The Difficulties of Type Resolution Algorithms

James T. Perry, Victor L. Winter, Harvey P. Siy, Satish M. Srinivasan, Benjamin D. Farkas, and James A. McCoy

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.
The Difficulties of Type Resolution Algorithms

James T. Perry, Victor L. Winter, Harvey P. Siy, and Satish M. Srinivasan
University of Nebraska at Omaha
Department of Computer Science
Omaha, Nebraska 68182

Benjamin D. Farkas and James A. McCoy
Surety Electronics & Software Department
Sandia National Laboratories
P.O. Box 580
Albuquerque, NM 87185

Sandia Contract No. 646539

Abstract

One method of confirming a newly developed tool’s resolution algorithm is to compare its results with extensively used tools that also make use of resolution algorithms, such as Netbeans and Eclipse. Since these tools are used extensively, one might assume that they can correctly determine resolution within a given set of java files. However, we have found that correctly resolving all corner cases of java resolution is so difficult that even the popular tools have bugs.
CONTENTS

1. Tools and background ................................................................................................................................. 9

2. Tool Problems .............................................................................................................................................. 11
   1.1. Problem 1 – Classes in Methods ........................................................................................................ 11
   2.2. Problem 2 – Duplicate Classes ........................................................................................................... 13
   2.3. Problem 3 – Types and Variables .................................................................................................... 14
   2.4. Problem 4 – Static Importing ............................................................................................................ 15
   2.5. Problem 5 – Inner class & Inheritance ............................................................................................ 17

3. Tool Problem Summary ................................................................................................................................ 21
   3.1. Discussion of Netbeans ....................................................................................................................... 21
   3.2. Discussion of Eclipse .......................................................................................................................... 21
   3.3. Discussion of IntelliJ ...................................................................................................................... 21

4. Conclusion .................................................................................................................................................. 23

5. References .................................................................................................................................................. 25

Appendix A: Netbeans Reports .......................................................................................................................... 27
   A.1. Report and Response to Netbeans Bug #1 ....................................................................................... 27
   A.2. Report and Response to Netbeans Bug #2 ....................................................................................... 29

Appendix B – Eclipse Reports .......................................................................................................................... 31
   B.1. Response to Eclipse Bug #1 ................................................................................................................. 31
       http://bugs.eclipse.org/bugs/show_bug.cgi?id=318401 .......................................................................... 31
   B.2. Response to Eclipse Bug #2 ................................................................................................................. 32

Distribution ...................................................................................................................................................... 35

FIGURES

Figure 1. An example Library with 2 packages, each with a class Foo. ........................................................ 9
Figure 2. The class Foo is referring to a class that exists in a different package p2. ................................. 9
Figure 3. Example of how Netbeans shows the resolution of a class in the current scope ..................... 10
Figure 4. Netbeans’ resolution is incorrect at the arrow ............................................................................ 11
Figure 5. Eclipse has the correct resolution ............................................................................................... 12
Figure 6. Netbeans prior to refactoring class B .......................................................................................... 12
Figure 7. Netbeans after refactoring, incorrectly refactors static class B ..................................................... 12
Figure 8. A file containing 2 top level classes .............................................................................................. 13
Figure 9. A duplicate class error not caught by Netbeans ........................................................................ 13
Figure 10. Eclipse catches the duplicate class error .................................................................................. 14
Figure 11. In Eclipse, a class and field with the same name B .................................................................... 14
Figure 12. Eclipse attempts to access the non-visible field ........................................................................ 14
Figure 13. Netbeans correctly finds the class ............................................................................................ 15
Figure 14. A static class and static field with the same name B ................................................................. 16
Figure 15. Another inner class B ................................................................................................................ 16
Figure 16. Eclipse finds the class B from the wrong package .................................................................... 16
Figure 17. Netbeans finds the class $B$ from the correct package. ................................................ 16
Figure 18. IntelliJ also finds the correct class $B$. ................................................................. 17
Figure 19. Eclipse prior to refactoring $B$ to $B123$. ............................................................... 17
Figure 20. After refactoring, Eclipse did not refactor the static import as it should have. .... 17
Figure 21. IntelliJ incorrectly finds private class $C2$. ............................................................. 18
Figure 22. Netbeans correctly finds $extra.C2$. ........................................................................ 18
Figure 23. IntelliJ prior to renaming the private class $C2$. ..................................................... 19
Figure 24. IntelliJ after refactoring: changes appear at the arrows. ......................................... 19

TABLES

Table 1. Tools and their corresponding resolution problems. .................................................. 21
## NOMENCLATURE

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
<tr>
<td>javac</td>
<td>The Java Compiler</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
</tbody>
</table>
1. TOOLS AND BACKGROUND

There currently exists a wide array of tools that have some capabilities of analyzing Java, (e.g. Netbeans). However, these tools are not equipped for all projects, particularly java library migration [2]. For this reason we have developed our own tools capable of analyzing and transforming the Java libraries.

To verify the accuracy our tool analysis, we have compared it with the most extensively used tools, including: Eclipse SDK, Netbeans, and IntelliJ IDEA. The main technique employed was to compare resolution algorithms. The resolution algorithm is used to determine if some field (or class, etc) is currently visible and also whether another field (or class, etc) is shadowing, hiding, or obscuring it. Resolution is a key aspect for Java analysis.

For example, suppose that there exists a class ImportantClass that makes use of a class Foo. If our algorithm earlier removed a class Foo in the same package, then a naive algorithm may automatically assume that ImportantClass must also be removed. However, it is possible that the deleted class Foo is not the same class used by ImportantClass as seen in Figures 1 & 2.

![Figure 1. An example Library with 2 packages, each with a class Foo.](image)

![Figure 2. The class Foo is referring to a class that exists in a different package p2.](image)

This example is very simple, if p1.Foo is deleted, it will not affect ImportantClass because it is actually referring to p2.Foo which was not deleted. But if the resolution algorithm is wrong, the member variable myFoo could be incorrectly deleted, and the resulting library may be much smaller than what can be supported, or could have invalid libraries. It is also important to note the resolution algorithm is much more complicated than simply looking for import
statements, this is where tools like Netbeans and Eclipse are helpful: they provide the user with the actual path to the class in use: as shown in figure 3.

Figure 3. Example of how Netbeans shows the resolution of a class in the current scope.

Using the resolution information provided by the tools, we developed multiple tests to compare our algorithm with those of the selected IDEs. To do this we start by coding example classes in a selected IDE. We then encode the fields with the path to the class it is referencing, and also encode if the class accessible (visible) in the current scope. Once the encoding is finished, we run our resolution algorithm and automatically compare the results with the expected answers (from the encoding), checking for both false-positives and false-negatives.

This method of comparison has proven very useful in confirming that our resolution algorithm is accurate. However, as we have developed increasingly more robust and complicated tests, we have found that making a complete resolution algorithm is very difficult. In fact, during the process of making these tests, we have found that Netbeans and other IDE’s have resolution problems of their own.
2. TOOL PROBLEMS

As mentioned, many IDEs have been found to contain errors in their resolution algorithm. The fact that such well-used and popular tools have errors with their resolution algorithm exemplifies the difficulty of this problem. This section will show the complicated cases we have developed and the tools that pass/fail these tests. Expected results were generated by Java (from the command line) often adding simple print statements when necessary.

1.1. Problem 1 – Classes in Methods

The first problem we developed examines the use of classes inside methods. Java defines that these classes are only accessible within the encompassing method, and according to Java’s specifications: a use of some class should first attempt to resolve to a corresponding class within the same scope. However, in the example shown in Figure 4, Netbeans has the incorrect result for the resolution of B in the extends statement shown by the arrow. The other IDEs however, conforms to Java’s specifications and returns the correct result as shown in Figure 5. This problem was also checked by running Java from the command line. Netbeans printed “0”, whereas the other IDEs and java from the command line printed “1”.

```
package pl;
public class A {
    public static void main(String[] args) {
        foo();
    }
    public static void foo(){
        class B {
            int x = 1;
        }
        class C extends B{
            System.out.println(new C().x);
        }
        static class B {
            int x = 0;
        }
    }
}
```

Figure 4. Netbeans’ resolution is incorrect at the arrow.
One thing especially bad about this bug is that it compiles without any warning and running it from Netbeans gives the wrong answer. However if you have Netbeans generate the jar for the corresponding code and then run the jar from the command line, then it gives the correct answer. Thus after compiling without any errors or warnings, the resultant executable will give a different answer than what was encountered in Netbeans.

This resolution problem has further consequences. If we try to refactor $B$ at the location
shown in figure 6 then the wrong class gets refactored as shown in figure 7. The output from running this class in Netbeans prior to refactoring (which was wrong) now becomes *the* actual output. Thus this simple act of refactoring has changed the meaning of the code. This bug has been reported to Netbeans’ bug report website, and responses have been given (see Appendix A.1).

### 2.2. Problem 2 – Duplicate Classes

Within Java, you are not allowed to have 2 files with the same name within a package. This includes package-private classes within other Java files. Consider Figures 8 and 9:

![Figure 8. A file containing 2 top level classes.](image)

As seen here, Netbeans won’t catch a duplicate Class hidden inside a different `.java` file. Netbeans allows users to run the code from the IDE, and whichever file was most recently saved is the one which Netbeans will use during execution (printing “5” or “10” depending on which file was saved last). Other IDEs, however, correctly catch this error, as seen in figure 10.

![Figure 9. A duplicate class error not caught by Netbeans.](image)
Fortunately, Netbeans will catch the original error prior to building the executable. However, strange things happen in Netbeans when attempting to rename `DuplicateClass` using refactoring. Regardless of where the refactoring is done (in the public class `DuplicateClass`, the package-private class `DuplicateClass` or even the at the print statement `DuplicateClass.x`) the results are sporadic. Sometimes the file name (`DuplicateClass.java`) and the package-private class would change, but it wouldn’t change the class inside the file `DuplicateClass.java`. Other times only the package-private class would change. Sometimes everything except the package-private class would change. As a result, sometimes refactoring would result in a compiler error and other times it would rename the public class even if the user directly tried to refactor the package-private class. This problem has also been reported to Netbeans’ Bug report site (see Appendix A.2).

### 2.3. Problem 3 – Types and Variables

In figures 11 and 12 we find a problem encountered in both Eclipse and IntelliJ IDEA.

![Figure 11. In Eclipse, a class and field with the same name B.](image)

This problem is associated with both shadowing and obscuring in the resolution algorithm. According to the Java specification, “fields obscure type”. That is: given the choice
between a class and a field with the same name the field should be used instead of the class. This is what Eclipse and IntelliJ appears to be doing. However, in this case the field B is not visible in package p1 since its access is package-private. Therefore, B in the print statement can only be referring to the class since the field is not available to obscure it. Unfortunately, these IDEs continue using the field (even though it is not visible), and results in the error shown in figure 12.

As seen in figure 13, Netbeans correctly identifies B as the class. This code compiles and runs in Netbeans and using java from the command line.

![Figure 13. Netbeans correctly finds the class.](image)

Attempting to refactor B in the print statement using IntelliJ, only changes the field and leaves the print statement alone! Thus, refactoring fixed the problem in IntelliJ. However, since Eclipse won’t allow refactoring to code errors, this problem has no further consequences (though one should be able to refactor the code since it should be valid). This problem has so far only been reported to Eclipse, further discussion on the problem can be seen in Appendix B.1.

### 2.4. Problem 4 – Static Importing

In section 7.5.2 of the Java Specification it states, “A type-import-on-demand declaration never causes any other declaration to be shadowed.”[1]

In relation to single-type import statements, section 7.5.1 of the Java Specification it states:

A single-type-import declaration \(d\) in a compilation unit \(c\) of package \(p\) that imports a type named \(n\) shadows the declarations of:

- any top level type named \(n\) declared in another compilation unit of \(p\).
- any type named \(n\) imported by a type-import-on-demand declaration in \(c\).
- any type named \(n\) imported by a static-import-on-demand declaration in \(c\).

Therefore, if an import-on-demand contains a class that is also in a single-type-import, the single-type-import would take precedence.

Consider figures 14, 15, and 16; the import statements \(p2.Bar.B\) and \(p3.Foo.*\) in Figure 16 implies that any use of \(B\) should come from \(p2.Bar\) and not from \(p3.Foo\). However, Eclipse does the opposite!
Generally, Eclipse works correctly at this level of resolution (import-on-demand and single-type-import), but this case is more complicated – the first import statement is an Import-Static rather than a typical import statement. Import-Static allows users to import static members of a class, which can include: static fields, static methods, or static classes. In this way “import static p2.Bar.B” is importing both the class and the field B. Also since p2.Bar has a field and a class B, as previously mentioned, the obscuring rule dictates that fields take precedence over a class with the same name (when the use is ambiguous). However, only classes can have a “.class” to access. Thus in this context B can only be referring to a class.
Figure 18. IntelliJ also finds the correct class B.

We have surmised that Eclipse first obscures the class B with the field B inside p2.Bar. Afterwards it realizes that the context requires a class – thus at this point the only class B it can find is within p3.Foo.* since the class within p2.Bar was obscured. Netbeans and IntelliJ (as seen in Figures 17 and 18) correctly determine first that the context requires a class and therefore choose p2.Bar.B since it was imported using a single-type-import.

Figure 19. Eclipse prior to refactoring B to B123.

Figure 20. After refactoring, Eclipse did not refactor the static import as it should have.

Refactoring this problem in Eclipse also caused an issue, as shown by figures 19 and 20. Refactoring should change the single-type import B, yet it actually refractors the type B from the import-on-demand.

This problem was reported to Eclipse, further discussion can be seen in Appendix B.2

2.5. Problem 5 – Inner class & Inheretence

In section 8.4.8 of the Java specification it states:

A class C inherits from its direct superclass and direct superinterfaces all non-private methods (whether abstract or not)
of the superclass and superinterfaces that are public, protected or declared with default access in the same package as C and are neither overridden nor hidden by a declaration in the class. [1]

However the Java specification also states:

A private class member or constructor is accessible only within the body of the top level class that encloses the declaration of the member or constructor. [1]

Together these rules say that a private inner class could be accessed by another class under the right circumstances. Consider Figure 21, IntelliJ believes that B2 has access to both private classes C2 and D1 that it inherited from B1. However, in reality there is a distinction between the visibility of C2 and D1. In attempting to resolve D1, the algorithm first finds the parent class C1. Since B1 and B2 are in the same package, B2 inherits the class C1. After the algorithm has found C1, it converts it to the full canonical name p.A.B1.C1. Since this class and B2 are in the same top level class, D1 is also visible. On the other hand, it should not find the private class C2 since private classes are not inherited – this is where IntelliJ is incorrect with its resolution algorithm.

![Figure 21. IntelliJ incorrectly finds private class C2.](image)

Figure 21. IntelliJ incorrectly finds private class C2.

![Figure 22. Netbeans correctly finds extra.C2.](image)

Figure 22. Netbeans correctly finds extra.C2.

As seen in Figure 22, Netbeans (as well as Eclipse and java from the command line) says that C2 should be the class from package extra. In fact, at runtime even IntelliJ uses the correct
algorithm (confirmed with print statements). However, since IntelliJ has the incorrect resolution prior to runtime, refactoring can change the meaning of the code.

Consider Figure 23 and 24, if we attempt to rename the private class \textit{B1.C2} then IntelliJ: does correctly rename the private class \textit{B1.C2}, but it incorrectly renames the type of the field \textit{B2.x1} as well as incorrectly removes what it considers an “unused” import. Conclusively, IntelliJ changed the meaning of the code, provided no errors prior to running, at runtime it failed to run, and there is no longer any evidence that \textit{C2} should have be found in package extra.
3. TOOL PROBLEM SUMMARY

We found it interesting that all of these tools had some problems with their resolutions. We will now briefly discuss each IDE and its problems (see also Table 1).

<table>
<thead>
<tr>
<th>Problems</th>
<th>Problem 1</th>
<th>Problem 2</th>
<th>Problem 3</th>
<th>Problem 4</th>
<th>Problem 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netbeans</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eclipse</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>IntelliJ (Free Community version)</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>IntelliJ (Full Version)</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

3.1. Discussion of Netbeans

Our testing on Netbeans used versions 6.8 and 6.9. It is important to note that Netbeans is supposed to use the same compiler as Java, making the problems more compelling. As discussed in Appendix A.1, some of the problems were thought to be errors in JDK1.7 javac that propagated to Netbeans (though Problem 2 appears to be a bug strictly in Netbeans). The current released JDK at the writing of this paper is ‘java 1.6update21’. Java also has the ‘JDK 7 project’ which is still considered in the ‘build’ stages. Netbeans, since it is directly associated with Oracle & Java, is using a preliminary version of JDK 1.7 javac. We have found that both Netbeans 6.8 and the latest version 6.9 have had Problem 1 & 2, yet when we downloaded the current binaries for JDK7 (build 99), neither problem occurred. After reporting this problem, Netbeans has mentioned that they now plan on bringing in the latest javac into the future release of Netbeans 6.10.

3.2. Discussion of Eclipse

Eclipse, unlike Netbeans has its own compiler separate from the JDK. The version of Eclipse we tested was the “Helios Service Release 1, Build id: 20100917-0705.” All of the bugs we submitted are reportedly in the process of being patched for future releases.

3.3. Discussion of IntelliJ

The most recent tool we found and tested was IntelliJ IDEA which claims to be “the most Intelligent java IDE” [3]. There is a free “community version” of the tool as well as a “full version” for purchase. We discovered that the same resolution is used in both versions of IntelliJ.
4. CONCLUSION

As we have shown, type resolution in Java can be extremely complicated. Popular tools such as Netbeans\(^1\), Eclipse\(^2\), and even IntelliJ\(^3\) do not have a completely correct resolution algorithm. In summary, the resolution problems in allowed:

1. Invalid code to be run
2. Valid code to be labeled invalid
3. Valid code to return the wrong result
4. Refactoring could change the meaning of the code

We would like to mention that in some cases the Java specification did not seem to be completely consistent which may have lead to some of the IDE confusion on a correct algorithm. For example, Accessibility specifications state that package-private members are accessible to other classes in the same package and private members are visible to other classes in the same top-level class. In general, extends allows inheritance only to public and protected members; however if both classes are in the same package, then package-private members are also inherited. Wouldn't similar rational suggest that if both classes are in the same top-level class, they would inherit private members? Yet the Java specification does not permit such inheritance. This seems counter intuitive.

---

\(^1\) Netbeans can be downloaded freely at: [www.netbeans.org](http://www.netbeans.org)
\(^2\) Eclipse can be downloaded freely at: [http://www.eclipse.org/](http://www.eclipse.org/)
\(^3\) Both editions of IntelliJ can be downloaded at: [http://www.jetbrains.com/idea/](http://www.jetbrains.com/idea/)
5. REFERENCES


APPENDIX A: NETBEANS REPORTS

A listing of Netbeans bug reports and responses from the manufacturers (as of 7/13/10). (“00jt” is the bug reporter from our group)

A.1. Report and Response to Netbeans Bug #1

http://netbeans.org/bugzilla/show_bug.cgi?id=187452

**Description**

From 00jt@netbeans.org 2010-06-10 20:01:35

When I have a class "x" in some method, and another inner class (Outside of the method) with the same name. Then any other class "y" that extends "x" inside the same method as "x" will incorrectly use the other class (outside of the method).

See link to forum.

----- Comment #1 From Jan Lahoda 2010-06-13 18:36:26 -----

Seems like a bug in JDK1.7 javac, which got pulled to NB. I tried to find out the cause, but did not succeed so far. Seems that the buggy changed occurred before the code was converted to mercurial.

There is a quite simple workaround: go to Project Properties/Compile tab (for J2SE Project) and disable Compile on Save. After that the javac from your JDK will be used to compile the project and as long as that javac does not have this bug, the problem should go away.

----- Comment #2 From 00jt@netbeans.org 2010-07-06 16:04:59 -----

(In reply to comment #1)
> Seems like a bug in JDK1.7 javac, which got pulled to NB. I tried to find out the cause, but did not succeed so far. Seems that the buggy changed occurred before the code was converted to mercurial.
> There is a quite simple workaround: go to Project Properties/Compile tab (for J2SE Project) and disable Compile on Save. After that the javac from your JDK will be used to compile the project and as long as that javac does not have this bug, the problem should go away.

So, is this bug going to be fixed for JDK1.7? Have you had any luck finding the cause?
------ Comment #3 From Jan Lahoda 2010-07-13 14:48:33 (-) [reply] ------

I took a deeper look. I am almost sure that this was caused by a fix for bug 5060485:

I will contact the javac maintainers to see what they think.

------ Comment #4 From 00j@netbeans.org 2010-07-13 18:11:57 (-) [reply] ------

(In reply to comment #3)

Sounds good, I have recently downloaded JDK1.7 (build 99) and the bug didn't occur. Perhaps Netbeans has an earlier build of JDK1.7?

------ Comment #5 From Jan Lahoda 2010-08-26 11:27:35 ------

I do not think this was fixed in JDK7 build 99 - but should be fixed by:
http://hg.openjdk.java.net/jdk7/tl/langtools/rev/a75770c0d7f6

I suppose we will take a javac with this fix for 6.10.

(No further responses)
A.2. Report and Response to Netbeans Bug #2
http://netbeans.org/bugzilla/show_bug.cgi?id=187343

Description From 00jt@netbeans.org 2010-06-08 17:15:54

I can have 2 classes with the same name inside a package as long as one class or both classes are an outer class within a separate class.. for example:

File 1:  test.java
----------
package pl;

public class test{

}
class otherClass{
----------

File 2: otherClass.java
----------
package pl;

public class otherClass{
----------

This should be an error ” 'otherClass' already defined in package pl”
But it does not give that error (unless i cleans/build)

Unfortunately, i am able to ”run” the code in Netbeans without any errors/warnings at all, and any usage of ”otherClass” in some other class in package ”pl” simply uses the class that was most recently used.

-------- Comment #1 From Dusan Balek 2010-07-20 11:42:30 --------

Fixed in jet-main.

http://hg.netbeans.org/jet-main/rev/8af0f812fa21
changepset doa03aa040b9 in main/nb-javac
details: http://hg.netbeans.org/main/nb-javac?cmd=changepset;node=doa03aa040b9

-------- Comment #2 From Quality Engineering 2010-07-21 03:15:55 --------

Integrated into 'main-golden', will be available in build *201007210001* on http://bits.netbeans.org/dev/nightly/ (upload may still be in progress)
Changeset: http://hg.netbeans.org/main/rev/8af0f812fa21
User: Dusan Balek <dbalek@netbeans.org>
Log: Issue #187343: uncought duplicate class error - fixed.
APPENDIX B – ECLIPSE REPORTS

A listing of Eclipse bug reports and responses from the manufacturers (as of 7/13/10). ("00jt" is the bug reporter from our group)

B.1. Response to Eclipse Bug #1
http://bugs.eclipse.org/bugs/show_bug.cgi?id=318401

<table>
<thead>
<tr>
<th>00jt</th>
<th>2010-06-24 12:49:26 EDT</th>
<th>Description</th>
<th>reply [-]</th>
</tr>
</thead>
</table>

Build Identifier: M20100211-1343

If a class "A" has a public static class "B" as well as a private field "B", and then a class in another package tries to use "A.B" - Eclipse tries to access the private field.

Small example code:

```
package p1;

public class Main {
    public static void main(String[] args) {
        System.out.println("value = "+ p2.A.B.length);
    }
}

package p2;

public class A {
    public final static class B {
        public final static String length = "very long";
    }
    private int [] B = new int[5];
}
```

This code works from command line.. as well as another IDE, but Eclipse doesn't work.

I currently dont know of a work-around.

<table>
<thead>
<tr>
<th>Olivier Thomann</th>
<th>2010-06-24 13:51:03 EDT</th>
<th>Comment 1</th>
<th>reply [-]</th>
</tr>
</thead>
</table>

javac 1.6 compiles this code fine and the Eclipse compiler is failing. Srikanth, please investigate.

<table>
<thead>
<tr>
<th>srikanth</th>
<th>2010-06-28 23:53:22 EDT</th>
<th>Comment 6</th>
<th>reply [-]</th>
</tr>
</thead>
</table>

Created an attachment (id=172983) [details]
Patch under consideration.

<table>
<thead>
<tr>
<th>srikanth</th>
<th>2010-06-29 01:20:36 EDT</th>
<th>Comment 8</th>
<th>reply [-]</th>
</tr>
</thead>
</table>

(In reply to comment #6)
> Created an attachment (id=172983) [details] [details]
> Patch under consideration.

Passes all tests. Ayush, please review, Thanks.
B.2. Response to Eclipse Bug #2

http://bugs.eclipse.org/bugs/show_bug.cgi?id=318401 &
https://bugs.eclipse.org/bugs/show_bug.cgi?id=318401

(I in reply to comment #8)
> (In reply to comment #6)
> > Created an attachment (id=172983) [details] [details] [details]
> > Patch under consideration.
> > Passes all tests. Ayush, please review, Thanks.

Patch looks good.

Released in HEAD for 3.7 M1.

---

B.2. Response to Eclipse Bug #2

http://bugs.eclipse.org/bugs/show_bug.cgi?id=318401 &
https://bugs.eclipse.org/bugs/show_bug.cgi?id=318401

(I in reply to comment #8)
> (In reply to comment #6)
> > Created an attachment (id=172983) [details] [details] [details]
> > Patch under consideration.
> > Passes all tests. Ayush, please review, Thanks.

I got the same bug, abused differently, to actually compile in Eclipse and give a different answer, I thought you might like to see it:

```
package p2;

import static p1.Bar.B;
import p3.Foo.*;

public class OuterTest {
    public static void main(String [] args)
    {
        new OuterTest().beginTest();
    }
    public void beginTest()
    { System.out.print("1 + 1 = ");
      if(alwaysTrue()){
        System.out.println("2");
      } else{
        System.out.println("3");
      }
    }
    public boolean alwaysTrue(){ // Returns FALSE in Eclipse
        String myB = B.class.getCanonicalName();
        String realB = p1.Bar.B.class.getCanonicalName();
        return myB.equals(realB);
    }
}
```
package p1;
public class Bar {
    public static class B {}
    final public static String B = new String("random");
}

package p3;

public class Foo {
    public class B {}
}

Eclipse will print out "1 + 1 = 3"

In the above bug, it appears that in "Bar" the field "B" is obscuring the class with the same name, and then it looks like the import-on-demand from "Foo.*" is shadowing the single-import "Bar.B". Perhaps the JSP is a bit to vague on these corner cases.

(In reply to comment #7)

> Ayush, please triage this case and see if there is a bug here.
> Open a new defect as needed and assign to yourself for
> follow up -- Thanks.

I have raised bug 318401 to track the imports shadowing issue and copied the relevant test cases and comments from here to there. Folks, over to bug 318401 for any discussions related to imports, please -- Thanks!
DISTRIBUTION

4 James T. Perry  
5811 Villa Drive, NE  
Albuquerque, NM 87111

2 Victor L. Winter  
PKI 174 C  
1110 South 67th Street  
Omaha, NE 68182

1 Harvey P. Siy  
PKI 281 B  
1110 South 67th Street  
Omaha, NE 68182

1 Satish M. Srinivasan  
PKI 355  
1110 South 67th Street  
Omaha, NE 68182

1 MS0340 Greg Wickstrom 2141
1 MS0451 Benjamin D. Farkas 2144
1 MS0451 James A. McCoy 2144
1 MS0451 Dominic Montoya 2144
1 MS0451 Eric Disch 2144

1 MS0899 Technical Library 9536 (electronic copy)