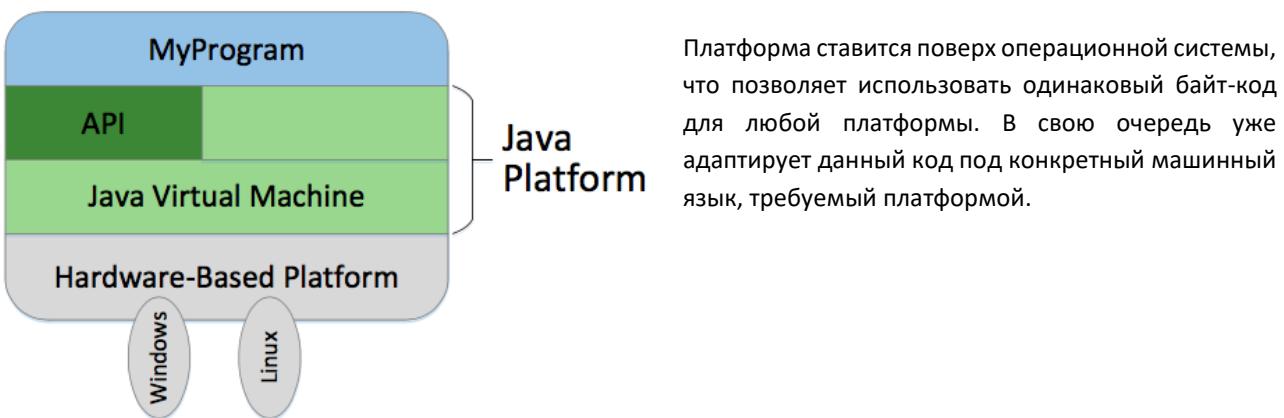


# LESSON 1 (JDK, JRE, JVM)

## WHAT WE NEED FOR JAVA DEV (INSTRUMENTS)

Java платформа – комплект программ, которые позволяют разрабатывать и выполнять прикладного программного обеспечения, написанные на языке Java. Она содержит исполняющий механизм (JVM), компилятор и набор библиотек. В нее также могут входить ряд дополнительных серверов и альтернативные библиотеки в зависимости от конкретных требований. Java не зависит от типа процессора и от операционной системы. Java платформа гарантирует одинаковое выполнение программных компонентов на любой ОС.



Виды Java платформ:

- **Java Platform, Standard Edition** — стандартная версия платформы Java 2, предназначенная для создания и исполнения апплетов и приложений, рассчитанных на индивидуальное пользование или на использование в масштабах малого предприятия.
- **Java Platform, Enterprise Edition** — Java SE, с добавлением специального набора спецификаций и соответствующей документации для языка Java, описывающих архитектуру серверной платформы (используется для написания мульти клиент-серверных приложений) для задач средних и крупных предприятий.
- **Java Platform, Micro Edition** — версия платформы Java, предназначенная для устройств, ограниченных в ресурсах (память, мощность, дисплей), например: мобильные телефоны, проигрыватели дисков Blu-ray, принтеры.

Составляющие Java платформы:

- Java компилятор (конвертирует исходных код .java в байт-код(промежуточный язык для JVM) .class). Компилятор является частью JDK.
- JIT (just-in-time) компилятор (интерпретатор), конвертирует промежуточный байт-код в нативный/родной машинный код на лету. Данный JIT компилятор содержит JRE, являющийся дополнением JDK.
- Расширяемый набор библиотек.

Для того, чтобы разрабатывать приложения (писать программы) на java, необходим JDK (Java Development Kit), т.е. комплект инструментальных средств разработки программ.

Обычно среда разработки включает в себя текстовый редактор, компилятор и/или интерпретатор, средства автоматизации сборки и отладчик. Иногда также содержит средства для интеграции с системами управления

версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя. Многие современные среды разработки также включают браузер классов, инспектор объектов и диаграмму иерархии классов — для использования при объектно-ориентированной разработке ПО.

## DIFFERENCE BETWEEN JDK & JRE

**JDK (Java Development Kit)** – программный инструментарий (набор) для полноценной работы с языком, который, наряду с компилятором, интерпретатором, отладчиком и другими инструментами включает в себя обширную библиотеку классов Java. Набор программ и классов JDK в основном содержит:

- компилятор javac из исходного кода в байт-коды;
- интерпретатор java, содержащий интерпретацию JVM (java virtual machine);
- облегченный интерпретатор jre;
- программу просмотра апплетов appletviewer;
- отладчик jdb;
- дисассемблер javap;
- программу архивации и сжатия jar;
- программу сбора документации javadoc;
- программу javah генерации заголовочных файлов языка Си;
- программу javakey добавления электронной подписи;
- программу native2ascii, преобразующую бинарники в текстовые файлы;
- программы rmic и rmiregistry для работы с удаленными объектами;
- программу serialver, определяющую номер версии класса;
- библиотеки и заголовочные файлы “родных” методов;
- библиотеку классов Java API.

**JRE (Java Runtime Environment)** - среда исполнения java-приложений, которая должна быть установлена на компьютере для запуска Java приложений. JDK включает в себя JRE.

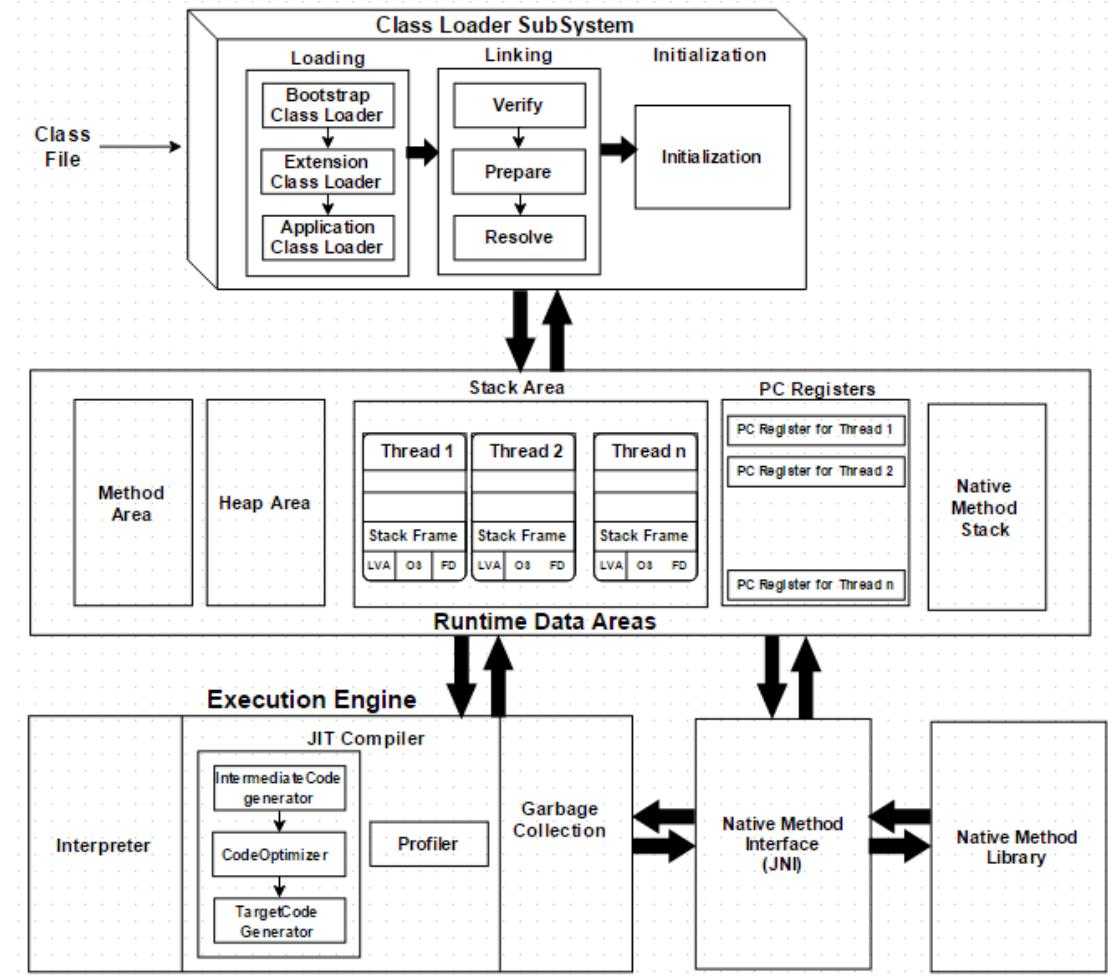
JDK представляет собой комплект инструментальных средств разработки программ на Java. В то время, как JRE предназначен для конечных пользователей программ на Java, в следствие чего данный пакет включает в себя виртуальную машину, но не содержит компилятор.

## DIFFERENCE BETWEEN JRE & JVM

JRE выполняет байт-код. JRE является реализацией JVM, которая анализирует байт-код, интерпретирует код и выполняет его.

### Что такое JVM?

Виртуальная машина – программная реализация физической машины, которая была разработана следуя концепту **WORA** «Write Once Run Anywhere», которая выполняется на виртуальной машине. Компилятор компилирует java-файл в class-файл, который подается на вход JVM, которая в свою очередь загружает и выполняет (loads and executes) класс-файл.



## Как работает JVM в Java?

Как показано на картинке, JVM состоит из трех основных подсистем:

1. Class Loader Subsystem
2. Runtime Data Area
3. Execution Engine

### 1. Class Loader Subsystem

Данная подсистема обеспечивает динамическую загрузку Java классов. Во время выполнения(*HE* при компиляции) при первом обращении к классу она загружает, связывает и инициализирует класс (т.е. выполняет три основные функции **Loading**, **Linking** и **Initialization**).

#### 1.1. Loading

Данный компонент обеспечивает загрузку классов, используя загрузчики классов (*BootStrap ClassLoader*, *Extension ClassLoader*, *Application ClassLoader*). Все загрузчики классов во время загрузки класс-файлов следуют алгоритму делегирования иерархий (**Delegation Hierarchy Algorithm**).

- 1) **BootStrap ClassLoader** – загрузка классов из classpath начальной загрузки (bootstrap classpath), т.е. ничего, кроме *rt.jar*. Данный загрузчик имеет наивысший приоритет.
- 2) **Extension ClassLoader** – загрузка классов, которые расположены в **ext** папке (**jre\lib**).
- 3) **Application ClassLoader** – загрузка **Application Level Classpath**, пути, который указан в переменной окружения (Environment Variable) и пр.

#### 1.2. Linking

- 1) **Verify** – верификатор байт-кода, который проверяет правильность сгенерированного байт-кода и в случае неправильной генерации генерирует **verification error**.

- 2) **Prepare** – выделяет память для всех статических переменных и инициализирует их значениями по умолчанию.
- 3) **Resolve** – заменяет все ссылки на символическую память (symbolic memory references) на настоящие ссылки (original references) из Method Area.

### 1.3. Initialization

Конечная фаза загрузки классов, где все статические переменные принимают реальные значения и выполняются статические блоки.

## 2. Runtime Data Area

Данная подсистема делится на 5 крупных компонентов:

- 1) **Method Area** – хранит все данные уровня класса, включая статические переменные. Существует по одному Method Area для каждой JVM (является разделяемым ресурсом).
- 2) **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.
- 3) **Stack Area (LIFO)** – здесь создается для каждого потока свой отдельный стек выполнения (**runtime stack**). На каждый вызов метода создается запись в стековой памяти (**Stack Frame**). Все локально объявленные переменные и ссылки хранятся в стековой памяти. Stack Area является потокобезопасной, т.к. не является разделяемым ресурсом. Stack Frame подразделяется на несколько составляющих:
  1. **Local Variable Array** – хранит количество локальных переменных, обрабатываемых в методе и соответствующие им значения.
  2. **Operand Stack** – является рабочим пространством на (*runtime workspace*) для выполнения промежуточных действий.
  3. **Frame Data** – хранит все обозначения, соответствующие методам. В случае возникновения исключения блок catch block information will be maintained in the frame data.

Объем Stack памяти намного меньше объема памяти в Heap.

Итого: все методы хранятся в сетке и попадают туда при вызове. Переменные в методах также имеют стековую память. Если в методе создается объект, то он помещается в кучу, но его ссылка помещается в стек.

<b>Heap Area</b>	<b>Stack Area</b>
Используется всеми частями приложения.	Используется только одним потоком исполнения программы.
Всякий раз при создании объекта он хранится в куче, а ссылка на него в стеке.	Хранит только локальные переменные примитивных типов и ссылки на объекты в куче.
Объекты в куче доступны из любой точки программы.	Стековая память не может быть доступна для других потоков.
	Управление памятью: по схеме LIFO
Память в куче живет с самого начала и до конца работы программы.	Стековая память существует лишь какое-то время работы программы, во время которого выполняется метод.
Можно определить начальный и максимальный размер памяти в куче: <b>-Xms</b> и <b>-Xmx</b> (опции JVM)	Размер памяти: <b>-Xss</b>
Если память кучи заполнена:	Если память стека заполнена:

<b>java.lang.OutOfMemoryError: Java Heap Space</b>	<b>java.lang.StackOverflowError</b>
Для кучи выделено больше памяти, чем для стека.	Из-за простоты распределения память (LIFO), стековая память работает намного быстрее кучи.

- 4) **PC Registers** – каждый поток имеет отдельно свои регистры для хранения адреса текущей выполняемой инструкции (current execution instruction). Как только операция завершается регистр программного счетчика обновляется значением следующей инструкции.
- 5) **Native Method stacks** – хранит информацию о нативных методах. Для каждого стека создается собственный стек нативных методов.

### 3. Execution Engine

Байт-код, хранящийся в Runtime Data Area выполняется Execution Engine. Механизм выполнения считывает байт-код и выполняет последовательно.

- 1) **Interpreter** – последовательно считывает байт-код, интерпретирует его и выполняет. Интерпретатор интерпретирует байт-код быстрее, но выполняет медленнее. Недостатком интерпретатора является то, что при вызове одного и того же метода несколько раз, интерпретация происходит каждый раз.
- 2) **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.
- 3) **Garbage Collector** – собирает и удаляет объекты, на которые уже никто не ссылается. Сборщик мусора срабатывает на вызов **System.gc()**, но это все-равно не гарантирует его выполнение. Сборщик мусора JVM собирает только объекты, которые были созданы с использованием ключевого слова **new**.

### 4. Java Native Interface (JNI)

JNI взаимодействует с **Native Method Libraries** и предоставляет механизму выполнения доступ к ним.

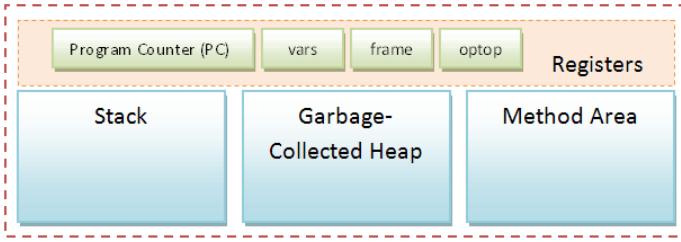
### 5. Native Method Libraries

Представляет собой **Collection of the Native Libraries**, которые необходимы механизму выполнения.

JVM – виртуальная машина Java — основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java интерпретирует и исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java. JVM предоставляет Java гибкость и независимость от платформы.

Виртуальная машина Java выполняет программу и генерирует выходное значение. Для выполнение любого кода, JVM использует различные компоненты.

JVM подразделяется на несколько компонентов (the stack, the garbage-collected heap, the registers and the method area).



## Stack

Стек в JVM(как счетчик/прилавок) хранит различные аргументы методов и их локальные переменные. Стек следит за всеми вызовами методов (**Stack Frame**, т.е. **структура стека**). Для своей работы стек использует 3 регистра, которые называются **vars**, **frame**, **optop** и уазывают на различные части текущего стека.

В структуре стека (Stack Frame) выделяют:

- **Local variables** состоит из локальных пересенных, используемых при текущем вызове метода.
- **Execution environment** используется для обслуживания операций самого стека. На эту секцию указывает регистор **frame**.
- **Operand stack** используется инструкциями байт-кода в качестве рабочего пространства (здесь размещены параметры для инструкций и здесь ищутся результаты выполнения этих инструкций). На вершину стека операндовказывает регистр **optop**.

## Method Area

Место, где хранится байт-код. Счетчик программы (Program Counter/PC) указывает на некоторый байт в зоне методов. Он также отслеживает текущую команду/инструкцию, которая сейчас выполняется (интерпретируется). После выполнения инструкции JVM устанавливает PC на следующую команду. Зона методов делится между всеми потоками процесса. Следовательно, необходима синхронизация, если больше, чем один поток обращается к методу или команде. Синхронизация в JVM достигается при помощи мониторов.

## Garbage-collected Heap

Место, хранятся объекты java-программ. Когда бы мы не создали объект, используя ключевое слово (оператор) new, приходит в действие heap и выделяется место для данного объекта оттуда. В отличие от C++, Java не имеет оператора для освобождения ранее зарезервированной памяти. Java выполняет это автоматически, используя механизм сборки мусора. До Java 6.0 в качестве логики сборки мусора использовался алгоритм **mark and sweep**. Ссылки на локальные объекты хранятся в стеке, но сами объекты хранятся только в heap. Аналогично для массивов, так как массивы в Java являются объектами.

JDK состоит из JRE и набора утилит. В свою очередь JRE состоит из JVM и исходных кодов библиотек стандартных классов, хранимых в rt.jar(JCL – Java Class Library). Java Class Library (JCL) – набор динамически подгружаемых библиотек, которые Java приложения могут использовать во время выполнения. Так как Java платформа не зависит от ОС, приложения не могут вызывать другие нативные для платформы библиотеки. Вместо этого платформа Java предоставляет расширенный набор стандартных библиотек классов (class libraries), содержащий функции, понятные ОС.

Основные задачи JCL в рамках Java платформы:

- Как и другие стандартные библиотеки кода, JCL предоставляет программисту набор утилит (Например: контейнер классов и обработчик регулярных выражений).
- Предоставляет абстрактный интерфейс для задач, которые в обычном случае должны зависеть от hardware и ОС (Например: доступ к сети и доступ к файлам).

- Некоторые платформы могут не поддерживать все действия (features), которые предлагает Java приложение. В таких случаях реализация библиотеки может эмулировать данные действия или же предоставить удобный способ проверки наличия определенных действий.

JVM содержит в себе загрузчик классов + верификатор байт-кода и «движок» для выполнения (интерпретатор и JIT компилятор). Компилятор конвертирует исходный код в промежуточный байт-код, который далее передается JVM для выполнения. В JVM загрузчик классов загружает байт-код и связывает его с библиотеками стандартных классов, которые содержатся в JRE. Далее код передается выполняющему «движку» в JVM, который интерпретирует неповторяющийся код и компилирует повторяющийся код (Напр. циклы) и в итоге конвертирует все это в машинный код (объектный код). Далее данный машинный код передается на выполнение микропроцессору.

<http://viralpatel.net/blogs/java-virtual-machine-an-inside-story/>

## CONSOLE APP RUN

### 1. Запуск приложения из командной строки

```
[MacBook-Air-MASA:src mariya$ javac -d ..\bin com.epam.console/Test.java
[MacBook-Air-MASA:src mariya$ cd ..
[MacBook-Air-MASA:Java mariya$ java -cp ..\bin com.epam.console/Test
Hello world!
```

```
Test.java
1 package com.epam.console;
2
3 public class Test{
4     public static void main(String[] args){
5         for(int i=0; i<10; i++){
6             System.out.println("Hello world!");
7         }
8     }
9 }
```

1. Написать программу и сохранить в .java формате.
2. Скомпилировать файл, находясь в его директории javac Test.java (учитывается регистр).
3. Запуска java-приложения java Test.

### 2. Компиляция и запуск приложения с несколькими файлами

```
Test.java
1 package com.epam.console.main;
2
3 import com.epam.console.util.Printer;
4
5 public class Test{
6     public static void main(String[] args){
7         Printer.print("Hello world!");
8     }
9 }
```

```
Printer.java
1 package com.epam.console.util;
2
3 public class Printer{
4     public static void print(String str){
5         System.out.println(str);
6     }
7 }
```

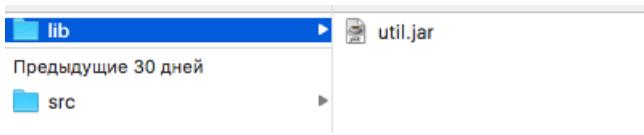
```
MacBook-Air-MASA:src mariya$  
MacBook-Air-MASA:src mariya$ javac -d .. /bin comepam/console/main/Test.java comepam/console/util/Printer.java  
[MacBook-Air-MASA:src mariya$ cd ..  
[MacBook-Air-MASA:Java mariya$ java -cp ./bin comepam/console/main/Test  
Hello world!  
[MacBook-Air-MASA:Java mariya$
```

### 3. Создание jar-файла

1. Создать папку и скопировать туда bin с содержимым для jar (без main в нашем случае)
2. Перейти в директорию bin.
3. Выполнить команду для создания jar (после этого в каталоге bin появится util.jar).

```
MacBook-Air-MASA:~ mariya$  
[MacBook-Air-MASA:~ mariya$ cd /Users/mariya/Desktop/Java  
[MacBook-Air-MASA:Java mariya$ cd TestJar/bin  
[MacBook-Air-MASA:bin mariya$  
[MacBook-Air-MASA:bin mariya$ jar cf util.jar comepam/console/util/Printer.class  
[MacBook-Air-MASA:bin mariya$
```

### 4. Запуск приложения с jar-файлом



## JDK TOOLS

### BASIC TOOLS:

- **java** – запуск (launcher) приложений (в java7 и выше используется и для разработки и для развертывания (development and deployment). Ранее развертывание выполнял лаунчер **jre**).
- **javac** – компилятор языка java.
- **javadoc** – генератор API документации.
- **jar** – создание и управление JAR (java archive) файлами.
- **javah** – используется для написание native методов. (C header and stub generator).
- **javap** – дизассемблер класс файлов.
- **jdb** – java дебаггер.
- **appletviewer** - запуск и отладка аплетов без использования веб-браузера.
- **apt** – инструмент для обработки аннотаций.
- **extcheck** – утилита для обнаружения jar конфликтов.

## SECURITY TOOLS

А) для установления политики безопасности в системе и создания приложений, которые могут работать внутри зоны действия политики безопасности удаленных сайтов:

- **keytool** – управление хранением ключей и сертификатов.
- **jarsigner** – генерирование и подтверждение JAR подписей.
- **policytool** – GUI инструмент для управления файлами политики безопасности.

Б) для достижения, перечисления и управления Kerberos tickets:

- **kinit** - tool for obtaining Kerberos v5 tickets. Equivalent functionality is available on the Solaris operating system via the kinit tool.
- **klist** - Command-line tool to list entries in credential cache and key tab. Equivalent functionality is available on the Solaris operating system via the klist tool.

- **ktab** - Command-line tool to help the user manage entries in the key table. Equivalent functionality is available on the Solaris operating system via the kadmin tool.

## INTERNATIONALIZATION TOOLS:

- **native2ascii** – конвертирует текст в Unicode Latin-1.

## REMOTE METHOD INVOCATION (RMI) ИНСТРУМЕНТЫ (ДЛЯ СОЗДАНИЯ ПРИЛОЖЕНИЙ, ВЗАИМОДЕЙСТВУЮЩИХ ЧЕРЕЗ ВЕБ ИЛИ ДРУГУЮ СЕТЬ):

- **rmic** - generate stubs and skeletons for remote objects.
- **rmiregistry** - remote object registry service.
- **rmid** - RMI activation system daemon.
- **serialver** - return class serialVersionUID.

## JAVA DEPLOYMENT TOOLS (ИНСТРУМЕНТЫ ДЛЯ РАЗВЕРТЫВАНИЯ ПРИЛОЖЕНИЙ И АППЛЕТОВ В ВЕБ):

- **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.

## JAVA TROUBLESHOOTING, PROFILING, MONITORING AND MANAGEMENT TOOLS:

- **jcmd** - JVM Diagnostic Commands tool – отправляет диагностические команды запросов в исполняемую JVM.
- **jconsole** – JMX-совместимый графический инструмент для мониторинга JVM. Может контролировать 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.)

## JAVA IDL AND RMI-IIOP ИНСТРУМЕНТЫ (TNAMESERV, IDLJ, ORBD, SERVERTOOL).

## JAVA WEB START TOOL (JAVAWS).

## JAVA WEB SERVICE TOOL (SCHEMAGEN, WSGEN, WSIMPORT, XJC).

ЕСТЬ ЕЩЕ РЯД ИНСТРУМЕНТОВ, КОТОРЫЕ НЕ ПОДДЕРЖИВАЮТСЯ И НАХОДЯТСЯ В СТАТУСЕ “ЭКСПЕРИМЕНТАЛЬНЫЕ”.

## \*.JAVA \*.CLASS FILES

```

1) package packagepath;
2) import classpath.ClassName;
3) static import classpath.ClassName;
4) ClassModifiers 5) class ClassName
   6) extends SuperclassName 7) implements Interfaces {
8) classFields;
9) attributes;
10) {
    //initializer block;
}
11) static{
    //static blocks;
}
12) constructors
13) methods
14) inner/nested classes
}

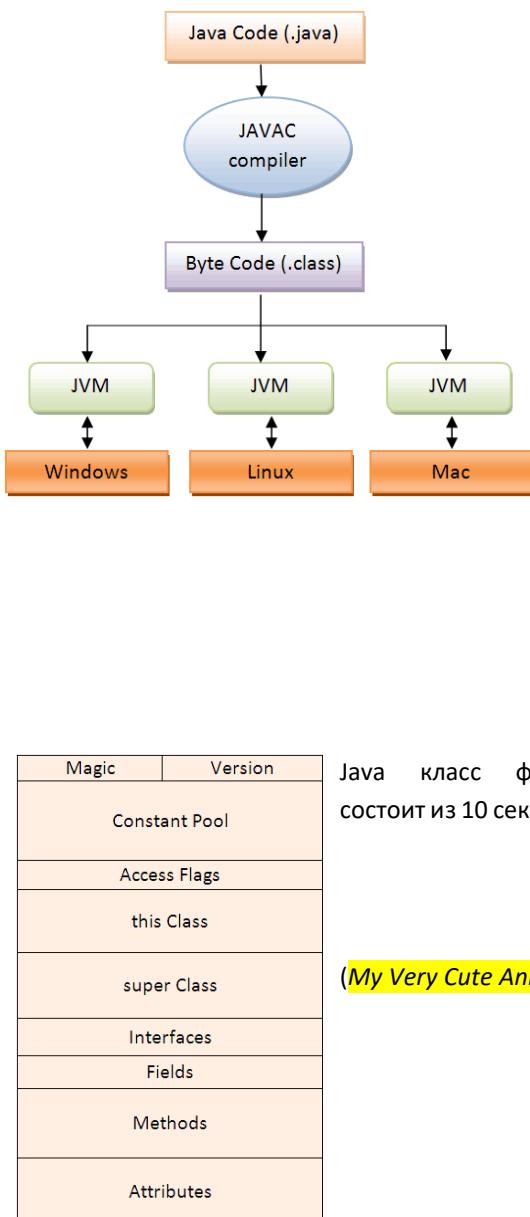
```

● - can't be skipped  
● - can be skipped

### Java class structure:

- 1) package
- 2) import
- 3) static import
- 4) class modifiers
- 5) class ClassName
- 6) super class
- 7) implemented interfaces
- 8-9) class fields and attributes
- 10) initializer block
- 11) static block
- 12) constructors
- 13) methods
- 14) inner/nested classes

}



Java код пишется и сохраняется в .java файлах. Данный код содержит один или несколько атрибутов (Classes, Methods, Variable, Objects etc.). Javac используется для компиляции такого кода и генерации .class файлов (байт-кода). JVN считывает байт-код, интерпретирует его под определенную ОС и выполняет программу.

```

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];
}

```

(My Very Cute Animal Turns Savage In Full Moon Areas):

- 1) **Magic Number:** 0xCAFEBAE – первые 4 байта, уникально идентифицирует формат класс-файла.
- 2) **Version of Class File Format:** 4 байта, версия формата класс файла (minor and major). Позволяет JVM проверять и идентифицировать класс-файл. Если число больше, чем то, что JVM может загрузить, возникнет `java.lang.UnsupportedClassVersionError`.
- 3) **Constant Pool:** 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) **Access Flags:** 2 байта – класс/интерфейс, модификаторы доступа класса (abstract, static, etc).
- 5) **This Class:** 2 байта - название данного класса, т.е. его адрес в пуле констант.
- 6) **Super Class:** 2 байта - название суперкласса (так же как и this class).
- 7) **Interfaces:** 2 байта - интерфейсы класса (полное количество интерфейсов, которые заимплементены, далее массив, состоящий из индексов на пул констант для каждого интерфейса).
- 8) **Fields:** поля класса/интерфейса, НО только те, которые определены в классе или интерфейсе ЭТОГО ФАЙЛА, а не поля, которые унаследованы от суперкласса или интерфейса.
- 9) **Methods:** методы класса (по аналогии с fields – только те методы, которые определены в данном файле).
- 10) **Attributes:** атрибуты класса (Например: имя source файла, из которого этот файл был скомпилирован, хранится в атрибуте source code).

Длина java-класса не известна до тех пор, пока он не загружен. В нем присутствуют секции с длинной переменных (пол констант, методы, атрибуты и прочее), которые организованы таким образом, что они предварены их размером или длиной, что фактически предоставляет информацию о размере переменных для JVM. Данные, записанные в класс-файле выравнены в одном байте и плотно упакованы.

Порядок данных секций строго определен и JVM четко знает, что за чем следует и в каком порядке что загружать.

Подробное описание каждого компонента:  
<http://viralpatel.net/blogs/tutorial-java-class-file-format-revealed/>

Почему ключевое слово `extends` идет перед ключевым словом `implements`, а не наоборот?

Major Version	Hex	JDK version
51	0x33	J2SE 7
50	0x32	J2SE 6.0
49	0x31	J2SE 5.0
48	0x30	JDK 1.4
47	0x2F	JDK 1.3
46	0x2E	JDK 1.2
45	0x2D	JDK 1.1

1. Так говорит нам спецификация Oracle
2. Когда код компилируется в \*.class файл секция Super Class идет перед секцией Interfaces. Для сохранения однозначности необходим такой порядок
3. В интерфейсе может быть определен такой же метод, как и в супер классе, если это так, то нет необходимости требовать реализацию метода интерфейса.

Сколько всего можно в классе определить полей и методов?

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

Сколько максимально можно имплементить интерфейсов?

По тем же причинам, из-за 2 ух байт 65535.

Ограничения на название класса

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.

## BYTECODE

Bytecode java – source code трансформированный в множество базовых операций. Каждая операция состоит из одного байта, который представляет инструкции для выполнения (называется **opcode** или **operator code**), вместе с нулем или более байт для передачи параметров (но большинство операций используют стек operandов для передачи параметров). Из возможных 256 однобайтовый opcodes (от значения 0x00 до 0xFF), 204 временно используются в спецификации Java8.

Рассмотрим пример:

Есть класс в файле Test.java:

```
public class Test{
{
    System.out.println("1 logic block");
}
private int a=10;
{
    System.out.println("2 logic block");
}
static{
    System.out.println("3 static");
}
private static int b=9;
static{
    System.out.println("4 static");
}
public Test(){
    a=19;
}

}
```

После компиляции и декомпиляции посредством **javap -verbose Test.class**

```
Classfile /C:/Users/Maryia_Bystrova/Desktop/Test.class
Last modified Jun 2, 2017; size 556 bytes
MD5 checksum 5c173a2bb7a5dc0e60a5a6e7d6036b57
Compiled from "Test.java"
public class Test
    minor version: 0
    major version: 52
    flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
#1 = Methodref    #11.#22    // java/lang/Object."<init>":()V
#2 = Fieldref     #23.#24    // java/lang/System.out:Ljava/io/PrintStream;
#3 = String        #25    // 1 logic block
#4 = Methodref     #26.#27    // java/io/PrintStream.println:(Ljava/lang/String;)V
#5 = Fieldref     #10.#28    // Test.a:l
#6 = String        #29    // 2 logic block
#7 = String        #30    // 3 static
#8 = Fieldref     #10.#31    // Test.b:l
#9 = String        #32    // 4 static
#10 = Class       #33    // Test
#11 = Class       #34    // java/lang/Object
#12 = Utf8          a
#13 = Utf8          l
#14 = Utf8          b
#15 = Utf8         <init>
#16 = Utf8          ()V
#17 = Utf8        Code
```

```

#18 = Utf8      LineNumberTable
#19 = Utf8      <clinit>
#20 = Utf8      SourceFile
#21 = Utf8      Test.java
#22 = NameAndType #15:#16    // "<init>":()V
#23 = Class     #35        // java/lang/System
#24 = NameAndType #36:#37    // out:Ljava/io/PrintStream;
#25 = Utf8      1 logc block
#26 = Class     #38        // java/io/PrintStream
#27 = NameAndType #39:#40    // println:(Ljava/lang/String;)V
#28 = NameAndType #12:#13    // a:I
#29 = Utf8      2 logc block
#30 = Utf8      3 static
#31 = NameAndType #14:#13    // b:I
#32 = Utf8      4 static
#33 = Utf8      Test
#34 = Utf8      java/lang/Object
#35 = Utf8      java/lang/System
#36 = Utf8      out
#37 = Utf8      Ljava/io/PrintStream;
#38 = Utf8      java/io/PrintStream
#39 = Utf8      println
#40 = Utf8      (Ljava/lang/String;)V
{
public Test();
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 logc block
9: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
12: aload_0
13: bipush     10
15: putfield    #5           // Field a:I
18: getstatic   #2           // Field java/lang/System.out:Ljava/io/PrintStream;
21: ldc        #6           // String 2 logc block
23: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
26: aload_0
27: bipush     19
29: putfield    #5           // Field a: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 3 static
5: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: bipush     9
10: putstatic   #8          // Field b:I
13: getstatic   #2           // Field java/lang/System.out:Ljava/io/PrintStream;
16: ldc        #9           // String 4 static
18: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
21: return
LineNumberTable:
line 10: 0

```

```
line 12: 8
line 14: 13
line 15: 21
}
SourceFile: "Test.java"
```

По сути получили следующую структуру файла .java:

```
import java.io.PrintStream;
public class Test
{
    private int a;
    private static int b;
    static
    {
        System.out.println("3 static");
        b = 9;
        System.out.println("4 static");
    }
    public Test()
    {
        System.out.println("1 logic block");
        a = 10;
        System.out.println("2 logic block");
        a = 19;
    }
}
```

Все статические блоки и инициализаторы сгруппировались в один статический блок.

Все логические блоки и инициализаторы полей группируются внутри конструктора до действий, прописанных в самом конструкторе.

Если в классе несколько конструкторов, то в каждом конструкторе будут прописаны действия из логических блоков и инициализаторов.

## WHAT IS JAVA CLASSPATH

CLASSPATH – параметр JDK/JVM (компилятора), который используется загрузчиком классов и указывает на расположение классов, определенных пользователем и пакетов. Данный параметр можно настроить через командную строку или через переменные окружения.

Предпочтительный способ задания через опции `-classpath` или `-cp` : **`java -cp "path" ClassToRun`**. Данный способ позволяет указывать CLASSPATH конкретно для каждого приложения. Данную опцию имеют следующие команды: `java`, `jdb`, `javac`, `javah` и `jdeps`.

Дефолтное значение переменной CLASSPATH “.” (просматривается только текущая директория). Явное указание переопределяет данное значение. Если в переменной CLASSPATH указано много директорий Java будет искать необходимый class-файл в первой директории и будет просматривать вторую директорию только если не найдет указанный файл в первой.

Разница между переменными PATH и CLASSPATH заключается в том, что первая указывает на расположение Java команд, в то время, как вторая – Java class файлов.

`ClassNotFoundException` – `Exception`, возникаемое, когда Java программа динамически пытается подгружать java-класс во время выполнения(at Runtime) и не находит соответствующий класс файл в CLASSPATH. Ключевым здесь является «**ДИНАМИЧЕСКИ**» и «**ВО ВРЕМЯ ВЫПОЛНЕНИЯ**». Классический пример данной ошибки, когда вы пытаетесь загрузить JDBC драйвер через `Class.forName("driver name")` и возникает `java.lang.ClassNotFoundException: com.mysql.jdbc.Driver`. Данная ошибка на самом деле возникает, когда Java пытается загрузить класс, используя `forName()` или `loadClass()` из `ClassLoader`. Ключевое то, что 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.

С другой стороны `NoClassDefFoundError` это ошибка и является более критичной. Данная ошибка возникает, когда **определенный класс присутствовал в Java classpath во время компиляции, но не доступен во время**

**выполнения программы**. Например: при использовании log4j.jar для логгирования не включили jar файл в classpath java во время выполнения. То есть *класс присутствовал во время компиляции, но стал недоступен во время выполнения*. NoClassDefFoundError может также возникнуть в статическом инициализаторе или когда класс не виден загрузчику классов. (<http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html>)

--чем параметр classpath в java отличается от этого параметра в javac

## ENVIRONMENT VARIABLES FOR JAVA

Указание переменных окружения не является обязательным, но тогда необходимо указывать полный путь команды, например: C:\Java\jdk1.7.0\bin\javac MyClass.java

Для удобного запуска javac.exe, java.exe, javadoc.exe и др. необходимо настроить переменные окружения (указать набор директорий через ; в переменной PATH). Директория **bin** должна быть указана только один раз, даже, если указано несколько раз – используется первое.

## COMPILATION/INTERPRETATION/TRANSLATION

**Трансляция** программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определённом смысле, равносильную первой. При трансляции выполняется перевод программы, понятной человеку, на язык, понятный компьютеру. Выполняется специальными программными средствами (транслятором).

**Трансляторы реализуются в виде компиляторов или интерпретаторов.** С точки зрения выполнения работы компилятор и интерпретатор существенно различаются.

**Компиляция** — преобразование программой-компилятором исходного текста программы, написанного на языке высокого уровня в машинный язык, в язык, близкий к машинному, или в объектный модуль. Результатом компиляции является объектный файл с необходимыми внешними ссылками для компоновщика. Компилятор читает всю программу целиком, делает ее перевод и создает законченный вариант программы на машинном языке, который затем и выполняется.

Виды компиляции:

- **Пакетная.** Компиляция нескольких исходных модулей в одном пункте задания.
- **Построчная.** То же, что и интерпретация.
- **Условная.** Компиляция, при которой транслируемый текст зависит от условий, заданных в исходной программе. Так, в зависимости от значения некоторой константы, можно включать или выключать трансляцию части текста программы.

**Интерпретация** — процесс непосредственного покомандного выполнения программы без предварительной компиляции, «на лету»; в большинстве случаев интерпретация намного медленнее работы уже скомпилированной программы, но не требует затрат на компиляцию, что в случае небольших программ может повышать общую производительность.

Типы интерпретаторов:

- **Простой интерпретатор** анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Его достоинство - мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.
- **Интерпретатор компилирующего типа** — это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или р-код, и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина). Его достоинство – большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе). Недостатки — большее требование к ресурсам и требование на корректность исходного кода.

Алгоритм работы простого интерпретатора:

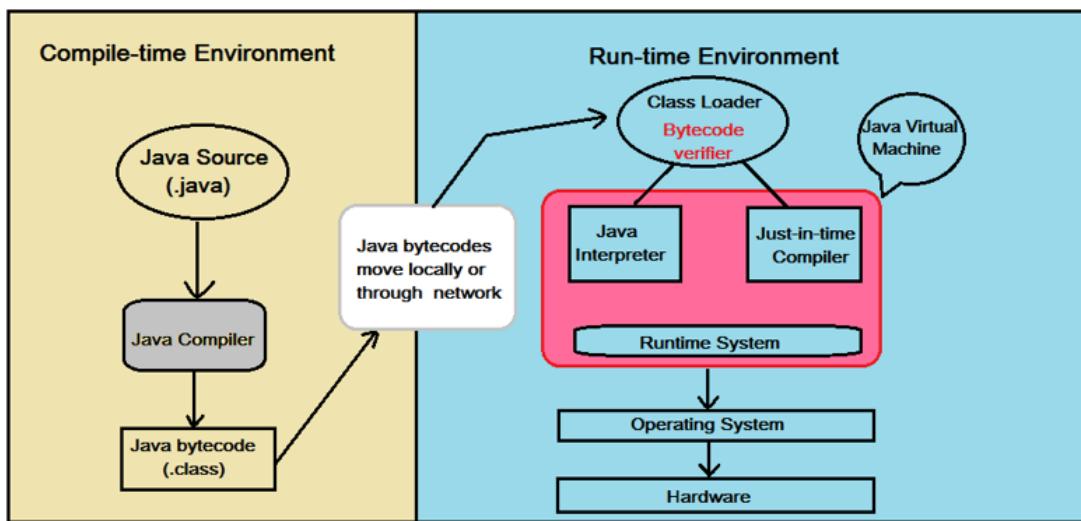
1. прочитать инструкцию;
2. проанализировать инструкцию и определить соответствующие действия;

3. выполнить соответствующие действия;
4. если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к пункту 2.

### Жизненный цикл программы Java



Java-программа транслируется в байт-код компилятором `javac.exe`. Оттранслированная в байт-код программа имеет расширение `class`. Для запуска программы нужно вызвать интерпретатор `java.exe`, указав в параметрах вызова, какую программу ему следует выполнять. Кроме того, ему нужно указать, какие библиотеки нужно использовать при выполнении программы. Библиотеки размещены в файлах с расширением `jar`.



1 Этап: компиляция исходного кода в байт-код java-компилятором.

2 Этап: Выполнение байт-кода JVM используя

- a) Just-In-Time компиляция на лету во время выполнения (компиляция байт-кода в язык, понятный софту).
- b) Полная интерпретация байт-кода в машинный код и далее непосредственный запуск его.

Также возможна компиляция наперед и выполнение машинного кода. Выполнение подпроцессором напрямую. (Байт код включен в набор машинных команд для некоторых процессоров).

Основные действия, предпринимаемые JVM:

- Сначала класс проходит верификацию (проверяется, что с байт-кодом ничего не случилось, class file format verifying)
- Затем класс загружается и считывается все его метаданные (class loading).

- Затем класс проходит инициализацию (т.е. все статические данные принимают значение по умолчанию, или указанное статическим блоком инициализации, или указанное при объявлении, `class initializing`)
- Затем подгружаются классы, с которыми взаимодействует “наш” класс (class linking (referencing)) JVM выполняет и другие функции, такие как выделение нужного количества CPU, управление рекурсивным стеком, управление runtime data areas и блокировками.

## MAVEN / GRADLE / ANT

### ANT

Ant «Another Neat Tool») — утилита для автоматизации процесса сборки программного продукта. Является платформонезависимым аналогом утилиты `make`, где все команды записываются в XML-формате. Является первым инструментом для сборки java приложений и встроен в java.

В отличие от `make`, утилита Ant полностью независима от платформы, требуется лишь наличие на применяемой системе установленной рабочей среды Java — JRE. Отказ от использования команд операционной системы и формат XML обеспечивают переносимость сценариев.

Управление процессом сборки происходит посредством XML-сценария, также называемого Build-файлом. В первую очередь этот файл содержит определение проекта, состоящего из отдельных целей (**Targets**). Цели сравнимы с процедурами в языках программирования и содержат вызовы команд-заданий (**Tasks**). Каждое задание представляет собой неделимую, атомарную команду, выполняющую некоторое элементарное действие.

Между целями могут быть определены зависимости — каждая цель выполняется только после того, как выполнены все цели, от которых она зависит (если они уже были выполнены ранее, повторного выполнения не производится).

Типичными примерами целей являются **clean** (удаление промежуточных файлов), **compile** (компиляция всех классов), **deploy** (развертывание приложения на сервере). Конкретный набор целей и их взаимосвязи зависят от специфики проекта. Ant позволяет определять собственные типы заданий путём создания Java-классов, реализующих определённые интерфейсы.

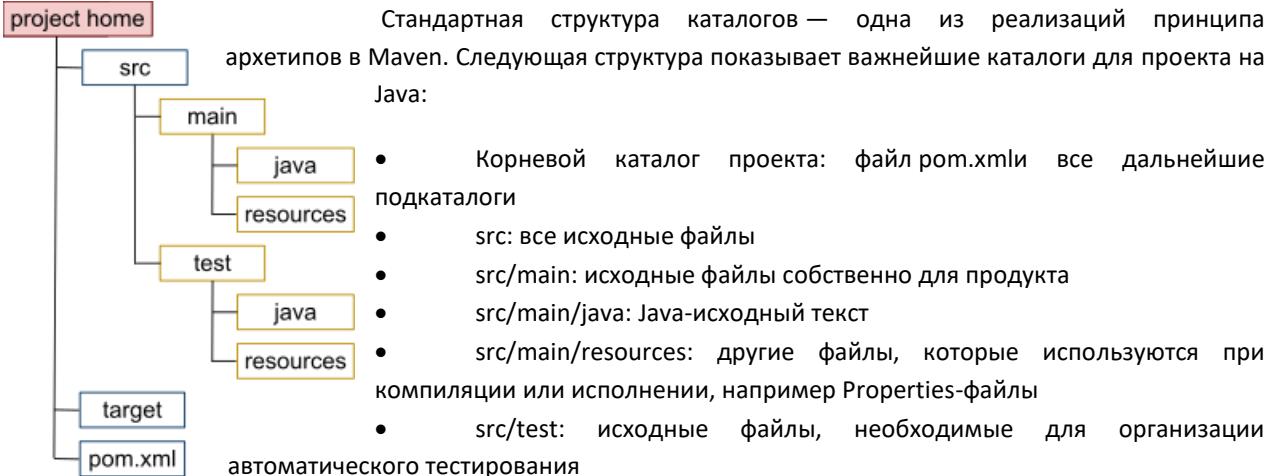
Код	Действие
<code>javac</code>	компиляция Java-кода
<code>copy</code>	копирование файлов
<code>delete</code>	удаление файлов и директорий
<code>move</code>	перемещение файлов и директорий
<code>replace</code>	замещение <a href="#">фрагментов</a> текста в файлах
<code>junit</code>	автоматический запуск <a href="#">юнит-тестов</a>
<code>exec</code>	выполнение внешней команды
<code>zip</code>	создание архива в формате <a href="#">Zip</a>
<code>cvs</code>	выполнение <a href="#">CVS</a> -команды
<code>mail</code>	отправка <a href="#">электронной почты</a>
<code>xslt</code>	наложение <a href="#">XSLT</a> -преобразования

### MAVEN

После Ant пришел Maven, который стандартизировал то, как Java проекты собираются и как управлять зависимостями, т.е их выбором и скачиванием (внешние JAR файлы используемые в Java проектах).

Maven — «собиратель знаний» — фреймворк для автоматизации сборки проектов на основе описания их структуры в файле, написанных на языке POM (Project Object Model), который является подмножеством XML. Maven обеспечивает декларативную, а не императивную (в отличие от средства автоматизации сборки Apache Ant) сборку проекта. Все задачи по обработке файлов, описанные в спецификации, Maven выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

Maven используется для построения и управления проектами, написанными на Java, C#, Ruby, Scala, и других языках.



- `src/test/java`: JUnit-тест-задания для автоматического тестирования
- `target`: все создаваемые в процессе работы Мавена файлы
- `target/classes`: компилированные Java-классы

## Жизненный цикл

Жизненный цикл Maven проекта — это список поименованных фаз, определяющий порядок действий при его построении. Жизненный цикл Maven содержит три независимых порядка выполнения:[15]

- `clean` — жизненный цикл для очистки проекта. Содержит следующие фазы:
  1. `pre-clean`
  2. `clean`
  3. `post-clean`
- `default` — основной жизненный цикл, содержащий следующие фазы:
  1. `validate` - выполняется проверка, является ли структура проекта полной и правильной.
  2. `generate-sources`
  3. `process-sources`
  4. `generate-resources`
  5. `process-resources`
  6. `compile` - компилируются исходные тексты.
  7. `process-test-sources`
  8. `process-test-resources`
  9. `test-compile`
  10. `test` - собранный код тестируется заранее подготовленным набором тестов.
  11. `package` - упаковка откомпилированных классов и прочих ресурсов. Например, в JAR-файл.
  12. `integration-test` - программное обеспечение в целом или его крупные модули подвергаются интеграционному тестированию. Проверяется взаимодействие между составными частями программного продукта.
  13. `install` - установка программного обеспечения в локальный Maven-репозиторий, чтобы сделать его доступным для других проектов текущего пользователя.
  14. `deploy` - стабильная версия программного обеспечения распространяется на удаленный Maven-репозиторий, чтобы сделать его доступным для других пользователей.
- `site` — жизненный цикл генерации проектной документации. Состоит из фаз:
  1. `pre-site`
  2. `site`
  3. `post-site`
  4. `site-deploy`
  - 5.

Gradle — система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но предоставляющая DSL на языке Groovy вместо традиционной XML-образной формы представления конфигурации проекта.

В отличие от Apache Maven, основанного на концепции жизненного цикла проекта, и Apache Ant, в котором порядок выполнения задач (*targets*) определяется отношениями зависимости (*depends-on*), Gradle использует направленный ациклический граф для определения порядка выполнения задач.

Gradle был разработан для расширяемых многопроектных сборок, и поддерживает инкрементальные сборки, определяя, какие компоненты дерева сборки не изменились и какие задачи, зависимые от этих частей, не требуют перезапуска.

Основные плагины предназначены для разработки и развертывания Java, Groovy и Scala приложений, но готовятся плагины и для других языков программирования.

## LESSON 2

### INTERFACE VS ABSTRACT CLASS

	Abstract class	Interface (абстрактное описание/контракт)
<b>Назначение</b>	Используются в качестве базовых классов для расширения.  Если в дополнение к разделению интерфейса от реализации необходимо еще предоставить базовый класс или реализацию по умолчанию интерфейса, добавляют абстрактный или обычный класс, реализующий интерфейс (в Java 8 это можно сделать и при помощи <i>default</i> и <i>static</i> методов интерфейсов).	Используются для разделения интерфейсной части от конкретной реализации ( <b>обеспечение полиморфизма</b> ). Другими словами, они позволяют обеспечить независимость от конкретной реализации посредством их использования (независимость классов, использующих интерфейсы, от классов, их реализующих). Таким образом, можно легко изменять реализацию интерфейсов, не меняя ничего в классе, использующем интерфейс.
<b>Сколько может наследовать/реализовывать Java класс?</b>	Java класс может наследовать только 1 абстрактный класс.	Java класс может реализовывать сколько угодно интерфейсов.
<b>Обозначение</b>	abstract class / extends	interface / implements
<b>Методы</b>	Может содержать как абстрактные, так и обычные методы. Может не содержать ни одного абстрактного метода (используется для того, чтобы запретить создание экземпляров этого класса).	Может содержать лишь абстрактные методы ( <i>implicitly abstract</i> ).  Начиная с Java 8 можно определять в интерфейсах:  A) <u>default</u> методы  B) <u>static</u> методы

<b>Модификаторы доступа методов по умолчанию</b>	friendly-package (как у обычных классов)	public abstract <b>Доступны только модификаторы: public, abstract, default, static и strictfp. Нельзя использовать protected и т.п.</b>
<b>Модификаторы доступа переменных (variables)</b>	frindly- package (как у обычных классов)	public final static
<b>Наследование (extends)</b>	Может наследовать только один класс (абстрактный или обычный) и реализовывать множество интерфейсов.	Может наследовать сколько угодно других интерфейсов. В таком случае, класс, реализующий интерфейс, должен реализовывать методы всей цепочки интерфейсов.
<b>Определение конструкторов</b>	Можно определить конструктор, но нельзя создать объект абстрактного класса, только если в анонимном классе определить все абстрактные методы.	<b>Не может содержать конструкторов.</b> Но как и для абстрактных классов работает вариант с анонимным классом.
<b>Запуск</b>	Может содержать метод main и служить точкой запуска приложения.	Не может этого сделать.

## Default методы интерфейсов Java 8

Default методы интерфейсов могут содержать дефолтную реализацию методов интерфейса. Они должны быть помечены ключевым словом **default**. Классы, реализующие интерфейсы с default методами и не содержащие собственную реализацию данных методов, автоматически получают их дефолтную реализацию.

Таким образом, это кладет конец классическому шаблону предоставления интерфейса и абстрактного класса, реализующего большинство или все методы (Например: Collection/AbstractCollection). Теперь можно просто добавить дефолтную реализацию в сами интерфейсы.

**Что произойдет, если один и тот же метод определен сначала как метод по умолчанию в одном интерфейсе, а затем в суперклассе или другом интерфейсе?**

1. Одерживают верх суперклассы. Если в суперклассе предоставляется конкретный метод, то методы по умолчанию с одинаковой сигнатурой (именем и типами параметров) просто игнорируются.
2. Конфликтуют интерфейсы. Если в суперинтерфейсе предоставляется метод по умолчанию, а в другом интерфейсе – метод с такой же сигнатурой определен либо же как default, либо же обычный абстрактный, то конфликт необходимо разрешить переопределив метод.

## Например:

- 1) Два интерфейса содержат default методы с одинаковой сигнатурой

```
interface Named {
    default String getName() {
        return getClass().getName() + "_" + hashCode();
    }
}

interface Person {
    long getId();
    default String getName() { return "John Q. Public"; }
}
```

Класс наследует два несогласованных метода getName(). Вместо того, чтобы отдавать предпочтение одному интерфейсу, компилятор java сообщает об ошибке, оставляя ее обработку программисту для *устранения неоднозначности*. Можно либо же переопределить, либо же выбрать один из двух конфликтующих методов.

```
class Student implements Person, Named {  
    public String getName() { return Person.super.getName(); }  
    ...  
}
```

2) Два интерфейса: один метод – default, а второй – abstract

```
interface Named {  
    String getName();  
}  
  
interface Person {  
    long getId();  
    default String getName() { return "John Q. Public"; }  
}
```

Все равно необходимо устранить неопределенность переопределением метода (для единобразия).

3) Два интерфейса: оба методы abstract.

Как и до Java 8 возникает бесконфликтная ситуация: нужно реализовать метод или же оставить его нереализованным (объявить класс abstract).

4) Базовый класс с методом и интерфейс с default методом с той же сигнатурой.

```
class Student extends Person implements Named { ... }
```

В этом случае имеет значение только метод из суперкласса, а любой другой метод по умолчанию из интерфейса игнорируется. Класс одерживает верх. Это правило введено для совместимости с предыдущими версиями.

*Нельзя создать метод по умолчанию, в котором переопределяется один из методов класса Object (toString, equals и т.д.), даже с учетом того, что это может показаться привлекательным для интерфейсов типа List. В итоге такой метод не может оказаться верх над методом Object.equals(), так как классы всегда одерживают верх.*

## Статические методы в интерфейсах Java 8

Формальных причин, по которым эта операция была незаконной, никогда не существовало. Это просто противоречило самому характеру интерфейсов как абстрактных описаний. Статические методы обычно сопутствовали классам. Например, в стандартной библиотеке можно обнаружить пары интерфейсов/классов, такие как Collection/Collections, Path/Paths. С возможностью добавления статических методов в интерфейсы, такие вспомогательные классы для служебных методов становятся не нужны.

*Статический метод доступен только по имени интерфейса, но не доступен на объекте, даже по интерфейской ссылке!*

```
public class App {  
    public static void main(String[] args) {  
        A a = new C();  
        a.defaultMethod();  
        a.abstractMethod();  
        a.staticMethod();  
        A.staticMethod();  
    }  
  
    interface A{  
        void abstractMethod();  
    }  
    default void defaultMethod(){  
        System.out.println("default A");  
    }  
    static void staticMethod(){  
        System.out.println("static A");  
    }  
}  
class C implements A{  
    @Override  
    public void abstractMethod() {  
        System.out.println("abstract from C");  
    }  
}
```

## Marker interfaces

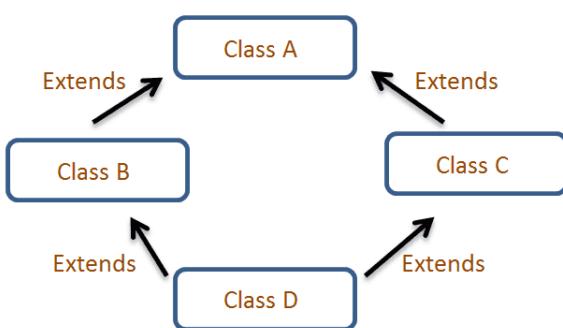
Marker интерфейсы – интерфейсы без переменных и методов (пустые интерфейсы). Они используются для того, чтобы обозначить что-то компилятору (JVM) так, что виртуальная машина, увидев интерфейс, например, Cloneable выполняла операции по обеспечению клонирования. Данные интерфейсы указывают, что класс относится 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

## Почему нельзя обойтись флаговыми переменными?

Вообще возможно ими обойтись и не использовать данные интерфейсы, но код становится менее читаем. Начиная с Java 1.5 необходимость в данных интерфейсах отпала благодаря использованию Java аннотаций.

## DIAMOND PROBLEM



- Class B и Class C наследуют Class A и методы класса Class A переопределены в обоих классах (B и C).
- Class D наследует оба Class B и C (**в Java НЕ ПОДДЕРЖИВАЕТСЯ**), в таком случае, если нам понадобится вызвать метод на экземпляре класса D, определенный в классе A и переопределенный в классах B и C, компилятор не будет знать, какой метод ему нужно вызывать, т.е. возникает неопределенность.

### Порядок инициализации полей класса

1. При создании объекта или при первом обращении к статическому методу (полю) статические поля инициализируются значениями по умолчанию.
2. Инициализаторы всех статических полей и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.
3. При вызове конструктора класса все поля данных инициализируются своими значениями, предусмотренными по умолчанию.
4. Инициализаторы всех полей и блоки инициализации выполняются в порядке из перечисления в объявлении класса.
5. Если в первой строке конструктора вызывается тело другого конструктора, то выполняется вызванный конструктор.
6. Выполняется тело конструктора,

### STATIC BLOCKS

Статический блок инициализации – блок кода внутри Java класса, который выполняется, когда класс впервые загружается в память JVM. В большинстве случаев, статические блоки используются для инициализации переменных.

Статический блок вызывается только один раз во время загрузки класса в память и не может иметь возвращающий тип и никакие ключевые слова (this/super).

Статический блок может использовать внутри себя только статические переменные и статические методы класса (только статику).

**Исключения:** Если во время работы статического блока выбрасывается исключение, оно обрамляется в ***java.lang.ExceptionInInitializerError***.

Еще один нюанс, блок статической инициализации может создаваться сам при компиляции программы:

Например:

```
1 | public static int MAX = 100;
```

Будет создан код:

```
1 | public static int MAX;
2 | static{
3 |     MAX = 100;
4 | };
```

```

1
2 | public class StaticBlocks {
3 |
4 |     public static int a;
5 |     static {
6 |         a=9;
7 |         b=10; Используем до объявления
8 |     }
9 |     public static int b;
10|     public static void main(String[] args) {
11|         System.out.println(a + " " + b);
12|     }
13| }
14|
15}
16

```

Problems @ Javadoc Declaration Console Servers  
<terminated> StaticBlocks [Java Application] /Library/Java/JavaVirtualMach  
9 10

### ВЛОЖЕННЫЕ КЛАССЫ (NESTED)

Класс, вложенный в интерфейс, статический по умолчанию. В Java top-level класс не может быть объявлен как статический.

#### ОБЪЯВЛЕНИЕ

- могут быть объявлены внутри класса

- внутри метода и внутри логического блока, но область видимости будет ограничиваться видимостью блока, в котором он объявлен. НО сохраняет доступ ко всем полям и методам внешнего класса, а также имеет доступ ко всем константам этого блока. НЕ ОБЪЯВЛЯЮТСЯ С ПОМОЩЬЮ СПЕЦИФИКАТОРА ДОСТУПА

## АТРИБУТЫ ДОСТУПА

Доступ к элементам внутреннего класса возможен из внешнего только посредством создания объекта внутреннего класса, а методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса.

Доступ внутреннему классу будет разрешен также к private полям.

- **public**
  - виден для любых классов
  - можно создавать объект такого внутреннего класса во внешнем классе и вне его
- **friendly-package**
  - виден для классов того же пакета, что и внешний класс
  - можно создать объект внутреннего класса только в классах того же пакета
- **protected**
  - доступ для класса другого пакета открыт только, если этот класс является СУПЕРКЛАССОМ внешнего класса для внутреннего
- **private**
  - видимость только внутри класса-владельца
  - можно создать объект внутреннего класса только во внешнем классе
- **final** (запрещено наследование)
- **abstract**

## ДОСТУП

Nested класс имеет прямой доступ ко всем статическим полям и методам внешнего класса.

Для доступа к нестатическим полям и методам внешнего класса необходимо создать объект внешнего класса.

## СОЗДАНИЕ ОБЪЕКТА

Нет необходимости для создания объекта вложенного класса создавать объект внешнего класса.

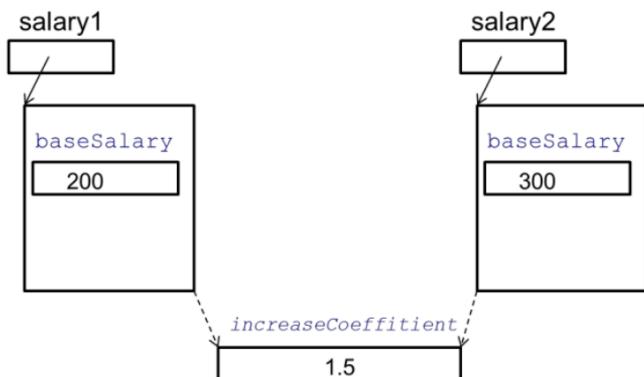
Создать объект Nested класса можно путем:

```
Outer.Nested n = new Outer.Nested();
```

## ВОЗМОЖНОСТИ INNER КЛАССОВ

- могут быть производными от других классов
- могут быть базовыми
- могут реализовывать интерфейсы

## STATIC ПОЛЯ



Поля данных, объявленные в классе как `static`, являются общими для всех объектов класса и называются переменными класса. Локальные переменные не могут быть обозначены, как статические. Это будет сопровождаться ошибкой компиляции “illegal start of expression”, так как память не сможет быть выделена во время загрузки класса в память.

Они используют память выделенную только один раз при загрузке класса в память. Таким образом если один объект изменит значение такого поля, то это изменение

увидят все объекты.

Статические поля используются довольно редко, а вот поля `static final` наоборот часто.

Статические константы нет смысла делать закрытыми, а обращаются к ним через имя класса:  
**имя\_класса.имя\_статической\_константы**

Статические поля и методы не могут обращаться к нестатическим полям и методам напрямую (по причине недоступности ссылки `this`), так как для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

Исходя из того, что статические поля являются общими для всех экземпляров класса – они не являются потокобезопасными.

При десериализации статические поля инициализируются значениями по умолчанию, или же, если в области видимости существует объект того же класса с проинициализированной статикой, берут у него значение.

## STATIC МЕТОДЫ

Статические методы не работают с объектами, поэтому их использовать следует в двух случаях:

- когда методу не нужен доступ к состоянию объекта, а все необходимые параметры задаются явно (например, метод `Math.pow(...)`);
- когда методу нужен доступ только к статическим полям класса (статический метод не может получить доступ к нестатическим полям класса, так как они принадлежат объектам, а не классам).

Статические методы можно вызывать, даже если ни один объект этого класса не создан. Кроме того, статические методы часто используют в качестве порождающих.

## STATIC IMPORT (Java 5 и выше)

Использование `import static` позволяет импортировать не классы, а статические методы и статические поля, описанные в этих классах. В таком случае в коде обращаться к таким методам и полям можно не указывая имя класса. Однако статическим импортом не стоит злоупотреблять, так как его применение может затруднить чтение кода.

### Можно ли перегружать (overload) статические методы?

Да, можно.

### Можно ли переопределять (override) статические методы?

Нет, нельзя, так как статические методы являются методами класса. Говорят, что при «переопределении» происходит скрытие или подмена метода. Хотя это нельзя назвать полноценным переопределением.

### Почему метод main() является статическим?

Если бы он не был статическим, JVM было бы необходимо сначала создавать объект, а затем вызывать метод, что вызывало бы дополнительный расход памяти.

### Может ли конструктор быть статическим?

Нет, т. к. конструктор используется для создания объекта и не имеет смысла делать его статическим.

### Могут ли статические методы быть абстрактными (static abstract)?

Нет. Статические методы вызываются напрямую по имени класса и таким образом можно было бы вызвать нереализованный метод.

## CONSTANTS IN JAVA

### КОНСТАНТНЫЕ ПОЛЯ

Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода.

Константа может быть объявлена **как поле экземпляра класса, но не проинициализирована**. В этом случае она должна быть проинициализирована в логическом блоке класса или конструкторе, но только в одном из указанных мест.

Константные статические поля могут быть проинициализированы или при объявлении, или в статическом блоке инициализации.

Значение по умолчанию константа получить не может в отличие от переменных класса.

### ПРЕДОТВРАЩЕНИЕ ПЕРЕОПРЕДЕЛЕНИЯ МЕТОДОВ

Чтобы предотвратить переопределение методов их необходимо объявить терминальными с помощью ключевого слова **final**.

### ПРЕДОТВРАЩЕНИЕ НАСЛЕДОВАНИЯ

Классы, объявленные как терминальными, нельзя расширять. Объявить терминальный класс можно следующим образом.

Если класс объявлен терминальным, то это не значит, что его поля стали константными.

```
public final class Book {}  
  
public class ProgrammerBook extends Book{} // error
```

### ЧТО МОЖНО ИСПОЛЬЗОВАТЬ В ЛОКАЛЬНЫХ ВНУТРЕННИХ КЛАССАХ?

Поля класса + локальные константные поля или неизменяемые поля (псевдоконстантные).

### Могут ли final поля быть непроинициализированы?

Нет. Они обязаны быть проинициализированы в одном из доступных мест.

## Что такое (static) blank variable?

Final переменная, которая не была проинициализирована на месте декларирования. Её необходимо проинициализировать в другом доступном для инициализации месте (иначе - ошибка компиляции).

## Могут ли final методы быть унаследованы?

Да, они наследуются, но не могут быть переопределены.

## ENUM JAVA 5 (ENUM VS CLASS VS INTERFACE) - КЛАСС

```
class Workdays {  
    public static final Workdays MONDAY;  
    public static final Workdays TUESDAY;  
    public static final Workdays WEDNESDAY;  
    public static final Workdays THURSDAY;  
    public static final Workdays FRIDAY;  
  
    static {  
        MONDAY = new Workdays();  
        TUESDAY = new Workdays();  
        WEDNESDAY = new Workdays();  
        THURSDAY = new Workdays();  
        FRIDAY = new Workdays();  
    }  
}
```

```
public enum Workday {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY  
}
```

В отличие от статических констант, предоставляют **типовизированный, 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, соответствующий значению передаваемой строки;
- int ordinal() — возвращает позицию элемента перечисления;
- 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**, каждый объект перечисления уникален, создается при загрузке перечисления в виртуальную машину, поэтому допустимо сравнение ссылок для объектов перечислений, **можно использовать в switch, if, foreach, еще используют в Map/Set**.

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 переследия через equals (не появится NullPointerException)

Метод equals у интерфейсов реализован:

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

Элементы перечислений удобно использовать в switch.

## КОНСТРУКТОРЫ

```
enum E{  
    EN("en"), RU("ru"), BY("by");  
  
    private final String str;  
  
    private E(String str){  
        this.str = str;  
    }  
}
```

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

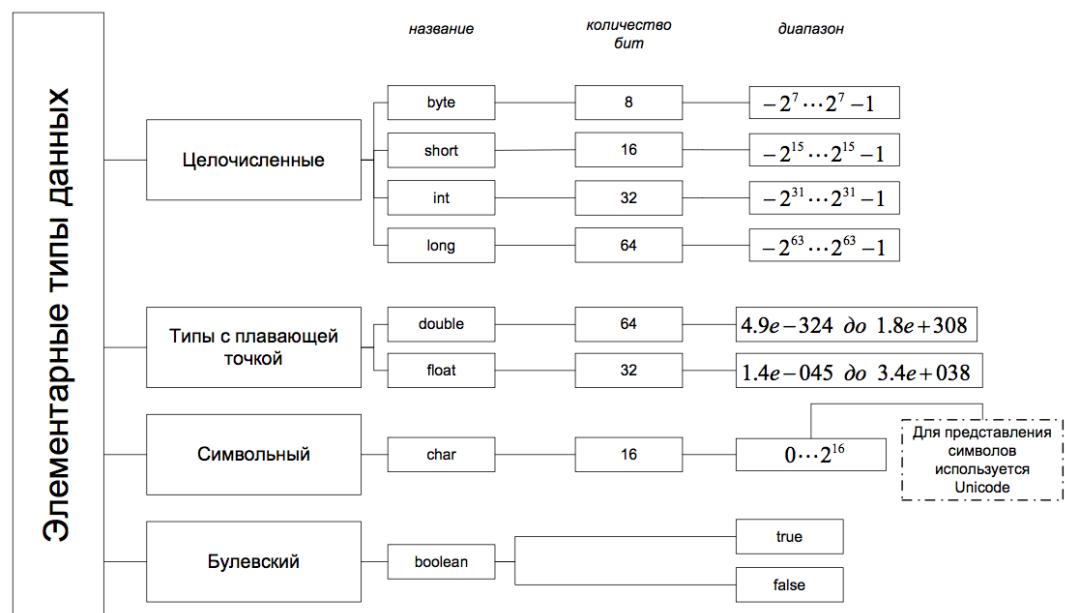
## АНОНИМНЫЕ КЛАССЫ ПЕРЕЧИСЛЕНИЯ

Отдельные элементы перечисления **могут** реализовывать свое собственное поведение.

## DATA TYPES (PRIMITIVE AND REFERENCES DATA TYPES, AUTOBOXING AND UNBOXING)

### ПРИМИТИВНЫЕ ТИПЫ

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.

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

```
int i = 10000000;
byte b = (byte)i;
System.out.println(b);
```

//-128

### Особенности работы с переменными

- Java не позволяет присваивать переменной значение более длинного типа.
- Исключение составляют операторы инкремента, декремента и операторы +=, \*=, \*=, /=.

### Преобразование примитивных типов

Java запрещает смешивать в выражениях величины разных типов, однако при числовых операциях такое часто бывает необходимо.

Различают:

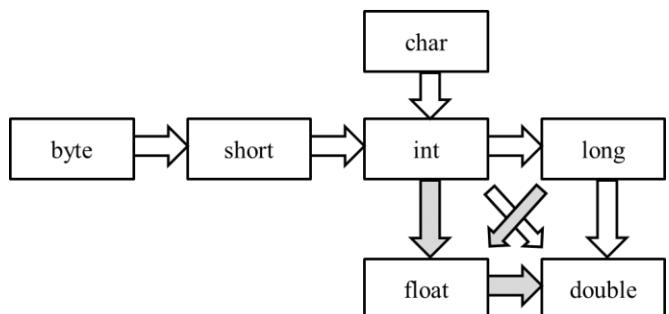
- повышающее** (разрешенное, неявное) преобразование;
- поникающее** (явное) приведение типа.

**Расширяющее (повышающее) преобразование:** результирующий тип имеет больший диапазон значений, чем исходный тип.

**Сужающее (поникающее) преобразование:** результирующий тип имеет меньший диапазон значений, чем исходный тип.

**Неявное (повышающее) преобразование:** повышающее преобразование осуществляется автоматически, даже в случае потери данных.

Серыми стрелками обозначены преобразования, при которых может произойти потеря точности.



### 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**.

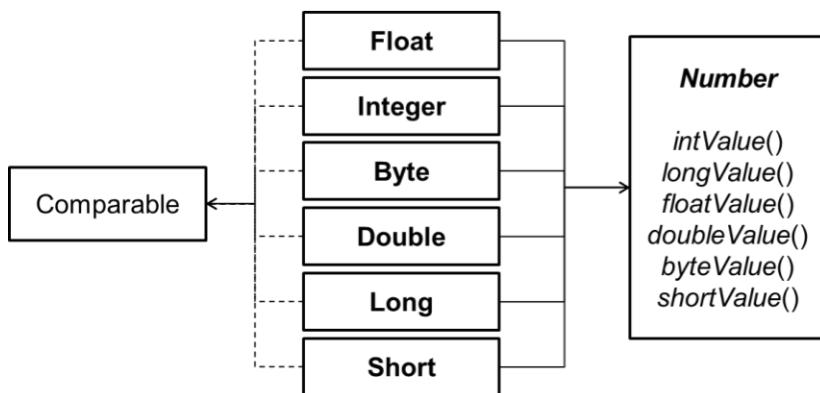
Арифметическое выражение над **byte**, **short** или **char** имеет тип **int**, поэтому для присвоения результата обратно в **byte**, **short** или **char** понадобится явное приведение типа.

### Классы-оболочки (**immutable**)

Кроме базовых типов данных широко используются соответствующие классы (wrapper классы):

- Boolean, Character, Integer, Byte, Short, Long, Float, Double.**
- Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы.
- Объекты этих классов представляют ссылки на участки динамической памяти, в которой хранятся их значения, и являются классами оболочками для значений базовых типов. Объекты этих классов являются константными.

Классы-оболочки (кроме **Boolean** и **Character**) являются наследниками абстрактного класса **Number** и реализуют интерфейс **Comparable**, представляющий собой интерфейс для работы со всеми скалярными типами.



Класс **Character** не наследуется от **Number**, так как ему нет необходимости поддерживать интерфейс классов, предназначенных для хранения результатов арифметических операций.

Класс **Character** имеет целый ряд специфических методов для обработки символьной информации.

У этого класса, в отличие от других классов оболочек, не существует конструктора с параметром типа **String**.

### АВТОУПАКОВКА/АВТОРАСПАКОВКА

В версии 5.0 введен процесс автоматической инкапсуляции данных базовых типов в соответствующие объекты оболочки и обратно (**автоупаковка**). При этом нет необходимости в создании соответствующего объекта с использованием оператора new.

```
Integer iob = 71;
```

**Автораспаковка** – процесс извлечения из объекта-оболочки значения базового типа. Вызовы таких методов, как **intValue()**, **doubleValue()** становятся излишними.

Допускается участие объектов в арифметических операциях, однако не следует этим злоупотреблять, поскольку упаковка/распаковка является ресурсоемким процессом.

```
public class IntegerCache {  
    public static void main(String[] args) {  
        Integer i1 = 10;  
        Integer i2 = 10;  
        System.out.println(i1 == i2); //true  
        i1 = 128;  
        i2 = 128;  
        System.out.println(i1 == i2); //false  
    }  
}
```

В классах **Long**, **Integer**, **Short** и **Byte** присутствует внутренний кеш ссылок на значения от -128 до 127.

При инициализации объекта класса-оболочки значением базового типа преобразование типов необходимо указывать явно.

Возможно создавать объекты и массивы, сохраняющие различные базовые типы без взаимных преобразований, с помощью ссылки на класс **Number**.

При автоупаковке значения базового типа возможны ситуации с появлением некорректных значений и непроверяемых ошибок.

## ПЕРЕДАЧА В МЕТОДЫ

В Java все передается по значению.

- В случае примитивных типов создается полная копия переменной в локальном стеке и далее производятся манипуляции над данной новой переменной.
- В случае ссылочного типа создается копия ссылки, ссылающаяся на тот же самый объект. При изменении по новой ссылке объекта, изменения имеют силу и из вне метода.
- Константный примитивный тип – создается копия примитивного типа, значение которой нельзя изменить.
- Константный ссылочный тип – создается константная копия ссылки и при этом мы можем изменять объект, но не можем изменить ссылку (поменять объект, на который она ссылается).

## VARARGS (IN SCOPE OF OVERLOADING/OVERRIDING) JAVA 5

Возникают ситуации, когда **заранее неизвестно количество** передаваемых экземпляров класса 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:

*(Tip... args)*

а в случае необходимости передачи параметров других типов (`vararg` должен быть написан обязательно в конце!)

*(Tip1 t1, Tip2 t2, TipN tn, Tip... 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[]...args, 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). Всели 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).*

```

3 public class Varargs {
4     public static void printArgCount(Object... args) { // 1
5         System.out.println("Object... args: " + args.length);
6     }
7     public static void printArgCount(Long... args) { // 2
8         System.out.println("Long... args: " + args.length);
9     }
10    // public static void printArgCount(long... args) { // 3
11    //     System.out.print("long... args: " + args.length);
12    // }
13    public static void main(String args[]) {
14        printArgCount(5, 7);
15    }

```

с long...args ругается  
и проще довести Integer[]  
до Object[], чем до Long[]

```

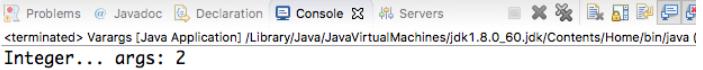
public static void printArgCount(Object... args) { // 1
    System.out.println("Object... args: " + args.length);
}
public static void printArgCount(Integer... args) { // 2
    System.out.println("Integer... args: " + args.length);
}
public static void printArgCount(int... args) { // 3
    System.out.print("int... args: " + args.length);
}
public static void main(String args[]) {
    printArgCount(5, 7); неопределенность 1 и 3
} требует убрать именно int...args

```

```

3 public class Varargs {
4     public static void printArgCount(Object... args) { // 1
5         System.out.println("Object... args: " + args.length);
6     }
7     public static void printArgCount(Integer... args) { // 2
8         System.out.println("Integer... args: " + args.length);
9     }
10 //    public static void printArgCount(int... args) { // 3
11 //        System.out.print("int... args: " + args.length);
12 //    }
13     public static void main(String args[]) {
14         printArgCount(5, 7);
15     }

```

Problems @ Javadoc Declaration Console Servers 

```

public static void main(String[] args) {
    print(2,32,5); подходит и то, и то
    Integer[] arr = { 1, 2, 3, 4, 5, 6, 7 };
    print(arr);
}

public static void print(Integer... integers) {
    System.out.println("Integer ...integers");
}

public static void print(int... integers) {
    System.out.println("Integer ...integers");
}

```

```

public static void main(String[] args) {
    print(2,32,5);
    Integer[] arr = { 1, 2, 3, 4, 5, 6, 7 };
    print(arr); Не работает привидение к массиву int
}

public static void print(int... integers) {
    System.out.println("Integer ...integers");
}

```

## GENERICS JAVA 5

### ПАРАМЕТРИЗОВАННЫЕ КЛАССЫ

С помощью шаблонов можно создавать параметризованные (родовые, generic) классы и методы, что позволяет использовать более строгую типизацию. Компилятор заменяет «фиктивные» типы на реальные создает соответствующий им объект.

#### Применение extends при параметризации

Объявление generic-типа в виде `<T>`, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса.

Переменные такого типа могут вызывать только методы класса `Object`.

Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```

public class OptionalExt<T extends Тип> {
    private T value;
}

```

Такая запись говорит о том, что в качестве типа `T` разрешено применять только классы, являющиеся наследниками (производными) класса `Тип`, и соответственно появляется возможность вызова методов ограничивающих (bound) типов.

### Метасимвол ?

Если возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом, то при определении метода следует применить метасимвол "?". (<?>)

Метасимвол также может использоваться с ограничением extends для передаваемого типа: <? extends Number>

### Использование extends с метасимволом ?

С помощью ссылки List<? extends T> невозможно добавлять элементы в коллекцию, так как невозможно гарантировать, что в список добавятся объекты допустимого типа.

Гарантируется только чтение объектов типа T или его подклассов.

### Использование super с метасимволом ?

При чтении из коллекции с помощью ссылки типа List<? super T> нельзя гарантировать тип возвращаемого объекта иным, кроме как тип Object, так как ссылка, параметризированная данным образом может ссылаться на коллекции, параметризованные типом T и его базовыми типами.

Добавление элементов в коллекцию элементов возможно, элементы должны иметь тип T или тип его подклассов.

```
List<? extends Doctor> list1 = new ArrayList<MedicalStaff>(); // error
List<? extends Doctor> list2 = new ArrayList<Doctor>();
List<? extends Doctor> list3 = new ArrayList<HeadDoctor>();

List<? super Doctor> list7 = new ArrayList<HeadDoctor>(); // error
List<? super Doctor> list6 = new ArrayList<Doctor>();
List<? super Doctor> list5 = new ArrayList<MedicalStaff>();
List<? super Doctor> list4 = new ArrayList<Object>();

list5.add(new Object()); // error
list5.add(new MedicalStaff()); // error
list5.add(new Doctor());
list5.add(new HeadDoctor());

Object object = list5.get(0);
MedicalStaff medicalDtaff = list5.get(0); // error
Doctor doctor = list5.get(0); // error
HeadDoctor headDoctor = list5.get(0); // error
```

## Параметризованные методы

Параметризованный (generic) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра.

<T Тип> Тип method(T arg) {} <T> Тип method(T arg) {}

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после extends. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

Параметризованные методы применяются когда необходимо разработать базовый набор операций, который будет работать с различными типами данных.

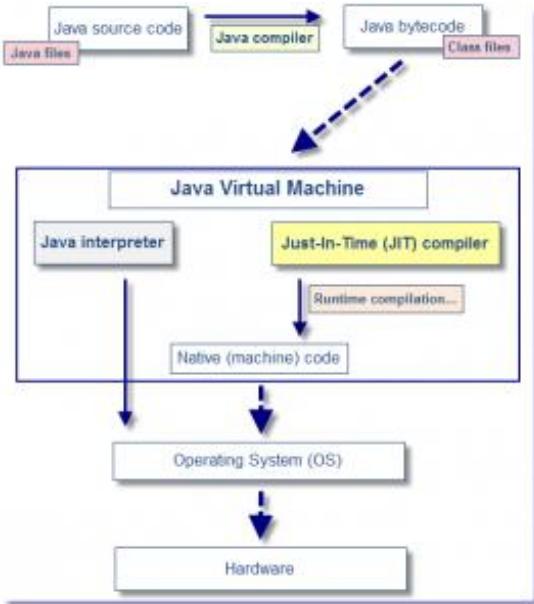
Описание типа всегда находится перед возвращаемым типом. Параметризованные методы могут размещаться как в обычных, так и в параметризованных классах.

Параметр метода может не иметь никакого отношения к параметру класса. Параметризованные методы можно перегружать.

**Ограничение при использовании параметризации:** параметризованные поля не могут быть статическими.

**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.



### What is JIT Compiler?

The Just In Time Compiler (JIT) concept and more generally adaptive optimization is well known concept in many languages besides Java (.Net, Lua, JRuby).

In order to explain what is JIT Compiler I want to start with a definition of compiler concept. According to wikipedia compiler is " a computer program that transforms the source language into another computer language (the target language)".

We are all familiar with static java compiler (javac) that compiles human readable .java files to a byte code that can be interpreted by JVM - .class files. Then what does JIT compile? The answer will given a moment later after explanation of what is "Just in Time".

According to most researches, 80% of execution time is spent in executing 20% of code. That would be great if there was a way to determine those 20% of code and to optimize them. That's exactly what JIT does - during runtime it gathers statistics, finds the "hot" code compiles it from JVM interpreted bytecode (that is stored in .class files) to a native code that is executed directly by Operating System and heavily optimizes it. Smallest compilation unit is single method. Compilation and statistics gathering is done in parallel to program execution by special threads. During statistics gathering the compiler makes hypotheses about code function and as the time passes tries to prove or to disprove them. If the hypothesis is dis-proven the code is deoptimized and recompiled again.

The name "Hotspot" of Sun (Oracle) JVM is chosen because of the ability of this Virtual Machine to find "hot" spots in code.

### What optimizations does JIT?

Let's look closely at more optimizations done by JIT.

Inline methods - instead of calling method on an instance of the object it copies the method to caller code. The hot methods should be located as close to the caller as possible to prevent any overhead.

- Eliminate locks if monitor is not reachable from other threads
- Replace interface with direct method calls for method implemented only once to eliminate calling of virtual functions overhead
- Join adjacent synchronized blocks on the same object
- Eliminate dead code
- Drop memory write for non-volatile variables

- Remove prechecking NullPointerException and IndexOutOfBoundsException
- Et cetera

When the Java VM invokes a Java method, it uses an invoker method as specified in the method block of the loaded class object. The Java VM has several invoker methods, for example, a different invoker is used if the method is synchronized or if it is a native method.

The JIT compiler uses its own invoker. Sun production releases check the method access bit for value ACC\_MACHINE\_COMPILED to notify the interpreter that the code for this method has already been compiled and stored in the loaded class. JIT compiler compiles the method block into native code for this method and stores that in the code block for that method. Once the code has been compiled the ACC\_MACHINE\_COMPILED bit, which is used on the Sun platform, is set.

How do we know what JIT is doing in our program and how can it be controlled?

First of all to disable JIT Djava.compiler=NONE parameter can be used.

There are 2 types of JIT compilers in Hotspot - one is used for client program and one for server (-server option in VM parameters). Program, running on server enjoys usually from more resources than program running on client and to server program top throughput is usually more important. Hence JIT in server is more resource consuming and gathering statistics takes more time to make the statistics more accurate. For client program gathering statics for a method lasts 1500 method calls, for server 15000. These default values can be changed by -XX:CompileThreshold=XXX VM parameter.

In order to find out whether default value is good for you try enabling "XX:+PrintCompilation" and "-XX:-CITime" parameters that print JIT statistics and time CPU spent by JIT.

## Benchmarks

Most of the benchmarks show that JITed code runs 10 to 20 times faster than interpreted code. There are many benchmarks done. Below given result graphs of two of them:

Its worth to mention that programs that run in JIT mode, but are still in "learning mode" run much slower than non JITed programs.

## Drawbacks of JIT

JIT Increases level of unpredictability and complexity in Java program. It adds another layer that developers don't really understand. Example of possible bugs - 'happens before relations' in concurrency. JIT can easily reorder code if the change is safe for a program running in single thread. To solve this problem developers make hints to JIT using "synchronized" word or explicit locking.

Increases non heap memory footprint - JITed code is stored in "Code Cache" generation.

## Advanced JIT

JIT and garbage collection.

For GC to occur program must reach safe points. For this purpose JIT injects yieldpoints at regular intervals in native code.

In addition to scanning of stack to find root references, registers must be scanned as they may hold objects created by JIT

# LESSON 3

## OBJECT (DIFFERENT WAYS TO CREATE OBJECT, WHAT HAPPENS INSIDE OF CREATION PROCESS)

Объект в Java – некоторая сущность, которая имеет состояние и которой присуще некоторое поведение (**state and behavior**). Объект – экземпляр класса, который хранит свое состояние в виде полей (fields/variables), а поведение описывается через методы. Методы оперируют внутренним состоянием объекта и являются главным механизмом взаимодействия между объектами.

Класс – группа объектов, имеющих общие свойства. Класс – шаблон (template/blueprint), на основе которого создаются объекты. Класс – логическая сущность (не может быть материализировано).

### МЕТОДЫ СОЗДАНИЯ ОБЪЕКТОВ В JAVA

#### 1. Ключевое слово new

```
Employee emp1 = new Employee();
```

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)

```
Employee emp2 = (Employee) Class.forName("org.programming.mittra.exercises.Employee").newInstance();  
Или  
Employee emp2 = Employee.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  
Test t = (Test)cl.loadClass("com.javainterviewpoint.Test").newInstance();
```

LoadClass аналогичен 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 способ создания объекта)

```
Constructor<Employee> constructor = Employee.class.getConstructor();
Employee emp3 = constructor.newInstance();
```

**В байткоде метод конвертируется в вызов invokevirtual, т.е. сам метод управляет созданием объекта.**

<u>Class.newInstance()</u>	<u>Class.newInstance()</u>
Лежит в пакете <i>java.lang</i>	Лежит в пакете <i>java.lang.reflection</i>
Для создания объекта используют reflection API	
Может вызывать только конструктор без параметров.	Может вызывать любой конструктор, независимо от числа параметров.
Необходима видимость конструктора.	Может вызывать даже private конструктор в некоторых случаях.
Выбрасывает любое исключение (checked или unchecked), выбрасываемое конструктором.	Всегда об оборачивает исключение в <i>InvocationTargetException</i> .
Внутренне вызывает <a href="#">Class.newInstance()</a> .	
	Используется фреймворками (Spring, Guava, Jackson и др.)

#### 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 сериализации и 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.**

## CLASS OBJECT (METHODS!)

Class Object – базовый класс для всех классов в Java, т.е. стоит на вершине иерархии наследования. Даже если класс не наследуется ни от какого класса, то он неявно является наследником класса Object. *НО это не относится к интерфейсам, т.к. интерфейсы не наследуются от классов.*

`obj.getClass().getSuperclass()` для класса Object вернет null, что в само просто доказывает то, что он базовый для всех классов Java.

Всего в Object 11 публичных методов, 5 обычных и 6 с нативной реализацией.

**public int hashCode()**

Возвращает значение `hashcode` (битовая строка фиксированной длины – в общем случае, int число – в java) java объекта.

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();
```

При вычислении значения по умолчанию используется алгоритм Park-Miller RNG. В основу его работы положен генератор случайных чисел, т.е. при каждом запуске программы у объекта будет разный хеш-код. Т.е. при каждом создании объекта, будем получать разные хеш-коды.?????

**public boolean equals(Object obj)**

Метод сравнивает содержимое двух объектов.

Он тесно связан с методом `hashCode()`, т.к. оба работают с содержимым полей.

Если содержимое двух объектов одинаковое, то и хеш-коды должны быть одинаковые.

		<p>Исходный код в классе Object (сравнение только ссылок):</p> <pre><code>public boolean equals(Object obj) {     return (this == obj); }</code></pre>
<b>protected Object clone() throws CloneNotSupportedException</b>		<p>Создает и возвращает точную копию объекта.</p> <ol style="list-style-type: none"> <li>1. <code>x.clone() != x</code> гарантирует, что клонируемый объект имеет отдельный участок памяти.</li> <li>2. <code>x.clone().getClass() == x.getClass()</code> (рекомендация) они должны иметь одинаковый тип класса, но это не обязательное условие.</li> <li>3. <code>x.clone().equals(x)</code> (рекомендация) должны иметь одинаковое содержимое.</li> </ol> <p>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.</p> <p>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.</p>
<b>public String toString()</b>		<pre><code>public String toString() {     return getClass().getName() + "@" + Integer.toHexString(hashCode()); }</code></pre>
<b>public final Class getClass()</b>		<p>Возвращает объект класса Class для этого объекта. Объект класса Class представляет собой методанные об определенном классе.</p>
<b>public final void notify()</b>		<p>Метод пробуждает один поток, который ждет монитор на данном объекте.</p>
<b>public final void notifyAll()</b>		<p>Метод пробуждает все потоки, которые ожидают монитор на данном объекте.</p>
<b>public final void wait(long timeout) throws InterruptedException</b>		<p>Заставляет текущий поток ждать, пока либо другой поток не вызовет <code>notify()</code> метод или <code>notifyAll()</code> метод этого объекта или указанное кол-во времени не пройдет.</p>

<b>public final void wait(long timeout,int nanos) throws InterruptedException</b>	
<b>public final void wait() throws InterruptedException</b>	Заставляет текущий поток ждать пока другой поток не вызовет notify() или notifyAll() на этом объекте.
<b>protected void finalize() throws Throwable</b>	<p>Во-первых, нет гарантии, что он будет вызван, т.к. где-то может остаться ссылка на объект.</p> <p>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 происходит в 2 этапа). Стоит отметить, что в очередь на финализацию попадают только те объекты, в которых переопределен метод finalize.</p> <p>При переопределении метода finalize() не вызываются все методы суперклассов, подобно конструкторам (not automatically chained), поэтому необходимо самостоятельно вызывать метод суперкласса при переопределении.</p> <p style="color: red;">Вызывается только один раз потоком GC и если объект будет использован в блоке finalize, то данный метод не будет вызван.</p> <p>Любое исключение, выброшенное в finalize методе, игнорируется GC и не перекидывается никуда дальше.</p> <p>Существует некоторый способ немного увеличить вероятность вызова finalize метода: System.runFinalization() и Runtime.getRuntime().runFinalization()</p>

## OOP PRINCIPLES (DEFINITION, EXAMPLES OF USAGE)

**Наследование** – один из принципов ООП. Наследование в Java – процесс, посредством которого один класс может повторно использовать методы и поля другого класса. Производный (derived/sub class) использует поля и методы базового/супер-класса.

Наследование выражается через отношение **IS-A (является)**, которое также называется отношением Родитель-Потомок. Родительский класс может иметь сколько угодно потомков, но дочерний класс может иметь лишь одного родителя.

Наследование бывает:

- одиночное
- множественное (запрещено в Java)

Пример наследование: Средство передвижения (скорость, механизм) и Машина (мощность, тип и т.д.).

**Полиморфизм** – способность использовать одно и то же имя для решения схожих, но технически разных задач (принимать более чем одну форму). Подклассы класса могут определять свое собственное поведение и имеют доступ к той же функциональности базового класса.

В Java существуют 2 способа обеспечения полиморфизма: перегрузка и переопределение.

Overloading - возможность иметь в классе методы с одинаковым именем, но разным числом параметров.

Overriding - возможность определять в подклассах методы с сигнатурой идентичной сигнатуре родительских методов, но разной реализацией.

*Чаще всего инкапсуляция выполняется посредством сокрытия информации (всех внутренних деталей), не влияющей на внешнее поведение. Инкапсуляция определяет четкие границы между различными абстракциями. Обычно скрываются и внутренняя структура объекта, и реализация его методов.*

**Инкапсуляция** – процесс, объединяющий код и данные вместе внутри единого элемента, посредством которого происходит сокрытие реализации деталей от пользователя. Если модификатор доступа переменной – `private`, тогда она может быть доступна только в пределах класса.

Преимущества инкапсуляции:

1. Улучшает поддержку кода. Доступ к переменным класса возможен только через методы самого класса, которые можно изменять, но из вне класса не нужно будет ничего менять.
2. Улучшает управление данными. Например, если какое-то поле не может иметь значение больше, чем 500, то можно в `set` это проверять.

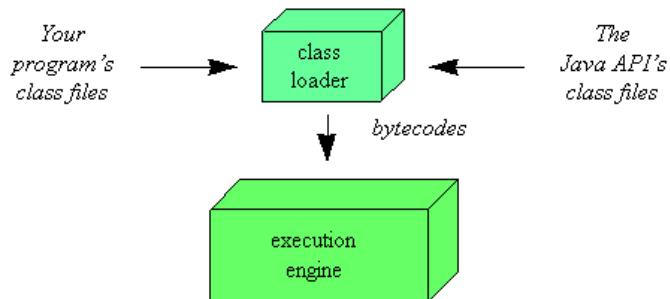
Можно создавать `read-only` (без сеттеров) и `write-only` поля (без геттеров).

*Абстрагирование является упрощенным описанием или изложением системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения, а опускает те, которые на данный момент несущественны.*

**Абстракция** - это приданье объекту характеристик, которые чётко определяют его концептуальные границы, отличая от всех других объектов. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов, подобно тому, как функциональная абстракция разделяет способ использования функции и деталей её реализации в терминах более примитивных функций, таким образом, данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня.

## CLASSLOADING (TYPES, PROCESS, EXCEPTIONS, DYNAMIC LOADING?)

**Classloader** – специальный объект, который грузит другие классы. В JVM загрузчики классов ответственны за импорт бинарных данных, которые определяет выполняющаяся программа. Исполнительной системе java(java runtime system) нет нужды иметь информацию о файлах и файловой системе благодаря наличию класслоадеров.



При старте JVM грузит Bootstrap classloader, который написан на языке С. Он в свою очередь загружает Extension Class Loader и System Class Loader и размещает их в памяти. При этом есть возможность указать родительского загрузчика класса (Every Java class loader has a parent class loader, defined when a new class loader is instantiated or set to the virtual machine's system default class loader). Все загруженные classloaders расположены в Method Area (PermGen).

Загрузчик классов (Class loader) считывает .class файл, генерирует соответствующие бинарные данные и сохраняет это в области методов. Для каждого .class файла, JVM хранит следующую информацию в области методов:

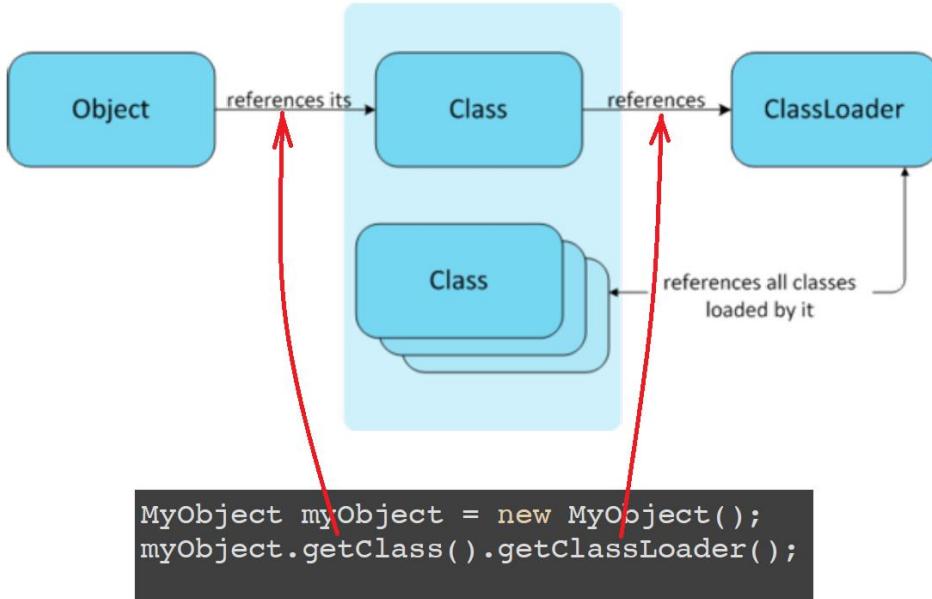
- Полное имя загружаемого класса и его непосредственные родительские классы.
- Определяет является ли .class файл Java классом, интерфейсом или Enum.
- Определяет поля, переменные и информацию о методах и т.д.

После загрузки .class файла, JVM создает объекты класса Class в области PermGen(java 7) или MetaSpace (java 8) памяти. Для каждого загруженного класса (.class), создается только один объект класса Class. В JVM имеется несколько типов класслоадеров.

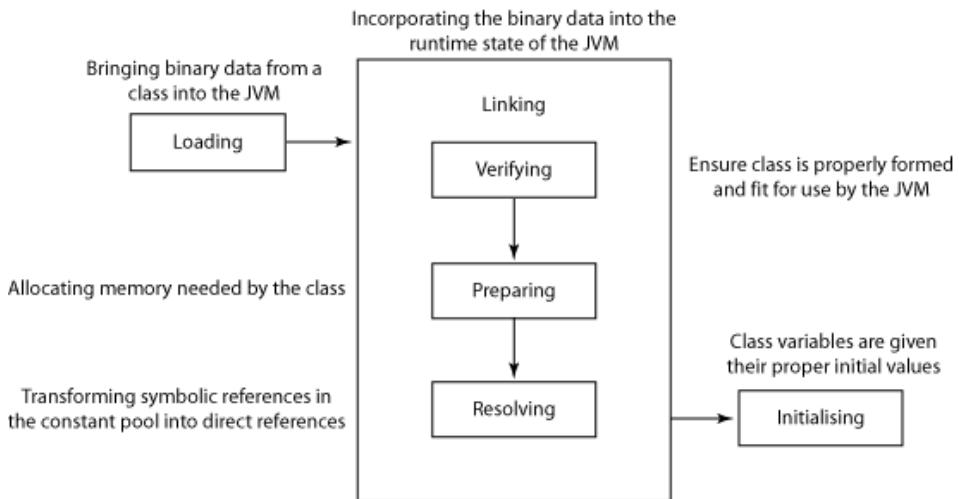
Далее, прежде, чем будет вызван public static void main(String[]), класс связывается и инициализируется (linked and initialized). Выполнение этого метода, в свою очередь, будет приводить к загрузке, связыванию и инициализации дополнительных классов и интерфейсов.

### Classloader Reference

Все загруженные классы содержат ссылку на загрузчик классов, который их загрузил. В свою очередь, загрузчик классов также содержит ссылки на все классы, которые он загрузил.



## Class Loading phases



**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 и имеют правильную major и minor версии + анализируется CAFEBABE. 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**. Необходимо помнить что этот объект является типом Class, который определен в пакете java.lang. Объект этого класса программист может использовать для получения информации 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 суперинтерфейса. 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 из 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 любой 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(java8).

**Resolving** не является обязательным этапом, который включает в себя проверку ссылок путем загрузки соответственных 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 ConstantPool) прямыми (реальные адреса объектов классов, которые уже загружены и лежат в Heap). При поиске в области методов обнаруживает зависимые сущности класса.

До этого он знал то, что A должен грузиться до B. На это этапе он заменяет на конкретную ссылку.

**Initialization** класса или интерфейса состоит в выполнении инициализирующего метода класса или интерфейса <`cinit`> (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` объекта `ClassLoader` стадия Initialization не выполняется. 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 будут сгенерированы компилятором и будут содержать в себе инструкции из определенных конструкторов, определенных в классе.) Может быть вызван только JVM.

## CLASSLOADER TYPES

В виртуальной машине Java существует несколько загрузчиков классов, выполняющих разные роли. Каждый загрузчик делегируется родительскому загрузчику (кроме Bootstrap classloader, который является самым верхним загрузчиком), который его загрузил.

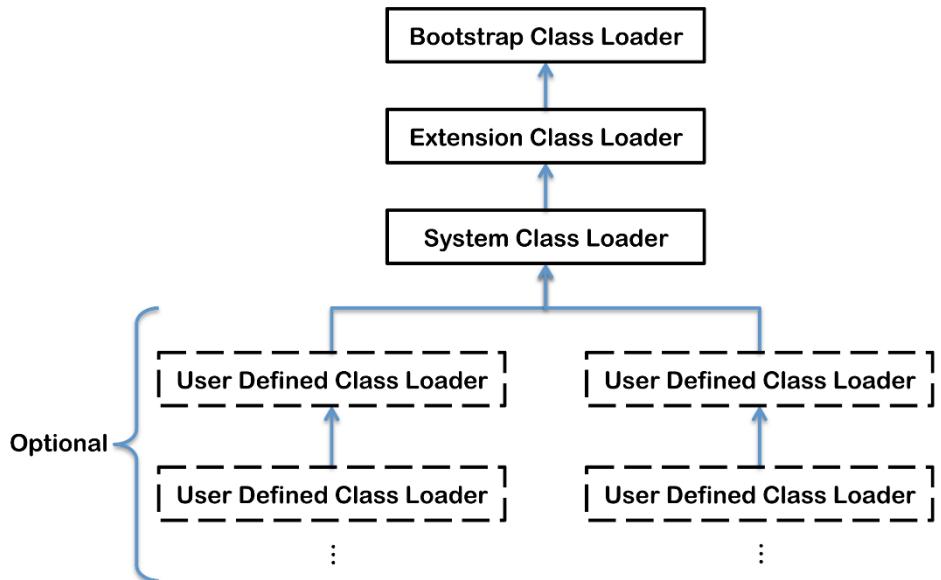
- **Primordial/Bootstrap Classloader** – реализован как нативный код, т.к. он создается в момент загрузки самой виртуальной машины Java (очень рано). Он же грузит Extension Classloader и System Classloader. Загрузчик реализован на языках C, C++. Данным загрузчиком загружаются основные классы Java из директории **JAVA\_HOME/jre/lib** (этот путь так же называют bootstrap path), включая **rt.jar**. Поэтому, попытка получения загрузчика у классов `java.*` всегда заканчивается null'ом. 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** является загрузчиком 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 загрузки (**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).

**Какой класс будет реально загружен, если в **\$JAVA\_HOME/lib/ext** и в **CLASSPATH** есть классы с одинаковыми полными именами?**

Правильно, класс из **\$JAVA\_HOME/lib/ext**, а на **CLASSPATH** никто не посмотрит. Хотя с ним тоже не все просто: классы загружаются в том порядке, в котором они были указаны в **CLASSPATH**. По этому если указать два jar-файла, к примеру `A.jar` и `B.jar`, содержащие одинаковые классы, то в память загрузится класс из `A.jar`, а класс из `B.jar` будет пропущен.

**Note:** каждый ClassLoader видит «свои» классы и классы «родителя». Классы «потомков», ни тем более классы «паралельных» 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.

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 пару, состоящую из полного имени класса и загрузчика**.



**Класс ClassLoader является абстрактным классом.**

getParent()	• Get parent class loader.
loadClass(String name)	• Load the class by given name, return a class instance.
findClass(String name)	• Find the class by given name, return a class instance
findLoadedClass(String name)	• Find the loaded class by given name
defineClass(String name, byte[] b, int off, int len)	• Generate class instance from its binary representation • Final function
resolveClass(Class<?> c)	• Link given class

```

public abstract class ClassLoader {
    public Class loadClass(String name);
    protected Class defineClass(byte[] b);

    public URL getResource(String name);
    public Enumeration getResources(String name);

    public ClassLoader getParent()
}

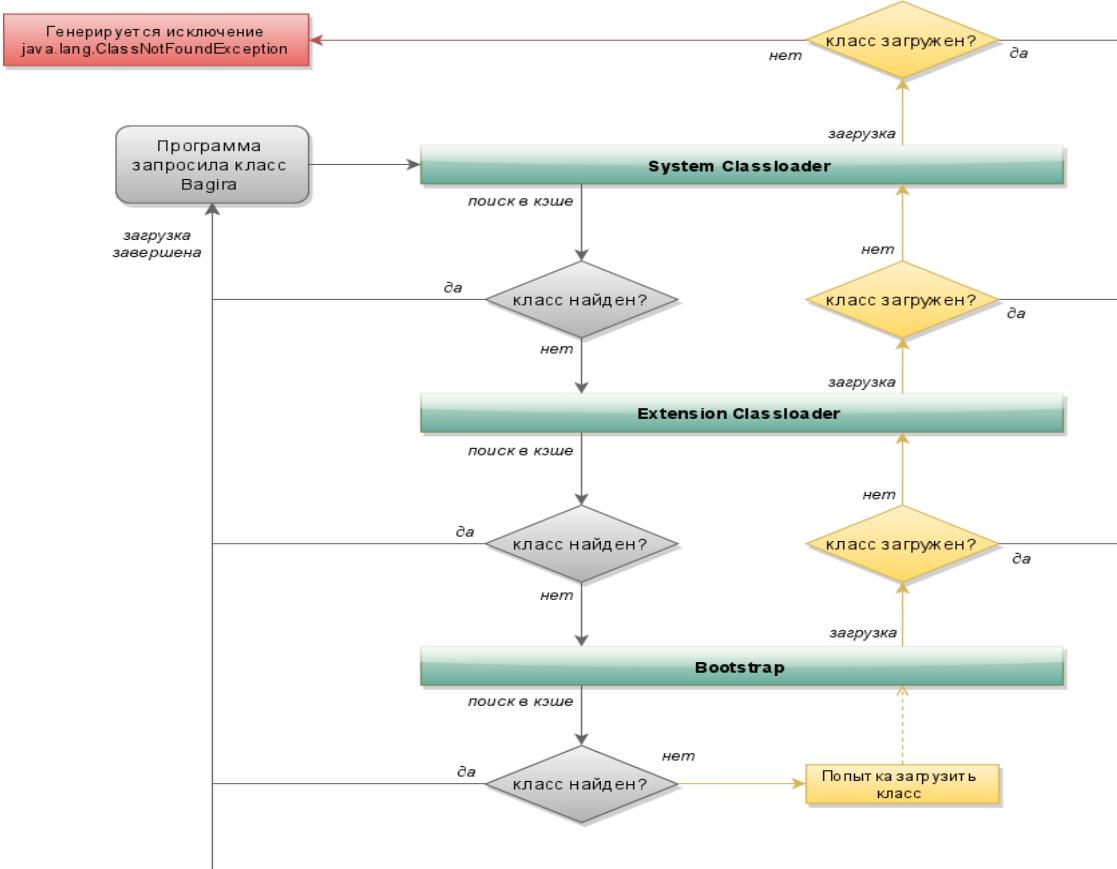
```

### Faster Class Loading

A feature called Class Data Sharing (CDS) was introduced in HotSpot JVM from version 5.0. During the installation process of the JVM the installer loads a set of key JVM classes, such as rt.jar, into a memory-mapped shared archive. CDS reduces the time it takes to load these classes improving JVM start-up speed and allows these classes to be shared between different instances of the JVM reducing the memory footprint.

### ПРИНЦИП ДЕЛЕГИРОВАНИЯ

JVM follows the principle of hierarchical delegation during class loading. A request for class loading starts at the Application ClassLoader, which delegates the request to the parent Extension Class Loader. This loader then delegates the request to the Boot-strap Class Loader. If the class is found in the boot-strap path (found in rt.jar), the class is loaded; otherwise, the request is sent back to the Extension Class Loader, which then checks the jre/lib/ext directory and finally attempts to load the class from the Java classpath. If the Application Class Loader fails to load the class, a java.lang.ClassNotFoundException is thrown.



#### Алгоритм загрузки класса:

1. Реквест на загрузку поступает в ApplicationClassLoader. Он проверяет свой кэш на наличие объекта этого класса. Если такой класс уже был загружен возвращается ссылка из кэша. Если нет, то реквест передается Extension класслоадеру.
2. Получив реквест Extension класслоадер проверяет свой кэш, если класс уже был загружен, то возвращается ссылка на него, если нет, то реквест передается BootStrap класслоадеру
3. Получив реквест BootStrap класслоадер проверяет свой кэш, если класс уже был загружен до этого, то возвращается ссылка из кэша, если нет, то этот класслоадер пытается загрузить класс из директории (JAVA\_HOME\jre\lib)(rt.jar). Если класс был загружен то ссылка помещается в кэш и возвращается для дальнейшей обработки. Если загрузить класс не удалось, то реквест передается Extension класслоадеру
4. Extension класслоадер пытается загрузить класс из директории ext (jre\lib). Если класс был загружен, то ссылка помещается в кэш и возвращается для дальнейшей обработки. Если нет, то реквест передается Application класслоадеру.
5. Application класслоадер пытается загрузить класс из CLASSPATH. Если класс был успешно загружен, то ссылка помещается в кэш и возвращается. Если класс загрузить не удалось, то будет выброшено ClassNotFoundException.

```

protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class c = findLoadedClass(name); Был ли ранее загружен?
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false); Если есть родитель
                } else {
                    c = findBootstrapClassOrNull(name); Если нет родителя
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
        }
        if (c == null) {
            // If still not found, then invoke findClass in order
            // to find the class.
            long t1 = System.nanoTime(); Если никто из родительских
            c = findClass(name); загрузчиков не нашел класс или не
                               смог загрузить, то сам загружает
        }
    }
}

```

```

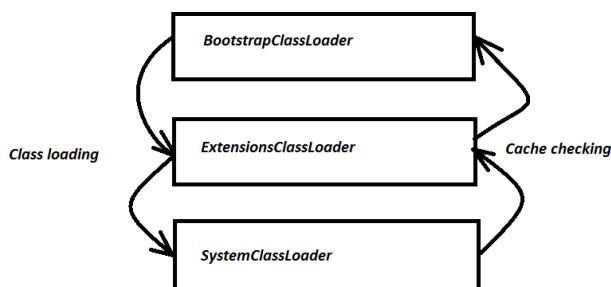
protected Class findClass(String name) throws ClassNotFoundException {
    Class result = findLoadedClass(name); Если уже загружен
    if (result != null) {
        logger.info("% Class " + name + " found in cache");
        return result;
    }
    File f = findFile(name.replace('.', '/') + ".class");
    logger.info("% Class " + name + f == null ? "" : " found in " + f);
    if (f == null) {
        return findSystemClass(name);
    }
    try {
        byte[] classBytes = loadFileAsBytes(f);
        result = defineClass(name,
                             classBytes, 0, classBytes.length);
    } catch (IOException e) {
        throw new ClassNotFoundException(
            "Cannot load class " + name + ":" + e);
    } catch (ClassFormatError e) {
        throw new ClassNotFoundException(
            "Format of class file incorrect for class "
            + name + ":" + e);
    }
    return result;
}

```

## 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!**

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                long t1 = System.nanoTime();
                c = findClass(name);
            }
        }
    }
}
```

## Когда классы загружаются ClassLoader?

1. Класс загружается в память JVM когда происходит выполнение нового байткода, например выполнение следующего кода: `FooClass f = new FooClass();`
2. Когда в осуществляется ссылка на статическое содержимое класса, например `System.out`.

## Exceptions

Exception	Description
<b>ClassCastException</b>	<p>Исключение может быть выброшено, если:</p> <ul style="list-style-type: none"> <li>Произведена попытка явна привести объект к типу, экземпляром которого он не является</li> <li>ClassLoader, который загрузил класс исходного объекта отличается от ClassLoader, который загрузил целевой класс.</li> </ul> <p>При попытке <b>явного приведения</b> типов ClassLoader проверяет:</p> <ul style="list-style-type: none"> <li><b>Для обычных объектов</b>(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</li> <li><b>Для массивов.</b> Класс к которому мы пытаемся привести должен быть типа массива или Object или Cloneable или Serializable</li> </ul> <p>Если одно из этих правил нарушается, то выкидывается это исключение.</p>
<b>ClassNotFoundException</b>	<p>Исключение выкидывается на этапе загрузки класса.</p> <p>Объект этого исключения создается если приложение пытается загрузить класс используя строку, представляющую его имя используя:</p> <ul style="list-style-type: none"> <li>forName() method in class Class</li> <li>findSystemClass method() in class ClassLoader</li> <li>loadClass() method in class ClassLoader</li> </ul> <p>Однако класс файл для класса с таким именем не может быть найден. Выкидывается, когда попытка явно загрузить класс проваливается.</p>
<b>NoClassDefFoundError</b>	<p>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. (<i>При компиляции класс был, но на этапе выполнения байт код недоступен(либо файл был перемещен или удален)</i>)</p>
<b>UnsatisfiedLinkError</b>	
<b>UnsatisfiedLinkError</b>	<p>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().</p> <ul style="list-style-type: none"> <li>Для классов загруженных BootStrap класслоадером просматривается директория на которую указывает sun.boot.library.path.</li> <li>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)</li> <li>Для классов, загруженных Application класслоадером просматривается директория на которую указывает sun.boot.library.path, а затем java.library.path</li> </ul>
<b>ClassCircularityError</b>	<p>Класс или интерфейс не могут быть загружены, так как это будут их собственные суперкласс или суперинтерфейс. Это исключение может быть выброшено на этапе resolving стадии linking. Является довольно странным исключением, т.к. Java компилятор гарантирует недопущение</p>

	такой ситуации. Может появиться если отдельно скомпилировать классы, а потом собрать их вместе.
<b>ClassFormatError</b>	Исключение выкидывается если <b>бинарные данные, которые представляют собой скомпилированный класс или интерфейс плохо форматированы</b> . (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 перекомпиляции.
<b>ExceptionInInitializerError</b>	Ошибка выбрасывается на стадии инициализации процесса загрузки класса. Ошибка выкидывается, если: <ul style="list-style-type: none"> <li>Выполнение статического блока инициализации оборвано 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 вместе исключения E</li> <li>Если JVM попыталась создать новый объект типа ExceptionInInitializerError, но ей это не удалось из-за OutOfMemoryError, то выбрасывается объект OutOfMemoryError.</li> </ul>
<b>VerifyError</b>	Выбрасывается на этапе верификации, если хорошо форматированный класс файл содержит внутренние противоречия или проблемы безопасности. (Класс файл не прошел верификацию)

<https://www.ibm.com/developerworks/library/j-dclp2/>

#### Что же такое статическая и что такое динамическая загрузка класса?

Статическая загрузка класса происходит при использовании оператора "new".

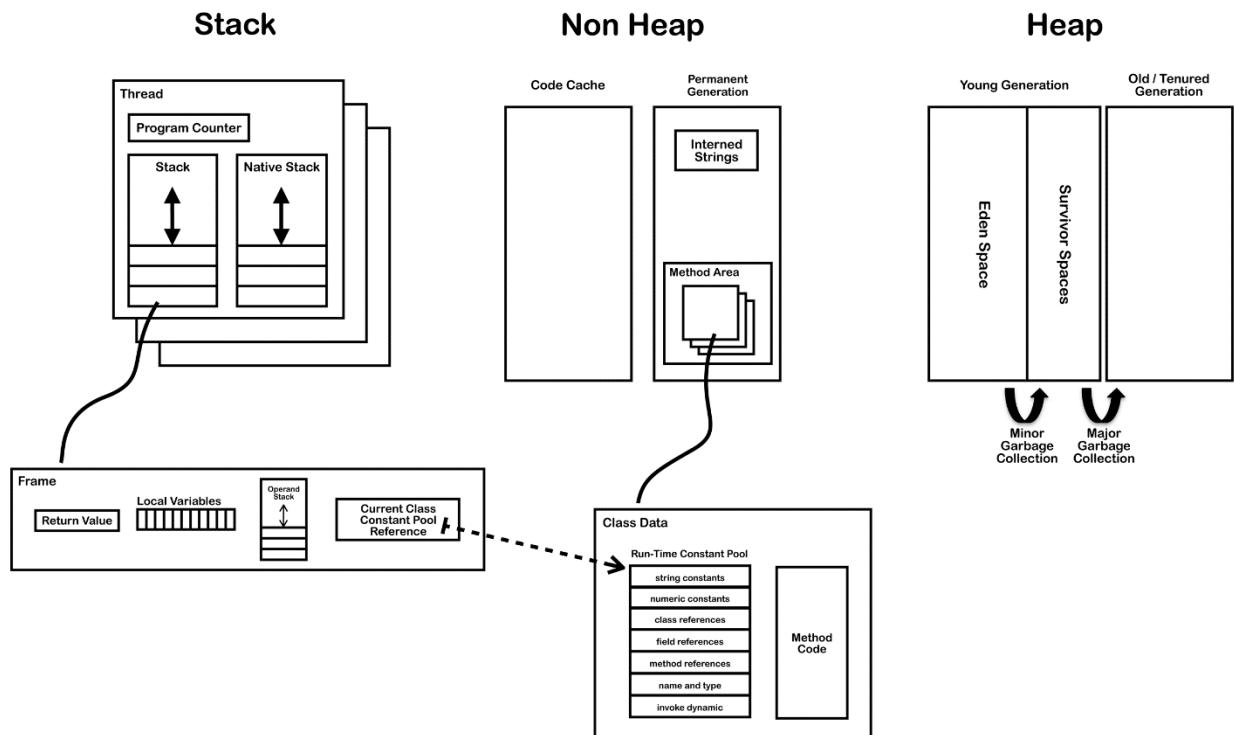
Динамическая загрузка происходит "на лету" в ходе выполнения программы с помощью статического метода класса Class.forName(имя класса). Для чего нужна динамическая загрузка? Например мы не знаем какой класс нам понадобится и принимаем решение в ходе выполнения программы передавая имя класса в статический метод forName().

# LESSON 4 (JAVA MEMORY MODEL)

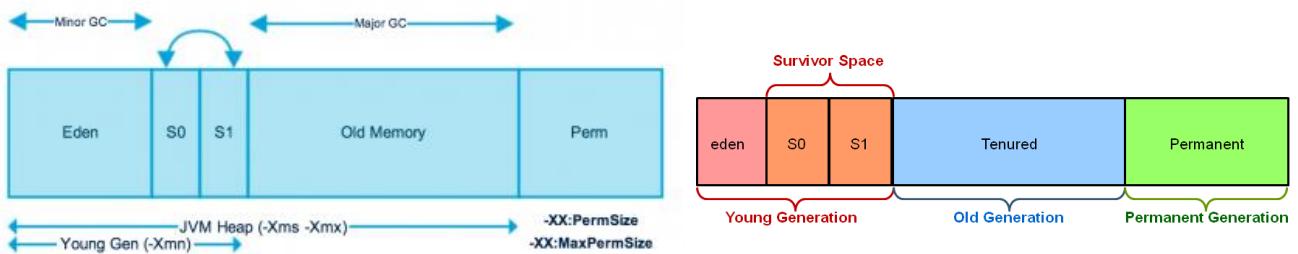
## JMM

JMM – абстракция, которая гарантирует, что один и тот же код будет выполнен одинаково на всех платформах.

Память виртуальной машины разделена на несколько частей.



## 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**.

## YOUNG GENERATION

**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)

**Old Generation (Tenured)** – содержит долго живущие объекты (крупные высокуровневые 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.

## STOP THE WORLD EVENT (STW)

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 ответственным 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.

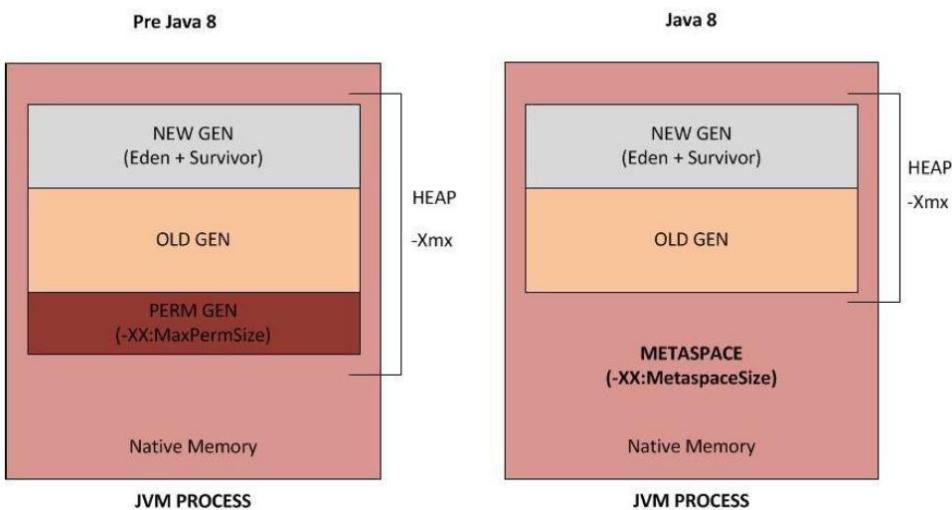
Длительность работы сборщика мусора зависит от выбранной им стратегии. Именно поэтому важно следить и корректировать работу GC.

## PERMANENT GENERATION OR RERM GEN (NON HEAP)

**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.

## JAVA 8 MEMORY MANAGEMENT



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 предупредительное сообщение о том, что они более не действуют), и вместо них вводится optionalный 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 объема 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.

## METHOD AREA

**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).

## MEMORY POOLS

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.

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. String Pool реализован в виде HashMap.

## CODE CACHE (NON HEAP)

**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 отдельны 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 заверщен 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. Фреймы могут быть размещены близко со стеком или же могут размещены в Heap, или, иногда, и там и там.

## STACK FRAME

Stack Frame разделен на три части:

- **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.

Для 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 класса файла для каждого метода.**

Компилятор размещает параметры в область локальных переменных и параметров методов в порядке их декларирования. Например:

```
class Example3a {  
    public static int runClassMethod(int i, long l, float f,  
        double d, Object o, byte b) {  
        return 0;  
    }  
    public int runInstanceMethod(char c, double d, short s,  
        boolean b) {  
        return 0;  
    }  
}
```

runClassMethod()			runInstanceMethod()		
index	type	parameter	index	type	parameter
0	int	int i	0	reference	hidden this
1	long	long l	1	int	char c
3	float	float f	2	double	double d
4	double	double d	4	int	short s
6	reference	Object o	5	int	boolean b
7	int	byte b			

- **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:

```
iload_0    // push the int in local variable 0
iload_1    // push the int in local variable 1
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
```

	before starting	after iload_0	after iload_1	after iadd	after istore_2
local variables	0 100 1 98 2 198				
operand stack		100	100 98	198	

Первые две инструкции push int значения переменных в нулевую и первую позиции тека операндов. Далее iadd 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 области локальных переменных.

- **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 пули 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 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>

## 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: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.
-XX:NewRatio	For providing ratio of old/new generation sizes. The default value is 2.

Итого: 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, но его ссылка помещается в стек.

Heap Area	Stack Area
Используется всеми частями приложения.	Используется только одним потоком исполнения программы.
Всякий раз при создании объекта он хранится в куче, а ссылка на него в стеке.	Хранит только локальные переменные примитивных типов и ссылки на объекты в куче.
Объекты в куче доступны из любой точки программы.	Стековая память не может быть доступна для других потоков.
	Управление памятью: по схеме LIFO
Память в куче живет с самого начала и до конца работы программы.	Стековая память существует лишь какое-то время работы программы, во время которого выполняется метод.
Можно определить начальный и максимальный размер памяти в куче: <b>-Xms</b> и <b>-Xmx</b> (опции JVM)	Размер памяти: <b>-Xss</b>
Если память кучи заполнена: <b>java.lang.OutOfMemoryError: Java Heap Space</b>	Если память стека заполнена: <b>java.lang.StackOverflowError</b>
Для кучи выделено больше памяти, чем для стека.	Из-за простоты распределения память (LIFO), стековая память работает намного быстрее кучи.

## GARBAGE COLLECTOR

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. Когда ссылка уничтожается, счетчик уменьшается. Если значение счетчика равно нулю, - объект можно считать мусором и память можно очистить.

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.

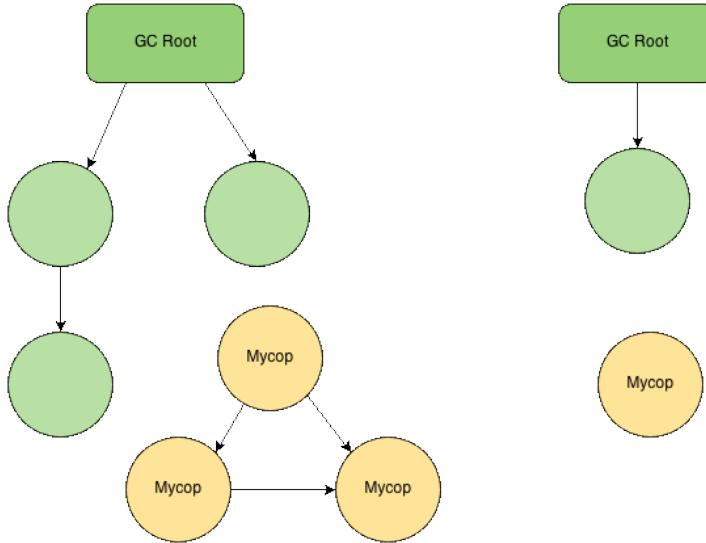
В общем, Reference counting редко используется из-за недостатков. Во всяком случае HotSpot VM его не использует. По этому мы можем отложить в памяти, что такой подход есть, и продолжить дальше.

#### Tracing

В "Tracing" главная идея состоит в мысли: "Живые объекты - 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 не возможно 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:

- Объектыalloцируются 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 alloцироваться они alloцируются 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?**

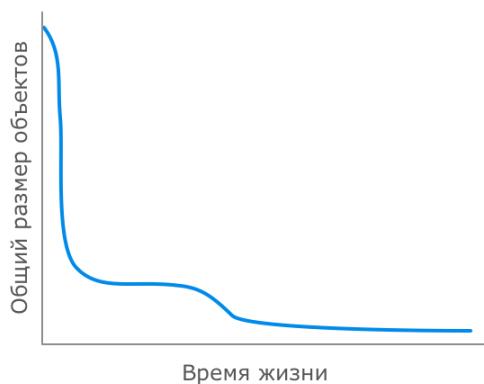
Сборщики мусора HotSpot VM используют подход "Generational Garbage Collection". Как мы увидим, этот подход позволяет использовать разные алгоритмы для разных этапов сборки мусора. Это позволяет использовать наиболее подходящий алгоритм.

Было замечено, что большинство приложений удовлетворяют двум правилам (weak generational hypothesis):

- **Большинство alloцированных (только что созданных) объектов быстро становятся мусором. (гипотеза о поколениях)**
- **Существует мало связей между объектами, которые были созданы в прошлом и только что alloцированными 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 связаннысти 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.

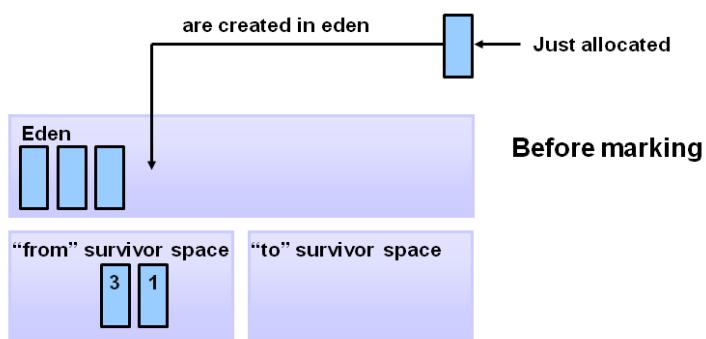
Рассмотрим такую ситуацию: У нас есть молодой объект A и ссылающийся на него объект B, уже заслуживший место в старшем поколении. В какой-то момент времени оба этих объекта стали нам не нужны и мы обнулили все имеющиеся у нас ссылки на них. Очевидно, объект A можно было бы удалить в ближайшую малую сборку 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 A 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

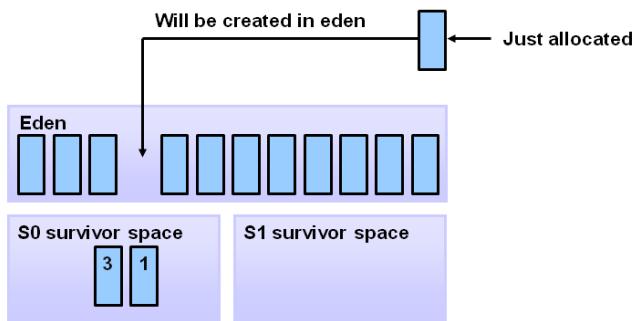
1. Все новосозданные объекты размещаются в Eden. Обе области Survivor пусты (в самом начале).

#### Object Allocation



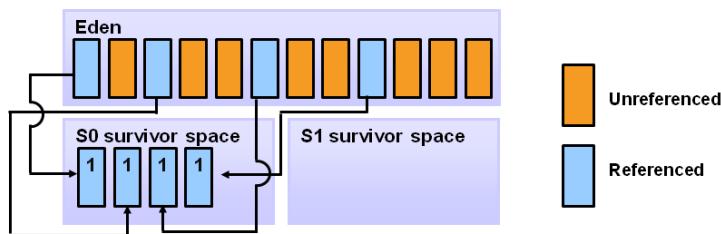
2. Когда Eden полностью заполняется, начинает работу minor gc.

### Filling the Eden Space



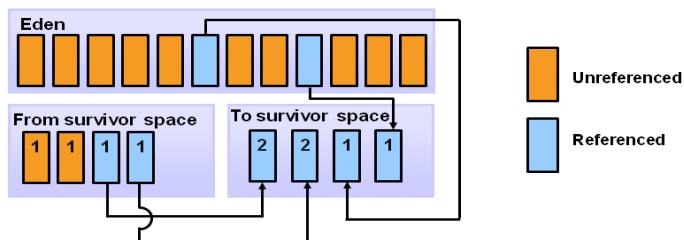
3. Живые объекты перемещаются в первую Survivor 0 область. Мертвые (без ссылок) – удаляются во время очистки Eden.

### Copying Referenced Objects



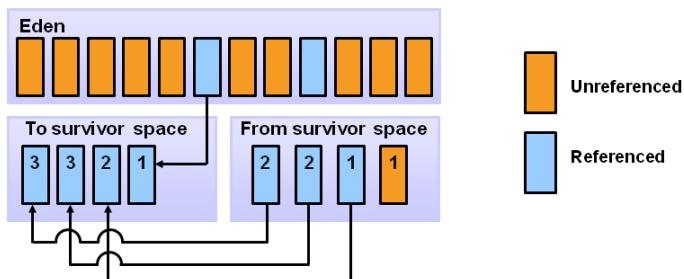
4. Во время следующей minor gc в Eden происходит то же самое. Мертвые объекты удаляются, а живые перемещаются в Survivor 1 (вторую область). В добавок, объекты, размещенные в S0 во время предыдущей minor gc инкрементируют свой возраст и переносятся в S1. После этого Eden и S0 пусты и еще теперь существуют объекты разного возраста.

### Object Aging



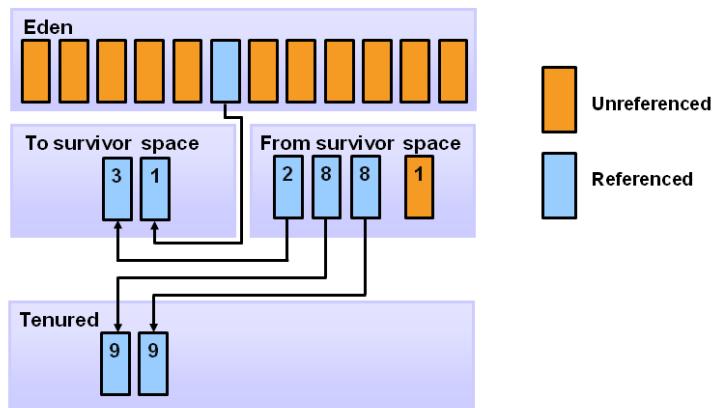
5. Во время следующей minor gc все повторяется (только поменяны местами S0 и S1).

### Additional Aging



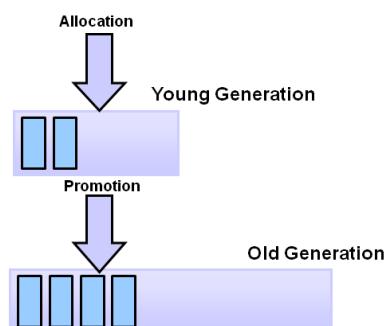
6. «Повышение» - после того, как возраст объектов достиг определенного порога, они перемещаются из New Generation в Old Generation.

### Promotion



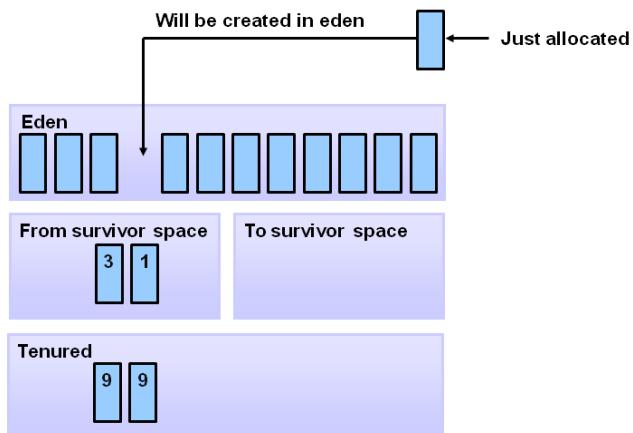
7. Так как minor gc продолжает размещать объекты, объекты продолжают проходить на «повышение».

### Promotion



8. В конечном счете запустится major gc для Old Generation, которая очистит и скомпактует место.

## GC Process Summary



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.

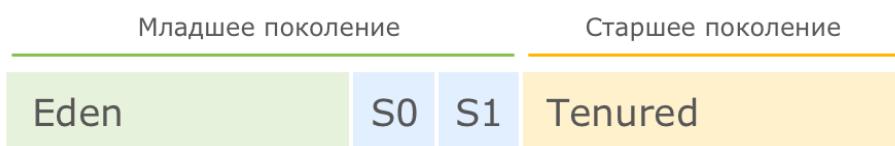
**Serial GC** (он же последовательный сборщик) — младший с точки зрения заложенной в него функциональности, но старший с точки зрения продолжительности присутствия в JVM сборщик мусора. Он медленно, но верно собирал мусор еще тогда, когда многие из нас даже не подозревали о существовании языка Java. 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, что бы "minor GC" проходил быстро, нужно что бы при нем не приходилось сканировать "old generation". Возникает вопрос: "Как выявить ссылки на объекты с "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 алоцируются. 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".

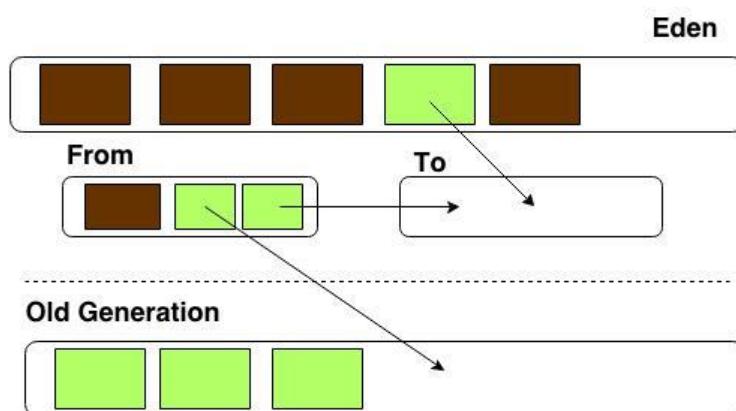
Алгоритм работы очень похож на "Copying collectors", отличие в том, что появился "Eden":

- 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.

Коричневый - мусор.

Зеленый - живой объект.

Сборка мусора:



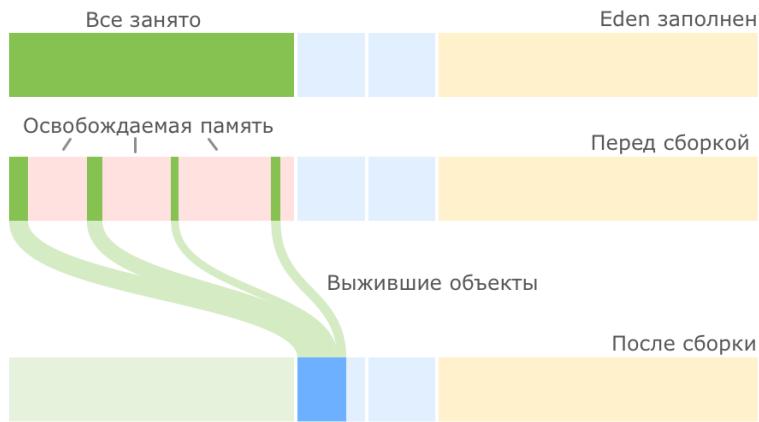
После сборки мусора:



После "minor gc" "Eden" и "to space" пустые, в "from space" лежат объекты пережившие сборку, немного долгоживущих объектов перекочевало в "old generation".

Итого:

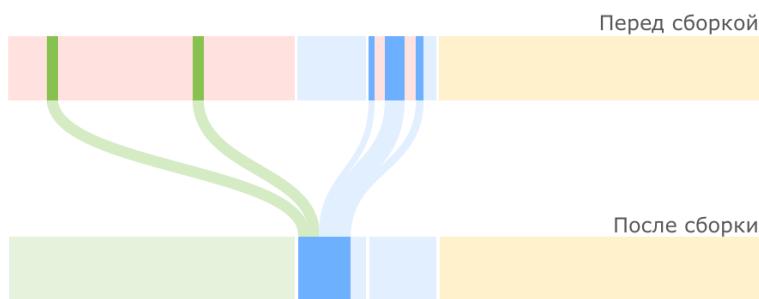
Первым делом такая сборка находит и удаляет мертвые объекты из Eden. Оставшиеся живые объекты переносятся в пустой регион Survivor. **Один из двух регионов Survivor всегда пустой**, именно он выбирается для переноса объектов из Eden:



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:



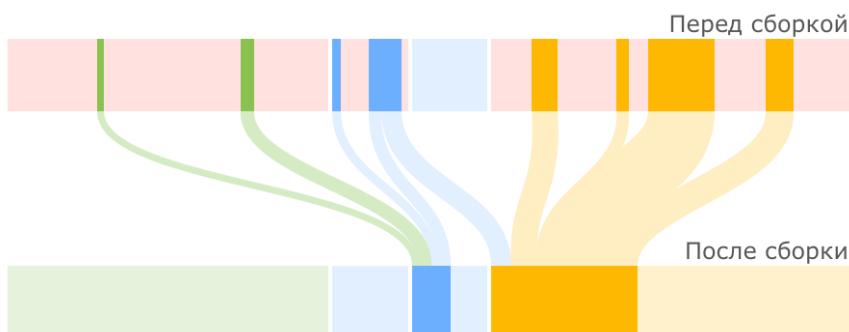
JVM постоянно следит за тем, как долго объекты перемещаются между Survivor 0 и Survivor 1, и выбирает подходящий порог для количества таких перемещений, после которого объекты перемещаются в Tenured, то есть переходят в старшее поколение. Если регион Survivor оказывается заполненным, то объекты из него 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 дефрагментированным 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 усложняется 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).



## Акселераторы

В разделе Eden создается **среднестатистический объект**, а не любой. Такая оговорка сделана неспроста. Дело в том, что бывают еще объекты-акселераторы, размер которых настолько велик, что создавать их в Eden, а потом таскать за собой по Survivor'ам слишком накладно. В этом случае они 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:



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 ожидаем 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=?	<p>Задают минимальную и максимальную долю свободного места в каждом поколении, при достижении которой размер поколения будет автоматически увеличен или уменьшен соответственно.</p> <p>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.</p>
-XX:NewRatio=?	<p>Установка отношения размера старшего поколения к суммарному размеру регионов младшего поколения.</p> <p>Например, NewRatio=3 означает, что для младшего поколения (Eden + S0 + S1) будет отведена четверть кучи, а для старшего — три четверти.</p>
-XX:NewSize=? и -XX:MaxNewSize=?	<p>Можно ограничить размер младшего поколения абсолютными величинами снизу и сверху.</p> <p>Если надо установить для NewSize и MaxNewSize одинаковые значения, то можно просто использовать опцию -Xmn. Например, -Xmn256m эквивалентно -XX:NewSize=256m -XX:MaxNewSize=256m.</p>
-XX:SurvivorRatio=?	<p>Можно настроить отношение размера Eden к размерам Survivor.</p> <p>Например, при SurvivorRatio=6 каждый регион Survivor будет занимать одну восьмую размера всего младшего поколения, а Eden — шесть восьмых (помним про правило опций *Ratio).</p>
-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.

## **Достоинства**

1. Непрятательность 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

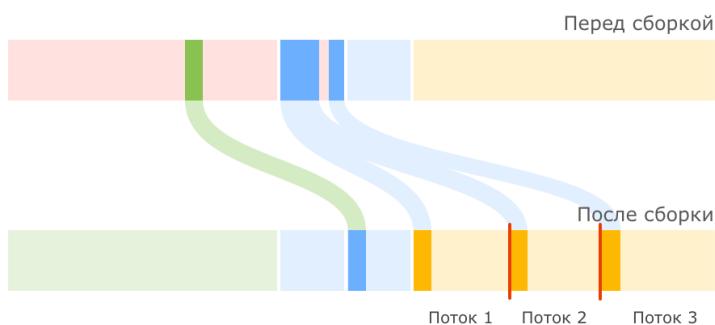
При подключении параллельного сборщика используются те же самые подходы к организации кучи, что и в случае с Serial GC — она делится на такие же регионы Eden, Survivor 0, Survivor 1 и Old Gen (знакомый нам под именем Tenured), функционирующие по тому же принципу.

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  $N \leq 8$ , 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:



**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 распаралеливает 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 <code>-XX:GCTimeRatio=49</code> 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 $1 / (1 + 49)$ ).
<b>-XX:YoungGenerationSizeIncrement=? и -XX:TenuredGenerationSizeIncrement=?</b>	Устанавливают, на сколько процентов следует при необходимости увеличивать младшее и старшее поколение соответственно. По умолчанию оба этих параметра равны 20.
<b>-XX:AdaptiveSizeDecrementScaleFactor=?</b>	<p>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</p> <p>Например, при <code>-XX:AdaptiveSizeDecrementScaleFactor=2</code> каждое уменьшение поколения будет в два раза меньше, чем его увеличение (то есть оба поколения будут уменьшаться на 10% при <code>-XX:GenerationSizeIncrement=20</code> и <code>-XX:TenuredGenerationSizeIncrement=20</code>).</p>

### Достоинства

1. Возможность автоматической подстройки под требуемые параметры производительности.
2. Меньшие паузы на время сборок. 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.

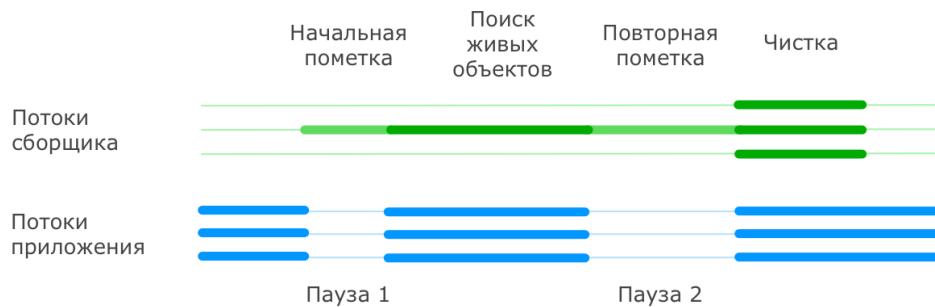
Важным отличием сборщика CMS от рассмотренных ранее является также то, что он не дожидается заполнения Tenured для того, чтобы начать старшую сборку. Вместо этого он трудится в фоновом режиме постоянно, пытаясь поддерживать Tenured в компактном состоянии.

### 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.



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.

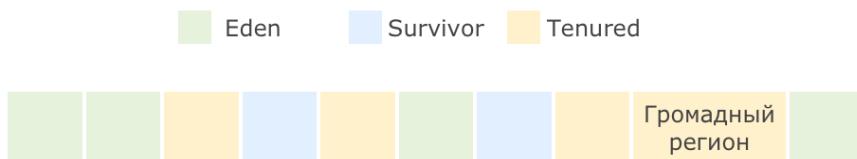
**G1** — самый молодой в составе сборщиков мусора виртуальной машины HotSpot. Он изначально позиционировался как сборщик **для приложений с большими кучами** (от 4 ГБ и выше), для которых важно **сохранять время отклика небольшим и предсказуемым**, пусть даже за счет уменьшения пропускной способности. 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):



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.

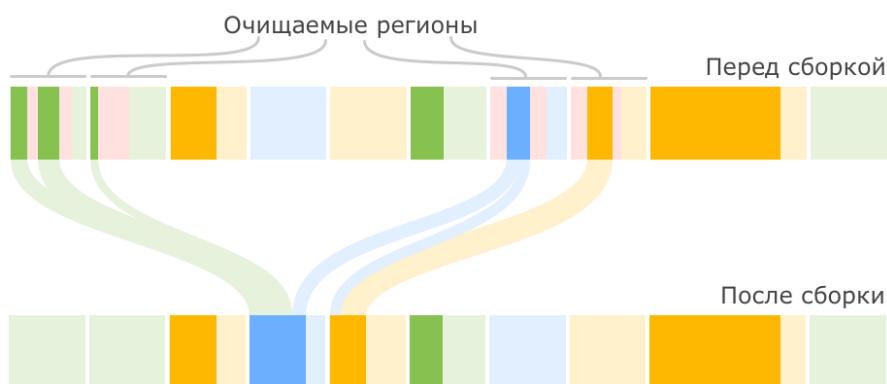
3. **Remark.** Дополнительный поиск не учтенных ранее живых объектов (с остановкой основного приложения).
4. **Cleanup.** Очистка вспомогательных структур учета ссылок на объекты и поиск пустых регионов, которые уже можно использовать для размещения новых объектов. Первая часть этого шага выполняется 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:



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-

акселераторы, 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

<b>-XX:MaxGCPauseMillis=?</b>	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.
<b>-XX:ParallelGCThreads=? и -XX:ConcGCThreads=?</b>	Задают количество потоков, которые будут использоваться для сборки мусора и для выполнения цикла пометок соответственно.
<b>-XX:G1HeapRegionSize=?</b>	Задает размер региона (по умолчанию установлен автоматический выбор размера региона). Значение должно быть степенью двойки, если мерить в мегабайтах. Например, -XX:G1HeapRegionSize=16m.
<b>-XX:InitiatingHeapOccupancyPercent=?</b>	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%.
<b>-XX:+UnlockExperimentalVMOptions</b>	Включение дополнительных функций.
<b>-XX:+AggressiveOpts</b>	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](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.
- Для программы, которые запускаются в фоновом режиме на серверах (программы-демоны) работа автоматического сборщика мусора не всегда подходит. Это потому, что серверы запущены 24x7 и, следовательно, демон будет тоже все время запущен.

## TYPES OF REFERENCES (STRONGREFERENCE, SOFTREFERENCE, WEAKREFERENCE, PHANTOMREFERENCE)

<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 удалит его 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** и **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  $I \leq F * \text{MSPerMB}$ , то не удаляем объект. Если больше то удаляем.**

Для изменения MSPerMB используем ключ **-XX:SoftRefLRUPolicyMSPerMB**. Дефолтовое 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  $F * \text{MSPerMB}$ . 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 особенность 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 находится 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 concatenateImageWithDefaultVersion(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.

Решили кешировать изображение:

```
public class CachedImageProcessor {  
    private static final String IMAGE_NAME = "bigImage.jpg";  
    private InputStream defaultImage;  
  
    public InputStream concatenateImageWithDefaultVersion(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. Приложение работает со многими изображениями и при очередной попытке пользователя обработать изображение, легко может свалиться OutOfMemoryError. И что с этим можно сделать? Использование SoftReference поможет 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<InputStream>(loadImage());  
  
    public InputStream concatenateImageWithDefaultVersion(InputStream userImageAsStream) {  
        if (defaultImageRef.get() == null) { // 1  
            defaultImage = this.getClass().getResourceAsStream(IMAGE_NAME);  
            defaultImageRef = new SoftReference<InputStream>(defaultImage);  
        }  
  
        defaultImage = defaultImageRef.get(); // 2  
        // calculate and return concatenated image  
    }  
}
```

Опасность данной реализации заключается в следующем:

В строке №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 concatenateImageWithDefaultVersion(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.

В реализации `Class` — разработчики создали soft-ссылку на массив конструкторов, полей и методов.

Можно 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.

Если говорить про производительность, то стоит 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 Мар 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

Когда GC определяет, что объект доступен только через weak-ссылки, то этот объект «сразу» удаляется с памяти. Тут стоит вспомнить про `ReferenceQueue` и проследить за порядком удаления объекта с памяти. 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.

## Применение

### 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. Т.е. 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.

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 с 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. У этого решения есть некоторые преимущества по сравнению с intern(). Итак, у нас есть очень 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();  
}
```

Итого: WeakReference используется во многих классах – Thread, ThreadLocal, ObjectOutputStream, Proxy, LogManager.

## PHANTOMREFERENCE

### Особенности GC

Особенностей у этого типа ссылок две:

- метод get() всегда возвращает null. Именно из-за этого PhantomReference имеет смысл использовать только вместе с ReferenceQueue.
- 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

Проблемы возникающие при использовании метода finalize() (переопределение этого метода позволяет нам очистить ресурсы связанные с объектом. Когда GC определяет что объект более недоступный, то перед тем как удалит его из памяти, он выполняет этот метод):

- 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 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.

При использовании 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. Поэтому, если у объекта переопределён метод finalize(), то для его удаления необходимы 3 запуска GC, а если метод не переопределён, то нужно, минимум, 2 запуска GC.**

## major GC

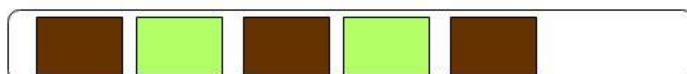
"major GC" работает по принципу "sliding compacting mark-sweep". Принцип работы похож на "Mark-and-sweep", но добавляется процедура "compacting", которая позволяет более эффективно использовать память.

Процедура заключается в перемещении живых объектов к началу "old generation space", таким образом мусор остается в конце. Для аллокации нужно иметь указатель на последний живой объекты и дальше просто алоцировать и сдвигать указатель к концу "old generation".

- Запускается GC
- Приложение приостанавливается
- Сборщик проходится по дереву объектов в "old generation", помечая живые объекты
- Сборщик проходится по всей памяти, находя все не отмеченные куски памяти, они помечаются как мусор
- Все живые объекты сдвигаются к началу "old generation", мусор становится одним куском памяти, который находится сразу за последним живым объектом
- Приложение возобновляет свою работу.

Находим мусор:

### Old Generation



После "compacting":

### Old Generation



# LESSON 5: MULTITHREADING

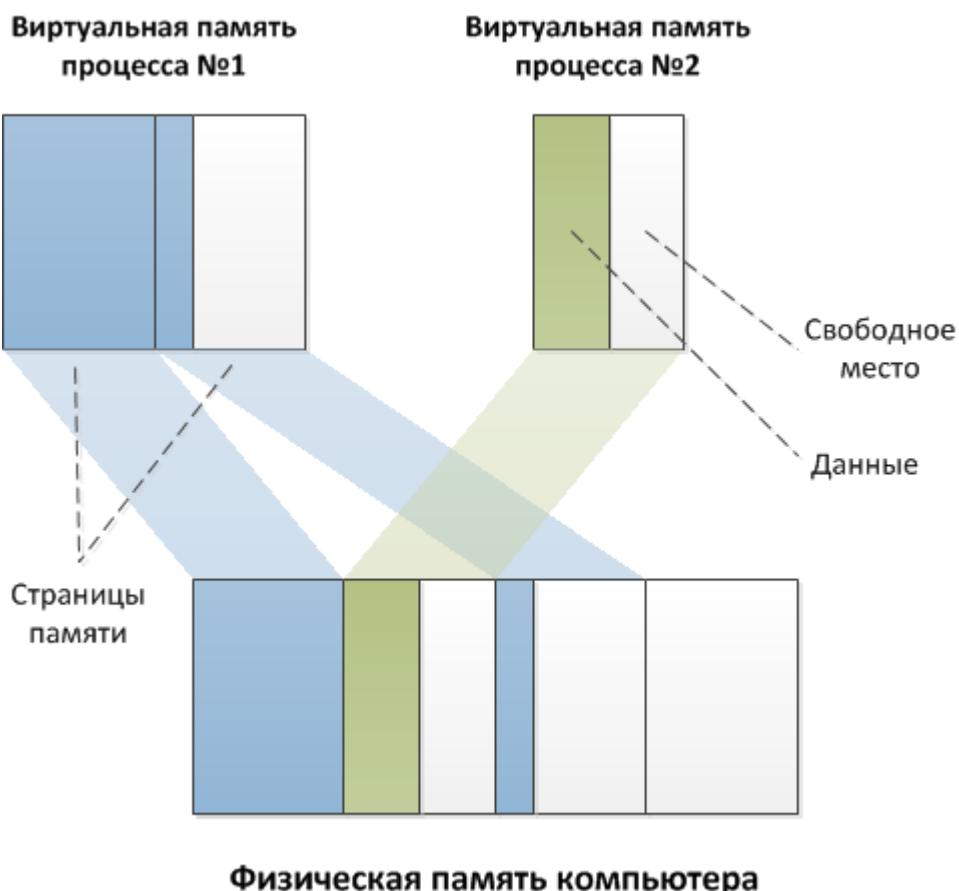
## PROCESS? THREAD? THREAD MANAGER?

### ПРОЦЕССЫ

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают исключения (например, браузер Chrome создает отдельный процесс для каждой вкладки, что дает ему некоторые преимущества, вроде независимости вкладок друг от друга). Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Операционная система оперирует так называемыми страницами памяти, которые представляют собой просто область определенного фиксированного размера. Если процессу становится недостаточно памяти, система выделяет ему дополнительные страницы из физической памяти. Страницы виртуальной памяти могут проецироваться на физическую память в произвольном порядке.



При запуске программы операционная система создает процесс, загружая в его адресное пространство код и данные программы, а затем запускает главный поток созданного процесса.

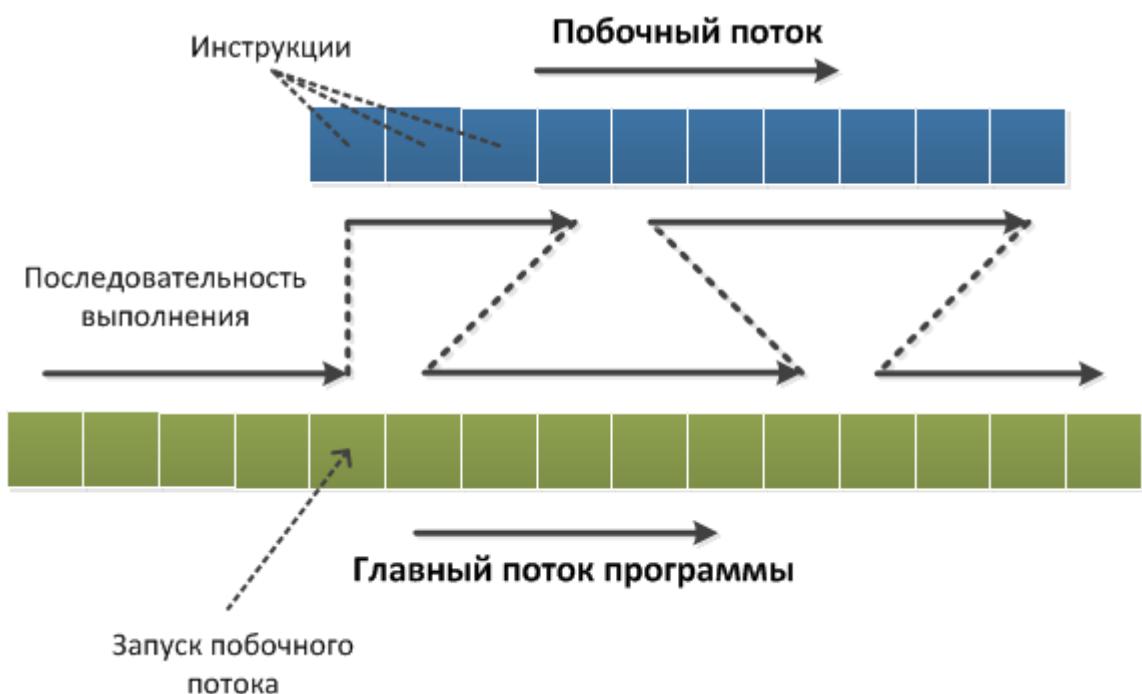
### ПОТОКИ

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

Следует отдельно обговорить фразу «параллельно с другими потоками». Известно, что на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. То есть одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае). Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. Такая схема называется псевдо-параллелизмом. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой поток, и восстанавливает его по возвращению к выполнению потока. В контекст потока входят такие параметры, как стек, набор значений регистров процессора, адрес исполняемой команды и прочее...

Проще говоря, при псевдопараллельном выполнении потоков процессор мечется между выполнением нескольких потоков, выполняя по очереди часть каждого из них.

Вот как это выглядит:



В Java есть Планировщик Потоков(Thread Scheduler), который контролирует все запущенные потоки во всех программах и решает, какие потоки должны быть запущены, и какая строка кода выполняться. Существует две характеристики потока, по которым планировщик идентифицирует процесс:

- более важная, это приоритет потока,
- является ли поток демоном(daemon flag).

Простейшее правило планировщика, это если запущены только daemon потоки, то Java Virtual Machine (JVM) выгрузится. Новые потоки наследуют приоритет и daemon flag от потока, который его создал.

Планировщик определяет какой поток должен быть запущен, анализируя приоритет всех потоков. Потоку с наивысшим приоритетом позволяет выполниться раньше, нежели потокам с более низкими приоритетами.

Планировщик может быть двух видов:

- с преимуществом
- без преимуществом.

Планировщик с преимуществом предоставляет определённый отрезок времени для всех потоков, которые запущены в системе. Планировщик решает, какой поток следующий запуститься или возобновить работу через некоторый постоянный период времени. Когда поток запуститься, через этот определённый промежуток времени, то выполняющийся поток будет приостановлен и следующий поток возобновит свою работу. Планировщик без приоритета решает, какой поток должен запуститься и выполнятся до того, пока не закончит свою работу. Поток имеет полный контроль над системой настолько долго, сколько ему захочется. Метод `yield()` можно использовать для того чтобы принудить планировщик выполнить другой поток, который ожидает своей очереди. В зависимости от системы, на которой запущена Java, планировщик может быть либо с преимуществом, либо без него.

## PROCESS OF THREAD CREATION

### КЛАССИЧЕСКИЙ ПОДХОД К ЗАПУСКУ ЗАДАЧ В МНОГОПОТОЧНОМ РЕЖИМЕ

Классический подход к запуску задач в многопоточном режиме в JSE предполагает использование класса `java.lang.Thread` или интерфейса `java.lang.Runnable`. В первом случае программист создает потомка класса `Thread` и переопределяет в нем метод `run`, куда помещается функциональность, которую необходимо выполнить в многопоточном режиме, как показано ниже:

```
public class ThreadSample extends Thread {  
    @Override  
    public void run() {  
        System.out.println("do some multithreaded task");  
    }  
    public static void main(String[] args) {  
        ThreadSample ts1 = new ThreadSample();  
        ts1.start();  
    }  
}
```

Использование интерфейса `Runnable` основывается на другой парадигме. Сначала программист должен реализовать интерфейс `Runnable` в собственном классе, а затем поместить объект этого класса в объект типа `Thread`. Интересующая функциональность также помещается в метод `run` класса, реализующего интерфейс `Runnable`, и впоследствии вызывается объектом-контейнером, как показано ниже:

```
public class RunnableSample implements Runnable {  
    public void run() {  
        System.out.println("do some multithreaded task");  
    }  
    public static void main(String[] args) {  
        RunnableSample rs1 = new RunnableSample();  
        Thread t1 = new Thread(rs1);  
        t1.start();  
    }  
}
```

Как видно в обоих примерах запуск задачи в многопоточном режиме выполняется через вызов метода `start` объекта типа `Thread`. Только в первом случае после вызова метода `start` класса `Thread` происходит вызов метода `run`, наследника этого класса (класса `ThreadSample`), в котором и находится код, относящейся к задаче. При выборе реализации на основе интерфейса `Runnable` сначала происходит вызов метода `start` класса `Thread`, затем обращение к методу `run` этого же класса, и уже из этого метода вызывается метод `run` реализации интерфейса `Runnable` (класса `RunnableSample`).

Точно предсказать, какой поток закончит высказываться последним, невозможно. Можно попытаться, и можно даже угадать, но есть большая вероятность того, что та же программа при следующем запуске будет иметь другого «победителя». Это происходит из-за так называемого «асинхронного выполнения кода». Асинхронность означает то, что нельзя утверждать, что какая-либо инструкция одного потока, выполнится раньше или позже инструкции другого. Или, другими словами, параллельные потоки независимы друг от друга, за исключением тех случаев, когда программист сам описывает зависимости между потоками с помощью предусмотренных для этого средств языка.

## СОЗДАНИЕ ЗАДАЧИ С ПОМОЩЬЮ ИНТЕРФЕЙСА JAVA.UTIL.CONCURRENT.CALLABLE

Интерфейс **Callable** гораздо больше подходит для создания задач, предназначенных для параллельного выполнения, нежели интерфейс **Runnable** или тем более класс **Thread**. При этом стоит отметить, что возможность добавить подобный интерфейс появилась только начиная с версии Java 5, так как ключевая особенность интерфейса **Callable** – это использование параметризованных типов (generics), как показано ниже:

```
import java.util.concurrent.Callable;
public class CallableSample implements Callable<String>{
    public String call() throws Exception {
        if(какое-то условие) {
            throw new IOException("error during task processing");
        }
        System.out.println("task is processing");
        return "result ";
    }
}
```

Сразу необходимо обратить внимание на строку 2, где указано, что интерфейс **Callable** является параметризованным, и его конкретная реализация – класс **CallableSample**, зависит от типа **String**. На строке 3 приведена сигнатура основного метода **call** в уже параметризованном варианте, так как в качестве типа возвращаемого значения также указан тип **String**. Фактически это означает, что была создана задача, результатом выполнения которой будет объект типа **String** (см. строку 8). Точно также можно создать задачу, в результате работы которой в методе **call** будет создаваться и возвращаться объект любого требуемого типа. Такое решение значительно удобнее по сравнению с методом **run** в интерфейсе **Runnable**, который не возвращает ничего (его возвращаемый тип – **void**) и поэтому приходится изобретать обходные пути, чтобы извлечь результат работы задачи.

Еще одно преимущество интерфейса **Callable** – это возможность «выбрасывать» исключительные ситуации, не оказывая влияния на другие выполняющиеся задачи. На строке 3 указано, что из метода может быть «выброшена» исключительная ситуация типа **Exception**, что фактически означает любую исключительную ситуацию, так как все исключения являются потомками **java.lang.Exception**. На строке 5 эта возможность используется для создания контролируемой (**checked**) исключительной ситуации типа **IOException**. Метод **run** интерфейса **Runnable** вообще не допускал выбрасывания контролируемых исключительных ситуаций, а выброс неконтролируемой (**runtime**) исключительной ситуации приводил к остановке потока и всего приложения.

## ЗАВЕРШЕНИЕ ПРОЦЕССА И ДЕМОНЫ

В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод **main()** уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

Однако это правило не относится к особому виду потоков – демонам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и

выполнение процесса закончится. Чаще всего потоки-демоны используются для выполнения фоновых задач, обслуживающих процесс в течение его жизни.

Объявить поток демоном достаточно просто — нужно перед запуском потока вызвать его метод **setDaemon(true);**

Проверить, является ли поток демоном, можно вызвав его метод **boolean isDaemon();**

## ЗАВЕРШЕНИЕ ПОТОКОВ

В Java существуют средства для принудительного завершения потока. В частности метод Thread.stop() завершает поток незамедлительно после своего выполнения. Однако этот метод, а также Thread.suspend(), приостанавливающий поток, и Thread.resume(), продолжающий выполнение потока, были объявлены устаревшими и их использование отныне крайне нежелательно. Дело в том что поток может быть «убит» во время выполнения операции, обрыв которой на полуслове оставит некоторый объект в неправильном состоянии, что приведет к появлению трудноотлавливаемой и случайным образом возникающей ошибке.

Вместо принудительного завершения потока применяется схема, в которой каждый поток сам ответственен за своё завершение. Поток может остановиться либо тогда, когда он закончит выполнение метода run(), (main() — для главного потока) либо по сигналу из другого потока. Причем как реагировать на такой сигнал — дело, опять же, самого потока. Получив его, поток может выполнить некоторые операции и завершить выполнение, а может и вовсе его проигнорировать и продолжить выполняться. Описание реакции на сигнал завершения потока лежит на плечах программиста.

Взаимодействовать с потоком можно с помощью метода changeAction() (для смены вычитания на сложение и наоборот) и метода finish() (для завершения потока).

В объявлении переменных mIsIncrement и mFinish было использовано ключевое слово volatile (изменчивый, не постоянный). Его необходимо использовать для переменных, которые используются разными потоками. Это связано с тем, что значение переменной, объявленной без volatile, может кэшироваться отдельно для каждого потока, и значение из этого кэша может различаться для каждого из них. Объявление переменной с ключевым словом volatile отключает для неё такое кэширование и все запросы к переменной будут направляться непосредственно в память.

## STATIC METHODS OF THREAD VS NOTIFY(), NOTIFYALL(), WAIT();

Методы класса Thread:

- **destroy()** - Принудительное завершение работы потока(Устаревшее)
- **getName()** - Получает имя потока. Имя потока – ассоциированная с ним строка, которая в некоторых случаях помогает понять, какой поток выполняет некоторое действие. Иногда это бывает полезным.
- **getPriority** - получает приоритет потока.
- **isAlive** - возвращает true если myThread() выполняется и false если поток еще не был запущен или был завершен.
- **join()** - В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод join(). Например, чтобы главный поток подождал завершения побочного потока myThready, необходимо выполнить инструкцию myThready.join() в главном потоке. Как только поток myThready завершится, метод join() вернет управление, и главный поток сможет продолжить выполнение.

Метод join() имеет перегруженную версию, которая получает в качестве параметра время ожидания. В этом случае join() возвращает управление либо когда завершится ожидаемый поток, либо когда закончится время ожидания. Подобно методу Thread.sleep() метод join может ждать в течение миллисекунд и наносекунд – аргументы те же.

- **run()**
- **getId()** -возвращает идентификатор потока. Идентификатор – уникальное число, присвоенное потоку.
- **start()** - Запуск потока на выполнение

- **getState()**
- **getThreadGroup()** - Определение группы, к которой принадлежит поток
- **interrupt()** - Прерывание потока.
- **activeCount()** - Текущее количество активных потоков в группе, к которой принадлежит поток
- **isDaemon()** - Определение, является ли поток демоном
- **isInterrupted()** - Определение, является ли поток прерванным
- **setDaemon()** - Метод вызывается в том случае, если поток был создан как объект с интерфейсом

Runnable run(). Установка для потока режима демона

- **setName()** - Задает имя потока.
- **setPriority()** - устанавливает приоритет потока.

Возможные значения priority — MIN\_PRIORITY, NORM\_PRIORITY и MAX\_PRIORITY.

Note: при создание потока приоритет выставляется по середине, если 1-ый поток создан с приоритетом выше, то и последующий поток будет с таким же приоритетом

- **currentThread()** - Определение текущего работающего потока
- **interrupted()** - Определение, является ли поток прерванным
- **yield()** - заставляет процессор переключиться на обработку других потоков системы. Метод может быть полезным, например, когда поток ожидает наступления какого-либо события и необходимо чтобы проверка его наступления происходила как можно чаще. В этом случае можно поместить проверку события и метод Thread.yield() в цикл:

```
//Ожидание поступления сообщения
while(!msgQueue.hasMessages())           //Пока в очереди нет сообщений
{
    Thread.yield();                     //Передать управление другим потокам
}
```

- **sleep()** - статический метод класса Thread, который приостанавливает выполнение потока, в котором он был вызван. Во время выполнения метода sleep() система перестает выделять потоку процессорное время, распределяя его между другими потоками. Метод sleep() может выполняться либо заданное кол-во времени (миллисекунды или наносекунды) либо до тех пор пока он не будет остановлен прерыванием (в этом случае он генерирует исключение InterruptedException).

```
Thread.sleep(1500); //Ждет полторы секунды
Thread.sleep(2000, 100); //Ждет 2 секунды и 100 наносекунд
```

Несмотря на то, что метод sleep() может принимать в качестве времени ожидания наносекунды, не стоит принимать это всерьез. Во многих системах время ожидания все равно округляется до миллисекунд а то и до их десятков.

- **static Thread.currentThread()** — статический метод, возвращающий объект потока, в котором он был вызваны

**Методы wait(), notify() и notifyAll()** Эти методы никогда не переопределяются и используются только в исходном виде. Вызываются только внутри синхронизированного блока или метода на объекте, монитор которого захвачен текущим потоком. Попытка обращения к данным методам вне синхронизации или на несинхронизированном объекте (со свободным монитором) приводит к генерации исключительной ситуации IllegalMonitorStateException. Эти методы используются для управления потоками в ситуации, когда необходимо задать определенную последовательность действий без повторного запуска потоков. Метод wait(), вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченный объект. Возвратить блокировку объекта потоку можно вызовом метода notify() для одного потока или notifyAll() для всех потоков. Если ожидающих потоков несколько, то после вызова метода notify() невозможно определить, какой поток из ожидающих потоков заблокирует объект. Вызов может быть осуществлен только из другого потока, заблокировавшего в свою очередь тот же самый объект.

*Wait() vs. Sleep()*

**Call on:**

- `wait()`: Call on an object; current thread must synchronize on the lock object.
- `sleep()`: Call on a Thread; always currently executing thread.

**Synchronized:**

- `wait()`: when synchronized multiple threads access same Object one by one.
- `sleep()`: when synchronized multiple threads wait for sleep over of sleeping thread.

**Hold lock:**

- `wait()`: release the lock for other objects to have chance to execute.
- `sleep()`: keep lock for at least t times if timeout specified or somebody interrupt.

**Wake-up condition:**

- `wait()`: until call `notify()`, `notifyAll()` from object
- `sleep()`: until at least time expire or call `interrupt()`.

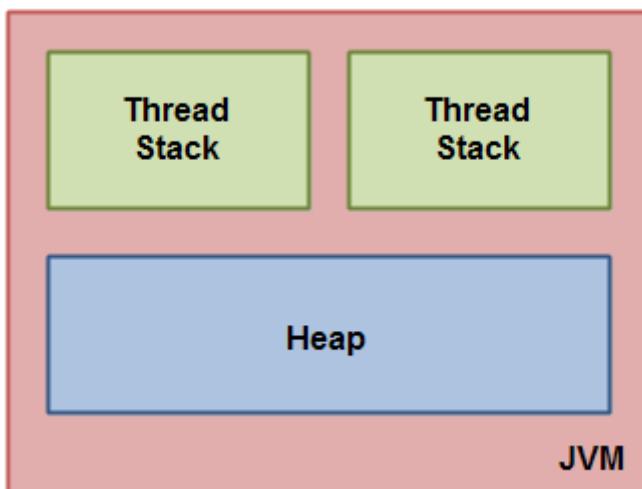
**Usage:**

- `sleep()`: for time-synchronization and;
- `wait()`: for multi-thread-synchronization.

## JAVA MEMORY MODEL FROM MULTITHREADING POINT OF VIEW

### THE INTERNAL JAVA MEMORY MODEL

The Java memory model used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:



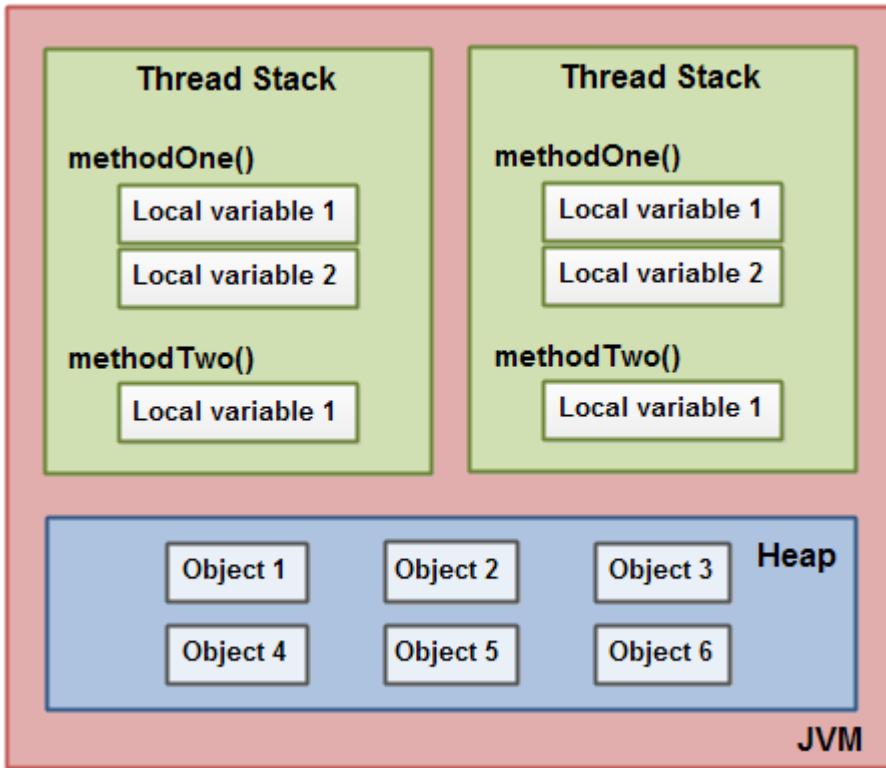
Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution. I will refer to this as the "call stack". As the thread executes its code, the call stack changes.

The thread stack also contains all local variables for each method being executed (all methods on the call stack). A thread can only access its own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it. Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, each thread has its own version of each local variable.

All local variables of primitive types ( boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a primitive variable to another thread, but it cannot share the primitive local variable itself.

The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.

Here is a diagram illustrating the call stack and local variables stored on the thread stacks, and objects stored on the heap:



### III

A local variable may be of a primitive type, in which case it is totally kept on the thread stack.

A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself is stored on the heap.

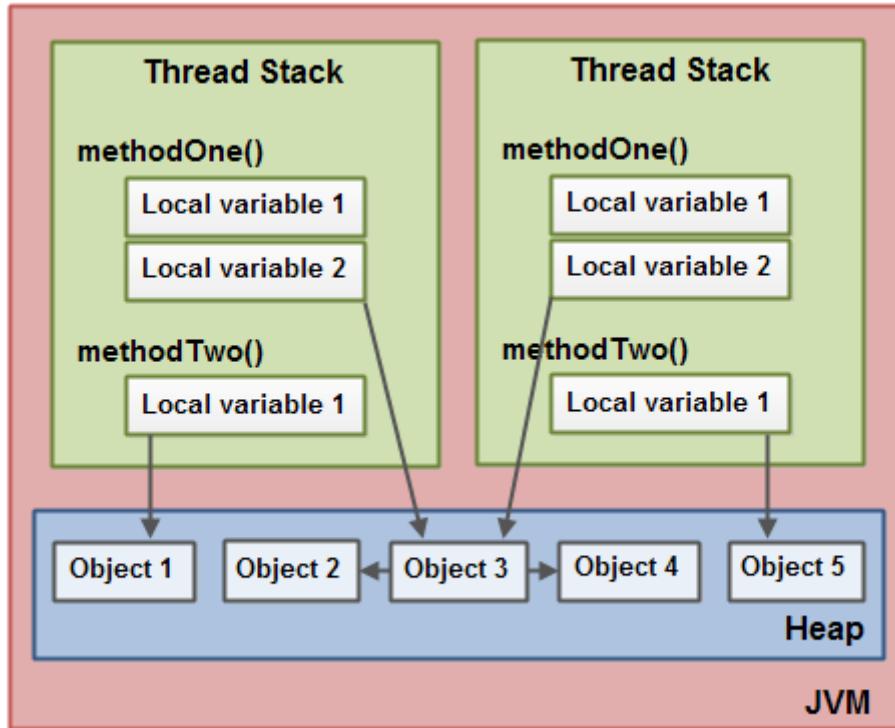
An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.

An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.

Static class variables are also stored on the heap along with the class definition.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.

Here is a diagram illustrating the points above:



Two threads have a set of local variables. One of the local variables (Local Variable 2) point to a shared object on the heap (Object 3). The two threads each have a different reference to the same object. Their references are local variables and are thus stored in each thread's thread stack (on each). The two different references point to the same object on the heap, though.

Notice how the shared object (Object 3) has a reference to Object 2 and Object 4 as member variables (illustrated by the arrows from Object 3 to Object 2 and Object 4). Via these member variable references in Object 3 the two threads can access Object 2 and Object 4.

The diagram also shows a local variable which points to two different objects on the heap. In this case the references point to two different objects (Object 1 and Object 5), not the same object. In theory both threads could access both Object 1 and Object 5, if both threads had references to both objects. But in the diagram above each thread only has a reference to one of the two objects.

So, what kind of Java code could lead to the above memory graph? Well, code as simple as the code below:

```
public class MyRunnable implements Runnable {
    public void run() {
        methodOne();
    }

    public void methodOne() {
        int localVariable1 = 45;

        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;

        //... do more with local variables.

        methodTwo();
    }
}
```

```

public void methodTwo() {
    Integer localVariable1 = new Integer(99);

    //... do more with local variable.
}
}

public class MySharedObject {

//static variable pointing to instance of MySharedObject

public static final MySharedObject sharedInstance =
    new MySharedObject();

//member variables pointing to two objects on the heap

public Integer object2 = new Integer(22);
public Integer object4 = new Integer(44);

public long member1 = 12345;
public long member1 = 67890;
}

```

If two threads were executing the run() method then the diagram shown earlier would be the outcome. The run() method calls methodOne() and methodOne() calls methodTwo().

methodOne() declares a primitive local variable (localVariable1 of type int) and an local variable which is an object reference (localVariable2).

Each thread executing methodOne() will create its own copy of localVariable1 and localVariable2 on their respective thread stacks. The localVariable1 variables will be completely separated from each other, only living on each thread's thread stack. One thread cannot see what changes another thread makes to its copy of localVariable1.

Each thread executing methodOne() will also create their own copy of localVariable2. However, the two different copies of localVariable2 both end up pointing to the same object on the heap. The code sets localVariable2 to point to an object referenced by a static variable. There is only one copy of a static variable and this copy is stored on the heap. Thus, both of the two copies of localVariable2 end up pointing to the same instance of MySharedObject which the static variable points to. The MySharedObject instance is also stored on the heap. It corresponds to Object 3 in the diagram above.

Notice how the MySharedObject class contains two member variables too. The member variables themselves are stored on the heap along with the object. The two member variables point to two other Integer objects. These Integer objects correspond to Object 2 and Object 4 in the diagram above.

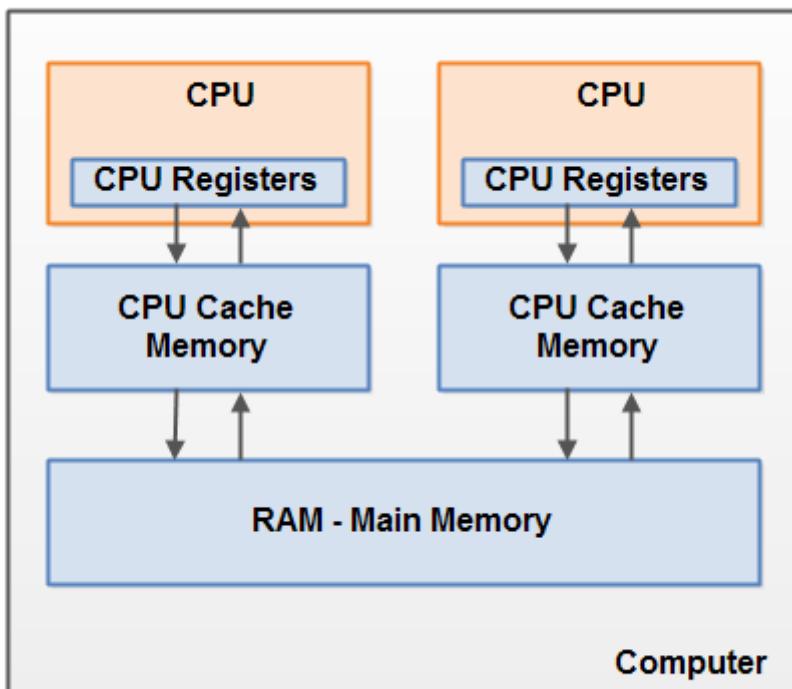
Notice also how methodTwo() creates a local variable named localVariable1. This local variable is an object reference to an Integer object. The method sets the localVariable1 reference to point to a new Integer instance. The localVariable1 reference will be stored in one copy per thread executing methodTwo(). The two Integer objects instantiated will be stored on the heap, but since the method creates a new Integer object every time the method is executed, two threads executing this method will create separate Integer instances. The Integer objects created inside methodTwo() correspond to Object 1 and Object 5 in the diagram above.

Notice also the two member variables in the class MySharedObject of type long which is a primitive type. Since these variables are member variables, they are still stored on the heap along with the object. Only local variables are stored on the thread stack.

## HARDWARE MEMORY ARCHITECTURE

Modern hardware memory architecture is somewhat different from the internal Java memory model. It is important to understand the hardware memory architecture too, to understand how the Java memory model works with it. This section describes the common hardware memory architecture, and a later section will describe how the Java memory model works with it.

Here is a simplified diagram of modern computer hardware architecture:



A modern computer often has 2 or more CPUs in it. Some of these CPUs may have multiple cores too. The point is, that on a modern computer with 2 or more CPUs it is possible to have more than one thread running simultaneously. Each CPU is capable of running one thread at any given time. That means that if your Java application is multithreaded, one thread per CPU may be running simultaneously (concurrently) inside your Java application.

Each CPU contains a set of registers which are essentially in-CPU memory. The CPU can perform operations much faster on these registers than it can perform on variables in main memory. That is because the CPU can access these registers much faster than it can access main memory.

Each CPU may also have a CPU cache memory layer. In fact, most modern CPUs have a cache memory layer of some size. The CPU can access its cache memory much faster than main memory, but typically not as fast as it can access its internal registers. So, the CPU cache memory is somewhere in between the speed of the internal registers and main memory. Some CPUs may have multiple cache layers (Level 1 and Level 2), but this is not so important to know to understand how the Java memory model interacts with memory. What matters is to know that CPUs can have a cache memory layer of some sort.

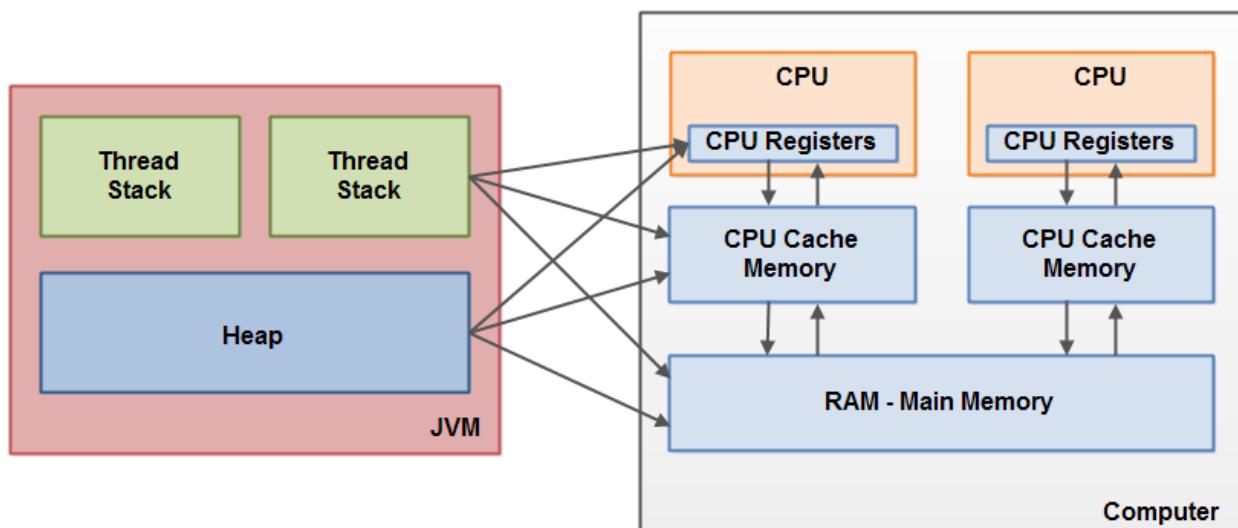
A computer also contains a main memory area (RAM). All CPUs can access the main memory. The main memory area is typically much bigger than the cache memories of the CPUs.

Typically, when a CPU needs to access main memory it will read part of main memory into its CPU cache. It may even read part of the cache into its internal registers and then perform operations on it. When the CPU needs to write the result back to main memory it will flush the value from its internal register to the cache memory, and at some point flush the value back to main memory.

The values stored in the cache memory is typically flushed back to main memory when the CPU needs to store something else in the cache memory. The CPU cache can have data written to part of its memory at a time, and flush part of its memory at a time. It does not have to read / write the full cache each time it is updated. Typically the cache is updated in smaller memory blocks called "cache lines". One or more cache lines may be read into the cache memory, and one or more cache lines may be flushed back to main memory again.

## BRIDGING THE GAP BETWEEN THE JAVA MEMORY MODEL AND THE HARDWARE MEMORY ARCHITECTURE

As already mentioned, the Java memory model and the hardware memory architecture are different. The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers. This is illustrated in this diagram:



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

- Visibility of thread updates (writes) to shared variables.
- Race conditions when reading, checking and writing shared variables.

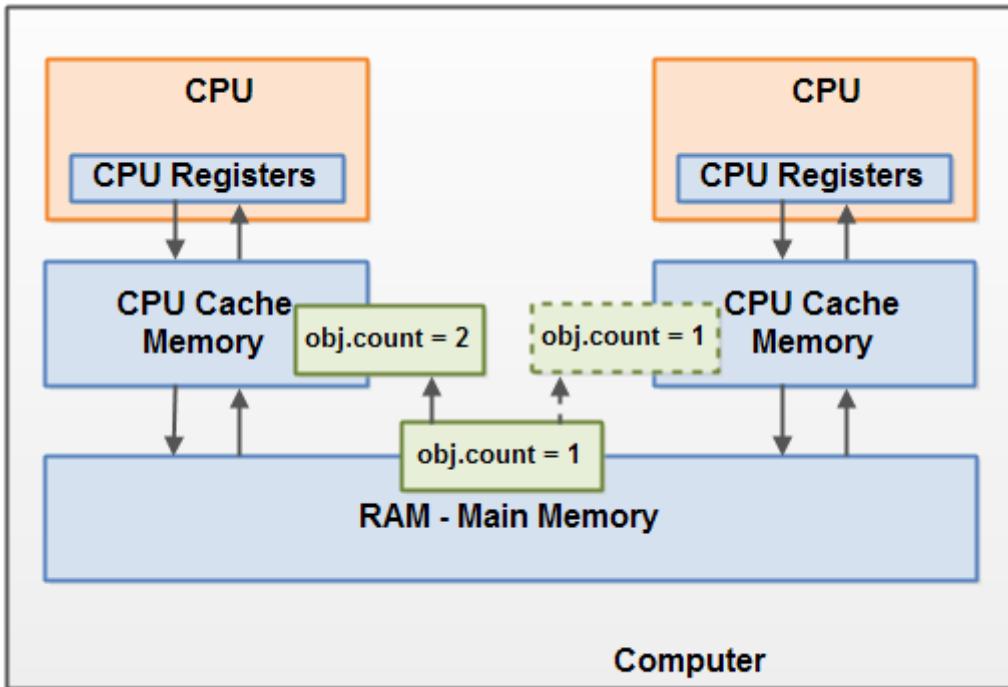
Both of these problems will be explained in the following sections.

## VISIBILITY OF SHARED OBJECTS

If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to the shared object made by one thread may not be visible to other threads.

Imagine that the shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. There it makes a change to the shared object. As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.

The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2. This change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.



To solve this problem you can use Java's `volatile` keyword. The `volatile` keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.

## RACE CONDITIONS

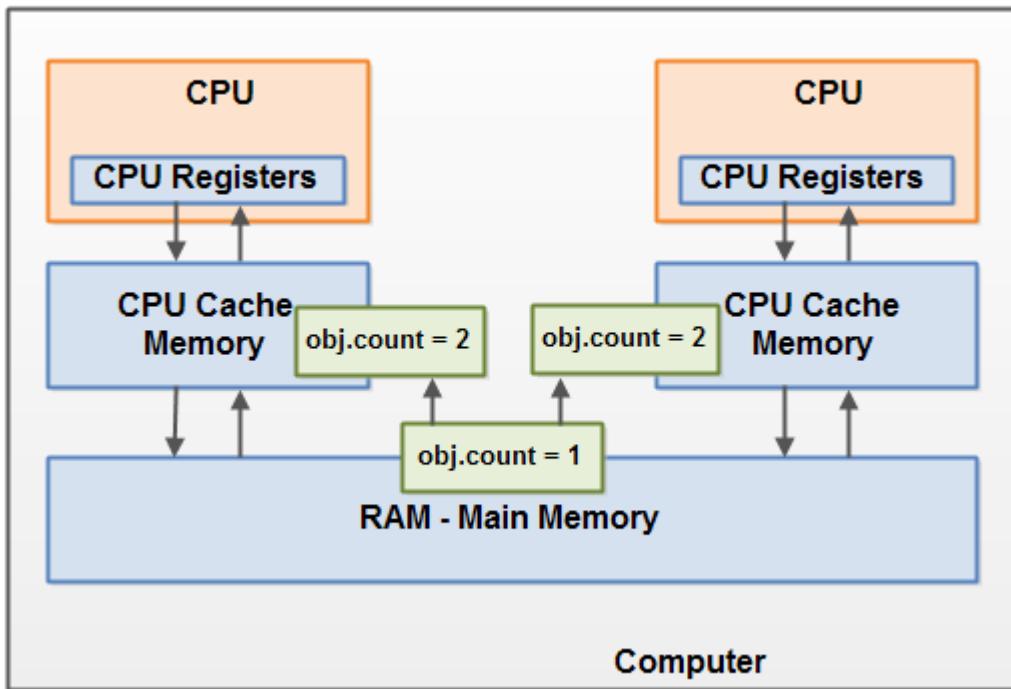
If two or more threads share an object, and more than one thread updates variables in that shared object, race conditions may occur.

Imagine if thread A reads the variable count of a shared object into its CPU cache. Imagine too, that thread B does the same, but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.

If these increments had been carried out sequentially, the variable count would be incremented twice and had the original value + 2 written back to main memory.

However, the two increments have been carried out concurrently without proper synchronization. Regardless of which of thread A and B that writes its updated version of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments.

This diagram illustrates an occurrence of the problem with race conditions as described above:



To solve this problem you can use a Java synchronized block. A synchronized block guarantees that only one thread can enter a given critical section of the code at any given time. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

## THREAD STATES AND LIFE CYCLE

*Note: Мы можем создать поток в Java и запустить его, но как состояние потока меняется от Runnable в Running и в Blocked зависит от реализации системного планировщика потоков (Thread scheduler) и в Java нет полного контроля над этим процессом. На рисунке 1 показан этот процесс:*

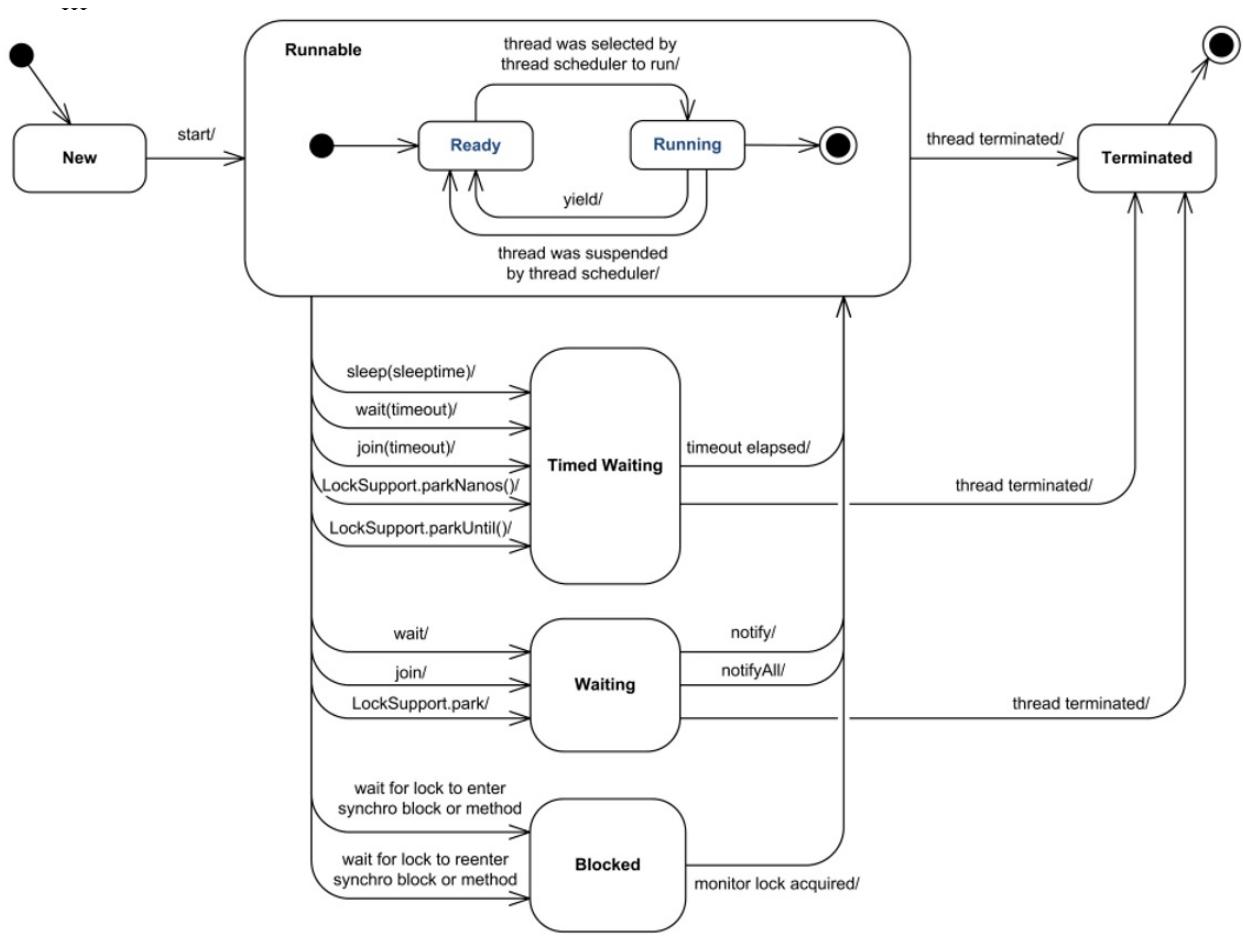


Рисунок 1 — Состояния потока в Java

## 1. СОСТОЯНИЕ ПОТОКА: NEW

Когда мы создаем новый объект класса Thread, используя оператор new, то поток находится в состоянии New. В этом состоянии поток еще не работает.

## 2. СОСТОЯНИЕ ПОТОКА: RUNNABLE

Когда мы вызываем метод start() созданного объекта Thread, его состояние изменяется на Runnable и управление потоком передается Планировщику потоков (Thread scheduler). Ли запустить эту нить мгновенно или сохранить его в работоспособный пула потоков перед запуском, это зависит от реализации ОС в планировщик потоков.

## 3. СОСТОЯНИЕ ПОТОКА: RUNNING

Когда поток будет запущен, его состояние изменится на Running. Планировщик потоков выбирает один поток из своего общего пула потоков и изменяет его состояние на Running. Сразу после этого процессор начинает выполнение этого потока. Во время выполнения состояние потока также может изменится на Runnable, Dead или Blocked.

## 4. СОСТОЯНИЕ ПОТОКА: BLOCKED ИЛИ WAITING

Поток может ждать другой поток для завершения своей работы, например, ждать освобождения ресурсов или ввода-вывода. В этом случае его состояние изменяется на Waiting. После того, как ожидание потока закончилось, его состояние изменяется на Runnable и он возвращается общий пул потоков.

## 5. СОСТОЯНИЕ ПОТОКА: TERMINATE

После того, как поток завершает выполнение, его состояние изменяется наTerminate, то есть он отработал свое и уже не нужен.

### 3)Monitor

Контроль за доступом к объекту-ресурсу обеспечивает понятие монитора. Монитор экземпляра может иметь только одного владельца. При попытке конкурентного доступа к объекту, чей монитор имеет владельца, желающий заблокировать объект-ресурс поток должен подождать освобождения монитора этого объекта и только после этого завладеть им и начать использование объекта-ресурса. Каждый экземпляр любого класса имеет монитор. Методы wait(), wait(long millis), notify(), notifyAll() корректно срабатывают только на экземплярах, чей монитор уже кем-то захвачен. Статический метод захватывает монитор экземпляра класса Class, того класса, на котором он вызван. Существует в единственном экземпляре. Нестатический метод захватывает монитор экземпляра класса, на котором он вызван.

Монитор – это средство обеспечения контроля за доступом к ресурсу. У монитора может быть максимум один владелец в каждый текущий момент времени. Следовательно, если кто-то использует ресурс и захватил монитор для обеспечения единоличного доступа, то другой, желающий использовать тот же ресурс, должен подождать освобождения монитора, захватить его и только потом начать использовать ресурс.

Удобно представлять монитор как id захватившего его объекта. Если этот id равен 0 – ресурс свободен. Если не 0 – ресурс занят. Можно встать в очередь и ждать его освобождения.

Это была общая теория. Перейдем к Java. Здесь у **каждого экземпляра** объекта есть монитор. Реализован он где-то в недрах native-кода и контролируется виртуальной машиной. Используется он так: любой нестатический synchronized-метод при своем вызове прежде всего пытается захватить монитор того объекта, у которого он вызван (на который он может сослаться как на this). Если это удалось – метод исполняется. Если нет – поток останавливается и ждет, пока монитор будет отпущен.

Как работает synchronized-блок. Пусть у нас есть следующий код:

```
Object sync = new Object();
synchronized(sync){  
}
```

В этом случае (синхронизация блока) захватывается монитор у объекта sync. Таким образом, объявление synchronized-метода ...

```
public synchronized void someMethod(){
    // code
}
```

... полностью эквивалентно следующей конструкции:

```
public void someMethod(){
    synchronized(this){
        // code
    }
}
```

У статического метода нет ссылки this. А synchronized-метод – реальность. Чей же монитор он захватывает? Все просто. Пусть у него нет ссылки this, но есть класс. В смысле, объект класса Class. И есть он в одном экземпляре. Идеальный кандидат на использование его монитора для синхронизации статических методов. Собственно, именно так и делается. Таким образом, следующая конструкция:

```

public class SomeClass{

    public static synchronized void someMethod(){
        //code
    }

}

```

... эквивалентна такой:

```

public class SomeClass{

    public static void someMethod(){
        synchronized(SomeClass.class){
            //code
        }
    }

}

```

Итак, резюме. Если два нестатических метода объявлены как `synchronized`, то в каждый момент времени из разных потоков **на одном объекте** может быть вызван только один из них. Поток, который вызывает метод первым, захватит монитор, и второму потоку придется ждать. Заостряю ваше внимание на трех моментах:

- Это верно только для **разных** потоков. Один и тот же поток может вызвать синхронизированный метод, внутри него – другой синхронизированный метод на том же экземпляре. Поскольку этот поток владеет монитором, проблем второй вызов не создаст.
- Это верно только для вызовов методов **одного** экземпляра. У разных экземпляров разные мониторы, потому одновременный вызов нестатических методов проблем не создаст
- В случае статических методов имеет значение только одно – разные ли потоки, вызывающие синхронизированные методы, или нет. Об экземпляре тут речи не идет, его роль исполняет объект класса.

Еще одно замечание. Объекты класса `Class` существуют в единственном экземпляре только в пределах одного `ClassLoader`-а. Следовательно, если вы установите контекстный загрузчик классов потоку – у разных потоков могут быть разные экземпляры одного и того же класса `Class` и, следовательно, будет возможен одновременный вызов синхронизированных статических методов. Если эти методы используют одни и те же ресурсы – это может вызвать проблемы.

Сразу же возникает вопрос о взаимодействии синхронизированных методов с несинхронизированными. Т.е. – возможен ли одновременный их вызов из двух потоков. Да, возможен. Ибо несинхронизированный метод при вызове не пытается захватить монитор, и, следовательно, ничто ему не может помешать. А потому – надо очень аккуратно подходить к синхронизации методов.

Для чего же все-таки нужно два различных варианта синхронизации – на уровне метода и на уровне блока? Почему нельзя обойтись одним?

Синхронизация на уровне метода имеет очевидный недостаток – метод может быть долгим. Если, скажем, нужно менять данные в некой реализации `Runnable` (например, выставлять флаг окончания работы потока, как я описывал выше) и при этом синхронизировать метод `run` – ничего хорошего не получится. Ибо `run` захватит монитор при старте и любой вызов любого синхронизированного метода из любого другого потока будет блокирован.

Можно было бы, конечно, ограничиться только синхронизацией на уровне блока. Функционально это то же самое. Но менее удобно. У синхронизированного метода ключевое слово `synchronized` фигурирует в сигнатуре, что сразу дает разработчику много информации о поведении данного метода. Если же синхронизация будет внутри, блоком от начала до конца метода – о том, что она есть, нужно будет упоминать в комментариях к методу. Учитывая любовь разработчиков к написанию комментариев, а также к их чтению (а тем паче – к чтению документации!) – лучше включить подобное указание в сигнатуру.

С механизмом синхронизации (захватом монитора) неразрывно связан такой неприятный феномен, как взаимные блокировки, *deadlocks*.