Theory: Call stack

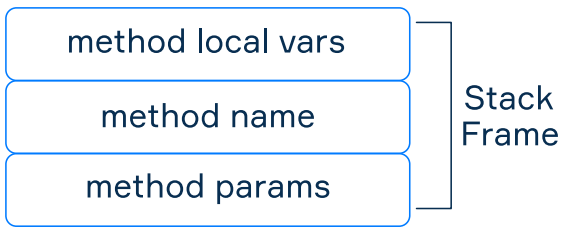
16 minutes 5 / 8 problems solved

Start practicing

When you write a program, it probably contains several methods invoking each other, either programmer-defined or standard ones, and all of them need to be executed. How does the machine understand the order of the execution? How does it switch between different methods? How does it know when the program execution is over? To shed light on these questions we need to learn about a special data structure — a call stack.

§1. Call stack structure

JVM uses a **call stack** (or **execution stack**) to understand which method should be invoked next and to access information regarding the method. The call stack is composed of **stack frames** that store information about methods that have not yet terminated. The information includes the address of a method, parameters, local variables, intermediate computations, and some other data.



As a regular stack, the call stack follows the rule **Last In First Out (LIFO)**. It means stack frames are pushed at the top and move everything down. A new stack frame is added when the execution enters the method. And the stack frame is removed from the call stack if the execution of a method is done.

§2. Stack frame example

Let's consider an example of a **call stack** for a program that prints the next even number of the given one. For simplicity, we will use the number 99 as the input.

> If you have forgotten or did not know, an **even number** is a number that can be divided exactly by 2. Otherwise, a number is called **odd**.

Here is the program:

```
public class Main {
    public static void main(String[] args) {
        int n = 99;
        printNextEvenNumber(n);
    }

    public static void printNextEvenNumber(int n) {
        int next = (n % 2 == 0) ? n + 2 : n + 1;
        System.out.println(next);

    }

}
```

The program declares two methods: `main` and `printNextEvenNumber`.

The first method to be invoked is `main`. Each time a method is invoked, a new stack frame is created. The stack frame for `main` is structured the following way:

1. The method parameters (`args`) are pushed on the stack.
2. The method address (shown in the scheme as the method name — `main`) is added to the stack frame to keep a reference to where to

Current topic:

- ✓ Call stack

Topic depends on:

- ✓ Stack
- ✓ Declaring a method — Stage 3
- ✗ Write, compile, and run — Stage 3

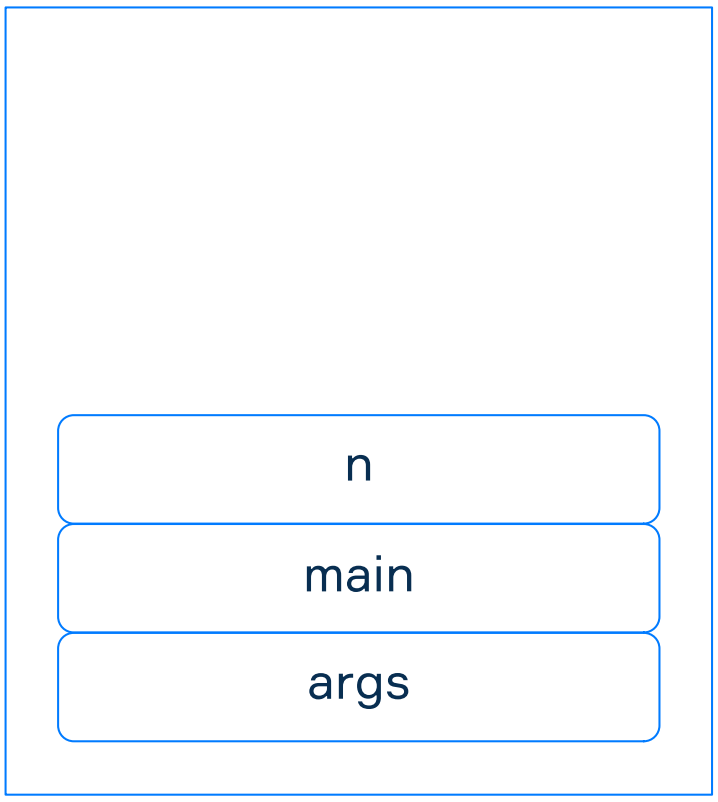
Topic is required for:

- Recursion

Table of contents:

return from the following method calls.
3. The local variables (`n`) are pushed on the stack.

The picture below presents the resulting call stack with `main` stack frame within.

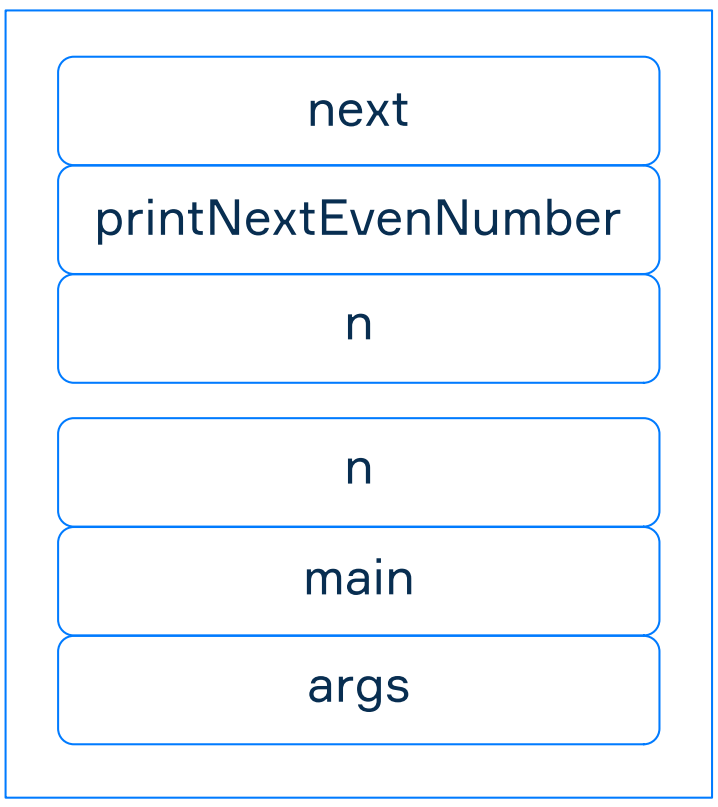


Actually, the stack stores just a reference to the `args` array since all reference types are stored in heap memory. But, the stack stores the actual value of `n` (which is 99 in our example).

§3. Stack and methods execution

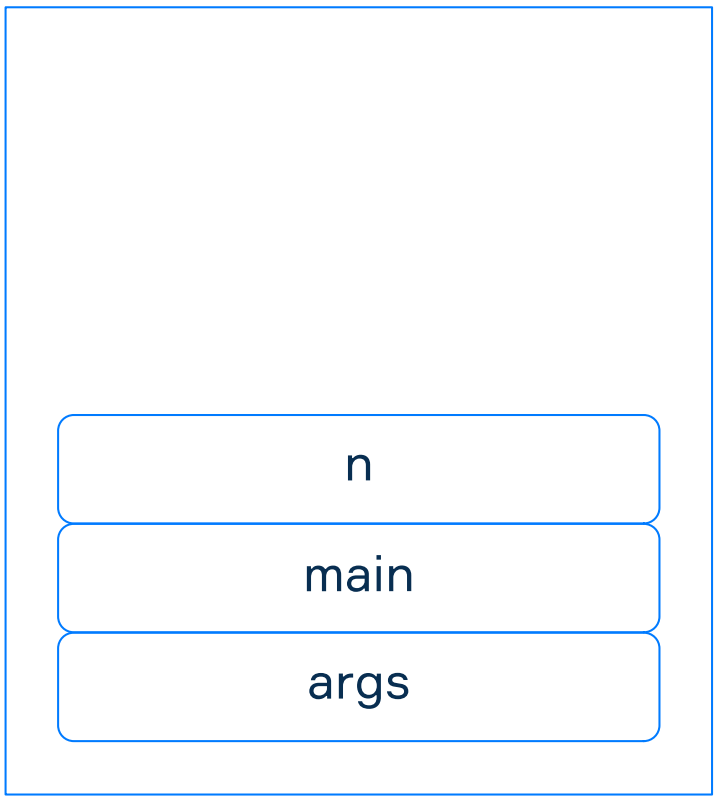
The next method to be invoked is `printNextEvenNumber`. As always, a new stack frame is created. The method parameters (`n`), address (`printNextEvenNumber` for simplicity), and local variables (`next`) are added to the new stack frame.

Now we have two complete stack frames for `main` and `printNextEvenNumber` methods within the execution stack:



Note, both frames have variables named `n`, but these variables are not the same since they belong to different methods.

Next, the program executes the method at the top of the call stack (`printNextEvenNumber`). After the execution, the current frame `printNextEvenNumber` is removed from the call stack and the previous frame `main` continues the execution.



The standard method `println` works in a similar way as the methods we have defined — the new stack frame is created and when `println` finishes its work, the `printNextEvenNumber` continues the execution.

Any Java program works almost in this way. When the stack is empty, the execution stops. We skip some details to simplify the explanation and give you only the general view.

§4. Stack overflow

The number of possible method invocations depends on the amount of memory allocated to the stack. When your stack contains too many stack frames, it can be overflowed. It leads to the `StackOverflowError` that will stop the execution. The stack size can be set with the `-Xss` command line switch like:

```
java YourProgramName -Xss256k
```

But we recommend you to be careful with it and read some articles on the Internet before modifying the default stack size. Also, sometimes the `StackOverflowError` points to an incorrect recursion calls in your program. In this case, increasing the size of the stack will not help you.

§5. Conclusion

- A call stack is a special data structure, following the LIFO rule, used by JVM to define methods execution order and to access methods information.
- A call stack consists of stack frames — stacks containing information about methods that were called and have not yet finished their execution.
- The machine executes the method on top of the call stack. If this method calls the new one, then the new stack frame is added to the call stack, and the execution goes to this new method, and so on. When the top method finishes the execution, the corresponding stack frame is removed from the call stack, and the execution goes to the next top method.
- Call stack containing too many stack frames may lead to `StackOverflowError`. Thus, you need to be careful using recursive calls in your methods.

Report a typo

219 users liked this theory. **4** didn't like it. **What about you?**

Start practicing

Comments (13) Hints (2) Useful links (0) Show discussion