Using MPS Debugger

Using MPS Debugger

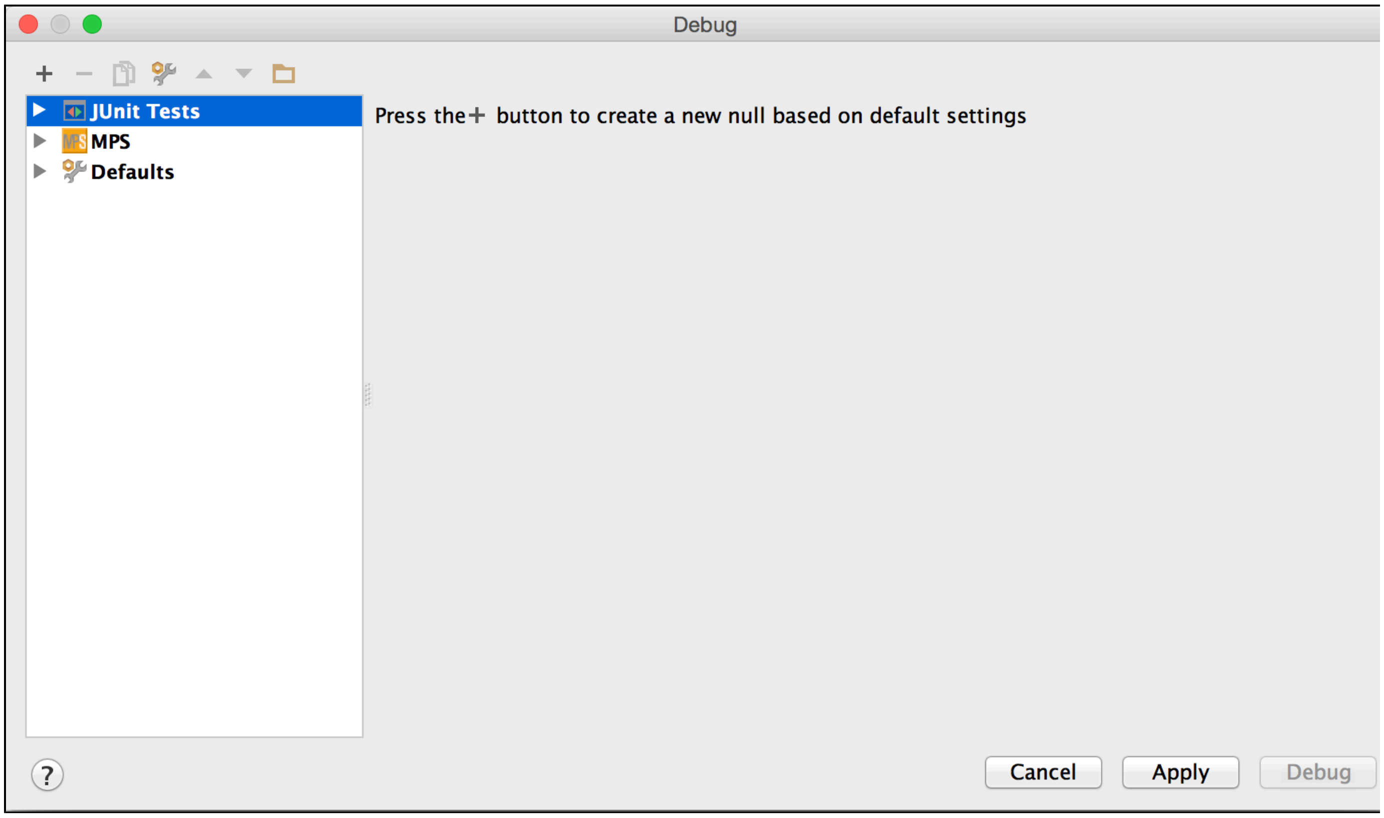
MPS Debugger provides an API to create debuggers for custom languages. Java Debugger plugin, included into MPS distribution, allows user to debug programs which are written in languages which are finally generated into Base Language/Java. We use this plugin below to illustrate MPS Debugger features, which all are available to other lagnuages via API.

Execution

We start with description of how to debug a java application. If a user has a class with main method, a Java Application run configuration should be used to run/debug such a program.

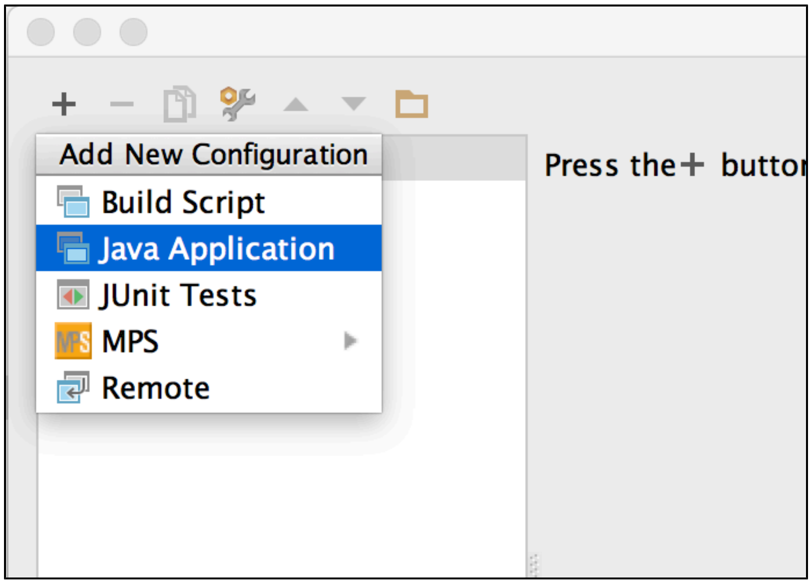
Creating an instance of run configuration

A Java Application or an MPS instance run configurations can be created for a class with a main method or an MPS project, respectively. Go to Run -> Edit Configurations menu and press a button with "+" as shown at the picture below:

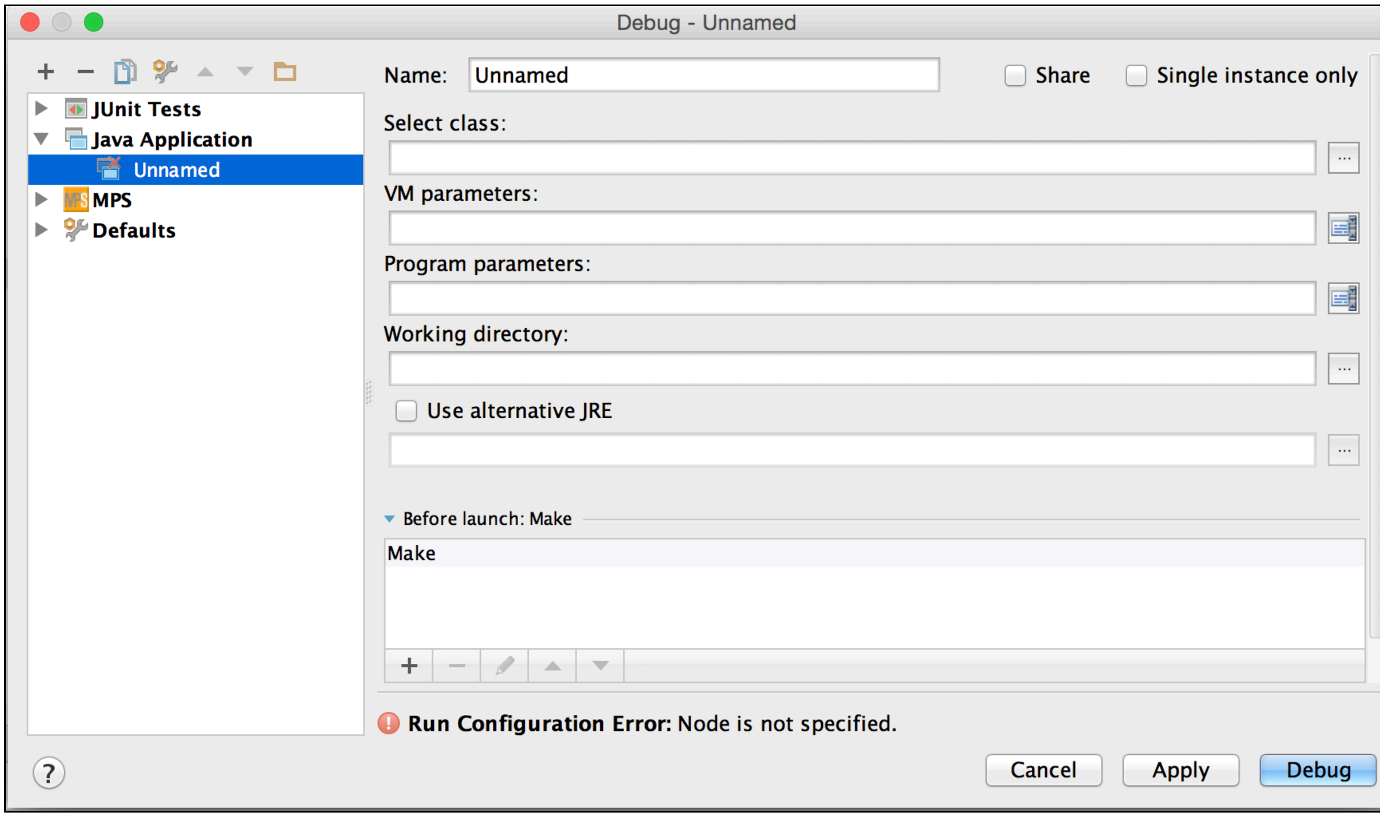


Press the + button to create a new null based on default settings

A menu appears, choose Java Application from it and a new Java Application configuration will be created:



If you select Java Application, you will be able to specify the Java class to run, plus a few optional configuration parameters:



Name: Unnamed

Select class:

VM parameters:

Program parameters:

Working directory:

Use alternative JRE

Make

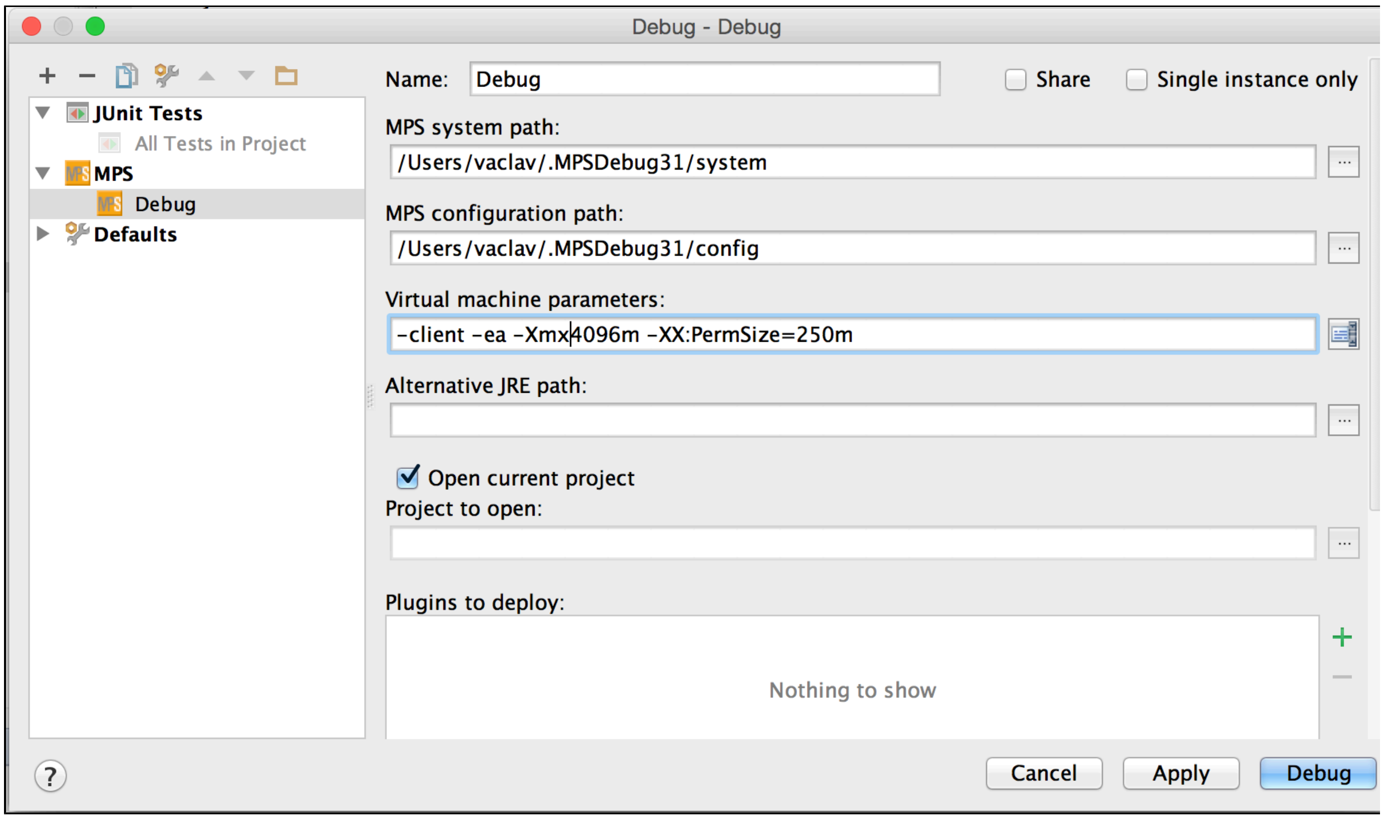
Run Configuration Error: Node is not specified.

A name should be given to each run configuration, and a main node i.e. a class with a main method should be specified. Also VM and program parameters may be specified in a configuration. Look at Run Configuration chapter to learn more about run configurations.

Debugging language definitions

Select MPS instance, if you want to debug MPS language definition code. MPS will start a new instance of MPS with a project that uses your language (it could also be the current project) and you will set breakpoints and debug in your original MPS instance.

In the Debug configuration dialog you need to indicate, which MPS project to open in the new MPS instance - either the current one by checking the Open current project check-box, or any project you specify in the field below. You could also leave both empty and create/open a project from he menu once the new MPS instance starts.


Debug - Debug

- JUnit Tests
  - All Tests in Project
- MPS
  - Debug
- Defaults

Name: Debug

Share [ ] Single instance only [ ]

MPS system path: /Users/vaclav/.MPSDebug31/system

MPS configuration path: /Users/vaclav/.MPSDebug31/config

Virtual machine parameters: -client -ea -Xmx4096m -XX:PermSize=250m

Alternative JRE path:

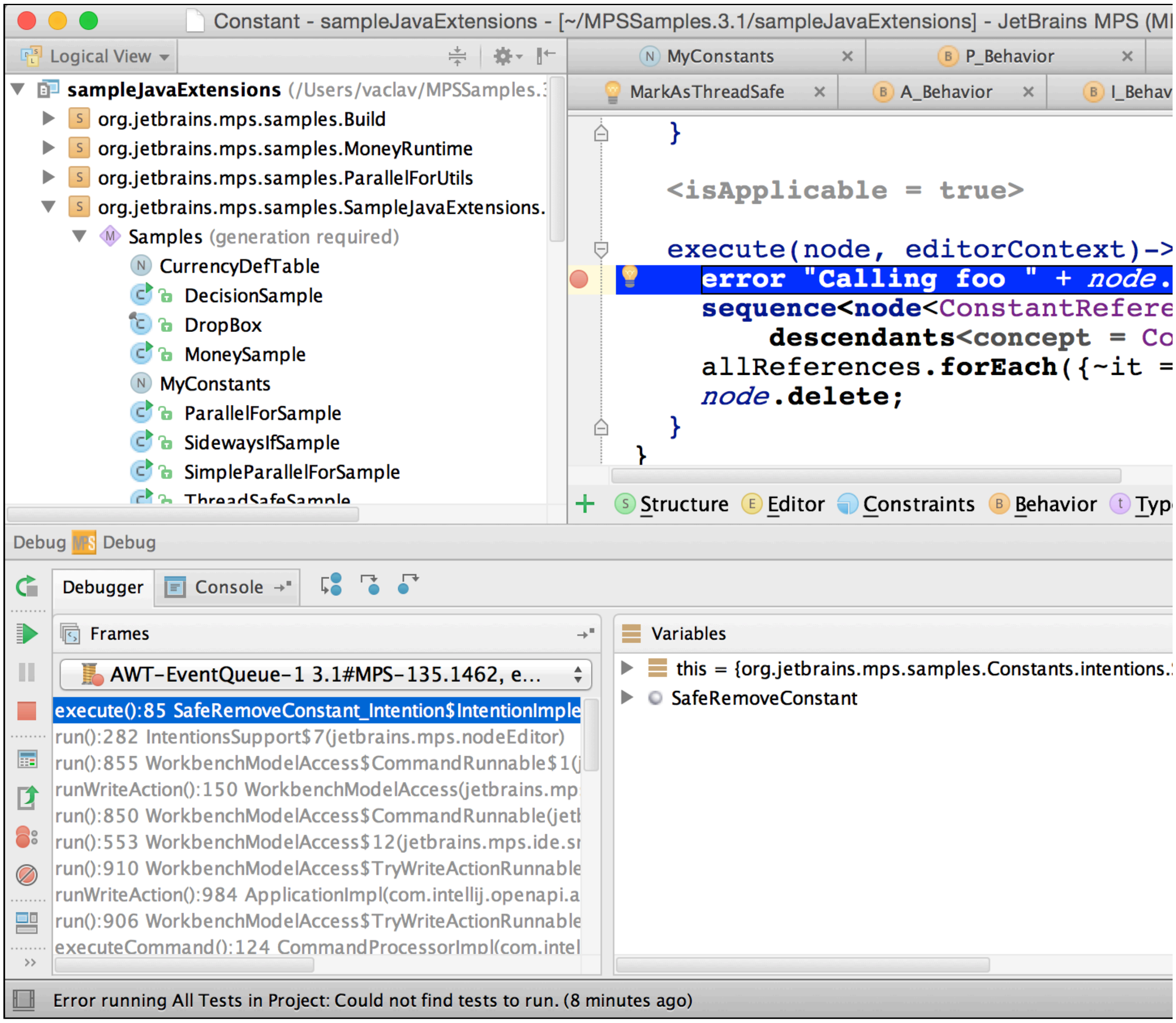
[✓] Open current project

Project to open:

Plugins to deploy: Nothing to show

Debugging a configuration

To debug a run configuration, select it from configurations menu and then press the Debug button. The debugger starts, and the Debugger tool window appears below.


Constant - sampleJavaExtensions - [~/MPSSamples.3.1/sampleJavaExtensions] - JetBrains MPS (MI

sampleJavaExtensions (/Users/vaclav/MPSSamples.3
- org.jetbrains.mps.samples.Build
- org.jetbrains.mps.samples.MoneyRuntime
- org.jetbrains.mps.samples.ParallelForUtils
- org.jetbrains.mps.samples.SampleJavaExtensions.
  - Samples (generation required)
    - CurrencyDefTable
    - DecisionSample
    - DropBox
    - MoneySample
    - MyConstants
    - ParallelForSample
    - SidewaysIfSample
    - SimpleParallelForSample
    - ThreadSafeSample

```
    }

    <isApplicable = true>

    execute(node, editorContext)->
      error "Calling foo " + node.
      sequence<node<ConstantRefere
          descendants<concept = Co
      allReferences.forEach({~it =
      node.delete;
    }
  }
```

AWT-EventQueue-1 3.1#MPS-135.1462, e...

```
execute():85 SafeRemoveConstant_Intention$IntentionImple
run():282 IntentionsSupport$7(jetbrains.mps.nodeEditor)
run():855 WorkbenchModelAccess$CommandRunnable$1(j
runWriteAction():150 WorkbenchModelAccess(jetbrains.mp
run():850 WorkbenchModelAccess$CommandRunnable(jetl
run():553 WorkbenchModelAccess$12(jetbrains.mps.ide.sı
run():910 WorkbenchModelAccess$TryWriteActionRunnable
runWriteAction():984 ApplicationImpl(com.intellij.openapi.a
run():906 WorkbenchModelAccess$TryWriteActionRunnable
executeCommand():124 CommandProcessorImpl(com.intel
```

```
this = {org.jetbrains.mps.samples.Constants.intentions.
SafeRemoveConstant
```

Error running All Tests in Project: Could not find tests to run. (8 minutes ago)

There are two tabs in a tool: one is for the console view and other for the debugger view. In the console an application's output is shown.

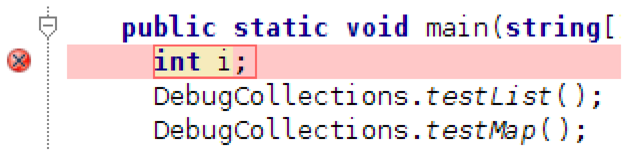
Breakpoints

Next section describes breakpoints usage.

Setting a breakpoint

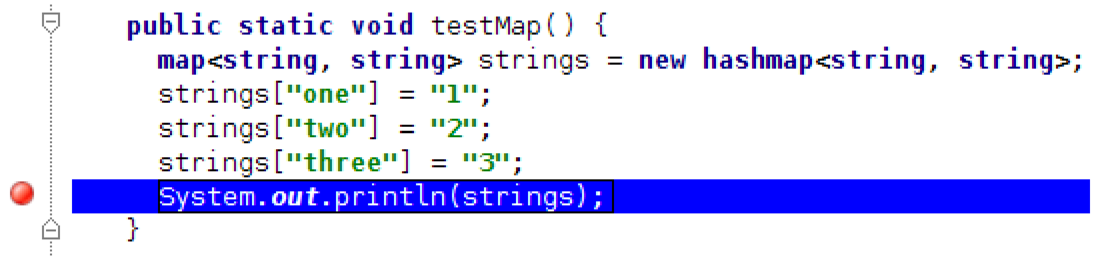
A breakpoint can be set on a statement, field or exception. To set or remove a breakpoint, press Ctrl-F8 on a node in the editor or click on a left margin near a node. A breakpoint is marked with a red bubble on the left margin, a pink line inside the editor and a red frame around a node for the breakpoint. Exception breakpoints are created from the breakpoints dialog.

When the program is stared, breakpoints on which debugger can not stop are specially highlighted.



```
public static void main(string[
  int i;
  DebugCollections.testList();
  DebugCollections.testMap();
```

When debugger stops at a breakpoint, the current breakpoint line is marked blue, and the actual node for the breakpoint is decorated with black frame around it.

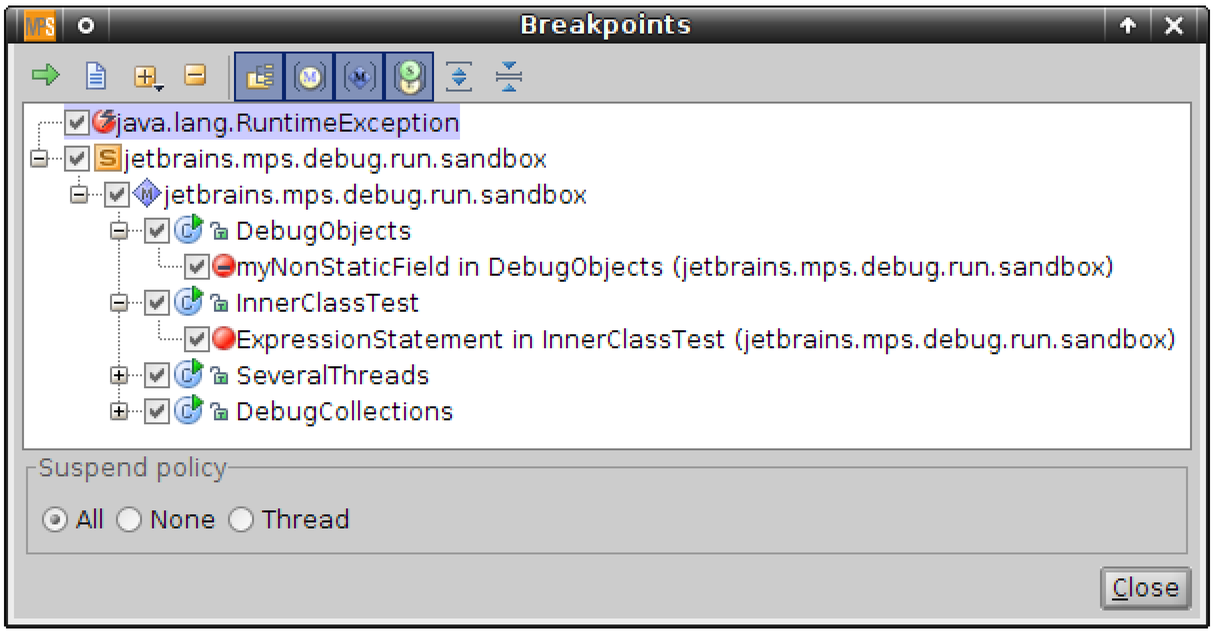


```
public static void testMap() {
  map<string, string> strings = new hashmap<string, string>;
  strings["one"] = "1";
  strings["two"] = "2";
  strings["three"] = "3";
  System.out.println(strings);
}
```

If the cell for a node on which the program is stopped is inside a table, table cell is highlighted instead of a line.

Viewing breakpoints via breakpoints dialog.

All breakpoints set in the project could be viewed via Breakpoints dialog.



```
☑ java.lang.RuntimeException
☑ jetbrains.mps.debug.run.sandbox
  ☑ jetbrains.mps.debug.run.sandbox
    ☑ DebugObjects
      ☑ myNonStaticField in DebugObjects (jetbrains.mps.debug.run.sandbox)
    ☑ InnerClassTest
      ☑ ExpressionStatement in InnerClassTest (jetbrains.mps.debug.run.sandbox)
    ☑ SeveralThreads
    ☑ DebugCollections
```

Suspend policy: [✓ All] None Thread

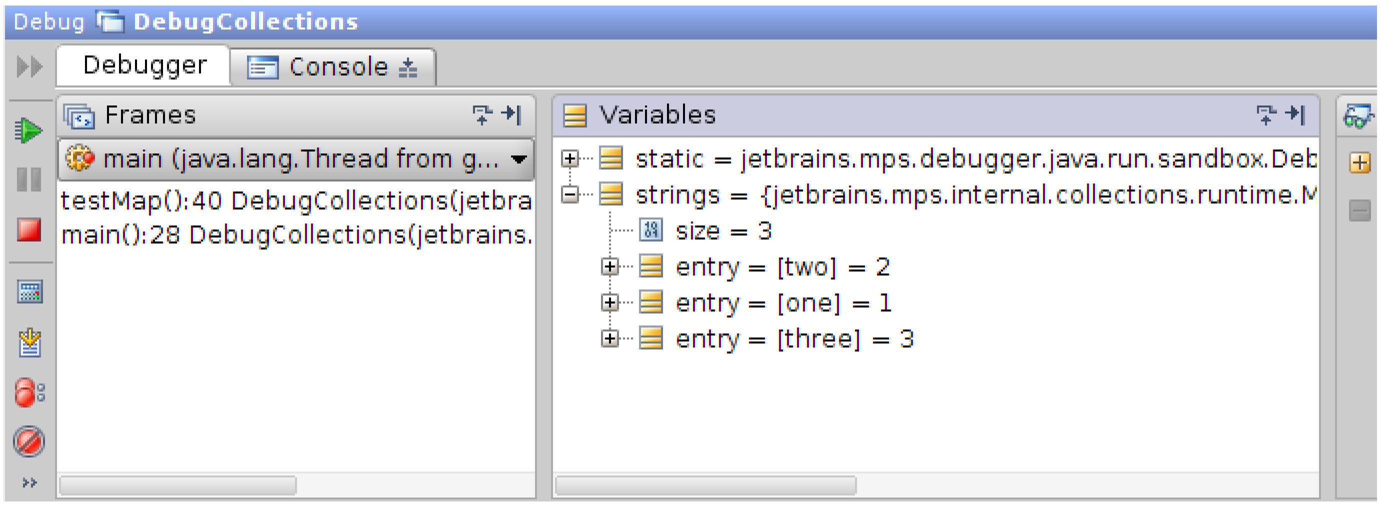
Java breakpoints features include:

- field watchpoints;
- exception breakpoints;
- suspend policy for java breakpoints;
- relevant breakpoint data (like thrown exception or changed field value) is displayed in variables tree.

Examining a state of a program at a breakpoint

When at a breakpoint, a Debugger tab can be used to examine a state of a program. There are three panels available:

- a "Frames" panel with a list of stack frames for a thread, selected using a combo box;
- a "Variable" tree which shows watchables (variables, parameters, fields and static fields) visible in the selected stack frame;
- a "Watches" panel with list of watches and their values.

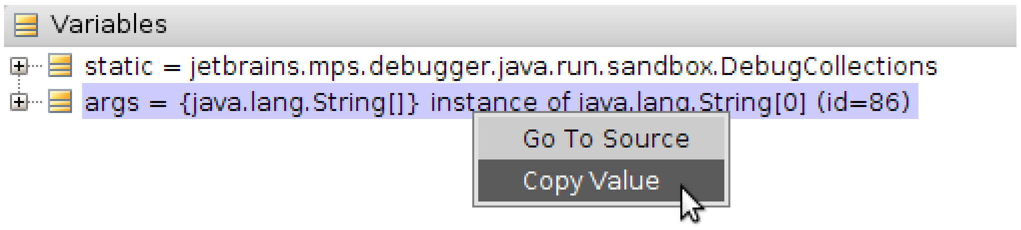


main (java.lang.Thread from g...

```
testMap():40 DebugCollections(jetbra
main():28 DebugCollections(jetbrains.
```

```
static = jetbrains.mps.debugger.java.run.sandbox.Deb
strings = {jetbrains.mps.internal.collections.runtime.M
    size = 3
    entry = [two] = 2
    entry = [one] = 1
    entry = [three] = 3
```

In java debugger "Copy Value" action is available from the context menu of the variable tree.



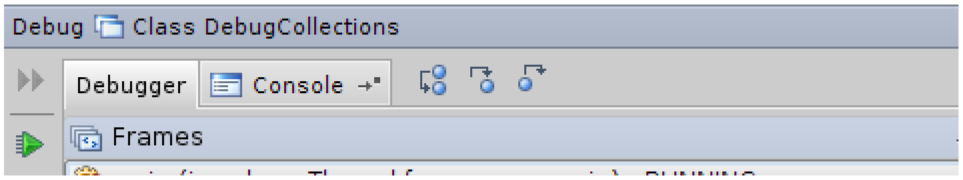
```
static = jetbrains.mps.debugger.java.run.sandbox.DebugCollections
args = {java.lang.String[]} instance of java.lang.String[0] (id=86)
```

Runtime

Controlling execution of a program

- To step over, use Run -> Step Over or F8.
- To step out from a method, use Run -> Step Out or Shift-F8.
- To step into a method call, use Run -> Step Into or F7.
- To resume program execution, use Resume button or Run -> Resume or F9.
- To pause a program manually, use Pause button or Run -> Pause. When paused manually i.e. not at a breakpoint, info about variables is unavailable.

There is a toolbar in Debugger window from where stepping actions are available.

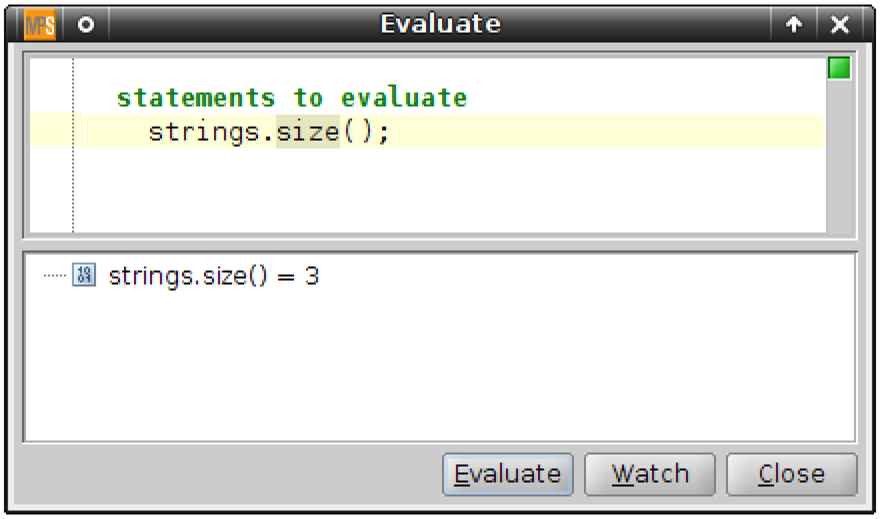


Expressions

Expression evaluation

MPS Java debugger allows user to evaluate expressions during debug, using info from program stack. It is called low-level evaluation, because user is only allowed to use pure java variables/fields/etc from generated code, not entities from high-level source code.

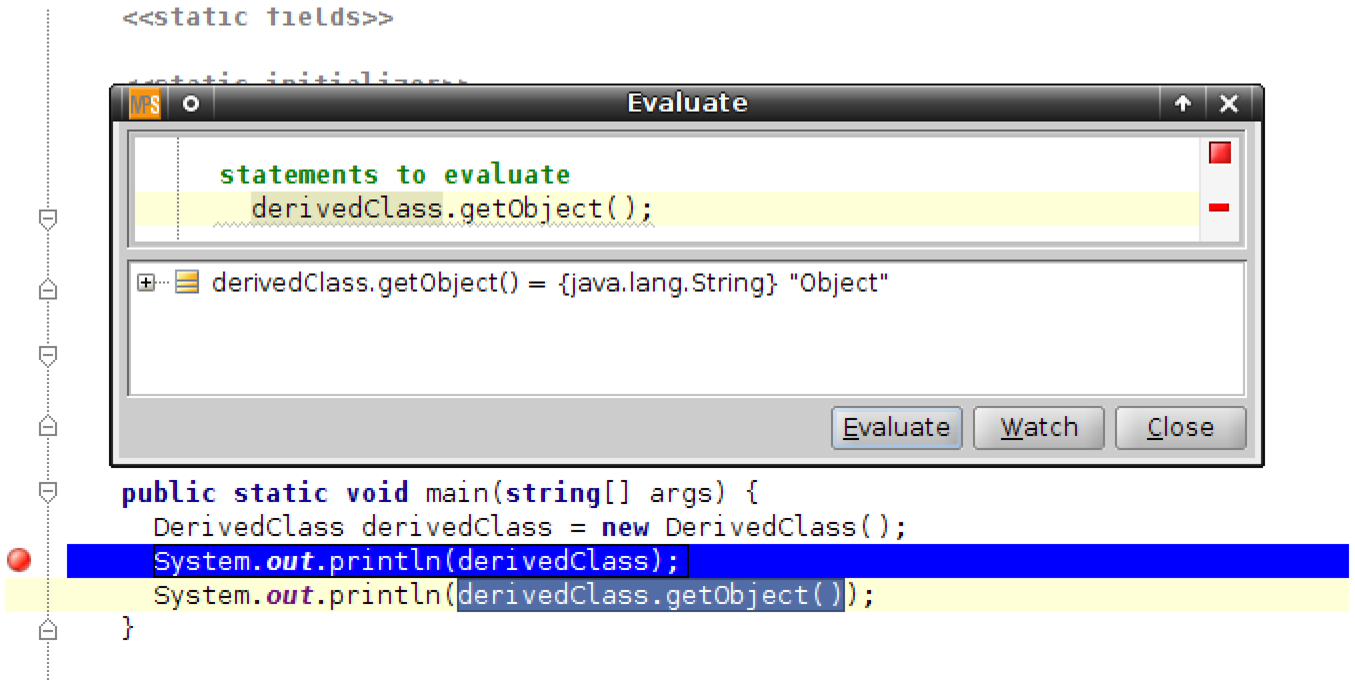
To activate evaluation mode, a program should be stopped at a breakpoint. Press Alt-F8, and a dialog appears.
In a dialog there's a MPS Editor with a statement list inside it. Some code may be written there, which uses variables and fields from stack frame. To evaluate this code, press Evaluate button. The evaluated value will appear in a tree view below.



```
statements to evaluate
  strings.size();
```

```
strings.size() = 3
```

To evaluate a piece of code from the editor, select it and press Alt+F8, and the code will be copied to the evaluation window.



```
statements to evaluate
  derivedClass.getObject();
```

```
derivedClass.getObject() = {java.lang.String} "Object"
```

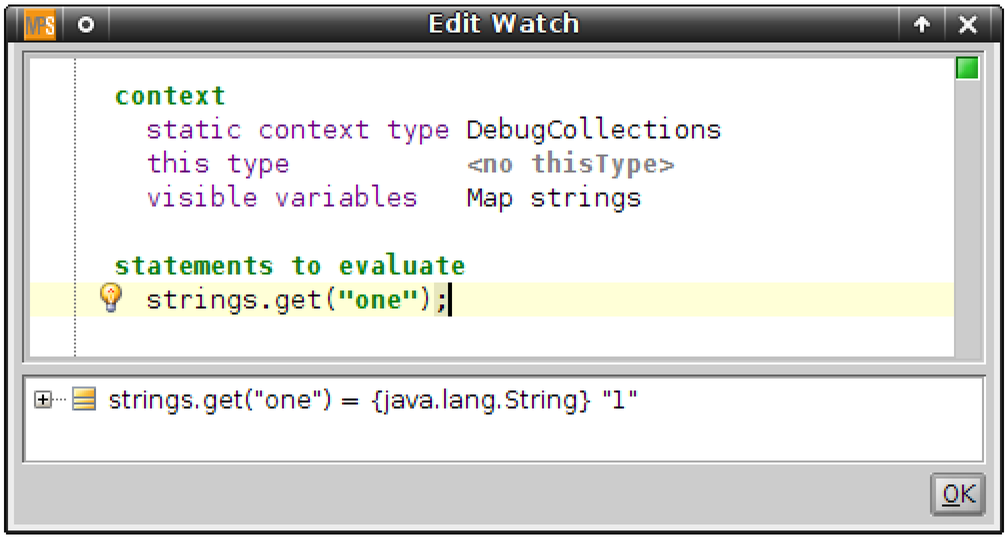
```
<<static fields>>

<<static initializer>>

public static void main(string[] args) {
  DerivedClass derivedClass = new DerivedClass();
  System.out.println(derivedClass);
  System.out.println(derivedClass.getObject());
}
```

Watches

Watches API and low-level watches for java debugger are implemented. "Low-level" means that user can write expressions using variables, available on the stack. To edit a watch, a so-called "context"(used variables, static context type and this type) must be specified. If the stack frame is available at the moment, context is filled automatically.

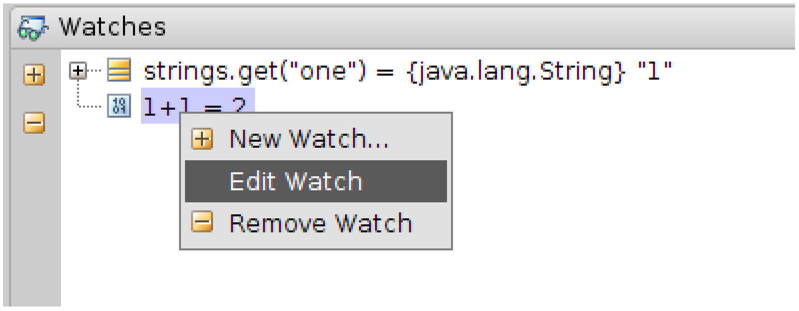


```
context
  static context type DebugCollections
  this type           <no thisType>
  visible variables   Map strings

statements to evaluate
  strings.get("one");
```

```
strings.get("one") = {java.lang.String} "1"
```

Watches can be viewed in "Watches" tree in "Debug" tool window. Watches could be created, edited and removed via context menu or toolbar buttons.



```
strings.get("one") = {java.lang.String} "1"
1+1 = 2
```