JDT Programmer's Guide

OTI

Table of Contents

JDT Programmer's Guide **1**
JDT Core 1
JDT UI 2
JDT Debug 2

JDT Extension Points **4**

Classpath Variable Initializers **5**
Identifier: 5
Since: 5

Classpath Container Initializers **7**
Identifier: 7
Since: 7

Code Formatters **9**
Identifier: 9
Since: 9

Java Logical Structures **11**
Identifier: 11
Since: 11

VM Install Type UI Page **14**
Identifier: 14
Description: 14

JUnit Launch Configurations **16**
Identifier: 16
Since: 16

Test Run Listeners **18**
Identifier: 18
Since: 18

Java Runtime Classpath Providers **20**
Identifier: 20
Since: 20

Java Runtime Classpath Entries **22**
Identifier: 22
Since: 22

Java Runtime Classpath Entry Resolvers **24**
Identifier: 24
Description: 24

Table of Contents

Java VM Connectors **26**
Identifier: 26
Description: 26

Java VM Install Types **28**
Identifier: 28
Description: 28

Classpath Container Entry Page **30**
Identifier: 30
Description: 30

Java Folding Structure Provider **32**
Identifier: 32
Since: 32

Javadoc Completion Processor **34**
Identifier: 34
Description: 34

Java Editor Text Hovers **36**
Identifier: 36
Description: 36

Java Element Filter Extensions **38**
Identifier: 38
Description: 38

Java Query Participants **41**
Identifier: 41
Since: 41

Quick Assist Processor **43**
Identifier: 43
Since: 43

Quick Fix Processor **50**
Identifier: 50
Since: 50

Other Reference Information **57**

JDT Overview – Map of JDT Plug–ins **58**

Installing the examples **59**
Java Example Projects 59
Introduction 59
Loading the Samples 59

Table of Contents

JDT Questions Index **60**
JDT Core 60
JDT UI 60
Running a Java program 60
Launching a compiled Java program 60
Compiling Java code 61
Compiling code 61
Using the batch compiler 62
Using the ant javac adapter 66
Problem determination 67
Setting the Java build path 68
Changing the build path 68
Classpath entries 69
Exclusion patterns 71
Inclusion patterns 71
Classpath resolution 72
Manipulating Java code 72
Code modification using Java elements 72
Code modification using the DOM/AST API 74
Responding to changes in Java elements 78
Using the Java search engine 80
Preparing for search 80
Searching 81
Collecting search results 82
JDT Core options 82
Project specific options 83
Major change in default JDT Core 3.0 options 83
JDT Core options descriptions 84
Performing code assist on Java code 123
Code completion 123
Code selection 124
Java model 125
Java elements 125
Java elements and their resources 128
Java projects 129
Opening a Java editor 129
Creating Java specific prompter dialogs 129
Presenting Java elements in a JFace viewer 130
Overlaying images with Java information 130
Adding problem and override decorators 130
Updating the presentation on model changes 131
Sorting the viewer 131
Programmatically Writing a Jar file 131
Java wizard pages 132
Configuring Java build settings 132
Creating new Java elements 132
Contributing a classpath container wizard page 133
Customizing a wizard page 133

Table of Contents

JDT Questions Index
Notices....134
About This Content....135
License....135

JDT Programmer's Guide

The Eclipse platform is delivered with a full featured Java integrated development environment (IDE). Java development tooling (JDT) allows users to write, compile, test, debug, and edit programs written in the Java programming language.

The JDT makes use of many of the platform extension points and frameworks described in the Platform Plug−in Developer Guide. It's easiest to think of the JDT as a set of plug−ins that add Java specific behavior to the generic platform resource model and contribute Java specific views, editors, and actions to the workbench.

This guide discusses the extension points and API provided by the JDT. We assume that you already understand the concepts of plug−ins, extension points, workspace resources, and the workbench UI.

Given that the JDT supplies a full featured Java IDE, why would you need to use the JDT API? If you are building a plug−in that interacts with Java programs or resources as part of its function, you may need to do one or more of the following things:

- Programmatically manipulate Java resources, such as creating projects, generating Java source code, performing builds, or detecting problems in code.
- Programmatically launch a Java program from the platform
- Provide a new type of VM launcher to support a new family of Java runtimes
- Add new functions and extensions to the Java IDE itself

The JDT is structured into three major components:

- JDT Core − the headless infrastructure for compiling and manipulating Java code.
- JDT UI − the user interface extensions that provide the IDE.
- JDT Debug − program launching and debug support specific to the Java programming language.

We'll examine each component's structure and the API it provides.

JDT Core

JDT Core (**org.eclipse.jdt.core**) is the plug−in that defines the core Java elements and API. You should always list this plug−in as a prerequisite when you are developing Java specific features.

JDT Core packages give you access to the Java model objects and headless Java IDE infrastructure. The JDT Core packages include:

- **org.eclipse.jdt.core** − defines the classes that describe the Java model.
- **org.eclipse.jdt.core.compiler** − defines API for the compiler infrastructure.
- **org.eclipse.jdt.core.dom** − supports Abstract Syntax Trees (AST) that can be used for examining the structure of a compilation unit down to the statement level.
- **org.eclipse.jdt.core.eval** − supports the evaluation of code snippets in a scrapbook or inside the debugger.
- **org.eclipse.jdt.core.jdom** − supports a Java Document Object Model (DOM) that can be used for walking the structure of a Java compilation unit.
- **org.eclipse.jdt.core.search** − supports searching the workspace's Java model for Java elements match

a particular description.
- **org.eclipse.jdt.core.util** – provides utility classes for manipulating .class files and Java model elements.

Manipulation of the structure of a compilation unit should be done using `org.eclipse.jdt.core.dom` instead of `org.eclipse.jdt.core.jdom`. `org.eclipse.jdt.core.jdom` will be deprecated in the 2.2. stream and replaced with `org.eclipse.jdt.core.dom`.

JDT UI

JDT UI (**org.eclipse.jdt.ui**) is the plug–in implementing the Java specific user interface classes that manipulate Java elements. The packages in the JDT UI implement the Java–specific extensions to the workbench. The JDT UI packages include:

- **org.eclipse.jdt.ui** – provides support classes for presenting Java elements in the user interface. This package exposes constants for retrieving Java user interface parts from the workbench registry and for retrieving preference settings from the Java preferences. Programming interfaces **ITypeHierarchyViewPart** and **IPackagesViewPart** are provided for interacting with Java views.
- **org.eclipse.jdt.ui.actions** – provides actions and action groups to populate tool bars, global menu bars and context menus with JDT specific functionality. You should use action groups to populate menus and tool bars instead of adding actions directly. This shields you from missing newly added actions or from using outdated menu structures.
- **org.eclipse.jdt.ui.jarpackager** – provides classes and interfaces to generate a JAR file.
- **org.eclipse.jdt.ui.refactoring** – provides support for running rename refactorings
- **org.eclipse.jdt.ui.search** – provides classes and interfaces to contribute participants to a Java search
- **org.eclipse.jdt.ui.text** – provides support classes for presenting Java text.
- **org.eclipse.jdt.ui.text.folding** – provides interfaces to implement code folding strategies for the Java editor.
- **org.eclipse.jdt.ui.text.java** – provides interfaces to implement code completion processors.
- **org.eclipse.jdt.ui.text.java.hover** – provides implementations for presenting text hovers in Java editors.
- **org.eclipse.jdt.ui.wizards** – provides wizard pages for creating and configuring Java elements.

JDT Debug

JDT Debug is comprised of several plug–ins that support the running and debugging of Java code.

- **org.eclipse.jdt.launching** is the plug–in that defines the Java launching and runtime support. You will require this plug–in if you need to launch Java virtual machines programmatically. The JDT launching is closely tied to the platform launching facility, which is described in Launching a program.

  The package **org.eclipse.jdt.launching** provides classes for launching Java runtimes from the platform. **JavaRuntime** implements static methods to access registered VMs and compute runtime classpaths and source lookup paths. A family of VM's (such as the JDK) is represented by the **IVMInstallType** class. **IVMInstall** represents particular installations within a family. The **IVMRunner** is used to start a particular Java VM and register its processes with the debug plug–in.

  The package **org.eclipse.jdt.launching.sourcelookup.containers** defines classes for locating source code elements in the file system.

- **org.eclipse.jdt.debug** is the plug−in that defines the Java debug model. You will require this plug−in if you need to programmatically access objects in a program being debugged. The JDT debug model is closely tied to the platform debug model, which is described in Platform debug model.

  The package **org.eclipse.jdt.debug.core** supports a Java debug model based on JDI/JDWP that can be used for controlling a Java program under debug.

  The package **org.eclipse.jdt.debug.eval** provides infrastructure for evaluating Java expressions and reporting results.
- **org.eclipse.jdt.debug.ui** is the plug−in that defines the Java debug UI extensions. Most of the debugger API is provided by the platform debugger infrastructure described in Debug model presentation and Debug UI utility classes. The Java debug UI API focuses on accessing the prompting source locator and Java launch configuration tabs.

  The package **org.eclipse.jdt.debug.ui.launchConfigurations** defines the launch configuration tabs for local and remote Java applications.

JDT Extension Points

The following extension points can be used to extend the capabilities of the JDT infrastructure:

- org.eclipse.jdt.core.classpathVariableInitializer
- org.eclipse.jdt.core.classpathContainerInitializer
- org.eclipse.jdt.core.codeFormatter
- org.eclipse.jdt.debug.javaLogicalStructures
- org.eclipse.jdt.debug.ui.vmInstallTypePage
- org.eclipse.jdt.junit.junitLaunchConfigs
- org.eclipse.jdt.junit.testRunListeners
- org.eclipse.jdt.launching.classpathProviders
- org.eclipse.jdt.launching.runtimeClasspathEntries
- org.eclipse.jdt.launching.runtimeClasspathEntryResolvers
- org.eclipse.jdt.launching.vmConnectors
- org.eclipse.jdt.launching.vmInstallTypes
- org.eclipse.jdt.ui.classpathContainerPage
- org.eclipse.jdt.ui.foldingStructureProviders
- org.eclipse.jdt.ui.javadocCompletionProcessor
- org.eclipse.jdt.ui.javaEditorTextHovers
- org.eclipse.jdt.ui.javaElementFilters
- org.eclipse.jdt.ui.queryParticipants
- org.eclipse.jdt.ui.quickAssistProcessors
- org.eclipse.jdt.ui.quickFixProcessors

Classpath Variable Initializers

Identifier:

org.eclipse.jdt.core.classpathVariableInitializer

Since:

2.0

Description:

This extension point allows clients to contribute custom classpath variable initializers, which are used to lazily bind classpath variables.

Configuration Markup:

<!ELEMENT extension (classpathVariableInitializer*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT classpathVariableInitializer EMPTY>

<!ATTLIST classpathVariableInitializer

variable CDATA #REQUIRED

class CDATA #REQUIRED>

- **variable –** a unique name identifying the variable for which this initializer will be activated.
- **class –** the class that implements this variable initializer. This class must implement a public subclass of `org.eclipse.jdt.core.ClasspathVariableInitializer` with a public 0–argument constructor.

Examples:

Example of a declaration of a `ClasspathVariableInitializer` for a classpath variable named "FOO":

<extension point=

"org.eclipse.jdt.core.classpathVariableInitializer"

>

<classpathVariableInitializer variable=

"FOO"

class=

"com.example.CPVInitializer"

/>

</extension>

Classpath Container Initializers

Identifier:

org.eclipse.jdt.core.classpathContainerInitializer

Since:

2.0

Description:

This extension point allows clients to contribute custom classpath container initializers, which are used to lazily bind classpath containers to instances of org.eclipse.jdt.core.IClasspathContainer.

Configuration Markup:

<!ELEMENT extension (classpathContainerInitializer*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT classpathContainerInitializer EMPTY>

<!ATTLIST classpathContainerInitializer

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – a unique name identifying all containers for which this initializer will be activated.
- **class** – the class that implements this container initializer. This class must implement a public subclass of `org.eclipse.jdt.core.ClasspathContainerInitializer` with a public 0–argument constructor.

Examples:

Example of a declaration of a `ClasspathContainerInitializer` for a classpath container named "JDK":

<extension point=

"org.eclipse.jdt.core.classpathContainerInitializer"

>

<classpathContainerInitializer id=

"JDK"

class=

"com.example.MyInitializer"

/>

</extension>

Code Formatters

Identifier:

org.eclipse.jdt.core.codeFormatter

Since:

2.0

Description:

This extension point allows clients to contribute new source code formatter implementations.

Configuration Markup:

<!ELEMENT extension (codeFormatter*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT codeFormatter EMPTY>

<!ATTLIST codeFormatter

class CDATA #REQUIRED>

- **class** – the class that defines the code formatter implementation. This class must be a public implementation of `org.eclipse.jdt.core.ICodeFormatter` with a public 0–argument constructor.

Examples:

Example of an implementation of `ICodeFormatter`:

<extension point=

"org.eclipse.jdt.core.codeFormatter"

>

<codeFormatter class=

"com.example.MyCodeFormatter"

/>

</extension>

Java Logical Structures

Identifier:

org.eclipse.jdt.debug.javaLogicalStructures

Since:

3.1

Description:

This extension point allows developers to define a logical structure for Java objects of a specified type. The logical value is created by evaluating the provided code snippet.

Configuration Markup:

<!ELEMENT extension (javaLogicalStructure)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT javaLogicalStructure (variable)>

<!ATTLIST javaLogicalStructure

type CDATA #REQUIRED

subtypes (true | false) "true"

value CDATA #IMPLIED

description CDATA #REQUIRED>

- **type –** Fully qualified name of the type.
- **subtypes –** specify if this Java logical structure should be used also for the objects of a subtype of the specified type, or only for the objects of the specified type. This attribute is optional, the default value is `true`.
- **value –** The code snippet to evaluate to create the logical value. This attribute is optional, if unspecified, the extension must declare one or more variables.

- **description** – a description of this logical structure.

<!ELEMENT variable EMPTY>

<!ATTLIST variable

name CDATA #REQUIRED

value CDATA #REQUIRED>

One variable of the logical value for the object of this type.

- **name** – The name of the variable which will be created
- **value** – The code snippet which will be evaluated as the value of the variable

Examples:

Following is an example of a Java logical structure extension point with two structures:

<extension point=

"org.eclipse.jdt.debug.javaLogicalStructures"

>

<javaLogitalStructure subtypes=

"true"

value=

"return entrySet().toArray();"

type=

"java.util.Map"

/>

<javaLogitalStructure subtypes=

"true"

type=

"java.util.Map$Entry"

>

<variable value=

"return getKey();"

name=

"key"

/>

<variable value=

"return getValue();"

name=

"value"

/>

</javaLogitalStructure>

</extension>

In the example above a Map is translated into its entries and a Map$Entry is translated into its key and value.

API Information:

[Enter API information here.]

Supplied Implementation:

[Enter information about supplied implementation of this extension point.]

VM Install Type UI Page

Identifier:

org.eclipse.jdt.debug.ui.vmInstallTypePage

Description:

This extension point provides a mechanism for contributing UI that will appear in the JRE tab of the launch configuration dialog. The UI is shown only when a VM of the specified install type is selected in the JRE tab.

Configuration Markup:

<!ELEMENT extension (vmInstallTypePage*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT vmInstallTypePage EMPTY>

<!ATTLIST vmInstallTypePage

id CDATA #REQUIRED

vmInstallTypeID CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** specifies a unique identifier for this vm install type UI page.
- **vmInstallTypeID –** specifies VM install type that this UI page is applicable to (corresponds to the id of a VM install type).
- **class –** specifies a fully qualified name of a Java class that implements `ILaunchConfigurationTab`.

Examples:

The following is an example of a VM install type page extension point:

```
<extension point=

"org.eclipse.jdt.debug.ui.vmInstallTypePage"

>

<vmInstallTypePage id=

"com.example.ExampleVMInstallTypePage"

vmInstallTypeID=

"com.example.ExampleVMInstallTypeIdentifier"

class=

"com.example.ExampleVMInstallTypePage"

>

</vmInstallTypePage>

</extension>
```

In the above example, the contributed page will be shown in the JRE tab of the launch configuration dialog whenever the currently selected JRE has a VM Install type identifier of `com.example.ExampleVMInstallTypeIdentifier`.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.ILaunchConfigurationTab**.

JUnit Launch Configurations

Identifier:

org.eclipse.jdt.junit.junitLaunchConfigs

Since:

3.0

Description:

Extension point to register JUnit based launch configurations that need to be updated during refactorings.

Configuration Markup:

<!ELEMENT extension (launchConfigType)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT launchConfigType EMPTY>

<!ATTLIST launchConfigType

configTypeID CDATA #REQUIRED>

- **configTypeID –**

Examples:

The following is an example of a JUnit launch config contribution:

<extension point=

"org.eclipse.jdt.junit.junitLaunchConfigs"

>

<launchConfigType configTypeID=

"com.example.JunitLaunchConfig"

/>

</extension>

Test Run Listeners

Identifier:

org.eclipse.jdt.junit.testRunListeners

Since:

2.1

Description:

Extension point to register additional test run listeners. A test run listeners is notified about the execution of a test run.

Configuration Markup:

<!ELEMENT extension (testRunListener)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

a fully qualified identifier of the target extension point

- **id –** a fully qualified identifier of the target extension point

<!ELEMENT testRunListener EMPTY>

<!ATTLIST testRunListener

class CDATA #REQUIRED>

- **class –** Test run class implementing org.eclipse.jdt.junit.ITestRunListener

Examples:

The following is an example of a test run listener contribution:

<extension point=

"org.eclipse.jdt.junit.testRunListeners"

>

<testRunListener class=

"com.example.SampleTestRunListener"

/>

</extension>

API Information:

Test run listeners must must implement the org.eclipse.jdt.junit.ITestRunListener interface.

Java Runtime Classpath Providers

Identifier:

org.eclipse.jdt.launching.classpathProviders

Since:

2.1

Description:

This extension point allows clients to dynamically compute and resolve classpaths and source lookup paths for Java launch configurations. A Java launch configuration is can be associated with a custom classpath provider via the launch configuration attriubte ATTR_CLASSPATH_PROVIDER and a custom source path provider via the attribute ATTR_SOURCE_PATH_PROVIDER. When specified, the launch configuration attributes correspond to the id of a classpath provider extension.

Configuration Markup:

<!ELEMENT extension (classpathProvider*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT classpathProvider EMPTY>

<!ATTLIST classpathProvider

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** a unique identifier that can be used to reference this classpath provider
- **class –** the class that implements this classpath provider. The class must implement `IRuntimeClasspathProvider`

Examples:

The following is an example of a classpath provider:

<extension point=

"org.eclipse.jdt.launching.classpathProviders"

>

<classpathProvider class=

"com.example.ProviderImplementation"

id=

"com.example.ProviderId"

>

</classpathProvider>

</extension>

Supplied Implementation:

A default implementation is provided for all launch configurations that do not specify a custom classpath provider.

Java Runtime Classpath Entries

Identifier:

org.eclipse.jdt.launching.runtimeClasspathEntries

Since:

3.0

Description:

This in an internal extension point that allows the Java debugger to extend the set of runtime classpath entries used for launching Java applications. Clients are not intended to use this extension point.

Configuration Markup:

<!ELEMENT extension (runtimeClasspathEntry*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT runtimeClasspathEntry EMPTY>

<!ATTLIST runtimeClasspathEntry

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** a unique identifier that can be used to reference this type of runtime classpath entry.
- **class –** the class that implements this runtime classpath entry. The class must implement `IRuntimeClasspathEntry2`.

Examples:

The following is an example of a resolver:

<extension point=

"org.eclipse.jdt.launching.runtimeClasspathEntries"

>

<runtimeClasspathEntry id=

"com.example.EnvVarEntry"

class=

"com.example.EnvVarClasspathEntry"

>

</runtimeClasspathEntry>

</extension>

Supplied Implementation:

An implementations is provided for Java projects, contributed with an identifier of `org.eclipse.jdt.launching.classpathentry.project`. A Java project classpath entry includes all references on its buildpath.

Java Runtime Classpath Entry Resolvers

Identifier:

org.eclipse.jdt.launching.runtimeClasspathEntryResolvers

Description:

This extension point allows clients to dynamically resolve entries used on the runtime classpath and source lookup path, for corresponding classpath variables and classpath containers.

Configuration Markup:

<!ELEMENT extension (runtimeClasspathEntryResolver*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT runtimeClasspathEntryResolver EMPTY>

<!ATTLIST runtimeClasspathEntryResolver

id CDATA #REQUIRED

class CDATA #REQUIRED

variable CDATA #IMPLIED

container CDATA #IMPLIED

runtimeClasspathEntryId CDATA #IMPLIED>

- **id –** a unique identifier that can be used to reference this resolver.
- **class –** the class that implements this resolver. The class must implement `IRuntimeClasspathEntryResolver`.
- **variable –** the name of the classpath variable this resolver is registered for. At least one of variable or container must be specified, and at most one resolver can be registered for a variable or container.
- **container –** the identifier of the classpath container this resolver is registered for. At least one of variable or container must be specified, and at most one resolver can be registered for a variable or container.

- **runtimeClasspathEntryId** – the identifier of the runtime classpath entry this resolver is associated with

Examples:

The following is an example of a resolver:

<extension point=

"org.eclipse.jdt.launching.runtimeClasspathEntryResolvers"

>

<runtimeClasspathEntryResolver class=

"com.example.ResolverImplementation"

id=

"com.example.ResolverId"

variable=

"CLASSPATH_VARIABLE"

>

</runtimeClasspathEntryResolver>

</extension>

Supplied Implementation:

Implementations are provided for the standard JRE_LIB classpath variable and JRE_CONTAINER classpath container.

Java VM Connectors

Identifier:

org.eclipse.jdt.launching.vmConnectors

Description:

This extension point represents different kinds of connections to remote VMs. Each extension must implement `org.eclipse.jdt.launching.IVMConnector`. An `IVMConnector` is responsible for establishing a connection with a remote VM.

Configuration Markup:

<!ELEMENT extension (vmConnector*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT vmConnector EMPTY>

<!ATTLIST vmConnector

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** a unique identifier that can be used to reference this IVMConnector.
- **class –** the class that implements this connector. The class must implement `IVMConnector`.

Examples:

The following is an example of an IVMConnector:

<extension point=

"org.eclipse.jdt.launching.vmConnectors"

>

<vmConnector class=

"com.example.ConnectorImplementation"

id=

"com.example.ConnectorId"

>

</vmConnector>

</extension>

Supplied Implementation:

An implementation of a standard socket attach connector is provided.

Java VM Install Types

Identifier:

org.eclipse.jdt.launching.vmInstallTypes

Description:

This extension point represents different kinds of Java runtime environments and development kits. Each extension must implement `org.eclipse.jdt.launching.IVMInstallType`. An `IVMInstallType` is responsible for creating and managing a set of instances of its corresponding `IVMInstall` class. Through creating different `IVMInstall` objects, an `IVMInstallType` allows for specific behaviour for various Java VMs. A UI for managing `IVMInstalls` is provided by the Java Debug UI plug−in.

Configuration Markup:

<!ELEMENT extension (vmInstallType*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT vmInstallType EMPTY>

<!ATTLIST vmInstallType

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id −** a unique identifier that can be used to reference this IVMInstallType.
- **class −** the class that implements this VM install type. The class must implement IVMInstallType.

Examples:

The following is an example of an IVMInstallType for the J9 VM:

<extension point=

"org.eclipse.jdt.launching.vmInstallTypes"

>

<vmInstallType class=

"org.eclipse.jdt.internal.launching.j9.J9VMInstallType"

id=

"org.eclipse.jdt.internal.launching.j9.J9Type"

>

</vmInstallType>

</extension>

Supplied Implementation:

Abstract implementations of IVMInstall and IVMInstallType are provided. The Java Development Tools Launching Support plug−in defines a VM install type for the standard 1.1.* and 1.2/1.3/1.4 level **JRE**.

Classpath Container Entry Page

Identifier:

org.eclipse.jdt.ui.classpathContainerPage

Description:

This extension point allows to add a wizard page to create or edit a classpath container entry.

Configuration Markup:

<!ELEMENT extension (classpathContainerPage*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT classpathContainerPage EMPTY>

<!ATTLIST classpathContainerPage

id CDATA #REQUIRED

name CDATA #IMPLIED

class CDATA #IMPLIED>

- **id –** identifies the classpath container as defined by `org.eclipse.jdt.core.classpathVariableInitializer`
- **name –** Name of the classpath container used when selecting a new container. This attribute should be a translated string.
- **class –** the name of the class that implements this container page. The class must be public and implement `org.eclipse.jdt.ui.wizards.IClasspathContainerPage` with a public 0–argument constructor.

Examples:

The following is an example of a classpath entry container page:

```
<extension point=
"org.eclipse.jdt.ui.classpathContainerPage"
>
<classpathContainerPage id=
"com.example.myplugin.myContainerId"
name=
"JRE System Libraries"
class=
"com.example.NewJDKEntryPage"
>
</classpathContainerPage>
</extension>
```

Java Folding Structure Provider

Identifier:

org.eclipse.jdt.ui.foldingStructureProviders

Since:

3.0

Description:

Contributions to this extension point define folding structures for the Java editor. That is, they define the regions of a Java source file that can be folded away. See `org.eclipse.jface.text.source.ProjectionViewer` for reference.

Extensions may optionally contribute a preference block which will appear on the Java editor preference page.

Configuration Markup:

<!ELEMENT extension (provider)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT provider EMPTY>

<!ATTLIST provider

id CDATA #REQUIRED

name CDATA #IMPLIED

class CDATA #REQUIRED

preferencesClass CDATA #IMPLIED>

- **id –** The unique identifier of this provider.
- **name –** The name of this provider. If none is given, the id is used instead.
- **class –** An implementation of org.eclipse.jdt.ui.text.folding.IJavaFoldingStructureProvider
- **preferencesClass –** An implementation of org.eclipse.jdt.ui.text.folding.IJavaFoldingPreferenceBlock

Examples:

See `org.eclipse.jdt.internal.ui.text.folding.DefaultJavaFoldingStructureProvider` for an example.

Supplied Implementation:

`org.eclipse.jdt.internal.ui.text.folding.DefaultJavaFoldingStructureProvider` provides the default folding structure for the Java editor.
`org.eclipse.jdt.internal.ui.text.folding.DefaultJavaFoldingPreferenceBlock` provides the preference block for the default structure provider.

Javadoc Completion Processor

Identifier:

org.eclipse.jdt.ui.javadocCompletionProcessor

Description:

This extension point allows to add a Javadoc completion processor to e.g. offer new Javadoc tags.

Configuration Markup:

<!ELEMENT extension (javadocCompletionProcessor*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT javadocCompletionProcessor EMPTY>

<!ATTLIST javadocCompletionProcessor

id CDATA #REQUIRED

name CDATA #IMPLIED

class CDATA #IMPLIED>

- **id –** Unique identifier for the Javadoc completion processor.
- **name –** Localized name of the Javadoc completion processor.
- **class –** The name of the class that implements this Javadoc completion processor. The class must be public and implement `org.eclipse.jdt.ui.text.java.IJavadocCompletionProcessor` with a public 0–argument constructor.

Examples:

The following is an example of a Javadoc completion processor contribution:

```
<extension point=

"org.eclipse.jdt.ui.javadocCompletionProcessor"

>

<javadocCompletionProcessor id=

"XDocletJavadocProcessor"

name=

"XDoclet Javadoc Processor"

class=

"com.example.XDocletJavadocProcessor"

>

</javadocCompletionProcessor>

</extension>
```

Java Editor Text Hovers

Identifier:

org.eclipse.jdt.ui.javaEditorTextHovers

Description:

This extension point is used to plug–in text hovers in a Java editor.

Configuration Markup:

<!ELEMENT extension (hover*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT hover EMPTY>

<!ATTLIST hover

id CDATA #REQUIRED

class CDATA #REQUIRED

label CDATA #IMPLIED

description CDATA #IMPLIED

activate (true | false) "false">

- **id –** the id, typically the same as the fully qualified class name.
- **class –** the fully qualified class name implementing the interface **org.eclipse.jdt.ui.text.java.hover.IJavaEditorTextHover**.
- **label –** the translatable label for this hover.
- **description –** the translatable description for this hover.
- **activate –** if the attribute is set to "true" it will force this plug–in to be loaded on hover activation.

Examples:

The following is an example of a hover definition:

```
<extension point=
"org.eclipse.jdt.ui.javaEditorTextHover"
>
<hover id=
"org.eclipse.example.jdt.internal.debug.ui.JavaDebugHover"
class=
"org.eclipse.example.jdt.internal.debug.ui.JavaDebugHover"
label=
"%javaVariableHover"
/>
</hover>
</extension>
```

Java Element Filter Extensions

Identifier:

org.eclipse.jdt.ui.javaElementFilters

Description:

This extension point is used to extend Java UI views with filters.

Configuration Markup:

<!ELEMENT extension (filter*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT filter EMPTY>

<!ATTLIST filter

id CDATA #IMPLIED

name CDATA #IMPLIED

description CDATA #IMPLIED

targetId CDATA #IMPLIED

enabled (true | false)

pattern CDATA #IMPLIED

class CDATA #IMPLIED>

- **id –** a unique id that will be used to identify this filter.
- **name –** a unique name that allows to identify this filter in the UI. This attribute should be a translated string. Though this attribute is not required for pattern filters (i.e. those using the `pattern` attribute) we suggest to provide a name anyway, otherwise the pattern string itself would be used to represent the filter in the UI.

- **description** – a short description for this filter. This attribute should be a translated string.
- **targetId** – the id of the target where this filter is contributed. If this attribute is missing, then the filter will be contributed to all views which use the `org.eclipse.jdt.ui.actions.customFiltersActionGroup`. This replaces the deprecated attributed "viewId".
- **enabled** – the filter will be enabled if this attribute is present and its value is "true". Most likely the user will be able to override this setting in the UI.
- **pattern** – elements whose name matches this pattern will be hidden. This attribute is here for backward compatibility and should no longer be used. All views that allow to plug–in a filter also allow to add pattern filters directly via UI.
- **class** – the name of the class used to filter the view. The class must extend `org.eclipse.jface.viewers.ViewerFilter`. If this attribute is here then the pattern attribute must not provided.

Examples:

The following is an example of Java element filter definition. It filters out inner classes and is initially selected.

<extension point=

"org.eclipse.jdt.ui.javaElementFilters"

>

<filter id=

"org.eclipse.jdt.ui.PackageExplorer.LibraryFilter"

name=

"%HideReferencedLibraries.label"

description=

"%HideReferencedLibraries.description"

targetId=

"org.eclipse.jdt.ui.PackageExplorer"

class=

"org.eclipse.jdt.internal.ui.filters.LibraryFilter"

enabled=

"false"

>

</filter>

</extension>

Java Query Participants

Identifier:

org.eclipse.jdt.ui.queryParticipants

Since:

3.0

Description:

This extension point allows clients to contribute results to java searches

Configuration Markup:

<!ELEMENT extension (queryParticipant)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT queryParticipant EMPTY>

<!ATTLIST queryParticipant

class CDATA #REQUIRED

id CDATA #REQUIRED

nature CDATA #REQUIRED

name CDATA #REQUIRED>

- **class –** The class that implements this query participant. The class must be public and implement `org.eclipse.jdt.ui.search.IQueryParticipant` with a zero−argument constructor.
- **id –** The unique id of this query participant.
- **nature –** The project nature id this participant should be active for. If the participant should be active for multiple project natures, multiple participants must be defined.
- **name –** A user readeable name for the participant.

Examples:

The following is an example of a query participant contribution:

```
<extension point=
"org.eclipse.jdt.ui.queryParticipants"
>
<queryParticipant label=
"Example Query Participant"
nature=
"org.eclipse.jdt.core.javanature"
class=
"org.eclipse.jdt.ui.example.TestParticipant"
id=
"org.eclipse.jdt.ui.example.TestParticipant"
>
</queryParticipant>
</extension>
```

API Information:

The contributed class must implement `org.eclipse.jdt.ui.search.IQueryParticipant`

Supplied Implementation:

none

Quick Assist Processor

Identifier:

org.eclipse.jdt.ui.quickAssistProcessors

Since:

3.0

Description:

This extension point allows to add a Quick Assist processor to offer new Quick Assists in the Java editor.

Configuration Markup:

<!ELEMENT extension (quickAssistProcessor*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT quickAssistProcessor (enablement)>

<!ATTLIST quickAssistProcessor

id CDATA #REQUIRED

name CDATA #IMPLIED

class CDATA #IMPLIED>

- **id –** Unique identifier for the Quick Assist processor
- **name –** Localized name of the Quick Assist processor.
- **class –** the name of the class that implements this Quick Assist processor. The class must be public and implement `org.eclipse.jdt.ui.text.java.IQuickAssistProcessor` with a public 0–argument constructor.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value −** a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

Since:

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when

evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type –** the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator –** either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The following is an example of a Quick Assist processor contribution:

<extension point=

"org.eclipse.jdt.ui.quickAssistProcessors"

>

<quickAssistProcessor id=

"AdvancedQuickAssistProcessor"

name=

"Advanced Quick Assist Processor"

class=

"com.example.AdvancedQuickAssistProcessor"

>

</quickAssistProcessor>

</extension>

API Information:

The contributed class must implement
`org.eclipse.jdt.ui.text.java.IQuickAssistProcessor`

Since:

Quick Fix Processor

Identifier:

org.eclipse.jdt.ui.quickFixProcessors

Since:

3.0

Description:

This extension point allows to add a Quick Fix processor to offer new Quick Fixes on Java problems.

Configuration Markup:

<!ELEMENT extension (quickFixProcessor*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT quickFixProcessor (enablement)>

<!ATTLIST quickFixProcessor

id CDATA #REQUIRED

name CDATA #IMPLIED

class CDATA #IMPLIED>

- **id –** Unique identifier for the Quick Fix processor
- **name –** Localized name of the Quick Fix processor.
- **class –** the name of the class that implements this Quick Fix processor. The class must be public and implement `org.eclipse.jdt.ui.text.java.IQuickFixProcessor` with a public 0–argument constructor.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** − a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

Since:

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  any number of elements
  ?
  no elements or one element
  +
  one or more elements
  !
  no elements
  integer value
  the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when

evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type –** the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator –** either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The following is an example of a Quick Fix processor contribution:

<extension point=

"org.eclipse.jdt.ui.quickFixProcessors"

>

<quickFixProcessor id=

"AdvancedQuickFixProcessor"

name=

"Advanced Quick Fix Processor"

class=

"com.example.AdvancedQuickFixProcessor"

>

</quickFixProcessor>

</extension>

API Information:

The contributed class must implement `org.eclipse.jdt.ui.text.java.IQuickFixProcessor`

Since:

Other Reference Information

The following specifications, white papers, and design notes describe various aspects of the Java development tooling.

- Map of Eclipse Java Development Tooling Plug−ins

JDT Overview – Map of JDT Plug–ins

The Eclipse Java development tooling provides a comprehensive Java development environment.

The Java development tooling itself is divided up into a number of separate plug–ins. The following table shows which API packages are found in which plug–ins as of Eclipse 3.0. This table is useful for determining which plug–ins a given plug–in should include as prerequisites.

API Package	**Required plug–in id**

`com.sun.jdi[.*]`
`org.eclipse.jdi[.*]`
`org.eclipse.jdt.debug[.*]`
	`org.eclipse.jdt.debug`

`org.eclipse.jdt.debug.ui[.*]`	`org.eclipse.jdt.debug.ui`
`org.eclipse.jdt.core[.*]`	`org.eclipse.jdt.core`
`org.eclipse.jdt.launching[.*]`	`org.eclipse.jdt.launching`
`org.eclipse.jdt.ui[.*]`	`org.eclipse.jdt.ui`
`org.eclipse.jdt.junit[.*]`	`org.eclipse.jdt.junit`
`junit.*`	`org.junit`

Installing the examples

To install the examples, download the zip file containing the examples.

The workbench should not be running while the examples are being installed. Extract the contents of the zip file to the root directory of your Eclipse installation.

For example, if you installed the Eclipse Project SDK on **d:\eclipse−sdk** then extract the contents of the examples zip file to **d:\eclipse−sdk**.

Start the workbench. The example plug−ins should be installed.

Java Example Projects

Introduction

The Java examples provide you with sample code for exploring the Eclipse Java tools.

Loading the Samples

1. Open the New wizard.
2. Select Java in the Examples category.
3. Select the project to be added to your workspace.

The wizard creates a new Java project for you and imports the sample code into this project.

JDT Questions Index

JDT Core

- What is available in the JDT Core API Packages?
- How do I launch a Java program from the platform?
- How do I programmatically compile a Java program?
- How do I setup a project's classpath?
- How do I manipulate Java code?
- How do I use the Java search engine?
- What are the JDT Core options?
- How do I programmatically use CodeAssist and CodeSelect?
- How are Java projects, folders, and files different from regular resources?

JDT UI

- What is available in the JDT UI packages?
- How do I programmatically open a Java editor and display a specific Java element in the editor?
- How do I programmatically open the Open Type dialog?
- How do I present Java elements in a standard JFace viewer?
- How do I write a Jar file?
- How do I create a customized new Java element wizard page?

Running a Java program

The JDT Debug component includes facilities for launching a Java program using the VM install that is currently configured by the user for a Java project.

Launching a compiled Java program

Java programs that have been compiled in a Java project can be run by getting the appropriate **IVMRunner** for the Java project and running the class by name. The following code snippet shows how the class **MyClass** inside **myJavaProject** can be launched.

```
IVMInstall vmInstall = JavaRuntime.getVMInstall(myJavaProject);
if (vmInstall == null)
   vmInstall = JavaRuntime.getDefaultVMInstall();
if (vmInstall != null) {
   IVMRunner vmRunner = vmInstall.getVMRunner(ILaunchManager.RUN_MODE);
   if (vmRunner != null) {
      String[] classPath = null;
      try {
         classPath = JavaRuntime.computeDefaultRuntimeClassPath(myJavaProject);
      } catch (CoreException e) { }
      if (classPath != null) {
         VMRunnerConfiguration vmConfig =
            new VMRunnerConfiguration("MyClass", classPath);
         ILaunch launch = new Launch(null, ILaunchManager.RUN_MODE, null);
         vmRunner.run(vmConfig, launch, null);
      }
   }
```

```
}
```

Another way to launch a Java program is to create a **Java application** launch configuration, and launch it. The following snippet shows how the class **MyClass** inside **myJavaProject** can be launched using a simple launch configuration. By default, the resulting running application uses the JRE and classpath associated with **myJavaProject**.

```
ILaunchManager manager = DebugPlugin.getDefault().getLaunchManager();
ILaunchConfigurationType type = manager.getLaunchConfigurationType(IJavaLaunchConfigurationCon
ILaunchConfigurationWorkingCopy wc = type.newInstance(null, "SampleConfig");
wc.setAttribute(IJavaLaunchConfigurationConstants.ATTR_PROJECT_NAME, "myJavaProject");
wc.setAttribute(IJavaLaunchConfigurationConstants.ATTR_MAIN_TYPE_NAME, "myClass");
ILaunchConfiguration config = wc.doSave();
config.launch(ILaunchManager.RUN_MODE, null);
```

Compiling Java code

The JDT plug−ins include an incremental and batch Java compiler for building Java .class files from source code. There is no direct API provided by the compiler. It is installed as a builder on Java projects. Compilation is triggered using standard platform build mechanisms.

The platform build mechanism is described in detail in Incremental project builders .

Compiling code

You can programmatically compile the Java source files in a project using the build API.

```
IProject myProject;
IProgressMonitor myProgressMonitor;
myProject.build(IncrementalProjectBuilder.INCREMENTAL_BUILD, myProgressMonitor);
```

For a Java project, this invokes the Java incremental project builder (along with any other incremental project builders that have been added to the project's build spec). The generated .class files are written to the designated output folder. Additional resource files are also copied to the output folder.

In the case of a full batch build, all the .class files in the output folder may be 'scrubbed' to ensure that no stale files are found. This is controlled using a JDT Core Builder Option (CORE_JAVA_BUILD_CLEAN_OUTPUT_FOLDER). The default for this option is to clean output folders. Unless this option is reset, you must ensure that you place all .class files for which you do not have corresponding source files in a separate class file folder on the classpath instead of the output folder.

The incremental and batch builders can be configured with other options that control which resources are copied to the output folder. The following sample shows how to set up a resource filter so that files ending with '.ignore' and folders named 'META−INF', are not copied to the output folder:

```
Hashtable options = JavaCore.getOptions();
options.put(JavaCore.CORE_JAVA_BUILD_RESOURCE_COPY_FILTER, "*.ignore,META-INF/");
JavaCore.setOptions(options);
```

Filenames are filtered if they match one of the supplied patterns. Entire folders are filtered if their name matches one of the supplied folder names which end in a path separator.

The incremental and batch builders can also be configured to only generate a single error when the .classpath file has errors. This option is set by default and eliminates numerous errors. See JDT Core Builder Options for a complete list of builder–related options and their defaults.

The compiler can also be configured using **JavaCore** options. For example, you can define the severity that should be used for different kinds of problems that are found during compilation. See JDT Core Compiler Options for a complete list of compiler–related options and their defaults.

When programmatically configuring options for the builder or compiler, you should determine the scope of the option. For example, setting up a resource filter may only apply to a particular project. The following example sets up the same resource filter shown earlier, but sets it only the individual project.

```
Hashtable options = myProject.getOptions(false);  // get only the options set up in this proje
options.put(JavaCore.CORE_JAVA_BUILD_RESOURCE_COPY_FILTER, "*.ignore,META-INF/");
myProject.setOptions(options);
```

Using the batch compiler

Finding the batch compiler

The batch compiler class is located in the internal classes of the JDT/Core plug–in. So it is in the *jdtcore.jar* file in the directory *plugins/org.eclipse.jdt.core*. The name of the class is *org.eclipse.jdt.internal.compiler.batch.Main*.

Running the batch compiler

- Using the main method.

  Using the main method. The Main class has a main method. This is the classical way to invoke the batch compiler on a command–line.

  - For example on a command–line:

    ```
    java -classpath org.eclipse.jdt.core_3.1.0.jar
    org.eclipse.jdt.internal.compiler.batch.Main -classpath rt.jar
    A.java
    ```

    or:

    ```
    java -jar org.eclipse.jdt.core_3.1.0.jar -classpath rt.jar
    A.java
    ```

  - For example in a java source:

    ```
    org.eclipse.jdt.internal.compiler.batch.Main.main(new String[]
    {"-classpath", "rt.jar", "A.java"});
    ```

- Using the static compile(String) method.

  The `compile(String)` method is a convenient method to invoke the batch compiler in a java application. Instead of `org.eclipse.jdt.internal.compiler.batch.Main.main(new String[] {"-classpath", "rt.jar", "A.java"});` you can simply write `org.eclipse.jdt.internal.compiler.batch.Main.compile("-classpath`

```
rt.jar A.java");
```

Which options are available?

With the orange background, these are suggested options.

Name	Usage
Classpath options	
–bootclasspath <dir 1>;<dir 2>;...;<dir P>	This is a list of directory or jar files used to bootstrap the class files used by the compiler. By default the libraries of the running VM are used. Entries are separated by the platform path separator.
–cp –classpath <dir 1>;<dir 2>;...;<dir P>	This is a list of directory or jar files used to compile the source files. The default value is the value of the property "java.class.path". Entries are separated by the platform path separator.
–extdirs <dir 1>;<dir 2>;...;<dir P>	This is a list of directory used to specify the location of extension zip/jar files. Entries are separated by the platform path separator.
–sourcepath <dir 1>;<dir 2>;...;<dir P>	This is a list of directory used to specify the source files. Entries are separated by the platform path separator.
–d <dir 1>|none	This is used to specify in which directory the generated .class files should be dumped. If it is omitted, no package directory structure is created. If you don't want to generate .class files, use –d none.
–encoding <encoding name>	Specify default source encoding format (custom encoding can also be specified on a per file basis by suffixing each input source file/folder name with [encoding <encoding name>]).
Compliance options	
–target 1.1|1.2|1.3|1.4|1.5|5|5.0	This specifies the .class file target setting. The possible value are: • 1.1 (major version: 45 minor: 3) • 1.2 (major version: 46 minor: 0) • 1.3 (major version: 47 minor: 0) • 1.4 (major version: 48 minor: 0) • 1.5, 5 or 5.0 (major version: 49 minor: 0) Defaults are: • 1.1 in –1.3 mode • 1.2 in –1.4 mode • 1.5 in –1.5 mode
–1.3	Set compliance level to 1.3. Implicit –source 1.3 –target 1.1.
–1.4	Set compliance level to 1.4 (default). Implicit –source 1.3 –target 1.2.
–1.5	Set compliance level to 1.5. Implicit –source 1.5 –target 1.5.
–source 1.3|1.4|1.5|5|5.0	This is used to enable the source level of the compiler. The possible value are: • 1.3

	• 1.4 • 1.5, 5 or 5.0 Defaults are: • 1.3 in −1.3 mode • 1.4 in −1.4 mode • 1.5 in −1.5 mode In 1.4, *assert* is treated as a keyword. In 1.5, *enum* and *assert* are treated as a keyword.

Warning options

−warn:
 	
allDeprecation
allJavadoc
assertIdentifier
boxing
charConcat
conditionAssign
constructorName
dep−ann
deprecation
emptyBlock
enumSwitch
fieldHiding
finalBound
finally
hiding
incomplete−switch
indirectStatic
intfAnnotation
intfNonInherited
javadoc
localHiding
maskedCatchBlocks
nls
noEffectAssign
null
over−ann
pkgDefaultMethod
semicolon
serial
specialParamHiding
static−access
staticReceiver
suppress
synthetic−access
syntheticAccess
tasks(<task1>|...|<taskN>)
typeHiding
	Set warning level.
	e.g. −warn:unusedLocals,deprecation
	
	In red are the default settings.
	
	`-warn:<warnings separated by ,>  enable exactly the listed warnings`
	`-warn:+<warnings separated by ,>  enable additional warnings`
	`-warn:-<warnings separated by ,>  disable specific warnings`
	
	allDeprecation deprecation even inside deprecated code
	allJavadoc invalid or missing javadoc
	assertIdentifier occurrence of *assert* used as identifier
	boxing autoboxing conversion
	charConcat when a char array is used in a string concatenation without being converted explicitly to a string
	conditionAssign possible accidental boolean assignment
	constructorName method with constructor name
	dep−ann missing @Deprecated annotation
	deprecation usage of deprecated type or member outside deprecated code
	emptyBlock undocumented empty block
	enumSwitch, incomplete−switch incomplete enum switch
	fieldHiding field hiding another variable
	finalBound type parameter with final bound
	finally finally block not completing normally
	hiding macro for fieldHiding, localHiding, typeHiding and maskedCatchBlock
	indirectStatic indirect reference to static member
	intfAnnotation annotation type used as super interface
	intfNonInherited interface non−inherited method compatibility
	javadoc invalid javadoc
	localHiding local variable hiding another variable
	maskedCatchBlocks hidden catch block

unchecked unnecessaryElse unqualified−field−access unqualifiedField uselessTypeCheck unused unusedArgument unusedImport unusedLocal unusedPrivate unusedThrown varargsCast warningToken	**nls**	non−nls string literals (lacking of tags //$NON−NLS−<n>)
	noEffectAssign	for assignment with no effect
	null	missing or redundant null check
	over−ann	missing @Override annotation
	pkgDefaultMethod	attempt to override package−default method
	serial	missing serialVersionUID
	semicolon	unnecessary semicolon or empty statement
	specialParamHiding	constructor or setter parameter hiding another field
	static−access	macro for indirectStatic and staticReceiver
	staticReceiver	if a non static receiver is used to get a static field or call a static method
	suppress	enable @SuppressWarnings
	syntheticAccess, synthetic−access	when performing synthetic access for innerclass
	tasks	enable support for tasks tags in source code
	typeHiding	type parameter hiding another type
	unchecked	unchecked type operation
	unnecessaryElse	unnecessary else clause
	unqualified−field−access, unqualifiedField	unqualified reference to field
	unused	macro for unusedArgument, unusedImport, unusedLocal, unusedPrivate and unusedThrown
	unusedArgument	unused method argument
	unusedImport	unused import reference
	unusedLocal	unused local variable
	unusedPrivate	unused private member declaration
	unusedThrown	unused declared thrown exception
	uselessTypeCheck	unnecessary cast/instanceof operation
	varargsCast	varargs argument need explicit cast
	warningToken	unhandled warning token in @SuppressWarningsb
−nowarn	No warning (equivalent to −warn:none)	
−deprecation	Equivalent to −warn:deprecation.	
Debug options		
−g[:nonel:lines,vars,source]	Set the debug attributes level **−g** All debug info (equivalent to −g:lines,vars,source) **−g:none** No debug info **−g:[lines,vars,source]** Selective debug info	
−preserveAllLocals	Explicitly request the compiler to preserve all local variables (for debug purpose). If omitted, the compiler will removed unused locals.	

Advanced options	
@<file>	Read command–line arguments from file
–maxProblems <n>	Max number of problems per compilation unit (100 by default)
–log <filename>	Specify a log file in which all output from the compiler will be dumped. This is really useful if you want to debug the batch compiler or get a file which contains all errors and warnings from a batch build. If the extension is **.xml**, the generated log will be a xml file.
–proceedOnError	Keep compiling when error, dumping class files with problem methods or problem types. This is recommended only if you want to be able to run your application even if you have remaining errors.
–verbose	Print accessed/processed compilation units in the console or the log file if specified.
–referenceInfo	Compute reference info. This is useful only if connected to the builder. The reference infos are useless otherwise.
–progress	Show progress (only in –log mode)
–time	Display speed information
–noExit	Do not call System.exit(n) at end of compilation (n=0 if no error)
–repeat <n>	Repeat compilation process <n> times (perf analysis).
–inlineJSR	Inline JSR bytecode (implicit if target >= 1.5)
–enableJavadoc	Consider references inside javadoc
Helping options	
–? –help	Display the help message
–v –version	Display the build number of the compiler. This is very useful to report a bug.
–showversion	Display the build number of the compiler and continue. This is very useful to report a bug.

Examples

`d:\temp -classpath rt.jar -time -g -d d:/tmp`	It compiles all source files in d:\temp and its subfolders. The classpath is simply rt.jar. It generates all debug attributes and all generated .class files are dumped in d:\tmp. The speed of the compiler will be displayed once the batch process is completed.
`d:\temp\Test.java -classpath d:\temp;rt.jar -g:none`	It compiles only Test.java and it will retrieve any dependant files from d:\temp. The classpath is rt.jar and d:\temp, which means that all necessary classes are searched first in d:\temp and then in rt.jar. It generates no debug attributes and all generated .class files are dumped in d:\tmp.

Using the ant javac adapter

The Eclipse compiler can be used inside an Ant script using the javac adapter. In order to use the Eclipse compiler, you simply need to define the **build.compiler** property in your script. Here is a small example.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="compile" default="main" basedir="../.">

<property name="build.compiler" value="org.eclipse.jdt.core.JDTCompilerAdapter"/>
```

```
        <property name="root" value="${basedir}/src"/>

        <property name="destdir" value="d:/temp/bin" />

        <target name="main">
                <javac srcdir="${root}" destdir="${destdir}" debug="on" nowarn="on" extdirs="d:/e
                    <classpath>
                      <pathelement location="${basedir}/../org.eclipse.jdt.core/bin"/>
                    </classpath>
                </javac>
        </target>
</project>
```

The syntax used for the javac Ant task can be found in the Ant javac task documentation. The current adapter supports the Javac Ant task 1.4.1 up to 1.6.5 versions.

If you are using a version above 1.5.0, you can use the nested compiler argument element to specify compiler specific options.

```
...
<javac srcdir="${root}" destdir="${destdir}" debug="on" nowarn="on" extdirs="d:/extdirs" source="
    <classpath>
      <pathelement location="${basedir}/../org.eclipse.jdt.core/bin"/>
    </classpath>
    <compilerarg compiler="org.eclipse.jdt.core.JDTCompilerAdapter" line="-1.5 -warn:+boxing"/>
</javac>
...
```

To prevent from getting compiler dependant scripts, we advice you to use the compiler argument set to `org.eclipse.jdt.core.JDTCompilerAdapter`. If this is not set, the script can only be used with the Eclipse compiler. If set, the nested compiler argument is ignored if the name is different from the compiler name specified by the `build.compiler` property.

Problem determination

JDT Core defines a specialized marker (marker type "**org.eclipse.jdt.core.problem** ") to denote compilation problems. To programmatically discover problems detected by the compiler, the standard platform marker protocol should be used. See Resource Markers for an overview of using markers.

The following snippet finds all Java problem markers in a compilation unit.

```
   public IMarker[] findJavaProblemMarkers(ICompilationUnit cu)
      throws CoreException {
      IResource javaSourceFile = cu.getUnderlyingResource();
      IMarker[] markers =
         javaSourceFile.findMarkers(IJavaModelMarker.JAVA_MODEL_PROBLEM_MARKER,
            true, IResource.DEPTH_INFINITE);
   }
```

Java problem markers are maintained by the Java project builder and are removed automatically as problems are resolved and the Java source is recompiled.

The problem id value is set by one of the constants in **IProblem** . The problem's id is reliable, but the message is localized and therefore can be changed according to the default locale. The constants defined in **IProblem** are self−descriptive.

An implementation of **IProblemRequestor** should be defined to collect the problems discovered during a Java operation. Working copies can be reconciled with problem detection if a **IProblemRequestor** has been supplied for the working copy creation. To achieve this, you can use the **reconcile** method. Here is an example:

```
ICompilationUnit unit = ..; // get some compilation unit

// create requestor for accumulating discovered problems
IProblemRequestor problemRequestor = new IProblemRequestor() {
  public void acceptProblem(IProblem problem) {
    System.out.println(problem.getID() + ": " + problem.getMessage());
  }
  public void beginReporting() {}
  public void endReporting() {}
  public boolean isActive() { return true; } // will detect problems if active
};

// use working copy to hold source with error
ICompilationUnit workingCopy = unit.getWorkingCopy(new WorkingCopyOwner() {}, problemRequestor,
((IOpenable)workingCopy).getBuffer().setContents("public class X extends Zork {}");

// trigger reconciliation
workingCopy.reconcile(NO_AST, true, null, null);
```

You can add an action on the reported problems in the acceptProblem(IProblem) method. In this example, the reported problem will be that **Zork cannot be resolved or is not a valid superclass** and its id is **IProblem.SuperclassNotFound**.

Setting the Java build path

This section describes how to set the Java build path. The build path is the classpath that is used for building a Java project (**IJavaProject**).

A classpath is simply an array of classpath entries (**IClassPathEntry**) that describe the types that are available. The types can appear in source or binary form and the ordering of the entries on the path defines the lookup order for resolving types during a build.

The Java build path is reflected in the structure of a Java project element. You can query a project for its package fragment roots (**IPackageFragmentRoot**). Each classpath entry maps to one or more package fragment roots, each of which further contains a set of package fragments.

This discussion of the build path does not involve the Java runtime path, which can be defined separately from the build path. (See **Running Java code** for a discussion of the runtime classpath.

Changing the build path

You can programmatically change a project's build path using **setRawClasspath** on the corresponding project's Java element. The following code sets the classpath for a project resource:

```
IProject project = ... // get some project resource
IJavaProject javaProject = JavaCore.create(project);
IClasspathEntry[] newClasspath = ...;
javaProject.setRawClasspath(newClasspath, someProgressMonitor);
```

(Note: The use of the term "raw" classpath is used to emphasize the fact that any variables used to describe entry locations have not been resolved.)

The Java build path is persisted into a file named '.classpath' in the project's file structure. The purpose of this file is to provide a way to share Java build path settings with others through some source code repository. In particular, this file should not be manually edited, since it may get corrupted.

Classpath entries

Classpath entries can be defined using factory methods defined on **JavaCore**. Classpath entries can reference any of the following:

- **a source folder –** a folder containing source compilation units organized under their corresponding package directory structure. Source folders are used to better structure source files in large projects, and may only be referenced within the containing project. The corresponding factory method is **newSourceEntry**. Inside a given source folder, each compilation unit is expected to be nested in the appropriate folder structure according to its package statement. For example, compilation unit 'X.java' in package 'p1' must be located inside sub−folder 'p1' of a source folder. It is possible to use multiple source folders, as long as they don't overlap. A source folder may be assigned its own output location which determines where generated class files should be placed. If none is specified, then class files will be placed in the containing project's output location (see **IJavaProject.setOutputLocation**).

  The following is an example classpath entry that denotes the source folder 'src' of project 'MyProject':

  ```
  IClassPathEntry srcEntry = JavaCore.newSourceEntry(new Path("/MyProject/src"));
  ```

- **a binary library –** either a class file folder (contained inside the workspace) or a class file archive file (contained inside or outside the workspace). Archive libraries can have attached source archives, which are extracted when asking a class file element for its source (**getSource**). The factory method for libraries is **newLibraryEntry**.

  The following is an example classpath entry that denotes the class file folder 'lib' of 'MyProject':

  ```
  IClassPathEntry libEntry = JavaCore.newLibraryEntry(
    new Path("/MyProject/lib"),
    null, //no source
    null, //no source
    false); //not exported
  ```

  The following classpath entry has a source attachment:

  ```
  IClassPathEntry libEntry = JavaCore.newLibraryEntry(
    new Path("d:/lib/foo.jar"), // library location
    new Path("d:/lib/foo_src.zip"), //source archive location
    new Path("src"), //source archive root path
    true); //exported
  ```

  The source archive root path describes the location of the root within the source archive. If set to null, the root of the archive will be inferred dynamically.

- **a prerequisite project –** another Java project. A prerequisite project always contributes its source folders to dependent projects. It can also optionally contribute any of its classpath entries which are tagged as exported (see factory methods supporting the extra boolean argument 'isExported'). This means that in addition to contributing its source to its dependents, a project will also export all classpath entries tagged as such. This allows prerequisite projects to better hide their own structure changes. For example, a given project may choose to switch from using a source folder to exporting a library. This can be done without requiring its dependent projects to change their classpath. The factory method for a project prerequisite is **newProjectEntry**.

  The following classpath entry denotes a prerequisite project 'MyFramework'.

```
IClassPathEntry prjEntry = JavaCore.newProjectEntry(new Path("/MyFramework"), true); //e
```

- **an indirect reference to a project or library, using some classpath variable –** The location of projects or libraries can be dynamically resolved relative to a classpath variable, which is specified as the first segment of the entry path. The rest of the entry path is then appended to the resolved variable path. The factory method for a classpath variable is **newVariableEntry**. Classpath variables are global to the workspace, and can be manipulated through JavaCore methods **getClasspathVariable** and **setClasspathVariable**.

  It is possible to register an automatic **classpath variable initializer** which is invoked through the extension point **org.eclipse.jdt.core.classpathVariableInitializer** when the workspace is started.

  The following classpath entry denotes a library whose location is kept in the variable 'HOME'. The source attachment is defined using the variables 'SRC_HOME' and 'SRC_ROOT' :

```
IClassPathEntry varEntry = JavaCore.newVariableEntry(
  new Path("HOME/foo.jar"), // library location
  new Path("SRC_HOME/foo_src.zip"), //source archive location
  new Path("SRC_ROOT"), //source archive root path
  true); //exported
JavaCore.setClasspathVariable("HOME", new Path("d:/myInstall"), null); // no progress
```

- **entry denoting a classpath container –** an indirect reference to a structured set of project or libraries. Classpath containers are used to refer to a set of classpath entries that describe a complex library structure. Like classpath variables, classpath containers **(IClasspathContainer)** are dynamically resolved. Classpath containers may be used by different projects, causing their path entries to resolve to distinct values per project. They also provide meta information about the library that they represent (name, kind, description of library.) Classpath containers can be manipulated through JavaCore methods **getClasspathContainer** and **setClasspathContainer**.

  It is possible to register an automatic **classpath container initializer** which is lazily invoked through the extension point **org.eclipse.jdt.core.classpathContainerInitializer** when the container needs to be bound.

  The following classpath entry denotes a system class library container:

```
IClassPathEntry varEntry = JavaCore.newContainerEntry(
  new Path("JDKLIB/default"), // container 'JDKLIB' + hint 'default'
  false); //not exported

JavaCore.setClasspathContainer(
  new Path("JDKLIB/default"),
  new IJavaProject[]{ myProject }, // value for 'myProject'
```

```
new IClasspathContainer[] {
  new IClasspathContainer() {
    public IClasspathEntry[] getClasspathEntries() {
      return new IClasspathEntry[]{
        JavaCore.newLibraryEntry(new Path("d:/rt.jar"), null, null, false);
      };
    }
    public String getDescription() { return "Basic JDK library container"; }
    public int getKind() { return IClasspathContainer.K_SYSTEM; }
    public IPath getPath() { return new Path("JDKLIB/basic"); }
  }
},
null);
```

Exclusion patterns

A classpath source entry may be assigned an exclusion pattern, which prevents certain resources in a source folder from being visible on the classpath. Using a pattern allows specified portions of the resource tree to be filtered out. Each exclusion pattern path is relative to the classpath entry and uses a pattern mechanism similar to Ant. Exclusion patterns can be used to specify nested source folders as long as the outer pattern excludes the inner pattern.

See **getExclusionPatterns()** for more detail on exclusion patterns.

The Java project API **isOnClasspath** checks both inclusion and exclusion patterns before determining whether a particular resource is on the classpath.

Remarks:

- Exclusion patterns have higher precedence than inclusion patterns; in other words, exclusion patterns can remove files for the ones that are to be included, not the other way around.
- A nested source folder excluded from build path can be set as an output location. The following is an example classpath entry that denotes the source folder 'src' of project 'MyProject' with an excluded nested source folder used as an output location:

```
IPath sourceFolder = new Path("/MyProject/src");
IPath outputLocation = sourceFolder.append("bin");
IClassPathEntry srcEntry = JavaCore.newSourceEntry(
  sourceFolder, // source folder location
  new Path[] { outputLocation }, // excluded nested folder
  outputLocation); // output location
```

Inclusion patterns

A classpath source entry may also be assigned an inclusion pattern, which explicitly defines resources to be visible on the classpath. When no inclusion patterns are specified, the source entry includes all relevant files in the resource tree rooted at this source entry's path. Specifying one or more inclusion patterns means that only the specified portions of the resource tree are to be included. Each path specified must be a relative path, and will be interpreted relative to this source entry's path. File patterns are case−sensitive. A file matched by one or more of these patterns is included in the corresponding package fragment root unless it is excluded by one or more of this entry's exclusion patterns.

See **getExclusionPatterns()** for a discussion of the syntax and semantics of path patterns. The absence of any inclusion patterns is semantically equivalent to the explicit inclusion pattern `**`.

The Java project API **isOnClasspath** checks both inclusion and exclusion patterns before determining whether a particular resource is on the classpath.

Examples:

- The inclusion pattern `src/**` by itself includes all files under a root folder named `src`.
- The inclusion patterns `src/**` and `tests/**` includes all files under the root folders named `src` and `tests`.
- The inclusion pattern `src/**` together with the exclusion pattern `src/**/Foo.java` includes all files under a root folder named `src` except for ones named `Foo.java`.

Classpath resolution

Since classpath variables and containers allow you to define dynamically bound classpath entries, the classpath API distinguishes between a raw and a resolved classpath. The raw classpath is the one originally set on the Java project using **setRawClasspath**, and can be further queried by asking the project for **getRawClasspath**. The resolved classpath can be queried using **getResolvedClasspath**. This operation triggers initialization of any variables and containers necessary to resolve the classpath. Many Java Model operations implicitly cause the Java build path to be resolved. For example, computing a project's package fragment roots requires the build path to be resolved.

Manipulating Java code

Your plug−in can use the JDT API to create classes or interfaces, add methods to existing types, or alter the methods for types.

The simplest way to alter Java objects is to use the Java element API. More general techniques can be used to work with the raw source code for a Java element.

Code modification using Java elements

Generating a compilation unit

The easiest way to programmatically generate a compilation unit is to use **IPackageFragment.createCompilationUnit**. You specify the name and contents of the compilation unit. The compilation unit is created inside the package and the new **ICompilationUnit** is returned.

A compilation unit can be created generically by creating a file resource whose extension is "**.java"** in the appropriate folder that corresponds to the package directory. Using the generic resource API is a back door to the Java tooling, so the Java model is not updated until the generic resource change listeners are notified and the JDT listeners update the Java model with the new compilation unit.

Modifying a compilation unit

Most simple modifications of Java source can be done using the Java element API.

For example, you can query a type from a compilation unit. Once you have the **IType**, you can use protocols such as **createField**, **createInitializer**, **createMethod**, or **createType** to add source code members to the type. The source code and information about the location of the member is supplied in these methods.

The **ISourceManipulation** interface defines common source manipulations for Java elements. This includes methods for renaming, moving, copying, or deleting a type's member.

Working copies

Code can be modified by manipulating the compilation unit (and thus the underlying **IFile** is modified) or one can modify an in−memory copy of the compilation unit called a working copy.

A working copy is obtained from a compilation unit using the **getWorkingCopy** method. (Note that the compilation unit does not need to exist in the Java model in order for a working copy to be created.) Whoever creates such a working copy is responsible for discarding it when not needed any longer using the **discardWorkingCopy** method.

Working copies modify an in−memory buffer. The **getWorkingCopy()** method creates a default buffer, but clients can provide their own buffer implementation using the **getWorkingCopy(WorkingCopyOwner, IProblemRequestor, IProgressMonitor)** method. Clients can manipulate the text of this buffer directly. If they do so, they must synchronize the working copy with the buffer from time to time using either the **reconcile(int, boolean, WorkingCopyOwner, IProgressMonitor)** method.

Finally a working copy can be saved to disk (replacing the original compilation unit) using the **commitWorkingCopy** method.

For example the following code snippet creates a working copy on a compilation unit using a custom working copy owner. The snippet modifies the buffer, reconciles the changes, commits the changes to disk and finally discards the working copy.

```
// Get original compilation unit
ICompilationUnit originalUnit = ...;

// Get working copy owner
WorkingCopyOwner owner = ...;

// Create working copy
ICompilationUnit workingCopy = originalUnit.getWorkingCopy(owner, null, null);

// Modify buffer and reconcile
IBuffer buffer = ((IOpenable)workingCopy).getBuffer();
buffer.append("class X {}");
workingCopy.reconcile(NO_AST, false, null, null);

// Commit changes
workingCopy.commitWorkingCopy(false, null);

// Destroy working copy
workingCopy.discardWorkingCopy();
```

Working copies can also be shared by several clients using a working copy owner. A working copy can be later retrieved using the **findWorkingCopy** method. A shared working copy is thus keyed on the original compilation unit and on a working copy owner.

The following shows how client 1 creates a shared working copy, client 2 retrieves this working copy, client 1 discards the working copy, and client 2 trying to retrieve the shared working copy notices it does not exist any longer:

```
// Client 1 & 2: Get original compilation unit
ICompilationUnit originalUnit = ...;

// Client 1 & 2: Get working copy owner
WorkingCopyOwner owner = ...;

// Client 1: Create shared working copy
ICompilationUnit workingCopyForClient1 = originalUnit.getWorkingCopy(owner, null, null);

// Client 2: Retrieve shared working copy
ICompilationUnit workingCopyForClient2 = originalUnit.findWorkingCopy(owner);

// This is the same working copy
assert workingCopyForClient1 == workingCopyForClient2;

// Client 1: Discard shared working copy
workingCopyForClient1.discardWorkingCopy();

// Client 2: Attempt to retrieve shared working copy and find out it's null
workingCopyForClient2 = originalUnit.findWorkingCopy(owner);
assert workingCopyForClient2 == null;
```

Code modification using the DOM/AST API

There are three ways to create a **CompilationUnit**. The first one is to use **ASTParser**. The second is to use **ICompilationUnit#reconcile(...)**. The third is to start from scratch using the factory methods on **AST** (Abstract Syntax Tree).

Creating an AST from existing source code

An instance of **ASTParser** must be created with ASTParser.newParser(int).

The source code is given to the **ASTParser** with one of the following methods:

- setSource(char[]): to create the AST from source code
- setSource(IClassFile): to create the AST from a classfile
- setSource(ICompilationUnit): to create the AST from a compilation unit

Then the AST is created by calling createAST(IProgressMonitor).

The result is an AST with correct source positions for each node. The resolution of bindings has to be requested before the creation of the tree with setResolveBindings(boolean). Resolving the bindings is a costly operation and should be done only when necessary. As soon as the tree has been modified, all positions and bindings are lost.

Creating an AST by reconciling a working copy

If a working copy is not consistent (has been modified) then an AST can be created by calling the method reconcile(int, boolean, WorkingCopyOwner, IProgressMonitor). To request AST creation, call the reconcile(...) method with AST.JLS2 as first parameter.

Its bindings are computed only if the problem requestor is active, or if the problem detection is forced. Resolving the bindings is a costly operation and should be done only when necessary. As soon as the tree has been modified, all positions and bindings are lost.

From scratch

It is possible to create a **CompilationUnit** from scratch using the factory methods on **AST**. These method names start with **new...**. The following is an example that creates a **HelloWorld** class.

The first snippet is the generated output:

```
package example;
import java.util.*;
public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello" + " world");
        }
}
```

The following snippet is the corresponding code that generates the output.

```
AST ast = new AST();
CompilationUnit unit = ast.newCompilationUnit();
PackageDeclaration packageDeclaration = ast.newPackageDeclaration();
packageDeclaration.setName(ast.newSimpleName("example"));
unit.setPackage(packageDeclaration);
ImportDeclaration importDeclaration = ast.newImportDeclaration();
QualifiedName name =
        ast.newQualifiedName(
                ast.newSimpleName("java"),
                ast.newSimpleName("util"));
importDeclaration.setName(name);
importDeclaration.setOnDemand(true);
unit.imports().add(importDeclaration);
TypeDeclaration type = ast.newTypeDeclaration();
type.setInterface(false);
type.setModifiers(Modifier.PUBLIC);
type.setName(ast.newSimpleName("HelloWorld"));
MethodDeclaration methodDeclaration = ast.newMethodDeclaration();
methodDeclaration.setConstructor(false);
methodDeclaration.setModifiers(Modifier.PUBLIC | Modifier.STATIC);
methodDeclaration.setName(ast.newSimpleName("main"));
methodDeclaration.setReturnType(ast.newPrimitiveType(PrimitiveType.VOID));
SingleVariableDeclaration variableDeclaration = ast.newSingleVariableDeclaration(
variableDeclaration.setModifiers(Modifier.NONE);
variableDeclaration.setType(ast.newArrayType(ast.newSimpleType(ast.newSimpleName(
variableDeclaration.setName(ast.newSimpleName("args"));
methodDeclaration.parameters().add(variableDeclaration);
org.eclipse.jdt.core.dom.Block block = ast.newBlock();
MethodInvocation methodInvocation = ast.newMethodInvocation();
name =
        ast.newQualifiedName(
                ast.newSimpleName("System"),
                ast.newSimpleName("out"));
methodInvocation.setExpression(name);
methodInvocation.setName(ast.newSimpleName("println"));
InfixExpression infixExpression = ast.newInfixExpression();
infixExpression.setOperator(InfixExpression.Operator.PLUS);
```

```
StringLiteral literal = ast.newStringLiteral();
literal.setLiteralValue("Hello");
infixExpression.setLeftOperand(literal);
literal = ast.newStringLiteral();
literal.setLiteralValue(" world");
infixExpression.setRightOperand(literal);
methodInvocation.arguments().add(infixExpression);
ExpressionStatement expressionStatement = ast.newExpressionStatement(methodInvoca
block.statements().add(expressionStatement);
methodDeclaration.setBody(block);
type.bodyDeclarations().add(methodDeclaration);
unit.types().add(type);
```

Retrieving extra positions

The DOM/AST node contains only a pair of positions (the starting position and the length of the node). This is not always sufficient. In order to retrieve intermediate positions, the **IScanner** API should be used. For example, we have an **InstanceofExpression** for which we want to know the positions of the *instanceof* operator. We could write the following method to achieve this:

```
private int[] getOperatorPosition(Expression expression, char[] source) {
        if (expression instanceof InstanceofExpression) {
                IScanner scanner = ToolFactory.createScanner(false, false, false, false);
                scanner.setSource(source);
                int start = expression.getStartPosition();
                int end = start + expression.getLength();
                scanner.resetTo(start, end);
                int token;
                try {
                        while ((token = scanner.getNextToken()) != ITerminalSymbols.Token
                                switch(token) {
                                        case ITerminalSymbols.TokenNameinstanceof:
                                                return new int[] {scanner.getCurrentToken
                                }
                        }
                } catch (InvalidInputException e) {
                }
        }
        return null;
}
```

The **IScanner** is used to divide the input source into tokens. Each token has a specific value that is defined in the **ITerminalSymbols** interface. It is fairly simple to iterate and retrieve the right token. We also recommend that you use the scanner if you want to find the position of the *super* keyword in a **SuperMethodInvocation**.

Source code modifcations

Some source code modifications are not provided via the Java element API. A more general way to edit source code (such as changing the source code for existing elements) is accomplished using the compilation unit's raw source code and the rewrite API of the DOM/AST.

To perform DOM/AST rewriting, there two set of API: the descriptive rewriting and the modifying rewriting.

The descriptive API does not modify the AST but use **ASTRewrite** API to generate the descritions of modifications. The AST rewriter collects descriptions of modifications to nodes and translates these descriptions into text edits that can then be applied to the original source.

```
// creation of a Document
ICompilationUnit cu = ... ; // content is "public class X {\n}"
String source = cu.getBuffer().getContents();
Document document= new Document(source);

// creation of DOM/AST from a ICompilationUnit
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource(cu);
CompilationUnit astRoot = (CompilationUnit) parser.createAST(null);

// creation of ASTRewrite
ASTRewrite rewrite = new ASTRewrite(astRoot.getAST());

// description of the change
SimpleName oldName = ((TypeDeclaration)astRoot.types().get(0)).getName();
SimpleName newName = astRoot.getAST().newSimpleName("Y");
rewrite.replace(oldName, newName, null);

// computation of the text edits
TextEdit edits = rewrite.rewriteAST(document, cu.getJavaProject().getOptions(true));

// computation of the new source code
edits.apply(document);
String newSource = document.get();

// update of the compilation unit
cu.getBuffer().setContents(newSource);
```

The modifying API allows to modify directly the AST:

- Request the recording of modifications (**CompilationUnit.recordModifications()**).
- Perform the modifications on the AST Nodes.
- And when the modifications are finished, generate text edits that can then be applied to the original source (**CompilationUnit.rewrite(...)**).

```
// creation of a Document
ICompilationUnit cu = ... ; // content is "public class X {\n}"
String source = cu.getBuffer().getContents();
Document document= new Document(source);

// creation of DOM/AST from a ICompilationUnit
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource(cu);
CompilationUnit astRoot = (CompilationUnit) parser.createAST(null);

// start record of the modifications
astRoot.recordModifications();

// modify the AST
TypeDeclaration typeDeclaration = (TypeDeclaration)astRoot.types().get(0)
SimpleName newName = astRoot.getAST().newSimpleName("Y");
typeDeclaration.setName(newName);

// computation of the text edits
TextEdit edits = astRoot.rewrite(document, cu.getJavaProject().getOptions(true));

// computation of the new source code
edits.apply(document);
String newSource = document.get();
```

```
// update of the compilation unit
cu.getBuffer().setContents(newSource);
```

Responding to changes in Java elements

If your plug−in needs to know about changes to Java elements after the fact, you can register a Java **IElementChangedListener** with **JavaCore**.

```
JavaCore.addElementChangedListener(new MyJavaElementChangeReporter());
```

You can be more specific and specify the type of events you're interested in using **addElementChangedListener(IElementChangedListener, int)**.

For example, if you're only interested in listening for events during a reconcile operation:

```
JavaCore.addElementChangedListener(new MyJavaElementChangeReporter(), ElementChangedEvent.POST
```

There are two kinds of events that are supported by **JavaCore**:

- **POST_CHANGE**: Listeners of this event kind will get notified during the corresponding **POST_CHANGE** resource change notification.
- **POST_RECONCILE**: Listeners of this event kind will get notified at the end of a reconcile operation on a working copy (see **ICompilationUnit.reconcile(int, boolean, WorkingCopyOwner, IProgressMonitor)**).

Java element change listeners are similar conceptually to resource change listeners (described in tracking resource changes). The following snippet implements a Java element change reporter that prints the element deltas to the system console.

```
public class MyJavaElementChangeReporter implements IElementChangedListener {
   public void elementChanged(ElementChangedEvent event) {
      IJavaElementDelta delta= event.getDelta();
      if (delta != null) {
         System.out.println("delta received: ");
         System.out.print(delta);
      }
   }
}
```

The **IJavaElementDelta** includes the element that was changed and flags describing the kind of change that occurred. Most of the time the delta tree is rooted at the Java Model level. Clients must then navigate this delta using **getAffectedChildren** to find out what projects have changed.

The following example method traverses a delta and prints the elements that have been added, removed and changed:

```
 void traverseAndPrint(IJavaElementDelta delta) {
     switch (delta.getKind()) {
         case IJavaElementDelta.ADDED:
             System.out.println(delta.getElement() + " was added");
             break;
         case IJavaElementDelta.REMOVED:
             System.out.println(delta.getElement() + " was removed");
             break;
         case IJavaElementDelta.CHANGED:
```

```
                System.out.println(delta.getElement() + " was changed");
                if ((delta.getFlags() & IJavaElementDelta.F_CHILDREN) != 0) {
                    System.out.println("The change was in its children");
                }
                if ((delta.getFlags() & IJavaElementDelta.F_CONTENT) != 0) {
                    System.out.println("The change was in its content");
                }
                /* Others flags can also be checked */
                break;
        }
        IJavaElementDelta[] children = delta.getAffectedChildren();
        for (int i = 0; i < children.length; i++) {
            traverseAndPrint(children[i]);
        }
    }
```

Several kinds of operations can trigger a Java element change notification. Here are some examples:

- Creating a resource, e.g. **IPackageFragment.createCompilationUnit** (the delta indicates the addition of the compilation unit)
- Modifying a resource, e.g. **ICompilationUnit.createType** (the delta indicates that the compilation unit has changed and that a type was added as a child of this compilation unit)
- Modifying a project's classpath, e.g. **IJavaProject.setRawClasspath** (the delta indicates that package fragment roots have been added to the classpath, removed from the classpath, or reordered on the classpath)
- Modifying a classpath variable value, e.g. **JavaCore.setClasspathVariable** (the delta also indicates that package fragment roots have been affected)
- Changing the source attachment of a .jar file, e.g. **IPackageFragmentRoot.attachSource** (the delta indicates that the source was detached then attached)
- Reconciling a working copy with its buffer, e.g. **ICompilationUnit.reconcile**
- Modifying an **IFile** that ends with ".java" and that is on the project's classpath, e.g. using **IFile.setContents** (the delta indicates that a compilation unit was changed, but no finer-grained information is provided as this was not done through a Java Model operation)

Similar to **IResourceDelta** the Java element deltas can be batched using an **IWorkspaceRunnable**. The deltas resulting from several Java Model operations that are run inside a **IWorkspaceRunnable** are merged and reported at once.

JavaCore provides a **run** method for batching Java element changes.

For example, the following code fragment will trigger 2 Java element change events:

```
// Get package
IPackageFragment pkg = ...;

// Create 2 compilation units
ICompilationUnit unitA = pkg.createCompilationUnit("A.java", "public class A {}", false, null
ICompilationUnit unitB = pkg.createCompilationUnit("B.java", "public class B {}", false, null
```

Whereas the following code fragment will trigger 1 Java element change event:

```
// Get package
IPackageFragment pkg = ...;

// Create 2 compilation units
JavaCore.run(
```

```
new IWorkspaceRunnable() {
        public void run(IProgressMonitor monitor) throws CoreException {
            ICompilationUnit unitA = pkg.createCompilationUnit("A.java", "public class A
            ICompilationUnit unitB = pkg.createCompilationUnit("B.java", "public class B
        }
},
null);
```

Using the Java search engine

Your plug−in can use the JDT API to search Java projects in the workspace for Java elements, such as method references, field declarations, implementors of an interface, etc.

The entry point for Java search is the **SearchEngine** class. You can search for particular patterns inside a Java element and scope the search to specific elements. Search patterns can be created using **createPattern**. A pattern is scoped using **createJavaSearchScope**. Once a pattern and scope are defined, the **search** method is used to collect the results.

Search results are reported to a **SearchRequestor** which you must extend in order to access the results.

Preparing for search

A search operation will use both a pattern for describing the nature of the search, and a scope for restraining the range of investigation.

Creating a Java search pattern

A search pattern defines how search results are found. You can either create a search pattern from a Java element (see **createPatternPattern(IJavaElement, int)**) or from a string (see **createPattern(String, int, int, int)**.) The last method supports wildcards (i.e. '*') and can be used to widen the search results.

For example, creating a search pattern for searching for references to a given method is done as follows:

```
// Get the method
IMethod method = ...;

// Create search pattern
SearchPattern pattern = SearchPattern.createPattern(method, IJavaSearchConstants.REFERENCES);
```

Or creating a search pattern for searching for declarations of all types starting with "Obj":

```
// Create search pattern
SearchPattern pattern = SearchPattern.createPattern("Obj*", IJavaSearchConstants.TYPE, IJavaS
```

The following search patterns are supported:

- Package declarations
- Type declarations
- Field declarations
- Method (and constructor) declarations
- Package references
- Type references

- Interface implementors
- Field references
- Field write accesses
- Field read accesses
- Method (and constructor) references
- Combinations of the above patterns using the OR pattern (see **createOrPattern**)

Creating a Java search scope

If you are interested in search results in a given project or even in a given package, or if you know that search results can be only in a hierarchy of a given type, you can create the appropriate search scope using **createJavaSearchScope(IJavaElement[])** or **createHierarchyScope(IType)**.

For example, creating a search scope on a given package is done as follows:

```
// Get the package
IPackageFragment pkg = ...;

// Create search scope
IJavaSearchScope scope = SearchEngine.createJavaSearchScope(new IJavaElement[] {pkg});
```

Or creating a search scope on the hierarchy of a given type is:

```
// Get the type
IType type = ...;

// Create search scope
IJavaSearchScope scope = SearchEngine.createHierarchyScope(type);
```

Finally, you can create a search scope on the entire workspace:

```
// Create search scope
IJavaSearchScope scope = SearchEngine.createWorkspaceScope();
```

Searching

Once you have created a search pattern and a search scope, and you have extended **SearchRequestor**, you can start a search query as follows:

```
// Get the search pattern
SearchPattern pattern = ...;

// Get the search scope
IJavaSearchScope scope = ...;

// Get the search requestor
SearchRequestor requestor = ...;

// Search
SearchEngine searchEngine = new SearchEngine();
searchEngine.search(pattern, new SearchParticipant[] {SearchEngine.getDefaultSearchParticipan
```

A notification that the search starts is sent to your search requestor using the **beginReporting** method. Then, each search result is reported using the **acceptSearchMatch** method. Finally **endReporting** indicates that the search has ended.

Collecting search results

Search results are reported using the **acceptSearchMatch** method. Paragraphs below detail the search match.

Resources and Java elements

A search result can correspond to a Java element (e.g. a type declaration) or it can be contained in a Java element (e.g. a reference to a type inside a method). The search engine always tries to find the innermost Java element that corresponds to or that contains the search result. For example, searching for references to a method could find such a reference in an initializer. The initializer that contains this method reference is the element of the search match.

The search engine also tries to find the resource that contains the Java element. So if the Java element is a method in a compilation unit, the resource is the corresponding **IFile**. If the element is contained in a .jar file, the resource is the .jar file, if this .jar file is in the workspace. If it is an external .jar file, then the resource is **null**.

Source positions

Source positions are given relative to the compilation unit that contains the search result. If the search result is contained in a .jar file, the source positions are relative to the attached source. They are (−1, −1) if there is no source attached to the .jar file.

Accurate versus inaccurate search results

In most cases search results are accurate, meaning that the search engine was able to determine that the given match is what was asked for. However in some cases the search engine is unable to do so, in such cases the match is inaccurate. Some possible reasons why a match could be inaccurate are:

- The classpath on the project that contains the result is not properly set. For example, it refers to a project that is not accessible, a jar on the classpath requires another jar that is not on the classpath, etc.
- The user code would not compile. For example, it refers to a class that is not yet defined.

JDT Core options

JDT Core options control the behavior of core features such as the Java compiler, code formatter, code assist, and other core behaviors. The APIs for accessing the options are defined in **JavaCore**. Options can be accessed as a group as follows:

- JavaCore.getDefaultOptions() − Answers the default value of the options.
- JavaCore.getOptions() − Answers the current values of the options.
- JavaCore.setOptions(Hashtable newOptions) − Replaces the options values by new values.

Options can also be accessed individually by a string name.

- JavaCore.getOption(String optionName) − Answers the value of a specific option.

Options are stored as a hash table of all known configurable options with their values. Helper constants have been defined on **JavaCore** for each option ID and its possible constant values.

The following code fragment restores the value of all core options to their defaults except for one (`COMPILER_PB_DEPRECATION`), which is set specifically.

```
// Get the current options
Hashtable options = JavaCore.getDefaultOptions();

// Change the value of an option
options.put(JavaCore.COMPILER_PB_DEPRECATION, JavaCore.ERROR);

// Set the new options
JavaCore.setOptions(options);
```

The following code fragment keeps the value of the current options and modifies only one (`COMPILER_PB_DEPRECATION`):

```
// Get the current options
Hashtable options = JavaCore.getOptions();

// Change the value of an option
options.put(JavaCore.COMPILER_PB_DEPRECATION, JavaCore.ERROR);

// Set the new options
JavaCore.setOptions(options);
```

Project specific options

The values of options can be overridden per project using protocol in **IJavaProject**.

The following code fragment retrieves the value of an option (`COMPILER_PB_DEPRECATION`) for a specific project in two different ways. The boolean parameter controls whether only the project–specific options should be returned in a query or whether the project's option values should be merged with the values in JavaCore.

```
// Get the project
IJavaProject project = ...;

// See if the value of an option has been set in this project
String value = project.getOption(JavaCore.COMPILER_PB_DEPRECATION, false);
if (value == null) {
  // no specific option was set on the project
  ...
}

// Get the value of an option from this project.  Use the value from
// JavaCore value if none is specified for the project
String value = project.getOption(JavaCore.COMPILER_PB_DEPRECATION, true);
```

Major change in default JDT Core 3.0 options

Default compliance level has been changed. Now default compliance level is `1.4` instead of `1.3` and default target platform is `1.2` instead of `1.1`.

JDT Core options descriptions

The following tables describe the available JDT Core options. The option id is shown in parentheses and the default value is shown in bold italics.

Options categories

- Compiler options
- Builder options
- JavaCore options
- Formatter options
- CodeAssist options

Compiler options

Description	Values
Inline JSR Bytecode Instruction (COMPILER_CODEGEN_INLINE_JSR_BYTECODE)	
When enabled, the compiler will no longer generate JSR instructions, but rather inline corresponding subroutine code sequences (mostly corresponding to try finally blocks). The generated code will thus get bigger, but will load faster on virtual machines since the verification process is then much simpler. This mode is anticipating support for the Java Specification Request 202.	ENABLED
	DISABLED
Defining Target Java Platform (COMPILER_CODEGEN_TARGET_PLATFORM)	
For binary compatibility reason, .class files can be tagged with certain VM versions and later. Note that the "1.4" target requires you to toggle the compliance mode to "1.4" also.	VERSION_1_1
	VERSION_1_2
	VERSION_1_3
	VERSION_1_4
Preserving Unused Local Variables (COMPILER_CODEGEN_UNUSED_LOCAL)	
Unless requested to preserve unused local variables (i.e. never read), the compiler will optimize them out, potentially altering debugging.	***PRESERVE***
	OPTIMIZE_OUT
Setting Compliance Level (COMPILER_COMPLIANCE)	
Select the compliance level for the compiler. In "1.3" mode, source and target settings should not go beyond "1.3" level.	VERSION_1_3
	VERSION_1_4
Javadoc Comment Support (COMPILER_DOC_COMMENT_SUPPORT)	
When this support is disabled, the compiler will ignore all javadoc problems options settings and will not report any javadoc problem. It will also not find any reference in javadoc comment and DOM AST Javadoc node will be only a flat text instead of having structured tag elements.	***ENABLED***
	DISABLED
Generating Line Number Debug Attribute (COMPILER_LINE_NUMBER_ATTR)	
When generated, this attribute will enable source code highlighting in the debugger (.class file is then bigger).	***GENERATE***
	DO_NOT_GENERATE
Generating Local Variable Debug Attribute (COMPILER_LOCAL_VARIABLE_ATTR)	

When generated, this attribute will enable local variable names to be displayed in the debugger, only in places where variables are definitely assigned (.class file is then bigger)	***GENERATE***
	DO_NOT_GENERATE
Reporting Use of Annotation Type as Super Interface (COMPILER_PB_ANNOTATION_SUPER_INTERFACE)	
When enabled, the compiler will issue an error or a warning whenever an annotation type is used as a super−interface. Though legal, this is discouraged.	ERROR
	WARNING
	IGNORE
Reporting Boxing/Unboxing Conversion (COMPILER_PB_ASSERT_IDENTIFIER)	
When enabled, the compiler will issue an error or a warning whenever a boxing or an unboxing conversion is performed.	ERROR
	WARNING
	IGNORE
Reporting Usage of 'assert' Identifier (COMPILER_PB_AUTOBOXING)	
When enabled, the compiler will issue an error or a warning whenever 'assert' is used as an identifier (reserved keyword in 1.4)	ERROR
	WARNING
	IGNORE
Reporting Usage of char[] Expressions in String Concatenations (COMPILER_PB_CHAR_ARRAY_IN_STRING_CONCATENATION)	
When enabled, the compiler will issue an error or a warning whenever a char[] expression is used in String concatenations (e.g. "hello" + new char[]{'w','o','r','l','d'}),	ERROR
	WARNING
	IGNORE
Reporting Deprecation (COMPILER_PB_DEPRECATION)	
When enabled, the compiler will signal use of deprecated API either as an error or a warning.	ERROR
	WARNING
	IGNORE
Reporting Deprecation Inside Deprecated Code (COMPILER_PB_DEPRECATION_IN_DEPRECATED_CODE)	
When enabled, the compiler will signal use of deprecated API either as an error or a warning.	ENABLED
	DISABLED
Reporting Deprecation When Overriding Deprecated Method (COMPILER_PB_DEPRECATION_WHEN_OVERRIDING_DEPRECATED_METHOD)	
When enabled, the compiler will signal the declaration of a method overriding a deprecated one.	ENABLED
	DISABLED
Reporting Discouraged Reference to Type with Restricted Access (COMPILER_PB_DISCOURAGED_REFERENCE)	
When enabled, the compiler will issue an error or a warning when referring to a type with discouraged access, as defined according to the access rule specifications.	ERROR
	WARNING

	IGNORE
Reporting Empty Statements and Unnecessary Semicolons (COMPILER_PB_EMPTY_STATEMENT)	
When enabled, the compiler will issue an error or a warning if an empty statement or a unnecessary semicolon is encountered.	ERROR
	WARNING
	IGNORE
Reporting Usage of 'enum' Identifier (COMPILER_PB_ENUM_IDENTIFIER)	
When enabled, the compiler will issue an error or a warning whenever 'enum' is used as an identifier (reserved keyword in 1.5)	ERROR
	WARNING
	IGNORE
Reporting Field Declaration Hiding another Variable (COMPILER_PB_FIELD_HIDING)	
When enabled, the compiler will issue an error or a warning whenever a field declaration is hiding some field or local variable (either locally, inherited or defined in enclosing type).	ERROR
	WARNING
	IGNORE
Reporting final Bound for Type Parameter (COMPILER_PB_FINAL_PARAMETER_BOUND)	
When enabled, the compiler will issue an error or a warning whenever a generic type parameter is associated with a bound corresponding to a final type; since final types cannot be further extended, the parameter is pretty useless.	ERROR
	WARNING
	IGNORE
Reporting Finally Blocks Not Completing Normally (COMPILER_PB_FINALLY_BLOCK_NOT_COMPLETING)	
When enabled, the compiler will issue an error or a warning when a finally block does not complete normally.	ERROR
	WARNING
	IGNORE
Reporting Finally Blocks Not Completing Normally (COMPILER_PB_FINALLY_BLOCK_NOT_COMPLETING)	
When enabled, the compiler will issue an error or a warning when a finally block does not complete normally.	ERROR
	WARNING
	IGNORE
Reporting Forbidden Reference to Type with Restricted Access (COMPILER_PB_FORBIDDEN_REFERENCE)	
When enabled, the compiler will issue an error or a warning when referring to a type that is non accessible, as defined according to the access rule specifications.	ERROR
	WARNING
	IGNORE
Reporting Hidden Catch Block (COMPILER_PB_HIDDEN_CATCH_BLOCK)	

Local to a try statement, some catch blocks may hide others , e.g.
`try {`
`  throw new java.io.CharConversionException();`
`} catch (java.io.CharConversionException e) {`
	ERROR

	WARNING
	IGNORE

`} catch (java.io.IOException e) {}`. When enabling this option, the compiler will issue an error or a warning for hidden catch blocks corresponding to checked exceptions	
Reporting Interface Method not Compatible with non−Inherited Methods (COMPILER_PB_INCOMPATIBLE_NON_INHERITED_INTERFACE_METHOD)	
When enabled, the compiler will issue an error or a warning whenever an interface defines a method incompatible with a non−inherited Object one.	ERROR ***WARNING*** IGNORE
Reporting Incomplete Enum Switch (COMPILER_PB_INCOMPLETE_ENUM_SWITCH)	
When enabled, the compiler will issue an error or a warning whenever an enum constant has no corresponding case label in an enum switch statement type has no case label matching an enum constant.	ERROR WARNING ***IGNORE***
Reporting Indirect Reference to a Static Member (COMPILER_PB_INDIRECT_STATIC_ACCESS)	
When enabled, the compiler will issue an error or a warning whenever a static field or method is accessed in an indirect way. A reference to a static member should preferably be qualified with its declaring type name.	ERROR WARNING> ***IGNORE***
Reporting Invalid Javadoc Comment (COMPILER_PB_INVALID_JAVADOC)	
This is the generic control for the severity of Javadoc problems. When enabled, the compiler will issue an error or a warning for a problem in Javadoc.	ERROR WARNING ***IGNORE***
Reporting Invalid Javadoc Tags (COMPILER_PB_INVALID_JAVADOC_TAGS)	
When enabled, the compiler will signal unbound or unexpected reference tags in Javadoc. A 'throws' tag referencing an undeclared exception would be considered as unexpected. Note that this diagnosis can be enabled based on the visibility of the construct associated with the Javadoc; also see the setting "org.eclipse.jdt.core.compiler.problem.invalidJavadocTagsVisibility".	***ENABLED*** DISABLED
Reporting Invalid Javadoc Tags with Deprecated References (COMPILER_PB_INVALID_JAVADOC_TAGS__DEPRECATED_REF)	
Specify whether the compiler will report deprecated references used in Javadoc tags. Note that this diagnosis can be enabled based on the visibility of the construct associated with the Javadoc; also see the setting "org.eclipse.jdt.core.compiler.problem.invalidJavadocTagsVisibility".	***ENABLED*** DISABLED
Reporting Invalid Javadoc Tags with Not Visible References (COMPILER_PB_INVALID_JAVADOC_TAGS__NOT_VISIBLE_REF)	
Specify whether the compiler will report non−visible references used in Javadoc tags. Note that this diagnosis can be enabled based on the visibility of the construct associated with the Javadoc; also see the setting	***ENABLED*** DISABLED

"org.eclipse.jdt.core.compiler.problem.invalidJavadocTagsVisibility".	
Visibility Level For Invalid Javadoc Tags (COMPILER_PB_INVALID_JAVADOC_TAGS_VISIBILITY)	
Set the minimum visibility level for Javadoc tag problems. Below this level problems will be ignored.	PUBLIC
	PROTECTED
	DEFAULT
	PRIVATE
Reporting Local Variable Declaration Hiding another Variable (COMPILER_PB_LOCAL_VARIABLE_HIDING)	
When enabled, the compiler will issue an error or a warning whenever a local variable declaration is hiding some field or local variable (either locally, inherited or defined in enclosing type).	ERROR
	WARNING
	IGNORE
Maximum number of problems reported per compilation unit (COMPILER_PB_MAX_PER_UNIT)	
Specify the maximum number of problems reported on each compilation unit (if the maximum is zero then all problems are reported).	a positive integer. ***Default value is 100***
Reporting Method With Constructor Name (COMPILER_PB_METHOD_WITH_CONSTRUCTOR_NAME)	
Naming a method with a constructor name is generally considered poor style programming. When enabling this option, the compiler will signal such scenarios either as an error or a warning.	ERROR
	WARNING
	IGNORE
Reporting Missing @Deprecated Annotation (COMPILER_PB_MISSING_DEPRECATED_ANNOTATION)	
When enabled, the compiler will issue an error or a warning whenever encountering a declaration carrying a @deprecated doc tag but has no corresponding @Deprecated annotation.	ERROR
	WARNING
	IGNORE
Reporting Missing Javadoc Comments (COMPILER_PB_MISSING_JAVADOC_COMMENTS)	
This is the generic control for the severity of missing Javadoc comment problems. When enabled, the compiler will issue an error or a warning when Javadoc comments are missing. Note that this diagnosis can be enabled based on the visibility of the construct associated with the expected Javadoc.	ERROR
	WARNING
	IGNORE
Reporting Missing Javadoc Comments on Overriding Methods (COMPILER_PB_MISSING_JAVADOC_COMMENTS_OVERRIDING)	
Specify whether the compiler will verify overriding methods in order to report missing Javadoc comment problems.	***ENABLED***
	DISABLED
Reporting Missing @Override Annotation (COMPILER_PB_MISSING_OVERRIDE_ANNOTATION)	
When enabled, the compiler will issue an error or a warning whenever encountering a method declaration which overrides a superclass method but has no @Override annotation.	ERROR
	WARNING

	IGNORE
Visibility Level For Missing Javadoc Comments (COMPILER_PB_MISSING_JAVADOC_COMMENTS_VISIBILITY)	
Set the minimum visibility level for missing Javadoc problems. Below this level problems will be ignored.	***PUBLIC***
	PROTECTED
	DEFAULT
	PRIVATE
Reporting Missing Javadoc Tags (COMPILER_PB_MISSING_JAVADOC_TAGS)	
This is the generic control for the severity of Javadoc missing tag problems. When enabled, the compiler will issue an error or a warning when tags are missing in Javadoc comments. Note that this diagnosis can be enabled based on the visibility of the construct associated with the Javadoc.	ERROR
	WARNING
	IGNORE
Reporting Missing Javadoc Tags on Overriding Methods (COMPILER_PB_MISSING_JAVADOC_TAGS_OVERRIDING)	
Specify whether the compiler will verify overriding methods in order to report Javadoc missing tag problems.	***ENABLED***
	DISABLED
Visibility Level For Missing Javadoc Tags (COMPILER_PB_MISSING_JAVADOC_TAGS_VISIBILITY)	
Set the minimum visibility level for Javadoc missing tag problems. Below this level problems will be ignored.	PUBLIC
	PROTECTED
	DEFAULT
	PRIVATE
Reporting Missing @Override Annotation (COMPILER_PB_MISSING_OVERRIDE_ANNOTATION)	
When enabled, the compiler will issue an error or a warning whenever encountering a method declaration which overrides a superclass method but has no @Override annotation.	ERROR
	WARNING
	IGNORE
Reporting Missing Declaration of serialVersionUID Field on Serializable Class (COMPILER_PB_MISSING_SERIAL_VERSION)	
When enabled, the compiler will issue an error or a warning whenever a serializable class is missing a local declaration of a serialVersionUID field. This field must be declared as static final and be of type long.	ERROR
	WARNING
	IGNORE
Reporting Assignment with No Effect (COMPILER_PB_NO_EFFECT_ASSIGNMENT)	
When enabled, the compiler will issue an error or a warning whenever an assignment has no effect (e.g. 'x = x').	ERROR
	WARNING
	IGNORE
Reporting Non−Externalized String Literal (COMPILER_PB_NON_NLS_STRING_LITERAL)	
	ERROR

When enabled, the compiler will issue an error or a warning for non externalized String literal (i.e. non tagged with //$NON−NLS−<n>$)	WARNING
	IGNORE
Reporting Attempt to Override Package−Default Method (COMPILER_PB_OVERRIDING_PACKAGE_DEFAULT_METHOD)	
A package default method is not visible in a different package, and thus cannot be overridden. When enabling this option, the compiler will signal such scenarios either as an error or a warning.	ERROR
	WARNING
	IGNORE
Reporting Possible Accidental Boolean Assignment(COMPILER_PB_POSSIBLE_ACCIDENTAL_BOOLEAN_ASSIGNMENT)	
When enabled, the compiler will issue an error or a warning if a boolean assignment is acting as the condition of a control statement (where it probably was meant to be a boolean comparison).	ERROR
	WARNING
	IGNORE
Reporting Special Parameter Hiding another Field (COMPILER_PB_SPECIAL_PARAMETER_HIDING_FIELD)	
When enabled, the compiler will signal cases where a constructor or setter method parameter declaration is hiding some field (either locally, inherited or defined in enclosing type).	ENABLED
	DISABLED
Reporting Non−Static Reference to a Static Member (COMPILER_PB_STATIC_ACCESS_RECEIVER)	
When enabled, the compiler will issue an error or a warning whenever a static field or method is accessed with an expression receiver.	ERROR
	WARNING
	IGNORE
Determining Effect of @SuppressWarnings (COMPILER_PB_SUPPRESS_WARNINGS)	
When enabled, the @SuppressWarnings annotation can be used to suppress some compiler warnings. When disabled, all @SupressWarnings annotations are ignored; i.e., warnings are reported.	***ENABLED***
	DISABLED>
Reporting Synthetic Access Emulation (COMPILER_PB_SYNTHETIC_ACCESS_EMULATION)	
When enabled, the compiler will issue an error or a warning whenever it emulates access to a non−accessible member of an enclosing type. Such access can have performance implications.	ERROR
	WARNING
	IGNORE
Reporting Type Parameter Declaration Hiding another Type (COMPILER_PB_TYPE_PARAMETER_HIDING)	
When enabled, the compiler will issue an error or a warning whenever a type parameter declaration is hiding some type.	ERROR
	WARNING
	IGNORE
Reporting Unchecked Type Operation (COMPILER_PB_UNCHECKED_TYPE_OPERATION)	
When enabled, the compiler will issue an error or a warning whenever an operation involves generic types, and potentially invalidates type safety since involving raw types (e.g. invoking #foo(X<String>) with arguments (X)).	ERROR
	WARNING

	IGNORE
Reporting Undocumented Empty Block (COMPILER_PB_UNDOCUMENTED_EMPTY_BLOCK)	
When enabled, the compiler will issue an error or a warning when an empty block is detected and it is not documented with any comment.	ERROR
	WARNING
	IGNORE
Reporting Unhandled Warning Token for @SuppressWarnings (COMPILER_PB_UNHANDLED_WARNING_TOKEN)	
When enabled, the compiler will issue an error or a warning when encountering a token it cannot handle inside a @SuppressWarnings annotation.	ERROR
	WARNING
	IGNORE
Reporting Unnecessary Else (COMPILER_PB_UNNECESSARY_ELSE)	
When enabled, the compiler will issue an error or a warning when a statement is unnecessarily nested within an else clause (in situation where then clause is not completing normally).	ERROR
	WARNING
	IGNORE
Reporting Unnecessary Type Check (COMPILER_PB_UNNECESSARY_TYPE_CHECK)	
When enabled, the compiler will issue an error or a warning when a cast or an instanceof operation is unnecessary.	ERROR
	WARNING
	IGNORE
Reporting Unqualified Access to Field (COMPILER_PB_UNQUALIFIED_FIELD_ACCESS)	
When enabled, the compiler will issue an error or a warning when a field is access without any qualification. In order to improve code readability, it should be qualified, e.g. 'x' should rather be written 'this.x'.	ERROR
	WARNING
	IGNORE
Reporting Unused Declared Thrown Exception (COMPILER_PB_UNUSED_DECLARED_THROWN_EXCEPTION)	
When enabled, the compiler will issue an error or a warning when a method or a constructor is declaring a thrown checked exception, but never actually raises it in its body.	ERROR
	WARNING
	IGNORE
Reporting Unused Declared Thrown Exception in Overriding Method (COMPILER_PB_UNUSED_DECLARED_THROWN_EXCEPTION_WHEN_OVERRIDING)	
When disabled, the compiler will not include overriding methods in its diagnosis for unused declared thrown exceptions.	ENABLED
	DISABLED
Reporting Unused Import (COMPILER_PB_UNUSED_IMPORT)	
When enabled, the compiler will issue an error or a warning for unused import reference	ERROR
	WARNING
	IGNORE
Reporting Unused Local (COMPILER_PB_UNUSED_LOCAL)	

When enabled, the compiler will issue an error or a warning for unused local variables (i.e. variables never read from)	ERROR
	WARNING
	IGNORE
Reporting Unused Parameter (COMPILER_PB_UNUSED_PARAMETER)	
When enabled, the compiler will issue an error or a warning for unused method parameters (i.e. parameters never read from)	ERROR
	WARNING
	IGNORE
Reporting Unused Parameter if Implementing Abstract Method (COMPILER_PB_UNUSED_PARAMETER_WHEN_IMPLEMENTING_ABSTRACT)	
When enabled, the compiler will signal unused parameters in abstract method implementations.	ENABLED
	DISABLED
Reporting Unused Parameter if Overriding Concrete Method (COMPILER_PB_UNUSED_PARAMETER_WHEN_OVERRIDING_CONCRETE)	
When enabled, the compiler will signal unused parameters in methods overriding concrete ones.	ENABLED
	DISABLED
Reporting Unused Private Members (COMPILER_PB_UNUSED_PRIVATE_MEMBER)	
When enabled, the compiler will issue an error or a warning whenever a private method or field is declared but never used within the same unit.	ERROR
	WARNING
	IGNORE
Reporting Varargs Argument Needing a Cast in Method/Constructor Invocation (COMPILER_PB_VARARGS_ARGUMENT_NEED_CAST)	
When enabled, the compiler will issue an error or a warning whenever a varargs arguments should be cast when passed to a method/constructor invocation. (e.g. Class.getMethod(String name, Class ... args) invoked with arguments ("foo", null)).	ERROR
	WARNING
	IGNORE
Setting Source Compatibility Mode (COMPILER_SOURCE)	
Specify whether source is 1.3 or 1.4 compatible. From 1.4 on, 'assert' is a keyword reserved for assertion support. Also note, than when toggling to 1.4 mode, the target VM level should be set to "1.4" and the compliance mode should be "1.4".	***VERSION_1_3***
	VERSION_1_4
Generating Source Debug Attribute (COMPILER_SOURCE_FILE_ATTR)	
When generated, this attribute will enable the debugger to present the corresponding source code.	***GENERATE***
	DO_NOT_GENERATE
Determine whether task tags are case−sensitive (COMPILER_TASK_CASE_SENSITIVE)	
When enabled, task tags are considered in a case−sensitive way.	***ENABLED***
	DISABLED
Define the Automatic Task Priorities (COMPILER_TASK_PRIORITIES)	

In parallel with the Automatic Task Tags, this list defines the priorities (high, normal or low) of the task markers issued by the compiler. If the default is specified, the priority of each task marker is "NORMAL". Possible priorities are "HIGH", "NORMAL" or "LOW".	{<priority>[,<priority>]*}. ***Default value is "NORMAL,HIGH, NORMAL"***
Define the Automatic Task Tags (COMPILER_TASK_TAGS)	
When the tag is non empty, the compiler will issue a task marker whenever it encounters one of the corresponding tag inside any comment in Java source code. Generated task messages will include the tag, and range until the next line separator or comment ending, and will be trimmed.	{<tag>[,<tag>]*}. ***Default value is "TODO,FIXME, XXX"***

Builder options

Description	Values
Cleaning Output Folder(s) (CORE_JAVA_BUILD_CLEAN_OUTPUT_FOLDER)	
Indicate whether the JavaBuilder is allowed to clean the output folders when performing full build operations.	***CLEAN***
	IGNORE
Reporting Duplicate Resources (CORE_JAVA_BUILD_DUPLICATE_RESOURCE)	
Instruct the builder to abort if the classpath is invalid	ERROR
	WARNING
Abort if Invalid Classpath (CORE_JAVA_BUILD_INVALID_CLASSPATH)	
Instruct the builder to abort if the classpath is invalid	ABORT
	IGNORE
Computing Project Build Order (CORE_JAVA_BUILD_ORDER)	
Indicate whether JavaCore should enforce the project build order to be based on the classpath prerequisite chain. When requesting to compute, this takes over the platform default order (based on project references).	COMPUTE
	IGNORE
Specifying Filters for Resource Copying Control (CORE_JAVA_BUILD_RESOURCE_COPY_FILTER)	
Specify filters to control the resource copy process. (<name> is a file name pattern (only * wild−cards allowed) or the name of a folder which ends with '/')	{<name>[,<name>]*}. ***Default value is ""***

JavaCore options

Description	Values
Reporting Classpath Cycle (CORE_CIRCULAR_CLASSPATH)	
Indicate the severity of the problem reported when a project is involved in a cycle.	***ERROR***
	WARNING
Enabling Usage of Classpath Exclusion Patterns (CORE_ENABLE_CLASSPATH_EXCLUSION_PATTERNS)	
When set to "disabled", no entry on a project classpath can be associated with an exclusion or inclusion pattern.	***ENABLED***
	DISABLED

Enabling Usage of Classpath Multiple Output Locations (CORE_ENABLE_CLASSPATH_MULTIPLE_OUTPUT_LOCATIONS)	
When set to "disabled", no entry on a project classpath can be associated with a specific output location, preventing thus usage of multiple output locations	***ENABLED***
	DISABLED
Specify Default Source Encoding Format (CORE_ENCODING)	
Get the encoding format for compiled sources. This setting is read−only, it is equivalent to ResourcesPlugin.getEncoding().	any of the supported encoding name. ***Default value is platform default***
Reporting Incompatible JDK Level for Required Binaries (CORE_INCOMPATIBLE_JDK_LEVEL)	
Indicate the severity of the problem reported when a project prerequisites another project or library with an incompatible target JDK level (e.g. project targeting 1.1 vm, but compiled against 1.4 libraries).	ERROR
	WARNING
	IGNORE
Reporting Incomplete Classpath (CORE_INCOMPLETE_CLASSPATH)	
Indicate the severity of the problem reported when an entry on the classpath doesn't exist, is not legitimate, or is not visible (e.g. a referenced project is closed).	***ERROR***
	WARNING

Formatter options

Description	Values
Option to align type members of a type declaration on column (FORMATTER_ALIGN_TYPE_MEMBERS_ON_COLUMNS)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of arguments in allocation expression (FORMATTER_ALIGNMENT_FOR_ARGUMENTS_IN_ALLOCATION_EXPRESSION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of arguments in enum constant (FORMATTER_ALIGNMENT_FOR_ARGUMENTS_IN_ENUM_CONSTANT)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call

Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of arguments in explicit constructor call (FORMATTER_ALIGNMENT_FOR_ARGUMENTS_IN_EXPLICIT_CONSTRUCTOR_CALL)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of arguments in method invocation (FORMATTER_ALIGNMENT_FOR_ARGUMENTS_IN_METHOD_INVOCATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of arguments in qualified allocation expression (FORMATTER_ALIGNMENT_FOR_ARGUMENTS_IN_QUALIFIED_ALLOCATION_EXPRESSION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of binary expression (FORMATTER_ALIGNMENT_FOR_BINARY_EXPRESSION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of compact if (FORMATTER_ALIGNMENT_FOR_COMPACT_IF)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_ONE_PER_LINE, INDENT_BY_ONE)
Option for alignment of conditional expression	

(FORMATTER_ALIGNMENT_FOR_CONDITIONAL_EXPRESSION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_ONE_PER_LINE, INDENT_DEFAULT)
Option for alignment of enum constants (FORMATTER_ALIGNMENT_FOR_ENUM_CONSTANTS)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_NO_SPLIT, INDENT_DEFAULT)
Option for alignment of expressions in array initializer (FORMATTER_ALIGNMENT_FOR_EXPRESSIONS_IN_ARRAY_INITIALIZER)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of multiple fields (FORMATTER_ALIGNMENT_FOR_MULTIPLE_FIELDS)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of parameters in constructor declaration (FORMATTER_ALIGNMENT_FOR_PARAMETERS_IN_CONSTRUCTOR_DECLARATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of parameters in method declaration (FORMATTER_ALIGNMENT_FOR_PARAMETERS_IN_METHOD_DECLARATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	

	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of selector in method invocation (FORMATTER_ALIGNMENT_FOR_SELECTOR_IN_METHOD_INVOCATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of superclass in type declaration (FORMATTER_ALIGNMENT_FOR_SUPERCLASS_IN_TYPE_DECLARATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_NEXT_SHIFTED, INDENT_DEFAULT)
Option for alignment of superinterfaces in enum declaration (FORMATTER_ALIGNMENT_FOR_SUPERINTERFACES_IN_ENUM_DECLARATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of superinterfaces in type declaration (FORMATTER_ALIGNMENT_FOR_SUPERINTERFACES_IN_TYPE_DECLARATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option for alignment of throws clause in constructor declaration (FORMATTER_ALIGNMENT_FOR_THROWS_CLAUSE_IN_CONSTRUCTOR_DECLARATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)

Option for alignment of throws clause in method declaration (FORMATTER_ALIGNMENT_FOR_THROWS_CLAUSE_IN_METHOD_DECLARATION)	
Possible value	values returned by `createAlignmentValue(boolean, int, int)` call
Default value	createAlignmentValue(false, WRAP_COMPACT, INDENT_DEFAULT)
Option to add blank lines after the imports declaration (FORMATTER_BLANK_LINES_AFTER_IMPORTS)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines after the package declaration (FORMATTER_BLANK_LINES_AFTER_PACKAGE)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines at the beginning of the method body (FORMATTER_BLANK_LINES_AT_BEGINNING_OF_METHOD_BODY)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines before a field declaration (FORMATTER_BLANK_LINES_BEFORE_FIELD)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines before the first class body declaration (FORMATTER_BLANK_LINES_BEFORE_FIRST_CLASS_BODY_DECLARATION)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines before the imports declaration (FORMATTER_BLANK_LINES_BEFORE_IMPORTS)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines before a member type declaration (FORMATTER_BLANK_LINES_BEFORE_MEMBER_TYPE)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines before a method declaration (FORMATTER_BLANK_LINES_BEFORE_METHOD)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"

Option to add blank lines before a new chunk (FORMATTER_BLANK_LINES_BEFORE_NEW_CHUNK)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines before the package declaration (FORMATTER_BLANK_LINES_BEFORE_PACKAGE)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to add blank lines between type declarations (FORMATTER_BLANK_LINES_BETWEEN_TYPE_DECLARATIONS)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to position the braces of an annotation type declaration (FORMATTER_BRACE_POSITION_FOR_ANNOTATION_TYPE_DECLARATION)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of an anonymous type declaration (FORMATTER_BRACE_POSITION_FOR_ANONYMOUS_TYPE_DECLARATION)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of an array initializer (FORMATTER_BRACE_POSITION_FOR_ARRAY_INITIALIZER)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of a block (FORMATTER_BRACE_POSITION_FOR_BLOCK)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of a block in a case statement when the block is the first statement following (FORMATTER_BRACE_POSITION_FOR_BLOCK_IN_CASE)	

Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of a constructor declaration (FORMATTER_BRACE_POSITION_FOR_CONSTRUCTOR_DECLARATION)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of an enum constant (FORMATTER_BRACE_POSITION_FOR_ENUM_CONSTANT)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of an enum declaration (FORMATTER_BRACE_POSITION_FOR_ENUM_DECLARATION)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of a method declaration (FORMATTER_BRACE_POSITION_FOR_METHOD_DECLARATION)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of a switch statement (FORMATTER_BRACE_POSITION_FOR_SWITCH)	
Possible values	***END_OF_LINE***
	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to position the braces of a type declaration (FORMATTER_BRACE_POSITION_FOR_TYPE_DECLARATION)	
Possible values	***END_OF_LINE***

	NEXT_LINE
	NEXT_LINE_SHIFTED
	NEXT_LINE_ON_WRAP
Option to control whether blank lines are cleared inside comments (FORMATTER_COMMENT_CLEAR_BLANK_LINES)	
Possible values	TRUE
	FALSE
Option to control whether comments are formatted (FORMATTER_COMMENT_FORMAT)	
Possible values	***TRUE***
	FALSE
Option to control whether the header comment of a Java source file is formatted (FORMATTER_COMMENT_FORMAT_HEADER)	
Possible values	TRUE
	FALSE
Option to control whether HTML tags are formatted. (FORMATTER_COMMENT_FORMAT_HTML)	
Possible values	***TRUE***
	FALSE
Option to control whether code snippets are formatted in comments (FORMATTER_COMMENT_FORMAT_SOURCE)	
Possible values	***TRUE***
	FALSE
Option to control whether description of Javadoc parameters are indented (FORMATTER_COMMENT_INDENT_PARAMETER_DESCRIPTION)	
Possible values	***TRUE***
	FALSE
Option to control whether Javadoc root tags are indented. (FORMATTER_COMMENT_INDENT_ROOT_TAGS)	
Possible values	***TRUE***
	FALSE
Option to insert an empty line before the Javadoc root tag block (FORMATTER_COMMENT_INSERT_EMPTY_LINE_BEFORE_ROOT_TAGS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a new line after Javadoc root tag parameters (FORMATTER_COMMENT_INSERT_NEW_LINE_FOR_PARAMETER)	
Possible values	***INSERT***
	DO_NOT_INSERT

Option to specify the line length for comments. (FORMATTER_COMMENT_LINE_LENGTH)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"80"
Option to compact else/if (FORMATTER_COMPACT_ELSE_IF)	
Possible values	***TRUE***
	FALSE
Option to set the continuation indentation (FORMATTER_CONTINUATION_INDENTATION)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"2"
Option to set the continuation indentation inside array initializer (FORMATTER_CONTINUATION_INDENTATION_FOR_ARRAY_INITIALIZER)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"2"
Option to indent body declarations compare to its enclosing enum constant header (FORMATTER_INDENT_BODY_DECLARATIONS_COMPARE_TO_ENUM_CONSTANT_HEADER)	
Possible values	***TRUE***
	FALSE
Option to indent body declarations compare to its enclosing enum declaration header (FORMATTER_INDENT_BODY_DECLARATIONS_COMPARE_TO_ENUM_DECLARATION_HEADER)	
Possible values	***TRUE***
	FALSE
Option to indent body declarations compare to its enclosing type header (FORMATTER_INDENT_BODY_DECLARATIONS_COMPARE_TO_TYPE_HEADER)	
Possible values	***TRUE***
	FALSE
Option to indent breaks compare to cases (FORMATTER_INDENT_BREAKS_COMPARE_TO_CASES)	
Possible values	***TRUE***
	FALSE
Option to indent statements inside a block (FORMATTER_INDENT_STATEMENTS_COMPARE_TO_BLOCK)	
Possible values	***TRUE***
	FALSE
Option to indent statements inside the body of a method or a constructor (FORMATTER_INDENT_STATEMENTS_COMPARE_TO_BODY)	
Possible values	***TRUE***
	FALSE

Option to indent switch statements compare to cases (FORMATTER_INDENT_SWITCHSTATEMENTS_COMPARE_TO_CASES)	
Possible values	***TRUE***
	FALSE
Option to indent switch statements compare to switch (FORMATTER_INDENT_SWITCHSTATEMENTS_COMPARE_TO_SWITCH)	
Possible values	***TRUE***
	FALSE
Option to specify the equivalent number of spaces that represents one indentation (FORMATTER_INDENTATION_SIZE)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"4"
Option to insert a new line after the opening brace in an array initializer (FORMATTER_INSERT_NEW_LINE_AFTER_ANNOTATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a new line after the opening brace in an array initializer (FORMATTER_INSERT_NEW_LINE_AFTER_OPENING_BRACE_IN_ARRAY_INITIALIZER)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a new line at the end of the current file if missing (FORMATTER_INSERT_NEW_LINE_AT_END_OF_FILE_IF_MISSING)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a new line before the catch keyword in try statement (FORMATTER_INSERT_NEW_LINE_BEFORE_CATCH_IN_TRY_STATEMENT)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a new line before the closing brace in an array initializer (FORMATTER_INSERT_NEW_LINE_BEFORE_CLOSING_BRACE_IN_ARRAY_INITIALIZER)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a new line before the else keyword in if statement (FORMATTER_INSERT_NEW_LINE_BEFORE_ELSE_IN_IF_STATEMENT)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a new line before the finally keyword in try statement (FORMATTER_INSERT_NEW_LINE_BEFORE_FINALLY_IN_TRY_STATEMENT)	

Possible values	INSERT
	DO_NOT_INSERT
Option to insert a new line before while in do statement (FORMATTER_INSERT_NEW_LINE_BEFORE_WHILE_IN_DO_STATEMENT)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a new line in an empty anonymous type declaration (FORMATTER_INSERT_NEW_LINE_IN_EMPTY_ANONYMOUS_TYPE_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a new line in an empty block (FORMATTER_INSERT_NEW_LINE_IN_EMPTY_BLOCK)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a new line in an empty enum constant (FORMATTER_INSERT_NEW_LINE_IN_EMPTY_ENUM_CONSTANT)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a new line in an empty enum declaration (FORMATTER_INSERT_NEW_LINE_IN_EMPTY_ENUM_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a new line in an empty method body (FORMATTER_INSERT_NEW_LINE_IN_EMPTY_METHOD_BODY)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a new line in an empty type declaration (FORMATTER_INSERT_NEW_LINE_IN_EMPTY_TYPE_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after and in wilcard (FORMATTER_INSERT_SPACE_AFTER_AND_IN_TYPE_PARAMETER)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after an assignment operator (FORMATTER_INSERT_SPACE_AFTER_ASSIGNMENT_OPERATOR)	
Possible values	***INSERT***
	DO_NOT_INSERT

Option to insert a space after at in annotation (FORMATTER_INSERT_SPACE_AFTER_AT_IN_ANNOTATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after at in annotation type declaration (FORMATTER_INSERT_SPACE_AFTER_AT_IN_ANNOTATION_TYPE_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after a binary operator (FORMATTER_INSERT_SPACE_AFTER_BINARY_OPERATOR)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the closing angle bracket in type arguments (FORMATTER_INSERT_SPACE_AFTER_CLOSING_ANGLE_BRACKET_IN_TYPE_ARGUMENTS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the closing angle bracket in type parameters (FORMATTER_INSERT_SPACE_AFTER_CLOSING_ANGLE_BRACKET_IN_TYPE_PARAMETERS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the closing brace of a block (FORMATTER_INSERT_SPACE_AFTER_CLOSING_BRACE_IN_BLOCK)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the closing parenthesis of a cast expression (FORMATTER_INSERT_SPACE_AFTER_CLOSING_PAREN_IN_CAST)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the colon in an assert statement (FORMATTER_INSERT_SPACE_AFTER_COLON_IN_ASSERT)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after colon in a case statement when a opening brace follows the colon (FORMATTER_INSERT_SPACE_AFTER_COLON_IN_CASE)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the colon in a conditional expression (FORMATTER_INSERT_SPACE_AFTER_COLON_IN_CONDITIONAL)	

Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after colon in a for statement (FORMATTER_INSERT_SPACE_AFTER_COLON_IN_FOR)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the colon in a labeled statement (FORMATTER_INSERT_SPACE_AFTER_COLON_IN_LABELED_STATEMENT)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in an allocation expression (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_ALLOCATION_EXPRESSION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in annotation (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_ANNOTATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in an array initializer (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_ARRAY_INITIALIZER)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the parameters of a constructor declaration (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_CONSTRUCTOR_DECLARATION_PARAMETERS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the exception names in a throws clause of a constructor declaration (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_CONSTRUCTOR_DECLARATION_THROWS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the arguments of an enum constant (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_ENUM_CONSTANT_ARGUMENTS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in enum declarations (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_ENUM_DECLARATIONS)	
Possible values	***INSERT***

	DO_NOT_INSERT
Option to insert a space after the comma in the arguments of an explicit constructor call (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_EXPLICIT_CONSTRUCTOR_CALL_ARGUMENTS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the increments of a for statement (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_FOR_INCREMENTS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the initializations of a for statement (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_FOR_INITS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the parameters of a method declaration (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_METHOD_DECLARATION_PARAMETERS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the exception names in a throws clause of a method declaration (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_METHOD_DECLARATION_THROWS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in the arguments of a method invocation (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_METHOD_INVOCATION_ARGUMENTS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in multiple field declaration (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_MULTIPLE_FIELD_DECLARATIONS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in multiple local declaration (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_MULTIPLE_LOCAL_DECLARATIONS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in parameterized type reference (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_PARAMETERIZED_TYPE_REFERENCE)	
Possible values	***INSERT***
	DO_NOT_INSERT

Option to insert a space after the comma in superinterfaces names of a type header (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_SUPERINTERFACES)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in type arguments (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_TYPE_ARGUMENTS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the comma in type parameters (FORMATTER_INSERT_SPACE_AFTER_COMMA_IN_TYPE_PARAMETERS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after ellipsis (FORMATTER_INSERT_SPACE_AFTER_ELLIPSIS)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after the opening angle bracket in parameterized type reference (FORMATTER_INSERT_SPACE_AFTER_OPENING_ANGLE_BRACKET_IN_PARAMETERIZED_TYPE_REF	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening angle bracket in type arguments (FORMATTER_INSERT_SPACE_AFTER_OPENING_ANGLE_BRACKET_IN_TYPE_ARGUMENTS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening angle bracket in type parameters (FORMATTER_INSERT_SPACE_AFTER_OPENING_ANGLE_BRACKET_IN_TYPE_PARAMETERS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening brace in an array initializer (FORMATTER_INSERT_SPACE_AFTER_OPENING_BRACE_IN_ARRAY_INITIALIZER)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening bracket inside an array allocation expression (FORMATTER_INSERT_SPACE_AFTER_OPENING_BRACKET_IN_ARRAY_ALLOCATION_EXPRESSION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening bracket inside an array reference (FORMATTER_INSERT_SPACE_AFTER_OPENING_BRACKET_IN_ARRAY_REFERENCE)	
Possible values	INSERT

	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in annotation (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_ANNOTATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a cast expression (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_CAST)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a catch (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_CATCH)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a constructor declaration (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_CONSTRUCTOR_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in enum constant (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_ENUM_CONSTANT)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a for statement (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_FOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in an if statement (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_IF)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a method declaration (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_METHOD_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a method invocation (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_METHOD_INVOCATION)	
Possible values	INSERT
	DO_NOT_INSERT

Option to insert a space after the opening parenthesis in a parenthesized expression (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_PARENTHESIZED_EXPRESSION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a switch statement (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_SWITCH)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a synchronized statement (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_SYNCHRONIZED)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after the opening parenthesis in a while statement (FORMATTER_INSERT_SPACE_AFTER_OPENING_PAREN_IN_WHILE)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after a postfix operator (FORMATTER_INSERT_SPACE_AFTER_POSTFIX_OPERATOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after a prefix operator (FORMATTER_INSERT_SPACE_AFTER_PREFIX_OPERATOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after question mark in a conditional expression (FORMATTER_INSERT_SPACE_AFTER_QUESTION_IN_CONDITIONAL)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after question mark in a wildcard (FORMATTER_INSERT_SPACE_AFTER_QUESTION_IN_WILDCARD)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space after semicolon in a for statement (FORMATTER_INSERT_SPACE_AFTER_SEMICOLON_IN_FOR)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space after an unary operator (FORMATTER_INSERT_SPACE_AFTER_UNARY_OPERATOR)	

Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before and in wildcard (FORMATTER_INSERT_SPACE_BEFORE_AND_IN_TYPE_PARAMETER)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before an assignment operator (FORMATTER_INSERT_SPACE_BEFORE_ASSIGNMENT_OPERATOR)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before at in annotation type declaration (FORMATTER_INSERT_SPACE_BEFORE_AT_IN_ANNOTATION_TYPE_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before an binary operator (FORMATTER_INSERT_SPACE_BEFORE_BINARY_OPERATOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing angle bracket in parameterized type reference (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_ANGLE_BRACKET_IN_PARAMETERIZED_TYPE_RE	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing angle bracket in type arguments (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_ANGLE_BRACKET_IN_TYPE_ARGUMENTS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing angle bracket in type parameters (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_ANGLE_BRACKET_IN_TYPE_PARAMETERS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing brace in an array initializer (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_BRACE_IN_ARRAY_INITIALIZER)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing bracket in an array allocation expression (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_BRACKET_IN_ARRAY_ALLOCATION_EXPRESSION	
Possible values	INSERT
	DO_NOT_INSERT

Option to insert a space before the closing bracket in an array reference (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_BRACKET_IN_ARRAY_REFERENCE)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in annotation (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_ANNOTATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a cast expression (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_CAST)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a catch (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_CATCH)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a constructor declaration (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_CONSTRUCTOR_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in enum constant (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_ENUM_CONSTANT)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a for statement (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_FOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in an if statement (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_IF)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a method declaration (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_METHOD_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a method invocation (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_METHOD_INVOCATION)	

Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a parenthesized expression (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_PARENTHESIZED_EXPRESSION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a switch statement (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_SWITCH)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a synchronized statement (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_SYNCHRONIZED)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the closing parenthesis in a while statement (FORMATTER_INSERT_SPACE_BEFORE_CLOSING_PAREN_IN_WHILE)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before colon in an assert statement (FORMATTER_INSERT_SPACE_BEFORE_COLON_IN_ASSERT)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before colon in a case statement (FORMATTER_INSERT_SPACE_BEFORE_COLON_IN_CASE)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before colon in a conditional expression (FORMATTER_INSERT_SPACE_BEFORE_COLON_IN_CONDITIONAL)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before colon in a default statement (FORMATTER_INSERT_SPACE_BEFORE_COLON_IN_DEFAULT)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before colon in a for statement (FORMATTER_INSERT_SPACE_BEFORE_COLON_IN_FOR)	
Possible values	***INSERT***
	DO_NOT_INSERT

Option to insert a space before colon in a labeled statement (FORMATTER_INSERT_SPACE_BEFORE_COLON_IN_LABELED_STATEMENT)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before comma in an allocation expression (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_ALLOCATION_EXPRESSION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in annotation (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_ANNOTATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in an array initializer (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_ARRAY_INITIALIZER)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the parameters of a constructor declaration (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_CONSTRUCTOR_DECLARATION_PARAMETERS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the exception names of the throws clause of a constructor declaration (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_CONSTRUCTOR_DECLARATION_THROWS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the arguments of enum constant (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_ENUM_CONSTANT_ARGUMENTS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in enum declarations (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_ENUM_DECLARATIONS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the arguments of an explicit constructor call (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_EXPLICIT_CONSTRUCTOR_CALL_ARGUMENTS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the increments of a for statement (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_FOR_INCREMENTS)	

Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the initializations of a for statement (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_FOR_INITS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the parameters of a method declaration (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_METHOD_DECLARATION_PARAMETERS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the exception names of the throws clause of a method declaration (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_METHOD_DECLARATION_THROWS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the arguments of a method invocation (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_METHOD_INVOCATION_ARGUMENTS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in a multiple field declaration (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_MULTIPLE_FIELD_DECLARATIONS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in a multiple local declaration (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_MULTIPLE_LOCAL_DECLARATIONS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in parameterized type reference (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_PARAMETERIZED_TYPE_REFERENCE)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in the superinterfaces names in a type header (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_SUPERINTERFACES)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before comma in type arguments (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_TYPE_ARGUMENTS)	
Possible values	INSERT

	DO_NOT_INSERT
Option to insert a space before comma in type parameters (FORMATTER_INSERT_SPACE_BEFORE_COMMA_IN_TYPE_PARAMETERS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before ellipsis (FORMATTER_INSERT_SPACE_BEFORE_ELLIPSIS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening angle bracket in parameterized type reference (FORMATTER_INSERT_SPACE_BEFORE_OPENING_ANGLE_BRACKET_IN_PARAMETERIZED_TYPE_RE	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening angle bracket in type arguments (FORMATTER_INSERT_SPACE_BEFORE_OPENING_ANGLE_BRACKET_IN_TYPE_ARGUMENTS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening angle bracket in type parameters (FORMATTER_INSERT_SPACE_BEFORE_OPENING_ANGLE_BRACKET_IN_TYPE_PARAMETERS)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening brace in an annotation type declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_ANNOTATION_TYPE_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in an anonymous type declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_ANONYMOUS_TYPE_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in an array initializer (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_ARRAY_INITIALIZER)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening brace in a block (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_BLOCK)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in a constructor declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_CONSTRUCTOR_DECLARATION)	

Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in an enum constant (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_ENUM_CONSTANT)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in an enum declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_ENUM_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in a method declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_METHOD_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in a switch statement (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_SWITCH)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening brace in a type declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACE_IN_TYPE_DECLARATION)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening bracket in an array allocation expression (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACKET_IN_ARRAY_ALLOCATION_EXPRESSION	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening bracket in an array reference (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACKET_IN_ARRAY_REFERENCE)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening bracket in an array type reference (FORMATTER_INSERT_SPACE_BEFORE_OPENING_BRACKET_IN_ARRAY_TYPE_REFERENCE)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in annotation (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_ANNOTATION)	
Possible values	INSERT
	DO_NOT_INSERT

Option to insert a space before the opening parenthesis in annotation type member declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_ANNOTATION_TYPE_MEMBER_DECLA	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a catch (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_CATCH)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a constructor declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_CONSTRUCTOR_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in enum constant (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_ENUM_CONSTANT)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a for statement (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_FOR)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in an if statement (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_IF)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a method declaration (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_METHOD_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a method invocation (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_METHOD_INVOCATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a parenthesized expression (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_PARENTHESIZED_EXPRESSION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a switch statement (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_SWITCH)	

Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a synchronized statement (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_SYNCHRONIZED)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before the opening parenthesis in a while statement (FORMATTER_INSERT_SPACE_BEFORE_OPENING_PAREN_IN_WHILE)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before a postfix operator (FORMATTER_INSERT_SPACE_BEFORE_POSTFIX_OPERATOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before a prefix operator (FORMATTER_INSERT_SPACE_BEFORE_PREFIX_OPERATOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before question mark in a conditional expression (FORMATTER_INSERT_SPACE_BEFORE_QUESTION_IN_CONDITIONAL)	
Possible values	***INSERT***
	DO_NOT_INSERT
Option to insert a space before question mark in a wildcard (FORMATTER_INSERT_SPACE_BEFORE_QUESTION_IN_WILDCARD)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before semicolon (FORMATTER_INSERT_SPACE_BEFORE_SEMICOLON)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before semicolon in for statement (FORMATTER_INSERT_SPACE_BEFORE_SEMICOLON_IN_FOR)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space before unary operator (FORMATTER_INSERT_SPACE_BEFORE_UNARY_OPERATOR)	
Possible values	INSERT
	DO_NOT_INSERT

Option to insert a space between brackets in an array type reference (FORMATTER_INSERT_SPACE_BETWEEN_BRACKETS_IN_ARRAY_TYPE_REFERENCE)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space between empty braces in an array initializer (FORMATTER_INSERT_SPACE_BETWEEN_EMPTY_BRACES_IN_ARRAY_INITIALIZER)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space between empty brackets in an array allocation expression (FORMATTER_INSERT_SPACE_BETWEEN_EMPTY_BRACKETS_IN_ARRAY_ALLOCATION_EXPRESSIO	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space between empty parenthesis in an annotation type member declaration (FORMATTER_INSERT_SPACE_BETWEEN_EMPTY_PARENS_IN_ANNOTATION_TYPE_MEMBER_DECLA	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space between empty parenthesis in a constructor declaration (FORMATTER_INSERT_SPACE_BETWEEN_EMPTY_PARENS_IN_CONSTRUCTOR_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space between empty parenthesis in enum constant (FORMATTER_INSERT_SPACE_BETWEEN_EMPTY_PARENS_IN_ENUM_CONSTANT)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space between empty parenthesis in a method declaration (FORMATTER_INSERT_SPACE_BETWEEN_EMPTY_PARENS_IN_METHOD_DECLARATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to insert a space between empty parenthesis in a method invocation (FORMATTER_INSERT_SPACE_BETWEEN_EMPTY_PARENS_IN_METHOD_INVOCATION)	
Possible values	INSERT
	DO_NOT_INSERT
Option to keep else statement on the same line (FORMATTER_KEEP_ELSE_STATEMENT_ON_SAME_LINE)	
Possible values	TRUE
	FALSE
Option to keep empty array initializer one one line (FORMATTER_KEEP_EMPTY_ARRAY_INITIALIZER_ON_ONE_LINE)	

Possible values	TRUE
	FALSE
Option to keep guardian clause on one line (FORMATTER_KEEP_GUARDIAN_CLAUSE_ON_ONE_LINE)	
Possible values	TRUE
	FALSE
Option to keep simple if statement on the one line (FORMATTER_KEEP_SIMPLE_IF_ON_ONE_LINE)	
Possible values	TRUE
	FALSE
Option to keep then statement on the same line (FORMATTER_KEEP_THEN_STATEMENT_ON_SAME_LINE)	
Possible values	TRUE
	FALSE
Option to specify the length of the page. Beyond this length, the formatter will try to split the code (FORMATTER_LINE_SPLIT)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"80"
Option to specify the number of empty lines to preserve (FORMATTER_NUMBER_OF_EMPTY_LINES_TO_PRESERVE)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"0"
Option to specify whether or not empty statement should be on a new line (FORMATTER_PUT_EMPTY_STATEMENT_ON_NEW_LINE)	
Possible values	TRUE
	FALSE
Option to specify the tabulation size (FORMATTER_TAB_CHAR)	
Possible values	***TAB***
	SPACE
	MIXED
Option to specify the equivalent number of spaces that represents one tabulation (FORMATTER_TAB_SIZE)	
Possible value	"<n>", where n is zero or a positive integer
Default value	"4"
Option to use tabulations only for leading indentations (FORMATTER_USE_TABS_ONLY_FOR_LEADING_INDENTATIONS)	
Possible values	TRUE
	FALSE

CodeAssist options

Description	Values
Define the Prefixes for Argument Name (CODEASSIST_ARGUMENT_PREFIXES)	
When the prefixes is non empty, completion for argument name will begin with one of the proposed prefixes.	{<prefix>[,<prefix>]*}. ***Default value is ""***
Define the Suffixes for Argument Name (CODEASSIST_ARGUMENT_SUFFIXES)	
When the suffixes is non empty, completion for argument name will end with one of the proposed suffixes.	{<suffix>[,<suffix>]*}. ***Default value is ""***
Activate Discouraged Reference Sensitive Completion (CODEASSIST_DISCOURAGED_REFERENCE_CHECK)	
When active, completion doesn't show that have discouraged reference.	ENABLED
	DISABLED
Define the Prefixes for Field Name (CODEASSIST_FIELD_PREFIXES)	
When the prefixes is non empty, completion for field name will begin with one of the proposed prefixes.	{<prefix>[,<prefix>]*}. ***Default value is ""***
Define the Suffixes for Field Name (CODEASSIST_FIELD_SUFFIXES)	
When the suffixes is non empty, completion for field name will end with one of the proposed suffixes.	{<suffix>[,<suffix>]*}. ***Default value is ""***
Activate Forbidden Reference Sensitive Completion (CODEASSIST_FORBIDDEN_REFERENCE_CHECK)	
When active, completion doesn't show that have forbidden reference.	***ENABLED***
	DISABLED
Automatic Qualification of Implicit Members (CODEASSIST_IMPLICIT_QUALIFICATION)	
When active, completion automatically qualifies completion on implicit field references and message expressions.	ENABLED
	DISABLED
Define the Prefixes for Local Variable Name (CODEASSIST_LOCAL_PREFIXES)	
When the prefixes is non empty, completion for local variable name will begin with one of the proposed prefixes.	{<prefix>[,<prefix>]*}. ***Default value is ""***
Define the Suffixes for Local Variable Name (CODEASSIST_LOCAL_SUFFIXES)	
When the suffixes is non empty, completion for local variable name will end with one of the proposed suffixes.	{<suffix>[,<suffix>]*}. ***Default value is ""***
Define the Prefixes for Static Field Name (CODEASSIST_STATIC_FIELD_PREFIXES)	
When the prefixes is non empty, completion for static field name will begin with one of the proposed prefixes.	{<prefix>[,<prefix>]*}. ***Default value is ""***
Define the Suffixes for Static Field Name (CODEASSIST_STATIC_FIELD_SUFFIXES)	
When the suffixes is non empty, completion for static field name will end with one of the proposed suffixes.	{<suffix>[,<suffix>]*}. ***Default value is ""***
Activate Visibility Sensitive Completion (CODEASSIST_VISIBILITY_CHECK)	

When active, completion doesn't show that you can not see (e.g. you can not see private methods of a super class).	ENABLED
	DISABLED

Performing code assist on Java code

The JDT API allows other plug−ins to perform code assist or code select on some Java elements. Elements that allow this manipulation should implement **ICodeAssist**.

There are two kinds of manipulation:

- Code completion − compute the completion of a Java token.
- Code selection − answer the Java element indicated by the selected text of a given offset and length.

In the Java model there are two elements that implement this interface: **IClassFile** and **ICompilationUnit**. Code completion and code selection only answer results for a class file if it has attached source.

Code completion

Performing a code completion

The only way to programmatically perform code completion is to invoke **ICodeAssist.codeComplete**. You specify the offset in the compilation unit after which the code completion is desired. You must also supply an instance of **ICompletionRequestor** to accept the possible completions.

Each method in **ICompletionRequestor** accepts a different kind of proposal for code completion. The parameters of each method include text that describes the proposed element (its name, declaring type, etc.), its proposed position for insertion in the compilation unit, and its relevance.

A completion requester can accept many different types of completions including the insertion of the following elements:

- anonymous types
- classes
- fields
- interfaces
- keywords
- labels
- local variables
- method call
- method declaration
- modifier
- package import or reference
- type
- variable name

The completion requester must also be able to accept compilation errors.

If your plug−in is not interested in every kind of code completion, a **CompletionRequestorAdapter** can be used so that you need only implement the kinds of completions you are interested in. The following example shows an adapter that is only used to accept class completions.

```
// Get the compilation unit
ICompilationUnit unit = ...;

// Get the offset
int offset = ...;

// Create the requester
ICompletionRequestor requestor = new CompletionRequestorAdapter() {
   public void acceptClass(
      char[] packageName,
      char[] className,
      char[] completionName,
      int modifiers,
      int completionStart,
      int completionEnd,
      int relevance) {
      System.out.println("propose a class named " + new String(className));
   }
};

// Compute proposals
unit.codeComplete(offset, requestor);
```

Completion relevance

Because there may be many different possible completions, the notion of relevance is used to compare the relevance of a suggested completion to other proposals. Relevance is represented by a positive integer. The value has no implicit meaning except to be used relative to the value for other proposals. The relevance of a code completion candidate can be affected by the expected type of the expression, as it relates to the types in the surrounding code, such as variable types, cast types, return types, etc. The presence of an expected prefix or suffix in a completion also affects its relevance.

Code completion options

The JDT Core plug–in defines options that control the behavior of code completion. These options can be changed by other plug–ins.

- Activate Visibility Sensitive Completion
  When this option is active, code completion will not answer elements that are not visible in the current context. (For example, it will not answer private methods of a super class.)
- Automatic Qualification of Implicit Members
  When this option is active, completion automatically qualifies completion on implicit field references and message expressions.

Additional options allow you to specify prefixes and suffixes for the proposed completion names for fields, static fields, local variables, and method arguments.

See JDT Core Code Assist Options for more information about the code assist options and their defaults.

Code selection

Performing a code selection

Code selection is used to find the Java element represented by a range of text (typically the selected text) in a compilation unit. To programmatically perform code selection, you must invoke **ICodeAssist.codeSelect**. You must supply the starting index location of the selection and its length. The result is an array of Java elements. Most of the time there is only one element in the array, but if the selection is ambiguous then all the possible elements are returned.

In the following example, code select is invoked for a compilation unit.

```
// Get the compilation unit
ICompilationUnit unit = ...;

// Get the offset and length
int offset = ...;
int length = ...;

// perform selection
IJavaElement[] elements = unit.codeSelect(offset, length);
System.out.println("the selected element is " + element[0].getElementName());
```

Selection at cursor location

When the selection length is specified as 0, a selection will be computed by finding the complete token that encloses the specified offset. Consider the following example method:

```
public void fooMethod(Object) {
}
```

If you specify an offset after the first character of *fooMethod*, and you specify a length of 0, then the selection will be computed to include the entire token *fooMethod*. If instead, you specify a length of 5, the selection will considered as *ooMet*.

Java model

The Java model is the set of classes that model the objects associated with creating, editing, and building a Java program. The Java model classes are defined in **org.eclipse.jdt.core**. These classes implement Java specific behavior for resources and further decompose Java resources into model elements.

Java elements

The package **org.eclipse.jdt.core** defines the classes that model the elements that compose a Java program. The JDT uses an in−memory object model to represent the structure of a Java program. This structure is derived from the project's class path. The model is hierarchical. Elements of a program can be decomposed into child elements.

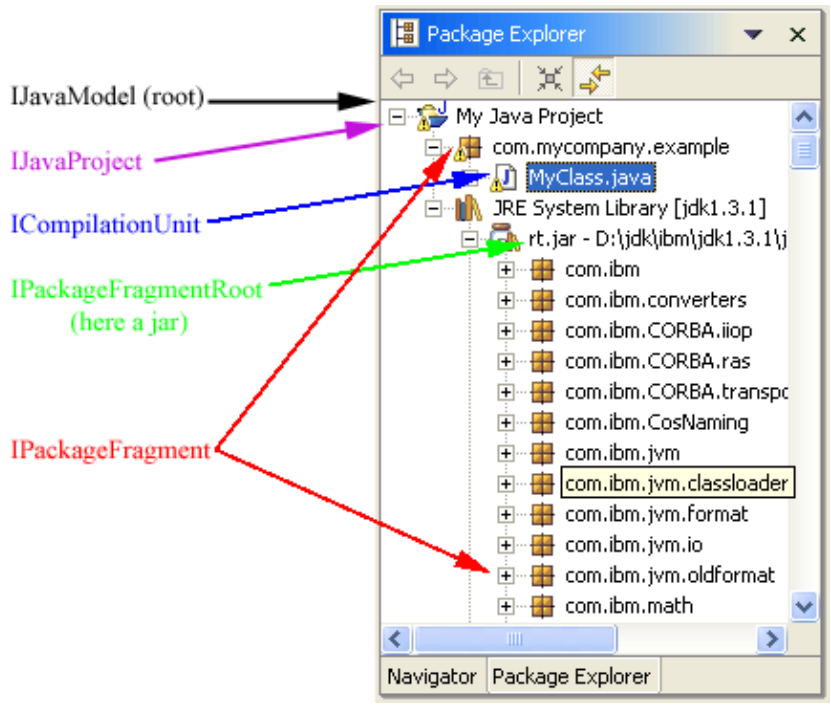
Manipulating Java elements is similar to manipulating resource objects. When you work with a Java element, you are actually working with a **handle** to some underlying model object. You must use the **exists()** protocol to determine whether the element is actually present in the workspace.

The following table summarizes the different kinds of Java elements.

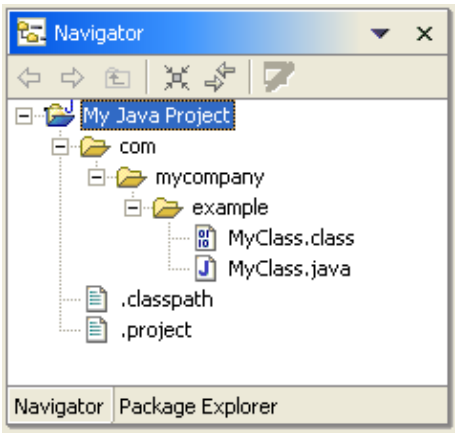
Element	Description
IJavaModel	Represents the root Java element, corresponding to the workspace. The parent of all projects with the Java nature. It also gives you access to the projects without the java nature.
IJavaProject	Represents a Java project in the workspace. (Child of **IJavaModel**)
IPackageFragmentRoot	Represents a set of package fragments, and maps the fragments to an underlying resource which is either a folder, JAR, or ZIP file. (Child of **IJavaProject**)
IPackageFragment	Represents the portion of the workspace that corresponds to an entire package, or a portion of the package. (Child of **IPackageFragmentRoot**)
ICompilationUnit	Represents a Java source (**.java**) file. (Child of **IPackageFragment**)
IPackageDeclaration	Represents a package declaration in a compilation unit. (Child of **ICompilationUnit**)
IImportContainer	Represents the collection of package import declarations in a compilation unit. (Child of **ICompilationUnit**)
IImportDeclaration	Represents a single package import declaration. (Child of **IImportContainer**)
IType	Represents either a source type inside a compilation unit, or a binary type inside a class file.
IField	Represents a field inside a type. (Child of **IType**)
IMethod	Represents a method or constructor inside a type. (Child of **IType**)
IInitializer	Represents a static or instance initializer inside a type. (Child of **IType**)
IClassFile	Represents a compiled (binary) type. (Child of **IPackageFragment**)

All Java elements support the **IJavaElement** interface.

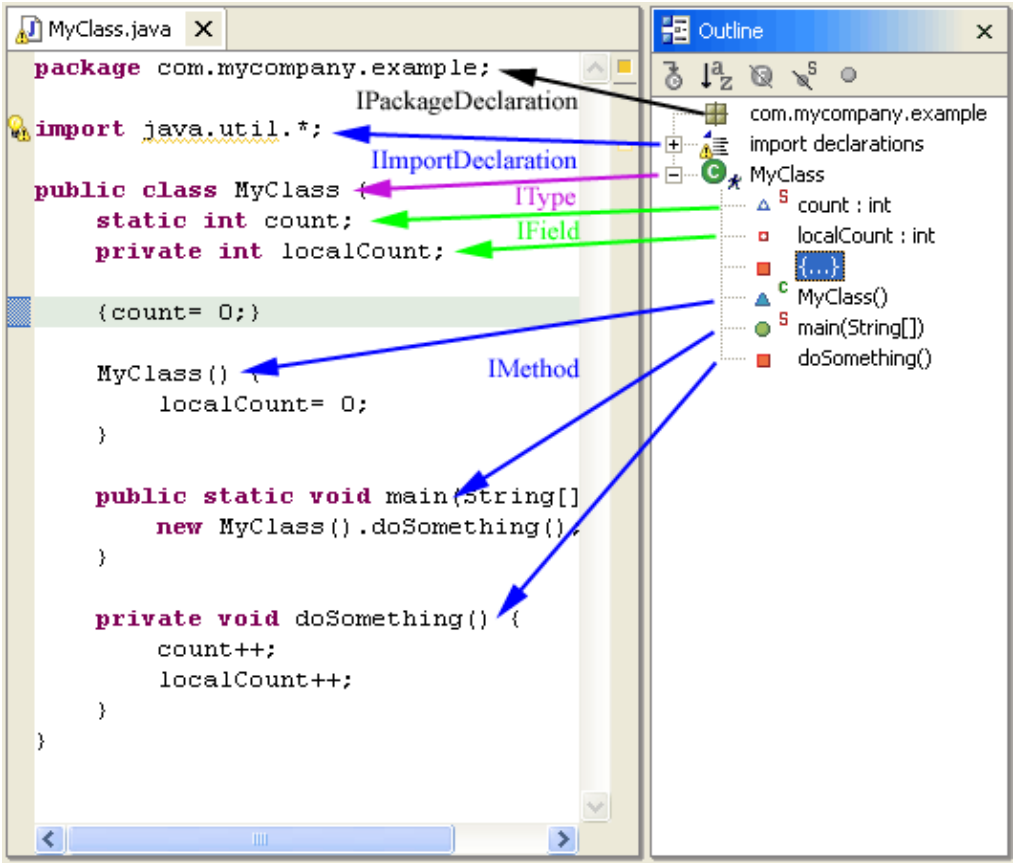
Some of the elements are shown in the Packages view. These elements implement the **IOpenable** interface, since they must be opened before they can be navigated. The figure below shows how these elements are represented in the Packages view.



The Java elements that implement **IOpenable** are created primarily from information found in the underlying resource files. The same elements are represented generically in the resource navigator view.



Other elements correspond to the items that make up a Java compilation unit. The figure below shows a Java compilation unit and a content outliner that displays the source elements in the compilation unit.



These elements implement the **ISourceReference** interface, since they can provide corresponding source code. (As these elements are selected in the content outliner, their corresponding source code is shown in the Java editor).

Java elements and their resources

Many of the Java elements correspond to generic resources in the workspace. When you want to create Java elements from a generic resource the class **JavaCore** is the best starting point. The following code snippet shows how to get Java elements from their corresponding resources.

```
    private void createJavaElementsFrom(IProject myProject, IFolder myFolder, IFile myFile) {
    IJavaProject myJavaProject= JavaCore.create(myProject);
    if (myJavaProject == null)
        // the project is not configured for Java (has no Java nature)
        return;

    // get a package fragment or package fragment root
    IJavaElement myPackageFragment= JavaCore.create(myFolder);

    // get a .java (compilation unit), .class (class file), or
    // .jar (package fragment root)
    IJavaElement myJavaFile = JavaCore.create(myFile);
}
```

Once you have a Java element, you can use the JDT API to traverse and query the model. You may also query the non–Java resources contained in a Java element.

```
    private void createJavaElementsFrom(IProject myProject, IFolder myFolder, IFile myFile)
```

```
...
// get the non Java resources contained in my project.
Object[] nonJavaChildren = myJavaProject.getNonJavaResources();
...
```

Java projects

When you create a Java project from a simple project, **JavaCore** will check to see if the project is configured with the Java nature. The JDT plug−in uses a project nature to designate a project as having Java behavior. This nature (**org.eclipse.jdt.core.JavaCore#NATURE_ID**) is assigned to a project when the "New Java project" wizard creates a project. If the Java nature is not configured on a project, **JavaCore** will return null when asked to create the project.

JavaCore is also used to maintain the Java class path, including locations for finding source code and libraries, and locations for generating output binary (**.class**) files.

What are the unique characteristics of Java projects? They record their classpath in a "**.classpath**" file and add the Java incremental project builder to the project's build spec. Otherwise, they are just regular projects and can be configured with other natures (and other incremental builders) by plug−ins. Plug−ins that want to configure projects with Java behavior in addition to their own behavior typically use the **NewJavaProjectWizardPage** to assign the Java nature to the project in addition to their own custom natures or behavior.

IJavaModel can be considered the parent of all projects in the workspace that have the Java project nature (and therefore can be treated as an **IJavaProject**).

Opening a Java editor

The following snippet shows how **JavaUI** can be used to open a Java editor and display a specific member of a Java compilation unit.

```
void showMethod(IMember member) {
    ICompilationUnit cu = member.getCompilationUnit();
    IEditorPart javaEditor = JavaUI.openInEditor(cu);
    JavaUI.revealInEditor(javaEditor, member);
}
```

The methods `openInEditor` and `revealInEditor` also work for class files and for members contained in class files.

Creating Java specific prompter dialogs

The following snippet uses the **JavaUI** class to open the Open Type dialog:

```
public IType selectType() throws JavaModelException {
    SelectionDialog dialog= JavaUI.createTypeDialog(
        parent, new ProgressMonitorDialog(parent),
        SearchEngine.createWorkspaceScope(),
        IJavaElementSearchConstants.CONSIDER_ALL_TYPES, false);
    dialog.setTitle("My Dialog Title");
    dialog.setMessage("My Dialog Message");
    if (dialog.open() == IDialogConstants.CANCEL_ID)
```

```
            return null;

        Object[] types= dialog.getResult();
        if (types == null || types.length == 0)
            return null;
        return (IType)types[0];
    }
```

JavaUI provides additional methods for creating Open Package and Open Main Type dialogs.

Presenting Java elements in a JFace viewer

The JDT UI API provides classes that allow you to present the Java model or parts of it in a standard JFace viewer. This functionality is provided primarily by:

- **StandardJavaElementContentProvider** – translates the Java element hierarchy into a data structure accessible by a tree, table or list viewer
- **JavaElementLabelProvider** – provides corresponding images and labels for a standard JFace viewer
- JavaElementLabels – provides corresponding text labels for Java elements

Content and label providers for JFace viewers are described in detail in JFace viewers.

If you understand the basic platform mechanism, then putting the Java content and label providers together is quite simple:

```
...
TreeViewer viewer= new TreeViewer(parent);
// Provide members of a compilation unit or class file, but no working copy elements
ITreeContentProvider contentProvider= new StandardJavaElementContentProvider(true, false);
viewer.setContentProvider(contentProvider);
// There are more flags defined in class JavaElementLabelProvider
ILabelProvider labelProvider= new JavaElementLabelProvider(
    JavaElementLabelProvider.SHOW_DEFAULT |
    JavaElementLabelProvider.SHOW_QUALIFIED |
    JavaElementLabelProvider.SHOW_ROOT);
viewer.setLabelProvider(labelProvider);
// Using the Java model as the viewers input present Java projects on the first level.
viewer.setInput(JavaCore.create(ResourcesPlugin.getWorkspace().getRoot()));
...
```

The example above uses aJava model (**IJavaModel**) as the input element for the viewer. The **StandardJavaElementContentProvider** also supports **IJavaProject**, **IPackageFragmentRoot**, **IPackageFragment**, and **IFolder** as input elements:

Overlaying images with Java information

JavaElementImageDescriptor can be used to create an image based on an arbitrary base image descriptor and a set of flags specifying which Java specific adornments (e.g. static, final, synchronized, ...) are to be superimposed on the image.

Adding problem and override decorators

When a viewer is supposed to include problem annotations, the JFace **DecoratingLabelProvider** together

with the **ProblemsLabelDecorator** is used. The snippet below illustrates the use of a problem label decorator.

```
...
DecoratingLabelProvider decorator= new DecoratingLabelProvider(labelProvider, new ProblemsLab
viewer.setLabelProvider(decorator);
...
```

In the same way the **OverrideIndicatorLabelDecorator** can be used to decorate a normal label provider to show the implement and override indicators for methods.

Updating the presentation on model changes

Neither the **OverrideIndicatorLabelDecorator** nor the **ProblemsLabelDecorator** listen to model changes. Hence, the viewer doesn't update its presentation if the Java or resource marker model changes. The reason for pushing the update onto the client for these classes is that there isn't yet a generic implementation that fulfills all performance concerns. Handling Java model delta inspection and viewer refreshing in each label decorator or provider would lead to multiple delta inspections and unnecessary viewer updates.

So what does the client need to do in order to update their viewers ?

- **OverrideIndicatorLabelDecorator**: the client must listen to Java model changes (see Responding to changes in Java elements) and decide if the change(s) described by the delta invalidates the override indicator of elements presented in the viewer. If so, the class inspecting the delta should trigger a repaint of the corresponding Java elements using the standard JFace viewer API (see update methods on StructuredViewer).
- **ProblemsLabelDecorator**: the client should listen to changes notified by the decorator via a **ProblemsLabelChangedEvent** (see also **ProblemsLabelDecorator.addListener**). Since the marker model is resource based, the listener has to map the resource notifications to its underlying data model. For an example showing how to do this for viewers presenting Java elements see the internal classes `ProblemTreeViewer.handleLabelProviderChanged.`

For the same reasons enumerated for label decorators the **StandardJavaElementContentProvider** doesn't listen to model changes. If the viewer needs to update its presentation according to Java model changes, then the client should add a corresponding listener to JavaCore. If the change described by the delta invalidates the structure of the elements presented in the viewer then the client should update the viewer using the standard JFace API (see refresh methods on StructuredViewer, and the add and remove methods on TableViewer and AbstractTreeViewer).

Sorting the viewer

JavaElementSorter can be plugged into a JFace viewer to sort Java elements according to the Java UI sorting style.

Programmatically Writing a Jar file

The **org.eclipse.ui.jarpackager** package provides utility classes to programmatically export files to a Jar file. Below is a code snippet that outlines the use of the **JarPackageData** class:

```
void createJar(IType mainType, IFile[] filestoExport) {
    Shell parentShell= ...;
```

```
        JarPackageData description= new JarPackageData();
        IPath location= new Path("C:/tmp/myjar.jar");
        description.setJarLocation(location);
        description.setSaveManifest(true);
        description.setManifestMainClass(mainType);
        description.setElements(filestoExport);
        IJarExportRunnable runnable= description.createJarExportRunnable(parentShell);
        try {
            new ProgressMonitorDialog(parentShell).run(true,true, runnable);
        } catch (InvocationTargetException e) {
            // An error has occurred while executing the operation
        } catch (InterruptedException e) {
            // operation has been canceled.
        }
    }
```

Additional API is provided to create a plug−in specific subclass of **JarPackageData**. This allows other plug−ins to implement their own Jar export/import wizards and to save the content of the **JarPackageData** object to a corresponding Jar description file.

Once the JAR is described by a **JarPackageData**, it can be programmatically written using a **JarWriter2**.

Java wizard pages

The **org.eclipse.jdt.ui.wizards** package provides wizard pages for creating and configuring Java elements. Several prefabricated pages are provided for your use.

Configuring Java build settings

JavaCapabilityConfigurationPage supports editing the Java build settings (source folder setup, referenced projects and, referenced and exported libraries).

If you need to provide a wizard that configures a project for your plug−in while also configuring it with the Java nature and other Java project capabilities, you should use this page (rather than subclassing **NewJavaProjectWizardPage**).

Creating new Java elements

A hierarchy of wizard pages support the creation of new Java elements.

NewElementWizardPage is the abstract class that defines the basic operation of the wizard. Additional abstract classes are provided in the hierarchy for making customizations to the functionality provided by the concrete wizards.

The concrete creation wizards can be used directly and generally are not intended to be subclassed.

- **NewClassWizardPage** allows users to define a new Java class. To customize the behavior for this wizard, you should subclass **NewTypeWizardPage**.
- **NewInterfaceWizardPage** allows users to define a new Java interface. To customize the behavior for this wizard, you should subclass **NewTypeWizardPage**.
- **NewEnumWizardPage** allows users to define a new Java enumeration. To customize the behavior for this wizard, you should subclass **NewTypeWizardPage**.

- **NewAnnotationWizardPage** allows users to define a new Java annotations. To customize the behavior for this wizard, you should subclass **NewTypeWizardPage**.
- **NewJavaProjectWizardPage** allows users to create a new Java project. To create a different kind of project with Java capabilities, you should use **JavaCapabilityConfigurationPage** where possible rather than subclassing this class.
- **NewPackageWizardPage** allows users to create a new Java package. To customize the behavior for this wizard, you should subclass **NewContainerWizardPage** rather than this class.

Contributing a classpath container wizard page

The interface **IClasspathContainerPage** defines a structure for contributing a wizard page that allows a user to define a new classpath container entry or edit an existing one. If your plug–in has defined its own type of classpath container using the JDT Core **org.eclipse.jdt.core.classpathContainerInitializer** extension point, then you will probably want to define a corresponding wizard page for editing and creating classpath containers of this type.

Your plug–in's markup should provide an extension **org.eclipse.jdt.ui.classpathContainerPage.** In the extension markup, you provide the name of your class that implements **IClasspathContainerPage**. If you want to provide additional information in your wizard page about a classpath's context when it is selected, you can implement **IClasspathContainerPageExtension** to initialize any state that depends on the entries selected in the current classpath.

Customizing a wizard page

Besides using prefabricated pages, you can subclass the wizard pages to add your own input fields or to influence the code generation. You should use the abstract classes in the **NewElementWizardPage** hierarchy to customize a wizard rather than subclassing the concrete classes.

Below is a sample of a new type wizard page that is customized to create JUnit test case classes. The page initializes the super class field with "junit.framework.TestCase" and adds a checkbox that controls whether method stubs for the `setUp()` and `tearDown()` method are to be created.

```
public class TestCaseWizardPage extends NewTypeWizardPage {
    private Button fCreateStubs;

    public TestCaseWizardPage() {
        super(true, "TestCaseWizardPage");
    }

    /**
     * The wizard managing this wizard page must call this method
     * during initialization with a corresponding selection.
     */
    public void init(IStructuredSelection selection) {
        IJavaElement jelem= getInitialJavaElement(selection);
        initContainerPage(jelem);
        initTypePage(jelem);
        doStatusUpdate();
    }

    private void doStatusUpdate() {
        // define the components for which a status is desired
        IStatus[] status= new IStatus[] {
            fContainerStatus,
```

```
            isEnclosingTypeSelected() ? fEnclosingTypeStatus : fPackageStatus,
            fTypeNameStatus,
        };
        updateStatus(status);
    }


    protected void handleFieldChanged(String fieldName) {
        super.handleFieldChanged(fieldName);

        doStatusUpdate();
    }

    public void createControl(Composite parent) {
        initializeDialogUnits(parent);
        Composite composite= new Composite(parent, SWT.NONE);
        int nColumns= 4;
        GridLayout layout= new GridLayout();
        layout.numColumns= nColumns;
        composite.setLayout(layout);

        // Create the standard input fields
        createContainerControls(composite, nColumns);
        createPackageControls(composite, nColumns);
        createSeparator(composite, nColumns);
        createTypeNameControls(composite, nColumns);
        createSuperClassControls(composite, nColumns);

        // Create the checkbox controlling whether we want stubs
        fCreateStubs= new Button(composite, SWT.CHECK);
        fCreateStubs.setText("Add 'setUp()' and 'tearDown()' to new class");
        GridData gd= new GridData();
        gd.horizontalSpan= nColumns;
        fCreateStubs.setLayoutData(gd);

        setControl(composite);

        // Initialize the super type field and mark it as read-only
        setSuperClass("junit.framework.TestCase", false);
    }

    protected void createTypeMembers(IType newType, ImportsManager imports, IProgressMonitor moni
        if (fCreateStubs.getSelection()) {
            String setUpMathod= "public void setUp() {}";
            newType.createMethod(setUpMathod, null, false, null);

            String tearDownMathod= "public void setUp() {}"
            newType.createMethod(tearDownMathod, null, false, null);
        }
    }
}
```

Notices

About This Content

February 24, 2005

License