Java Tool Smithing
Extending the Eclipse Java Development Tools

Martin Aeschlimann
IBM Research, Switzerland
martin_aeschlimann@ch.ibm.com

Dirk Bäumer
IBM Research, Switzerland
dirk_baeumer@ch.ibm.com

Jerome Lanneluc
IBM Rational Software
jerome_lanneluc@fr.ibm.com

Outline

- A guided tour through services offered by JDT core and JDT UI
  - Java Model
  - Search Engine
  - Abstract Syntax Tree (AST)
  - What's new in 3.1: J2SE 5.0

- Each part is associated with more detailed information, possibly with code example, targeted for Java tool developers

- 3 'Hands-on' parts where we implement some small example applications

The Java Model - Design Motivation

Requirements for a Java model:

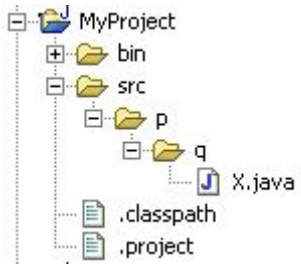
- Light weight
  - Need elements to which a reference can be kept, e.g. to show in a viewer
  - Must work for big workspaces (10’000 types and more). Can not hold on resources, Eclipse is not just a Java IDE
- Fault tolerant
  - Some source does not (yet) compile, missing brackets, semicolons. Tooling should be as helpful as possible
  - Viewers like the outline want to show the structure while typing. Structure should stay as stable as possible

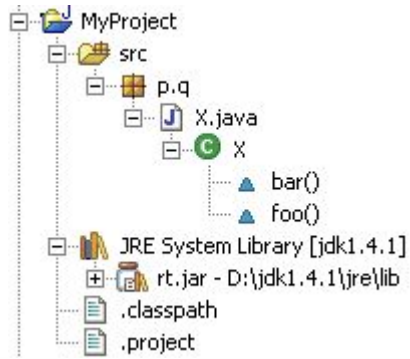
Chosen solution:

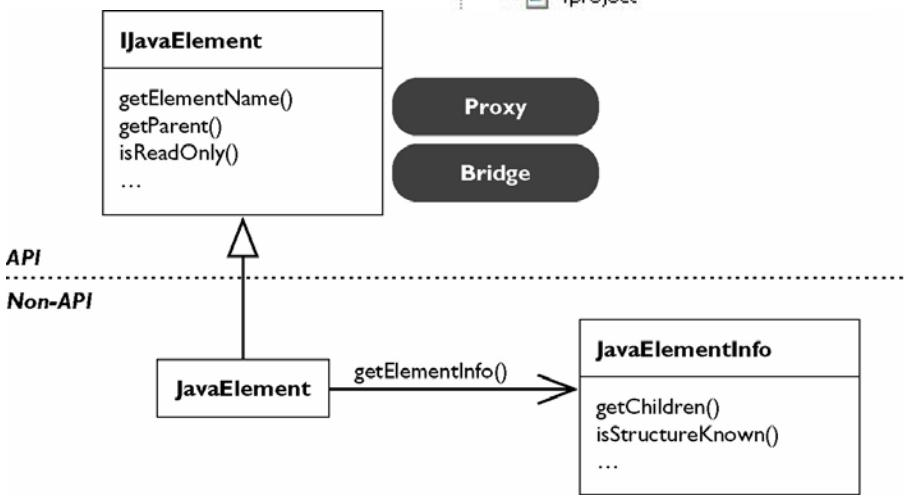
- Handle based, lazily populated model
- No resolved information kept
- Wrappers existing resource model

The Java Model

- Render entire workspace from Java angle
  - using '.classpath'
    - classpath entry is package fragment root
    - can even denote JAR outside workspace
  - granularity down to individual fields or methods
- Pure source model
  - accurate independently of build actions
  - fault-tolerant
  - no resolved information
- Handle/Info design
  - scalability: model non exhaustive
  - info lazily populated, LRU cache
  - stable handle

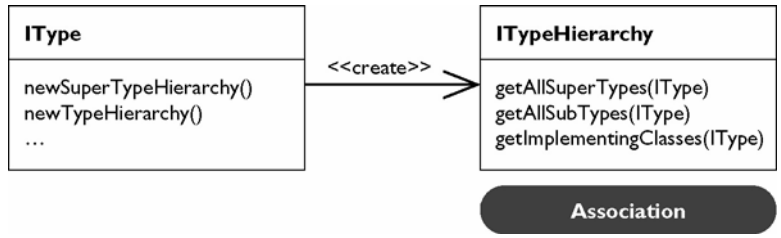


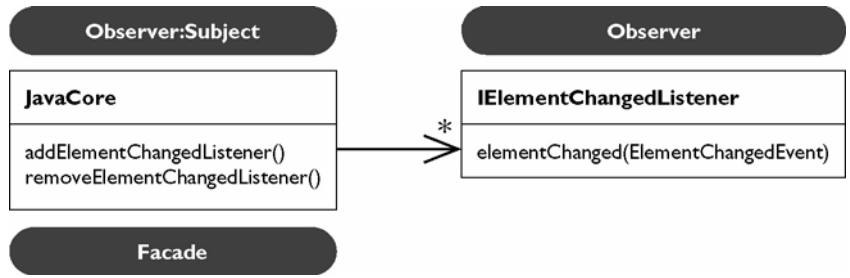


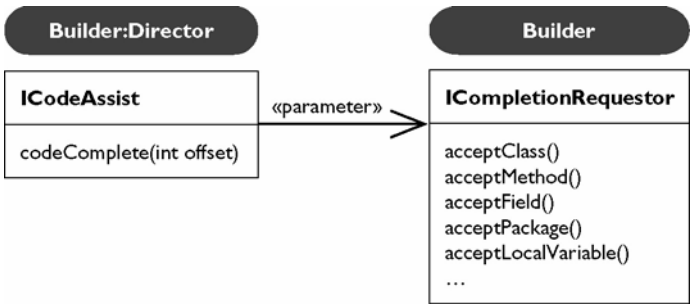


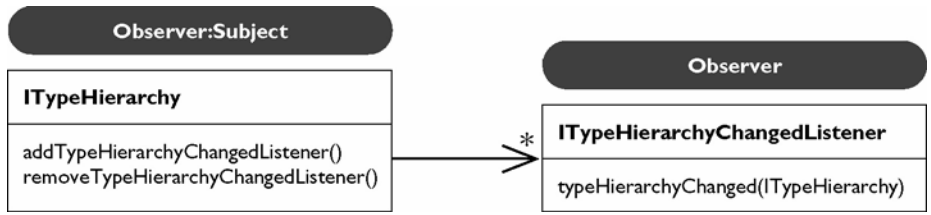
The Java Model Offering

- Navigation
- Type hierarchies
- Self-updating with change notification
- Basic modifying operations
- Many more…
  - code completion, selection
  - formatting, sorting
  - resolving, evaluating









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(null);
```

Set the Java nature

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

Set the Java build path

Java Classpath: Source & Project entries

Classpath entries define the roots of package fragments

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);
```

Project entry: Dependency to another project

- Inherit all exported classpath entries from an other project
- Each project can specify its exported entries.
- Project output locations (containing compiled source files) are automatically exported

```
IClasspathEntry prjEntry=
  JavaCore.newProjectEntry(otherProject.getPath(), true);

IClasspathEntry[] classPath= { srcEntry, prjEntry };
javaProject.setRawClasspath(classPath, defaultOutputLocation, null);
```

Java Classpath: Library & Variable entries

Library entry: Class folder or archive

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

Variable entry: Class folder or archive through an indirection

- Path relative to a ‘class path variable’
- Variable points to a (external) folder or file
- Variable absorbs file system dependent path: Sharing in a team
- `JavaCore.set/getClasspathVariable(name)`
- Variable initialization in extension point ‘org.eclipse.jdt.core. classpathVariableInitializer’

Java Classpath: Container entries

Container entry: Multiple entries through an indirection

- Path denotes name and arguments for a ‘classpath container’

```
entry= JavaCore.newContainerEntry(new Path("containerId/containerArguments"));
```

- Classpath containers are conributed by extension point
- Classpath containers can compute classpath entries at call time
- Build-in containers: JRE, User library, PDE dependencies

```
jreCPEntry= JavaCore.newContainerEntry(new Path(JavaRuntime.JRE_CONTAINER));
```

- Extension point ‘org.eclipse.jdt.core.classpathContainerInitializer’
  - Initializes and manages containers (using JavaCore.setClasspathContainer(..))
- Extension point ‘org.eclipse.jdt.ui.classpathContainerPage’
  - Contributes a classpath container configuration page

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

New in Eclipse 3.1:

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

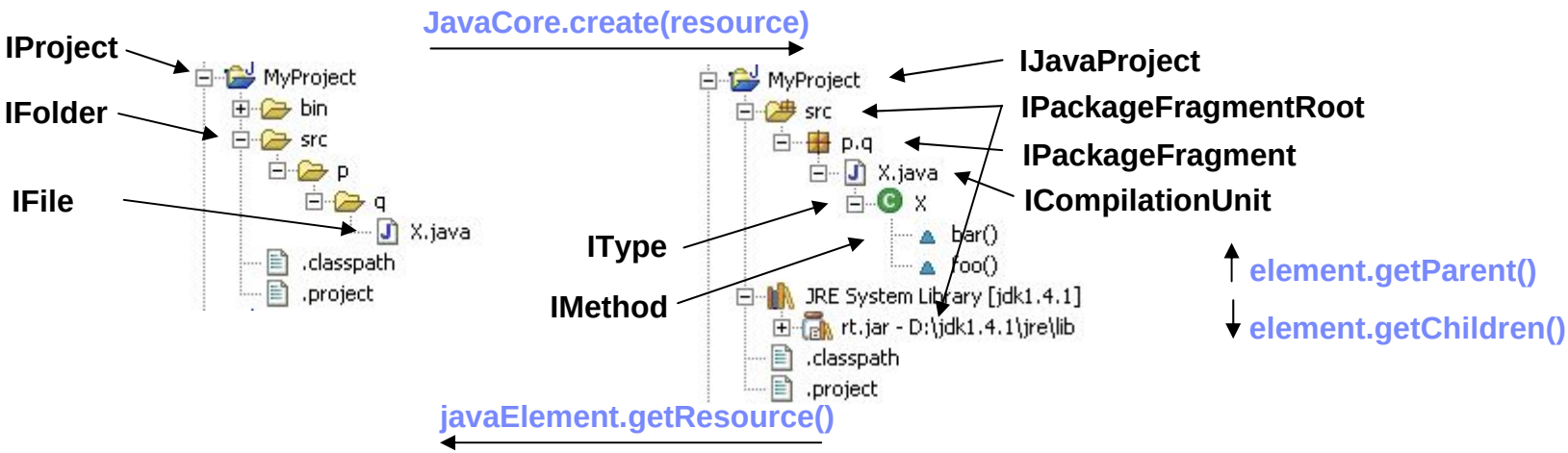
See new preferences story

- Platform.getPreferencesService()

Java Elements

Hierarchy of IJavaElement reflecting the Java world

- Separate hierarchy from Resources: IJavaElements have underlying resources, but one resource can contain several Java elements



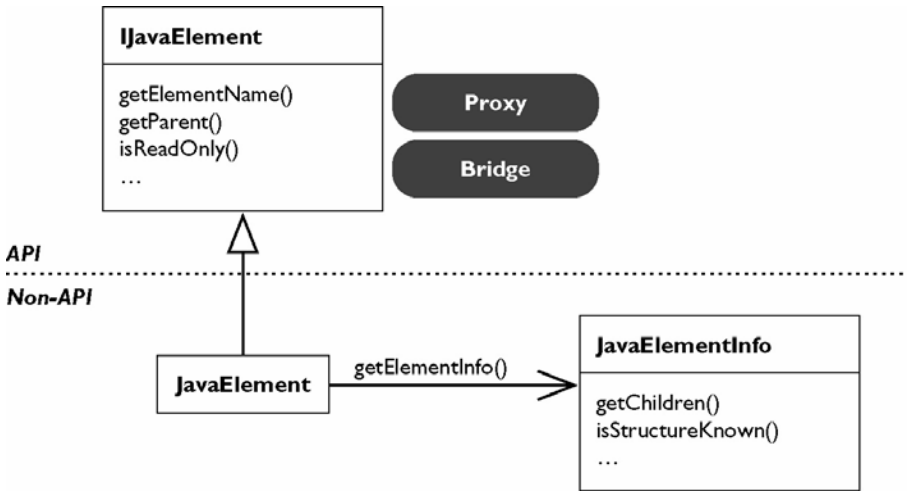
- Completely source based, independent of builder
- Fault tolerant
- Source ranges of source elements and their names
- No resolved information: Code resolve required to get e.g. a referenced type
- Very lightweight: only handles

Java Element Handles

Handle/Info design

- IJavaElement objects are lightweight: Ok to keep references
- Underlying buffer created on demand
- Element doesn't need to exist or be on the build path (anymore). Use IJavaElement#exists() to test
- Handle representation stable between workspace instances

```
String handleId= javaElement.getHandleIdentifier();
IJavaElement elem= JavaCore.create(handleId);
```



Creating Java Elements

Set the build path

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

Create the source folder

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

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);
```

Create the compilation unit, including a type

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

Create a field

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

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

Get your hands wet

- Programmatically create project, set build path, create package and compilation units
- Use the PDE tooling to create an action that invokes your code or use the example code provided by us.

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 (plugin 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 applies them if compilable
- Working copy owner: Connects working copies so that they reference each other

Working Copies

Opening Java editor

ICompilationUnit
becomeWorkingCopy()

ITextFileBufferManager
connect()

Saving Java editor

ICompilationUnit
commitWorkingCopy()

ITextFileBufferManager
commit()

Close Java editor

ICompilationUnit
discardWorkingCopy()

ITextFileBufferManager
disconnect()

- IJavaModel uses IBuffer that wraps an IDocument managed by the file buffers plugin
- No API to get a IDocument from an IBuffer: Work with file buffers directly if you need IDocuments

```
bufferManager= FileBuffers.getTextFileBufferManager();
IPath path= compilationUnit.getPath();
try {
  bufferManager.connect(path, null);
  IDocument document= bufferManager.getTextFileBuffer(path).getDocument();
  // work with document
} finally {
  bufferManager.disconnect(path, null);
}
```

Increments buffer reference count

Decrements buffer reference count

Java Element Change Notifications

Change Listeners: `JavaCore.addElementChangedListener(IElementChangedListener)`

- 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 `IJavaElementDelta[] getAffectedChildren()`
	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 IElementChangedListener() {

  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.COMPILATION_UNIT:
        ICompilationUnit cu= (ICompilationUnit) elem;
        if (cu.getPrimary().equals(cu)) {
          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

JavaElementListener – cont'd

```
private static boolean isPossibleStructuralChange(int flags) {
  return hasSet(IJavaElementDelta.F_CONTENT) && !hasSet(IJavaElementDelta.F_FINE_GRAINED));
}

private boolean processChildrenDelta(IJavaElementDelta[] children) {
  for (int i= 0; i < children.length; i++) {
    if (hasTypeAdderOrRemoved(children[i]))
      return true;
  }
  return false;
}
```

'Fine Grained' set means that children deltas have been computed. If not it is a unknown change (potentially full change)

Visit delta children recursively

Type Hierarchy - Design Motivation

Subtype hierarchies are expensive to create and maintain.

Why not having an API IType.getSubtypes()?

- Requires to index and resolve all compilation unit in a project and project dependants. Takes minutes for a normal-sized workspace

Why not keep a constantly updated hierarchy in memory?

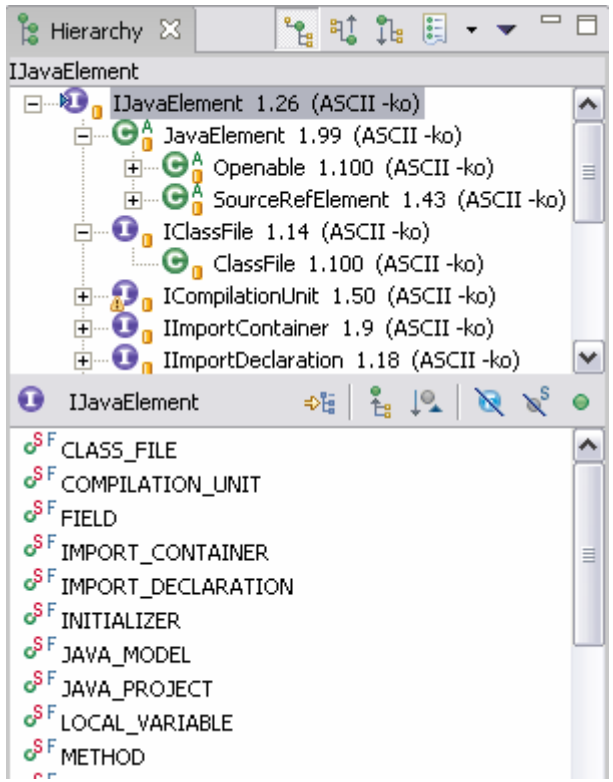
- Does not scale for big workspaces. Eclipse is not just a Java IDE (can not hold on 10 MB of structure)
- Expensive updating. Every class path change would require types to recheck if they still resolve to the same type

Chosen solution:

- Instantiated hierarchy object
  - Defined life cycle
  - Well known creation costs (sub type relationship is stored in index files)

Type Hierarchy

- Connect ITypes in a sub/super type relationship
- Used in Hierarchy view



Type Hierarchy

- Source based operation
  - No need for build state
  - Connect source types (.java) and binary types (.class)
  - Uses same indexes as search
  - Can be expensive (index query + resolution on the fly)
- Scoping the hierarchy
  - Hierarchy on IRegion (= set of IJavaElements)
  - Hierarchy on working copies (or WorkingCopyOwner)
  - Supertype hierarchy (much faster)
- Listening for hierarchy changes
  - ITypeHierarchy#addTypeHierarchyChangeListener(…)
  - ITypeHierarchy#refresh(…)
  - ITypeHierarchy#removeTypeHierarchyChangeListener(…)
- Hierarchy serialization
  - ITypeHierarchy#store(OutputStream, IProgressMonitor)
  - IType#loadTypeHierarchy(InputStream, IProgressMonitor)

Type Hierarchy – two examples

Super and subtype hierarchy on a focus type

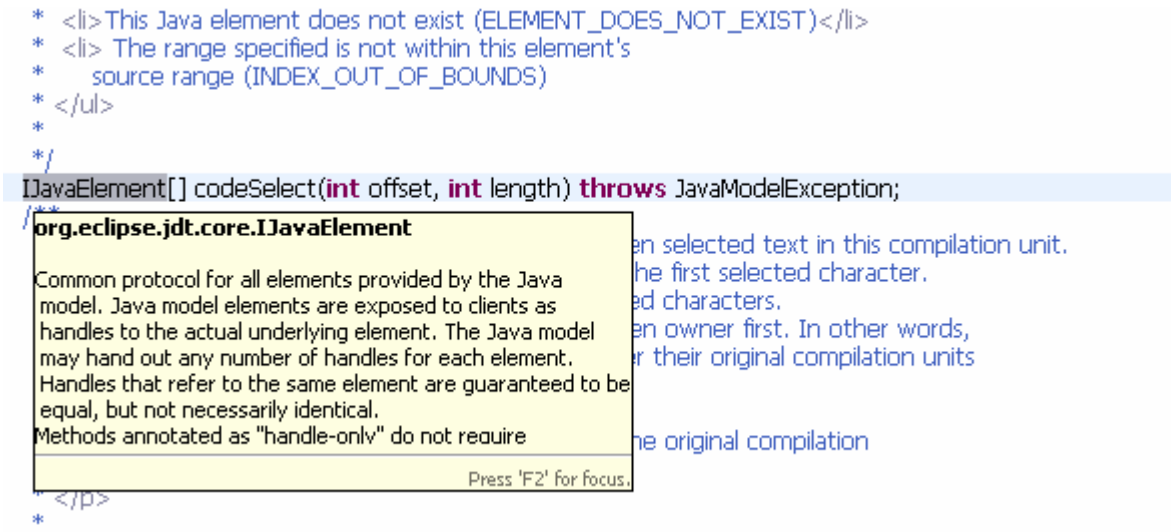
```
IType type = . . .
ITypeHierarchy hierarchy =
  type.newTypeHierarchy(null/*no progress monitor*/);
```

Hierarchy on two packages

```
IJavaProject project = . . .
IPackageFragment pkg1 = . . .
IPackageFragment pkg2 = . . .
IRegion region = JavaCore.newRegion();
region.add(pkg1);
region.add(pkg2);
ITypeHierarchy hierarchy =
  project.newTypeHierarchy(region, null/*no progress monitor*/)
```

Code resolve

- Finds declaring IJavaElement inside some source
- Navigate > Open (F3)
- Search
- Tooltip



Code resolve

- API
  - Source range + context (compilation unit or class file)
  - ICodeAssist#codeResolve(int, int)
  - ICodeAssist#codeResolve(int, int, WorkingCopyOwner)
- Resolution on the fly
  - Bind references to their declarations
  - Uses compiler internals
  - Need correct imports
  - Performs ok in non syntactically correct code
  - Resolve only what's needed (ignore sibling methods)

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’

Search Engine – Design Motivation

Need quick access to all references or declaration of a Java element

- Searching for all references of type “A”
- Call graphs
- All types in workspace

Tradeoff between search and update performance

Chosen solution:

- Index based search engine
- Index is “word” based. It doesn’t contain resolve information (e.g. class U references method foo(), not method A#foo()).
- Special resolve step needed to narrow down matches reported from index (e.g. searching for B#foo() must not report U).

Search Engine

- Pattern-based
  - construct: package, type, field, method, constructor
  - occurrence: declaration, reference, read/write-access, extends/implements
  - wildcards: find all references to any method foo (any arguments)
  - pattern can also be based on Java element
- Scoped search
  - scope = set of Java elements
  - predefined workspace and hierarchy scopes
- Match contains details
  - range information, accuracy, enclosed element
- Indexing
  - pure source, updated automatically in background
  - built-in support to perform concurrent queries
  - looking up in indexes to find possible document matches
- Locating
  - document matches are processed to issue actual search matches

Search Engine – Using the APIs

- Creating a search pattern
  - SearchPattern#createPattern(…)
- Creating a search scope
  - SearchEngine#createJavaSearchScope(IJavaElement[] …)
  - SearchEngine#createHierarchyScope(IType)
  - SearchEngine#createWorkspaceScope()
- Participating in Java search (optional)
  - Subclass SearchParticipant
  - Participate in indexing:
    - indexDocument(SearchDocument, IPath)
    - selectIndexes(…)
    - scheduleDocumentIndexing(…)
  - Participate in match locating:
    - locateMatches(…)
- Collecting results
  - Subclass SearchRequestor
  - Results are reported as SearchMatch

Search Engine – an example

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

Search pattern

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

Search scope

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

Result collector

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

Start search

```
SearchEngine searchEngine = new SearchEngine();
searchEngine.search(
  pattern,
  new SearchParticipant[0],
  scope,
  requestor,
  null /*progress monitor*/);
```

Get your hands wet - 2

- Write a Junit plug-in test case
- Resolve the element under the cursor
- Search for all references
- Test case checks correctness

JUnit Plug-in test case:

- A test case that is run inside a new instance of Eclipse
- Runs on a new, empty workspace

JUnit Plug-in test case has to be in a plugin

- Plugin should require plugin ‘org.junit’
- Plugin should require all used infrastructure ‘org.eclipse.jdt.core’

Use the JUnit Plug-in Test launcher

Abstract Syntax Tree - Design Motivation

Java Model and type hierarchy are on demand, fault-tolerant and optimized to present model elements in a viewer.

Refactorings and code manipulation features need fully resolved information down to statement level to perform exact code analysis.

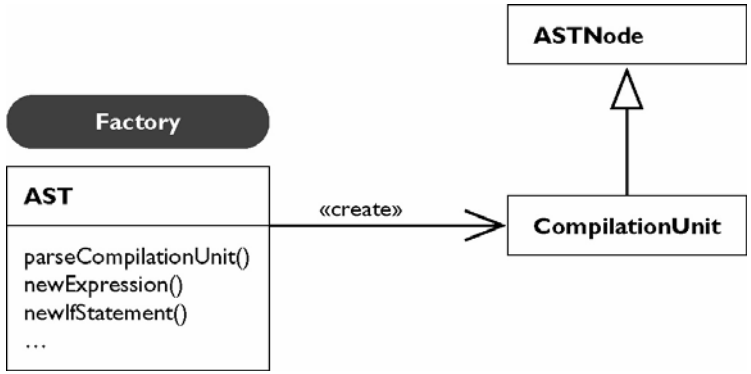
Need way to manipulate source code on a higher abstraction than characters

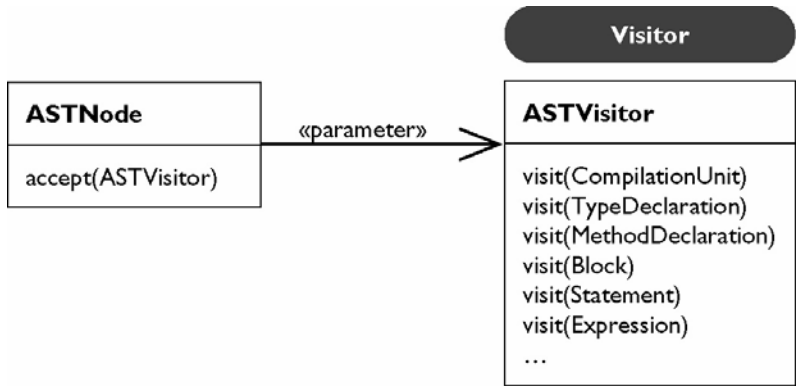
Chosen solution:

- Instantiated abstract syntax tree with all resolved bindings
  - Defined life cycle
  - Well known creation costs
- Abstract syntax tree rewriter to manipulate code on language element level

Abstract Syntax Tree

- Zooming in a compilation unit
  - a node for every construct in the language
  - source range
  - editable, with some simple validation
- Pure source model
  - instant life-cycle, fully populated
  - can resolve all references to bindings
  - bindings are fully connected
- Analysis using visitor or node properties





Creating an AST

- Build AST from element or source
  - Java model element: ICompilationUnit, IClassFile
  - Source string plus file name and IJavaProject as context
  - User created nodes with class ‘AST’ (factory)
- Bindings or no bindings
  - Bindings contain resolve information. Only available on syntax errors free code.
- 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

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

Create AST on an element

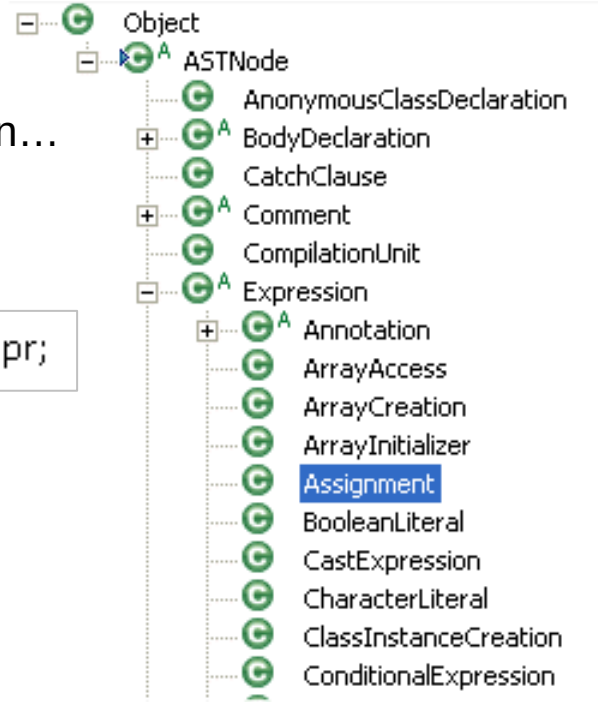
```
ASTParser parser= ASTParser.newParser(AST.JLS3);
parser.setSource("System.out.println();".toCharArray());
parser.setProject(javaProject);
parser.setKind(ASTParser.K_STATEMENTS);
ASTNode node= parser.createAST(null);
```

Create AST on source string

AST Tree structure

A Java type for each syntactic construct

Assignment, CastExpression, ConditionalExpression…



Typed access to the node children:

ConditionalExpression:

getExpression()

getThenExpression()

getElseExpression():

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

Homogenous access using node properties:

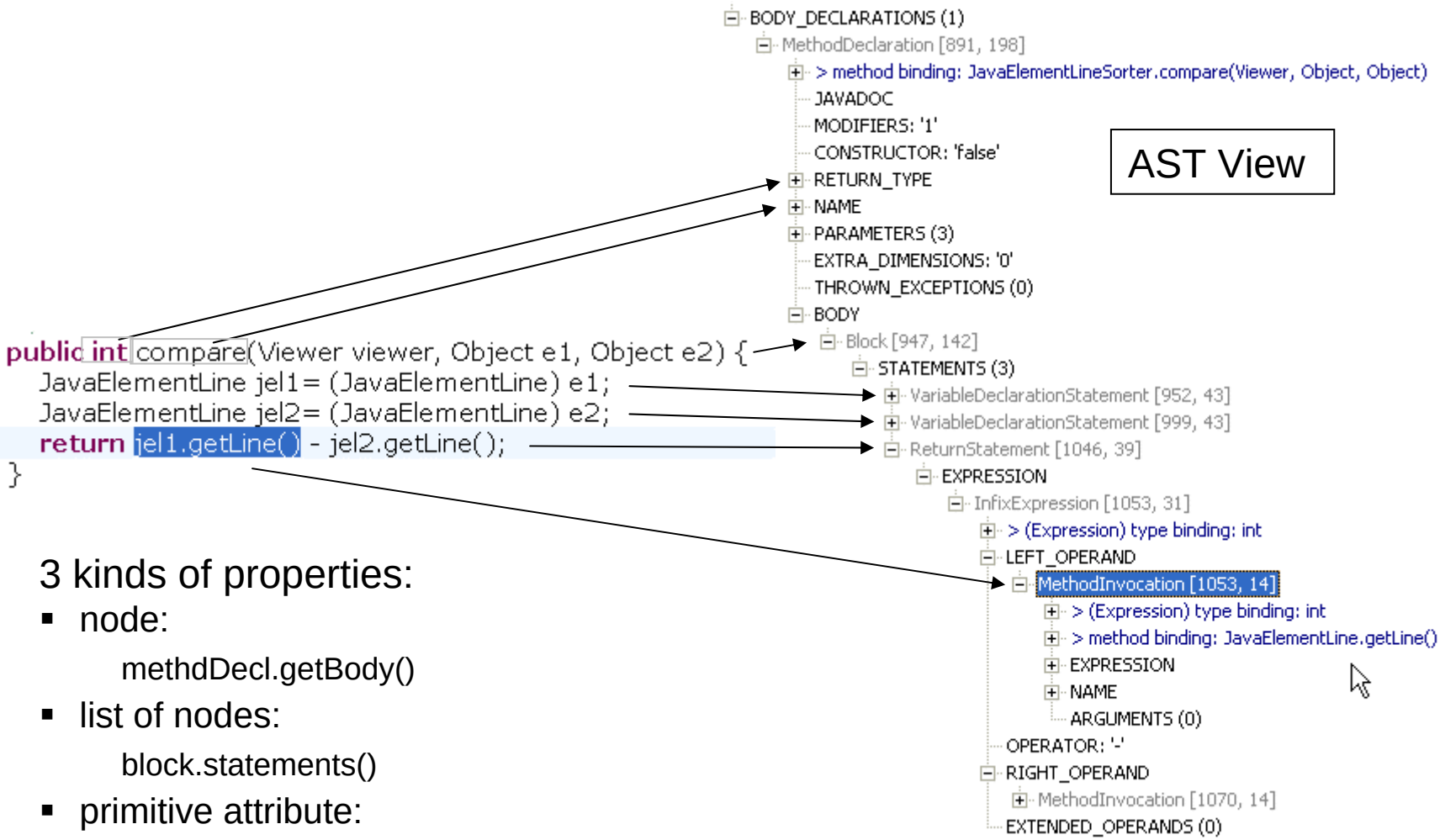
```
List allProperties= node.structuralPropertiesForType();
```

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 nodes and node properties



3 kinds of properties:

- node:
  methdDecl.getBody()
- list of nodes:
  block.statements()
- primitive attribute:
  methodDecl.isConstructor()

Bindings

Bindings contain resolved information

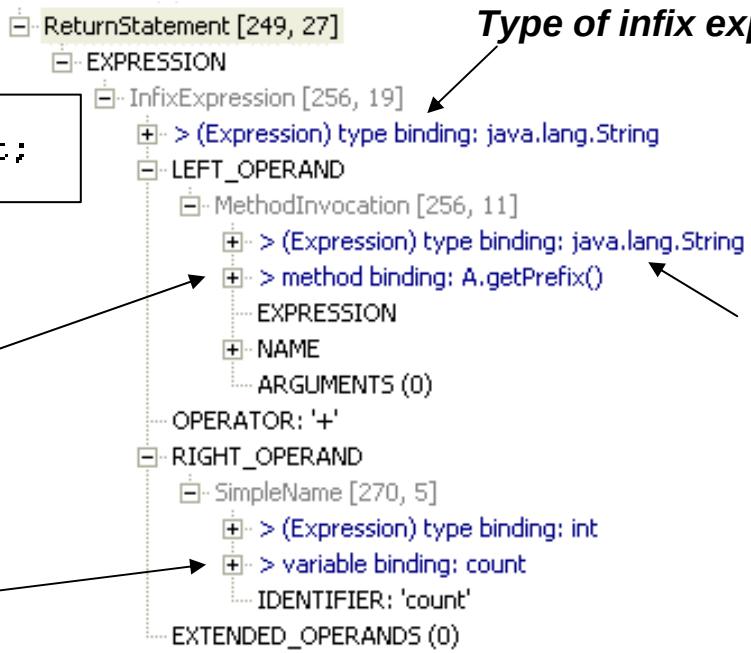
- Referenced symbol (type, method, variable)

  IBinding binding= name.resolveBinding()

- Type of an expression

  ITypeBinding type= expression.resolveTypeBinding()

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



Type of infix expression is 'String'

'getPrefix()' resolves to a method in 'A'

Type of method invocation is 'String'

'count' resolves to a variable

Bindings

Bindings are fully connected

- ITypeBinding has binding of super type, interfaces, all members
- IMethodBinding has binding of parameter types, exceptions, return type
- IVariableBinding has binding of variable type

Bindings are very expensive:

- Do not hold on bindings
- Do not hold on ASTNodes that contain bindings

Within an AST:

- Binding identity (can use '==' to compare bindings)

Bindings from different ASTs:

- Compare binding.getKey()
- In 3.1: isEqualTo(…)

Bindings

From a binding to its declaration ASTNode:

- astRoot.findDeclaringNode(binding) (on CompilationUnit)

From a binding to a IJavaElement (new in 3.1):

- binding.getJavaElement()

Using the Abstract Syntax Tree

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

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

Count the number of casts

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

Count the number of references to a static 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

AST Rewriting

Current 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
- In 3.1: 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();
  ASTRewrite astRewrite= ASTRewrite.create(ast);

  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);

  edit.apply(document);
}
```

Create the rewriter

Change the method name

Insert a new parameter as first parameter

Create resulting edit script

API in JDT.UI

Labels, images, structure, order for IJavaElements:

- JavaElementLabelProvider
- StandardJavaElementContentProvider
- JavaElementSorter

Selection and configuration dialog, wizards

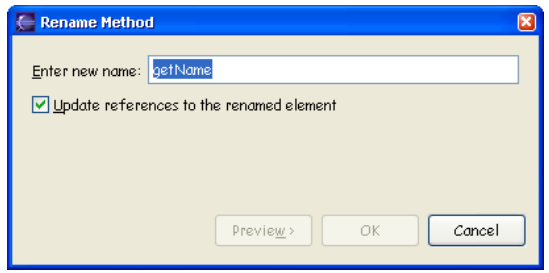
- JavaUI.createPackageDialog(..), JavaUI.createTypeDialog(..)
- BuildPathDialogAccess
- NewClassWizardPage, NewInterfaceWizardPage…

Java Actions to add to context menus

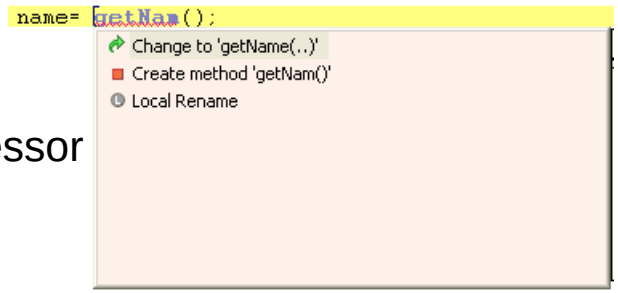
- package org.eclipse.jdt.ui.actions

Code Manipulation Toolkits

- Refactoring – org.eclipse.refactoring
  - refactorings - org.eclipse.refactoring.core.Refactoring
    - responsible for precondition checking
    - create code changes
  - code changes - org.eclipse.refactoring.core.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



Get your hands wet – 3

Use the AST by implementing a quick assist:
e.g.: replace a simple type reference by fully qualified one

```
<extension point="org.eclipse.jdt.ui.quickAssistProcessors">
  <quickAssistProcessor
    name="My Assist"
    class="org.eclipse.jdt.tutorial.MyQuickAssistProcessor"
    id="org.eclipse.jdt.tutorial.MyQuickAssistProcessor">
  </quickAssistProcessor>
</extension>
```

```
public class MyQuickAssistProcessor implements IQuickAssistProcessor {

 public boolean hasAssists(IInvocationContext context) throws CoreException {
   ASTNode covering= context.getCoveringNode();
   return covering instanceof SimpleName && ...;
 }

 public IJavaCompletionProposal[] getAssists(IInvocationContext context,
             IProblemLocation[] locations) throws CoreException {
    ...
 }
```

J2SE 5.0

Eclipse 3.1 supports all new J5SE 5.0 features

New language constructs	Major changes in Java Model
Type Parameters	IType and IMethod can have type parameters Code resolve can return ITypeParameter Signatures support wildcards, type parameters Search support for generic types
Enum	IType can be an Enum (flag), IField can be an enum constant declaration (flag)
Annotations	IType can be an Annotation (flag)
Var args	IMethod can have the ‘vararg’ flag set. If set the last argument is not T[] but T…
Static imports	Static flag in IImportDeclaration
Auto boxing, Enhanced for	-

J2SE 5.0

Updates in the AST

- New properties, new nodes
- To avoid breaking existing clients: Introduced a new version of AST tree: Select level on AST creation: ASTParser.newParser(astLevel)

  JLS2: Parses 1.1 - 1.4 Java source

  JLS3: Parses 1.1 - 1.5 Java source

- AST.JLS3 has a different structure and more nodes than AST.JLS2:

  JLS2 API: MethodDeclaration.getModifiers() : int

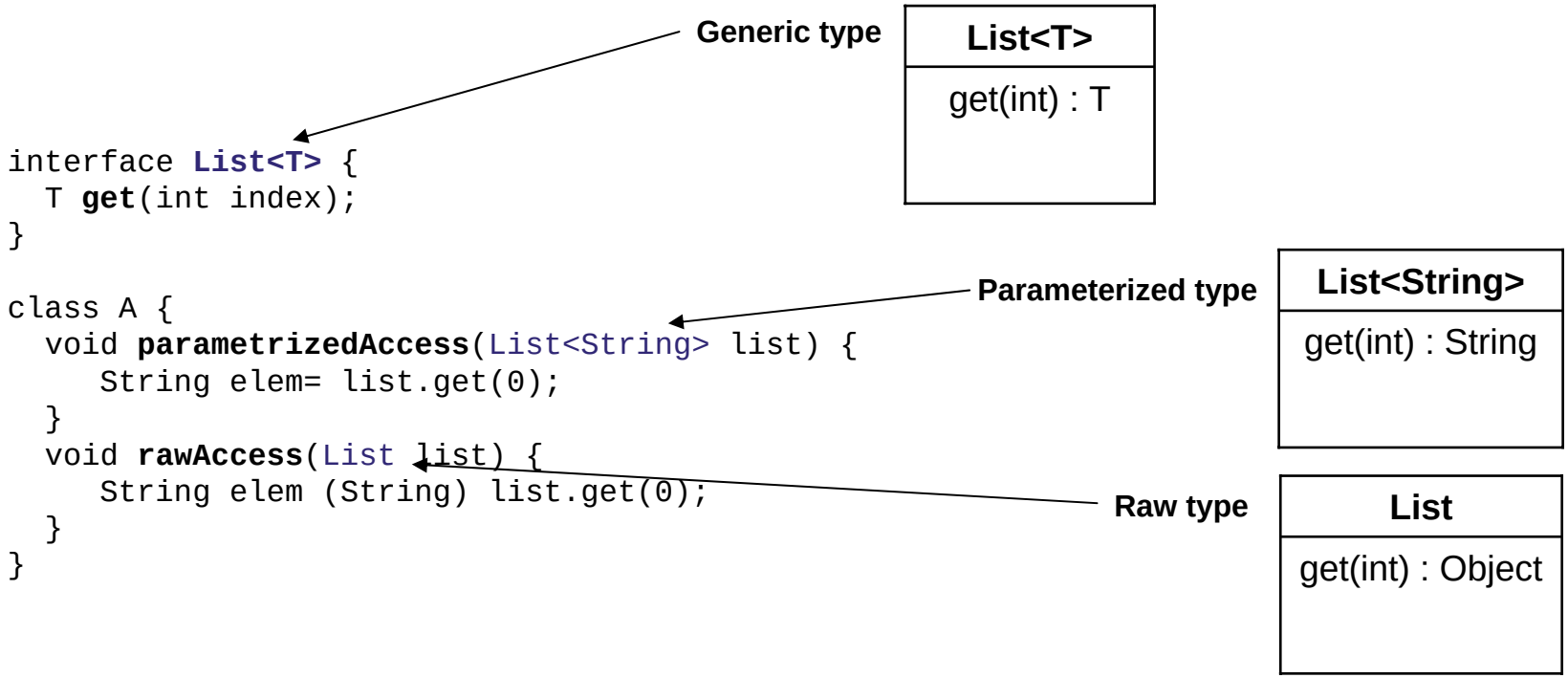
  JLS3 API: MethodDeclaration.modifiers() : List

- If you created an AST with JLS3, calling old API with throw an IllegalArgumentException
- Controlled upgrade to the new AST API's. JLS2 API will be marked as deprecated in 3.1

J2SE 5.0

Major change with bindings

- New kinds on type bindings: Type variables, Wildcards
- Difference between type declaration and type instantiation



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, …
- New in 3.1
  - J2SE 5.0 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`

The JDT Team

Philippe Mulet

Jérôme Lanneluc

Kai-Uwe Mätzel

André Weinand

Tom Eicher

Kent Johnson

Jim des Rivières

Frédéric Fusier

Martin Aeschlimann

Olivier Thomann

David Audel

Tobias Widmer

Christof Marti

Daniel Megert

Markus Keller

Erich Gamma

Dirk Bäumer