Base Language - Bug #4978

Static method call in base class

10/22/2020 06:14 AM - Marian Edu

Status:	Review	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Greg Shah	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD		

Description

History

#1 - 10/22/2020 07:31 AM - Greg Shah

- Description updated

- Assignee set to Constantin Asofiei

From Marian:

> The 4GL seems to allow calling a static method defined in the base class as if it is part of the class itself and this is perfectly valid. However when doing that the code conversion acts strange, there is no error but the line with the offended code is simply skipped and missing in generated code (if no-error is used it just put an empty silent block).
>
> A test case here - oo/quirks/static_method/test_static_method.p

#2 - 10/23/2020 09:41 AM - Constantin Asofiei

Marian, this patch solves the issue.

Greg: if this looks OK, I'll commit to 3821c.

```
### Eclipse Workspace Patch 1.0
#P p2j
Index: src/com/goldencode/p2j/uast/SymbolResolver.java
===================================================================
--- src/com/goldencode/p2j/uast/SymbolResolver.java    (revision 2576)
+++ src/com/goldencode/p2j/uast/SymbolResolver.java    (working copy)
@@ -436,6 +436,9 @@
 ** 093 IAS 20200624          Added checks for null
 ** 094 CA  20200804          Emit bulk setter and getter for an extent property.
 *     CA  20200930          Use an exemplar instance to pre-populate the dictionaries at runtime conversion.
*     CA  20201023          A lookup for a class qualified, static method, must include any super-class, too;
*                           this is done by assuming we are in 'internal' mode, to allow lookup up the
+*                           hierarchy.
 */
```

```
/*
@ -4355,7 +4358,7 @@
   {
       ClassDefinition current  = getCurrentClassDef();
       ClassDefinition cls      = (cname  null) ? current : lookupClass(cname);
-      boolean         internal = (cls  current);
+      boolean         internal = (cls == current) || cname != null;
       int             access   = calcAccessMode(cname);

if (cls != null)
@ -5063,7 +5066,7 @@
{
       ClassDefinition current  = getCurrentClassDef();
       ClassDefinition cls      = (cname  null) ? current : lookupClass(cname);
-      boolean         internal = (cls  current);
+      boolean         internal = (cls == current) || cname != null;
       int             access   = calcAccessMode(cname);

return (cls == null) ? -1 : cls.guessMethodType(method, access, isStatic, internal);
```

#3 - 10/23/2020 10:16 AM - Greg Shah

I don't think this is correct. With this change, internal will always be true.

#4 - 10/23/2020 10:23 AM - Constantin Asofiei

I don't recall why we needed to distinguish between internal and non-static method lookup. Hierarchy lookup is performed only if parents != null && (!isStatic || internal). What was the point of (!isStatic || internal), better said what was the case where we must not lookup in the hierarchy?

You are right, internal will always be true with my change. So maybe is better to just remove the (!isStatic || internal) condition, in ClassDefinition like 2937, 3071, 3465?

#5 - 10/23/2020 10:50 AM - Greg Shah

I'm going to copy in the testcase code so what we are discussing is clear to future readers.

Base.cls:

```
using Progress.Lang.*.

block-level on error undo, throw.

class oo.quirks.static_method.Base:

    method public static logical getFlag ():
        return true.
    end.

end class.
```

Extension.cls:

```
using Progress.Lang.*.

block-level on error undo, throw.

class oo.quirks.static_method.Extension inherits oo.quirks.static_method.Base:

end class.
```

test_static_method.p:

```
block-level on error undo, throw.

oo.quirks.static_method.Base:getFlag().
oo.quirks.static_method.Base:getFlag() no-error.

// this doesn't get converted
oo.quirks.static_method.Extension:getFlag().
oo.quirks.static_method.Extension:getFlag() no-error.
```

This is certainly an interesting quirk. It is not a good practice in my opinion, but we clearly must handle it. On a positive note, once we have detected the case, it is trivial to handle because we can just emit the base class name instead of the child class name.

The (!isStatic || internal) was meant to exclude this case because we did not know that isStatic == true would need to lookup the hierarchy. We can't get rid of the internal check, that is also quite important. For example, protected methods in a class would be visible to all callers. On the other hand, we can extend the idea of internal to cases where current is a child class of cls. I think that is the correct solution. We will also need to detect this special case and rewrite the class name to be the parent class so that it emits correctly.

#6 - 02/19/2021 01:28 PM - Constantin Asofiei

- Status changed from New to WIP

The same behavior where the static member from a super-class can be accessed from a sub-class is allowed for variables/properties. In variable's case, ClassDefinition.lookupHierarchy has no protection against internal/non-internal modes. Although variables are not technically overridden, they can be hidden in a sub-class (if one is defined with the same name).

During lookup, if we consider if this is a qualified or non-qualified access (be it method or other member), then:

- in non-qualified mode, this is possible only when the call is from a within a class, parent lookup is done for PROTECTED/PUBLIC members.
- in qualified mode (either via a class name or a variable name which sets the target class), then the call can be:
  - from within a class C, and from parent can be accessed:
    - PROTECTED members, only if the qualified type Q is the same as or a super-class of type C
    - PUBLIC members, otherwise
  - from outside a class, from parent only PUBLIC members can be accessed.

we can extend the idea of internal to cases where current is a child class of cls

I'm not sure that is enough. 'internal' will be false if we are executing calls from outside any class (like in the example, the call is made from within an external program).

The calcAccessMode already gives us a limit of the member visibility which can be found:

- if the call is in a class C, then:
  - if the call is qualified for Q and C is Q, access is PRIVATE and above
  - if the call is qualified for Q and Q is a super-class of C, access is PROTECTED and above
  - if the call is not qualified, access is PRIVATE and above
- if the call is not from a class, then access must be PUBLIC

And when doing the parent lookup, the access is already changed to PROTECTED and above.

I'm trying to find a reason for excluding a parent lookup, but I can find any, as all these cases must allow parent lookup:

- in non-qualified mode - parents of the class C where the call is made (parent access PROTECTED or PUBLIC)
- in qualified mode, when the qualified Q is the same as or a super-class of C (where the call is performed) - lookup only in parents of Q (parent access PROTECTED or PUBLIC).
- in qualified mode, when qualified Q is not the same or a super-class of C (parent access PUBLIC)
- in qualified mode, when the call is from outside a class (parent access PUBLIC)

In all cases, what limits is the access mode used for lookup. I know it looks like I'm repeating the same information above, but I don't see something wrong in my reasoning. Am I missing something?

#7 - 02/19/2021 03:57 PM - Greg Shah

We need to be careful using qualified/unqualified. Most of the time when we say qualified, we mean the "package" portion of a class name. I suspect here you mean that there is a <class_or_instance_reference>:<member_or_method> (explicit class) as opposed to just a <member_or_method> (implicit class). From our perspective, if something is just a <member_or_method> then this means that the implicit class upon which we are doing the lookup will be the same as the current class definition (if any). In the case where we are in a class definition, then this will mean that internal true for these implicit references. But it is no different from the explicit case, where the explicit class used (<class_or_instance_reference>) is the same as the current class definition. In that case internal true as well.

> I don't recall why we needed to distinguish between internal and non-static method lookup. Hierarchy lookup is performed only if parents != null && (!isStatic || internal). What was the point of (!isStatic || internal), better said what was the case where we must not lookup in the hierarchy?

The core idea was to allow us to know if the call originated from code in the current class hierarchy or not. I don't think we can otherwise calculate that if it is not passed in. We use it for 2 purposes.

1. We have an optional 2nd lookup **in the same initial ClassDefinition instance** for !isStatic && internal cases. From ClassDefinition.lookupMethodWorker():

```
         mdat = exactMethodLookup(name, sig, access, isStatic, internal);

         if (mdat == null && !isStatic && internal)
         {
            // can't find an instance member, check for static
            mdat = exactMethodLookup(name, sig, access, true, internal);
         }
```

Here we try a static true lookup if the first lookup was non-static and we are an internal lookup. This is used in the implicit method case. I don't know if it has any other usage, but I can't think of one at the moment.

We also pass the flag downstream to exactMethodLookup() where we use it for this:

```
         // ignore if the static/instance does not match
         if (mdat != null && !((!isStatic && (!mdat.isStatic || internal)) ||
                               (isStatic && mdat.isStatic)))
         {
            // clear our result
            mdat = null;
         }
```

I don't recall when exactly this code comes into play, but I do remember it was needed to exclude some cases. I'm not sure if it is still needed.

2. For whatever reason, we implemented parent hierarchy limitations on method lookups. We only do a parent lookup for instance methods or for static methods where the referring code was inside the class hierarchy (internal true). Your testcase above shows this is not correct. I don't know if there is some case I'm not considering here. From exactMethodLookup():

```
         // only look up the parent hierarchy if we haven't got a match yet; static lookups only
         // work for internal usage, non-static lookups always work
         if (mdat == null && parents != null && (!isStatic || internal))
         {
            // change access mode since we aren't allowed to see private stuff in parent
            int _access = (access == KW_PRIVATE) ? KW_PROTECTD : access;

            for (ClassDefinition parent : parents)
            {
               // recurse up inheritence hierarchy
               mdat = parent.exactMethodLookup(name, sig, _access, isStatic, internal);

               if (mdat != null)
                  break;
            }
         }
```

Since the issue here is purpose 2 (missing parent hierarchy lookup), I think the solution is to change the code in exactMethodLookup() and

fuzzyMethodLookup() to open up the parent hierarchy searches for methods. In other words, the solution is to remove the && (!isStatic || internal) for these parent hierarchy lookup cases. I don't think we can avoid purpose 1.

#8 - 02/19/2021 04:15 PM - Constantin Asofiei

Lets assume you have a protected instance method Base.foo and a class Extension which extends Base.

1. in a method in class Bar, you have a code like o:foo(), where o is Extension or Bar - in this case, the method can not be called (compile error).
2. in a method in class Extension, you have a code like o:foo(), where o is Base - the method can not be called (compile error).
3. in a method in class Extension, you have a code like o:foo(), where o is Extension - the method **can** be called.

If you make the foo() method static, and call it (from Extension) as Base:foo() or Extension:foo(), both will work. I think that was the reason for **internal**, to handle the case of 'qualified' instance method calls.

I need to think a little more about this. Maybe the test should be (isStatic || (!isStatic && internal)).

#9 - 03/07/2021 10:07 AM - Greg Shah

As part of the changes for #4350, I'm switching the exactMethodLookup() to use candidates() for making a list of all methods throughout the inheritance hierarchy as a single list. This was previously already done for fuzzyMethodLookup() and is needed to get the right behavior for exact lookups (which are not always strictly "exact"). So if you need to make the change to the && (!isStatic || internal) logic, then you'll need to modify the logic in candidates() because that is the only place it will exist. In other words, don't change the exactMethodLookup() because I'm rewriting it now.

> in a method in class Extension, you have a code like o:foo(), where o is Base - the method can not be called (compile error).

This is unexpected for me. I don't understand why this would be a limitation.

#10 - 03/07/2021 10:34 AM - Constantin Asofiei

- File modifiers.zip added

See attached for the tests I used.

o:foo() produces a compile error when called from Extension because o is Base and foo() is protected.

#11 - 03/08/2021 12:47 PM - Greg Shah

> o:foo() produces a compile error when called from Extension because o is Base and foo() is protected.

I don't doubt your results. :) I just find it surprising that the 4GL does not allow child classes to access protected methods in the parent instance. Being able to access a parent class' protected methods is a standard feature of the protected concept and it certainly is implemented that way in Java.

Then again, the 4GL lacks the concept of a package as well, so there is a range of things missing in that implementation. Still, it seems wierd to me.

#12 - 03/08/2021 12:57 PM - Constantin Asofiei

Greg Shah wrote:

> I just find it surprising that the 4GL does not allow child classes to access protected methods in the parent instance. Being able to access a parent class' protected methods is a standard feature of the protected concept and it certainly is implemented that way in Java.

The same behavior is in Java, too. You just have to put the base and extension classes in different packages. This code will not compile when called from with Extension:

```
package com.goldencode.testcases.oo;

import com.goldencode.testcases.JBase;

public class JExtension
extends JBase
{
   public void m2()
   {
      JExtension j = new JExtension();
      j.m1();  // this works fine

      JBase b = new JBase();
      b.m1(); // compile error here
   }
}
```

and JBase in a different package:

```
package com.goldencode.testcases;

public class JBase
{
   protected void m1()
   {

   }
}
```

#13 - 03/08/2021 01:31 PM - Greg Shah

Interesting. I had not run across that restriction before. I guess the package visibility rules limit the cases where it matters. It is too bad the 4GL doesn't have packages.

#14 - 03/08/2021 01:35 PM - Greg Shah

Another reason it is not often seen is the use of super in the child class as the reference to call overridden parent methods.

#15 - 03/10/2021 02:14 AM - Marian Edu

Greg Shah wrote:

> o:foo() produces a compile error when called from Extension because o is Base and foo() is protected.

I don't doubt your results. :) I just find it surprising that the 4GL does not allow child classes to access protected methods in the parent instance. Being able to access a parent class' protected methods is a standard feature of the protected concept and it certainly is implemented that way in Java.

I don't think this has anything to do with 'protected', as with Java there is an exception when from a class one can see everything from another instance of the same type... you can access everything if the instance it's the same class as "this", even private members so really isn't about inheritance.

Then again, the 4GL lacks the concept of a package as well, so there is a range of things missing in that implementation. Still, it seems wierd to me.

I'm not very impressed with the package protection concept, I find that more like an anti-pattern one of those "just because you can doesn't mean you should" kind of situations :)

#16 - 06/30/2021 12:12 PM - Greg Shah

I'm working on parsing a new application and it depends on this feature.

Constantin: Do you have an updated patch that you are proposing for this fix?

#17 - 06/30/2021 12:47 PM - Constantin Asofiei

Greg Shah wrote:

> Constantin: Do you have an updated patch that you are proposing for this fix?

No, I don't have a better patch.

#18 - 07/02/2021 02:42 PM - Greg Shah

- Assignee changed from Constantin Asofiei to Greg Shah

#19 - 07/02/2021 02:48 PM - Greg Shah

- Status changed from WIP to Review

- File static_method_lookup_quirk.patch added

- % Done changed from 0 to 100

The attached fixes this issue, including rewriting the class to the correct parent class name during downstream conversion. I plan to include it in 3821c with some other changes that I have pending.

Constantin: Please review.

#20 - 07/08/2021 11:52 AM - Greg Shah

Fixed in 3821c revision 12645.

#21 - 07/25/2021 05:28 AM - Constantin Asofiei

There is a weird bug with the changes from convert/oo_calls.rules, in 12645. This code:

```
      <!-- emit class names as referents for a static member -->
      <rule>evalLib("isStaticClassReference", this)
         <action>
            createJavaAst(java.reference,
                          #(java.lang.String) execLib("resolveClassReference", copy, null),
                          closestPeerId)
         </action>
      </rule>
```

originally was:

```
      <rule>this.type == prog.class_name                              and
            parent.type == prog.object_invocation                     and
            this.indexPos == 0
            <!--
               following is required to not emit in case of a SUBSCRIBE(oo.Bar:staticMethod)
            -->
            and not(parent.parent.type == prog.object_invocation and
                    parent.parent.firstChild.type == prog.class_event)
```

If I move back to the original code, the classes bellow convert fine. Otherwise, the ErrorChanged:Publish (this-object, oo.EventArgs:Empty). line is missing the oo.EventArgs:Empty argument. The two files are:

1. oo/ExceptionService.cls:

```
class oo.ExceptionService:

    define static event ErrorChanged void (
        input sender as Progress.Lang.Object,
        input evt as oo.EventArgs).

    method protected void PublishErrorChanged ():
        ErrorChanged:Publish (this-object, oo.EventArgs:Empty).
    end method.

end.
```

2. oo/EventArgs.cls

```
class oo.EventArgs:
    define public static property Empty as oo.EventArgs no-undo
        get.
        private set.
end class.
```

The 'bug' in 12645 is because the original code has the following is required to not emit in case of a ... comment inlined in the rule's logical expression. Once a comment is found, looks like TRPL drops everything after it!

So this condition:

```
and not(parent.parent.type == prog.object_invocation and parent.parent.firstChild.type == prog.class_event)
```

is not actually used!

And, to make things even more confusing, if I enable -Drules.tracing=true for the original way (without the function call), the conversion fails the same way: publish_errorChanged(ObjectOps.thisObject(),); is emitted.

#22 - 07/26/2021 08:32 AM - Greg Shah

> and not(parent.parent.type prog.object_invocation and parent.parent.firstChild.type prog.class_event)
>
> is not actually used!

Please just drop that code then. I didn't put it there, I just moved it. I have no reason to believe we need to maintain it.

#23 - 07/26/2021 10:22 AM - Constantin Asofiei

Greg Shah wrote:

> > and not(parent.parent.type prog.object_invocation and parent.parent.firstChild.type prog.class_event)
> >
> > is not actually used!
>
> Please just drop that code then. I didn't put it there, I just moved it. I have no reason to believe we need to maintain it.

The condition is required for oo.Foo:evt:(un)subscribe(oo.Bar:m1). cases. I've fixed the condition in 3821c rev 12718.

Files

modifiers.zip	1.5 KB	03/07/2021	Constantin Asofiei
static_method_lookup_quirk.patch	11 KB	07/02/2021	Greg Shah