

# Soot overview/Disassembling classfiles

Raja Vallée-Rai (rvalleerai@sable.mcgill.ca)

March 15, 2000

## 1 Goals

By the end of this lesson, the student should be able to:

- understand what Soot is, and its two main uses
- have Soot correctly installed
- have the CLASSPATH environment variable properly set up
- produce baf, grimp or jimple code for any classfile

## 2 Testing your Installation

This is an interactive tutorial. So the first thing you must do is test your installation. This can be done by typing `java soot.Main` at the shell prompt. If your installation is incorrect you should get a class "soot.Main" not found exception. Please refer to the installation instructions which came with the Soot software if this occurs. If your installation is correct you should see something like:

```
> java soot.Main
Soot version 1.0.0 (build 1.beta.6.dev.60)
Copyright (C) 1997-2000 Raja Vallee-Rai (rvalleerai@sable.mcgill.ca).
All rights reserved.
```

Contributions are copyright (C) 1997-2000 by their respective contributors.  
See individual source files for details.

Soot comes with ABSOLUTELY NO WARRANTY. Soot is free software,  
and you are welcome to redistribute it under certain conditions.  
See the accompanying file 'license.html' for details.  
<...snip...>

## 3 What is Soot?

Soot has two fundamental uses; it can be used as a stand-alone command line tool or as a Java compiler framework. As a command line tool, Soot can:

1. disassemble classfiles
2. assemble classfiles
3. optimize classfiles

As a Java compiler framework, Soot can be used as a testbed for developing new optimizations. These new optimizations can then be added to the base set of optimizations invoked by the command line Soot tool. The optimizations that can be added can either be applied to single classfiles or entire applications.

Soot accomplishes these myriad tasks by being able to process classfiles in a variety of different forms. Currently Soot inputs two different intermediate representations (classfiles or Jimple code), and outputs 8 representations. By invoking Soot without arguments, you can see the output formats:

```
> java soot.Main
<...snip...> Output options:
  -b, --b                produce .b (abbreviated .baf) files
  -B, --baf              produce .baf code
  -j, --jimp             produce .jimp (abbreviated .jimple) files
  -J, --jimple           produce .jimple code
  -g, --grimp            produce .grimp (abbreviated .grimple) files
  -G, --grimple          produce .grimple files
  -s, --jasmin           produce .jasmin files
  -c, --class            produce .class files
<...snip...>
```

There are five intermediate representations currently being used in Soot: baf, jimple, grimp, jasmin, and classfiles. A brief explanation of each form follows:

baf a streamlined representation of bytecode. Used to inspect Java bytecode as stack code, but in a much nicer form. Has two textual representations (one abbreviated (.b files), one full (.baf files).)

jimple typed 3-address code. A very convenient representation for performing optimizations and inspecting bytecode. Has two textual representations (.jimp files, and .jimple files.)

grimp aggregated (with respect to expression trees) jimple. The best intermediate representation for inspecting disassembled code. Textual representations end with ".grimp".

jasmin a messy assembler format. Used mainly for debugging Soot. Jasmin files end with ".jasmin".

classfiles the original Java bytecode format. A binary (non-textual) representation. The usual .class files.

## 4 Setting up your CLASSPATH and generating a Jimple file

Soot looks for classfiles by examining your CLASSPATH environment variable or by looking at the contents of the soot-classpath command line option. Included in this lesson is the Hello.java program. Download this file, compile it (using javac or other compilers), and try the following command in the directory where Hello.class is located.

```
> java soot.Main --jimple Hello
```

This may or not work. If you get the following:

```
> java soot.Main --soot-classpath /localhome/plam Hello --jimple
Warning: Hello is a phantom class!
Jimplifying Hello...
Transforming Hello...
```

This means that a classfile is not being located. Either it can not find the Hello classfile, or it can not load the Java Development Kit libraries. Soot resolves classfiles by examining the directories in your CLASSPATH environment variable or the soot-classpath command line option.

Potential problem #1: Soot can not locate the Hello classfile. To make sure that it can find the classfile "Hello", (1) add "." to your CLASSPATH or (2) specify "." on the command line.

To carry out (1) on UNIX-style systems using bash,

```
> export CLASSPATH=$CLASSPATH:.
```

and on Windows machines,

```
> SET CLASSPATH=%CLASSPATH%;.
```

and to do (2),

```
> java soot.Main --soot-classpath . --jimple Hello
```

Potential problem #2: Soot cannot locate the class libraries. In this case, Soot will report that the type `"java.lang.Object"` could not be found.

Under JDK1.2, the class libraries do not need to be explicitly specified in the `CLASSPATH` for the Java Virtual Machine to operate. Soot requires them to be specified either on the `CLASSPATH` or in the `soot-classpath` command line option. Theoretically, this means adding the path to a `"rt.jar"` file to the `CLASSPATH` or the `soot-classpath`. This usually does not solve the problem because of a JDK bug concerning jar files, so in practice one must unjar this file. So there are two steps for getting Soot to work with the standard Java Development Kit class libraries.

**Important note for the release 1.2.1** Note that as of release 1.2.1, Soot fixed the jar problem, it can accept `'rt.jar'` now. You can use either the old way to unjar the `'rt.jar'` file, or supply it to `soot-classpath` command line option.

## 4.1 Locating the `rt.jar` file

It is usually in a directory of the form `"$root/jdk1.2.2/jre/lib"` where `$root` is `"/usr/local"` or some similarly named directory. If you can not find it, you can attempt a find command such as:

```
> cd /usr ; find . -name "rt.jar" -print
```

which may be able to locate it for you. Otherwise your best bet is to contact your system administrator.

## 4.2 Unjarring the `rt.jar` file

After having located the `rt.jar` file, you must unjar it. If you have access to the directory where it is located then you can simply issue:

```
> cd $root/jdk1.2.2/jre/lib
> mkdir rt
> cd rt
> jar xvf ../rt.jar
```

and add it to your classpath:

```
> export CLASSPATH=$CLASSPATH:$root/jdk1.2.2/jre/lib/rt
```

Otherwise you will need to unjar it in a separate location and then add it to your `CLASSPATH`. The `rt.jar` file must not appear in your `CLASSPATH` prior to the unjarred copy of the library.

```
> cd $HOME
> mkdir rt
> cd rt
> jar xvf $root/jdk1.2.2/jre/lib/rt.jar
```

**Important note for Windows users** Note that as of release 1, Soot will treat drive letters correctly, but under Windows the path separator *must* be a backslash (\), not a forward slash.

Summing up, you must issue a command of the form:

```
> export CLASSPATH=./usr/local/pkgsg/jdk1.2.2/jre/lib/rt
```

or if you use the soot-classpath option which is more cumbersome:

```
> java soot.Main --jimple --soot-classpath ./usr/local/pkgsg/jdk1.2.2/jre/lib/rt Hello
```

Once your *CLASSPATH* is set up properly, you should get:

```
> java soot.Main --jimple Hello
Transforming Hello...
```

The file called Hello.jimple should contain:

```
public class Hello extends java.lang.Object
{
    public void <init>()
    {
        Hello r0;

        r0 := @this: Hello;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }

    public static void main(java.lang.String[])
    {
        java.lang.String[] r0;
        java.io.PrintStream $r1;

        r0 := @parameter0: java.lang.String[];
        $r1 = <java.lang.System: java.io.PrintStream out>;
        virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String
)>("Hello world!");
        return;
    }
}
```

## 5 Generating jimple, baf, grimp for java.lang.String

By simple extrapolation, you should be able to now generate *.b*, *.baf*, *.jimp*, *.jimple*, *.grimp*, and *.grimple* files for any of your favorite classfiles. A particularly good test is a classfile from the JDK library. So a command like:

```
> java soot.Main --baf java.lang.String
```

should yield a file java.lang.String.baf containing text of the form:

```
public static java.lang.String valueOf(char[], int, int)
{
    word r0, i0, i1;
```

```

    r0 := @parameter0: char[];
    i0 := @parameter1: int;
    i1 := @parameter2: int;
    new java.lang.String;
    dup1.r;
    load.r r0;
    load.i i0;
    load.i i1;
    specialinvoke <java.lang.String: void <init>(char[],int,int)>;
    return.r;
}

```

## 6 History

- February 8, 2000: Initial version.
- February 16, 2000: Added changes for Soot version 021400 (Soot now prints the missing type) and emitted the title at the beginning. -PL
- March 1, 2000: Added changes for Release 1 (phantom class error printed instead) and emphasized that rt.jar should not occur in CLASSPATH. -PL
- March 11, 2000: Added note for Windows users in section about the classpath.
- March 15, 2000: Final tweaks for Release 1.
- January 29, 2001: Add the note of the release 1.2.1 .
- February 3, 2001: Added a hyperlink to Hello.java.