

JDT fundamentals – Become a JDT tool smith

Martin Aeschlimann

IBM Research, Switzerland

martin_aeschlimann@ch.ibm.com

Overview – The 3 Pillars

Java Model – Lightweight model for views

- OK to keep references to it
- Contains unresolved information
- From project to declarations (types, methods..)

Search Engine

- Indexes of declarations, references and type hierarchy relationships

AST – Precise, fully resolved compiler parse tree

- No references to it must be kept: Clients have to make sure only a limited number of ASTs is loaded at the same time
- Fully resolved information
- From a Java file ('Compilation Unit') to identifier tokens

The 3 Pillars – First Pillar: Java Model



Java Model – Lightweight model for views

- Java model and its elements
- Classpath elements
- Java project settings
- Creating a Java element
- Change notification
- Type hierarchy
- Code resolve

Search Engine

AST – Precise, fully resolved compiler parse tree

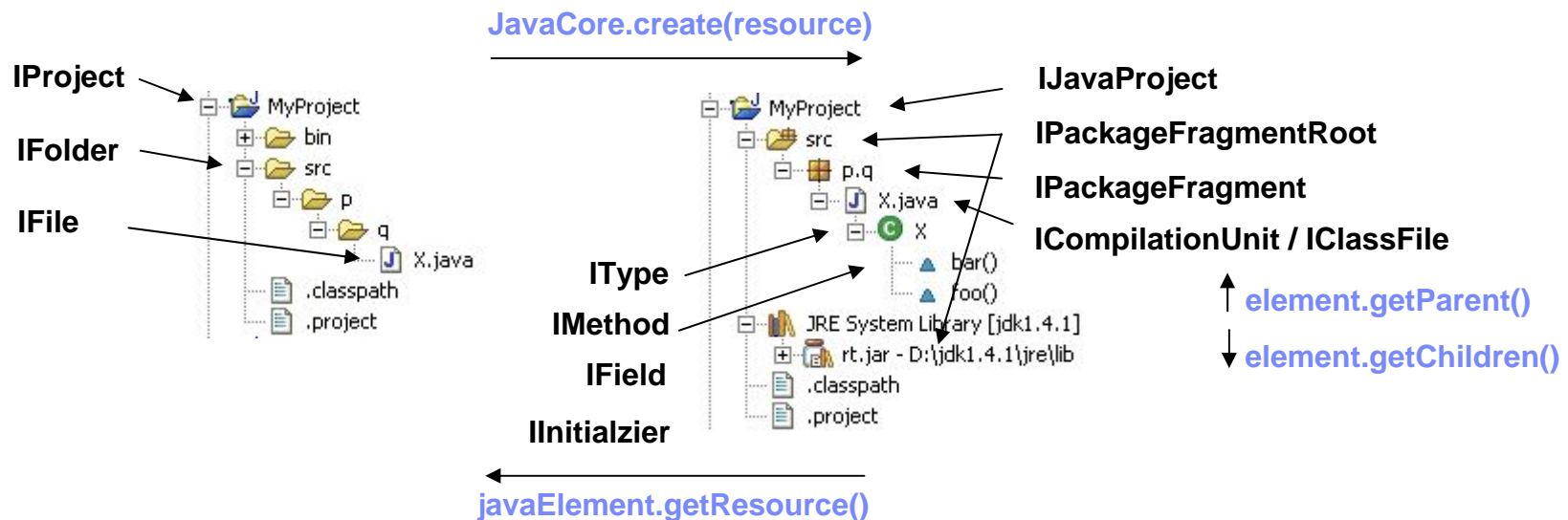
Java Elements API

IJavaElement form a hierarchy that renders the entire workspace from Java angle

Separated hierarchy from resources:

Important to note:

- Not all Java elements must have an underlying resource (elements inside a JAR, external JAR files)
- A Java package doesn't have the same children as a folder (no concept of subfolder)



Using the Java Model

Setting up a Java project

- A Java project is a project with the Java nature set
- Java nature enables the Java builder
- Java builder needs a Java class path

```
IWorkspaceRoot root= ResourcesPlugin.getWorkspace().getRoot();  
IProject project= root.getProject(projectName);  
project.create(null);  
project.open(null);
```

Create a project

```
IProjectDescription description = project.getDescription();  
description.setNatureIds(new String[] { JavaCore.NATURE_ID });  
project.setDescription(description, null);
```

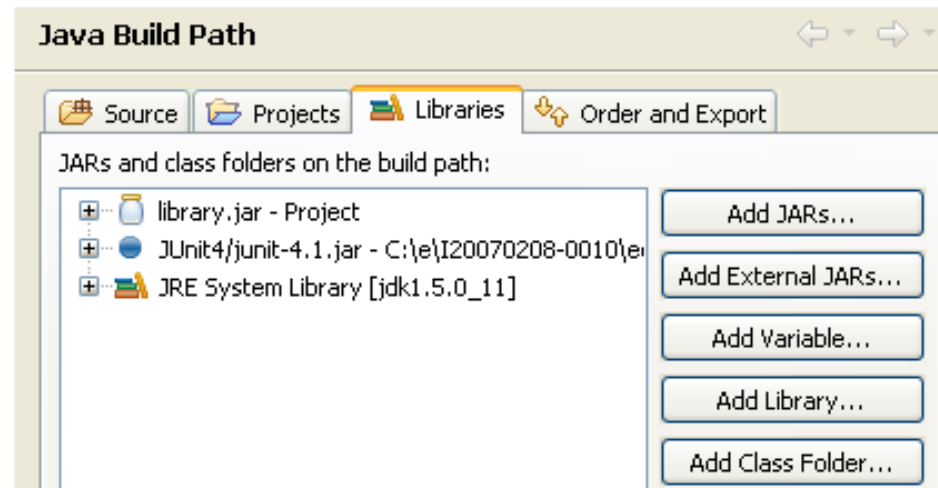
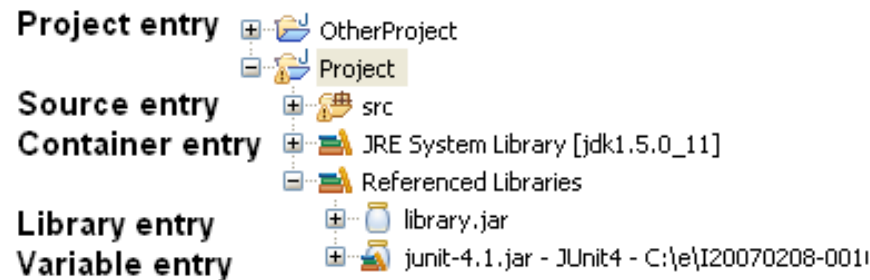
**Set the
Java nature**

```
IJavaProject javaProject= JavaCore.create(project);  
javaProject.setRawClasspath(classPath, defaultOutputLocation, null);
```

**Set the Java
build path**

Java Classpath

The Java element hierarchy is defined by the Java classpath:
Classpath entries define the roots of package fragments.



Classpath – Source and Library Entries



Source entry: Java source files to be built by the compiler

- Folder inside the project or the project itself
- Possibility to define inclusion and exclusion filters
- Compiled files go to either a specific or the projects default output location

```
IPath srcPath= javaProject.getPath().append("src");  
IPath[] excluded= new IPath[] { new Path("doc") };  
IClasspathEntry srcEntry= JavaCore.newSourceEntry(srcPath, excluded);
```

Library entry: Class folder or archive

- Class files in folders, archive in workspace or external
- Source attachment specifies location of libraries source

New in 3.4: External class folders

Creating Java Elements



```
IJavaProject javaProject= JavaCore.create(project);  
IClasspathEntry[] buildPath= {  
    JavaCore.newSourceEntry(project.getFullPath().append("src")),  
    JavaRuntime.getDefaultJREContainerEntry()  
};  
javaProject.setRawClasspath(buildPath, project.getFullPath().append("bin"), null);
```

Set the build path

```
IFolder folder= project.getFolder("src");  
folder.create(true, true, null);
```

Create the source folder

```
IPackageFragmentRoot srcFolder= javaProject.getPackageFragmentRoot(folder);  
Assert.assertTrue(srcFolder.exists()); // resource exists and is on build path
```

Create the package fragment

```
IPackageFragment fragment= srcFolder.createPackageFragment("x.y", true, null);
```

```
String str=  
    "package x.y;"           + "\n" +  
    "public class E {"       + "\n" +  
    "    String first;"      + "\n" +  
    "}";  
ICompilationUnit cu= fragment.createCompilationUnit("E.java", str, false, null);
```

Create the compilation unit, including a type

```
IType type= cu.getType("E");
```

Create a field

```
type.createField("String name;", null, true, null);
```


Java Project Settings



Configure compiler settings on the project

- Compiler compliance, class file compatibility, source compatibility
- Compiler problems severities (Ignore/Warning/Error)

```
javaProject.setOption(JavaCore.COMPILER_COMPLIANCE, JavaCore.VERSION_1_5);
```

If not set on the project, taken from the workspace settings

- Project settings persisted (project/.settings/org.eclipse.jdt.core.prefs).
Shared in a team
- More project specific settings: Formatter, code templates...

See Platform preferences story

- Platform.getPreferencesService()

Working Copies

- A compilation unit in a buffered state is a working copy
- Primary working copy: shared buffer shown by all editors
 - based on the eclipse.platforms buffer manager (plug-in `org.eclipse.core.filebuffers`)
 - `becomeWorkingCopy(...)`: Increment count, internally create buffer, if first
 - `commitWorkingCopy()`: Apply buffer to underlying resource
 - `discardWorkingCopy()`: Decrement count, discard buffer, if last
 - Element stays the same, only state change
- Private working copy: Build a virtual Java model layered on top of the current content
 - `ICompilationUnit.getWorkingCopy(workingCopyOwner)` returns a new element with a new buffer (managed by the `workingCopyOwner`) based on the underlying element
 - `commitWorkingCopy()`: Apply changes to the underlying element
 - Refactoring uses this to first try all changes in a sandbox to only apply them if compilable
- Working copy owner: Connects working copies so that they reference each other

Java Element Change Notifications

Change Listeners: `JavaCore.addElementChangeListener(IElementChangeListener)`

- Java element delta information for all changes: class path changes, added/removed elements, changed source, change to buffered state (working copy)
- Changes triggered by resource change notifications (resource deltas), call to 'reconcile()'

IJavaElementDelta: Description of changes of an element or its children

Delta kind	Descriptions and additional flags	
ADDED	Element has been added	
REMOVED	Element has been removed	
CHANGED	F_CONTENT	Content has changed. If F_FINE_GRAINED is set: Analysis of structural changed has been performed
	F_MODIFIERS	Changed modifiers
	F_CHILDREN	Deltas in children <code>IJavaElementDelta[] getAffectedChildren()</code>
	F_ADDED_TO_CLASSPATH, F_SOURCEATTACHED, F_REORDER, F_PRIMARY_WORKING_COP,...	

JavaElementListener – an Example

Find out if types were added or removed

```
fJavaListener= new IElementChangeListener() {
    public void elementChanged(ElementChangedEvent event) {
        boolean res= hasTypeAddedOrRemoved(event.getDelta());
    }
    private boolean hasTypeAddedOrRemoved(IJavaElementDelta delta) {
        IJavaElement elem= delta.getElement();
        boolean isAddedOrRemoved= (delta.getKind() != IJavaElementDelta.CHANGED);
        switch (elem.getElementType()) {
            case IJavaElement.JAVA_MODEL: case IJavaElement.JAVA_PROJECT:
            case IJavaElement.PACKAGE_FRAGMENT_ROOT: case IJavaElement.PACKAGE_FRAGMENT:
                if (isAddedOrRemoved) return true;
                return processChildrenDelta(delta.getAffectedChildren());
            case IJavaElement.COMPILOATION_UNIT:
                ICompilationUnit cu= (ICompilationUnit) elem;
                if (!cu.getPrimary().equals(cu))
                    return false;
                if (isAddedOrRemoved || isPossibleStructuralChange(delta.getFlags()))
                    return true;
                return processChildrenDelta(delta.getAffectedChildren());
            case IJavaElement.TYPE:
                if (isAddedOrRemoved) return true;
                return processChildrenDelta(delta.getAffectedChildren()); // inner types
            default: // fields, methods, imports...
                return false;
        }
    }
}
```

**Parent constructs:
Recursively go
down the delta tree**

**Be aware of
private working
copies**

Code Resolve

- Resolve the element at the given offset and length in the source

```
javaElements= compilationUnit.codeResolve(50, 10);
```

- Used for Navigate > Open (F3) and tool tips

```

* <i>This Java element does not exist (ELEMENT_DOES_NOT_EXIST)</i>
* <i>The range specified is not within this element's
*   source range (INDEX_OUT_OF_BOUNDS)
* </ul>
*
*
*
IJavaElement[] codeSelect(int offset, int length) throws JavaModelException;

```

org.eclipse.jdt.core.IJavaElement

Common protocol for all elements provided by the Java model. Java model elements are exposed to clients as handles to the actual underlying element. The Java model may hand out any number of handles for each element. Handles that refer to the same element are guaranteed to be equal, but not necessarily identical. Methods annotated as "handle-only" do not require

Press 'F2' for focus.

```

</p>
*

```

Code Resolve – an Example

Resolving the reference to “String” in a compilation unit

**Set up a
compilation unit**



```
String content =  
    "public class X {" + "\n" +  
    "    String field;" + "\n" +  
    "}";  
ICompilationUnit cu=  
    fragment.createCompilationUnit("X.java", content, false, null);
```

```
int start = content.indexOf("String");  
int length = "String".length();  
IJavaElement[] declarations = cu.codeSelect(start, length);
```

**Contains a single IType:
'java.lang.String'**



Second Pillar: Search Engine

Java Model – Lightweight model for views

Search Engine

- Design motivation
- Using the search engine
- Code example

AST – Precise, fully resolved compiler parse tree

Search Engine

- Indexed search for declarations and references
 - packages, types, fields, methods and constructors
 - using wildcards (including camel-case) or from a Java element
- Scoped search
 - scope = set of Java elements
 - predefined workspace and hierarchy scopes
- Precise and non-precise matches
 - Code with errors, incomplete class paths

- **New in 3.4:** Limit the match locations
 - in casts, in catch clauses, only return types...

Search Engine – Using the APIs



- Creating a search pattern

```
SearchPattern.createPattern("foo*",  
    IJavaSearchConstants.FIELD, IJavaSearchConstants.REFERENCES,  
    SearchPattern.R_PATTERN_MATCH | SearchPattern.R_CASE_SENSITIVE);
```

- Creating a search scope

```
SearchEngine.createWorkspaceScope();  
SearchEngine.createJavaSearchScope(new IJavaElement[] { project });  
SearchEngine.createHierarchyScope(type);
```

- Collecting results

- Subclass SearchRequestor
- Results are reported as SearchMatch

Search Engine – an Example



Searching for all declarations of methods “foo” that return an int

```
SearchPattern pattern = SearchPattern.createPattern(  
    "foo(*) int",  
    IJavaSearchConstants.METHOD,  
    IJavaSearchConstants.DECLARATIONS,  
    SearchPattern.R_PATTERN_MATCH);
```

Search pattern

```
IJavaSearchScope scope = SearchEngine.createWorkspaceScope();
```

Search scope

```
SearchRequestor requestor = new SearchRequestor() {  
    public void acceptSearchMatch(SearchMatch match) {  
        System.out.println(match.getElement());  
    }  
};
```

Result collector

```
SearchEngine searchEngine = new SearchEngine();  
searchEngine.search(  
    pattern,  
    new SearchParticipant[] { SearchEngine.getDefaultSearchParticipant() },  
    scope,  
    requestor,  
    null /*progress monitor*/);
```

Start search

The 3 Pillars – Third Pillar: AST

Java Model – Lightweight model for views

Search Engine

AST – Precise, fully resolved compiler parse tree

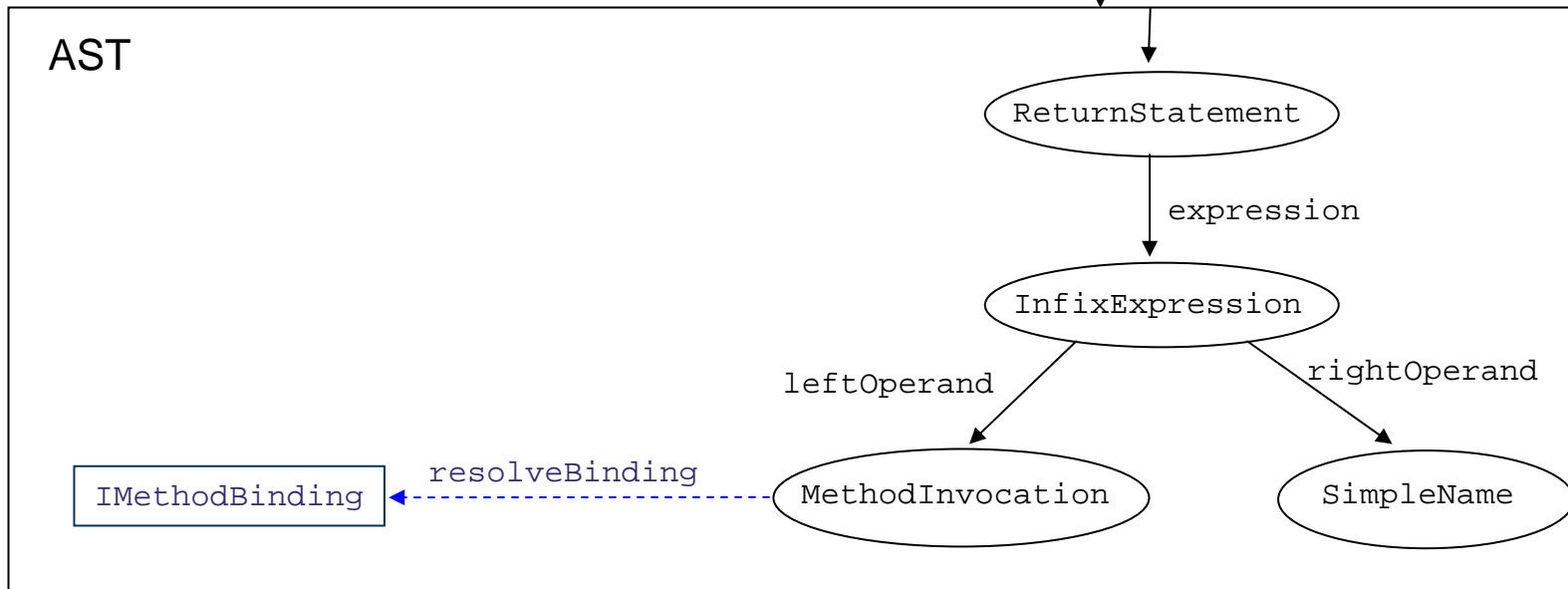
- Overall design
- Creating an AST
- AST node details
- Bindings
- AST rewrite
- Refactoring toolkit

Abstract Syntax Tree

Source Code

```
return getPrefix() + count;
```

ASTParser#createAST(...)



Creating an AST

- Build AST from element or source
 - AST factory: **ASTParser**
 - Either from Java model elements: ICompilationUnit, IClassFile (ITypeRoot)
 - Or source string, file name and IJavaProject as context
- Bindings or no bindings
 - Bindings contain resolve information. Fully available on syntax errors free code, best effort when there are errors.
- Full AST or partial AST
 - For a given source position: All other method have empty bodies
 - AST for an element: Only method, statement or expression

Creating an AST

```
ASTParser parser= ASTParser.newParser(AST.JLS3);  
parser.setSource(cu);  
parser.setResolveBindings(true);  
parser.setStatementsRecovery(true);  
ASTNode node= parser.createAST(null);
```

Create AST on an element

```
ASTParser parser= ASTParser.newParser(AST.JLS3);  
parser.setSource("System.out.println();".toCharArray());  
parser.setProject(javaProject);  
parser.setKind(ASTParser.K_STATEMENTS);  
parser.setStatementsRecovery(false);  
ASTNode node= parser.createAST(null);
```

Create AST on source string

AST Browsing

Typed access to the node children:

ConditionalExpression:

```
getExpression()  
getThenExpression()  
getElseExpression()
```

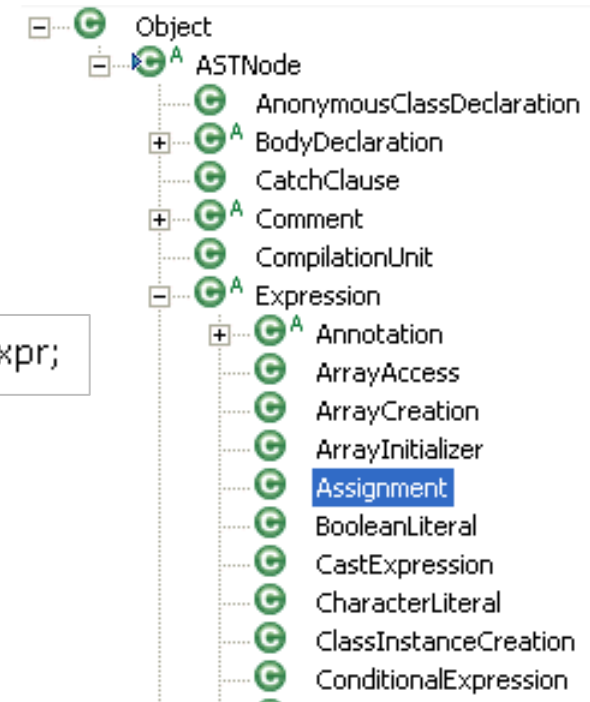
```
expr ? thenExpr : elseExpr;
```

Homogenous access using node properties:

```
List allProperties= node.structuralPropertiesForType();
```

Will contain 3 elements of type 'StructuralPropertyDescriptor':
ConditionalExpression.EXPRESSION_PROPERTY,
ConditionalExpression.THEN_EXPRESSION_PROPERTY,
ConditionalExpression.ELSE_EXPRESSION_PROPERTY,

```
expression=  
node.getStructuralProperty(ConditionalExpression.EXPRESSION_PROPERTY);
```



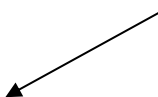
AST Visitor

```
ASTParser parser= ASTParser.newParser(AST.JLS3);
parser.setSource(cu);
parser.setResolveBindings(true);
```


```
ASTNode root= parser.createAST(null);
root.accept(new ASTVisitor() {
```

```
public boolean visit(CastExpression node) {
    fCastCount++;
    return true;
}
```

Count the number of casts



Count the number of references to a field of 'java.lang.System' ('System.out', 'System.err')



```
public boolean visit(SimpleName node) {
    IBinding binding= node.resolveBinding();
    if (binding instanceof IVariableBinding) {
        IVariableBinding varBinding= (IVariableBinding) binding;
        ITypeBinding declaringType= varBinding.getDeclaringClass();
        if (varBinding.isField() &&
            "java.lang.System".equals(declaringType.getQualifiedName())) {
            fAccessesToSystemFields++;
        }
    }
    return true;
}
```


AST Rewriting



- Instead of manipulating the source code change the AST and write changes back to source
- Descriptive approach
 - describe changes without actually modifying the AST
 - allow reuse of the AST over several operations
 - support generation of a preview
- Modifying approach
 - directly manipulates the AST
 - API is more intuitive
 - implemented using the descriptive rewriter
- Rewriter characteristics
 - preserve user formatting and markers
 - generate an edit script

Implementation of descriptive rewrite is currently more powerful:

- String placeholders: Use a node that is a placeholder for an arbitrary string of code or comments
- Track node positions: Get the new source ranges after the rewrite
- Copy a range of nodes
- Modify the comment mapping heuristic used by the rewriter (comments are associated with nodes. Operation on nodes also include the associated comments)

AST Rewrite

Example of the descriptive AST rewrite:

```
public void modify(MethodDeclaration decl) {
```

```
    AST ast= decl.getAST();
```

Create the rewriter

```
    ASTRewrite astRewrite= ASTRewrite.create(ast);
```

Change the method name

```
    SimpleName newName= ast.newSimpleName("newName");
    astRewrite.set(decl, MethodDeclaration.NAME_PROPERTY, newName, null);
```

```
    ListRewrite paramRewrite=
        astRewrite.getListRewrite(decl, MethodDeclaration.PARAMETERS_PROPERTY);
```

```
    SingleVariableDeclaration newParam= ast.newSingleVariableDeclaration();
    newParam.setType(ast.newPrimitiveType(PrimitiveType.INT));
    newParam.setName(ast.newSimpleName("p1"));
```

```
    paramRewrite.insertFirst(newParam, null);
```

```
    TextEdit edit= astRewrite.rewriteAST(document, null);
```

Insert a new parameter as first parameter

```
    edit.apply(document);
```

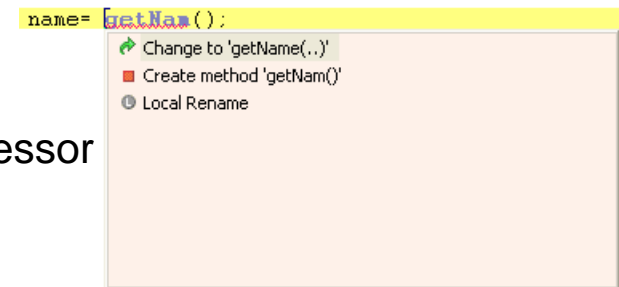
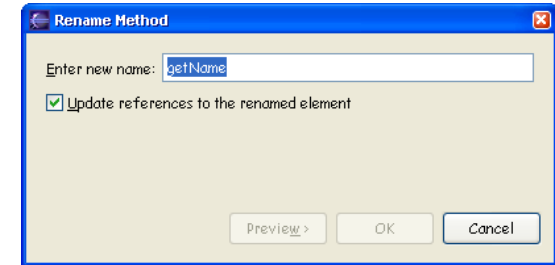
Create resulting edit script

Apply edit script to source buffer

```
}
```

Code Manipulation Toolkits

- Refactoring – org.eclipse.ltk.refactoring
 - refactorings - org.eclipse.ltk.core.refactoring.Refactoring
 - responsible for precondition checking
 - create code changes
 - code changes - org.eclipse.ltk.core.refactoring.Change
 - provide Undo/Redo support
 - support non-textual changes (e.g. renaming a file)
 - support textual changes based on text edit support
 - user interface is dialog based
- Quick fix & Quick Assist – org.eclipse.jdt.ui.text.java
 - AST based
 - processors - org.eclipse.jdt.ui.text.java.IQuickFixProcessor
 - check availability based on problem identifier
 - generate a list of fixes
 - user interface is provided by editor



Summary



- JDT delivers powerful program manipulation services
 - Java Model, Search engine and DOM AST
 - Add your own tool to the Eclipse Java IDE
 - but also in headless mode (can be used programmatically)
 - Visual Editor, EMF, metric tools, ...
 - Full J2SE 5.0/6.0 support

- Community feedback is essential
 - bug reports: `http://bugs.eclipse.org/bugs`
 - mailing lists: `http://www.eclipse.org/mail/index.html`
 - newsgroups: `news://news.eclipse.org/eclipse.tools.jdt`

Legal Notice

- Copyright © IBM Corp., 2007-2008. All rights reserved. Source code in this presentation is made available under the EPL, v1.0, remainder of the presentation is licensed under Creative Commons Att. Nc Nd 2.5 license.
- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- Eclipse and the Eclipse logo are trademarks of Eclipse Foundation, Inc.
- Other company, product, or service names may be trademarks or service marks of others.
- THE INFORMATION DISCUSSED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, SUCH INFORMATION. ANY INFORMATION CONCERNING IBM'S PRODUCT PLANS OR STRATEGY IS SUBJECT TO CHANGE BY IBM WITHOUT NOTICE