ExtractParameter in Java

This section discusses the Extract Parameter refactoring in Java.

Examples

When extracting a new parameter to a method, the following two general approaches may be used depending on how the existing method calls should be handled:

- If it's possible to change all the existing method calls, a new parameter may be added to an existing method. The method calls in this case are changed accordingly, see the first of the examples.
- If the existing method calls cannot be changed, a method with the existing signature is kept. The new parameter in this case is defined in a new, overloading method, see the second of the examples.

In this example, the string value `"Hello, World!"` in the method `generateText()` is replaced with the new parameter `text`. The value `"Hello, World!"` is passed to the method in the updated method call `generateText("Hello, World!")`.

Before	After

`public class HelloWorldPrinter {`
`  public static void print() {`
`  System.out.println(generateText());`
`  }`
`  private static String generateText() {`
`  return "Hello, World!".toUpperCase();`
`  }`
`}`
	`public class HelloWorldPrinter {`
	`  public static void print() {`
	`  System.out.println(generateText("Hello, World!"));`
	`  }`
	`  private static String generateText(String text) {`
	`  return text.toUpperCase();`
	`  }`
	`}`

In this example a new overloading method is created and the new parameter is extracted in the definition of this method (the second of the `generateText()` methods). The signature of the existing `generateText()` method is not changed. However, the method itself has been modified. Now, it calls the new `generateText()` method and passes the value `"Hello, World!"` to it in this call. Note that the existing call of `generateText()` (in the method `print()`) is not changed.

In IntelliJ IDEA , this way of extracting a parameter corresponds to the option **Delegate via overloading method**.

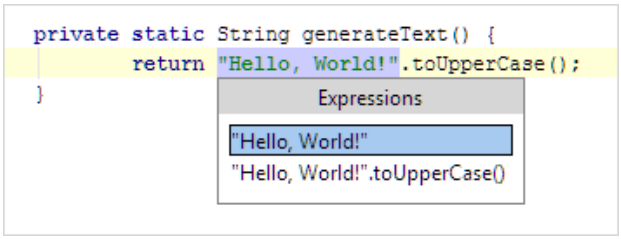
Before	After

`public class HelloWorldPrinter {`
`  public static void print() {`
`  System.out.println(generateText());`
`  }`
`  private static String generateText() {`
`  return "Hello, World!".toUpperCase();`
`  }`
`}`
	`public class HelloWorldPrinter {`
	`  public static void print() {`
	`  System.out.println(generateText());`
	`  }`
	`  private static String generateText() {`
	`  return generateText("Hello, World!");`
	`  }`
	`  private static String generateText(String text) {`
	`  return text.toUpperCase();`
	`  }`
	`}`

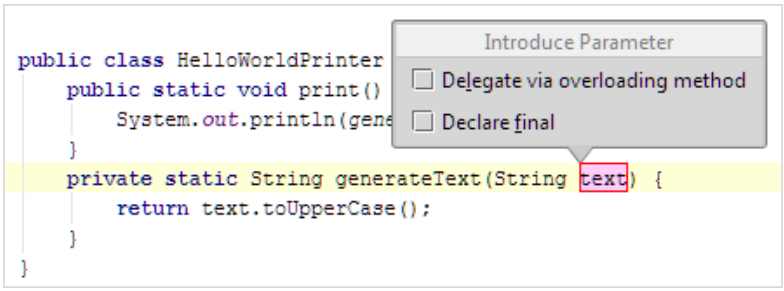
Extracting a parameter in Java in-place

The in-place refactorings are enabled in IntelliJ IDEA by default. So, if you haven't changed this setting, the Extract Parameter refactorings for Java are performed in-place, right in the editor:

1. In the editor, place the cursor within the expression to be replaced by a parameter.
2. Do one of the following:
   - Press **Ctrl+Alt+P**.
   - Choose **Refactor | Extract | Parameter** in the main menu.
   - Choose **Refactor | Extract | Parameter** from the context menu.
3. If more than one expression is detected for the current cursor position, the **Expressions** list appears. If this is the case, select the required expression. To do that, click the expression. Alternatively, use the **Up** and **Down** arrow keys to navigate to the expression of interest, and then press **Enter** to select it.



4. Type the parameter name in the box with a red border.



5. Select the necessary options in the **Extract Parameter** option box.
   - If more than one occurrence of the expression is found within the method body, you can choose to replace only the selected occurrence or all the found occurrences with the references to the new parameter. Use the **Replace all occurrences** check box to specify your intention.
   - If you don't want to change the existing method calls, select the **Delegate via overloading method** check box. For more information on how this option works, see Examples.
   - To declare the parameter `final`, select the **Declare final** check box.
6. If necessary, change the type of the new parameter. To do that, press **Shift+Tab** and edit the type in the box with the read border. (If, at this step, you want to return to editing the parameter name, press **Tab**.)
7. To complete the refactoring, press **Tab** or **Enter**.

   If you haven't completed the refactoring and want to cancel the changes you have made, press **Escape**.

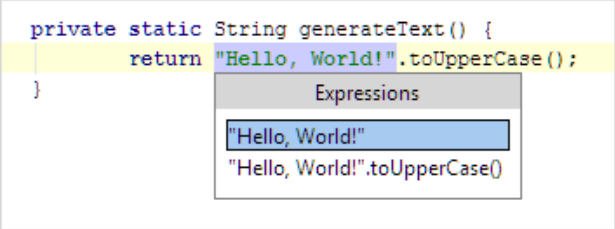
   Note that sometimes you may need to press the corresponding key more than once.

Extracting a parameter in Java using the Extract Parameter dialog

To be able to use the **Extract Parameter** dialog (instead of performing the refactoring in-place), make sure that the Enable in place refactorings option is off in the editor settings.

Once this is the case, you perform the Extract Parameter refactoring as follows:

1. In the editor, place the cursor within the expression to be replaced by a parameter.
2. Do one of the following:
   - Press `Ctrl+Alt+P`.
   - Choose **Refactor | Extract | Parameter** in the main menu.
   - Choose **Refactor | Extract | Parameter** from the context menu.
3. If more than one expression is detected for the current cursor position, the **Expressions** list appears. If this is the case, select the required expression. To do that, click the expression. Alternatively, use the `Up` and `Down` arrow keys to navigate to the expression of interest, and then press `Enter` to select it.



4. In the Extract Parameter dialog:
   1. Usually, IntelliJ IDEA sets a proper parameter type itself. If necessary, you can select another appropriate type from the **Parameter of type** list.
   2. Specify the parameter name in the **Name** field.
   3. If more than one occurrence of the expression is found within the method body, you can choose to replace only the selected occurrence or all the found occurrences with the references to the new parameter. Use the **Replace all occurrences** check box to specify your intention.
   4. To declare the parameter `final`, select the **Declare final** check box.
   5. If the expression contains a direct call to a class field that has a getter, specify the getter-related options.
   6. If the expression contains local variables, specify how these local variables will be treated in the corrected method calls.
5. Preview and apply changes.

Special notes

The following specific issues should be mentioned:

- If you want a field to be provided as a new parameter in the method declaration, in a method call this field will be presented as a field of a class instance.
- If a class member is inaccessible (for instance, in the example above the field is private), it will be inserted in a method call but will be highlighted as an error making this file uncompilable.
- If you use for the Extract Parameter refactoring a field with a getter, you will be prompted with an extended dialog. The dialog has an option group **Replace fields used in expressions with their getters**:
  - **Do not replace**: None of the fields will be replaced with calls to the getter.
  - **Replace fields inaccessible in usage context**: Only the fields that cannot be directly accessed from the usage context will be replaced with calls to the getter.
  - **Replace all fields**: All fields will be replaced with calls to the getter.

- Applying the refactoring on a local variable will call the **Extract Parameter** dialog box with additional check boxes:
  - **Replace all occurrences (<number_of_occurrences> occurrences)**: If enabled, all occurrences of the selected variable will be replaced with a parameter and the **Delete variable definition** check box is enabled. Otherwise, only the selected variable usage will be replaced with a parameter.
  - **Delete variable definition**: If enabled, the variable definition will be deleted.
  - **Use variable initializer to initialize parameter**: If enabled, the variable initializer will be used to initialize the parameter in the method call.

Side effects

Using the **Extract Parameter** refactoring can have unexpected side effects if applied on class instances or expressions which are actual method parameters. For instance, in case of such code:

```
      class AClass {
    intfield;
    int method() {
        returnfield;
    }
}

class Usage {
    void method(List list) {
        int sum = 0;
        for(Iterator it = list.iterator(); it.hasNext(); ) {
            sum += ((AClass) it.next()).method();
        }
    }
}
```

And the code after the refactoring applied for the field `field`:

```
      class AClass {
    intfield;
    int method(int newfield) {
        return newfield;
    }
}

class Usage {
    void method(List list) {
        int sum = 0;
        for(Iterator it = list.iterator; it.hasNext(); ) {
            sum += ((AClass) it.next()).method(((AClass) it.next()).field);
        }
    }
}
```

The iterator value is increased twice which is, actually, not the behavior you would expect.

However, IntelliJ IDEA can use a temporary variable successfully and resolve such cases as increment/decrement/assignment operations and the new keyword usage. For instance:

```
      public int myMethod(List list) {
    return list.size();
}

public void anotherMethod() {
    myMethod(new ArrayList());
}
```

And the code after refactoring:

```
      public int myMethod(List list, int newPar) {
    return list.size();
}

public void anotherMethod() {
    final ArrayList list = new ArrayList();
    myMethod(list, list.size());
}
```

The new variable `list` was created and all parameters used for the method call are provided using this variable.