Java Editor

Contents

Java editor

Toolbar actions

Toolbar Button	Command	Description
	Toggle Java Editor Breadcrumb	This button enables the Java editor breadcrumb. The enablement is remembered for each perspective separately.
	Toggle Mark Occurrences 共现	Turns mark occurrences on and off in the Java editor.
	鼠标双击 Toggle ~~Block Selection Mode~~	This button enables block (aka column) selection mode in the editor.
	Show ~~Whitespace Characters~~	This button enables the display of whitespace characters in an editor.
	Show Source of Selected Element Only	This button enabled display of a segmented view of the source of a compilation unit. This button is only shown if you customize your perspective to show the **Editor Presentation** actions. For example, if a method is selected in the Outline view, the **Show Source Of Selected Element Only** option causes only that method to be displayed in the editor, as opposed to

Toolbar Button	Command	Description
		the entire class. **Off**: The entire compilation unit is displayed in the editor, with the selected Java element highlighted in the marker bar with a range indicator. **On**: Only the selected Java element is displayed in the editor, which is linked to the selection in the Outline or Type Hierarchy view.
	Go to Next Problem	This command navigates to the next problem marker in the active editor.
	Go to Previous Problem	This command navigates to the previous problem marker in the active editor.

Key binding actions

The following actions can only be reached through key bindings. The **Key bindings** field in **Window > Preferences > General > Keys** must be set to 'Emacs'.

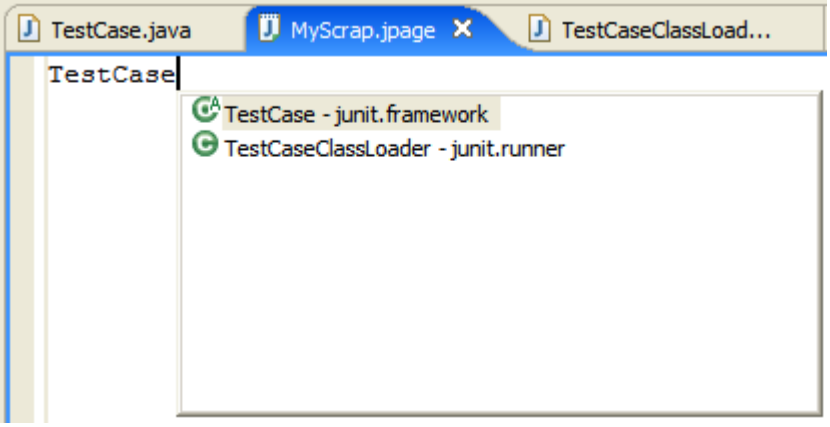
Key binding	Description
`Alt+0 Ctrl+K`, `Esc 0 Ctrl+K`	Deletes from the cursor position to the beginning of the line.
`Ctrl+K`	Deletes from the cursor position to the end of the line.
`Ctrl+Space`, `Ctrl+2`	Sets a mark at the current cursor position.
`Ctrl+X Ctrl+X`	Swaps the cursor and mark position if any.

1. Content/Code Assist

Alt + /

In the Java editor press **Ctrl + Space** on code to complete. This opens a list of available code completions. Some tips for using code assist are listed in the following paragraph:

- You can use the mouse or the keyboard (Up Arrow, Down Arrow, Page Up, Page Down, Home, End, Enter) to navigate and select lines in the list.
- If you select a line in the content assist list, you can view Javadoc information for that line.
- Clicking or pressing Enter on a selected line in the list inserts the selection into the editor.
- You can access specialized content assist features inside Javadoc comments.



Configure the behavior of the content assist in the Java > Editor > Code Assist preference page.

2. Formatter

In the Java editor press **Ctrl+Shift+F** on code to format it. If no selection is set then the entire source is formatted otherwise only the selection will be. Some tips for using the formatter are listed in the paragraphs of this chapter.

Note that the Java Formatter preferences are accessible on the **Java Formatter** preference page.

Disabling formatter inside sections

You can disable/enable the formatter in one or several sections in the code as shown in the sample below:

```
public class Example {
    // area 1: formatted
    int x;

/**
 * The method foo.
 * @formatter:off
 */
public    void
foo()   {
for (int i=0;i<10;i++){
//  area    2:     NOT     formatted   !
}/* @formatter:on */
        for (int i = 0; i < 10; i++) {
            // area 3: formatted
        }/* @formatter:on */
    }

    void bar() {
        int i = 0;
        while (i++ < 10) {
        }
//@formatter:off
int j=0;while (j++<10){
//  area    4:     NOT     formatted   !
}}}
```

The snippet above use default tag names, but they can be changed on the **Off/On tags** tab of the Java Formatter preference page.

Wrap outermost method calls

Since version 3.6, the Java formatter now tries to wrap the outermost method calls first to have a better output when wrapping nested method calls.

Here is an example of a formatted code where the formatter has wrapped the line between the arguments of the outermost message call to keep each nested method call on a single line:

```
public class X {
    void test() {
        foo(bar(1, 2, 3, 4),
                bar(5, 6, 7, 8));
    }
```

A new preference allows you to disable this strategy, typically if you want to format your code as before, then uncheck the **Prefer wrapping outer expressions** preference accessible on the **Line wrapping** tab of the Java Formatter preference page.

Note: Currently the new strategy only applies to nested method calls, but that might be extended to other nested expressions in future versions.

Condense Javadoc and block comments

Users can reduce the number of lines of formatted multi-lines comments as shown in the example below:

```
/* The bar private method. */
private void bar() {
}
```

To activate this behavior uncheck the **/* and */ on separate lines** preference accessible on the **Comments** tab of the Java Formatter preference page.

The same kind of preference is also available for the Javadoc comments.

Preserve user line breaks

Users can preserve line breaks by not joining lines in code or comments.

For example, the already wrapped lines of the `return` statement in the following test case:

```
class X {
String foo() {
return "select x "
      + "from y "
      + "where z=a";
}
}
```

will be preserved by the formatter when the **Never join lines** preference is used, hence produces the following output when formatted:

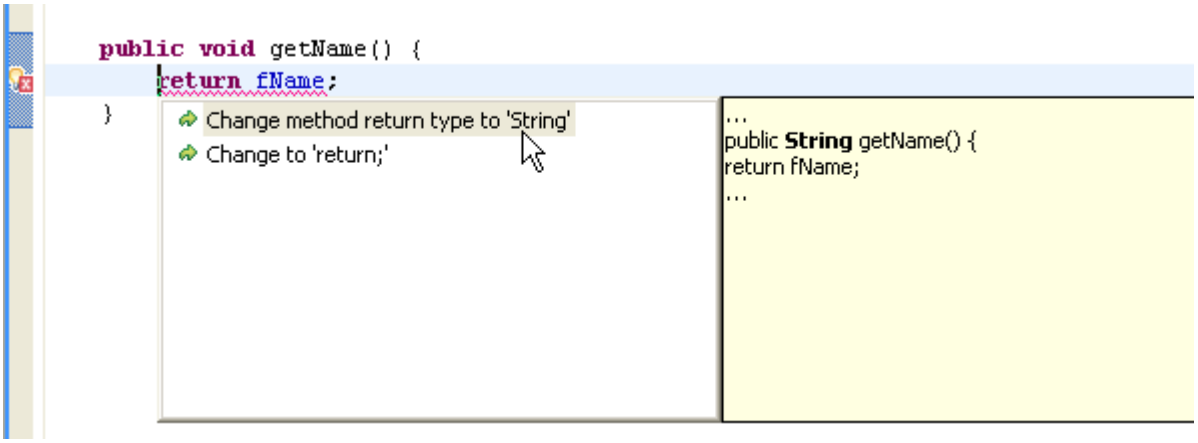
```
class X {
   String foo() {
      return "select x "
            + "from y "
            + "where z=a";
   }
}
```

To activate this behavior check the **Never join lines** preference accessible on the **Line Wrapping** and the **Comments** tabs of the Java Formatter preference page.

3. Quick Fix

The Java editor offers corrections to problems found while typing and after compiling. To show that correction proposals are available for a problem or warning, a 'light bulb' is visible on the editor's annotation bar.

Left click on the light bulb or invoking **Ctrl+1 (Edit > Quick Fix)** brings up the proposals for the problem at the cursor position.
Each quick fix shows a preview when selected in the proposal window.



Some selected quick fixes can also be assigned with direct shortcuts. You can configure these shortcuts on the **General > Keys** preference page (in the 'Source' category).

Some quick fixes offer to fix all problems of the same kind in the current file at once. The information text in proposal window contains this information for all applicable proposals. To fix all problems of the same kind, press **Ctrl + Enter**.

The following quick fixes are available:

Package Declaration

- Add missing package declaration or correct package declaration
- Move compilation unit to package that corresponds to the package declaration

Imports

- Remove unused, unresolvable or non-visible import
- Invoke 'Organize imports' on problems in imports

Types

- Create new class, interface, enum, annotation or type variable for references to types that can not be resolved
- Change visibility for types that are accessed but not visible
- Rename to a similar type for references to types that can not be resolved
- Add import statement for types that can not be resolved but exist in the project
- Add explicit import statement for ambiguous type references (two import-on-demands for the same type)
- If the type name is not matching with the compilation unit name either rename the type or rename the compilation unit
- Remove unused private types
- Add missing type annotation attributes

Constructors

- Create new constructor for references to constructors that can not be resolved (this, super or new class creation)
- Reorder, add or remove arguments for constructor references that mismatch parameters

- Change method with constructor name to constructor (remove return type)
- Change visibility for constructors that are accessed but not visible
- Remove unused private constructor
- Create constructor when super call of the implicit default constructor is undefined, not visible or throws an exception
- If type contains unimplemented methods, change type modifier to 'abstract' or add the method to implement

Methods

- Create new method for references to methods that can not be resolved
- Rename to a similar method for references to methods that can not be resolved
- Reorder or remove arguments for method references that mismatch parameters
- Correct access (visibility, static) of referenced methods
- Remove unused private methods
- Correct return type for methods that have a missing return type or where the return type does not match the return statement
- Add return statement if missing
- For non-abstract methods with no body change to 'abstract' or add body
- For an abstract method in a non-abstract type remove abstract modifier of the method or make type abstract
- For an abstract/native method with body remove the abstract or native modifier or remove body
- Change method access to 'static' if method is invoked inside a constructor invocation (super, this)
- Change method access to default access to avoid emulated method access
- Add 'synchronized' modifier
- Override hashCode()
- Open the 'Generate hashCode() and equals()' wizard

Fields and variables

- Correct access (visibility, static) of referenced fields
- Create new fields, parameters, local variables or constants for references to variables that can not be resolved
- Rename to a variable with similar name for references that can not be resolved
- Remove unused private fields
- Correct non-static access of static fields
- Add 'final' modifier to local variables accessed in outer types
- Change field access to default access to avoid emulated method access
- Change local variable type to fix a type mismatch
- Initialize a variable that has not been initialized
- Create getter and setters for invisible or unused fields

Exception Handling

- Remove unneeded catch block
- Handle uncaught exception by surrounding with try/catch or adding catch block to a surrounding try block
- Handle uncaught exception by adding a throw declaration to the parent method or by generalize an existing throw declaration

Build Path Problems

- Add a missing JAR or library for an unresolvable type
- Open the build path dialog for access restriction problems or missing binary classes.
- Change project compliance and JRE to 1.5
- Change workspace compliance and JRE to 1.5

Others

- Add cast or change cast to fix type mismatches
- Let a type implement an interface to fix type mismatches
- Add type arguments to raw references
- Complete switch statements over enums
- Remove dead code
- Insert '//$FALL-THROUGH$'
- Insert null check
- For non-NLS strings open the NLS wizard or mark as non-NLS
- Add missing @Override, @Deprecated annotations
- Add missing Javadoc comments
- Add missing Javadoc tags
- Suppress a warning using @SuppressWarnings
- Throw the allocated object
- Return the allocated object

看了"Quick Fix"，让我想去尝试迭代式开发模式：
以主干逻辑为核心，一步一步延伸出去。

Quick Assists are proposals available even if there is no problem or warning. See the Quick Assist page for more information.

4. Quick Assist

Quick assists perform local code transformations. They are invoked on a selection or a single cursor in the Java editor and use the same shortcut as quick fixes (**Ctrl+1**), but quick assist are usually hidden when an error is around. To show them even with errors present on the same line, press **Ctrl+1** a second time.

A selection of quick assists can be assigned to a direct shortcut. By default, these are:

- Rename in file: **Ctrl+2, R**
- Assign to local: **Ctrl+2, L**
- Assign to field: **Ctrl+2, F**

Assign more shortcuts or change the default shortcuts on the **General > Keys** preference page (in the 'Source' category).

A quick assist light bulb can be turned on on the **Java > Editor** preference page.

Name	Code example			Invocation location
Inverse if statement	`if (x) a(); else b();`	>	`if (!x) b(); else a();`	On 'if' statements with 'else' block
Inverse boolean expression	`a && !b`	>	`!a || b`	On a boolean expression
Invert local variable	`boolean a = false;` `if (a) {}`	>	`boolean notA = true;` `if (!notA) {}`	On a boolean variable
Invert equals	`a.equals(b)`	>	`b.equals(a)`	On a invocation of 'equals'
Inverse conditional expression	`x ? b : c`	>	`!x ? c : b`	On a conditional expression
Pull negation up	`b && c`	>	`!(!b || !c)`	On a boolean expression
Push negation down	`!(b && c)`	>	`!b || !c`	On a negated boolean expression
Remove extra parentheses	`if ((a == b) && (c != d) {}`	>	`if (a == b && c != d) {}`	On selected expressions
Put expression in parentheses	`return a > 10 ? 1 : 2;`	>	`return (a > 10) ? 1 : 2;`	On selected expression
Put expressions in parentheses	`if (a == b && c != d) {}`	>	`if ((a == b) && (c != d)) {}`	On selected expressions
Join nested if statements	`if (a) { if (b) {} }`	>	`if (a && b) {}`	On a nested if statement
Swap nested if statements	`if (a) { if (b) {} }`	>	`if (b) { if (a) {} }`	On a nested if statement

Split if statement with and'ed expression	`if (a && b) {}`	>	`if (a) { if (b) {} }`	On an and'ed expression in a 'if'
Join selected 'if' statements with ||	`if (a) x();  if (b) x();`	>	`if (a || b) x();`	On selected 'if' statements
Join 'if' sequence in if-else-if	`if (a) x();  if (b) y();`	>	`if (a) x();  else if (b) y();`	On selected 'if' statements
Split if statement with or'd expression	`if (a || b) x();`	>	`if (a) x();  if (b) x();`	On an or'd expression in a 'if'
If-else assignment to conditional expression	`if (a) x= 1; else x= 2;`	>	`x= a ? 1 : 2;`	On an 'if' statement
If-else return to conditional expression	`if (a) return 1;` `else return 2;`	>	`return a ? 1 : 2;`	On an 'if' statement
Conditional expression assignment to If-else	`x= a ? 1 : 2;`	>	`if (a) x= 1; else x= 2;`	On a conditional expression
Conditional expression return to If-else	`return  a ? 1 : 2;`	>	`if (a) return 1; else return 2;`	On a conditional expression

Switch to If-else	`switch (kind) {`
	`case 1: return -1;`
	`case 2: return -2;`
	`}`
		>	`if (kind == 1) {`
			`  return -1;`
			`} else if (kind == 2) {`
			`  return -2;`
			`}`
				On a switch statement

Add missing case statements on enums	`switch (e){`
	`}`
		>	`switch (e){`
			`case E1: break;`
			`case E2: break;`
			`}`
				On a switch statement

Exchange operands	`a + b`	>	`b + a`	On an infix operation

Cast and assign	`if (obj instanceof Vector)`
	`{`
	`}`
		>	`if (obj instanceof Vector)`
			`{`
			`  Vector vec= (Vector)obj;`
			`}`
				On an instanceof expression in an 'if' or 'while' statement

Add finally block	`try { } catch (Expression e) { }`	>	`try { } catch (Expression e) { } finally {}`	On a try/catch statement
Add else block	`if (a) b();`	>	`if (a) b(); else { }`	On a if statement
Replace statement with block	`if (a) b();`	>	`if (a) { b(); }`	On a if statement
Unwrap blocks	`{ a() }`	>	`a()`	On blocks, if/while/for statements
Pick out string	`"abcdefgh"`	>	`"abc" + "de" + "fgh"`	select a part of a string literal
Convert string concatenation to StringBuilder (J2SE 5.0) or StringBuffer	`"Hello " + name`	>	`StringBuilder builder= new StringBuilder(); builder.append("Hello "); builder.append(name);`	select a string literal
Convert string concatenation to MessageFormat	`"Hello " + name`	>	`MessageFormat.format("Hello {0}", name);`	select a string literal
Split variable	`int i= 0;`	>	`int i; i= 0;`	On a variable with initialization
Join variable	`int i; i= 0;`	>	`int i= 0`	On a variable without initialization
Assign to variable	`foo()`	>	`X x= foo();`	On an expression statement
Extract to local	`foo(getColor());`	>	`Color color= getColor(); foo(color);`	On an expression
Assign parameter to field	`public A(int color) {}`	>	`Color fColor; public A(int color) { fColor= color; }`	On a parameter
Array initializer to Array creation	`int[] i= { 1, 2, 3 }`	>	`int[] i= new int[] { 1, 2, 3 }`	On an array initializer

Convert to 'enhanced for loop' (J2SE 1.5)	`for (Iterator i= c.iterator();i.hasNext();) { }`	>	`for (x : c) { }`	On a for loop
Create method in super class				On a method declaration
Rename in file				On identifiers
Rename in workspace				On identifiers
Extract to local variable	`a= b*8;`	>	`int x= b*8;` `a= x;`	On expressions
Extract to constant	`a= 8;`	>	`final static int CONST= 8;` `a= CONST;`	On expressions
Extract method	`int x= p * 5;`	>	`int x= getFoo(p);`	On expressions and statements
Inline local variable	`int a= 8, b= a;`	>	`int b= 8;`	On local variables
Convert local variable to field	`void foo() { int a= 8; }`	>	`int a= 8; void foo() {}`	On local variables
Convert anonymous to nested class	`new Runnable() { };`	>	`class RunnableImplementation implements Runnable { }`	On anonymous classes
Replace with getter and setter (Encapsulate Field)	`p.x;`	>	`p.getX();`	On fields

5. Excluding warnings using @SuppressWarnings

Since Java 5.0, you can disable compilation warnings relative to a subset of a compilation unit using the `java.lang.SuppressWarning` annotation.

```
@SuppressWarning("unused") public void foo() {
 String s;
}
```

Without the annotation, the compiler would complain that the local variable `s` is never used. With the annotation, the compiler silently ignores this warning locally to the `foo` method. This enables to keep the warnings in other locations of the same compilation unit or the same project.

The list of tokens that can be used inside a `SuppressWarnings` annotation is:

- all to suppress all warnings
- boxing to suppress warnings relative to boxing/unboxing operations
- cast to suppress warnings relative to cast operations
- dep-ann to suppress warnings relative to deprecated annotation
- deprecation to suppress warnings relative to deprecation
- fallthrough to suppress warnings relative to missing breaks in switch statements
- finally to suppress warnings relative to finally block that don't return
- hiding to suppress warnings relative to locals that hide variable
- incomplete-switch to suppress warnings relative to missing entries in a switch statement (enum case)
- javadoc to suppress warnings relative to javadoc warnings
- nls to suppress warnings relative to non-nls string literals
- null to suppress warnings relative to null analysis
- rawtypes to suppress warnings relative to usage of raw types
- restriction to suppress warnings relative to usage of discouraged or forbidden references
- serial to suppress warnings relative to missing serialVersionUID field for a serializable class
- static-access to suppress warnings relative to incorrect static access
- static-method to suppress warnings relative to methods that could be declared as static
- super to suppress warnings relative to overriding a method without super invocations
- synthetic-access to suppress warnings relative to unoptimized access from inner classes
- unchecked to suppress warnings relative to unchecked operations
- unqualified-field-access to suppress warnings relative to field access unqualified
- unused to suppress warnings relative to unused code and dead code

Preferences -> Java -> Compiler -> Errors/Warnings
自定义错误、警告、忽略级别，降低出错的概率！