Java Platform

---

1. JRE - назначение, состав

Java Runtime Enviroment (JRE) – минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Состоит из Java Virtual Machine и библиотеки Java-классов.

Java Virtual Machine (JVM) – виртуальная машина Java, является основной частью **JRE**. **JVM** интерпретирует байт-код Java, предварительно созданный из исходного текста Java-программы компилятором (javac). **JVM** может также использоваться для выполнения программ, написанных на других языках программирования (Kotlin, Scala и т.п.).

---

2. JDK - назначение, состав

Java Development Kit – по сути является комплектом разработчика приложений на языке Java, включает в себя компилятор Java (javac), стандартные библиотеки классов Java, примеры, документацию, утилиты и **JRE**.

Java Runtime Enviroment (JRE) – минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Состоит из Java Virtual Machine и библиотеки Java-классов.

Java Virtual Machine (JVM) – виртуальная машина Java, является основной частью **JRE**. **JVM** интерпретирует байт-код Java, предварительно созданный из исходного текста Java-программы компилятором (javac). **JVM** может также использоваться для выполнения программ, написанных на других языках программирования (Kotlin, Scala и т.п.).

---

3. Типы памяти в Java

В связи с тем, что Java постоянно развивается, на данный момент существует ряд противоречий в терминологии относительно типов памяти в Java. Например, часть ресурсов термины non-heap и stack приравнивают друг к другу. В своем ответе я разделяю память в Java на три области: **stack**, **heap**, **non-heap**.

Stack (Стек) (5)

- имеет небольшой размер в сравнении с кучей, размер стека ограничен, его переполнение может привести к возникновению исключительной ситуации;
- содержит только локальные переменные примитивных типов и ссылки на объекты в куче;
- стековая память может быть использована только одним потоком, т.е. каждый поток обладает своим стеком;
- работает по принципу LIFO(last-in-first-out), благодаря чему работает быстро;
- когда в методе объявляется новая переменная, она добавляется в стек, когда переменная пропадает из области видимости, она автоматически удаляется из стека, а эта область памяти становится доступной для других стековых переменных.

Heap (Куча) (4)

- куча имеет больший размер памяти, чем стек (stack);
- используется для выделения памяти под объекты, объекты хранятся в куче;
- объекты кучи доступны разным потокам;
- в куче работает сборщик мусора: освобождает память, удаляя объекты, на которые нет ссылок.

Non-heap делится на 2 части:

- **Permanent Generation** – это пул памяти, который содержит интернированные строки, информацию о метаданных, классах, это пространство зарезервировано для классов, статических данных и т.п.
  Metaspace – замена **Permanent Generation** в Java 8. Основное различие в том, что **Metaspace** может динамически расширять свой размер во время выполнения. Размер **Metaspace** по умолчанию не ограничен.
- **Code Cache** – используется JVM при JIT-компиляции, здесь кэшируется скомпилированный код.

---

4. Компиляция и запуск в консольном режиме

Сначала нужно скомпилировать класс, то есть перевести его в байт-код. Для этого в состав JDK входит компилятор Java кода javac:

E:\EJC>javac -sourcepath src -d out src/tasks/task_01/Main.java

-sourcepath задает каталог исходного кода, что позволяет компилятору найти не только тот класс, который мы отправили на компиляцию, но и найти в каталоге исходного кода другие классы, в т.ч. классы из других пакетов, которые используются в классе, отправленном на компиляцию.

С помощью **-d** мы отметили директорию, в которой будут лежать скомпилированные классы. Именно в **out** компилятор автоматически создает структуру каталогов идентичную структуре каталогов в **src**. Далее компилятор находит класс, который мы отправили на компиляцию **Main.java**, и компилируется он и все классы, которые он использует, в байткод в соответствующие каталоги в каталоге **out**.

Теперь мы можем запустить нашу программу с помощью java:

E:\EJC>java -cp out tasks.task_01.Main

С помощью **-cp** (или **-classpath**) указываем из какого каталога мы загружаем файлы, указываем класс, который запускаем.

Чтобы не прописывать полный путь к **java** и **javac** в консоли, можно задать переменную среды окружения. Если этого не делать, то придется полностью прописывать путь размещения **javac** и **java**.

---

5. Java Heap: принципы работы, структура

В связи с тем, что Java постоянно развивается, на данный момент существует ряд противоречий в терминологии относительно типов памяти в Java. Например, часть ресурсов термины non-heap и stack приравнивают друг к другу. В своем ответе я разделяю память в Java на три области: **stack**, **heap**, **non-heap**. Принцип работы памяти **heap** я описываю на примере применения одного из сборщиков мусора HotSpot VM – Serial GC. Есть и другие сборщики мусора, принцип работы которых отличается в той или иной мере.

Heap (Куча) (4)

- куча имеет больший размер памяти, чем стек (stack);
- используется для выделения памяти под объекты, объекты хранятся в куче;
- объекты кучи доступны разным потокам;
- в куче работает сборщик мусора.

Heap делится на 2 части:

- **Young Generation Space тоже делится на 2 части:**
  - **Eden Space** – область памяти, в которую попадают только созданные объекты, после сборки мусора эта область памяти должна освободиться полностью, а выжившие объекты перемещаются в Survivor Space
  - **Survivor Space** – область памяти, которая обычно подразделяется на две подчасти («from-space» и «to-space»), между которыми объекты перемещаются по следующему принципу:

    - «from-space» постепенно заполняется объектами из Eden после сборки мусора;
    - возникает необходимость собрать мусор в «from-space»;
    - работа приложения приостанавливается и запускается сборщик мусора;
    - все живые объекты «from-space» копируются в «to-space»;
    - после этого «from-space» полностью очищается;
    - «from-space» и «to-space» меняются местами («to-space» становится «from-space» и наоборот);
    - все объекты, пережившие определенное количество перемещений между двумя частями Survivor Space перемещаются в Old Generation;
- **Old Generation Space** – область памяти, в которую попадают долгоживущие объекты, пережившие определенное количество сборок мусора. Работает по принципу перемещения живых объектов к началу «old generation space», таким образом мусор остается в конце (мусор не очищается, поверх него записываются новые объекты), имеется указатель на последний живой объект, для дальнейшей аллокации памяти указатель просто сдвигается к концу «old generation space»;

Non-heap делится на 2 части:

- **Permanent Generation** – это пул памяти, который содержит интернированные строки, информацию о метаданных, классах, это пространство зарезервировано для классов, статических данных и т.п. **Metaspace** – замена **Permanent Generation** в Java 8. Основное различие в том, что **Metaspace** может динамически расширять свой размер во время выполнения. Размер **Metaspace** по умолчанию не ограничен.
- **Code Cache** – используется JVM при JIT-компиляции, здесь кэшируется скомпилированный код.

---

6. Представление о JIT

JIT (Just-In-Time) компиляция – динамическая компиляция, технология увеличения производительности посредством компиляции байт-кода в машинный код во время выполнения программы с применением различных оптимизаций. Благодаря этому достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом. **Основная цель JIT** – приблизиться в производительности к языкам со статической компиляцией (C/C++). С появлением **JIT** Java стала несколько более производительной.

Пример оптимизации **JIT**:

Некий метод исполняется очень часто, поэтому этот метод передается в **JIT**, он компилируется, все последующие вызовы этого метода будет кэшироваться и вызываться без перекомпиляции.

JIT позволяет достигать более высокой скорости выполнения, сохраняя одно из главных преимуществ Java-языка – переносимость. Несмотря на то, что языки со статической компиляцией все равно выигрывают в скорости, статическая компиляция лишает эти языки той переносимости, которая есть в Java.

---

7. Entry point в Java классе: назначение, структура (точка входа, метод main)

Каждому приложению требуется точка входа, чтобы Java знала, откуда начать выполнение кода. В Java такой точкой входа является метод main(), который имеет следующую сигнатуру:

public static void main(String[] args)
public static void main(String... args)

По крайней мере один класс в программе должен иметь метод main(). В программе может быть несколько точек входа, но начинается выполнение кода всегда с какой-то одной. Аргументы командной строки передаются в метод **main()** в массив строк **args** (называться он может иначе, это не принципиально), эти аргументы могут быть каким-то образом использованы в методе **main()**.

В методе **main()** обычно описывают самый основной алгоритм работы, например, создание каких-то объектов и вызов их методов, но конкретная программная реализация подсчетов и прочего обычно находится вне этого метода, скорее этот метод лучше использовать для консолидации функциональности других классов, которые созданы для решения определенных задач.

---

8. Распространение Java программ: состав, инструменты создания и способы использования jar

Для хранения классов языка Java, связанных с ними ресурсов и конфигурационных файлов в языке Java используются сжатые архивные **jar-файлы**.

Полноценно работать с технологией **JAR** можно как с помощью утилиты **jar**, входящей в состав JDK, так и с использованием классов **JAR API**. Для работы с архивами в спецификации Java есть два пакета – **java.util.zip** и **java.util.jar** соответственно для архивов **zip** и **jar**. Различие форматов заключается только в расширении архива zip.

Java-приложения гораздо удобнее собирать в один **jar-файл**, нежели хранить сложную структуру из нескольких тысяч файлов.

Такой подход позволяет **(4)**:

- подписывать содержимое jar-файла, повышая таким образом уровень безопасности;
- сокращать объем приложения;
- упростить процесс создания библиотеки;
- осуществлять контроль версий.

Так же существуют системы и утилиты для автоматизации сборки программных продуктов:

- **Apache Ant**
- **Apache Maven**
- **Gradle**