекты которые находятся в "грязных" картах. "Young generation" делится на:  Eden. Кусок памяти, где объекты аллоцируются. После сборки мусора "Eden" - пустой, мусор должен удалиться, а выжившие объекты попасть в "Survivor space"  Survivor space 1,2. То, что в разделе "Copying collectors" называлось "from-space" и "to-space". Тут находятся объекты, которые выжили при предыдущей сборке мусора, но перед отправкой в "old generation" им дан шанс стать мусором во время следующей сборки. Survivor space 1 будем называть "from space", Survivor space 2 - "to space". Алгоритм работы очень похож на "Copying collectors", отличие в том, что появился "Eden":  Начало сборки мусора, приложение приостанавливается.  Живые объекты из "Eden" копируются в "to space".  Живые объекты из "from space" копируются в "to space" или в "old generation", если они достаточно старые.  "Eden" и "from space" очищаются, так как в них остался только мусор.  "to space" и "from space" меняются местами  Приложение продолжает работу. Сборка мусора:  Коричневый - мусор. Зеленый - живой объект.  https://lh4.googleusercontent.com/YNnA8hqfSn_7Ix9EMvI8eAiXA2WCuN8o4hrLQ2bhFK1WYqa3qLbUlU_UwGjSPPR_SCdC236d37nZy7QGk8elAeRNEjTH1YA_2fVqf0D7XbPVdaKNy1immH4JlyTS3L_THja1nOIg  После сборки мусора: После "minor gc" "Eden" и "to space" пустые, в "from space" лежат объекты пережившие сборку, немного долгоживущих объектов перекочевало в "old generation". И того: Первым делом такая сборка находит и удаляет мертвые объекты из Eden. Оставшиеся живые объекты переносятся в пустой регион Survivor. Один из двух регионов Survivor всегда пустой, именно он выбирается для переноса объектов из Eden: После малой сборки регион Eden пуст и может быть использован для размещения новых объектов. Но рано или поздно наше приложение опять займет всю область Eden и JVM снова попытается провести малую сборку, на этот раз очищая Eden и частично занятый Survivor 0, после чего перенося все выжившие объекты в пустой регион Survivor 1: В следующий раз в качестве региона назначения опять будет выбран Survivor 0. Пока места в регионах Survivor достаточно, все идет хорошо: JVM постоянно следит за тем, как долго объекты перемещаются между Survivor 0 и Survivor 1, и выбирает подходящий порог для количества таких перемещений, после которого объекты перемещаются в Tenured, то есть переходят в старшее поколение. Если регион Survivor оказывается заполненным, то объекты из него также отправляются в Tenured: Описанный процесс малой сборки мусора достаточно прост, но причины использования регионов Survivor, причем именно двух, не всегда понятны. После удаления мусора регион оказывается сильно фрагментированным и если вы хотите это исправить, то у вас есть два варианта: либо уплотнять объекты в рамках этого же региона, либо скопировать их в другой, пока еще пустой регион, располагая один-к-одному, а старый регион объявить пустым. Но задача осложняется тем, что объекты ссылаются друг на друга и при перемещении любого объекта необходимо производить обновление всех имеющихся на него ссылок. И вот эту задачу намного легче решать при копировании, причем сразу объединяя ее с задачей поиска живых объектов. Из двух основных способов работы с выжившими объектами — уплотнение и копирование — в Sun при разработке малого сборщика мусора пошли по второму пути, так как он проще в реализации и зачастую оказывается производительнее. В случае, когда места для новых объектов не хватает уже в Tenured, в дело вступает полная сборка мусора, работающая с объектами из обоих поколений. При этом старшее поколение не делится на подрегионы по аналогии с младшим, а представляет собой один большой кусок памяти, поэтому после удаления мертвых объектов из Tenured производится не перенос данных (переносить уже некуда), а их уплотнение, то есть размещение последовательно, без фрагментации. Такой механизм очистки называется Mark-Sweep-Compact по названию его шагов (пометить выжившие объекты, очистить память от мертвых объектов, уплотнить выжившие объекты). Акселераты В разделе Eden создается среднестатистический объект, а не любой. Такая оговорка сделана неспроста. Дело в том, что бывают еще объекты-акселераты, размер которых настолько велик, что создавать их в Eden, а потом таскать за собой по Survivor’ам слишком накладно. В этом случае они размещаются сразу в Tenured. Размер кучи и ее регионов Важными факторами в описанных процессах являются абсолютный размер кучи и относительные размеры регионов внутри нее. По мере заполнения кучи данными JVM может не только проводить чистку памяти, но и запрашивать у ОС выделение дополнительной памяти для расширения регионов. Причем в случае, если реально используемый объем памяти падает ниже определенного порога, JVM может вернуть часть памяти операционной системе. Для регулирования аппетита виртуальной машины существуют известные всем опции Xms и Xmx. Тут также стоит отметить, что по умолчанию младшее поколение занимает одну треть всей кучи, а старшее, соответственно, две трети. При этом каждый регион Survivor занимает одну десятую младшего поколения, то есть Eden занимает восемь десятых. В итоге реальные пропорции регионов по умолчанию выглядят так:  https://lh4.googleusercontent.com/NsDkp2ULRfu0BiYPoHiStdZ0ygVtho_1c6FURfCj6SwuMHPPBPqg61GKrA02Y99P-whd33_IcgMDfffB6Uxkx_ioBxd1HXSMNy4dxaiRzMwrx-EoBU1_1avQUIwmEZktkcCzB2s0  А что же происходит, если даже после выделения максимального объема памяти и ее полной чистки, места для новых объектов так и не находится? В этом случае мы ожидаемо получаем java.lang.OutOfMemoryError: Java heap space и приложение прекращает работу, оставляя нам на память свою кучу в виде файла для анализа. Технически, это происходит в случае, если работа сборщика начинает занимать не менее 98% времени и при этом сборки мусора освобождают не более 2% памяти. Ситуации STW С этим сборщиком все достаточно просто, так как вся его работа — это один сплошной STW. В начале каждой сборки мусора работа основных потоков приложения останавливается и возобновляется только после окончания сборки. Причем всю работу по очистке Serial GC выполняет не торопясь, в одном потоке, последовательно, за что и удостоился своего имени. Настройка С помощью опций Xms и Xmx можно настроить начальный и максимально допустимый размер кучи соответственно.  -XX:MinHeapFreeRatio=? и -XX:MaxHeapFreeRatio=?  Задают минимальную и максимальную долю свободного места в каждом поколении, при достижении которой размер поколения будет автоматически увеличен или уменьшен соответственно.  Например, если MinHeapFreeRatio=35, то при падении доли свободного места в каком-либо поколении ниже 35%, этому поколению будет предоставлено дополнительное место, чтобы не менее 35% стало свободным. Аналогично, если MaxHeapFreeRatio=65, то при увеличении доли свободного места в поколении до 65% и более, часть выделенной этому поколению памяти будет освобождена для возвращения к желаемому порогу. Значения данных параметров по умолчанию зависят от аппаратных характеристик компьютера.  -XX:NewRatio=?  Установка отношение размера старшего поколения к суммарному размеру регионов младшего поколения.  Например, NewRatio=3 означает, что для младшего поколения (Eden + S0 + S1) будет отведена четверть кучи, а для старшего — три четверти.  -XX:NewSize=? и -XX:MaxNewSize=?  Можно ограничить размер младшего поколения абсолютными величинами снизу и сверху.  Если надо установить для NewSize и MaxNewSize одинаковые значения, то можно просто использовать опцию -Xmn. Например, -Xmn256m эквивалентно -XX:NewSize=256m -XX:MaxNewSize=256m.  -XX:SurvivorRatio=?  Можно настроить отношение размера Eden к размерам Survivor.  Например, при SurvivorRatio=6 каждый регион Survivor будет занимать одну восьмую размера всего младшего поколения, а Eden — шесть восьмых (помним про правило опций \*Ratio).  -XX:-UseGCOverheadLimit  Можно отключить порог активности сборщика в 98%, при достижении которого возникает OutOfMemoryError.  -XX:+PrintTenuringDistribution  Для того, чтобы последить за тем, как стареют объекты в регионе Survivor и какие целевые значения для его размера установлены в данный момент. Данная опция добавляет статистику по Survivor к выводу информации о некоторых сборках мусора.  Достоинство  Непритязательность по части ресурсов компьютера. Так как всю работу он выполняет последовательно в одном потоке, никаких заметных оверхедов и негативных побочных эффектов у него нет.  Недостатки  1. Долгие паузы на сборку мусора при заметных объемах данных.  2. Все настройки Serial GC крутятся вокруг размеров различных регионов кучи. То есть для тонкой настройки требуется самому что-то изучать, настраивать, экспериментировать и прочее.  Если вашему приложению не требуется большой размер кучи для работы (Oracle указывает условную границу 100 МБ), оно не очень чувствительно к коротким остановкам и ему для работы доступно только одно ядро процессора, то можно приглядеться к этому варианту. В противном случае можно поискать вариант по-лучше.  **PARALLEL GC** Parallel GC (параллельный сборщик) развивает идеи, заложенные последовательным сборщиком, добавляя в них параллелизм и немного интеллекта. Если на вашем компьютере больше одного процессорного ядра и вы явно не указали, какой сборщик хотите использовать в своей программе, то почти наверняка JVM остановит свой выбор на Parallel GC. Он достаточно простой, но в то же время достаточно функциональный, чтобы удовлетворить потребности большинства приложений. Параллельный сборщик включается опцией -XX:+UseParallelGC. Принципы работы При подключении параллельного сборщика используются те же самые подходы к организации кучи, что и в случае с Serial GC — она делится на такие же регионы Eden, Survivor 0, Survivor 1 и Old Gen (знакомый нам под именем Tenured), функционирующие по тому же принципу. Но есть два принципиальных отличия в работе с этими регионами: во-первых, сборкой мусора занимаются несколько потоков параллельно; во-вторых, данный сборщик может самостоятельно подстраиваться под требуемые параметры производительности. Для определения количества потоков, которые будут использоваться при сборке мусора, на компьютере с N ядрами процессора, JVM по умолчанию применяет следующую формулу: если N ≤ 8, то количество потоков равно N, иначе для получения количества потоков N домножается на коэффициент, зависящий от других параметров, обычно это 5/8, но на некоторых платформах коэффициент может быть меньше. По умолчанию и малая и полная сборка задействуют многопоточность. Малая пользуется ею при переносе объектов в старшее поколение, а полная — при уплотнении данных в старшем поколении. Каждый поток сборщика получает свой участок памяти в регионе Old Gen, так называемый буфер повышения (promotion buffer), куда только он может переносить данные, чтобы не мешать другим потокам. Такой подход ускоряет сборку мусора, но имеет и небольшое негативное последствие в виде возможной фрагментации памяти:  https://lh4.googleusercontent.com/mPTOz2sQY3RqXlGYhYca-qz_LGw6gIMW_r0yoecGA4M6VRdtWkL0-zd72oRDBiIEMGdCXt-s5pvEkv1pxsc5mBsG6crNmN9eObBBa2hZ_7GhpRuYCpr1_S7DZnmLruQosq3RuYK5  Интеллектуальная составляющая улучшений параллельного сборщика относительно последовательного заключается в том, что у него есть настройки, ориентированные на достижение необходимой вам эффективности сборки мусора. Вы можете указать устраивающие вас параметры производительности — максимальное время сборки и/или пропускную способность — и сборщик будет изо всех сил стараться не превышать заданные пороги. Для этого он будет использовать статистику уже прошедших сборок мусора и исходя из нее планировать параметры дальнейших сборок: варьировать размеры поколений, менять пропорции регионов. Например, если при малой сборке JVM не удается укладываться в отведенное вами время, размер младшего поколения может быть уменьшен. Если не удается достигнуть заданной пропускной способности, а с задержкой проблем нет, то размер поколения будет увеличен. И так далее.  При этом следует иметь в виду, что в статистике игнорируются сборки мусора, запущенные вручную. Конечно, стопроцентной гарантии достижения желаемых параметров никто не даст, но часто установки нужных опций оказывается достаточно. В случае, если задали слишком жесткие требования, которые сборщик не может выполнить, он будет ориентироваться на следующие приоритеты (в порядке убывания важности): 1. Снижение максимальной паузы. 2. Повышение пропускной способности. 3. Минимизация используемой памяти. При этом Parallel GC оставляет нам возможность самостоятельно корректировать размеры регионов, как и в последовательном сборщике. Но не рекомендуется делать и то и другое одновременно, чтобы не дезориентировать алгоритмы автоматической подстройки. Либо мы выделяем приложению достаточно памяти, указываем желаемые параметры производительности и наблюдаем со стороны, либо сами залезаем в настройки регионов, но тогда лишаемся права требовать от сборщика автоматической подстройки под нужные нам критерии производительности. Так как в противном случае, эффективно выполнять свою работу он не сможет. Ситуации STW Как и в случае с последовательным сборщиком, на время операций по очистке памяти все основные потоки приложения останавливаются. Разница только в том, что пауза, как правило, короче за счет выполнения части работ в параллельном режиме. Настройка Для параллельного сборщика применимы все те же опции, что и для последовательного. Вы можете вручную устанавливать размеры регионов памяти или пропорции между ними. Ниже перечислены те опции, которые добавляются параллельным сборщиком к тому, что мы уже рассматривали выше.  -XX:ParallelGCThreads=? Вручную указать количество потоков, которое необходимо выделить для сборки мусора. Имейте в виду, что увеличение количества потоков не только сильнее распараллеливает сборку, но и увеличивает фрагментацию региона Tenured, а также добавляет накладные расходы на синхронизацию этих потоков. Например, -XX:ParallelGCThreads=9 ограничит количество потоков девятью.  -XX:-UseParallelOldGC Полностью отключить параллельные работы по уплотнению объектов в старшем поколении.  XX:MaxGCPauseMillis=? и -XX:GCTimeRatio=? Установка желаемых параметров производительности сборщика.  -XX:MaxGCPauseMillis=? Установка ограничения на максимальное время приостановки программы для сборки мусора. По умолчанию такого ограничения нет. При установке данного параметра следует помнить, что ограничение на время сборки может приводить к необходимости выполнять ее чаще, в результате чего будет страдать общая пропускная способность. Например, -XX:MaxGCPauseMillis=400, укажет JVM, что паузы на сборку мусора желательно не затягивать дольше, чем на 400 миллисекунд.  -XX:GCTimeRatio=? Указать желаемый порог пропускной способности (отношения времени работы программы ко времени сборки мусора). Например, при -XX:GCTimeRatio=49 JVM будет пытаться выполнять сборки таким образом, чтобы они суммарно занимали не больше 2% времени работы программы (отношение времени сборки ко времени работы программы будет 1 / (1 + 49)).  -XX:YoungGenerationSizeIncrement=? и -XX:TenuredGenerationSizeIncrement=? Устанавливают, на сколько процентов следует при необходимости увеличивать младшее и старшее поколение соотвественно. По умолчанию оба этих параметра равны 20.  -XX:AdaptiveSizeDecrementScaleFactor=? Задать скорость уменьшения размеров поколений (регулируется не процентами, а специальным фактором). Она указывает, во сколько раз уменьшение должно быть меньше увеличения. Эта опция распространяется на оба поколения Например, при -XX:AdaptiveSizeDecrementScaleFactor=2 каждое уменьшение поколения будет в два раза меньше, чем его увеличение (то есть оба поколения будут уменьшаться на 10% при -XX:GenerationSizeIncrement=20 и -XX:TenuredGenerationSizeIncrement=20).  Достоинства 1. Возможность автоматической подстройки под требуемые параметры производительности. 2. Меньшие паузы на время cборок. При наличии нескольких процессорных ядер выигрыш в скорости будет практически во всех приложениях. Недостатки 1. фрагментация памяти, конечно (но она не существенна для большинства приложений, так как сборщиком используется относительно небольшое количество потоков). В целом, Parallel GC — это простой, понятный и эффективный сборщик, подходящий для большинства приложений. У него нет скрытых накладных расходов, мы всегда можем поменять его настройки и ясно увидеть результат этих изменений.  **CONCURRENT MARK SWEEP GC (CMS GC)**  Сборщик CMS (расшифровывается как Concurrent Mark Sweep) появился в HotSpot VM в одно время с Parallel GC в качестве его альтернативы для использования в приложениях, имеющих доступ к нескольким ядрам процессора и чувствительных к паузам STW. В то время существовала еще одна альтернатива — Incremental GC, но он не прошел естественный отбор за неимением явных преимуществ. А CMS выжил. И хотя пик его популярности, видимо, уже прошел, на его внутреннее устройство интересно будет взглянуть, так как некоторые заложенные в него идеи перекочевали в более современный G1 GC. Использование CMS GC включается опцией -XX:+UseConcMarkSweepGC. Принципы работы Mark и Sweep обозначают два шага в процессе сборки мусора в старшем поколении: пометку выживших объектов и удаление мертвых объектов. Сборщик CMS получил свое название благодаря тому, что выполняет указанные шаги параллельно с работой основной программы. При этом CMS GC использует ту же самую организацию памяти, что и уже рассмотренные Serial / Parallel GC: регионы Eden + Survivor 0 + Survivor 1 + Tenured и такие же принципы малой сборки мусора. Отличия начинаются только тогда, когда дело доходит до полной сборки. В случае CMS ее называют старшей (major) сборкой, а не полной, так как она не затрагивает объекты младшего поколения. В результате, малая и старшая сборки здесь всегда разделены. Одним из побочных эффектов такого разделения является то, что все объекты младшего поколения (даже потенциально мертвые) могут играть роль корней при определении статуса объектов в старшем поколении. Важным отличием сборщика CMS от рассмотренных ранее является также то, что он не дожидается заполнения Tenured для того, чтобы начать старшую сборку. Вместо этого он трудится в фоновом режиме постоянно, пытаясь поддерживать Tenured в компактном состоянии.  Major GC Начинается она с остановки основных потоков приложения и пометки всех объектов, напрямую доступных из корней. После этого приложение возобновляет свою работу, а сборщик параллельно с ним производит поиск всех живых объектов, доступных по ссылкам из тех самых помеченных корневых объектов (эту часть он делает в одном или в нескольких потоках). Естественно, за время такого поиска ситуация в куче может поменяться, и не вся информация, собранная во время поиска живых объектов, оказывается актуальной. Поэтому сборщик еще раз приостанавливает работу приложения и просматривает кучу для поиска живых объектов, ускользнувших от него за время первого прохода. При этом допускается, что в живые будут записаны объекты, которые на время окончания составления списка таковыми уже не являются. Эти объекты называются плавающим мусором (floating garbage), они будут удалены в процессе следующей сборки. После того, как живые объекты помечены, работа основных потоков приложения возобновляется, а сборщик производит очистку памяти от мертвых объектов в нескольких параллельных потоках. При этом следует иметь в виду, что после очистки не производится упаковка объектов в старшем поколении, так как делать это при работающем приложении весьма затруднительно.  https://lh3.googleusercontent.com/VQl8NDTUNSC4xmD9IghVrJvquTiogVhULQfhRjMH2_Y2sDVwEetKqqQCwmxocxtaPcmV1pr4BLa3636MJlzGwB4LRPvOA_oqdlqef3967wc_f5qa5QC1R2J9Ne2Lzthp4EhH5Le7  Сборщик CMS достаточно интеллектуальный. Например, он старается разносить во времени малые и старшие сборки мусора, чтобы они совместно не создавали продолжительных пауз в работе приложения (дополнительные подробности об этом разнесении в комментариях). Для этого он ведет статистику по прошедшим сборкам и исходя из нее планирует последующие. Отдельно следует рассмотреть ситуацию, когда сборщик не успевает очистить Tenured до того момента, как память полностью заканчивается. В этом случае работа приложения останавливается, и вся сборка производится в последовательном режиме. Такая ситуация называется сбоем конкурентного режима (concurrent mode failure). Сборщик сообщает нам об этих сбоях при включенных опциях -verbose:gc или -Xloggc:filename. У CMS есть один интересный режим работы, называемый Incremental Mode, или i-cms, который заставляет его временно останавливаться при выполнении работ параллельно с основным приложением, чтобы на короткие периоды высвобождать ресурсы процессора (что-то вроде АБС у автомобиля). Это может быть полезным на машинах с малым количеством ядер. Но данный режим уже помечен как не рекомендуемый к применению и может быть отключен в будущих релизах, поэтому подробно его разбирать не будем. Ситуации STW Из всего сказанного выше следует, что при обычной сборке мусора у CMS GC существуют следующие ситуации, приводящие к STW:  Малая сборка мусора. Эта пауза ничем не отличается от аналогичной паузы в Parallel GC.  Начальная фаза поиска живых объектов при старшей сборке (так называемая initial mark pause). Эта пауза обычно очень короткая.  Фаза дополнения набора живых объектов при старшей сборке (известная также как remark pause). Она обычно длиннее начальной фазы поиска. В случае же возникновения сбоя конкурентного режима пауза может затянуться на достаточно длительное время. Настройка Так как подходы к организации памяти у CMS аналогичны используемым в Serial / Parallel GC, для него применимы те же опции определения размеров регионов кучи, а также опции автоматической подстройки под требуемые параметры производительности. Обычно CMS, основываясь на собираемой статистике о поведении приложения, сам определяет, когда ему выполнять старшую сборку, но у него также есть порог наполненности региона Tenured, при достижении которого должна обязательно быть инициирована старшая сборка. Этот порог можно задать с помощью опции -XX:CMSInitiatingOccupancyFraction=?, значение указывается в процентах. Значение -1 (иногда устанавливается по умолчанию) указывает на отключение сборки по такому условию. Достоинства 1. Ориентированность на минимизацию времен простоя, что является критическим фактором для многих приложений. Недостатки 1. Для минимизации времени простоя приходится жертвовать ресурсами процессора и зачастую общей пропускной способностью. 2. Фрагментация Tenured (данный сборщик не уплотняет объекты в старшем поколении). Этот факт в совокупности с наличием плавающего мусора приводит к необходимости выделять приложению (конкретно — старшему поколению) больше памяти, чем потребовалось бы для других сборщиков (Oracle советует на 20% больше). 3. Долгие паузы при потенциально возможных сбоях конкурентного режима могут стать неприятным сюрпризом. Хотя они не частые, и при наличии достаточного объема памяти CMS’у удается их полностью избегать. Тем не менее, такой сборщик может подойти приложениям, использующим большой объем долгоживущих данных. В этом случае некоторые его недостатки нивелируются.  **G1 (GARBAGE FIRST) GC** Он не является явным продолжением линейки Serial / Parallel / CMS, добавляющим параллельность еще в какую-нибудь фазу сборки мусора, а использует уже существенно отличающийся подход к задаче очистки памяти. G1 — самый молодой в составе сборщиков мусора виртуальной машины HotSpot. Он изначально позиционировался как сборщик для приложений с большими кучами (от 4 ГБ и выше), для которых важно сохранять время отклика небольшим и предсказуемым, пусть даже за счет уменьшения пропускной способности. На этом поле он конкурировал с CMS GC, хотя изначально и не так успешно, как хотелось бы. Но постепенно он исправлялся, улучшался, стабилизировался и, наконец, достиг такого уровня, что Oracle говорит о нем как о долгосрочной замене CMS, а в Open JDK даже серьезно рассматривают его на роль сборщика по умолчанию для серверных конфигураций в 9-й версии. G1 включается опцией Java -XX:+UseG1GC. Принципы работы В G1 изменили подход к организации кучи. Здесь память разбивается на множество регионов одинакового размера. Размер этих регионов зависит от общего размера кучи и по умолчанию выбирается так, чтобы их было не больше 2048, обычно получается от 1 до 32 МБ. Исключение составляют только так называемые громадные (humongous) регионы, которые создаются объединением обычных регионов для размещения очень больших объектов. Разделение регионов на Eden, Survivor и Tenured в данном случае логическое, регионы одного поколения не обязаны идти подряд и даже могут менять свою принадлежность к тому или иному поколению. Пример разделения кучи на регионы может выглядеть следующим образом (количество регионов сильно приуменьшено):  https://lh6.googleusercontent.com/-I1aVXasD77gLyl55KdsH1AJYuzl87bf96cOXotA87YunS68dRImy2Qtrl_2n8JMZZaIC2Ah5qa_f91nUhQnATAP36sJHsfVc1szRbRplLpXNKip58Rj4oJOIE3dWGJR2aD2xsIY  Малые сборки выполняются периодически для очистки младшего поколения и переноса объектов в регионы Survivor, либо их повышения до старшего поколения с переносом в Tenured. Над переносом объектов трудятся несколько потоков, и на время этого процесса работа основного приложения останавливается. Это уже знакомый нам подход из рассмотренных ранее сборщиков, но отличие состоит в том, что очистка выполняется не на всем поколении, а только на части регионов, которые сборщик сможет очистить не превышая желаемого времени. При этом он выбирает для очистки те регионы, в которых, по его мнению, скопилось наибольшее количество мусора и очистка которых принесет наибольший результат. Отсюда как раз название Garbage First — мусор в первую очередь. Mixed GC А с полной сборкой (точнее, здесь она называется смешанной (mixed)) все немного хитроумнее, чем в рассмотренных ранее сборщиках. В G1 существует процесс, называемый циклом пометки (marking cycle), который работает параллельно с основным приложением и составляет список живых объектов. За исключением последнего пункта, этот процесс выглядит уже знакомо для нас: 1. Initial mark. Пометка корней (с остановкой основного приложения) с использованием информации, полученной из малых сборок. 2. Concurrent marking. Пометка всех живых объектов в куче в нескольких потоках, параллельно с работой основного приложения. 3. Remark. Дополнительный поиск не учтенных ранее живых объектов (с остановкой основного приложения). 4. Cleanup. Очистка вспомогательных структур учета ссылок на объекты и поиск пустых регионов, которые уже можно использовать для размещения новых объектов. Первая часть этого шага выполняется при остановленном основном приложении. Следует иметь в виду, что для получения списка живых объектов G1 использует алгоритм Snapshot-At-The-Beginning (SATB), то есть в список живых попадают все объекты, которые были таковыми на момент начала работы алгоритма, плюс все объекты, созданные за время его выполнения. Это, в частности, означает, что G1 допускает наличие плавающего мусора, с которым мы познакомились при рассмотрении сборщика CMS. После окончания цикла пометки G1 переключается на выполнение смешанных сборок. Это значит, что при каждой сборке к набору регионов младшего поколения, подлежащих очистке, добавляется некоторое количество регионов старшего поколения. Количество таких сборок и количество очищаемых регионов старшего поколения выбирается исходя из имеющейся у сборщика статистики о предыдущих сборках таким образом, чтобы не выходить за требуемое время сборки. Как только сборщик очистил достаточно памяти, он переключается обратно в режим малых сборок. Очередной цикл пометки и, как следствие, очередные смешанные сборки будут запущены тогда, когда заполненность кучи превысит определенный порог. Смешанная сборка мусора в приведенном выше примере кучи может пройти вот так:  https://lh6.googleusercontent.com/-GEiOaF-nyM0k95bVjaYAdCKubPjsSRFvEk2zqex8TiH5RccloT14LEAzYf2Kh8bTcLcUUSTwg-_zTYOrylO3CHEM6lFL4-KOobJ-_A18aEiJZnukaYSmM5hMMiO9dW6A08UG2q0  Может оказаться так, что в процессе очистки памяти в куче не остается свободных регионов, в которые можно было бы копировать выжившие объекты. Это приводит к возникновению ситуации allocation (evacuation) failure, подобие которой мы видели в CMS. В таком случае сборщик выполняет полную сборку мусора по всей куче при остановленных основных потоках приложения.  Опираясь на уже упомянутую статистику о предыдущих сборках, G1 может менять количество регионов, закрепленных за определенным поколением, для оптимизации будущих сборок.  Гиганты  Существуют громадные регионы, в которых хранятся так называемые громадные объекты (humongous objects). С точки зрения JVM любой объект размером больше половины региона считается громадным и обрабатывается специальным образом:  Он никогда не перемещается между регионами.  Он может удаляться в рамках цикла пометки или полной сборки мусора.  В регион, занятый громадным объектом, больше никого не подселяют, даже если в нем остается свободное место.  Вообще, эти пункты иногда имеют далеко идущие последствия. Объекты большого размера, особенно короткоживущие, могут доставлять много неудобств всем типам сборщиков, так как не удаляются при малых сборках, а занимают драгоценное пространство в регионах старшего поколения (помните объекты-акселераты, обсуждавшиеся в предыдущей главе?) Но G1 оказывается более уязвимым к их негативному влиянию в силу того, что для него даже объект в несколько мегабайт (а в некоторых случаях и 500 КБ) уже является громадным.  Ситуации STW  Если резюмировать, то у G1 мы получаем STW в следующих случаях:  1. Процессы переноса объектов между поколениями. Для минимизации таких пауз G1 использует несколько потоков.  2. Короткая фаза начальной пометки корней в рамках цикла пометки.  3. Более длинная пауза в конце фазы remark и в начале фазы cleanup цикла пометки.  Настройка  -XX:MaxGCPauseMillis=?  Задает приемлемое максимальное время разовой сборки мусора. Хотя в документации Oracle и говориться, что по умолчанию время сборки не ограничено, но по факту это не всегда так.  -XX:ParallelGCThreads=? и -XX:ConcGCThreads=?  Задают количество потоков, которые будут использоваться для сборки мусора и для выполнения цикла пометок соответственно.  -XX:G1HeapRegionSize=?  Задает размер региона (по умолчанию установлен автоматический выбор размера региона). Значение должно быть степенью двойки, если мерить в мегабайтах. Например, -XX:G1HeapRegionSize=16m.  -XX:InitiatingHeapOccupancyPercent=?  Изменяет порог заполненности кучи (в процентах), при достижении которого инициируется выполнение цикла пометок и переход в режим смешанных сборок.  По умолчанию, этот порог равен 45%.  -XX:+UnlockExperimentalVMOptions  Включение дополнительных функций.  -XX:+AggressiveOpts  Использование экспериментальных настроек.  Достоинства 1. Считается, что G1 GC более аккуратно предсказывает размеры пауз, чем CMS, и лучше распределяет сборки во времени, чтобы не допустить длительных остановок приложения, особенно при больших размерах кучи. 2. Не фрагментирует память.  Недостатки 1. Ресурсы процессора, которые он использует для выполнения достаточно большой части своей работы параллельно с основной программой. 2. В результате страдает пропускная способность приложения. Целевым значением пропускной способности по умолчанию для G1 является 90%. Для Parallel GC, например, это значение равно 99%. Это, конечно, не значит, что пропускная способность с G1 всегда будет почти на 10% меньше, но данную особенность следует всегда иметь в виду.  **Статьи:** Виды неплохо расписаны: https://habrahabr.ru/post/269707/#comment\_8633685 (3 части) Общая инфа про подходы: https://ggenikus.github.io/blog/2014/05/04/gc/ https://plumbr.eu/handbook/garbage-collection-algorithms-implementations#serial-gc http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java  ***Pros and Cons of GC***  **Плюсы:**  Увеличение скорости разработки (не нужно заботиться о выделении и удалении памяти для объектов).  Уменьшена утечка памяти (выделенная память, которая не может быть освобождена).  Исследования показали, что автоматическая сборка мусора использует меньше циклов CPU, чем ручная очистка. Т.е. это миф, что автоматических сборщик мусора ухудшает производительность. Современные GC довольно ненавязчивые.  **Минусы:**  Low level управление памятью абстрагировано и скрыто от разработчиков. Т.е. не оставлено возможности для улучшения со стороны разработчиков. Вдобавок, если возникнет утечка памяти, будет сложно дебажить и исправлять.  ***JIT compiler***  JIT Compiler – JIT компилятор устраняет недостаток интерпретатора. Механизм выполнения использует интерпретатор для преобразования, но, как только находит повторяющийся код, он начинает использовать JIT компилятор, который уже компилирует весь байт-код целиком и преобразует его в нативный/родной/машинный код. Данный машинный код будет использоваться напрямую для повторяющихся методов, что улучшает производительность системы. 1. **Intermediate Code generator** – создает промежуточный код. 2. **Code Optimizer** – отвечает за оптимизацию промежуточного кода. 3. **Target Code Generator** – отвечает за создание машинного/нативного кода (Generating Machine Code/ Native Code). 4. **Profiler** – отвечает за поиск хотспотов (горячих точек), которые используются для пометки того, вызывается ли метод несколько раз, или нет.  https://lh4.googleusercontent.com/8GfCYXdFETpPU61GbhP8RexGU0VthAMcBKgKfxVB4R68es6Fq4SWBuxF5Aiorf20hoOYvgu8T1Q17hsH1STKXkYmHXfAOxEfiOuezmGnxHH9VD4TQEMfcT6ssJ8WMsspwRDkVLUy Что такое JIT-компилятор?  Концепция Just In Time Compiler (JIT) и более общая адаптивная оптимизация - хорошо известная концепция на многих языках помимо Java (.Net, Lua, JRuby).  Чтобы объяснить, что такое JIT Compiler, я хочу начать с определения концепции компилятора. Согласно wikipedia, компилятор представляет собой «компьютерную программу, которая преобразует исходный язык на другой компьютерный язык (целевой язык)».  Мы все знакомы со статическим java-компилятором (javac), который компилирует читаемые пользователем .java-файлы в байтовый код, который может быть интерпретирован файлами JVM-.class. Тогда что делает JIT? Ответ будет дан через минуту после объяснения того, что такое «Just in Time».  Согласно большинству исследований, 80% времени выполнения тратится на выполнение 20% кода. Было бы здорово, если бы был способ определить эти 20% кода и оптимизировать их. Именно это делает JIT - во время выполнения он собирает статистику, обнаруживает, что «горячий» код компилирует его из JVM, интерпретируемого байт-кодом (который хранится в файлах .class), в собственный код, который выполняется непосредственно операционной системой и сильно оптимизирует его. Самый маленький блок компиляции - единственный метод. Сбор статистики выполняется параллельно с выполнением программы специальными потоками. Во время сбора статистики компилятор делает гипотезы о кодовой функции, и по прошествии времени пытается доказать или опровергнуть их. Если гипотеза не доказана, код деоптимизируется и снова перекомпилируется.  Название «Hotspot» Sun (Oracle) JVM выбрано из-за способности этой виртуальной машины находить «горячие» пятна в коде.  Какие оптимизации делает JIT?  Давайте внимательно рассмотрим больше оптимизаций, сделанных JIT.  Встроенные методы - вместо вызова метода на экземпляр объекта он копирует метод в код вызывающего абонента. Горячие методы должны располагаться как можно ближе к вызывающему, чтобы предотвратить накладные расходы.   * Исключить блокировки, если монитор недоступен из других потоков * Заменить интерфейс прямыми вызовами метода для метода, реализованного только один раз, чтобы исключить накладные расходы виртуальных функций * Присоединение смежных синхронизированных блоков на одном объекте * Исключить мертвый код * Запись падающей памяти для энергонезависимых переменных * Удалить предварительную проверку NullPointerException и IndexOutOfBoundsException * Et cetera   Когда Java VM вызывает метод Java, он использует метод invoker, как указано в блоке метода загруженного объекта класса. Java VM имеет несколько методов invoker, например, используется другой invoker, если метод синхронизирован или является родным методом.  Компилятор JIT использует свой собственный invoker. Выпуски выпуска Sun проверяют бит доступа метода для значения ACC\_MACHINE\_COMPILED, чтобы уведомить интерпретатора о том, что код этого метода уже скомпилирован и сохранен в загруженном классе. Компилятор JIT компилирует блок метода в собственный код для этого метода и сохраняет его в блоке кода для этого метода. После компиляции кода устанавливается бит ACC\_MACHINE\_COMPILED, который используется на платформе Sun.  Как мы знаем, что делает JIT в нашей программе и как ее можно контролировать?  Прежде всего отключить JIT Djava.compiler = параметр NONE.  Существует два типа компиляторов JIT в Hotspot - один используется для клиентской программы и один для сервера (опция -server в параметрах VM). Программа, работающая на сервере, обычно имеет больший объем ресурсов, чем обычно работает программа, работающая на клиенте, и для максимальной производительности серверной программы. Следовательно, JIT на сервере больше потребляет ресурсы, и для сбора статистики требуется больше времени, чтобы сделать статистику более точной. Для сбора клиентской программы статика для метода длится 1500 вызовов методов, для сервера 15000. Эти значения по умолчанию могут быть изменены на  - XX:CompileThreshold=XXX VM parameter. Чтобы узнать, подходит ли значение по умолчанию для вас, попробуйте включить "XX:+PrintCompilation" and "-XX:-CITime" параметры, которые печатают статистику JIT и процессор времени, затраченный JIT.  Ориентиры  Большинство тестов показывают, что код JITed работает в 10-20 раз быстрее, чем интерпретируемый код. Есть много тестов.  Стоит упомянуть, что программы, которые работают в режиме JIT, но все еще находятся в «режиме обучения», работают намного медленнее, чем программы, не поддерживающие JITed.  Недостатки JIT JIT Повышает уровень непредсказуемости и сложности в Java-программе. Он добавляет еще один уровень, который разработчики действительно не понимают. Пример возможных ошибок - «происходит до отношений» в параллелизме. JIT может легко переупорядочить код, если изменение безопасно для программы, работающей в одном потоке. Для решения этой проблемы разработчики делают намеки на JIT, используя «синхронизированное» слово или явное блокирование.  Увеличивает объем памяти “non heap”. Код JITed хранится в генерации «Code Cache».  Расширенный JIT  JIT и сбор мусора.  Для выполнения GC программа должна достигать безопасных точек. Для этого JIT вводит выходные значения через регулярные интервалы в собственном коде.  В дополнение к сканированию стека для поиска корневых ссылок регистры должны проверяться, поскольку они могут содержать объекты, созданные JIT.  https://dzone.com/articles/just-time-compiler-jit-hotspot  ***Types of references (strong, weak. etc)***  <https://habrahabr.ru/post/169883/>  Начиная с версии 1.2 в Java появился пакет java.lang.ref.\* с классами SoftReference, WeakReference, PhantomReference. Эти классы помогают бороться с OutOfMemoryError.  Общее Описание  В общих чертах Garbage Collector работает так:  при запуске сборщика виртуальная машина рекурсивно находит, для всех потоков, все доступные объекты в памяти и помечает их неким образом.  на следующем шаге GC удаляет из памяти все непомеченные объекты. Таким образом, после чистки, в памяти будут находиться только те объекты, которые могут быть полезны программе.  В Java есть несколько видов ссылок. Есть StrongReference — это самые обычные ссылки которые мы создаем каждый день (StringBuilder builder = new StringBuilder(); // builder это и есть strong-ссылка на объект StringBuilder.) И есть 3 «особых» типа ссылок — SoftReference, WeakReference, PhantomReference. По сути, различие между всеми типами ссылок только одно — поведение GC с объектами, на которые они ссылаются:  SoftReference — если GC видит что объект доступен только через цепочку soft-ссылок, то он удалит его из памяти. Это не обязательно так + не факт, что удаление произойдет сразу.  WeakReference — если GC видит что объект доступен только через цепочку weak-ссылок, то он удалит его из памяти.  PhantomReference — если GC видит что объект доступен только через цепочку phantom-ссылок, то он его удалит из памяти. После нескольких запусков GC.  Эти 3 типа ссылок наследуются от одного родителя — Reference, у которого они собственно и берут все свои public методы и конструкторы.  **StringBuilder builder = new StringBuilder();**  **SoftReference<StringBuilder> softBuilder = new SoftReference(builder);**  После выполнения этих двух строчек у нас будет 2 типа ссылок на 1 объект StringBuilder: 1. builder — strong-ссылка 2. softBuilder — soft-ссылка (формально это strong-ссылка на soft-ссылку, но для простоты я буду писать soft-ссылка) И если во время выполнения программы, переменная builder станет недоступной, но при этом ссылка на объект, на который ссылается softBuilder, будет еще доступна И запустится GC -> то объект StringBuilder будет помечен как доступный только через цепочку soft-ссылок. Рассмотрим доступные методы:  softBuilder.get() — вернет strong-ссылку на объект StringBuilder в случае если GC не удалил этот объект из памяти. В другом случае вернется null.  softBuilder.clear() — удалит ссылку на объект StringBuilder (то есть soft-ссылки на этот объект больше нет) Все то же самое работает и для WeakReference и для PhantomReference. Правда, PhantomReference.get() всегда будет возвращать null. Есть еще такой класс – ReferenceQueue. Он позволяет отслеживать момент, когда GC определит что объект более не нужен и его можно удалить. Именно сюда попадает Reference объект после того как объект на который он ссылается удален из памяти. При создании Reference мы можем передать в конструктор ReferenceQueue, в который будут помещаться ссылки после удаления.  **ДЕТАЛИ SOFTREFERENCE** Особенности GC GC ведет себя следующим образом, когда видит что объект доступен только по цепочке soft-ссылок:  GC начал свою работу и проходит по всем объектам в куче.  В случае, если объект в куче это Reference, то GC помещает этот объект в специальную очередь в которой лежат все Reference объекты.  После прохождения по всем объектам GC берет очередь Reference объектов и по каждому из них решает удалять его из памяти или нет. Как именно принимается решение об удалении объекта — зависит от JVM. Но общий контракт звучит следующим образом: GC гарантировано удалит с кучи все объекты, доступные только по soft-ссылке, перед тем как бросит OutOfMemoryError. Принятие решения об удалении В реализации SoftReference видно, что в классе есть 2 переменные   * private static long clock * private long timestamp.   Каждый раз при запуске GC, он устанавливает текущее время в переменную clock.  Каждый раз при создании SoftReference, в timestamp записывается текущее значение clock. timestamp обновляется каждый раз при вызове метода get() (каждый раз, когда мы создаем strong-ссылку на объект). Это позволяет вычислить, сколько времени существует soft-ссылка после последнего обращения к ней. Обозначим этот интервал буквой I. Буквой F обозначим количество свободного места в куче в MB(мегабайтах). Константой MSPerMB обозначим количество миллисекунд, сколько будет существовать soft-ссылка для каждого свободного мегабайта в куче. Дальше все просто, если I <= F \* MSPerMB, то не удаляем объект. Если больше то удаляем. Для изменения MSPerMB используем ключ -XX:SoftRefLRUPolicyMSPerMB. Дефолтное значение — 1000 ms, а это означает что soft-ссылка будет существовать (после того как strong-ссылка была удалена) 1 секунду за каждый мегабайт свободной памяти в куче. Главное не забыть что это все примерные расчеты, так как фактически soft-ссылка удалится только после запуска GC. Обратите внимание на то, что для удаления объекта, I должно быть строго больше чем F \* MSPerMB. Из этого следует что созданная SoftReference проживет минимум 1 запуск GC. (\*если не понятно почему, то это останется вам домашним заданием). В случае VM от IBM, привязка срока жизни soft-ссылки идет не к времени, а к количеству переживших запусков GC. Применение Главная качество SoftReference в том, что JVM сама следит за тем нужно удалять из памяти объект или нет. И если осталось мало памяти, то объект будет удален. Это именно то, что нам нужно при кэшировании. Кэширование с использованием SoftReference может пригодится в системах чувствительных к объему доступной памяти. Например, обработка изображений: Допустим, есть громадное изображение, которое находиться где-то в файловой системе и это изображение всегда статично. Иногда пользователь хочет соединить это изображение с другим изображением.  **public class ImageProcessor {  private static final String IMAGE\_NAME = "bigImage.jpg";  public InputStream concatenateImegeWithDefaultVersion(InputStream userImageAsStream) {  InputStream defaultImage = this.getClass().getResourceAsStream(IMAGE\_NAME);                  *// calculate and return concatenated image*  }             }**  Недостатков в таком подходе много, но один из них это то что мы должны каждый раз загружать с файловой системы изображение. А это не самая быстрая процедура. Давайте тогда будем кэшировать загруженное изображение. Вот вторая версия:  **public class CachedImageProcessor {  private static final String IMAGE\_NAME = "bigImage.jpg";  private InputStream defaultImage;               public InputStream concatenateImegeWithDefaultVersion(InputStream userImageAsStream) {  if (defaultImage == null) {  defaultImage = this.getClass().getResourceAsStream(IMAGE\_NAME);  }                  *// calculate and return concatenated image*  }             }**  Этот вариант уже лучше, но проблема все ровно есть. Изображение большое и забирает много памяти. Наше приложение работает со многими изображениями и при очередной попытке пользователя обработать изображение, легко может свалиться OutOfMemoryError. И что с этим можно сделать? Получается, что нам нужно выбирать, либо быстродействие либо стабильность. Но мы то знаем о существовании SoftReference. Это поможет нам продолжать использовать кеширование, но при этом в критических ситуациях выгружать их из кэша для освобождения памяти. Да еще и при этом нам не нужно беспокоиться о детектировании критической ситуации. Вот так будет выглядеть наша третья реализация:  **public class SoftCachedImageProcessor {  private static final String IMAGE\_NAME = "bigImage.jpg";  private SoftReference<InputStream> defaultImageRef = new SoftReference(loadImage());   public InputStream concatenateImegeWithDefaultVersion(InputStream userImageAsStream) {                  if (defaultImageRef.get() == null) {        *//  1*  defaultImage = this.getClass().getResourceAsStream(IMAGE\_NAME);  defaultImageRef = new SoftReference(defaultImage);  }            defaultImage = defaultImageRef.get();        *//  2*  *// calculate and return concatenated image*  }             }**  Эта версия не идеальна, но она показывает как просто мы можем контролировать размер занимаемый кэшем, а точнее возложить контроль на виртуальную машину. Опасность данной реализации заключается в следующем. В строчке №1 мы делаем проверку на null, фактически мы хотим проверить, удалил GC данные с памяти или нет. Допустим, что не удалил. Но перед выполнением строки №2 может начать работу GC и удалить данные. В таком случае результатом выполнения строчки №2 будет defaultImage = null. Для безопасной проверки существования объекта в памяти, нам нужно создать strong-ссылку, defaultImage = defaultImageRef.get(); Вот как будет выглядеть финальная реализация:  **public class SoftCachedImageProcessor {  private static final String IMAGE\_NAME = "bigImage.jpg";  private SoftReference<InputStream> defaultImageRef = new SoftReference(loadImage());;   public InputStream concatenateImegeWithDefaultVersion(InputStream userImageAsStream) {  defaultImage = defaultImageRef.get();  if (defaultImage == null) {  defaultImage = this.getClass().getResourceAsStream(IMAGE\_NAME);  defaultImageRef = new SoftReference(defaultImage);  }                  *// calculate and return concatenated image*  }             }**  Пойдем дальше. java.lang.Class тоже использует SoftReference для кэширования. Он кэширует данные о конструкторах, методах и полях класса. Интересно посмотреть, что именно они кешируют. После того как решено использовать SoftReference для кеширования, нужно решить что именно кешировать. Допустим нам нужно кешировать List. Мы можем использовать как List<SoftReference> так и SoftReference<List>. Второй вариант более приемлемый. Нужно помнить, что GC применяет специфическую логику при обработке Reference объектов, да и освобождение памяти будет происходить быстрее если у нас будет 1 SoftReference а не их список. Это мы и видим в реализации Class — разработчики создали soft-ссылку на массив конструкторов, полей и методов. Если говорить про производительность, то стоить отметить что часто, ошибочно, люди используют WeakReference для построения кэша там где стоит использовать SoftReference. Это приводит к низкой производительности кэша. На практике weak-ссылки быстро будут удалены из памяти, как только исчезнут strong-ссылки на объект. И когда нам реально понадобиться вытянуть объект с кэша, мы увидим что его там уже нет. Ну и еще один пример использования кэша на основе SoftReference. В Google Guava есть класс MapMaker. Он поможет нам построить ConcurrentMap в которой будут следующая особенность — ключи и значения в Map могут заворачиваться в WeakReference или SoftReference. Допустим в нашем приложении есть данные, которые может запросить пользователь и эти данные достаются с базы данных очень сложным запросом. Например, это будет список покупок пользователя за прошлый год. Мы можем создать кэш в котором значения (список покупок) будут храниться с помощью soft-ссылок. А если в кэше не будет значения то нужно вытянуть его с БД. Ключом будет ID пользователя. Вот как может выглядеть реализация:  **ConcurrentMap<Long, List<Product>> oldProductsCache = new MapMaker().softValues().            .makeComputingMap(new Function<User, List<Product>>() {                    @Override                    public List<Product> apply(User user) {                      return loadProductsFromDb(user);                    }              });**  **WeakReference**  Особенности GC Теперь рассмотрим более детально, что же собой представляет WeakReference. Когда GC определяет, что объект доступен только через weak-ссылки, то этот объект «сразу» удаляется из памяти. Тут стоить вспомнить про ReferenceQueue и проследить за порядком удаления объекта с памяти. Напомню что для WeakReference и SoftReference алгоритм попадания в ReferenceQueue одинаковый. Итак, запустился GC и определил что объект доступен только через weak-ссылки. Этот объект был создан так:  **StrIngBuilder AAA = new StringBuilder(); ReferenceQueue queue = new ReferenceQueue(); WeakReference weakRef = new WeakReference(AAA, queue);**  Сначала GC очистит weak-ссылку, то есть weakRef.get() – будет возвращать null. Потом weakRef будет добавлен в queue и соответственно queue.poll() вернет ссылку на weakRef. Вот и все что хотелось написать про особенности работы GC с WeakReference. Теперь посмотрим, как это можно использовать. Применение Ну конечно WeakHashMap. Это реализация Map<K,V> которая хранит ключ, используя weak-ссылку. И когда GC удаляет ключ с памяти, то удаляется вся запись с Map. Думаю не сложно понять, как это происходит. При добавлении новой пары <ключ, значение>, создается WeakReference для ключа и в конструктор передается ReferenceQueue. Когда GC удаляет ключ с памяти, то ReferenceQueue возвращает соответствующий WeakReference для этого ключа. После этого соответствующий Entry удаляется с Map. Все довольно просто. Но хочется обратить внимание на некоторые детали. WeakHashMap не предназначена для использования в качестве кэша. WeakReference создается для ключа а не для значения. И данные будут удалены только после того как в программе не останется strong-ссылок на ключ а не на значение. В большинстве случаев это не то чего вы хотите достичь кэшированием. Данные с WeakHashMap будут удалены не сразу после того как GC обнаружит что ключ доступен только через weak-ссылки. Фактически очистка произойдет при следующем обращении к WeakHashMap. В первую очередь WeakHashMap предназначен для использования с ключами, у которых метод equals проверяет идентичность объектов (использует оператор ==). Как только доступ к ключу потерян, его уже нельзя создать заново. Хорошо, тогда в каких случаях удобно использовать WeakHashMap? Допустим нам нужно создать XML документ для пользователя. Конструированием документа будут заниматься несколько сервисов, которые на вход будут получать org.w3c.Node в который будут добавлять необходимые элементы. Так же для сервисов нужно много информации о пользователе из Базы Данных. Эти данные мы будем складировать в классе UserInfo. Класс UserInfo занимает много места в памяти и актуален только для построения конкретного XML документа. Кешировать UserInfo не имеет смысла. Нам нужно только ассоциировать его с документом и желательно удалить из памяти, когда документ более не используется программой. Все что нам нужно сделать:  **private static final NODE\_TO\_USER\_MAP =**  **new WeakHashMap<Node, UserInfo>()**;  Создание XML документа будет выглядеть примерно так:  **Node mainDocument = createBaseNode(); NODE\_TO\_USER\_MAP.put(mainDocument, loadUserInfo());**  Ну а вот чтение:  **UserInfo userInfo = NODE\_TO\_USER\_MAP.get(mainDocument); If(userInfo != null) {  // … }**  UserInfo будет находиться в WeakHashMap до тех пор пока GC не заметит, что на mainDocument остались только weak-ссылки.  Другой пример использования WeakHashMap. Многие знают про метод String.intern(). Так вот с помощью WeakReference можно создать нечто подобное. (Давайте не будет обсуждать, в рамках этой статьи, целесообразность этого решения, и примем факт, что у этого решения есть некоторые преимущества по сравнению с intern()). Итак, у нас есть ооочень много строк. Мы знаем что строки повторяются. Для сохранения памяти мы хотим использовать повторно уже существующие объекты, а не создавать новые объекты для одинаковых строк. Вот как в этом нам поможет WeakHashMap:  **private static Map<String, WeakReference<String>> stringPool = new WeakHashMap<String, WeakReference<String>>;  public String getFromPool(String value) {  WeakReference<String> stringRef = stringPool.get(value);  if (stringRef == null || stringRef.get() == null ) {  stringRef = new WeakReference<String>(value);  stringPool.put(value, stringRef);  }   return stringRef.get(); }**  И на последок добавлю, что WeakReference используется во многих классах – Thread, ThreadLocal, ObjectOutpuStream, Proxy, LogManager. Вы можете посмотреть на их реализацию для того чтоб понять в каких случаях вам может помочь WeakReference.  **PhantomReference**  Особенности GC Особенностей у этого типа ссылок две. Первая это то, что метод get() всегда возвращает null. Именно из-за этого PhantomReference имеет смысл использовать только вместе с ReferenceQueue. Вторая особенность – в отличие от SoftReference и WeakReference, GC добавит phantom-ссылку в ReferenceQueue после того как выполниться метод finalize(). То есть фактически, в отличии от SoftReference и WeakReference, объект еще есть в памяти.  Практика На первый взгляд не ясно как можно использовать такой тип ссылок. Для того чтобы объяснить как их использовать, ознакомимся сначала с проблемами возникающими при использовании метода finalize(): переопределение этого метода позволяет нам очистить ресурсы связанные с объектом. Когда GC определяет что объект как более недоступный, то перед тем как удалит его из памяти, он выполняет этот метод. Вот какие проблемы с этим связаны: GC запускается непредсказуемо, мы не можем знать когда будет выполнен метод finalize() Методы finalize() запускаются в одном потоке, по очереди. И до тех пор, пока не выполниться этот метод, объект не может быть удален с памяти Нет гарантии, что этот метод будет вызван. JVM может закончить свою работу и при этом объект так и не станет недоступным. Во время выполнения метода finalize() может быть создана strong-ссылка на объект и он не будет удален, но в следующий раз, когда GC увидит что объект более недоступен, метод finalize() больше не выполнится. Вернемся к PhantomReference. Этот тип ссылок в комбинации с ReferenceQueue позволяет нам узнать, когда объект более недоступен и на него нет других ссылок. Это позволяет нам сделать очистку ресурсов, используемых объектом, на уровне приложения. В отличии от finalize() мы сами контролируем процесс очистки ресурсов. Помимо этого, мы можем контролировать процесс создания новых объектов. Допустим у нас есть фабрика, которая будет возвращать нам объект HdImage. Мы можем контролировать, сколько таких объектов будет загружено в память:  **public HdImageFabric {  public static final int IMAGE\_LIMIT = 10;  public static int count = 0;  public static ReferenceQueue<HdImage> queue = new ReferenceQueue<HdImage>();   public HdImage loadHdImage(String imageName) {  while (true) {  if (count < IMAGE\_LIMIT) {  return wrapImage(loadImage(imageName));   } else {  Reference<HdImage> ref = queue.remove(500);  if (ref != null) {  count--;  System.out.println(“remove old image”);  }  }  }  }   private HdImage wrapImage(HdImage image) {  PhantomReference<HdImage> refImage = new PhantomReference(image, queue);  count++;  return refImage ;  } }**  Этот пример не потокобезопасный и имеет другие недостатки, но зато он показывает, как можно использовать на практике PhantomReference. Из-за того что метод get() всегда возвращает null, становится непонятным а как все же понять какой именно объект был удален. Для этого нужно создать собственный класс, который будет наследовать PhantomReference, и который содержит некий дескриптор, который в будущем поможет определить какие ресурсы нужно чистить.  Когда вы используете PhantomReference нужно помнить о следующих вещах: Контракт гарантирует что ссылка появится в очереди после того как GC заметит что объект доступен только по phantom-ссылкам и перед тем как объект будет удален из памяти. Контракт не гарантирует, что эти события произойдут одно за другим. В реальности между этими событиями может пройти сколько угодно времени. Поэтому не стоит опираться на PhantomReference для очистки критически важных ресурсов. Выполнение метода finalize() и добавление phantom-ссылки в ReferenceQueue выполняется в разных запусках GC. По этому если у объекта переопределен метод finalize() то для его удаления необходимы 3 запуска GC, а если метод не переопределен, то нужно, минимум, 2 запуска GC.   |  |  | | --- | --- | | **Topic 6** | **OOP principles** | |
|  | ***AIPI***  I dunno what it is it.  ***SOLID***  Принцип **SOLID** в упрощенном варианте означает, что когда при написании кода используется несколько принципов вместе, то это значительно облегчает дальнейшую поддержку и развитие программы. Полностью акроним расшифровывается так:  **Single responsibility principle** — принцип единственной обязанности (на каждый класс должна быть возложена одна-единственная обязанность);  **Open/closed principle** — принцип открытости/закрытости (программные сущности должны быть закрыты для изменения но открыты для расширения);  **Liskov substitution principle** — принцип подстановки Барбары Лисков (функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Подклассы не могут замещать поведения базовых классов. Подтипы должны дополнять базовые типы);  **Interface segregation principle** — принцип разделения интерфейса (много специализированных интерфейсов лучше, чем один универсальный);  **Dependency inversion principle** — принцип инверсии зависимостей (зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций);  ***IoC***  Инверсия управления (англ. **Inversion of Control**, IoC) — важный принцип объектно-ориентированного программирования, используемый для уменьшения зацепления.  Одной из реализаций IoC в применении к управлению зависимостями является внедрение зависимостей (англ. **dependency injection**).  Предоставление программисту инструментов внедрения зависимостей дало значительно бо́льшую гибкость в разработке и удобство в тестировании кода.  ***DRY***  **DRY** — расшифровывается как **Don’t Repeat Youself** — не повторяйся, также известен как DIE — Duplication Is Evil — дублирование это зло. Этот принцип заключается в том, что нужно избегать повторений одного и того же кода. Лучше использовать универсальные свойства и функции.  ***KISS***  **KISS — Keep It Simple, Stupid** — не усложняй! Смысл этого принципа программирования заключается в том, что стоит делать максимально простую и понятную архитектуру, применять шаблоны проектирования и не изобретать велосипед. |