Debugger

Debugger

MPS provides an API for creating custom debuggers as well as integrating with debugger for java. See Debugger features overview for a description of MPS debugger features.

Integration with java debugger

To integrate your java-generated language with java debugger, provided by MPS, you should specify:

- on which nodes breakpoints could be created;
- nodes which should be traced;
- how to start your application under debug;
- custom viewers for your data.

Not all of those steps are absolutely necessary; which are – depends on the language. See next parts for details.

Nodes to trace and breakpoints

> ⚠ This part is different in MPS2.0 See the changes in MPS 2.0 documentation.

Suppose you have a language, let's call it highLevelLanguage which generates code on some lowLevelLanguage, which in turn is generated directly into text (there can be several other languages between highLevelLanguage and lowLevelLanguage, it does not really metter). Suppose that the text generated from lowLevelLanguage is essentially java, and you whant to have your highLevelLanguage integrated with java debugger. See the following table:

	lowLevelLanguage is baseLanguage	lowLevelLanguage is not baseLanguage
highLevelLanguage extends baseLanguage (uses concepts `Statement`,`Expression`, `BaseMethodDeclaration` etc)	Do not have to do anything.	Fully implement `DebugInfoInitializer` for lowLevelLanguage.
highLevelLanguage does not extend baseLanguage	Specify breakpointable concepts in `DebugInfoInitializer` for highLevelLanguage.	Fully implement `DebugInfoInitializer` for lowLevelLanguage. Specify breakpointable concepts in `DebugInfoInitializer` for highLevelLanguage.

See section About `DebugInfoInitializer` of this document for further information.

Startup of a run configuration under java debugger

MPS provides a special language for creating run configurations for languages generated into java – jetbrains.mps.baseLanguage.runConfigurations. Those run configurations are able to start under debugger automatically. See Run configurations for languages generated into java for details.

Custom viewers

> ⚠ Note that in MPS2.0 M1 customViewers language has been significantly improved (see MPS 2.0 documentation).

When one views variables and fields in a variable view, one may want to define one's own way to show certain values. For instance, collections could be shown as a collection of elements rather than as an ordinary object with all its internal structure.

For creating custom viewers MPS has jetbrains.mps.debug.customViewers language.

A language jetbrains.mps.debug.customViewers enables one to write one's own viewers for data of certain form. During a debug session, a raw data from stack comes in special form: as proxies for values in target JVM. Such proxies are reflected in customViewers language with language constructs and types.

A main concept of customViewers language is a custom data viewer. It receives a raw java value (which comes from objects on stack) and returns a list of so-called watchables. A watchable is a pair of a value and its label (a string which cathegorizes a value, i.e. whether a value is a method, a field, an element, a size etc.)

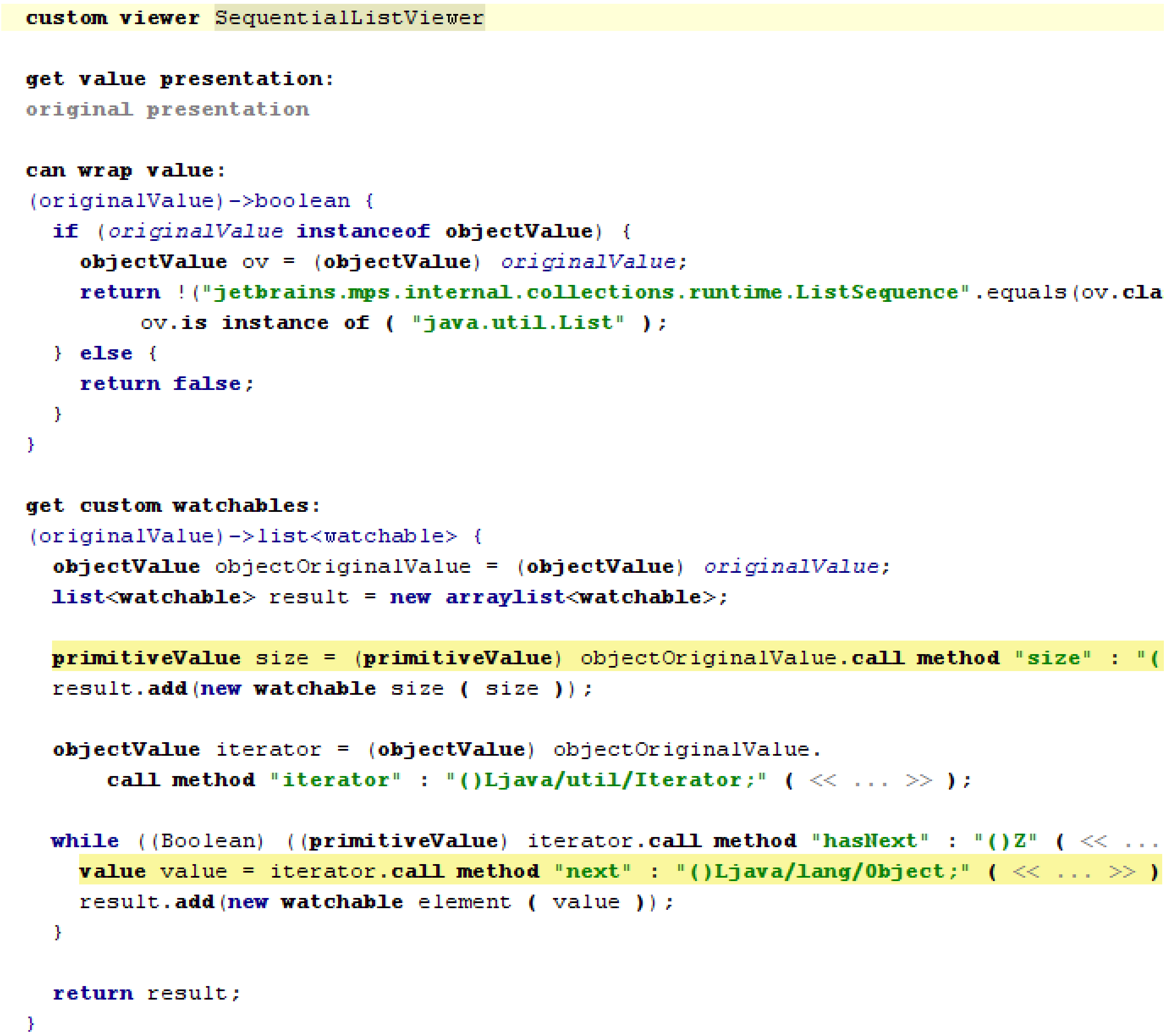
The types introduced in customViewers language are:

- value, its descendants:
- arrayValue,
- primitiveValue,
- objectValue, which in turn has descendant:
- stringValue;
- watchable, a different type.

In the following table those types are described in detail:

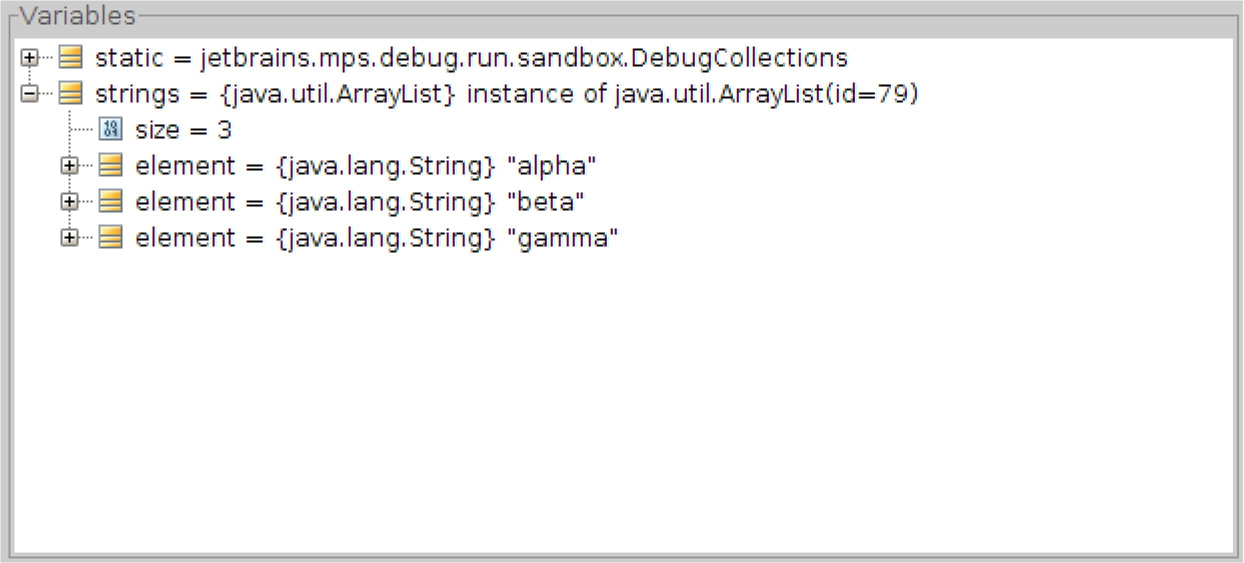
Type name	Operations
arrayValue	• element – returns value by index; • allElements – returns `list<value>`; • elementsRange – returns elements from first index to second index as list<value>; • size – returns size of an array as int.
primitiveValue	javaValue – returns an Object (which in fact is int or long or char etc); this is a java value which is reflected by this primitiveValue.
objectValue	• field – gets a field's name and returns a value of that field (as "value"); • fields – returns all fields as `list<value>`; • call method – takes method's name and its JNI signature and returns the result of method call (as "value"); call method operation also receives method arguments but currently only those of primitive type (because objects can't be just written as java code executed in MPS JVM, but they should be created somehow within target JVM, and currently there's no possibility in customViewers language to do so; however it's planned to implement); • classFQName – returns a string which is an object's class' fq name; • is instance of – takes class fq name and returns whether object is instance of that class or not.
stringValue	• all the operations of objectValue; • javaStringValue which returns a string which is equal to the string reflected by this stringValue.

This is the custom viewer specification for `java.util.List` class:



```
custom viewer SequentialListViewer

get value presentation:
original presentation

can wrap value:
(originalValue)->boolean {
  if (originalValue instanceof objectValue) {
    objectValue ov = (objectValue) originalValue;
    return !("jetbrains.mps.internal.collections.runtime.ListSequence".equals(ov.cla
         ov.is instance of ( "java.util.List" );
  } else {
    return false;
  }
}

get custom watchables:
(originalValue)->list<watchable> {
  objectValue objectOriginalValue = (objectValue) originalValue;
  list<watchable> result = new arraylist<watchable>;

  primitiveValue size = (primitiveValue) objectOriginalValue.call method "size" : "(
  result.add(new watchable size ( size ));

  objectValue iterator = (objectValue) objectOriginalValue.
      call method "iterator" : "()Ljava/util/Iterator;" ( << ... >> );

  while ((Boolean) ((primitiveValue) iterator.call method "hasNext" : "()Z" ( << ...
    value value = iterator.call method "next" : "()Ljava/lang/Object;" ( << ... >> )
    result.add(new watchable element ( value ));
  }

  return result;
}
```

And here we see how a list is displayed in debugger view:



```
Variables
⊞ static = jetbrains.mps.debug.run.sandbox.DebugCollections
⊟ strings = {java.util.ArrayList} instance of java.util.ArrayList(id=79)
    size = 3
  ⊞ element = {java.lang.String} "alpha"
  ⊞ element = {java.lang.String} "beta"
  ⊞ element = {java.lang.String} "gamma"
```

Creating a non-java debugger

You can create a non-java debugger using the API provided by MPS. You can see how it is done in jetbrains.mps.samples.nanoc language – a toy language generated into C. This language is supplied with MPS among other sample languages. The project location is `%HOME_PATH%/MPSSamples.1.5/nanoc/nanocProject/nanocProject.mpr`.

About **`DebugInfoInitializer`**

`DebugInfoInitializer` concept surves two purposes:

- spesify which nodes require to save some additional information in `.debug` file for (like information about positions text, generated from the node, visible variables, name of the file the node was generated into etc.);
- specify how to create a breakpoint on a node.

> In MPS2.0 this is done in two different concepts.

`.debug` files

> In MPS2.0 .debug files were renamed to trace.info files.

`.debug` files contain information allowing to connect nodes in MPS with generated text. For example, if a breakpoint is hit, java debugger tells MPS the line number in source file and to get the actual node from this information MPS uses information from `.debug` files.

.debug files contain the following information:

- position information: name of text file and position in it where the node was generated;
- scope information: for each "scope" node (such that has some variables, associated with it and visible in the scope of the node) – names and ids of variables visible in the scope;
- unit information: for each "unit node" (such that represent some unit of a language, for example a class in java) – name of the unit the node is generated into.

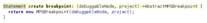
How to write **`DebugInfoInitializer`**

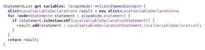
> In MPS2.0 concepts `TraceableConcept`, `ScopeConcept` and `UnitConcept` of textGen language are used for that purpose. See MPS 2.0 documentation.

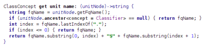
To write a `DebugInfoInitializer` for your language create an instance of `DebugInfoInitializer` concept in the language's plugin.

In the following table sections of `DebugInfoInitializer` are described:

Section	Description	Example

concepts to add debug info	Concepts for which location in text is saved and for which breakpoints could be created.	
		`Statement create breakpoint: (debuggableNode, project)->AbstractMPSBreakpoint {`
		`  return new MPSBreakpoint(debuggableNode, project);`
		`}`

scope concepts	Concepts which have some local variables, visible in the scope.	
		`StatementList get variables: (scopeNode)->nlist<INamedConcept> {`
		`  nlist<LocalVariableDeclaration> result = new nlist<LocalVariableDeclaration>;`
		`  for (node<Statement> statement : scopeNode.statement) {`
		`    if (statement.isInstanceOf(LocalVariableDeclarationStatement)) {`
		`      result.add(statement : LocalVariableDeclarationStatement.localVariableDeclaration);`
		`    }`
		`  }`
		`  return result;`
		`}`

unit concepts	Concepts which are generated into separate units, like classes or inner classes in java.	
		`ClassConcept get unit name: (unitNode)->string {`
		`  string fqName = unitNode.getFqName();`
		`  if (unitNode.ancestor<concept = Classifier> == null) { return fqName; }`
		`  int index = fqName.lastIndexOf(".");`
		`  if (index <= 0) { return fqName; }`
		`  return fqName.substring(0, index) + "$" + fqName.substring(index + 1);`
		`}`

Previous Next