Содержание:

1. Структура JDK и ЖЦ приложения.
2. Работа JVM (порядок инициализации при создании объекта)
3. Память Java
4. Garbage collector

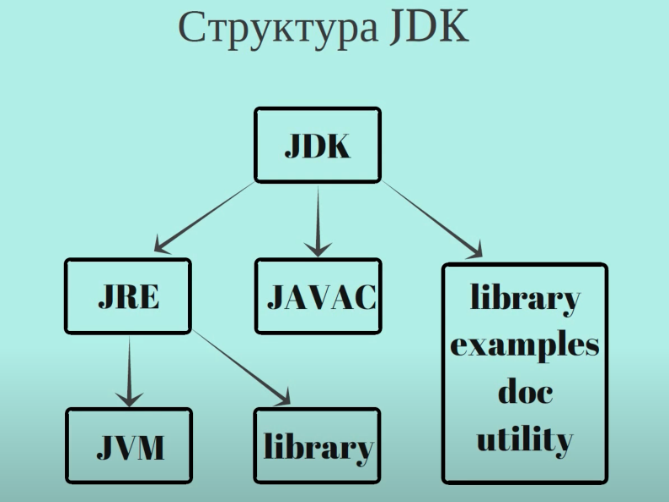
**Структура JDK и жизненный цикл Java приложения.**

JDK – комплект ПО, необходимое для написания, тестирования и запуска Java приложений, включает в себя инструменты для разработки программ и средства для запуска программ (JRE, javac, API, библиотеки классов, отладчик, примеры кода, утилиты)

Javac – компилятор, проверяющий синтаксис, лексику, семантику исходного кода (может его оптимизировать) генерирующий файл с расширением .class, содержащий байт-код. Генерит байт код из исходного кода.

Java Runtime Environment – среда выполнения Java-приложения, которая включает в себя JVM и набор библиотек Java;

JVM – программа для выполнения байт кода, для каждой ОС своя, что обеспечивает наилучшую производительность для каждой конкретной платформы (с помощью различных вариаций JVM, однажды скомпилированная в байт-код программа может быть запущена на любой платформе WORA - Write Once, Run Anywhere) работает с байт-кодом (файлами с расширением .class). Состав JVM – Execution engine (движок), Загрузчик классов, сборщик мусора, интерпретатор, JIT-компилятор, компоненты управления потоками etc.



***Жизненный цикл программы:***

1)Создание файла с кодом, имеет расширение .java (Исходный код) – написан на высокоуровневом языке, понятен человеку.

2)Компиляция Java-файла в байт-код, понятный JVM. Исполняется с помощью javac, на выходе получаем файл с расширением .class

3)Исполнение байт-кода на JVM – выполнение интерпретации в машинный код или JIT-компиляции.

*Компиляция и интерпретация:*

Компиляция – трансляция исходного кода в другую форму без исполнения этого кода (javac транслирует в исходный код в байт-код, JIT транслирует байт-код в машинный код).

Интерпретация - пооператорный (покомандный, построчный) анализ, обработка и выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется без её выполнения). JVM использует комбинированный подход – сочетает интерпретацию байт-кода в машинный код и JIT-компиляцию – смотри ниже.

***Работа JVM***

Чтобы получить код, который можно исполнять на JVM необходимо выполнить:

1. Загрузку класса
2. Связывание или линковку, которая состоит из 3х этапов: верификация, проверка, разрешение

3) Инициализация

**1) Загрузка класса** – поиск соответствующего .class файла, его чтение и передача содержимого в JVM (загрузка в память) и создание экземпляра класса Class.

В Java реализована динамическая (ленивая загрузка классов) – класс загружается только при обращении к нему в коде (создание объекта, обращение к статик переменной или статик методу).

Типы загрузчиков классов:

**Bootstrap classloader** – базовый нативный загрузчик (является частью JVM, не входит в иерархию классов Classloader). Загружает остальные загрузчики (так как они являются java классами, которые находятся в java.lang) и все базовые классы из java.base типа java.lang и java.util.

Нельзя получить доступ из исполняемой среды – String.class.getClassLoader==null – что означает, что String загружает базовый загрузчик и доступа к нему нет. При загрузке этим загрузчиком не выполняется проверка классов, т.к. JVM изначально известно, что все эти классы уже проверены. Загружает классы из rt.jar (с Java 9 введена модульная система jigsaw и грузится как-то по-другому.)

**Extension classloader/Platform Classloader** – Наследник абстрактного класса Classloader, загружает классы, которые включены в JDK, но не являются частью его ядра (http, sql и др.)

**System classloader/Application Classloader**. Загружает классы самого приложения, классы в, зависимости, которые переданы в classpath, то есть все классы, которые не являются частью ядра или платформы.

Пользовательские загрузчики классов – наследники абстрактного класса ClassLoader. [Про создание и назначение пользовательских загрузчиков.](https://for-each.dev/lessons/b/-java-classloaders)

**Принципы загрузки классов:**

**Делегирование**

Запрос на загрузку класса после поиска класса в кэше передается родительскому загрузчику, и попытка загрузить класс самостоятельно выполняется, только если родительский загрузчик не смог найти и загрузить класс. Такой подход позволяет загружать классы тем загрузчиком, который максимально близко находится к базовому. Так достигается максимальная область видимости классов. Каждый загрузчик ведет учет классов, которые были загружены именно им, помещая их в свой кэш. Множество этих классов и называется областью видимости.

**Видимость**

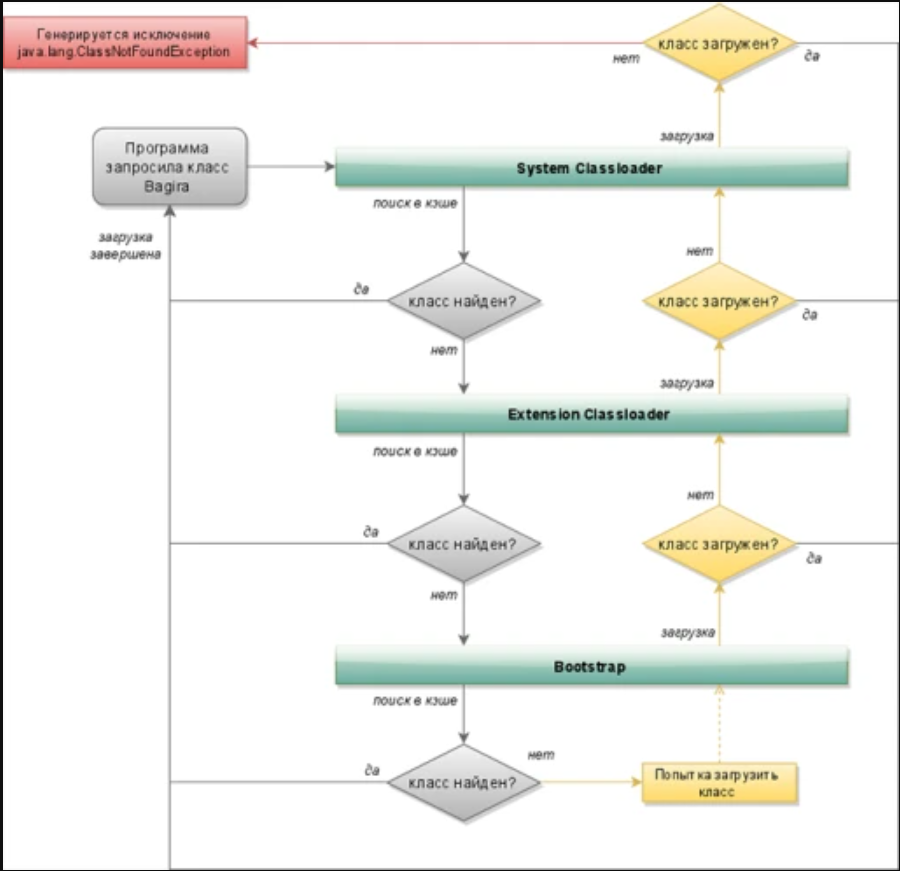
Загрузчик видит только «свои» классы и классы «родителя» и понятия не имеет о классах, которые были загружены его «потомком».

**Уникальность**

Класс может быть загружен только однажды. Механизм делегирования позволяет убедиться, что загрузчик, инициирующий загрузку класса, не перегрузит загруженный ранее в JVM класс.

**Схема загрузки классов по принципу делегирования** – Поиск класса в кэше загрузчика, если в кэше нет, то передается выше по иерархии и процесс повторяется, как только доходит до bootstrap, он пытается загрузить этот класс, если класс не найден, то выбрасывается исключение ClassNotFoundException и управление передается загрузчику ниже по иерархии, если ни один из загрузчиков не смог загрузить класс из известного ему источника, то вылетает ClassNotFoundException, если класс найден, то возвращается объект Class.

С помощью собственных загрузчиков имеется возможность создать ситуацию, когда один и тот же класс будет загружен одновременно несколькими загрузчиками. Для этого достаточно определить два загрузчика на одном уровне иерархии. Подобные дублированные классы будут рассматриваться JVM как различные и попытка привести объект одного класса к типу другого вызовет ClassCastException. Дело в том, что с точки зрения JVM уникальный идентификатор класса образует пара, состоящая из полного имени класса и загрузчика.



**2) Связывание или линковка –** проверка корректности кода, выделение памяти под его компонентов и преобразование символьных ссылок в ссылки на области памяти.  
Состоит из 3х шагов:  
*— verification* или проверка байт-кода: дополнительная проверка класса на корректность его структуры, то есть на соответствие спецификации языка и требованиям безопасности (локальные переменные инициализированы перед использованием, типы параметров соответствуют аргументам, не нарушаются права доступа, методы не приводят к переполнению стека и тд.).

*— preparation* или подготовка: на данном этапе выделяется память в MetaSpace под статические поля, происходит их инициализация значениями по умолчанию;  
*— resolution* или разрешение: Преобразование символьных ссылок на имена классов, методов и полей в прямые ссылки на определенную область памяти.

Этот этап необязательно следует за этапом подготовки, он может происходить на этапе компиляции, загрузки, после подготовки, что объединяется в общую концепцию: Eager Resolution – разрешение до выполнения программы. Lazy Resolution – разрешение, которое происходит динамически во время выполнения программы. Если компилятор или JVM может точно определить, какой метод или его реализация будет вызвана, то он применит Eager resolution, если однозначно это не определить, то Lazy. (Помимо статических методов, например, если есть виртуальный метод, но у его класса нет наследников, то есть компилятор или JVM могут явно определить, что он не мог быть переопределен может быть применено Eager resolution) – непроверенная инфа.

**Статическое (раннее) и динамическое (позднее) связывание:**

Статическое – происходит на этапе компиляции, применимо к приватным (неявно final – нельзя переопределить) и статическим методам, так как их нельзя переопределить, и не переопределённых виртуальных методов, так как компилятор заранее знает, какой метод будет вызван. Определяется за счет типа переменной.

Динамическое – происходит в рантайме, связывание каждого переопределенного метода, с его конкретной реализацией, т.е. определение, какая реализация виртуального метода должна быть вызвана на основе типа объекта, на котором был вызван метод. (Динамическая диспетчеризация).

**3) Инициализация полученного объекта Class**  
Выполняются статические блоки инициализации и инициализируются статические переменные – в порядке их написания в коде. Данный этап начинается только при обращении к статическому полю или методу класса, или создании первого объекта класса.

*Полный порядок инициализации класса при создании объекта:*

* Инициализация статических полей и блоков в Parent-классе в порядке их объявления
* Инициализация статических полей и блоков в Child-классе в порядке объявления
* Нестатические Parent поля и блоки инициализации
* Конструктор Parent класса
* Нестатические Child поля и блоки инициализации
* Конструктор Child класса

Загрузка класса сопровождается загрузкой всех его родительских классов и интерфейсов.

4)Исполнение байт кода на JVM:

Интерпретация – построчное исполнение байт кода с помощью Execution engine (движок JVM). Это достаточно медленный процесс, для увеличения производительности могут быть задействованы механизмы компиляции байт-кода: AOT (появился с Java 9) и JIT.

Динамическая компиляция - JIT (Just in time) – части кода, которые повторяются многократно по мере интерпретации «горячий код», будут скомпилированы в машинный код, который сразу будет исполняться на процессоре и не будет требовать повторной интерпретации, в случае, если снова встретится в коде. Требует место для хранения этого кода – code cash. Встает проблема разделения ресурсов – использовать ресурсы машины на работу приложения или на JIT компиляцию, чтобы произвести более сложную оптимизацию.

Статическая компиляция - AOT (Ahead of Time) – Трансляция в машинный код происходит до исполнения программы. Возможность делать сложные оптимизации в угоду будущей производительности, но компиляция ограничена только информацией, которая присутствует в байт-коде т.е. классы, которые получились динамически (рефлексией или по сети) не могут быть статически скомпилированы

***Память в Java***

Подразделяется на **Non-heap** и **Heap**.

**Non-heap** включает в себя: Stack, Method Area, Code cash, Native method stack.

*Stack* – область памяти, выделяемого для каждого потока (потокобезопасен, т.к. нет разделяемого доступа), состоит из блоков – фреймов, каждый блок соответствует вызову определенного метода потока и отвечает за хранение локальных переменных этого метода (переданных в качестве аргумента или созданных внутри метода). То есть на stack хранятся примитивные типы, не принадлежащие объектам и ссылки на объекты. Работает по принципу LIFO.

*Method area* – Общее понятие, которое описывает область памяти для хранения различных метаданных класса: информация о загруженных классах (поля, методы, константы), байт-код методов, пул констант – константы, используемые в байт-коде.

*Реализации*:

PermGen (до Java 8) – помимо основного, содержал string pool (до Java 7), не мог динамически расширяться (со временем вылетал OutOfMemoryError)

*MetaSpace* – выполнял основные функции, не содержит string pool, использует динамическое расширение (по умолчанию ограничен размером нативной памяти).

*Code cash* – область памяти, предназначенная для JIT компилированного кода.

*Native method stack* – Используется для выполнения нативных методов, которые вызываются из Java-кода, но реализованы на языке более низкого уровня.

**Heap** – Область памяти используются для хранения объектов и их полей (примитив, принадлежащий объекту, хранится в куче). Является общим для всех потоков – не потокобезопасен. С Java 7 содержит String pool.

**Структура Heap и сборка мусора:**

Куча разбита на 2 области: Young generation и Old generation, такое разделение обосновано гипотезой поколений (большинство объектов умирают молодыми и старые объекты редко ссылаются на молодые).

Сборщик мусора (GC) – компонент JVM автоматически управляющий памятью приложения, освобождая ее от мертвых объектов. Также участвует в аллокации памяти – выделяет участки памяти в хипе (thread local allocation buffer) по запросу потока (то есть выделяет сразу кусок памяти на поток для того, чтобы он мог создавать объекты без конкуренции).

*Minor GC* – сборщик мусора, ответственный за его сборку в молодом поколении, является mark and copy сборщиком, т.е. помечает живые объекты и копирует их в новую область памяти, при этом помечает каждый живой объект числом, показывающим кол-во сборок, которые пережил объект (при достижении 15 (вроде) объект перемещается в Old gen).

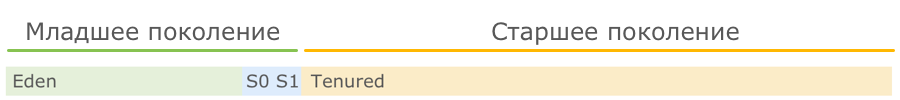
*\**Исключение: объект сразу помещается в Old gen, если он слишком большой и его не выгодно переносить множество раз.

Young generation включает в себя 3 региона: Eden, Survivor 0 и Survivor 1 зоны.  
Eden содержит все новоаллоцированные объекты, как только область заполняется инициируется процесс сборки мусора и живые объекты перемещаются в S0, после следующего заполнения Eden инициируется сборка в нем и S0 и живые объекты попадают в S1, после нескольких таких циклов (достижения числа пройденных сборок) объект перемещается в Old gen.

Old generation (Tenured gen) – область памяти, предназначенная для хранения объектов старого поколения (объектов, которые пережили определенное кол-во сборок мусора).

*Major GC –* сборщик мусора в Old gen или в обоих поколениях, инициируется при заполнении Old gen, может использовать алгоритм mark-and-sweet (маркирует живой объект и оставляет его на своем месте в куче) или mark-and-compact (маркирует живые объекты и дефрагментирует память). Запускается реже, чем minor и требует больше времени на исполнение.

*Пропорции Heap:*



***Живой объект:*** Живым объектом считается достижимый объект, т.е. объект, до которого возможно «дотянуться» через явно живые объекты (gc root) – это статические члены классов, ссылки из стеков всех потоков, ссылки из native stack method, для young gen живыми объектами считаются объекты из Old gen.

[**Виды GC:**](https://habr.com/ru/articles/269621/)

1. Serial GC – базовый сборщик мусора, использует один поток выполнения, при запуске всегда инициирует SWT (stop the world), т.е. останавливает все потоки выполнения. Простой сборщик без оверхеда, но с заметными паузами.

2. Parallel GC - параллельный сборщик мусора, использующий несколько потоков сборки. Уменьшает SWT за счет увеличения кол-ва потоков сборки, но увеличивается дефрагментация из-за того, что каждый поток GC получает свой участок в Old gen для работы. Также умеет автоматически подстраиваться под заданные параметры производительности.

3. CMS (Concurrent Mark Sweep) GC - сборщик мусора, который позволяет выполнять сборку мусора параллельно с работой приложения. Старшая и младшая сборки разделены (при выполнении Major не запускается minor), minor сборка работает также как предыдущая (с STW), major сборка происходит в конкурентном режиме Mark-and-sweep (в фоновом режиме помечает живые объекты не перемещая), если сборщик не справляется, то уходит в STW и отрабатывает в Mark and compact.

4. G1 (Garbage First) GC – c Java 9 - отличается организацией кучи: куча разделена на мелкие регионы одинакового размера, которые динамически распределены между молодым и старым поколением (при эвакуации объектов любой регион может поменять свое назначение). Помечает объекты конкурентно, не умеет эвакуировать конкурентно.

5. Epsilon GC – не собирает мусор, завершает работу приложения, в случае, если невозможно выделить потоку новый буффер для аллокации объектов. (Для короткоживущих приложений или приложений, которые аллоцируют все необходимые для себя объекты при старте)

6/7. ZGC (Oracle)/Shenandoah GC (RedHat) – регионализованные конкурентные сборщики, пометка и эвакуация объектов производится параллельно.

Материал про [Shenandoah](https://www.youtube.com/watch?v=CnRtbtis79U)