Database - Bug #8621

better resolution of ambiguous constructors/methods in RuntimeJastInterpreter

04/11/2024 12:26 PM - Eric Faulhaber

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD	**version:**	

Description

History

#1 - 04/11/2024 12:59 PM - Eric Faulhaber

- Subject changed from better resolution of ambiguous constructors in RuntimeJastInterpreter to better resolution of ambiguous constructors/methods in RuntimeJastInterpreter

RuntimeJastInterpreter.findMatchingConstructor currently returns null when it finds more than one c'tor whose signature can be applied to the signature it is trying to match:

```
private static Constructor<?> findMatchingConstructor(Class<?> aClass, Class<?>[] signature)
{
   Constructor<?> ret = null;
   for (Constructor<?> ctor : aClass.getDeclaredConstructors())
   {
      Class<?>[] required = ctor.getParameterTypes();
      if (Function.matchSignature(required, signature, -1, true))
      {
         if (ret != null)
         {
            return null;
            // error: at least two constructors found to match the supplied parameter type list
         }

         // do not return yet, check for sanity
         ret = ctor;
      }
   }
   return ret;
}
```

This will derail, during the query prepare stage, a dynamic query with a substitution parameter which needs to construct an object which has ambiguous constructors (according to the current matching algorithm), because the caller of this method throws an exception when null is returned.

This method currently is being changed to break the for loop and log a warning message when ret != null. However, that change is palliative and diagnostic only. It will make it easier to detect when an ambiguous constructor match occurs, but simply choosing the first c'tor encountered is not a sound approach, and may result in more subtle bugs than a dynamic query failing during the prepare step. Hence, if the warning message appears in the server log, it should not be ignored.

This ambiguity has been encountered in real application code and so far, the short-term solution has been to remove the "extra" constructors which are considered ambiguous by the matching algorithm. The better solution is to fix the matching algorithm to behave the way the Java compiler behaves.

There is concern that implementing a javac-compatible c'tor/method matching algorithm may be CPU intensive. However, I think it is a safe assumption that for the life of a server process, a c'tor signature match, once found, should be stable. So, caching the result for a fast lookup on subsequent occasions should be feasible. **If anyone has a reason this assumption may not be valid, please speak up.**

The core signature matching algorithm in use is Function.matchSignature in the com.goldencode.expr package. Note that this method is used heavily

by TRPL in conversion, and is complicated by its support for varargs. As such, I don't know if this is the best place to implement the change (at minimum, it is not the least risky place). Perhaps the change should be in RuntimeJastInterpreter, but this needs to be considered further.

Note that it looks like we have a similar problem in RJI.findMatchingMethodFromClass. It is a little more forgiving in its current form, in that it returns one of the ambiguous matches, but there is a TODO in that code about implementing a better (i.e., more specific) matching algorithm. So, the solution for this task should likely be common between this method and the c'tor matching method.

#3 - 04/11/2024 01:07 PM - Constantin Asofiei

What we need to implement for method/constructor matching is actually the Java compiler's algorithm for linking the call to the target method/constructor. For fuzzy matching in OpenEdge, we have to compute a score based on the distance between the argument's type and the parameter's type.

For example, in Java, if the argument is int and the parameter is double, then this is a distance of two (because you have int > long > double). Same for objects - get the argument's type and the parameter's type, and calculate the distance between them (i.e. the number of super-classes/interfaces between the argument and the parameter).

Now, what I'm not sure is how the overloads gets sorted by this score - I think lower score should always win, but do you sort them by comparing the type score 'left to right' or calculate a specific, single score, for the entire match?

If the javac's code is in open domain, we can take a look at that and figure out the rules.

Otherwise, I agree, we can always cache the argument's type for this overload call, as this specific argument types will always end up calling the same method.

#4 - 04/11/2024 03:56 PM - Ovidiu Maxiniuc

The caching mechanism is in place since the early implementation. Since the structure of the JAST in which the constructor/method does not change, it would be fair to assume the type of parameters do not change, even if some values (which may be initially constant) or parameters do.

However, this is an **interpreter** and not a compiler, and the variable are typeless. This means it knows the type of arguments/variables at the execution time, by inspecting the current value. For example: if we have a method which returns a NumberType. And the result is used a parameter for an overloaded process method, with following signatures: (NumberType), (int64), (decimal). Evidently, the compiler compiler will chose process(NumberType), based the static information available at the time it is executed. OTOH, RJI takes the result which is an int64, for example (it cannot be a NumberType because it is abstract, anyway), and will attempt to match one of the signatures. Of course, it will choose the process(int64), the choice of the compiler will never be picked by the interpreter.

While the above is a good thing for _poly functions, it will not work as the compiler each time. At the same time, we expect that the static and dynamic queries to have the same outcome, but this depends on the actual methods/constructors called.

The main conclusion here is that overloaded methods and constructors with similar signatures must be implemented in such a way that the result is identical, regardless of which instance RJI will choose.