Change Method Signature in ActionScript

In ActionScript, you can use the Change Method Signature refactoring to:

- Change the method name and return type.
- Add new parameters and remove the existing ones. Note that you can also add a parameter using a dedicated Extract Parameter refactoring.
- Reorder parameters.
- Change parameter names and types.
- Propagate new parameters through the method call hierarchy.

On this page:

Example

Before

```
// The function paint() is declared in
// the IShape interface.


public interface IShape {
    function paint(g: Graphics): void;
}

// This function is then called within the
// paint() function of the Canvas class.


public class Canvas {
    private var shapes: Vector.<IShape>;

    public function paint(g: Graphics): void {
        for each (var shape: IShape in shapes) {
            shape.paint(g);
        }
    }
}

// Now we are going to show an example of the
// Change Signature refactoring for the function
// paint() of the IShape interface.
```

After

```
// In this refactoring example we have changed
// and introduced two new parameters. Note that
// a required parameter while the second is opt
// for it is specified in the function definit

public interface IShape {
    function paint(graphics:Graphics, wireframe
}

// When performing this refactoring, the new pa
// the paint() function of the Canvas class. As
// Canvas.paint() has changed. Also note how IS
// Canvas.paint() is called now.

public class Canvas {
    private var shapes: Vector.<IShape>;

    public function paint(g:Graphics, wireframe
        for each (var shape: IShape in shapes)
            shape.paint(g, wireframe);
        }
    }
}

// Other results for this refactoring are poss
// For more information, see the discussion tha
```

Initializer, default value, and propagation of new parameters

For each new parameter added to a function, you can specify:

- A value (or an expression) to be used for initializing the parameter (the **Initializer** field in IntelliJ IDEA).
- A default value (or an expression) (the **Default value** field).

You can also propagate the parameters you have introduced to the functions that call the function whose signature you are changing.

The refactoring result depends on whether or not you specify those values and use the propagation.

Propagation. New parameters can be propagated to any function that call the function whose signature you are changing. In such a case, generally, the signatures of the calling functions change accordingly. These changes, however, also depend on the combination of the initializer and the default value set for the new parameters.

Initializer. The value specified in the **Initializer** field is added to the function definition as the default parameter value. This makes the corresponding parameter an optional parameter. (See the discussion of required and optional parameters in Flex/ActionScript documentation.)

If the default value for the new parameter is not specified (in the **Default value** field), irrespective of whether or not the propagation is used, the function calls and the signatures of the calling functions don't change.

If both, the initializer and the default value are specified, the refactoring result depends on whether or not the propagation is used:

- If the propagation is not used, the initializer value don't affect the function calls and the signatures of the calling functions.
- If the propagation is used, the initializer value is added to the definition of the calling function as the default value for the corresponding parameter (in the same way as in the function whose signature you are changing).

Default value. Generally, this is the value to be added to the function calls.

If the new parameter is not propagated to a calling function, the function calls within such a function will also use this value.

If the propagation is used, this value won't matter for the function calls within the calling functions.

More refactoring examples

To see how different refactoring settings discussed above affect the refactoring result, let us consider the following examples.

All the examples are a simplified version of the refactoring shown earlier. In all cases, a new parameter `wireframe` of the type `Boolean` is added to the function `paint()` defined in the `IShape` interface.

In different examples, different combinations of the initializer and the default value are used, and the new parameter is either propagated to `Canvas.paint()` (which calls `IShape.paint()`) or not.

Initializer	Default value	Propagation used	Result

	`false`	Yes	`public interface IShape {`
			`  function paint(g:Graphics,`
			`  wireframe:Boolean):void;`
			`}`
			
			`// The function paint() in the Canvas class:`
			
			`  public function paint(g:Graphics,`
			`  wireframe:Boolean): void {`
			`  for each (var shape: IShape in shapes) {`
			`  shape.paint(g, wireframe);`
			`  }`
			`  }`

Initializer	Default value	Propagation used	Result

	false	No	public interface IShape {
			function paint(g:Graphics,
			wireframe:Boolean):void;
			}
			
			// The function paint() in the Canvas class:
			
			public function paint(g:Graphics): void {
			for each (var shape: IShape in shapes) {
			shape.paint(g, false);
			}
			}

true		Yes	public interface IShape {
			function paint(g:Graphics,
			wireframe:Boolean = true):void;
			}
			
			// The function paint() in the Canvas class:
			
			public function paint(g:Graphics): void {
			for each (var shape: IShape in shapes) {
			shape.paint(g);
			}
			}

true		No	public interface IShape {
			function paint(g:Graphics,
			wireframe:Boolean = true):void;
			}
			
			// The function paint() in the Canvas class:
			
			public function paint(g:Graphics): void {
			for each (var shape: IShape in shapes) {
			shape.paint(g);
			}
			}

true	false	Yes	public interface IShape {
			function paint(g:Graphics,
			wireframe:Boolean = true):void;
			}
			
			// The function paint() in the Canvas class:
			
			public function paint(g:Graphics,
			wireframe:Boolean = true): void {
			for each (var shape: IShape in shapes) {
			shape.paint(g, wireframe);
			}
			}

true	false	No	public interface IShape {
			function paint(g:Graphics,
			wireframe:Boolean = true):void;
			}
			
			// The function paint() in the Canvas class:
			
			public function paint(g:Graphics): void {
			for each (var shape: IShape in shapes) {
			shape.paint(g, false);
			}
			}

Changing a method signature

1. In the editor, place the cursor within the name of the method whose signature you want to change.

2. Do one of the following:
   - Press `Ctrl+F6`.
   - Choose **Refactor | Change Signature** in the main menu.
   - Select **Refactor | Change Signature** from the context menu.
3. In the Change Signature dialog, make the necessary changes to the method signature and specify which other, related changes are required.

   You can:

   - Change the method return type by editing the contents of the **Return type** field.

     Code completion is available in this field and also in certain fields of the table that contains the function parameters.

   - Change the method name. To do that, edit the text in the **Name** field.
   - Manage the method parameters using the table of parameters and the buttons to the right of it:
     - To add a new parameter, click + (`Alt+Insert`) and specify the properties of the new parameter in the corresponding fields.

       When adding parameters, you may want to propagate these parameters to the methods that call the current method.

     - To remove a parameter, click any of the cells in the corresponding row and click — (`Alt+Delete`).
     - To reorder the parameters, use ↑ (`Alt+Up`) and ↓ (`Alt+Down`). For example, if you wanted to make a certain parameter the first in the list, you would click any of the cells in the row corresponding to that parameter, and then click ↑ the required number of times.
     - To change the name or type for a parameter, make the necessary edits in the corresponding table cells.
   - Propagate new method parameters (if any) along the hierarchy of the methods that call the current method.

     (There may be the methods that call the method whose signature you are changing. These methods, in their turn, may be called by other methods, and so on. You can propagate the changes you are making to the parameters of the current method through the hierarchy of the calling methods and also specify which calling methods should be affected and which shouldn't.)

     To propagate the new parameters:

     1. Click (`Alt+G`).
     2. In the left-hand pane of the **Select Methods to Propagate New Parameters** dialog, expand the necessary nodes and select the check boxes next to the methods you want the new parameters to be propagated to.

        To help you select the necessary methods, the code for the calling method and the method being called is shown in the right-hand part of the dialog (in the **Caller Method** and **Callee Method** panes respectively).

        As you switch between the methods in the left-hand pane, the code in the right-hand part changes accordingly.

     3. Click **OK**.
4. To perform the refactoring right away, click **Refactor**.

   To see the expected changes and make the necessary adjustments prior to actually performing the refactoring, click **Preview**.

See Also

Reference:

- Change Signature Dialog for ActionScript

External Links:

- The Change Method Signature refactoring for ActionScript and Flex

Web Resources:

- Developer Community