Creating a class from scratch with Soot

Feng Qian (fqian@sable.mcgill.ca)
Patrick Lam (plam@sable.mcgill.ca)
Chris Goard (cgoard@sable.mcgill.ca)

February 4, 2005

This tutorial is based on the createclass example, written by Raja-Vallée-Rai and distributed with the Ashes tools.

1 Goals

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

- name the basic classes of Soot and describe their functionality
- create a simple program which uses Soot to create a classfile from scratch.

The `createclass` example creates the Java class file `HelloWorld.class` from scratch, using the Soot framework.

The student should refer to the `Main.java` file, which puts all of the steps together in a working Java file. Even though a typical use of Soot would be to write a new `Transformer`, extending Soot's functionality, we illustrate a standalone application here; the same classes and methods are used in either case.

2 Creating a class file using Soot

First, we need to create a class to put methods into. The following steps are necessary to create a class file.

2.1 Loading `java.lang.Object` and Library Classes

Load `java.lang.Object`*, the root of the Java class hierarchy.*

This step is not necessary when building code that extends the Soot framework; in that case, loading of classfiles is already done when user code is called.

```
Scene.v().loadClassAndSupport("java.lang.Object");
```

This line of code causes Soot to load the `java.lang.Object` class and create the corresponding `SootClass` object, as well as `SootMethods` and `SootFields` for its fields. Of course, `java.lang.Object` has references to other objects. The call to `loadClassAndSupport` will load the transitive closure of the specified class, so that all types needed in order to load `java.lang.Object` are themselves loaded.

This process is known as *resolution*.

Since our HelloWorld program will be using classes in the standard library, we must also resolve these:

```
Scene.v().loadClassAndSupport("java.lang.System");
```

These lines reference `Scene.v()`. The `Scene` is the container for all of the `SootClasses` in a program, and provides various utility methods. There is a singleton `Scene` object, accessible by calling `Scene.v()`. *Implementation note:* Soot loads these classes from either classfiles or `.jimple` input files. When the former is used, Soot will load all class names referred to in the constant pool of each class file. Loading from `.jimple` will make Soot load only the required types.

2.2 Creation of a new `SootClass` object

Create the 'HelloWorld' `SootClass`*, and set its super class as "java.lang.Object".*

```
sClass = new SootClass("HelloWorld", Modifier.PUBLIC);
```

This code creates a `SootClass` object for a public class named `HelloWorld`.

```
sClass.setSuperclass(Scene.v().getSootClass("java.lang.Object"));
```

This sets the superclass of the newly-created class to the `SootClass` object for `java.lang.Object`. Note the use of the utility method `getSootClass` on the Scene.

```
Scene.v().addClass(sClass);
```

This adds the newly-created `HelloWorld` class to the `Scene`. All classes should belong to the `Scene` once they are created.

3 Adding methods to `SootClasses`

Create a `main()` *method for* `HelloWorld` *with an empty body.*

Now that we have a `SootClass`, we need to add methods to it.

```
method = new SootMethod("main",
    Arrays.asList(new Type[] {ArrayType.v(RefType.v("java.lang.String"), 1)}),
    VoidType.v(), Modifier.PUBLIC | Modifier.STATIC);
```

We create a new `public static` method, `main`, declare that it takes an array of `java.lang.String` objects, and that it returns `void`.

The constructor for `SootMethod` takes a list, so we call the Java utility method `Arrays.asList` to create a list from the one-element array which we generate on the fly with `new Type[]   ...   `. In the list, we put an array type, corresponding to a one-dimensional ArrayType of `java.lang.String` objects. The call to `RefType` fetches the *type* corresponding to the `java.lang.String` class.

Types Each `SootClass` represents a Java object. We can instantiate the class, giving an object with a given type. The two notions – type and class – are closely related, but distinct. To get the type for the `java.lang.String` class, by name, we call `RefType.v("java.lang.String")`. Given a `SootClass` object `sc`, we could also call `sc.getType()` to get the corresponding type.

```
sClass.addMethod(method);
```

This code adds the method to its containing class.

4 Adding code to methods

A method is useless if it doesn't contain any code. We proceed to add some code to the `main` method. In order to do so, we must pick an intermediate representation for the code.

4.1 Create JimpleBody

In Soot, we attach a *Body* to a SootMethod to associate some code with the method. Each Body knows which SootMethod it corresponds to, but a SootMethod only has one active Body at once (accessible via `SootMethod.getActiveBody()`). Different types of Body's are provided by the various intermediate representations; Soot has `JimpleBody`, `BafBody` and `GrimpBody`.

More precisely, a `Body` has three important features: chains of locals, traps and units. A *chain* is a list-like structure that provides *O(1)* access to insert and delete elements. *Locals* are the local variables in the body; *traps* say which units catch which exceptions; and *units* are the statements themselves.

Note that "unit" is the term which denotes both statements (as in Jimple) and instructions (as in Baf).

Create a Jimple Body for 'main' class, adding locals and instructions to body.

```
JimpleBody body = Jimple.v().newBody(method);
method.setActiveBody(body);
```

We call the Jimple singleton object to get a new `JimpleBody` associated with our method, and make it the active body for our method.

4.2 Adding a Local

```
arg = Jimple.v().newLocal("l0", ArrayType.v(RefType.v("java.lang.String"), 1));
body.getLocals().add(arg);
```

We create a few new Jimple `Locals` and add them to our Body.

4.3 Adding a Unit

```
units.add(Jimple.v().newIdentityStmt(arg,
      Jimple.v().newParameterRef(ArrayType.v
        (RefType.v("java.lang.String"), 1), 0)));
```

The `SootMethod` declares that it has parameters, but these are not bound to the locals of the Body. The IdentityStmt does this; it assigns into `arg` the value of the first parameter, which has type "array of strings".

```
// insert "tmpRef.println("Hello world!")"
{
    SootMethod toCall = Scene.v().getMethod
      ("<java.io.PrintStream: void println(java.lang.String)>");
    units.add(Jimple.v().newInvokeStmt
        (Jimple.v().newVirtualInvokeExpr
           (tmpRef, toCall.makeRef(), StringConstant.v("Hello world!"))));
}
```

We get the method with signature `<java.io.PrintStream:  void println(java.lang.String)>` (it is named `println`, belongs to `PrintStream`, returns `void` and takes a `String` as its argument – this is enough to uniquely identify the method), and invoke it with the `StringConstant` "Hello world!".

5 Write to class file

In order to write the program out to a `.class` file, the method bodies must be converted from Jimple to Jasmin, and assembled into bytecode. Assembly into bytecode is performed by a `JasminOutputStream`.

We first construct the output stream that will take Jasmin source and output a `.class` file. We can either specify the filename manually, or we can let soot determine the correct filename. We do the latter, here.

```
String fileName = SourceLocator.v().getFileNameFor(sClass, Options.output_format_class);
OutputStream streamOut = new JasminOutputStream(
                             new FileOutputStream(fileName));
PrintWriter writerOut = new PrintWriter(
                            new OutputStreamWriter(streamOut));
```

We now convert from Jimple to Jasmin, and print the resulting Jasmin class to the output stream.

```
JasminClass jasminClass = new soot.jimple.JasminClass(sClass);
jasminClass.print(writerOut);
writerOut.flush();
streamOut.close();
```

If we wished to output jimple source instead of a `.class` file, we would use the following code:

```
String fileName = SourceLocator.v().getFileNameFor(sClass, Options.output_format_jimple);
OutputStream streamOut = new FileOutputStream(fileName);
PrintWriter writerOut = new PrintWriter(
                            new OutputStreamWriter(streamOut));
Printer.v().printTo(sClass, writerOut);
writerOut.flush();
streamOut.close();
```

We have omitted the `JaminOuputStream`, and are calling the `printTo` method on `Printer`.

The Jimple created for the HelloWorld class is:

```
public class HelloWorld extends java.lang.Object
{
   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;
   }
}
```

6 Conclusion

We've seen how to use the basic objects and methods of Soot, and how to create Jimple statements. This tutorial was brought to you by these classes: `Scene, SootClass, SootMethod, Body, JimpleBody, Local,` and `Unit`.

Appendix A: Complete code for `createclass` example

The code for this example is reproduced below. It can be downloaded at:
`http://www.sable.mcgill.ca/soot/tutorial/createclass/Main.java.`

```
/* Soot - a J*va Optimization Framework
 * Copyright (C) 1997-1999 Raja Vallee-Rai
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

/*
 * Modified by the Sable Research Group and others 1997-1999.
 * See the 'credits' file distributed with Soot for the complete list of
 * contributors.  (Soot is distributed at http://www.sable.mcgill.ca/soot)
 */


import soot.*;
import soot.jimple.*;
import soot.options.Options;
import soot.util.*;
import java.io.*;
import java.util.*;

/** Example of using Soot to create a classfile from scratch.
 * The 'createclass' example creates a HelloWorld class file using Soot.
 * It proceeds as follows:
 *
 * - Create a SootClass <code>HelloWorld</code> extending java.lang.Object.
 *
 * - Create a 'main' method and add it to the class.
 *
 * - Create an empty JimpleBody and add it to the 'main' method.
 *
 * - Add locals and statements to JimpleBody.
 *
 * - Write the result out to a class file.
 */

public class Main
{
    public static void main(String[] args) throws FileNotFoundException, IOException
```

```
{
    SootClass sClass;
    SootMethod method;

    // Resolve Dependencies
       Scene.v().loadClassAndSupport("java.lang.Object");
       Scene.v().loadClassAndSupport("java.lang.System");

    // Declare 'public class HelloWorld'
       sClass = new SootClass("HelloWorld", Modifier.PUBLIC);

    // 'extends Object'
       sClass.setSuperclass(Scene.v().getSootClass("java.lang.Object"));
       Scene.v().addClass(sClass);

    // Create the method, public static void main(String[])
       method = new SootMethod("main",
            Arrays.asList(new Type[] {ArrayType.v(RefType.v("java.lang.String"), 1)}),
            VoidType.v(), Modifier.PUBLIC | Modifier.STATIC);

       sClass.addMethod(method);

    // Create the method body
    {
        // create empty body
        JimpleBody body = Jimple.v().newBody(method);

        method.setActiveBody(body);
        Chain units = body.getUnits();
        Local arg, tmpRef;

        // Add some locals, java.lang.String l0
            arg = Jimple.v().newLocal("l0", ArrayType.v(RefType.v("java.lang.String"), 1));
            body.getLocals().add(arg);

        // Add locals, java.io.printStream tmpRef
            tmpRef = Jimple.v().newLocal("tmpRef", RefType.v("java.io.PrintStream"));
            body.getLocals().add(tmpRef);

        // add "l0 = @parameter0"
            units.add(Jimple.v().newIdentityStmt(arg,
                 Jimple.v().newParameterRef(ArrayType.v(RefType.v("java.lang.String"), 1), 0)));

        // add "tmpRef = java.lang.System.out"
            units.add(Jimple.v().newAssignStmt(tmpRef, Jimple.v().newStaticFieldRef(
                Scene.v().getField("<java.lang.System: java.io.PrintStream out>").makeRef())));

        // insert "tmpRef.println("Hello world!")"
        {
            SootMethod toCall = Scene.v().getMethod("<java.io.PrintStream: void println(java.lan
            units.add(Jimple.v().newInvokeStmt(Jimple.v().newVirtualInvokeExpr(tmpRef, toCall.ma
            }

            // insert "return"
                units.add(Jimple.v().newReturnVoidStmt());

        }

        String fileName = SourceLocator.v().getFileNameFor(sClass, Options.output_format_class);
        OutputStream streamOut = new JasminOutputStream(
                                    new FileOutputStream(fileName));
        PrintWriter writerOut = new PrintWriter(
                                    new OutputStreamWriter(streamOut));
        JasminClass jasminClass = new soot.jimple.JasminClass(sClass);
        jasminClass.print(writerOut);
        writerOut.flush();
        streamOut.close();
    }

}
```

7 History

- March 8, 2000: Initial version.
- September 1, 2000: Changed syntax to conform with the current release.
- May 31, 2003: Updated for Soot 2.0.
- February 4, 2005: Updated for Soot 2.2.