# Tutorial

In this tutorial we will demonstrate how to use **Aprof**. Note, that this guide is actual for version 21.

- Getting Started
- Running Aprof
- Understanding Statistics
- Examples
    - Fibonacci Numbers

## Getting Started

First of all, extract *aprof.jar* from downloaded zip file with **Aprof** binaries. This JAR is the file you will be using.

Note, that you should not rename the *aprof.jar* file, otherwise it will not work.

## Running Aprof

**Aprof** comes with integrated help explaining all supported options and their default values. Execute the following line from the console to read it:

```
java -jar aprof.jar
```

You will see that in order to profile your Java application with **Aprof** you should start JVM with *-javaagent:aprof.jar* option. For example, to profile well-known *SwingSet2* demo you need to start it this way:

```
java -javaagent:aprof.jar -jar SwingSet2.jar
```

That's it! While you are using the demo, **Aprof** collects its memory allocation statistics and flushes it to *aprof.txt* file every minute.

## Understanding Statistics

In order to understand *aprof.txt* file you should be aware of the following specifics of **Aprof**.

Imagine we are interested in allocations of *char[]*. **Aprof** will collect all locations where these allocations take place. Naturally, some of them occur in constructor of *String* object. However, it is not enough to know that, for instance, 3 *char[]* with total size of 120 bytes were allocated at location *java.lang.String.<init>*. We are also interested in locations from which we called constructor of class *String*. In order to collect such additional information without taking stack-traces the following method is used.

In **Aprof** configuration some constructors/methods are marked as *tracked*. What exactly does that mean? When we mark the method as *tracked*, we tell **Aprof** that we are interested in occurrences of the method on stack-trace taken at the moment of memory allocation. The trick we use is the fact that knowledge of the outermost tracked method and location from which it was called is enough to pinpoint the problem. Hence, for each memory allocation the following data is collected by **Aprof**:

- type of allocated object;
- location where the allocation took place;
- the outermost tracked method (if any) on stack-trace of the allocation;
- location where the tracked method was called from.

These data is organized in a tree structure with types of objects being tree roots and reverse "stack-traces" of allocation context that can be at most 3 items deep as show in the above list. Items in the tree are ordered by the allocated size with most heavily allocated data types and most heavily used allocation context going first. Here is the example of how it might look like in the actual output:

```
char[]: 488,329,896 (63%) bytes in 1,602,894 (25%) objects (avg size 305 bytes)
        java.lang.AbstractStringBuilder.expandCapacity: 335,290,736 (68%) bytes in 170,262 (10%) objects (avg
size 1,969 bytes)
                java.lang.reflect.Constructor.newInstance: 310,510,624 (92%) bytes in 56,810 (33%) objects (avg
size 5,466 bytes)
                        SwingSet2.loadDemo: 310,508,240 (99%) bytes in 56,768 (99%) objects (avg size 5,470
bytes)
                        java.awt.AWTKeyStroke.getCachedStroke: 1,456 (0%) bytes in 26 (0%) objects (avg size 56
bytes)
                        sun.swing.SwingLazyValue.createValue: 928 (0%) bytes in 16 (0%) objects (avg size 58
bytes)
                java.lang.StringBuilder.append: 20,999,752 (6%) bytes in 35,880 (21%) objects (avg size 585
bytes)
                        DemoModule.loadSourceCode: 19,179,264 (91%) bytes in 1,412 (3%) objects (avg size
13,583 bytes)
                        com.devexperts.aprof.transformer.Context.getLocation: 662,968 (3%) bytes in 11,364
(31%) objects (avg size 58 bytes)
                        com.devexperts.aprof.transformer.MethodTransformer.visitAllocateArray: 355,408 (1%)
bytes in 6,336 (17%) objects (avg size 56 bytes)
                        com.devexperts.aprof.transformer.AbstractMethodVisitor.visitTypeInsn: 214,640 (1%)
bytes in 4,944 (13%) objects (avg size 43 bytes)
...
```

As you can see, **AProf** also profiles itself.

## Examples

Aforementioned SwingSet2 demo is a complex application, hence its profiling results are not suitable for this tutorial. We will be using a series of specifically designed samples. Some of them make no sense, some are written in non-optimal way, some might even contain bugs. Most likely you will never see them in real applications. However, they help us demonstrate key features of **Aprof**.

### Fibonacci Numbers

Let's take a look at the following program.

```
public class FibonacciNumbers {
        private static Integer fib(int n) {
                if (n < 2)
                        return 1;
                return fib(n - 1) + fib(n - 2);
        }


        public static void main(String[] args) {
                int n = Integer.parseInt(args[0]);
                System.out.printf("fib(%d)=%d\n", n, fib(n));
        }
}
```

It calculates $n^{th}$ Fibonacci number and prints it to *stdout*. However, the method returns *Integer* instead of *int* which leads to a lot of garbage generated by the program. In large applications it is usually hard to find all such ineffective pieces of code.

Let's run this program under **Aprof**.

```
java -javaagent:aprof.jar com.devexperts.sample.FibonacciNumbers 40
```

And look at generated file *aprof.txt*.

```
TOTAL allocation dump for 29,423 ms (0h00m29s)
Allocated 66,155,144 bytes in 2,870,108 objects in 1,108 locations of 230 classes
-------------------------------------------------------------------------------
java.lang.Integer: 34,953,568 (52%) bytes in 2,184,598 (76%) objects (avg size 16 bytes)
        java.lang.Integer.valueOf: 34,931,824 (99%) bytes in 2,183,239 (99%) objects
                FibonacciNumbers.fib: 34,852,928 (99%) bytes in 2,178,308 (99%) objects
...
```

We see that 52% of all memory allocations were made for *Integer* objects. And almost all of them were done in method *fib* of class *FibonacciNumbers*.