Change Signature

In this section:

Basics

The *Change Signature* refactoring combines several different modifications that can be applied to a method signature. You can use this refactoring for the following purposes:

- To change the method name.
- To change the method return type.
- To add new parameters and remove the existing ones.
- To assign default values to the parameters.
- To reorder parameters.
- To change parameter names and types.
- To propagate new parameters through the method call hierarchy.

When changing a method signature, IntelliJ IDEA searches for all usages of the method and updates all the calls, implementations, and overridings of the method that can be safely modified to reflect the change.

- The Change Method Signature refactoring is supported for Java, PHP, JavaScript and ActionScript.
- You can access this refactoring from a UML Class diagram.
- The *Change Method Signature* refactoring is applicable to a constructor. In this case, however, the name and the return type cannot be changed.

Example

Before	After

// 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.
	// 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 possi
	// For more information, see the discussion tha

Initializer, default value, and propagation of new parameters

For each new parameter added to a method, 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 methods that call the function whose signature you are changing.

The refactoring result depends on whether you specify the default value and whether you use propagation.

Propagation. New parameters can be propagated to any method that calls the method whose signature you are changing. In such case, generally, the signatures of the calling methods 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 method calls and the signatures of the calling methods 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 method calls.

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

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

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

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

Initializer	Default value	Propagation used	Result

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

Initializer	Default value	Propagation used	Result

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

The following table shows 3 different ways of performing the same *Change Signature* refactoring.

In all the cases, the function `result()` is renamed to `generate_result()` and a new parameter `input` is added to this function.

The examples show how the function call, the calling function (`show_result()`) and other code fragments may be affected depending on the refactoring settings.

Before	A

// This is the function whose signature will be changed:
 	
function doSomething($a) {
// some code here
}
 	
function refactor($a) {
 	
// Here is the function call:
$this->doSomething($a);
}
 	
// Now we'll rename the function and
// add one parameter.
	// The function has been renamed
	// The new parameter $b has been
	
	function doSomeRefactoring($a,$b
	// some code here
	}
	
	function refactor($a,$b) {
	
	// The function call has been ch
	$this->doSomeRefactoring($a,$b)
	}

// This is the function whose signature will be changed:
 	
function doSomething($a) {
// some code here
}
 	
function refactor($a) {
 	
// Here is the function call:
$this->doSomething($a);
}
 	
// Now we'll rename the function and add one parameter.
//We will also specify the default value 'new_param'
//for this new parameter
	// The function has been renamed
	// The new parameter $b has been
	
	function doSomeRefactoring($a,$b
	// some code here
	}
	
	function refactor($a) {
	
	// The function call has been ch
	$this->doSomeRefactoring($a,'new
	}
	
	// When performing the refactor
	// the default parameter value.

Before	A

// This is the function whose signature will be changed:
 	
function doSomething($a) {
// some code here
}
 	
function refactor($a) {
 	
// Here is the function call:
$this->doSomething($a);
}
 	
// Now we'll rename the function and add one parameter.
//We will also propagate this new parameter
//through the calling function refactor() to the function call.
	// The function has been renamed
	// The new parameter $b has been
	
	function doSomeRefactoring($a,$b
	// some code here
	}
	
	// Note the new function parameter
	function refactor($a,$b) {
	
	// The function call has been ch
	$this->doSomeRefactoring($a,$b);
	}

To change 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** on the main menu.
   - Choose **Refactor | Change Signature** on 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 name. To do that, edit the text in the **Name** field.
   - Change the method return type by editing the contents of the **Return type** field.
   - Manage the method parameters using the table and the buttons in the **Parameters** area:
     - To add a new parameter, click + and specify the properties of the new parameter in the corresponding table row.

       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 —.
     - To reorder the parameters, use the ↑ and ↓ buttons. For example, if you wanted to put a certain parameter 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, type, the initializer, or the default value of a parameter, make the necessary edits in the table of parameters (in the fields **Name Type**, **Initializer** and **Default value** respectively).
   - Propagate new method parameters (if any) along the hierarchy of the methods that call the current method.

     (There may be 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 calling methods and also specify which calling methods should be affected and which shouldn't.)

     To propagate a new parameter:

     1. Click the **Propagate Parameters** button or press `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 pane 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**.

Code completion is available in the **Default value** field of the table in the **Parameters** area.

See Also

Code Examples:

- Java
- ActionScript
- More refactoring examples for ActionScript

Procedures:

- Changing a method signature in Java
- Changing a method signature in ActionScript

Reference:

- Change Signature Dialog for Java
- Change Signature Dialog for ActionScript
- Change Signature Dialog for JavaScript

Web Resources:

- Developer Community