Base Language - Feature #3867

direct java class access from 4GL code

01/11/2019 10:14 AM - Greg Shah

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

Description

Related issues:

Related to Base Language - Feature #3751: implement support for OO 4GL and st...	**Closed**
Related to Conversion Tools - Bug #4620: Direct Java conversion fails for arr...	**New**

History

#1 - 01/11/2019 10:14 AM - Greg Shah

- Related to Feature #3751: implement support for OO 4GL and structured error handling added

#2 - 01/11/2019 10:28 AM - Greg Shah

Once OO 4GL features are available, it is a natural extension to use the same syntax for Java class access. For example:

```
using java.util.* from java.
using java.lang.* from java.

def var hmap as HashMap.
def var whatever as char init "yep".

hmap = new HashMap().

hmap:put("something", whatever).

if hmap:containsKey("nothing") then
do:
   message "not here".
end.

System:out:println("statics should work too").
```

This is similar to how the 4GL supports direct .NET classes, except that there is no "bridge" needed since the converted code is already in the JVM and is real Java code. The other thing that the 4GL does is there is a implicit type conversion that occurs between the core 4GL and .NET data types.

Since we already have wrapper classes that can be used to pass 4GL data to Java, there is not a hard requirement for this. However, we may want to implement a similar facility so that any 3rd party code can be used in a simple way. If we don't do this, then we need to ensure that there are simple ways for the 4GL code to explicitly cast and/or convert data into the core Java types.

I would expect that much of the work for enabling this would be at parse time since we will need to dynamically resolve classes, methods, data members. This means that during parsing, any Java code that is accessed must be present in the classpath.

Conversion will have some changes as well to enable the rest of this. The runtime changes are minimal except for any extra features needed for casting/type conversion.

This will be a very powerful capability. It will enable direct integration of any Java technology.

This will also allow us to implement extensions that do not need to be written as handle-based resources. Things like our JasperReports or SMTP Email support can simply be Java code that is written for easy usage from converted 4GL code.

#3 - 01/11/2019 10:29 AM - Greg Shah

Does anyone see any additional problems or effort needed that I missed above?

#4 - 08/19/2019 05:12 PM - Greg Shah

- Status changed from New to WIP

- Assignee set to Greg Shah

I have started work on this. I'm planning to put the changes in 4069a.

#5 - 09/26/2019 10:57 AM - Greg Shah

Status

4069a revision 11370 is the first set of changes related to this task.

- The parser and the SymbolResolver have been modified to add the KW_JAVA as an alternative in the USING statement. When a Java class or package is referenced in a USING statement, the 4GL code MUST include a FROM JAVA clause. Otherwise we will not treat that USING statement as related to Java processing. Forcing this explicitly seems to be no real cost to future 4GL devs but it makes the 4GL code more clear and it ensures there is no conflict in searching with legacy 4GL/.NET cases.
- The SymbolResolver has been modified to honor Java in class loading, class lookup, class name resolution and annotations.
- Added a JavaClassDefinition as a subclass of ClassDefinition and a helper class JavaFieldDefinition as a subclass of Variable. These two honor the minimum API so that the parser and SymbolResolver can operate properly with Java classes/methods/constructors/fields. The core of it uses reflection to dynamically resolve/lookup methods/constructors/fields, inspect/return data about them and process annotations. Stubbed out all code used just for handling the builtin 4GL classes or which can only be used from a 4GL class/interface/enum definition.

The method lookup is tricky. When doing our early lookup processing (i.e. before we have parsed the parameter list), we use JavaClassDefinition.guessMethod() to find any method with the same name, access rights and static/instance. If we get at least one back, we return the first item in the list.

We handle annotations after we have a signature for lookup. In this case, I've created a multi-tier search algorithm (see JavaClassDefinition.lookupMethodWorker()).

1. Exact Matching - Convert the parameters into their direct Java class counterparts and see if there is an exact match (see exactMethodLookup()).
2. Fuzzy Matching (see fuzzyMethodLookup())
    - Make a list those methods/constructors which match in name (methods only), number of parameters, access mode and static/instance. See getCandidateMethods().
    - Iterate through that list and process each with testFuzzyMatch() matching logic.
        - This method iterates through every parameter and compares the required method parameter with one of the following caller's parameter (by order of precedence):
            1. The caller's type is the same exact type as the method parameter being checked.
            2. For a caller's type as object<? extends something> it checks if the method's parameter is something.
            3. Is the method's parameter in the set of "near matches" which is the list possible boxing transformations for the caller's type?
            4. Is the method's parameter isAssignableFrom() the caller's type?
            5. For a caller's type as object<? extends something> is the method's parameter isAssignableFrom() the something type?
        - The number of exact (case 1), unwrap (case 2), near (case 3) and parent (cases 4 and 5) matches is recorded.
        - If all parameters match in one of the 5 ways, then this is a match and a FuzzyMatch instance is returned with all the matching metrics.
        - Any non-null FuzzyMatch is added to a TreeSet<FuzzyMatch>. This is a naturally sorted set and FuzzyMatch implements Comparable so that it has a natural sorting order (higher precedence sorts first):
            - The combined number of exact matches and unwrap matches.
            - Near matches.
            - Parent matches.
            - The order in which the method was returned by Java reflection.
    - After iteration through the candidates, if the TreeSet is empty then there is no fuzzy match.
    - If the TreeSet is non-empty, then the first entry is considered the best match and that is returned.

Design Points

- We support static and instance method calls.
- We support static and instance member field references.
- We require 4GL code to use case-sensitive names for all Java names (classes/interfaces, methods and fields).
- Only INPUT mode parameters are supported (you can leave off the mode or specify INPUT, but any use of INPUT-OUTPUT or OUTPUT will generate an error).
- USING
    - You must include from java in any USING statement that references either a Java package or an explicit Java class.
    - All USING statements must be explicit. There is no implicit USING java.lang.* FROM JAVA.
    - There is no equivalent to the PROPATH search for a Java class. Any search from USING some.package.* FROM JAVA will add that package to the list of package prefixes that will be prepended to possible Java class names to try to get the Class<?> instance for that name. We use Class.forName() to obtain the class instance. This will lookup from whatever sources are in the Java classpath for the ConversionDriver. This is not the same idea as a PROPATH lookup which is inherently file-system based.
    - Any Java classes that are referenced must be present in the classpath of both conversion and the FWD server runtime.
- Boxing/Unboxing

- The intention is to map the following to/from natural Java types:
  - method return values
  - method parameters
  - member field access
  - member field assignment
- Supported conversions:
  - character
    - java.lang.String
    - longchar
  - date
    - java.util.Date
    - datetime
    - datetimetz
  - datetime
    - java.sql.Timestamp
    - datetimetz
  - decimal
    - double
    - java.lang.Double
    - java.lang.BigDecimal
  - integer
    - int
    - java.lang.Integer
    - long
    - java.lang.Long
    - double
    - java.lang.Double
    - java.lang.BigDecimal
  - int64
    - long
    - java.lang.Long
    - java.lang.BigDecimal
  - logical
    - boolean
    - java.lang.Boolean
  - longchar
    - java.lang.String
  - raw
    - byte[]
- Both array and non-array forms should be handled.
  - Non-array forms should be handled by helper methods in a given type (e.g. integer.intValue()).
  - Array forms will need static helper methods to convert at runtime. For example, a character extent var should be able to be passed as a String[] parameter (e.g. String[] c2s(character[] c)).

- No support for:
  - generics
  - varargs
  - multi-dimensional arrays
  - lambdas

Known Issues and Next Steps

- The example in #3867-2 parses everything except the System:out:println(...). code at the end.
- Conversion of the method calls and member field access/assignment must be added.
- Boxing/Unboxing
  - Method Parameters
    - Parse time support is handled up to the annotations for methods, including leaving behind "wrap_parameter" and "classname" annotations.
    - I suspect more work is needed here, to handle unwrapping of object.ref() forms.
    - Conversion support is not there yet.
    - Runtime conversion helpers are not written yet.
  - Method Return Value - nothing has been done on this yet.
  - Member Field Usage - nothing has been done on this yet.

Future Steps

- Inheritance of OO 4GL class from Java class.
- Allow differentiating of name conflicts in an AS clause by adding the KW_JAVA (e.g. def var i as java class Integer would force a lookup of the Java class in this case).
- Using an OO 4GL class type in Java code as a method parameter, method return type or field data type.

#6 - 09/26/2019 11:01 AM - Greg Shah

The testcase in #3867-2 is checked in as testcases/uast/direct_java_usage_simple.p.

#7 - 09/27/2019 05:07 AM - Constantin Asofiei

Greg, I can parse and convert (but not compile) your sample. Some questions:

- should the Java objects be defined as java.util.HashMap or via the object<? extends java.util.HashMap>? If we are not using BDT object, then we have problems with the final var can not be changed issue, if you recall. And if we are using BDT object, then the runtime needs changes (can't tell how major these changes are), as i.e. object class is defined as T extends _BaseObject_ - and it can't hold the java.lang.Object instances.
- for System.out.println, the parser finds the println(java.lang.Object) as a match, instead of println(java.lang.String). I don't think this is a big problem, if the corresponding API exists (and it exists in this case), to unbox i.e. character to java.lang.String.
- do you want more complex examples than direct_java_usage_simple.p?

I still need to add the boxing/unboxing.

#8 - 09/27/2019 07:37 AM - Constantin Asofiei

Another question is using Java-style vars in 4GL constructs. Think something like this:

```
def var ji as java.lang.Integer.
do ji = 1 to 10:
   message ji.
end.
```

How should this behave? It matters because is dependant how ji gets converted: as object<? extends java.lang.Integer> or as java.lang.Integer.

#9 - 09/27/2019 07:44 AM - Constantin Asofiei

Or the goal is to just access Java APIs (be it a Java method or a Java field, declared in some existing Java class), and not use the Java types directly from 4GL code?

#10 - 09/27/2019 08:41 AM - Greg Shah

> I can parse and convert (but not compile) your sample.

Exciting! How extensive were the changes for that?

I don't see any new revisions in 4069a. Are your changes safe to check in?

> should the Java objects be defined as java.util.HashMap or via the object? If we are not using BDT object, then we have problems with the final var can not be changed issue, if you recall. And if we are using BDT object, then the runtime needs changes (can't tell how major these changes are), as i.e. object class is defined as T extends *BaseObject* - and it can't hold the java.lang.Object instances.

Hmm. Good point.

I want to avoid the object wrapping and keep the access as clean and direct as possible. A solution would be to make all Java variables into business logic class members so they do not need to be final. But for Java variables, this would mean that the normal variable scoping assumptions would be broken. The primary implication is for top level code blocks that are not external procedures (internal procedures, functions, methods, triggers, property getters/setters). Without extra code, these blocks could not duplicate variable names for Java vars. But we can resolve this by disambiguating any scoped vars that have naming conflicts.

Example: A Java var named hmap is defined in the external procedure and a Java var named hmap is defined in an internal procedure blah, then we would create two different business logic class members, one named hmap and the other named hmap_1 (or hmap_blah or whatever).

Can you see any flaw in this idea?

> for System.out.println, the parser finds the println(java.lang.Object) as a match, instead of println(java.lang.String). I don't think this is a big problem, if the corresponding API exists (and it exists in this case), to unbox i.e. character to java.lang.String.

Another good point. I had not planned to support varargs and then I put a varargs method into my "simple" example. Not my best moment. :)

Perhaps we should consider supporting varargs after all. I guess it may be an easy thing to encounter by accident.

> do you want more complex examples than direct_java_usage_simple.p?

Yes, I think we should try more method and field variations to ensure the core functionality works.

> How should this behave? It matters because is dependant how ji gets converted: as object or as java.lang.Integer.

object would not help in the TO clause of the DO block. There is merit in allowing such usage but there are some cases which may need to be excluded. I can envision direct boolean usage in if statements and so forth, which actually should not be too hard. The problem here is that the control flow mechanism is assigning ji internally. The fact that Integer is immutable and it also doesn't mimic all the behavior (unknown etc..) of integer means that boxing/unboxing can't help here. It will depend on the control flow mechanism. Some exclusions will be needed.

> Or the goal is to just access Java APIs (be it a Java method or a Java field, declared in some existing Java class), and not use the Java types directly from 4GL code?

We definitely DO want to use the Java types directly in 4GL code but some control flow cases are special. Anything that can be handled with simple boxing/unboxing should "just work". I think the message ji should work, but the do ji = 1 to 10 should not. I also expect that we should be able to pass these as parameters to converted 4GL code. I suspect the parameter helper code will need to changes to enable that.

We can accept limits (exclusions) on assignment of anything immutable.

#11 - 09/27/2019 09:02 AM - Constantin Asofiei

Greg Shah wrote:

> I can parse and convert (but not compile) your sample.

Exciting! How extensive were the changes for that?

Just some pinpointed fixes, for now. See 4069a rev 11371

> Example: A Java var named hmap is defined in the external procedure and a Java var named hmap is defined in an internal procedure blah, then we would create two different business logic class members, one named hmap and the other named hmap_1 (or hmap_blah or whatever). Can you see any flaw in this idea?

The problem here is that any recursive call will be broken... we had this in the past, where we promoted local vars as instance fields, and it took us a while to figure out some bugs. I'd like to avoid promoting vars scoped to top-level blocks, as this breaks the var's lifetime completely.

> Another good point. I had not planned to support varargs and then I put a varargs method into my "simple" example. Not my best moment. :)

I didn't mean varags, I meant that java.lang.Object has priority when looking up a method, over the one which can match exactly (after unboxing).

> object would not help in the TO clause of the DO block. There is merit in allowing such usage but there are some cases which may need to be excluded. I can envision direct boolean usage in if statements and so forth, which actually should not be too hard. The problem here is that the control flow mechanism is assigning ji internally. The fact that Integer is immutable and it also doesn't mimic all the behavior (unknown etc..) of integer means that boxing/unboxing can't help here. It will depend on the control flow mechanism. Some exclusions will be needed.

Good points.

#12 - 09/27/2019 09:35 AM - Greg Shah

> The problem here is that any recursive call will be broken... we had this in the past, where we promoted local vars as instance fields, and it took us a while to figure out some bugs. I'd like to avoid promoting vars scoped to top-level blocks, as this breaks the var's lifetime completely.

Perhaps we need a jobject<? extends java.lang.Object> that is a locally scoped final instance which can be unwrapped/accessed, assigned and passed around. I guess it never gets passed as a parameter of type object which the object cases all derive from _BaseObject_ which is a separate hierarchy. I that is correct, then passing this as a parameter is about boxing/unboxing to the BDT primitive types OR for cases where the 4GL code is defined with parameter types that are Java objects outside of the _BaseObject_ hierarchy (e.g. java.util.HashMap).

Can you see a flaw in my assumption that jobject doesn't ever have to be passed to object and vice versa?

> I didn't mean varags, I meant that java.lang.Object has priority when looking up a method, over the one which can match exactly (after unboxing).

I intended the fuzzy method lookup to prioritize the near matches (boxing/unboxing) over the parent matches. This sounds like a bug.

#13 - 09/27/2019 09:50 AM - Constantin Asofiei

Greg Shah wrote:

> > The problem here is that any recursive call will be broken... we had this in the past, where we promoted local vars as instance fields, and it took us a while to figure out some bugs. I'd like to avoid promoting vars scoped to top-level blocks, as this breaks the var's lifetime completely.
>
> Perhaps we need a jobject<? extends java.lang.Object> that is a locally scoped final instance which can be unwrapped/accessed, assigned and passed around. I guess it never gets passed as a parameter of type object which the object cases all derive from _BaseObject_ which is a separate hierarchy. I that is correct, then passing this as a parameter is about boxing/unboxing to the BDT primitive types OR for cases where the 4GL code is defined with parameter types that are Java objects outside of the _BaseObject_ hierarchy (e.g. java.util.HashMap).
>
> Can you see a flaw in my assumption that jobject doesn't ever have to be passed to object and vice versa?

I think this can work; the alternative is to create a i.e. java.util.HashMap[] someVar = new java.util.HashMap[1]; and use this someVar[0] reference, to avoid the 'effective final' problem. We already do this for some cases of extents. But the code will be a little weird to read.

The other good point about using jobject is that it makes the reference mutable, and we could enable OUTPUT/INPUT-OUTPUT when passing this to 4GL methods (but not Java methods).

#14 - 09/27/2019 03:02 PM - Constantin Asofiei

A limitation here is that any file system access, process launching, etc, is done from the FWD server process (and machine). I'm trying to find an example which would show how to use Java code from 4GL 'in real life'.

#15 - 09/27/2019 04:34 PM - Greg Shah

Yes, this opens up potential security (and reliability) issues. I think we are offering this under the credo: "with great power comes great responsibility".

> I'm trying to find an example which would show how to use Java code from 4GL 'in real life'.

A simple example might use trigonometric functions that are easily available in Java but missing from the 4GL.

For things like JasperReports integration or SMTP email support, we might have implemented differently if we had this support. Helper classes might be enough for some really powerful capabilities, by using this direct Java access.

#16 - 09/29/2019 11:49 AM - Constantin Asofiei

Can we assume normal Java behaviour (e.g. NPE throw) for cases where the variable is uninitialized, and it is used in an expression, like ji + 1? Same for cases like if jl then message "yes, can do!".

I ask because assignments like pi = ji + 1 or ji = ji + 1 should be converted like pi.assign(ji.ref() + 1); and ji.assign(ji.ref() + 1));, but this will not work if we want to assign ji to null if one operand is null/unknown.

#17 - 09/29/2019 12:43 PM - Greg Shah

I think we can assume normal Java NPE behavior, with some caveats.

- We will need to allow Java exceptions in 4GL catch blocks.
- At this time we haven't considered that pure Java operators would be used. This means that we are assuming that the converted 4GL operators would be in use. pi = ji + 1 should be converted as pi.assign(plus(ji.ref(), 1)); and we **should** be handling null safely in our operator methods, treating it as unknown value. So I actually would not expect an NPE in this case.
- I guess assigning unknown to a jobject should set the reference to null.

#18 - 09/30/2019 09:50 AM - Constantin Asofiei

Greg Shah wrote:

> - At this time we haven't considered that pure Java operators would be used. This means that we are assuming that the converted 4GL operators would be in use. pi = ji + 1 should be converted as pi.assign(plus(ji.ref(), 1)); and we **should** be handling null safely in our operator methods, treating it as unknown value. So I actually would not expect an NPE in this case.

I understand, but the problem here is that I need to convert all MathOps APIs to use Java objects instead of native types (like java.lang.Double instead of double). But this will not automatically convert an int literal to a Double instance, if there is no explicit cast... so I need to add more APIs to treat each Java number type.

#19 - 09/30/2019 09:57 AM - Constantin Asofiei

And also about using jobject.ref(); for a call like Math:sin(jrad), where jrad is java.lang.Double, if this gets converted to Math.sin(jrad.ref()) and jrad is unknown, I have no way to prevent the NPE.

I think I need to emit a non-null check for cases where a Java var is used in 4GL APIs (like MathOps.plus), and a null check in cases where we pass this value to an API which requires a Java native value (like Math.sin). This will allow me to raise an ERROR condition if the jobject instance is unknown, before passing this as argument.

#20 - 09/30/2019 11:35 AM - Greg Shah

I think the 4GL operators would require wrapping the Java type in the correct BDT. Then the 4GL operators will work naturally.

#21 - 09/30/2019 11:39 AM - Greg Shah

> I think I need to emit a non-null check for cases where a Java var is used in 4GL APIs (like MathOps.plus),

Using BDT wrapping, a null should be treated like unknown value, which we already handle in the operators.

> and a null check in cases where we pass this value to an API which requires a Java native value (like Math.sin). This will allow me to raise an ERROR condition if the jobject instance is unknown, before passing this as argument.

How about a variant of ref() like refNonNull() which will raise ERROR if the reference is null.

#22 - 10/01/2019 06:52 PM - Constantin Asofiei

Greg, I have a code like this:

```
jrad = jdeg * Math:PI / 180.

jsin = Math:sin(jrad).
jcos = Math:cos(jrad).
jtan = Math:tan(jrad).
jcot = Math:atan(jrad).

jsum = jsum + 1.
```

which is working from internal procedures, functions, or OO methods, with arguments/parameters either 4GL or Java-style, input, output and output-mode combinations. This converts to something like:

- if the vars are Java types:

```
jrad_1.assign(divide(multiply(new decimal(jdeg_1), java.lang.Math.PI), 180));
```

```
jsin_1.assign(java.lang.Math.sin(jrad_1.ref()));
jcos_1.assign(java.lang.Math.cos(jrad_1.ref()));
jtan_1.assign(java.lang.Math.tan(jrad_1.ref()));
jcot_1.assign(java.lang.Math.atan(jrad_1.ref()));
jsum_1.assign(plus(new decimal(jsum_1), 1));
```

- for 4GL types:

```
prad_1.assign(divide(multiply(pdeg_1, java.lang.Math.PI), 180));
psin_1.assign(java.lang.Math.sin(jobject.toJava(double.class, prad_1).ref()));
pcos_1.assign(java.lang.Math.cos(jobject.toJava(double.class, prad_1).ref()));
ptan_1.assign(java.lang.Math.tan(jobject.toJava(double.class, prad_1).ref()));
pcot_1.assign(java.lang.Math.atan(jobject.toJava(double.class, prad_1).ref()));
psum_1.assign(plus(psum_1, 1));
```

I need to cleanup the code and comment it, will do it tomorrow. I think Known Issues and Next Steps can be considered fixed. What I haven't touched:

- array/extent values
- testing for byte[], java.util.Date and java.sql.Timestamp

#23 - 10/02/2019 10:16 AM - Greg Shah

This looks very good. Check in the sample to testcases/uast/ when you are ready.

#24 - 10/02/2019 11:50 AM - Constantin Asofiei

The FWD changes are in 4069a rev 11387

The test is in rev 1950.

#25 - 10/15/2019 05:29 PM - Constantin Asofiei

There is a regression related to jobject.fromJava in 4069a. Working on a fix.

#26 - 10/15/2019 06:21 PM - Constantin Asofiei

I have a fix for this, will commit after regression testing.

#27 - 10/16/2019 01:44 AM - Constantin Asofiei

Fixed in 4069a rev 11428.

#28 - 11/13/2019 04:18 PM - Greg Shah

Task branch 4069a has been merged to trunk as revision 11339. This is the majority of the necessary support. There is one known issue which will be worked next.

#29 - 11/13/2019 04:19 PM - Greg Shah

- % Done changed from 0 to 70

#30 - 11/19/2019 05:28 PM - Greg Shah

At this time, the 4GL catch block is incompatible with Java's checked exceptions. This is due to the fact that we are handling these as delegated processing inside the BlockManager and there is no try/catch in the converted Java code. I'm working on this right now.

#31 - 11/19/2019 05:34 PM - Greg Shah

A testcase like this:

```
def var i as int.

repeat:
   i = i / 9000.
end.

catch e as Progress.Lang.AppError:
   message "caught 1".
end.

catch e as Progress.Lang.Error:
   message "caught 2".
end.
```

Results in this Java code:

```
...
public class SimpleCatch3
{
   /**
    * External procedure (converted to Java from the 4GL source code
    * in oo/simple_catch_3.p).
    */
   public void execute()
   {
      integer i = UndoableFactory.integer();

      externalProcedure(SimpleCatch3.this, new Block((Body) () ->
      {
         repeat("loopLabel0", new Block((Body) () ->
         {
            i.assign(divide(i, 9000));
         }));
      }));
   }
}
```

Notice that the catch blocks are missing in Java. If I remove the REPEAT block, the two catchError() instances are emitted as you would expect.

I think the problem is in convert/control_flow.rules:

```
      <rule>type == prog.kw_catch and parent.type == prog.statement
         <rule>copy.getAncestor(3).type == prog.inner_block
            <action>controlid = getReferenceNoteLong(copy.getAncestor(3).id, "controlid")</action>
         </rule>

         <action>
            lastid = createPeerAst(java.static_method_call, "catchError", controlid)
```

```
         </action>
...
```

Any catch block directly inside any top level block will not be parented at a prog.inner_block. This means that controlid will have whatever value it already has. If there are no nested blocks, then this is not a problem because the creation of the top level block in Java will leave a valid controlid. But as soon as you put a nested block there (not a simple DO, but something with properties) then the controlid will no longer be a valid value and we probably try to attach the catchError() calls to a non-existent init() method of the nested block. I think if the nested block had an init() section, then the catchError() would be attached there.

Constantin: What do you think?

I plan to fix this as part of my work. This seems like a common case. It must be affecting real code in current projects.

#32 - 11/19/2019 05:37 PM - Constantin Asofiei

Greg Shah wrote:

> Any catch block directly inside any top level block will not be parented at a prog.inner_block. This means that controlid will have whatever value it already has. If there are no nested blocks, then this is not a problem because the creation of the top level block in Java will leave a valid controlid. But as soon as you put a nested block there (not a simple DO, but something with properties) then the controlid will no longer be a valid value and we probably try to attach the catchError() calls to a non-existent init() method of the nested block. I think if the nested block had an init() section, then the catchError() would be attached there.
>
> Constantin: What do you think?

Yes, I missed to test this case. The controlid now is leaked from whatever other inner-block was previously processed; this needs to be for the top-level block.

#33 - 11/21/2019 04:59 PM - Greg Shah

In BlockManager.topLevelBlock(), it seems like this code should be aware of BLOCK-LEVEL/ROUTINE-LEVEL ON ERROR UNDO, THROW.:

```
               catch (ErrorConditionException err)
               {
                  wa.tm.notifyCondition(err);
                  wa.tm.honorError(err);
                  wa.tm.rollback("methodScope");
                  wa.tm.triggerRetry("methodScope");
               }
```

Also, I think this TODO in BlockManager.processCondition() is probably also an issue:

```
            case THROW:
               if (ce instanceof LegacyErrorException)
               {
                  throw (LegacyErrorException) ce;
               }
               else
               {
                  // TODO: build a SysError?
               }
               break;
```

The SysError may only be needed in the case of BLOCK-LEVEL ON ERROR UNDO, THROW. (but not ROUTINE-LEVEL ON ERROR UNDO, THROW.) is specified, it is not clear to me.

Constantin: Please provide your thoughts.

#34 - 12/13/2019 04:44 PM - Greg Shah

Our current parsing for catch blocks creates an INNER_BLOCK child of the KW_CATCH. This results in the following Java:

```
         catchError(com.goldencode.p2j.oo.lang.LegacyError.class,
         e2 ->
         {
            blockLabel6:
            {
               message("external proc 2");
            }
         });
```

Notice the blockLabel6 and the extra curly braces. I don't see any reason for these. I plan to remove the INNER_BLOCK node and clean up the result.

Constantin: Do you see any flaw in my logic?

#35 - 12/13/2019 05:15 PM - Constantin Asofiei

Greg Shah wrote:

> Constantin: Do you see any flaw in my logic?

No, I can't see any way to jump to the catch block in 4GL.

#36 - 01/09/2020 04:24 PM - Greg Shah

I just committed revision 11467 to branch 4335a (which is derived from 3809e revision 11466). This revision provides the following features/fixes:

- Fixes a latent bug which did not emit CATCH blocks properly for top-level blocks which had nested inner blocks (see #3867-31).
- Removes the extra/unnecessary block emitted in CATCH blocks (see #3867-34).
- Adds the ability to reference a fully qualified Java classname directly as a type (e.g. DEF VAR hmap AS java.util.HashMap.) or in a NEW. No corresponding USING ... FROM JAVA is needed for this.
- Supports CATCH blocks that reference Java exceptions (checked or unchecked) such that the code will emit a proper try/catch into the business logic so that javac can be satisfied for any checked exceptions.

In regard to the new checked exceptions support, it works like this:

1. We implement all Java exceptions the same way. This means that unchecked exceptions (anything deriving from Error or RuntimeException) will emit the same way as any checked exception (e.g. IOException). In both cases, we emit a try and catch into the Java code. In the catch block, we emit the block as a lambda that is passed to BlockManager.processCatch(). The block manager does not catch the exception directly. The business logic catches it and then calls to processCatch() inside the catch block itself.

2. The 4GL exceptions are emitted as before (although without the extra/unnecessary label plus braces). This means it is still delegated to the block manager using BlockManager.catchError(). The try/catch is in the BlockManager in this case and the lambda is passed to the BM in the init() of the containing Block instance.

3. It is perfectly valid to include catch blocks for Java and 4GL exceptions. The conversion emits each according to the type as described above.

To illustrate the results, this 4GL code:

```
using java.lang.* from java.

def var i as int.

function f returns int ():
   message "whatever".

   catch e as java.lang.RuntimeException:
      message "user-defined function".
   end.
end.

/* CATCH block may only be associated with an undoable block. (14140) */
/*
do:
   run bogus.

   catch e as Progress.Lang.AppError:
      message "simple do".
   end.
end.
*/

do transaction:
   run bogus.

   catch e as java.lang.RuntimeException:
      message "transaction do".
   end.
end.
```

```
repeat:
   i = i / 9000.
   leave.

   catch e as java.lang.RuntimeException:
      message "repeat".
   end.
end.

on f10 anywhere
do:
   message "whatever".

   catch e as java.lang.RuntimeException:
      message "UI trigger".
   end.
end.

/* CATCH block may only be associated with an undoable block. (14140) */
/*
update i editing:
   readkey.
   apply lastkey.

   catch e as Progress.Lang.AppError:
      message "editing block".
   end.
end.
*/

catch e as java.lang.RuntimeException:
   message "external proc".
end.

catch e2 as Progress.Lang.Error:
   message "external proc 2".
end.

procedure bogus:
   message "whatever".

   catch e as java.lang.RuntimeException:
      message "internal proc".
   end.
end.
```

will generate this Java code:

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

import com.goldencode.p2j.util.*;
import java.lang.*;
import com.goldencode.p2j.ui.*;

import static com.goldencode.p2j.util.BlockManager.*;
import static com.goldencode.p2j.util.InternalEntry.Type;
import static com.goldencode.p2j.util.MathOps.*;
import static com.goldencode.p2j.ui.LogicalTerminal.*;

/**
 * Business logic (converted to Java from the 4GL source code
 * in oo/catch_blocks_in_many_variations_with_direct_java.p).
 */
public class CatchBlocksInManyVariationsWithDirectJava
{
   /**
    * External procedure (converted to Java from the 4GL source code
    * in oo/catch_blocks_in_many_variations_with_direct_java.p).
    */
   public void execute()
   {
      integer i = UndoableFactory.integer();
```

```
      externalProcedure(CatchBlocksInManyVariationsWithDirectJava.this, new Block((Init) () ->
      {
         catchError(com.goldencode.p2j.oo.lang.LegacyError.class,
         e2 ->
         {
            message("external proc 2");
         });
      },
      (Body) () ->
      {
         try
         {
            doBlock(TransactionType.FULL, "blockLabel0", new Block((Body) () ->
            {
               try
               {
                  ControlFlowOps.invoke("bogus");
               }
               catch (java.lang.RuntimeException e)
               {
                  processCatch(() ->
                  {
                     message("transaction do");
                  });
               }
            }));

            repeat("loopLabel0", new Block((Body) () ->
            {
               try
               {
                  i.assign(divide(i, 9000));
                  leave("loopLabel0");
               }
               catch (java.lang.RuntimeException e_1)
               {
                  processCatch(() ->
                  {
                     message("repeat");
                  });
               }
            }));

            registerTrigger(new EventList("f10", true), CatchBlocksInManyVariationsWithDirectJava.this, () ->
new Trigger()
            {
               public void pre()
               {
               }

               public void init()
               {
               }

               public void body()
               {
                  try
                  {
                     message("whatever");
                  }
                  catch (java.lang.RuntimeException e_2)
                  {
                     processCatch(() ->
                     {
                        message("UI trigger");
                        /* CATCH block may only be associated with an undoable block. (14140) */
                        /*
                        update i editing:
                           readkey.
                           apply lastkey.

                           catch e as Progress.Lang.AppError:
                              message "editing block".
                           end.
```

```
                        end.
                        */
                     });
                  }
               }
            });
         }
         catch (java.lang.RuntimeException e_2)
         {
            processCatch(() ->
            {
               message("external proc");
            });
         }
      }));
   }

   @LegacySignature(type = Type.FUNCTION, name = "f")
   public integer f()
   {
      return function(this, "f", integer.class, new Block((Body) () ->
      {
         try
         {
            message("whatever");
         }
         catch (java.lang.RuntimeException e)
         {
            processCatch(() ->
            {
               message("user-defined function");
               /* CATCH block may only be associated with an undoable block. (14140) */
               /*
               do:
                  run bogus.

                  catch e as Progress.Lang.AppError:
                     message "simple do".
                  end.
               end.
               */
            });
         }
      }));
   }

   @LegacySignature(type = Type.PROCEDURE, name = "bogus")
   public void bogus()
   {
      internalProcedure(new Block((Body) () ->
      {
         try
         {
            message("whatever");
         }
         catch (java.lang.RuntimeException e_3)
         {
            processCatch(() ->
            {
               message("internal proc");
            });
         }
      }));
   }
}
```

I've checked in several examples in testcases/uast/oo/. The example above is oo/catch_blocks_in_many_variations_with_direct_java.

#37 - 01/09/2020 04:28 PM - Greg Shah

I believe that we are at a good "first pass" level of support here. A future improvement would be to support inheriting 4GL classes from Java classes, but I think that will be out of scope for now.

Constantin/Hynek: Do you know of any other critical bugs or missing features before we can accept this level as a working first version?

#38 - 01/10/2020 01:12 AM - Hynek Cihlar

Greg Shah wrote:

> Constantin/Hynek: Do you know of any other critical bugs or missing features before we can accept this level as a working first version?

One problem I had was with referencing enums, importing the enum class with using didn't work. I had to fully qualify the enum class.

#39 - 01/10/2020 04:10 AM - Constantin Asofiei

Greg Shah wrote:

> I believe that we are at a good "first pass" level of support here. A future improvement would be to support inheriting 4GL classes from Java classes, but I think that will be out of scope for now.
>
> Constantin/Hynek: Do you know of any other critical bugs or missing features before we can accept this level as a working first version?

I didn't test much, except what I already fixed in 3809e (related to null values). So I'm OK with this first version.

#40 - 01/15/2020 10:12 AM - Greg Shah

Hynek Cihlar wrote:

> Greg Shah wrote:
>
> > Constantin/Hynek: Do you know of any other critical bugs or missing features before we can accept this level as a working first version?
>
> One problem I had was with referencing enums, importing the enum class with using didn't work. I had to fully qualify the enum class.

I looked into this case. I think you are talking about ladder-safety-gen.p where it references g:setColor(java.awt.Color:RED);? When you use Color:RED, it cannot parse because Color will lex as KW_COLOR which is reserved. When setColor is being processed, the parser looks ahead at LA(2) to check that it is LPARENS and at LA(3) to see if it can be the first token of an expr. KW_COLOR can never appear as the first token of an expr so the code in downstream_chained_reference cannot match with the method_call rule and fails.

I hesitate to make changes for this case because such changes could cause serious parsing problems/compatibility issues with the existing 4GL OO

code. If I think of a way that is safe, I will look into it, but for now I don't see a safe fix.

Currently we will have the limitation that classnames and member names which are reserved keywords might need to be qualified to work as a standalone reference or as the leftmost reference (the "anchor") of a chain of object invocations.

#41 - 01/15/2020 10:13 AM - Greg Shah

- Status changed from WIP to Test

- % Done changed from 70 to 100

#42 - 01/15/2020 11:12 AM - Hynek Cihlar

Greg Shah wrote:

> I looked into this case. I think you are talking about ladder-safety-gen.p where it references g:setColor(java.awt.Color:RED);?

Yes, this was the case.

#43 - 03/13/2020 03:37 PM - Greg Shah

- Status changed from Test to Closed

#44 - 03/13/2020 03:39 PM - Greg Shah

Task branch 4335a was merged to trunk as revision 11345.

#45 - 04/18/2020 06:35 AM - Greg Shah

- Related to Bug #4620: Direct Java conversion fails for arrays of objects added