Урок 1

Хранение данных и устройство памяти

Где и как хранятся данные в Java? Ссылочные типы.

Где и как хранятся данные в Java?

Мы уже умеем создавать переменные разных типов – ссылочные и примитивные, сделанные на базе стандартных или написанных нами классов. Теперь разберемся, как это происходит в памяти JVM, чтобы лучше понимать структуры, с которыми работаем, и писать более эффективный код.

При работе с Java интересны две области памяти — куча (**heap**) и стек (**stack**). При старте JVM получает часть памяти от операционной системы, на базе которой она запущена, и использует этот ресурс для выполнения программ.

В Java **heap** используется для выделения памяти под создаваемые в коде объекты и встроенные классы JRE. Когда мы добавляем новый объект, область памяти для него выделяется именно в **heap**. Любой объект, созданный в ней, имеет глобальный доступ, и на него могут ссылаться из любой части приложения. **Heap** — более медленная, но и более структурированная память с большим объемом.

Стек в Java работает по схеме LIFO (последний вошел — первый вышел). Это быстрая память, хранящая значения примитивных переменных, а также ссылки на объекты, размещенные в **heap**. Размер стековой памяти намного меньше объема памяти в **heap**.

При вызове метода в стеке создается новый блок, содержащий примитивы и ссылки на другие объекты в вызванном методе — это стековый фрейм. По окончании работы метода данный блок перестает использоваться и предоставляет следующему методу доступ к стеку. На вершине стека всегда располагается метод, выполняющийся в данный момент, и остается там до завершения.

Предположим, у нас есть три метода:

```
public void doSomething() {
    makeStep();
}

public void makeStep(){
    int x = y + 42;
    finishLogic();
}

public void finishLogic(){
    boolean b = true;
}
```

В стеке это будет выглядеть следующим образом:

1. Пользовательский код вызывает метод **doSomething**, помещая его на вершину стека.
2. Внутри этого метода вызывается **makeStep()**, занимающий верхнее место. В его фрейме создаются переменные **x** и **y**.
3. Затем вызывается метод **finishLogic**, во фрейме которого будет присутствовать переменная **b**.
4. По завершении **finishLogic** его фрейм удаляется из стека, после чего управление возвращается к методу **makeStep** на строку кода после **finigshLogic**. И так далее.

Для переменных ссылочных типов (вне зависимости от места вызова) также будет храниться только ссылка на объект.

В продвинутых курсах Java вы познакомитесь с многопоточностью — возможностью кода создавать в программе независимые нити команд, которые могут работать с общими ресурсами. В данный момент важно запомнить, что **heap** в JVM всегда одна, но стек будет выделен для каждого потока персонально.

Ссылочные типы

При создании нового экземпляра класса Java выделяет для него место в **heap**. Для примитивных типов известен точный размер необходимой памяти, а для объектов выделяется ровно столько места, чтобы разместились все переменные данного экземпляра. Следовательно, переменные экземпляра класса размещаются именно в **heap** вне зависимости от того, ссылочные они или примитивные.

Если одним из свойств объекта **Car** является ссылочная переменная **Engine** (ссылка на объект), Java выделит для свойства **Engine** место только под ссылку. Сам объект-свойство **Engine** будет храниться в отдельной области, созданной при его инициализации.

При переполнении памяти стека или **heap** Java будет оповещать пользователя ошибкой-исключением о том, что память в определенной области закончилась.

Подведем итоги:

- **Heap** (куча) используется всеми частями приложения, а стек — только одним потоком выполнения программы;
- Когда создается объект, он хранится в **heap**, а в памяти стека содержится ссылка на него. Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в **heap**;
- Объекты в **heap** доступны из любой точки программы, а стековая память недоступна для других потоков;
- Управление памятью в стеке осуществляется по схеме LIFO;
- Стековая память существует ограниченное время при работе программы, а память в **heap** «живет» с начала до конца ее выполнения;
- Если память стека полностью занята, Java Runtime бросает **java.lang.StackOverflowError**. Если память **heap** заполнена, бросается исключение **java.lang.OutOfMemoryError: Java Heap Space**;
- Размер памяти стека намного меньше памяти в **heap**. Из-за простоты распределения (LIFO) стековая память работает намного быстрее **heap**.