Debugger

Debugger

MPS provides an API for creating custom debuggers as well as integrating with debugger for java. See Debugger Usage page 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

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 want to have your highLevelLanguage integrated with java debugger. See the following explanatory table:

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

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

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.

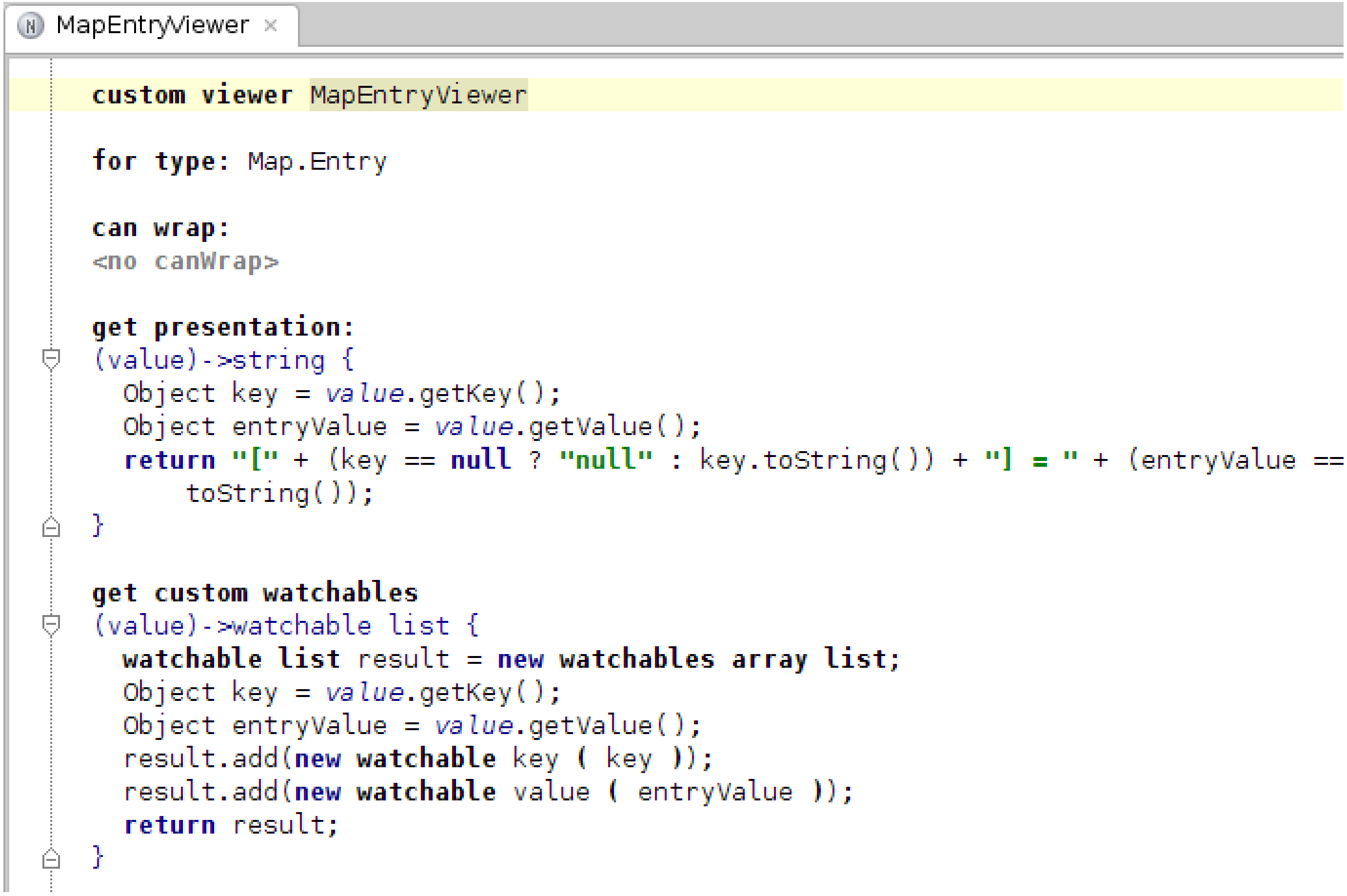
A main concept of customViewers language is a custom data viewer. It receives a raw java value (an objects on stack) and returns a list of so-called watchables. A watchable is a pair of a value and its label (a string which cathegorizes a value, i.e. whether a value is a method, a field, an element, a size etc.) Labels for watchables are defined in `custom watchables container`. Each label could be assigned an icon.

The viewer for a specific type is defined in a `custom viewer` root. In the following table `custom viewer` parts are described:

Part	Description
for type	A type for which this viewer is intended.
can wrap	An additional filter for viewed objects.
get presentation	A string representation of an object.
get custom watchables	Subvalues of this object. Result of this funtion must be of type `watchable list`.

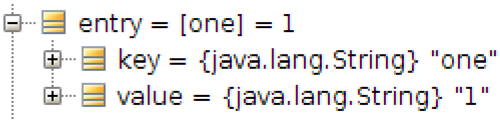
Custom Viewers language introduces two new types: `watchable list` and `watchable`.

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



```
custom viewer MapEntryViewer

for type: Map.Entry

can wrap:
<no canWrap>

get presentation:
(value)->string {
  Object key = value.getKey();
  Object entryValue = value.getValue();
  return "[" + (key == null ? "null" : key.toString()) + "] = " + (entryValue ==
      toString());
}

get custom watchables
(value)->watchable list {
  watchable list result = new watchables array list;
  Object key = value.getKey();
  Object entryValue = value.getValue();
  result.add(new watchable key ( key ));
  result.add(new watchable value ( entryValue ));
  return result;
}
```

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



```
entry = [one] = 1
  key = {java.lang.String} "one"
  value = {java.lang.String} "1"
```

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.2.0/nanoc/nanocProject/nanocProject.mpr`.

Traceable Nodes

To upgrade from MPS 1.5 use migration script "Upgrade Trace Info Generation" (Tools->Scripts->By Language->jetbrains.mps.lang.plugin->Upgrade Trace Info Generation) and regenerate textGen and plugin model of your language

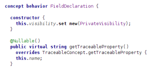
This section describes how to spesify which nodes require to save some additional information in `trace.info` file (like information about positions text, generated from the node, visible variables, name of the file the node was generated into etc.). `trace.info` files contain information allowing to 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 `trace.info` files.

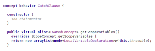
Specifically, `trace.info` files contain the following information:

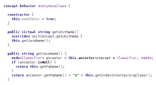
- position information: name of text file and position in it where the node was generated;
- scope information: for each "scope" node (such that has some variables, associated with it and visible in the scope of the node) – names and ids of variables visible in the scope;
- unit information: for each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

Concepts `TraceableConcept`, `ScopeConcept` and `UnitConcept` of language `jetbrains.mps.lang.traceable` are used for that purpose. To save some information into `trace.info` file, user should derive from one of those concepts and implement the specific behavior method. The concepts are described in the table below.

Concept	Description	Behavior method to implement	Example

`TraceableConcept`	Concepts for which location in text is saved and for which breakpoints could be created.	`getTraceableProperty` – some property to be saved into `trace.info` file.	
			`concept behavior FieldDeclaration {`
			`  constructor {`
			`    this.visibility.set new(PrivateVisibility);`
			`  }`
			`  @Nullable()`
			`  public virtual string getTraceableProperty()`
			`    overrides TraceableConcept.getTraceableProperty {`
			`    this.name;`
			`  }`

`ScopeConcept`	Concepts which have some local variables, visible in the scope.	`getScopeVariables` – variable declarations in the scope.	
			`concept behavior CatchClause {`
			`  constructor {`
			`    <no statements>`
			`  }`
			`  public virtual nlist<INamedConcept> getScopeVariables()`
			`    overrides ScopeConcept.getScopeVariables {`
			`    return new arraylist<node<LocalVariableDeclaration>>{this.throwable};`
			`  }`

`UnitConcept`	Concepts which are generated into separate units, like classes or inner classes in java.	`getUnitName` – name of the generated unit.	
			`concept behavior AnonymousClass {`
			`  constructor {`
			`    this.nonStatic = true;`
			`  }`
			`  public virtual string getUnitName()`
			`    overrides UnitConcept.getUnitName {`
			`    this.getJavaName();`
			`  }`
			`  public string getJavaName() {`
			`    node<Classifier> ancestor = this.ancestor<concept = Classifier, root>;`
			`    if (ancestor.isNull) {`
			`      return this.getFqName();`
			`    }`
			`    return ancestor.getFqName() + "$" + this.getIndexInContainingClass();`
			`  }`

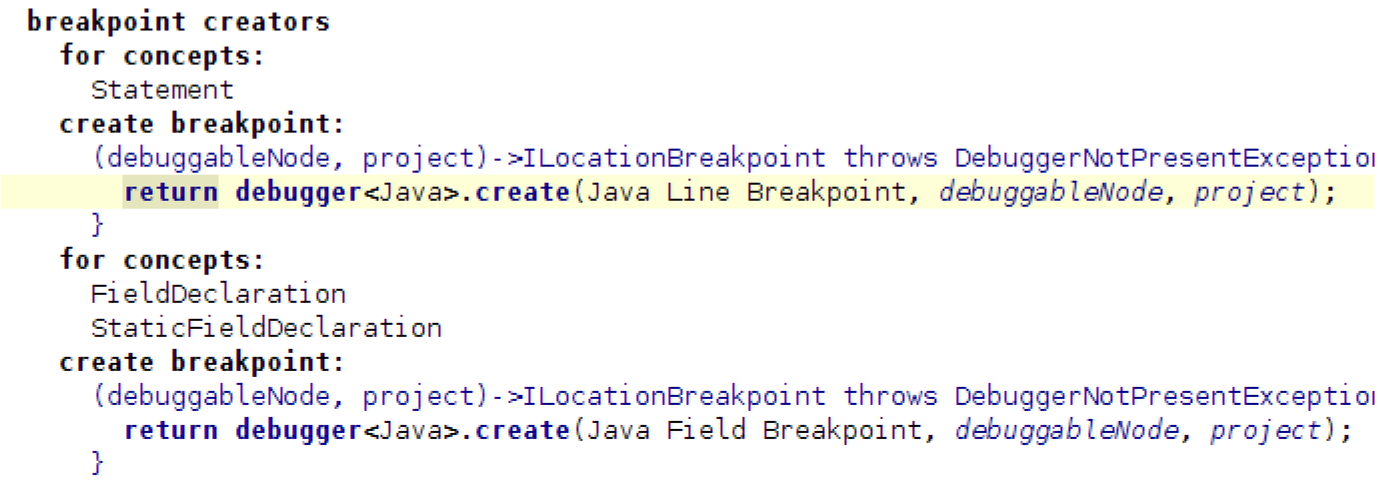
`trace.info` files are created on the last stage of generation – while generating text. So the decsribed concepts are only to be used in languages generated into text.

Breakpoint Creators

To specify how breakpoints are created on various nodes, root `breakpoint creators` is used. This is a root of concept `BreakpointCreator` from `jetbrains.mps.debug.apiLang` language. The root should be located in the language plugin model. It contains a list of `BreakpointableNodeItem`, each of them specify a list of concept to create breakpoint for and a method actually creating a breakpoint. `jetbrains.mps.debug.apiLanguage` provides several concepts to operate with debuggers, and specifially to create breakpoints. They are described below.

- DebuggerReference – a reference to a specific debugger, like java debugger;
- CreateBreakpointOperation – an operation which creates a location breakpoint of specified kind on a given node for a given project;
- DebuggerType – a special type for references to debuggers.

On the following example `breakpoint creators` node from `baseLanguage` is shown.



```
breakpoint creators
  for concepts:
    Statement
  create breakpoint:
    (debuggableNode, project)->ILocationBreakpoint throws DebuggerNotPresentExceptio
      return debugger<Java>.create(Java Line Breakpoint, debuggableNode, project);
    }
  for concepts:
    FieldDeclaration
    StaticFieldDeclaration
  create breakpoint:
    (debuggableNode, project)->ILocationBreakpoint throws DebuggerNotPresentExceptio
      return debugger<Java>.create(Java Field Breakpoint, debuggableNode, project);
    }
```

Previous Next