Урок 2

JVM

Расширенное понимание видов JVM, опций и возможностей

Что такое JVM

- Различия между JVM

Опции JVM

- Что такое JVRM
- Запуск приложения с параметрами в командной строке.

Практика

- Доделываем ядро серверной части
- Обработка входящих данных
- Отправка исходящих данных по результату входящих

Домашнее задание

Дополнительные материалы

Используемая литература

Что такое JVM

JVM (Java virtual machine) - специальная виртуальная машина, из-под которой запускаются приложения, написанные на языке Java. Это тот самый инструмент, обеспечивающий выполнение Java-приложений на практически любом компьютере, так как задачей программиста является создание программ, а за выполнение на различных операционных системах отвечает компания-разработчик Oracle.

Различия между JVM

Из заголовка вполне можно догадаться, что таких понятий, как JVM на самом деле великое множество.

Список всех наиболее используемых виртуальных машин:

- Azul VM — сегментированная Java Virtual Machine, основанная на оригинальной микропроцессорной архитектуре, оптимизированной для работы с Java. Возможно использование до 54 микропроцессоров и до терабайт памяти без накладных расходов на сборку мусора.
- CEE-J реализация Java-технологии с нуля без лицензирования от Sun.
- Excelsior JET (с компилятором AOT).
- Hewlett-Packard, Java для HP-UX, OpenVMS, Tru64 и Reliant (Tandem) UNIX-платформ
- J9 (IBM), для AIX, Linux, Windows, MVS, OS/400, Pocket PC, z/OS.
- Apogee предоставляет встроенную Java, использующую IBM J9 и библиотеку классов Apache Harmony для X86/ARM/MIPS/PowerPC, работающих под Linux/LynxOS/WinCE.
- Jbed, (Esmertec) — Java VM с поддержкой реального времени для встроенных систем и программно-аппаратных комплексов, работающих с Интернет[1]
- JamaicaVM, (aicas) — Java VM с поддержкой приложений реального времени. Предназначена для встроенных систем.
- JBlend, (Aplix) реализация Java ME.
- JRockit (изначально разрабатываемая BEA Systems) приобретена корпорацией Oracle для Linux, Windows и Solaris.
- Mac OS Runtime for Java (MRJ).
- MicroJvm (IS2T — Industrial Smart Software Technology) Широкий спектр виртуальных машин, предназначенных для встроенных систем (в том числе систем жёсткого реального времени), ARM7, ARM9, AVR, AVR32, PPC, MIPS, …
- Microsoft Java Virtual Machine (поддержка прекращена в 2001 году).
- OJVM (иногда также «JServer») от Oracle Corporation.
- PERC (Aonix/Atego) Java реального времени для встраиваемых систем.
- SAPJVM (SAP) лицензированная у Sun и модифицированная Sun JVM, портированная на платформы, поддерживаемые ПО SAP NetWeaver. Поддерживает Java 5 и частично Java 6 compatible (Windows i386, x64, IA64, Linux x86, IA64, PPC, AIX PPC, HP-UX Sparc/IA64, Solaris Sparc/x86_64, i5/OS PPC)

Данные виртуальные машины предназначены не только для настольных ПК, но и для различных устройств, поддерживающих Java Mobile Edition, в том числе микроволновые печи, мобильные телефоны и прочая умная электроника. На настольных ПК самая распространённая JVM - HotSpot VM, поставляемая компанией Oracle.

Опции JVM

Что такое JVRM

JVMR (Java Virtual Machine Runtime) – это основа работы Java-приложений, которая отвечает за большой спектр задач: когда запускается JVM, именно JVMR обрабатывает параметры терминала или командной строки, загружает классы, отвечает за жизненный цикл виртуальной машины, работает с just-in-time компилятором, обрабаывает ошибки, потоки и отвечает за их синхронизацию, а также многое другое, о чём пойдет речь в этом уроке.

Запуск приложения с параметрами в командной строке.

Для начала давайте познакомимся с параметрами командной строки. Так же, как и большинство приложений, виртуальную машину можно запустить с набором определённых предпусковых параметров. Шаблон заполнения параметров имеет вид: «-[Options]»

Всего бывает три вида параметров: стандартные, нестандартные и продвинутые.

К стандартным можно отнести следующее действие: к примеру, нам требуется вывести версию используемой виртуальной машины, для этого в командной строке потребуется ввести команду **java –version.**

К нестандартным видам относятся параметры, которые могут не поддерживаться на следующих поколениях виртуальных машин и имеют шаблон: «-X[Options]». К примеру команда **java –XshowSettings:all.**

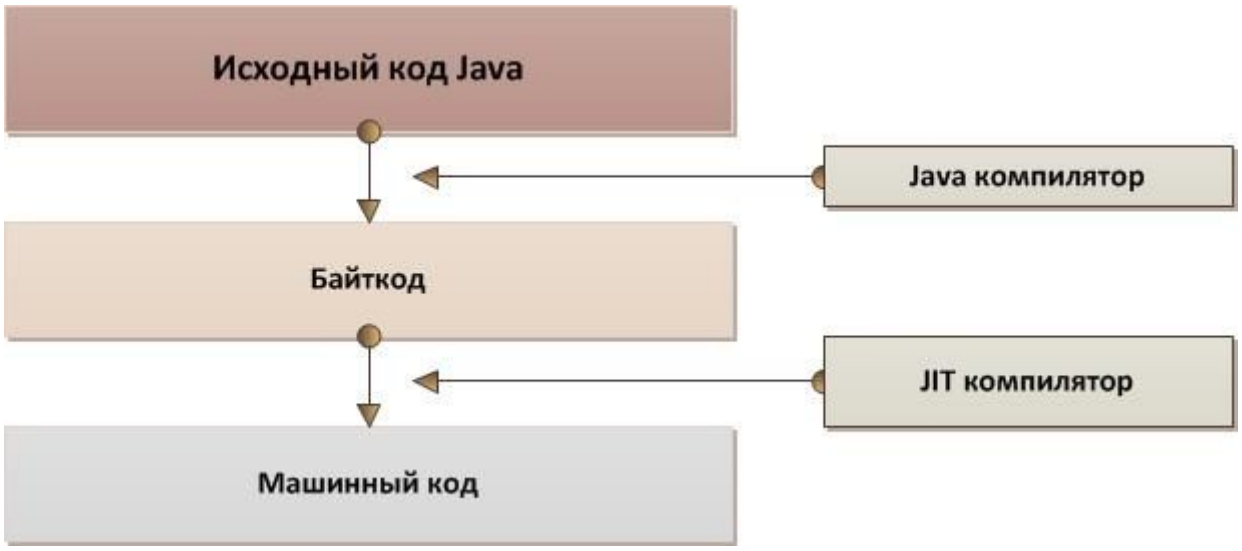
Продвинутые опции задаются шаблоном «-XX:[Options]», могут быть указаны с двумя специальными значениями: «+» или «-» для включения или отключения опции, либо со знаком «=» для указания конкретного значения. Например, для включения трассировки загрузчика классов в виртуальной машине, требуется при запуске указать опцию **java –XX:+TraceClassLoading** или установить параметр размера кучи размером 256Mb опцией **java –XX:MaxHeapSize=256m.** Для примера напишем программу и понаблюдаем поведение виртуальной машины при установке различных опций.

```
public class Test {
        public Test() {
    System.out.println("Hello Java!");
        }
        public static void main(String args[]) {
        new Test();
        }
}
```

Указанный тестовый класс имеет самый простейший вид и просто выводит сообщение “Hello, Java!” в консоль. Компилировать класс можно с помощью различных IDE, так и вручную. Для ручной компиляции требуется пройти в каталог с написанным классом Test.java и в этой же директории запустить команду **javac Test.java**. После компиляции в каталоге появится откомпилированный файл Test.class, который и потребуется запустить командой **java –cp ”.” Test**.

Результат выполнения будет надпись “Hello, Java!” в консоли.

Организация данного процесса сложна. Требуется сначала откомпилировать класс в байт-код, который впоследствии будет запускаться интерпретатором JIT, который будет переводить всё написанное в машинный низкоуровневый код, после чего программа выполнится, и в консоль будет выведен результат выполнения. Зачем столько промежуточных действий для запуска небольшой программы? Ответом на данный вопрос станет один очень весомый факт: кроссплатформенность.

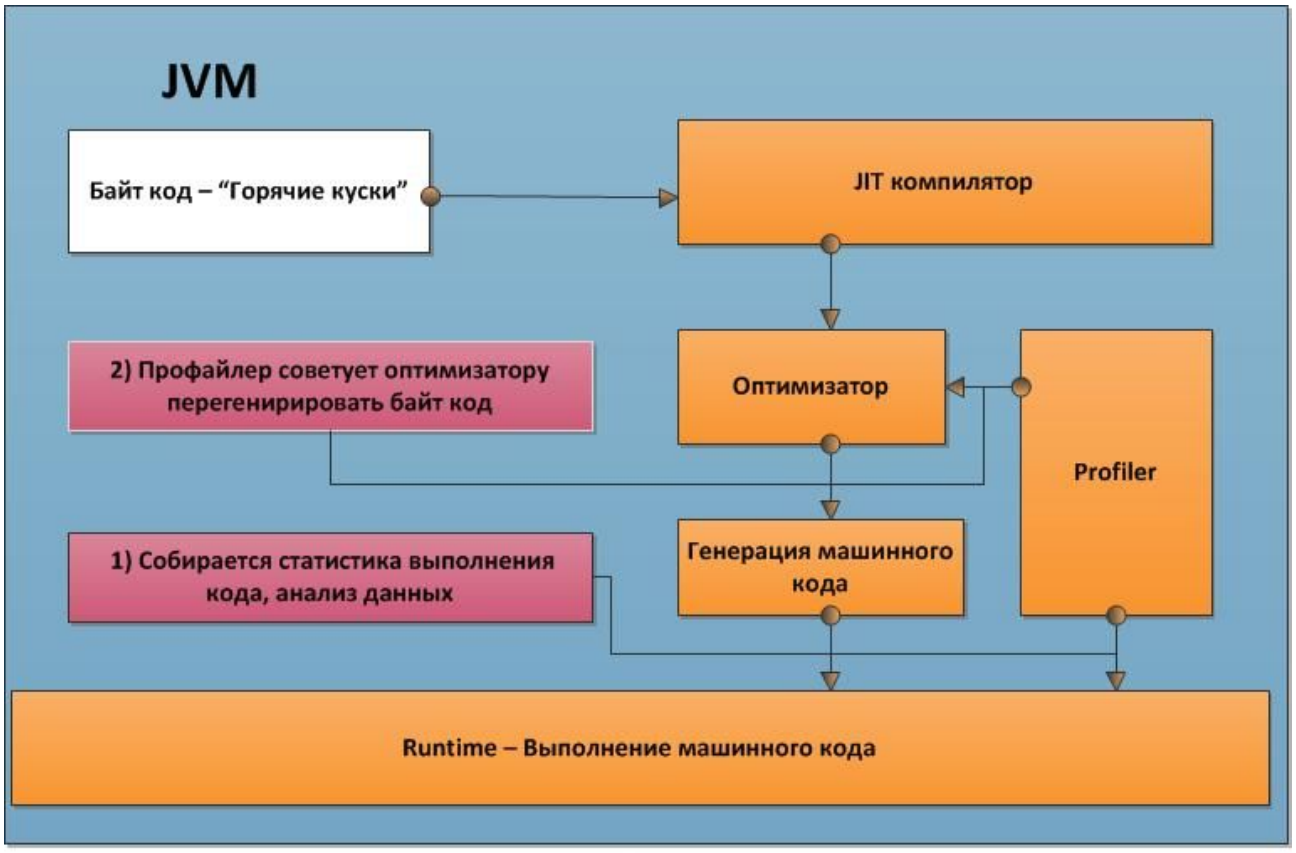


Конечно, при выполнении стольких операций для получения определённого результата производительность снижается, так как машинный код, переведённый интерпретатором, работает намного медленнее. Но правильно сконфигурированная программа под конкретную архитектуру будет вести себя практически без отличий от нативно разработанных приложений. Достаточно правильно выставить конфигурацию среды выполнения и провал в скорости снизится. JVM действительно работает на всех операционных системах, однако, устанавливая её, нужно понимать, что все настройки выставлены по умолчанию и для ускорения производительности нужно просто правильно выстроить архитектуру.

На самом деле JVM несёт в себе много дополнительного материала, которое не обязательно должно работать при любом запуске JVM. Для примера хочу привести для учеников операционную систему Linux. При установке её из исходных кодов эта операционная система очень весома и также может нести модули, которые в данный момент не нужны. После удаления этих модулей и повторной сборке ядра Linux операционная система работает намного быстрее, так как больше не обременена ненужными для пользователя вещами. JVM имеет похожую архитектуру. Если замерять тест производительности одного и того же написанного приложения после нескольких компиляций, то можно заметить одну особенность: приложение с каждым разом работает быстрее. Это результат особенности работы JVM HotSpot: при каждом выполнении приложение анализируется и из него убираются ненужные блоки кода, собирается статистика, код перестраивается и снова перекомпилируется уже с учётом новой информации. При всех этих этапах. При перестроении затрачиваются ресурсы, требуется запустить виртуальную машину, после чего ещё нужно работать с интерпретатором. В следствие чего в современной JVM имеются два различных JIT-интерпретатора, которые можно запустить с дополнительными параметрами клиента (-client) или сервера (-server).

Чуть ранее была разработана JVM, которая выполнялась только в роли интерпретатора. На тот момент от такого инструмента разработчикам приходилось отказываться в силу низкой скорости выполнения программ. После был разработан jit-компилятор, задачей которого стало преобразование байт-кода в машинные инструкции, при этом, исходя из названия, компиляция происходила только тех кусков кода, которые требовались в данный момент. Такой принцип работы позволяет запустить программу перед всей компиляцией кода в виртуальной машине. Таким образом JVM начала поставляться с двумя инструментами: интерпретатором и компилятором. Часть кода. выполняемая чаще всего теперь компилируются, но более изменяемая часть программы интерпретируется. За

определение частых и нечастых кусков кода отвечает оптимизатор профилировщика, который выстраивает статистику и оптимизирует программу. Собственно название виртуальной машины HotSpot произошло от названия «горячего места». В итоге не затрачиваются ресурсы для компиляции «горячего кода», тем самым оптимизируется работа интерпретатора.



Виртуальная машина, запущенная с параметром –server, имеет компилятор, оптимизированный для повышения скорости работы и предназначенный для протяженных по времени серверных приложений. Требует большое количество ресурсов для выполнения программы. Виртуальная машина, запущенная с параметром –client имеет компилятор, который оптимизирован для уменьшения времени начального запуска приложения и занимаемого объема памяти, реализует менее сложную оптимизацию, чем серверный компилятор, и, следовательно, требует меньше времени для компиляции, используется в основном для клиентских программ. В будущем мы попробуем запускать наше курсовое приложение именно с этими параметрами для оптимизации работы.

Также следует отметить ещё один параметр виртуальной машины **-XX:+TieredCompilation** - это гибридный параметр, который берёт в себя всё лучшее из двух различных виртуальных машин, быстроту запуска, от клиентской и оптимизацию от серверной. Этот параметр запускается парой с параметром –server -XX:+TieredCompilation, он по умолчанию включён у 64-битных серверных виртуальных машин, подходит для запуска, как клиентских приложений, так и серверных. Итак, попробуем запустить виртуальные машины с параметром –version, который показывает версию виртуальной машины:

```
>java -version
java version "1.8.0_58"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

В данный момент запущено две виртуальные машины на различных компьютерах с различными ОС. Исходя из полученных данных можно проанализировать информацию в выводе. При запуске операционной системы по умолчанию, в JVM предустановлена опция запуска, т.е. виртуальная машина может запуститься либо в режиме клиента, либо в режиме сервера. Посмотреть параметры, установленные в опцию «по умолчанию» можно, если запустить команду:

-**XX:+PrintCommandLineFlags**

При запуске приложения, написанного выше, командой **java –cp Test**, виртуальная машина выведет параметры запуска:

```
-XX:InitialHeapSize=133309505
-XX:MaxHeapSize=2132952128
-XX:+PrintCommandLineFlags
-XX:-UseCompressedClassPointers
-XX:-UseCompressedOops
-XX:-UseLargePagesIndividualAllocation
-XX:-UseParallelGC
Hello JVM!
```

В данном случае виртуальная машина была запущена с параметрами по умолчанию.

К сожалению, на каждой операционной системе имеются ограничения по запуску. К примеру, на Windows XP возможно запустить только клиентский вариант JVM, однако на Windows 7 можно запустить только серверную опцию. Однако для некоторых случаев можно запустить опцию mixed mode.

Mixed Mode – опция JIT-компилятора, она устанавливается нестандартным видом опции по умолчанию (**-Xmixed**). Данная опция выполняет весь байт-код в режиме интерпретатора, кроме тех самых «горячих блоков», которые компилируются в машинные инструкции. Но помимо этой опции есть также опция **–Xint**, при которой виртуальная машина запускается только в режиме интерпретатора, т.е. никакой компиляции «горячих блоков» в этом режиме не происходит.

-Xcomp – режим, при котором JIT-компилятор компилирует код при первых вызовах методов. По умолчанию байт-код компилируется в машинный код только после многоразовой интерпретации. Это означает, что компилируется код не сразу, а после какого-то размера «обкатки» кода. Обкатка кода у разных режимов установлена в разное количество раз. К примеру, в режиме сервера компиляция достигается только при прохождении интерпретации 10000 раз, а для режима клиента всего 1000 раз. Однако этот параметр также можно изменить командой:

```
-XX:CompileThreshold = [Value]
```

В значение Value этой команды можно подставить любое число, и это будет обозначать, что после обкатки метода указанное количество раз, метод будет скомпилирован в машинный код.

И, наконец, опция, которая покажет нам все остальные опции, которые включены у виртуальной машины:

```
-XX:+PrintFlagsFinal
```

При выполнении данной программы, можно узнать размер “обкатки” кода:

```
intx CompileThreshold = 10000
```

Для наглядности большинство кода опущено, так как виртуальная машина выдаёт около 600-800 параметров, которые также можно изменять, но обычно для оптимизации достаточно изменить несколько десятков, после чего проводят анализ производительности и таким путём приходят к оптимальной работе виртуальной машины.

Практика

Доделываем ядро серверной части

В прошлом уроке мы сделали основной каркас нашего курсового приложения.

Настало время разобраться с логикой работы нашего сервера. Задача прошлого домашнего задания стояла в организации каркаса сервера, который повторяет архитектуру клиентской части. В сегодняшнем уроке мы напишем сервер, который будет принимать строку и обрабатывать её в отдельном потоке.

Схема потоков для нашего приложения следующая: сервер ждёт входящего подключения. Если на сервер приходит запрос на подключение - создаётся новый поток, в котором начинается работа с клиентом.

Для нашего серверного приложения потоков должно быть больше 2-х. Так как 1-ый поток - это главный поток, остальные ответвляются от главного потока и выполняются сами в себе при

подключении нового клиента. Соответственно, для этих целей нам требуется реализовать интерфейс Runnable для получения и отправки данных от одного клиента.

Класс назовём FileThread:

```
public class FilesThread implements Runnable {
    public Socket socket;
    public FilesThread(Socket socket){
        this.socket = socket;
    }
    public void run() {

     }
}
```

В написанном классе у нас реализуется интерфейс Runnable и, соответственно, имплементится метод run(). В методе run() нам требуется реализовать функционал по обмену данными между клиентом и сервером. Прежде всего давайте определим, какой именно способ получения данных нам нужен? Java имеет в себе огромный выбор всевозможных способов передачи данных через создание потоков между клиентом и сервером. Я предлагаю выбрать ObjectInputStream. Мой выбор обусловлен тем, что у нас имеется огромный набор типов, которые мы можем передать нашему серверу. В одном и том же потоке нам потребуется передать экземпляр строки (String) а также экземпляр некоего файла из пакета java.io (File). Если использовать передачу через поток наиболее “обобщённых” объектов, в последующем, при желании, возможно реализовать шифрование, передачу частично, p2p и многое другое.

```
public class FilesThread implements Runnable {
    public Socket socket;
    public FilesThread(Socket socket, List<File> files){
        this.socket = socket;
    }
    public void run() {
    InputStream in;
    OutputStream os;
    try {
    in = socket.getInputStream();
    os = socket.getOutputStream();
    ObjectInputStream inputObject = new ObjectInputStream(in);
    ObjectOutputStream outputStream = new ObjectOutputStream(os);
    } catch (Exception e){

    }
      }
}
```

Объявляя потоки, мы не забываем про то, какие объекты может принимать сервер. Для того чтобы понимать, какие именно данные нам приходят в поток ObjectInputStream, мы будем инициализировать пустой объект и записывать в него пришедшие данные. В последующем мы будем проверять функцией instanceof принадлежность данных к определённому типу для того, чтобы правильно обработать входной запрос.

Обработка входящих данных

Для начала принимаемые данные должны быть присвоены пустому объекту для того, чтобы при чтении из потока и передачи основной логике приложения у нас не выпадал EOFException при неожиданном конце входящих данных.

```
public class FilesThread implements Runnable {
    public Socket socket;
    public FilesThread(Socket socket){
        this.socket = socket;
    }
    public void run() {
InputStream in;
OutputStream os;
try {
   in = socket.getInputStream();
   os = socket.getOutputStream();
   ObjectInputStream inputObject = new ObjectInputStream(in);
   ObjectOutputStream outputStream = new ObjectOutputStream(os);
   while(true){
       Object request = new Object();
       while (true){
           request = inputObject.readObject();
       }
   }
} catch (Exception e){
}
     }
}
```

После инициализации мы ждём входные данные. Сервер не продолжит работу, если при подключении сокета клент не отправит ему определённы набор данных. Когда это произошло, в текущем случае сервер ждёт новых данных, так как старые он уже принял. На этом моменте нам нужно определиться, какого типа данные мы хотим отправлять на сервер, и при получении каких данных это должно работать.

Отправка исходящих данных по результату входящих

Для работы приложения по замещению данных мы будем отправлять для начала с клиента строку типа “get”, которую сервер должен принять и отправить файлы, которые у него имеются. При старте приложения требуется синхронизировать данные на сервере с клиентом, поэтому мы будем отправлять те данные, которые имеются на сервере.

```
import java.io.*;
import java.net.Socket;
import java.util.List;
public class FilesThread implements Runnable {
    public Socket socket;
    public List<File> files;
    public FilesThread(Socket socket, List<File> files){
        this.socket = socket;
        this.files = files;
    }
    public void run() {
        InputStream in;
        OutputStream os;
        try {
            in = socket.getInputStream();
            os = socket.getOutputStream();
            ObjectInputStream inputObject = new ObjectInputStream(in);
            ObjectOutputStream outputStream = new ObjectOutputStream(os);
            while(true){
                Object request = new Object();
                while (true){
                    request = inputObject.readObject();
                    if (request instanceof File){
                        File requestFile = (File) request;
                        System.out.println("Получен файл "+requestFile.getName());
                        if (files.contains(requestFile)){
                            files.remove(requestFile);
                            System.out.println("Количество объектов после удаления "+files.size());
                        }
                        else {
                            files.add(requestFile);
                            System.out.println("Количество объектов после добавления "+files.size());
                        }
                    }
                    if (request instanceof String){
                        String question = request.toString();
                        if (question.equals("get")){
                            System.out.println(" Получена команда get");
                            outputStream.writeObject(files);
                            outputStream.flush();
                            System.out.println("Файлы отправлены клиенту!");
                        }
                    }
                }
            }
        } catch (Exception e){
        }
    }
}
```

При получении данных мы сравниваем их на схожесть с определённым классом. Если это строка - мы обрабатываем её как строку, смотрим, является ли строка значением “get” и, в случае успеха, отправляем все имеющиеся файлы одним списком обратно клиенту. Если это файлы, мы смотрим в список со всеми загруженными файлами на сервере, если он имеется в списке, этот файл мы удаляем, если его нет - добавляем. Таким образом у нас сформировался небольшой сервер, в котором будут получаться данные для удаления или добавления в список.

Домашнее задание

1. Класс Fail имеет методы для создания директорий. Написать алгоритм для создания директории, в которой будут храниться файлы из списка на сервере.
2. Написать небольшой клиент для отправки строковых значений.
3. *Дописать обработку строк, в которых внедрён логин и пароль, сверять их с теми, что есть на сервере, и отправлять назад строку об успешности, к примеру: “Поздравляем! Вы залогинились!”

Дополнительные материалы

1. Java Virtual Machine - Статья на Википедии
2. The Java® Virtual Machine Specification
3. JVM на Stack Overflow

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

4. Java Virtual Machine - Статья на Википедии
5. The Java® Virtual Machine Specification
6. JVM на Stack Overflow
7. Опции JVM - статья на Хабрахабр
8. Брюс Эккель - Философия Java