

Jakarta Debugging Support for Other Languages

Jakarta Server Pages Team, <https://projects.eclipse.org/projects/ee4j.jsp>

2.0, October 07, 2020:

Table of Contents

Eclipse Foundation Specification License	1
Disclaimers	2
1. Goal	3
2. Terminology	4
3. Approach	5
3.1. Single Translation	5
3.2. Multiple Translations	5
3.3. Diagram	6
4. Scope	7
4.1. Variables	7
4.2. Multi-Level Source View	7
4.3. Finding Source Files	7
4.4. Multiple Source Files per Class File	7
5. Source Map Format	8
5.1. General Format	8
5.2. Header	8
5.3. StratumSection	8
5.4. FileSection	9
5.5. LineSection	9
5.6. VendorSection	11
5.7. EndSection	11
5.8. Embedded Source Maps	12
5.9. SMAP Syntax	12
6. SMAP Resolution	15
6.1. LineInfo Composition Algorithm	16
6.2. Resolution Example	17
7. JPDA Support	21
8. SourceDebugExtension Support	22
8.1. SourceDebugExtension Access	22
8.2. SourceDebugExtension Class File Attribute	22
9. Example	24
9.1. Input Source	24
9.2. Language Processor	24
9.3. Post Processor	26
9.4. Debugging	26

Specification: Jakarta Debugging Support for Other Languages

Version: 2.0

Status: Final Release

Release: October 07, 2020

Copyright (c) 2018, 2020 Eclipse Foundation.

Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

¥ link or URL to the original Eclipse Foundation document.

¥ All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [\$date-of-document] Eclipse Foundation, Inc. <<url to this license>>"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright © 2018, 2020 Eclipse Foundation. This software or document includes material copied from or derived from Jakarta Debugging Support for Other Languages
<https://jakarta.ee/specifications/debugging/2.0/>"

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Chapter 1. Goal

A mechanism is needed by which programs executed under the Java™ virtual machine but written in languages other than the Java programming language, can be debugged with references to the original source (for example, source file and line number references).

Constraints:

- ¥ No change to the Java programming language.
- ¥ Optional change to Java programming language compiler.
- ¥ No change to JPDA clients (debuggers, É) for basic functionality. With the exception being, tools that arbitrarily prohibit non-Java programming language source.
- ¥ Minimal change to Java virtual machine.
- ¥ No change to the Java platform class libraries.

Chapter 2. Terminology

Term	Definition
final-source	The final form of source. This source will be compiled into a class file . Typically, final-source is Java programming language source.
translated-source	Source that will be translated by a language-processor into another language or form.
language-processor	Converts translated-source to final-source or to different translated-source .
class file	A collection of bytes in Java virtual machine class file format.
compiler	Compiler which converts final-source to class files. For example, <code>javac</code> is a compiler.
post-processor	Takes a class file and a Source Map File as input; generates a class file as output. Inserts a SourceDebugExtension attribute.
Source Map SMAP	Specifies the mapping of source between one or more sets of input source and the output source. See: Source Map Format .
Source Map File SMAP-file	Specifies the mapping created by a language-processor between translated-source input and the resultant output source. It is a Source Map stored in a file in Source Map Format .
stratum	A view of a class from a particular, named, programming language level.
JPDA	The Java Platform Debugger Architecture .
JDI	The Java Debug Interface, which is the high-level Java programming language interface of the Java Platform Debugger Architecture . See: JDI Specification .
SourceDebugExtension attribute	A Java virtual machine class file attribute that holds information about the source. In this case, the information is a Source Map in Source Map Format . The source is mapped to final-source (output source). The Source Map has potentially multiple sets of input source (strata). See: SourceDebugExtension Class File Attribute

Chapter 3. Approach

3.1. Single Translation

A [language-processor](#) translates [translated-source](#) to [final-source](#) (the case of translation from one [translated-source](#) to another translated-source, is addressed [below](#)). It also creates a second output, an [SMAP-file](#), whose format is described in [Source Map Format](#). This file describes the mapping between input source and output source (e.g. line number and source file).

The [final-source](#) generated by the [language-processor](#) is compiled by the [compiler](#).

The [post-processor](#) takes the class file generated by the [compiler](#) and the [SMAP-file](#) as input. A [SourceDebugExtension](#) attribute containing the SMAP in the SMAP-file is added to the class file and the new class file is written.

Optionally, the [compiler](#) may take both [final-source](#) and the SMAP-file as input, and perform both compilation and installation of the [SourceDebugExtension](#).

When the resultant program is debugged using a debugging tool based on the Java Debug Interface (JDI) of [JPDA](#), the final-source line number information is converted to the specified language view ([strata](#)).

3.2. Multiple Translations

A language-processor might translate source into source which will become input to another language-processor, and so on. Eventually, after possibly many translated-source forms, final-source is produced. Each translation produces SMAP information. This information must be preserved and placed in context, so that each [stratum](#) can be mapped to the final-source.

A language-processor checks for an [SMAP-file](#) in a location parallel to that of the input source. For example, if the source repository is a file system and the input source is located at [path name.extension](#) then [path name.extension.smap](#) will be checked for an SMAP. The input SMAPs will be copied into the generated SMAP. See [Embedded Source Maps](#) for specifics on the embedding of input SMAPs.

In the case of multiple translations, the [post-processor](#) must resolve the embedded SMAPs. See [SMAP Resolution](#).

Note that final-source need not be Java programming language source, as compilers for other languages may directly generate class files, including the [SourceFile](#) and [LineNumberTable](#) class file attributes. SMAPs and the mechanism presented here are still useful for handling multiple translations.

A programming language implementor, directly generating class files might also choose to generate SMAPs (thus functioning as both language-processor and compiler) since the SMAP is useful for describing source configurations (such as multiple source files per class file) which cannot be

represented with the `SourceFile` and `LineNumberTable` attributes. In this case, the input is translated-source and the final-source is represented in the attributes but is never generated.

3.3. Diagram

This diagram demonstrates data flow. The particular case shown has two levels of translation, with file inclusion on the second level (as is the case in the example in [SMAP Resolution](#)).

Where `TS` is `translated-source` and `FS` is `final-source`.

Chapter 4. Scope

4.1. Variables

The complexity of mapping semantics (like variable and data views) across languages is such that this feature has not been included in this version of the specification and will be considered for a future version.

4.2. Multi-Level Source View

The ability to choose the source level to view is addressed in this specification. These are referred to as [strata](#).

4.3. Finding Source Files

Currently, [final-source](#) is found by combining the follow elements:

- ¥ A source path
- ¥ The package name converted to a directory path
- ¥ The source file name from a [JDI](#) call (derived from the [SourceFile](#) class file attribute)

Since existing debuggers use this mechanism (the only way for an existing debugger to find [translated-source](#)) each aspect must be addressed:

- ¥ The source path must be set-up to include translated-source directories
- ¥ Source must be placed in a directory corresponding to the package
- ¥ The [JDI](#) call must return the translated-source name.

For debuggers written against the new APIs, a new method has been added which returns the source path - this makes the translated-source directory structure flexible.

4.4. Multiple Source Files per Class File

When an inclusion mechanism is used, a class file will contain source from multiple [translated-source](#) files. The [SourceFile](#) attribute of class files only associates one source file with a class file which is one reason the approach of simply rewriting the [SourceFile](#) and [LineNumberTable](#) attributes had to be abandoned. The SMAP allows a virtually unlimited number of source files per stratum.

Chapter 5. Source Map Format

A Source Map (SMAP) describes a mapping between source positions in an input language ([translated-source](#)) and source positions in a generated output language. A view of the source through such a mapping is called a [stratum](#). The [SMAP-file](#) contains an unresolved SMAP. The [SourceDebugExtension](#) class file attribute, when used as described in this document, contains an SMAP. The SMAP stored in a [SourceDebugExtension](#) attribute must be [resolved](#), and thus will have no [embedded SMAPs](#) and will have the [final-source](#) language as the output language.

An SMAP consists of a header and one or more sections of mapping information.

There are currently seven types of section: stratum sections, file sections, line sections, vendor sections, end sections, and open and close embedded sections. New section types may be added in the future - to facilitate this, any unknown sections must be ignored without error.

The semantics of each section is discussed below. For clarity, an informal description of the syntax of each section is included in the discussion. See the formal [SMAP syntax](#) for syntax questions.

5.1. General Format

The SMAP consists of lines of [Unicode](#) text, with a concrete representation of [UTF-8](#). Line termination is with line-feed, carriage-return or carriage-return followed by line-feed. Because SMAPs are included in class files, size of the SMAP was an important constraint on the format chosen for them.

5.2. Header

The first line of an SMAP is the four letters [SMAP](#) which identifies it as an SMAP. The next line is the name of the generated file. This name is without path information (and thus if the generated file is [final-source](#), the name should match the [SourceFile](#) class file attribute). The last line of the header is the default stratum for this class. The default stratum is the stratum used when a debugger does not explicitly specify interest in another stratum. In an unresolved SMAP the default stratum can be unspecified (blank line). In a resolved SMAP the default stratum must be specified. A specified stratum must either be one represented with a stratum section or [Java](#) which indicates the standard [final-source](#) information should be used by default.

5.3. StratumSection

An SMAP may map more than one [translated-source](#) to the output source (the output source is [final-source](#) if the SMAP is in a [SourceDebugExtension](#)). A view of the source is a stratum (whether viewed as [translated-source](#) or [final-source](#)). Each [translated-source](#) language should have its own stratum section with a unique stratum name. The [final-source](#) stratum (named `0Java0`) is created automatically and should not have a stratum section. The stratum section should be followed by a file section and a line section which will be associated with that stratum.

The format of the section is simply the stratum section marker `*S` followed by the name of the stratum. The section ends with a line termination. One [FileSection](#) and one [LineSection](#) (in either order) must follow the [StratumSection](#) (before the next [StratumSection](#) or the [EndSection](#)). One or more [VendorSections](#) may follow a [StratumSection](#). There must be at least one [StratumSection](#).

5.4. FileSection

The file section describes the translated-source file names. Each line maps a file ID to a source name and, optionally, to a source path. File IDs are used only in the [LineSection](#). The source name is the name of the translated-source. The source path is the path to the translated-source, the `/` symbol is translated to the local file separator. In the case where the source repository is a file system, source name is the file name (without directory information) and source path is a path name (often relative to one of the compilation source paths). For example: `Bar.foo` would be a source name, and `here/there/Bar.foo` would be a source path. The first file line denotes the primary file.

The format of the file section is the file section marker `*F` on a line by itself, followed by file information. File information has two forms, source name only and source name / source path. The source name only form is one line: the integer file ID followed by the source name. The source name / source path form is two lines: a plus sign `+`, file ID, and source name on the first line and the source path on the second. The file ID must be unique within the file section. A [FileSection](#) may only occur after a [StratumSection](#). The `FileName` must have at least one character. The `AbsoluteFileName`, if specified, must have at least one character.

For example:

```
*F
+ 1 Foo.xyz
here/there/Foo.xyz
2 Incl.xyz
```

declares two source files. File ID #1 has source name `Foo.xyz` and source path `here/there/Foo.xyz`. File ID #2 has source name `Incl.xyz` and a source path to be computed by the debugger.

5.5. LineSection

The [line section](#) associates line numbers in the output source with line numbers and source names in the input source.

The format of the line section is the line section marker `*L` on a line by itself, followed by the lines of [LineInfo](#). Each [LineInfo](#) has the form:

```
InputStartLine # LineFileID , RepeatCount : OutputStartLine , OutputLineIncrement
```

where all but

$$\text{InputStartLine} : \text{OutputStartLine}$$

are optional.

A range of output source lines is mapped to a single input source line. Each [LineInfo](#) describes [RepeatCount](#) of these mappings. [OutputLineIncrement](#) specifies the number of lines in the output source range; this line increment is applied to each mapping in the [LineInfo](#). The source file containing the input source line is specified by [LineFileID](#) via the [FileSection](#).

More precisely, for each n between zero and

$$\text{RepeatCount} - 1$$

the input source line number

$$\text{InputStartLine} + n$$

maps to the output source line numbers from

$$\text{OutputStartLine} + (n * \text{OutputLineIncrement})$$

through

$$\text{OutputStartLine} + ((n + 1) * \text{OutputLineIncrement}) - 1$$

If absent, [RepeatCount](#) and [OutputLineIncrement](#) default to one. If absent, [LineFileID](#) defaults to the most recent value (initially zero).

The first line of a file is line one. [RepeatCount](#) is greater than or equal to one. Each [LineFileID](#) must be a file ID present in the [FileSection](#). [InputStartLine](#) is greater than or equal to one. [OutputStartLine](#) is greater than or equal to one. [OutputLineIncrement](#) is greater than or equal to zero. A [LineSection](#) may only occur after a [StratumSection](#).

For example:

```
*L
123: 207
130, 3: 210
140: 250, 7
160, 3: 300, 2
```

Creates this mapping:

Input Source	Output Source	
Line	Begin Line	End Line
123	207	207
130	210	210
131	211	211
132	212	212
140	250	256
160	300	301
161	302	303
162	304	305

Note that multiple [LineInfo](#) may map multiple input source lines to a single output source line, when such a [LineSection](#) is being used to map output source lines to input source lines, a first matching [LineInfo](#) rule applies.

Note also that multiple [LineInfo](#) may map a single input source line to a multiple, possibly disjoint, output source lines, when such a [LineSection](#) is being used to map input source lines to output source lines, a first matching [LineInfo](#) rule again applies.

5.6. VendorSection

The vendor section is for vendor specific information.

The format is ***V** on the first line to mark the section. The second line is the vendor ID which is formed by the same rules by which unique package names are formed in the Java language specification, third edition ([7.7](#)) [Unique Package Names](#).

5.7. EndSection

The end section marks the end of an SMAP, it consists simply of a ***E** marker. The end section must be the last line of an SMAP.

5.8. Embedded Source Maps

The `OpenEmbeddedSection` marks the beginning and `CloseEmbeddedSection` the end of a set of `EmbeddedSourceMaps`. These SMAPs correspond to the input source for a language-processor. The stratum of the language-processor is indicated on both sections. These sections must not occur in a `resolved` SMAP.

The format is the `*0` marker and the name of the output stratum on the first line. This is followed by the set of embedded SMAPs. The embedded SMAPs are included "whole" - from the `SMAP` to the `EndSection *E` marker - inclusive. Finally, the `*C` marker and the name of the output stratum on the last line terminates the embedded SMAPs.

5.9. SMAP Syntax

```

SMAP:
Ê          Header { Section } EndSection

Header:
Ê          ID OutputFileName DefaultStratumId

ID:
Ê          SMAP CR

OutputFileName:
Ê          NONASTERISKSTRING CR

DefaultStratumId:
Ê          NONASTERISKSTRING CR

Section:
Ê          StratumSection
Ê          FileSection
Ê          LineSection
Ê          EmbeddedSourceMaps
Ê          VendorSection
Ê          FutureSection

EmbeddedSourceMaps:
Ê          OpenEmbeddedSection { SMAP } CloseEmbeddedSection

OpenEmbeddedSection:
Ê          *0 StratumID CR

CloseEmbeddedSection:
Ê          *C StratumID CR

```

StratumSection:

Ê *S StratumID CR

StratumID:

Ê NONASTERISKSTRING

LineSection:

Ê *L CR { LineInfo }

LineInfo:

Ê InputLineInfo : OutputLineInfo CR

InputLineInfo:

Ê InputStartLine , RepeatCount

Ê InputStartLine

OutputLineInfo:

Ê OutputStartLine , OutputLineIncrement

Ê OutputStartLine

InputStartLine:

Ê NUMBER

Ê NUMBER # LineFileID

LineFileID:

Ê FileID

RepeatCount:

Ê NUMBER

OutputStartLine:

Ê NUMBER

OutputLineIncrement:

Ê NUMBER

FileSection:

Ê *F CR { FileInfo }

FileInfo:

Ê FileID FileName CR

Ê + FileID FileName CR AbsoluteFileName CR

FileID:

Ê NUMBER

FileName:

Ê NONASTERISKSTRING

```

AbsoluteFileName:
    Ê          NONASTERISKSTRING

VendorSection:
    Ê          *V CR VENDORID CR { VendorInfo }

VendorInfo:
    Ê          NONASTERISKSTRING CR

FutureSection:
    Ê          * OTHERCHAR CR { FutureInfo }

FutureInfo:
    Ê          NONASTERISKSTRING CR

EndSection:
    Ê          *E CR

```

Where $\{x\}$ denotes zero or more occurrences of x . And where the terminals are defined as follows (whitespace is a sequence of zero or more spaces or tabs):

Terminals	
NONASTERISKSTRING	Any sequence of characters (excluding the terminal carriage-return or new-line) which does not start with "*". Leading whitespace is ignored.
NUMBER	Non negative decimal integer. The number is terminated by the first non-digit character. Leading and trailing whitespace is ignored.
CR	a line terminator: carriage-return, carriage-return followed by new-line or new-line.
OTHERCHAR	Any character (other than carriage-return, new-line, space or tab) not already used as a section header (not S,F,L,V,O,C or E).
VENDORID	A sequence of characters that identifies a vendor. The name is formed by the same rules that unique package names are formed in the Java language specification. Leading and trailing whitespace is ignored. The terminal carriage-return or new-line is excluded.

Chapter 6. SMAP Resolution

Before the SMAP in a SMAP-file can be installed into the `SourceDebugExtension` attribute it must be resolved into an SMAP with no embedded SMAPs and with `final-source` as the output source. A set of [embedded SMAPs](#) is specific to a stratum and is resolved in the context of the matching `StratumSection` in the outer SMAP. The resolved SMAP includes `StratumSections` computed from each set of embedded SMAPs as well as the unchanged `StratumSections` of the outer SMAP. If embedded SMAPs are nested, the inner-most is resolved first.

The structure of an SMAP with embedded SMAPs is as follows:

```
SMAP
...
*O B
SMAP
...
*S A
...
*E
*C B
...
*S B
...
*E
```

The structure is a set of embedded SMAPs (for a stratum, here named *B*), an outer `StratumSection` (for *B*), and an embedded SMAP with a `StratumSection` (for a stratum, here named *A*). Note that: there may be many sets of embedded SMAPs, many embedded SMAPs within the set of embedded SMAPs, and many `StratumSections` within an SMAP. A `StratumSection` maps source information from its stratum to an output stratum. Thus, the embedded `StratumSection` maps stratum *A* to stratum *B*. We know it is mapped to stratum *B* because the set of embedded SMAPs for stratum *B* corresponds to the input for the language-processor for *B*. The outer `StratumSection` maps stratum *B* to its output stratum (let's call this stratum *C*), if the shown SMAP is the outer-most SMAP then stratum *C* is the final-source stratum. The purpose of resolution is to create a non-embedded `StratumSection` for *A* which maps to *C* (all `StratumSections` within an SMAP must map to the same output stratum, in a resolved outer-most SMAP all `StratumSections` will map to the final-source stratum). This is done by composing the mapping in the embedded `StratumSection` (from *A* to *B*) with the mapping in the outer `StratumSection` (from *B* to *C*). Since there may be many embedded `StratumSections` for *A*, these sections must be merged.

A `StratumSection` is computed for each stratum present in the embedded SMAPs. The computed `StratumSection` is the merge of each embedded `StratumSection`, for that stratum. Line number information is composed with the line number information of the outer `StratumSection` (note that the embedded `StratumSections` cannot be for the same stratum as the outer `StratumSection`). Specifically, a computed `StratumSection` consists of a merged [FileSection](#), a composed [LineSection](#), and direct copies

of any [VendorSections](#) or unknown sections. The merged FileSection includes each unique [FileInfo](#), with FileIDs reassigned to be unique. The composition the LineSections is described in the algorithm below.

6.1. LineInfo Composition Algorithm

The following pseudo-code sketches the algorithm for resolving LineInfo in embedded SMAPs. LineInfo resolution is by composition - discussed above. An embedded LineInfo which maps stratum *A* to stratum *B* is composed with an outer LineInfo which maps stratum *B* to stratum *C* to create a new resolved LineInfo which maps stratum *A* to stratum *C*.

The SMAPs and their components are marked by subscript:

¥ Embedded SMAP - level_E

¥ Outer StratumSection - level_O

¥ Resolved computed StratumSection - level_R

The inputs and outputs of the algorithm are LineInfo tuples. Line information is represented in this algorithm in its [LineInfo format](#) which is discussed in the [LineSection](#). This algorithm is invoked for each LineInfo_E in each embedded SMAP.

ResolveLineInfo:

Ê InputStartLine_E #LineFileID_E, RepeatCount_E : OutputStartLine_E, OutputLineIncrement_E
as follows:

```

if RepeatCountE > 0 then {
Ê for each LineInfo0 in the stratum of the embedded SMAP:
Ê   InputStartLine0 #LineFileID0, RepeatCount0: OutputStartLine0, OutputLineIncrement0
Ê   which includes OutputStartLineE
Ê   that is, InputStartLine0 + N == OutputStartLineE
Ê     for some offset into the outer input range N where 0 ! N < RepeatCount0
Ê and for which LineFileID0 has a sourceName matching the embedded SMAP's OutputFileName {
Ê   compute the number of outer mapping repetitions which can be applied
Ê   available := RepeatCount0 - N ;
Ê   compute the number of embedded mapping repetitions which can be applied
Ê   completeCount := floor(available / OutputLineIncrementE) min RepeatCountE ;
Ê   if completeCount > 0 then {
Ê     output resolved LineInfo
Ê       InputStartLineE # unify(LineFileIDE), completeCount :
Ê       (OutputStartLine0 + (N * OutputLineIncrement0)),
Ê       (OutputLineIncrementE * OutputLineIncrement0) ;
Ê     ResolveLineInfo
Ê       (InputStartLineE + completeCount) #LineFileIDE, (RepeatCountE - completeCount) :
Ê       (OutputStartLineE + completeCount * OutputLineIncrementE), OutputLineIncrementE ;
Ê   } else {
Ê     output resolved LineInfo
Ê       InputStartLineE # unify(LineFileIDE), 1 :
Ê       (OutputStartLine0 + (N * OutputLineIncrement0)), available ;
Ê     ResolveLineInfo
Ê       InputStartLineE #LineFileIDE, 1 :
Ê       (OutputStartLineE + available), (OutputLineIncrementE - available) ;
Ê     ResolveLineInfo
Ê       (InputStartLineE + 1) #LineFileIDE, (RepeatCountE - 1):
Ê       (OutputStartLineE + OutputLineIncrementE), OutputLineIncrementE ;
Ê   }
Ê }
}
```

where *unify* converts a LineFileID_E to a corresponding LineFileID_R

6.2. Resolution Example

The following example demonstrates resolution with this algorithm. The [general example](#) will provide context before walking through this example. In this example, **Incl.bar** is included by **Hi.bar**, but each is the result of a prior translation.



If the unresolved SMAP (in `Hi.java.smap`) is as follows:

SMAP Hi.java Java	<i>Outer Header</i>
*O Bar	<i>OpenEmbeddedSection</i>
SMAP Hi.bar Java *S Foo *F 1 Hi.foo *L 1#1, 5: 1, 2 *E	<i>Embedded SMAP (Hi.bar)</i>
SMAP Incl.bar Java *S Foo *F 1 Incl.foo *L 1#1, 2: 1, 2 *E	<i>Embedded SMAP (Incl.bar)</i>
*C Bar	<i>CloseEmbeddedSection</i>
*S Bar *F 1 Hi.bar 2 Incl.bar *L 1#1: 1 1#2, 4: 2 3#1, 8: 6	<i>Outer StratumSection</i>

*E	Final EndSection
----	------------------

The merged level_R FileSection is (in stratum Foo):

```
*F
1 Hi . foo
2 Incl . foo
```

The computation proceeds as follows:

LineInfo _E	LineInfo _E recursion 1	LineInfo _E recursion 2	matching outer LineInfo _O	resolved LineInfo _R	discussion
1#1,5:1,2			1#1:1	1#1,1:1,1	ResolveLineInfo is called for 1#1, 5: 1, 2 (from the first embedded SMAP - OutputFileName is Hi . bar). 1#1: 1 is found as the outer StratumSection LineInfo _O with InputStartLine0 of 1 and LineField0 has a sourceName matching Hi . bar. N is 0, and completeCount is 0, thus the else branch is taken. available is 1 and thus output is 1#1, 1: 1, 1.
	1#1,1:2,1		no match		The remaining half of the initial LineInfo _E mapping then must be resolved recursively, but there is no match and it is ignored.
	2#1,4:3,2		3#1,8:6	2#1,4:6,2	The remaining mappings are also handled recursively. There is a matching LineInfo _O . N is 0, and completeCount is 4, thus the if branch is taken.
		6#1,0:11,2	n/a		The recursive resolve descends deeper but does nothing since all of RepeatCount _E has been mapped. The first LineInfo _E is now resolved. Since it had only one LineInfo _E the first SMAP is also resolved.

LineInfo _E	LineInfo _E recursion 1	LineInfo _E recursion 2	matching outer LineInfo _O	resolved LineInfo _R	discussion
1#1,2:1,2			1#2,4:2	1#2,2:2,2	Now for the second SMAP (OutputFileName is Incl . bar). FileID _E 1 in this SMAP is Incl . foo which corresponds to the remapped FileID _R 2. So the matching LineInfo _O is 1#2, 4: 2 . N is 0, and completeCount is 2, so the <i>if</i> branch is taken.
	3#1,0:5,2		n/a		The recursive resolve does nothing since all the maps have been handled. Resolution is complete.

The resultant resolved SMAP is:

```

SMAP
Hi . java
Java
*S Foo
*F
1 Hi . foo
2 Incl . foo
*L
1#1, 1: 1, 1
2#1, 4: 6, 2
1#2, 2: 2, 2
*S Bar
*F
1 Hi . bar
2 Incl . bar
*L
1#1: 1
1#2, 4: 2
3#1, 8: 6
*E

```

Chapter 7. JPDA Support

The [Java Platform Debugger Architecture](#) in the Java 1.4 release was extended in support of debugging other languages. The new APIs and APIs with comments changed to include reference to [strata](#) are listed below:

New APIs	APIs with Changed Comments
JVMDI	
<code>getSourceDebugExtension</code>	
JDWP - ReferenceType (2) Command Set	
<code>SourceDebugExtension Command (12)</code>	
JDWP - VirtualMachine (1) Command Set	
<code>SetDefaultStratum Command (19)</code>	
JDI - VirtualMachine interface	
<code>void setDefaultStratum(String stratum)</code>	
<code>String getDefaultStratum()</code>	
JDI - ReferenceType interface	
<code>String sourceNames(String stratum)</code>	<code>String sourceName()</code>
<code>String sourcePaths(String stratum)</code>	
<code>List allLineLocations(String stratum, String sourceName)</code>	<code>List allLineLocations()</code>
<code>List locationsOfLine(String stratum, String sourceName, int lineNumber)</code>	<code>List locationsOfLine(int lineNumber)</code>
<code>List availableStrata()</code>	
<code>String defaultStratum()</code>	
<code>String sourceDebugExtension()</code>	
JDI - Method interface	
<code>List allLineLocations(String stratum, String sourceName)</code>	<code>List allLineLocations()</code>
<code>List locationsOfLine(String stratum, String sourceName, int lineNumber)</code>	<code>List locationsOfLine(int lineNumber)</code>
JDI - Location interface	
	<code>class comment</code> (strata defined)
<code>int lineNumber(String stratum)</code>	<code>int lineNumber()</code>
<code>String sourceName(String stratum)</code>	<code>String sourceName()</code>
<code>String sourcePath(String stratum)</code>	
<code>String sourcePath()</code>	

Chapter 8. SourceDebugExtension Support

Debugger applications frequently need debugging information about the source that exceeds what is delivered by the existing Java™ Virtual Machine class file attributes (SourceFile, LineNumber, and LocalVariable). This is particularly true for debugging the source of other languages. In a distributed environment side files may not be accessible, the information must be directly associated with the class.

The solution is the addition of a class file attribute which holds a string. The string contains debugging information in a standardized format which allows for evolution and vendor extension.

8.1. SourceDebugExtension Access

This string is made opaquely accessible at the three layers of the [Java Platform Debugger Architecture \(JPDA\)](#):

JVMDI	<code>GetSourceDebugExtension(jclass clazz, char **sourceDebugExtensionPtr)</code>
JDWP	<code>SourceDebugExtension</code> Command (12) in the <code>ReferenceType</code> (2) Command Set
JDI	<code>String sourceDebugExtension()</code> in the <code>ReferenceType</code> interface

8.2. SourceDebugExtension Class File Attribute

Java virtual machine class file attributes are described in [section 4.7](#) of the [The Java Virtual Machine Specification](#). The definition of the added attribute is in the context of The Java Virtual Machine Specification:

The `SourceDebugExtension` attribute is an optional attribute in the `attributes` table of the `ClassFile` structure. There can be no more than one `SourceDebugExtension` attribute in the `attributes` table of a given `ClassFile` structure.

The `SourceDebugExtension` attribute has the following format:

```

ËSourceDebugExtension_attribute {
Ë  u2 attribute_name_index;
Ë  u4 attribute_length;
Ë  u1 debug_extension[attribute_length];
Ë}

```

The items of the `SourceDebugExtension_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "SourceDebugExtension".

`attribute_length`

The value of the `attribute_length` item indicates the length of the attribute, excluding the initial six bytes. The value of the `attribute_length` item is thus the number of bytes in the `debug_extension[]` item.

`debug_extension[]`

The `debug_extension` array holds a string, which must be in UTF-8 format. There is no terminating zero byte.

The string in the `debug_extension` item will be interpreted as extended debugging information. The content of this string has no semantic effect on the Java Virtual Machine.

Chapter 9. Example

The example below shows how the process described above would apply to a tiny JSP program.

9.1. Input Source

The input consists of two JSP files, the first is `Hello.jsp`:

#1	<HTML>
#2	<HEAD>
#3	<TITLE>Hello Example</TITLE>
#4	</HEAD>
#5	<BODY>
#6	<%@ include file="greeting.jsp" %>
#7	</BODY>
#8	</HTML>

The second JSP file is the included file `greeting.jsp`:

#1	Hello There!<P>
#2	Goodbye on <%= new Date() %>

9.2. Language Processor

When a JSP compiler (the [language-processor](#)) compiles these files it will produce two outputs - a Java programming language source file and a [SMAP-file](#). The generated Java programming language source file is `HelloServlet.java`:

1	import jakarta.servlet.*;
2	import jakarta.servlet.http.*;
3	
4	public class HelloServlet extends HttpServlet {
5	public void doGet(HttpServletRequest request,
6	HttpServletResponse response)
7	throws ServletException, IOException {
8	response.setContentType("text/html");
9	PrintWriter out = response.getWriter();

10	ÊÊÊÊ// Hel lo.j sp: 1
11	ÊÊÊÊout.println("<HTML>");
12	ÊÊÊÊ// Hel lo.j sp: 2
13	ÊÊÊÊout.println("<HEAD>");
14	ÊÊÊÊ// Hel lo.j sp: 3
15	ÊÊÊÊout.println("<TITLE>Hel lo Example</TITLE>");
16	ÊÊÊÊ// Hel lo.j sp: 4
17	ÊÊÊÊout.println("</HEAD>");
18	ÊÊÊÊ// Hel lo.j sp: 5
19	ÊÊÊÊout.println("<BODY>");
20	ÊÊÊÊ// greet ing.j sp: 1
21	ÊÊÊÊout.println("Hel lo There! <P>");
22	ÊÊÊÊ// greet ing.j sp: 2
23	ÊÊÊÊout.println("Goodbye on " + new Date());
24	ÊÊÊÊ// Hel lo.j sp: 7
25	ÊÊÊÊout.println("</BODY>");
26	ÊÊÊÊ// Hel lo.j sp: 8
27	ÊÊÊÊout.println("</HTML>");
28	ÊÊ}
29	}

The generated [SMAP-file](#) is Hel loServl et.j ava. smap:

```

SMAP
Hel loServl et.j ava
JSP
*S JSP
*F
1 Hel lo.j sp
2 greet ing.j sp
*L
1#1, 5: 10, 2
1#2, 2: 20, 2
7#1, 2: 24, 2
*E

```

A couple things are interesting to note about this SMAP!Ñ!the user has chosen to make JSP the default

stratum (perhaps by a command line option) and even though there are ten lines of input source and 29 lines of generated source, only three `LineInfo` lines describe the transformation: the first and last are for the lines before and after the include (respectively) and the middle is for the included file `greeting.jsp`.

The three `LineInfo` lines describe these mappings:

```
1#1, 5: 10, 2
Ê   Hello.jsp:   line 1 ->   HelloServlet.java: lines 10, 11
Ê                               line 2 ->                               lines 12, 13
Ê                               line 3 ->                               lines 14, 15
Ê                               line 4 ->                               lines 16, 17
Ê                               line 5 ->                               lines 18, 19

1#2, 2: 20, 2
Ê   greeting.jsp: line 1 ->   HelloServlet.java: lines 20, 21
Ê                               line 2 ->                               lines 22, 23

7#1, 2: 24, 2
Ê   Hello.jsp:   line 7 ->   HelloServlet.java: lines 24, 25
Ê                               line 8 ->                               lines 26, 27
```

9.3. Post Processor

Next `HelloServlet.java` is compiled by a Java programming language compiler (for example `javac`) producing the class file `HelloServlet.class`. Then the `post-processor` is run. It takes `HelloServlet.class` and `HelloServlet.java.smap` as input. It creates a `SourceDebugExtension` attribute whose content is the SMAP in `HelloServlet.java.smap` and rewrites `HelloServlet.class` with this attribute.

9.4. Debugging

Now the program is run under the control of a debugger (which is a client of `JDI`). Let's say we are stepping through this code and the debugger has just received a `JDI StepEvent` for the line that is just about to output `<BODY>`. The debugger's code might look like this (the `StepEvent` is in the variable `stepEvent`):

```
Location location = stepEvent.location();
String sourceName = location.sourceName("Java");
int lineNumber = location.lineNumber("Java");
displaySource(sourceName, lineNumber);
```

where `displaySource` is a debugger routine that displays a source location. Because the `Java` stratum has been specified `sourceName` would be `HelloServlet.java`, the `lineNumber` would be `19` and the

displayed line would be:

```
out.println("<BODY>");
```

However, if `sourceName` and `lineNumber` were derived as follows:

```
String sourceName = location.sourceName("JSP");
int lineNumber = location.lineNumber("JSP");
```

Since the `JSP` stratum has been specified, `sourceName` would be `Hello.jsp`, the `lineNumber` would be `5` and the displayed line would be:

```
<BODY>
```

This occurs because the `SourceDebugExtension` attribute was stored when the VM read `HelloServlet.class` and it was retrieved with the `SourceDebugExtension JDWP` command which in turn caused the JVMDI function call `GetSourceDebugExtension`. The SMAP in the `SourceDebugExtension` was parsed which provided the above transformation of source location. Specifically, the line:

```
1#1, 5: 10, 2
```

is the basis of this transformation - which refers to `FileId #1`

```
1 Hello.jsp
```

and whence the `sourceName` information. Since the default stratum specified in the SMAP is `JSP`, the code:

```
String sourceName = location.sourceName();
int lineNumber = location.lineNumber();
```

would have the same effect. Since this is the form code would have taken before these extensions were introduced, existing debuggers can be utilized if they are run under the new implementation of `JDI`.