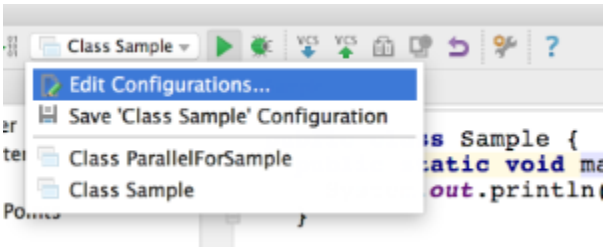
Run Configurations

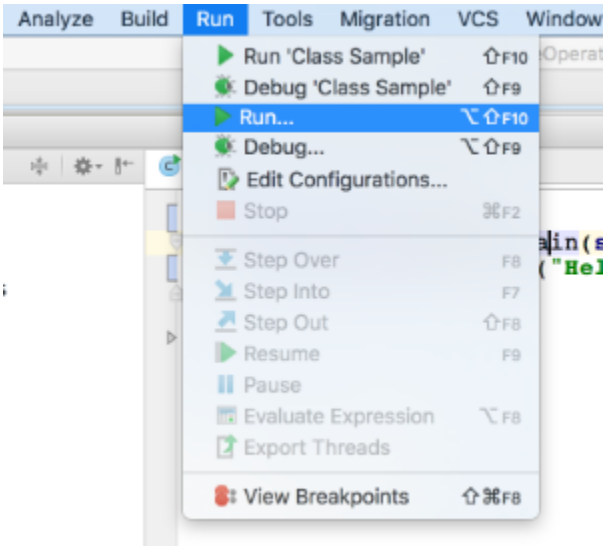
Introduction

Run configurations allow users to define how to execute programs written in their language.

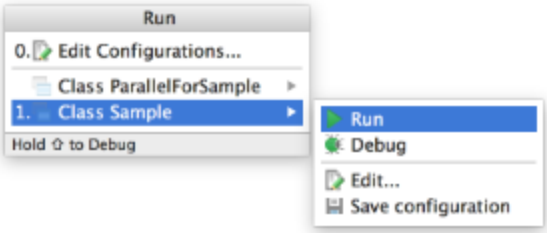
An existing run configuration can be executed either from run configurations box, located on the main toolbar,



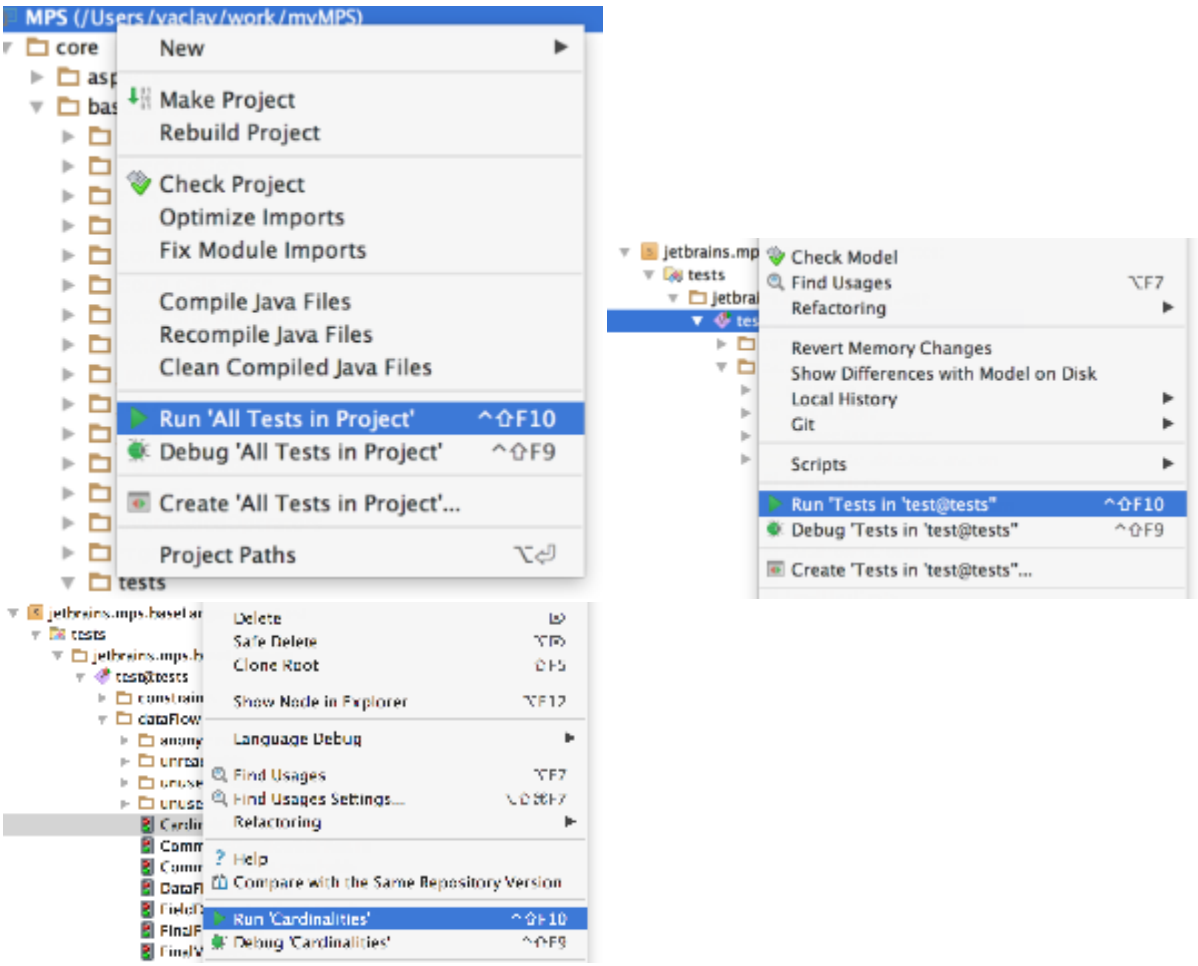
by the "Run" menu item in the main menu



or through the run/debug popup (Alt+Shift+F10/Alt+Shift+F9).



Also run configurations could be executed/created for nodes, models, modules and projects. For example, the JUnit run configuration could run all tests in a selected project, module or model. See Producers on how to implement such behavior for your own run configurations.



To summarize, run configurations define the following things:

- In the creation stage:
  - the configurations name, caption, icon;
  - the configurations kind;
  - how to create a configuration from node(s), model(s), module(s), project.
- In the configuration stage:
  - persistent parameters;
  - an editor for persistent parameters;
  - a checker of persistent parameters validity.
- In the execution stage:
  - the process, which is actually executed;
  - a console with all its tabs, action buttons and actual console window;
  - the things required for debugging this configuration (if it is possible).

The following languages have been introduced to support run configurations in MPS.

- `jetbrains.mps.execution.common` (common language) – contains concepts utilized by the other execution*

languages;
- `jetbrains.mps.execution.settings` (settings language) – a language for defining different setting editors;
- `jetbrains.mps.execution.commands` (command languages) – processes invocations from java;
- `jetbrains.mps.execution.configurations` (configurations language) – the run configurations definition;

Settings

The Settings language allows to create setting editors and integrate them into one another. What we need from a settings editor is the following:

- the fields to edit;
- validation of fields' correctness;
- an editor UI component;
- apply/reset functions to apply settings from the UI component and to reset settings in the UI component to the saved state;
- a dispose function to destroy the UI component when it is no longer needed.

As you can see, settings have UI components. Usually, one UI component is created for multiple instances of settings. In the settings language settings are usually called "configurations" and their UI components are called "editors".

The main concept of settings language is `PersistentConfigurationTemplate`. It has the following sections:

- persistent properties - This section describes the actual settings we are editing. Since we want also to persist these settings (i.e. to write to xml/read from xml) and to clone our configurations, there is a restriction on their type: each property should be either `Cloneable` or `String` or any primitive type. There is also a special kind of property named `template persistent property`, but they are going to be discussed later.
- editor - This section describes the editor of the configuration. It holds the following functions: `create`, `apply to`, `reset from`, `dispose`. A section can also define fields to store some objects of the editor. A `create` function should return a swing component – the main UI component of the editor. `apply to`/`reset from` functions apply or reset settings in the editor to/from configuration given as a parameter. `dispose` function disposes the editor.
- check - In this section persistent properties are checked for correctness. If some properties are not valid, a `report error` statement can be used. Essentially, this statement throws `RuntimeConfigurationException`.
- additional methods - This section is for methods, used in the configurations. Essentially, these methods are configuration instance methods.

Persistent properties

It was mentioned above that persistent properties could be either `Cloneable` or `String` or any primitive type. But if one uses the Settings language inside run configurations, those properties should also support xml persistence. Strings and primitives are persisted as usual. For objects the persistence is more complicated. Two types of properties are persisted for an object: public instance fields and properties with setXXX and getXXX methods. So, if one wish to use some complex type for a persistent property, he should either make all important fields public or provide setXXX and getXXX methods for whatever he wants to persist.

Integrating configurations into one another

One of the two basic features of the Settings language is easy integration of one configuration into another. For that `template persistent properties` are used.

Template parameters

The second basic feature of the Settings language is template parameters. These somewhat resemble constructor parameters in java. For example, if one creates a configuration for choosing a node, he may want to parametrize the configuration with nodes concept. The concept is not a persistent parameter in this case: it is not chosen by the user. This is a parameter specified at configuration creation.

Commands

The Commands language allows you to start up processes from the code in a way it is done from a command line. The main concept of the language is `CommandDeclaration`. In the declaration, command parameters and the way to start process with this parameters are specified. Also, commands can have debugger parameters and some utility methods.

Execute command sections

Each command can have several `execute` sections. Each of this sections defines several execution parameters. There are parameters of two types: required and optional. Optional parameters can have default values and could be ommited when the

command is started, while required cannot have default values and they are mandatory. Any two execute sections of the command should have different (by types) lists of required parameters. One execute section can invoke another execute section. Each execute section should return either values of `process` or `ProcessHandler` types.

`ProcessBuilderExpression`

To start a process from a command execute section `ProcessBuilderExpression` is used. It is a simple list of command parts. Each part is either `ProcessBuilderPart`, which consists of an expression of type `string` or `list<string>`, or a `ProcessBuilderKeyPart`, which represents a parameter with a key (like "-classpath /path/to/classes"). When the code generated from `ProcessBuilderExpression` is invoked, each part is tested for being null or empty and ommited if so. Then, each part is split into multiple parts by spaces. So if you intent to provide a command part with a space in it and do not wish it to be split (for example, you have a file path with spaces), you have to put it into double quotes ("). The working directory of created process could be specified in the Inspector.

Debugger integration

To integrate a command with the debugger, two things are required to be specified:

- the specific debugger to integrate with;
- the command line arguments for a process.

To specify a debugger you can use `DebuggerReference` – an expression of `debugger` type in `jetbrains.mps.debug.apiLang` – to reference a specific debugger. Debugger settings must be an object of type `IDebuggerSettings`.

Configurations

The Configurations language allows to create run configurations. To create a run configuration, on should create an instance of `RunConfiguration` (essentially, configuration from the settings language) and provide a `RunConfigurationExecutor` for it. One also may need a `RunConfigurationKind` to specify a kind of this configuration, `RunConfigurationProducer` to provide a way of creating this configuration from nodes, models, modules etc and a `BeforeTask` to specify, how to prepare the configuration before execution.

Executors

Executor is a node, which describes how a process is started for this run configuration. It takes the settings that the user entered and creates a process from it. So, the executor's execute methods should return an instance of type `process`. This is done via `StartProcessHandlerStatement`. Anything that has a type `process` or `ProcessHandler` could be passed to it. A `process` could be created in three different ways:

1. via command;
2. via `ProcessBuilderExpression` (recommended to use in commands only);
3. by creating new instance of the ProcessHandler class; this method is recommended only if the above two do not fit you, for example when you are creating a run configuration for remote debugging and you do not really need to start a process.

The executor itself consists of the following sections:

1. "for" section where the configuration this executor is for and an alias for it is specified;
2. "can" section where the ability of run/debug this configuration is specified; if the command is not used in this executor, one must provide an instance of `DebuggerConfiguration` here;
3. "before" section with the call of tasks, which could be executed before this configuration run, such as Make;
4. "execute" section where the process itself is created.

Debugger integration

If a command is used to start a process, nothing should be done apart from specifying a configuration as debuggable (by selecting "debug" in the executor). However, if a custom debugger integration is required, it is done the same way as in the command declaration.

Producers

Producers for a run configuration describe how to create this configuration for various nodes or groups of nodes, models, modules or a project. This makes run configurations easily discoverable for users since for each producer they will see an action in the context menu suggesting to run the selected item. Also this simplifies configuring because it gives a default way to execute something without seeing the editing dialog first.

Each producer specifies one run configuration that it creates. It can have several produce from sections for each kind of source

the configuration can be produced from. This source should be one of the following: node<>, nlist<>, model, module, project. Apart from source, each produce from section has a create section – a concept function parametrized with a source. The function should return either the created run configuration or null if it cannot be created for some reason.

Useful examples

In this section you can find some useful tips and examples of run configurations usages.

Person Editor

In this example an editor for a "Person" is created. This editor edits two properties of a person: name and e-mail address.

```
persistent configuration template PersonEditor <>

  persistent properties:
    string myName;
    string myEmail;

    check()->void {
      if (!(this.myName.?matches("(\\w|\\s)*"))) {
        report error "Name should contain only letterts and spaces.";
      }
      if (!(this.myEmail.?matches("(\\w|\\.)*@(\\w|\\.)*"))) {
        report error "Email should be valid.";
      }
    }

  editor:
    JTextField myNameField;
    JTextField myEmailField;

    create()->JComponent {
      JPanel panel = new JPanel(new GridBagLayout());
      myNameField = new JTextField();
      myEmailField = new JTextField();
      panel.add(new JLabel("Name"), grid bag constraints/label, 0/);
      panel.add(myNameField, grid bag constraints/field, 1/);
      panel.add(new JLabel("Email"), grid bag constraints/label, 2/);
      panel.add(myEmailField, grid bag constraints/field, 3/);
      return panel;
    }
    reset from(configuration)->void {
      myNameField.setText(configuration.myName);
      myEmailField.setText(configuration.myEmail);
    }
    apply to(configuration)->void {
      configuration.myName = myNameField.getText();
      configuration.myEmail = myEmailField.getText();
    }
    <no dispose>

<< ... >>
```

PersonEditor could be used from java code in the following way:

```
public class SettingsEditorDialogTest {
  public static void main(string[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        // create PersonEditor
        template configuration<PersonEditor> personEditor = new PersonEditor();
        // create dialog
        SettingsEditorDialog dialog = new SettingsEditorDialog(personEditor);
        // show dialog
        dialog.setVisible(true);
        // print name and email, specified by user
        System.out.println("name = " + personEditor.myName);
        System.out.println("email = " + personEditor.myEmail);
      }
    });
  }
}
```

Exec command

This is an example of a simple command, which starts a given `executable` with `programParameters` in a given `workingDirectory`.

```
command exec
  debugger: <no debuggerConfiguration>

  execute(File executable (required)                                    )
          File workingDirectory = new File(SystemProperties.getUserHome())
          string programParameters
    {
      return this.protect(executable.getAbsolutePath()) programParameters in workin
    }

  private string protect(string rawString) {
    if (rawString.contains(" ")) { return "\"" + rawString + "\""; }
    return rawString;
  }
```

Compile with gcc before task

This is an example of a `BeforeTask` which performs compilation of a source file with the `gcc` command. It also demonstrates how to use commands outside of run configurations executors.

```
before task Compile with gcc

  node<File> myFile

execute(project)->boolean {
  try {
    // create count down to synchronize
    final CountDownLatch countDown = new CountDownLatch(1);
    // create a process
    command process<gcc> process = gcc(file = myFile) > new ProcessAdapter() {
      @Override
      public void processTerminated(ProcessEvent event) {
        // notify process termination
        countDown.countDown();
      }

      @Override
      public void onTextAvailable(ProcessEvent event, Key key) {
        // check which type of output it is
        if (ProcessOutputTypes.STDERR.equals(key)) {
          // log error output as an error message
          error event.getText();
        } else {
          // log other output as information
          info event.getText();
        }
      }

      <add members (ctrl+space)>
    };
    // start notification (necessary to use when not using StartProcessHandlerStatem
    process.startNotify();
    // wait for process termination
    countDown.await();
    // return whether the result of compilation exists
    return new File(gcc.getExecutableFile(myFile).getAbsolutePath()).exists();
  } catch (ExecutionException e) {
    error "", e;
    return false;
  } catch (InterruptedException e) {
    error "", e;
    return false;
  }
}
```

Note that this is just a toy example, in a real-life scenarios the task should show a progress window while compiling, for example.

Java Executor

This is an actual executor for the Java run configuration from MPS.

```
executor
  for Java as myRunConfiguration
  can:  run , debug

  before:
    Make(new arraylist<node<>>{myRunConfiguration.myNode.getNode()})

  execute(project)->join(process | ProcessHandler) {
    // create a console
    console console = new console(project, false);
    // add a mesage filter to parse stack traces
    console.addMessageFilter(new JavaStackTraceFilter());
    // start java command process with tool console
    start java(node = myRunConfiguration.myNode.getNode(),
        runParameters = myRunConfiguration.myRunParameters.myJavaRunParameters) with
  }
```

Java Producer

This is a producer for Java run configuration from MPS.

```
producer for Java

  produce from node<ClassConcept>
    create(source)->configuration {
      if (source.getMainMethod().isNull) { return null; }
      Java configuration = new Java("Class " + source.name);
      configuration.myNode.setNode(source);
      return configuration;
    }
  produce from node<StaticMethodDeclaration>
    create(source)->configuration {
      if (!(source.isMainMethod())) { return null; }
      node<Classifier> classifier = source.ancestor<concept = Classifier>;
      if (classifier.isNull) { return null; }
      Java configuration = new Java("Class " + classifier.name);
      configuration.myNode.setNode(classifier);
      return configuration;
    }
  produce from node<IMainClass>
    create(source)->configuration {
      if (!(source.isNodeRunnable())) { return null; }
      string name = source.isInstanceOf(INamedConcept) ? source : INamedConcept.name
      Java configuration = new Java("Node " + name);
      configuration.myNode.setNode(source);
      return configuration;
    }
```

You can see here three "produce from" sections. A Java run configuration is created here from nodes of `ClassConcept`, `StaticMethodDeclaration` or `IMainClass`.

Running a node, generated into java class

Lets suppose you have a node of a concept which is generated into a java class with a main method and you wish to run this

node from MPS. Then you do not have to create a run confgiuration in this case, but you should do the following:

1. The concept you wish to run should implement `IMainClass` concept from `jetbrains.mps.execution.util` language. To specify when the node can be executed, override `isNodeRunnable` method.
2. Unit information should be generated for the concept. Unit information is required to correctly determine the class name which is to be executed. You can read more about unit information, as whell as about all trace information, in Debugger section of MPS documentation. To ensure this check that one of the following conditions are satisfied:
    a. a `ClassConcept` from `jetbrains.mps.baseLanguage` is generated from the node;
    b. a node is generated into text (the language uses `textGen` aspect for generation) and the concept of the node implements `UnitConcept` interface from `jetbrains.mps.traceable`;
    c. a node is genearted into a concept for which one of the three specified conditions is satisfied.

Previous Next