IntelliJ IDEA Action System

Added by Dmitry Jemerov, last edited by Nikolay Chashnikov on Mar 03, 2014 (view change)

Executing and Updating Actions

The system of actions allows plugins to add their own items to IDEA menus and toolbars. An action is a class, derived from the **AnAction** class, whose actionPerformed method is called when the menu item or toolbar button is selected. For example, one of the action classes is responsible for the "File | Open File..." menu item and for the "Open File" toolbar button.

Actions are organized into groups, which, in turn, can contain other groups. A group of actions can form a toolbar or a menu. Subgroups of the group can form submenus of the menu.

Every action and action group has an unique identifier. Identifiers of many of the standard IDEA actions are defined in the **IdeActions** class.

Every action can be included in multiple groups, and thus appear in multiple places within the IDEA user interface. Different places where actions can appear are defined by constants in the **ActionPlaces** interface. For every place where the action appears, a new `Presentation` is created. Thus, the same action can have different text or icons when it appears in different places of the user interface. Different presentations for the action are created by copying the presentation returned by the `AnAction.getTemplatePresentation()` method.

To update the state of the action, the method `AnAction.update()` is periodically called by IDEA. The object of type `AnActionEvent` passed to this method carries the information about the current context for the action, and in particular, the specific presentation which needs to be updated.

To retrieve the information about the current state of the IDE, including the active project, the selected file, the selection in the editor and so on, the method `AnActionEvent.getData()` can be used. Different data keys that can be passed to that method are defined in the `DataKeys` class.

The `AnActionEvent` instance is also passed to the `actionPerformed` method.

Registering Actions

There are two main ways to register an action: either by listing it in the <actions> section of the plugin.xml file, or through Java code.

Registering Actions in plugin.xml

Registering actions in plugin.xml is demonstrated in the following example. The example section of plugin.xml demonstrates all elements which can be used in the <actions> section, and describes the meaning of each element.

```
<!-- Actions -->
    <actions>
        <!-- The <action> element defines an action to register.
               The mandatory "id" attribute specifies an unique identifier for the action.
               The mandatory "class" attribute specifies the full-qualified name of the
class implementing the action.
               The mandatory "text" attribute specifies the text of the action (tooltip
for toolbar button or text for menu item).
              The optional "use-shortcut-of" attribute specifies the ID of the action
whose keyboard shortcut this action will use.
              The optional "description" attribute specifies the text which is displayed
in the status bar when the action is focused.
               The optional "icon" attribute specifies the icon which is displayed on the
toolbar button or next to the menu item. -->
        <action id="VssIntegration.GarbageCollection" class="com.foo.impl.CollectGarbage"
text="Collect _Garbage" description="Run garbage collector"
                icon="icons/garbage.png">
              <!-- The <add-to-group> node specifies that the action should be added to an
existing group. An action can be added to several groups.
                    The mandatory "group-id" attribute specifies the ID of the group to
which the action is added.
                    The group must be implemented by an instance of the DefaultActionGroup
```

```
class.
                    The mandatory "anchor" attribute specifies the position of the action
in the group relative to other actions. It can have the values
                    "first", "last", "before" and "after".
                    The "relative-to-action" attribute is mandatory if the anchor is set
to "before" and "after", and specifies the action before or after which
                    the current action is inserted. -->
            <add-to-group group-id="ToolsMenu" relative-to-action="GenerateJavadoc"
anchor="after"/>
             <!-- The <keyboard-shortcut> node specifies the keyboard shortcut for the
action. An action can have several keyboard shortcuts.
                    The mandatory "first-keystroke" attribute specifies the first
keystroke of the action. The key strokes are specified according to the regular Swing
rules.
                    The optional "second-keystroke" attribute specifies the second
keystroke of the action.
                    The mandatory "keymap" attribute specifies the keymap for which the
action is active. IDs of the standard keymaps are defined as
                    constants in the com.intellij.openapi.keymap.KeymapManager class. -->
            <keyboard-shortcut first-keystroke="control alt G" second-keystroke="C"
keymap="$default"/>
             <!-- The <mouse-shortcut> node specifies the mouse shortcut for the action.
An action can have several mouse shortcuts.
                    The mandatory "keystroke" attribute specifies the clicks and modifiers
for the action. It is defined as a sequence of words separated by spaces:
                    "button1", "button2", "button3" for the mouse buttons; "shift",
"control", "meta", "alt", "altGraph" for the modifier keys;
                    "doubleClick" if the action is activated by a double-click of the
button.
                    The mandatory "keymap" attribute specifies the keymap for which the
action is active. IDs of the standard keymaps are defined as
                    constants in the com.intellij.openapi.keymap.KeymapManager class. -->
            <mouse-shortcut keystroke="control button3 doubleClick" keymap="$default"/>
        </action>
        <!-- The <group> element defines an action group. <action>, <group> and
<separator> elements defined within it are automatically included in the group.
                The mandatory "id" attribute specifies an unique identifier for the action.
                The optional "class" attribute specifies the full-qualified name of the
```

```
class implementing the group. If not specified,
               com.intellij.openapi.actionSystem.DefaultActionGroup is used.
               The optional "text" attribute specifies the text of the group (text for the
menu item showing the submenu).
               The optional "description" attribute specifies the text which is displayed
in the status bar when the group is focused.
               The optional "icon" attribute specifies the icon which is displayed on the
toolbar button or next to the group.
               The optional "popup" attribute specifies how the group is presented in the
menu. If a group has popup="true", actions in it
               are placed in a submenu; for popup="false", actions are displayed as a
section of the same menu delimited by separators. -->
        <group class="com.foo.impl.MyActionGroup" id="TestActionGroup" text="Test Group"
description="Group with test actions"
               icon="icons/testgroup.png" popup="true">
            <action id="VssIntegration.TestAction" class="com.foo.impl.TestAction"
text="My Test Action" description="My test action"/>
            <!-- The <separator> element defines a separator between actions. It can also
have an <add-to-group> child element. -->
            <separator/>
            <group id="TestActionSubGroup"/>
             <!-- The <reference> element allows to add an existing action to the group.
The mandatory "ref" attribute specifies the ID of the action to add. -->
            <reference ref="EditorCopy"/>
            <add-to-group group-id="MainMenu" relative-to-action="HelpMenu"
anchor="before"/>
        </group>
    </actions>
```

Registering Actions from Code

To register an action from code, two steps are required. First, an instance of the class derived from `AnAction` must be passed to the `registerAction` method of the ActionManager class, to associate the action with an ID. Second, the action needs to be added to one or more groups. To get an instance of an action group by ID, it is necessary to call `ActionManager.getAction()` and cast the returned value to the DefaultActionGroup class.

You can create a plugin that registers actions on IDEA startup using the following procedure.

To register an action on IDEA startup

1. Create a new class that implements the `ApplicationComponent` interface.
2. In this class, override the `getComponentName`, `initComponent`, and `disposeComponent` methods.
3. Register this class in the <application-components> section of the plugin.xml file.

To clarify the above procedure, consider the following sample Java class `MyPluginRegistration` that registers an action defined in a custom `TextBoxes` class and adds a new menu command to the **Window** menu group on the main menu:

```
public class MyPluginRegistration implements ApplicationComponent {
  // Returns the component name (any unique string value).
    @NotNull public String getComponentName() {
        return "MyPlugin";
    }


// If you register the MyPluginRegistration class in the <application-components> section
of
// the plugin.xml file, this method is called on IDEA start-up.
    public void initComponent() {
        ActionManager am = ActionManager.getInstance();
        TextBoxes action = new TextBoxes();
      // Passes an instance of your custom TextBoxes class to the registerAction method of
the ActionManager class.
        am.registerAction("MyPluginAction", action);
      // Gets an instance of the WindowMenu action group.
        DefaultActionGroup windowM = (DefaultActionGroup) am.getAction("WindowMenu");
      // Adds a separator and a new menu command to the WindowMenu group on the main
menu.
        windowM.addSeparator();
        windowM.add(action);
    }
// Disposes system resources.
    public void disposeComponent() {
    }
}
```

Note, that the sample `TextBoxes` class is described here .

To ensure that your plugin is initialized on IDEA start-up, make the following changes to the <application-components> section of the plugin.xml file:

```
<application-components>
    <!-- Add your application components here -->
      <component>
          <implementation-class>MypackageName.MyPluginRegistration</implementation-class>
      </component>
</application-components>
```

Building UI from Actions

If a plugin needs to include a toolbar or popup menu built from a group of actions in its own user interface, that can be accomplished through the `ActionPopupMenu` and `ActionToolbar` classes. These objects can be created through calls to `ActionManager.createActionPopupMenu` and `ActionManager.createActionToolbar`. To get a Swing component from such an object, simply call the getComponent() method.

If your action toolbar is attached to a specific component (for example, a panel in a toolwindow), you usually need to call `ActionToolbar.setTargetComponent()` and pass the instance of the related component as a parameter. This ensures that the state of the toolbar buttons depends on the state of the related component, and not on the current focus location within the IDE frame.

Labels: migrated

4 Comments

Saravana Thiyagaraj Mar 01, 2014

In the plugin.xml snippet (under Registering Action in plugin.xml) , a **group** with *id="TestActionGroup"* is present inside a **group** with the same id. · (*<group id="TestActionGroup"/>*).

Is this expected?

Nikolay Chashnikov Mar 03, 2014

Fixed, thank you.

Igor Bogomolov Apr 01, 2014

Hi! You have great instruction how to bind an action to menu item or toolbar button.

But is it possible to bind an action to some UI element, folding +, for example?

Or even create an icon in the gutter area and bind an action to it?

Kaustubh Feb 02, 2015

I am trying to add a plugin icon (in android studio) near SDK manager. Does anyone knows how to add it? Or a group Id for the same?