Exploring Eclipse's ASTParser

How to use the parser to generate code

Level: Intermediate

Manoel Marques (manoel@themsslink.com), Senior Consultant, The Missing Link Inc.

12 Apr 2005

If you ever tried to write an application that manipulates code, you know things can get complicated -- especially if the language is as sophisticated as Java™. Well, for those already using Eclipse, there is good news: You can take advantage of the Java Development Tooling (JDT) and ASTParser. This article shows you how.

How Eclipse JDT can help

The Eclipse JDT provides APIs to manipulate Java source code, detect errors, perform compilations, and launch programs. In this article, I will show how you can create Java classes from scratch using the **ASTParser**. You will also learn how JDT services can be used to compile your generated code.

Eclipse's JDT has its own **Document Object Model (DOM)** in the same spirit of the well-known XML DOM: the **Abstract Syntax Tree (AST)**.

Eclipse V3.0.2 supports the Java Language Specification, Second Edition (JLS2). It will correctly parse programs written in all versions of the Java language up to J2SE 1.4. There is work under way for JLS3 support, so you should be able to parse programs written using the new J2SE 1.5 constructs in the next major Eclipse release.

Free code for all

There are two sample applications provided with this article. Both are contained in an Eclipse project called **ASTExplorer**:

- ASTMain
- ASTExplorer

ASTMain generates a Java class, compiles it, then runs its main() method. This method creates a SWT Shell widget with a button.

ASTExplorer parses and shows the AST hierarchy for a given Java class. It contains three panes: one with an SWT Tree view showing the AST hierarchy, another with the original source code, and a third for parser errors.

Figure 1 shows **ASTExplorer** in action. Notice that when you select a node, the correspondent place in the source code is highlighted in blue. Parsing errors are highlighted in red.

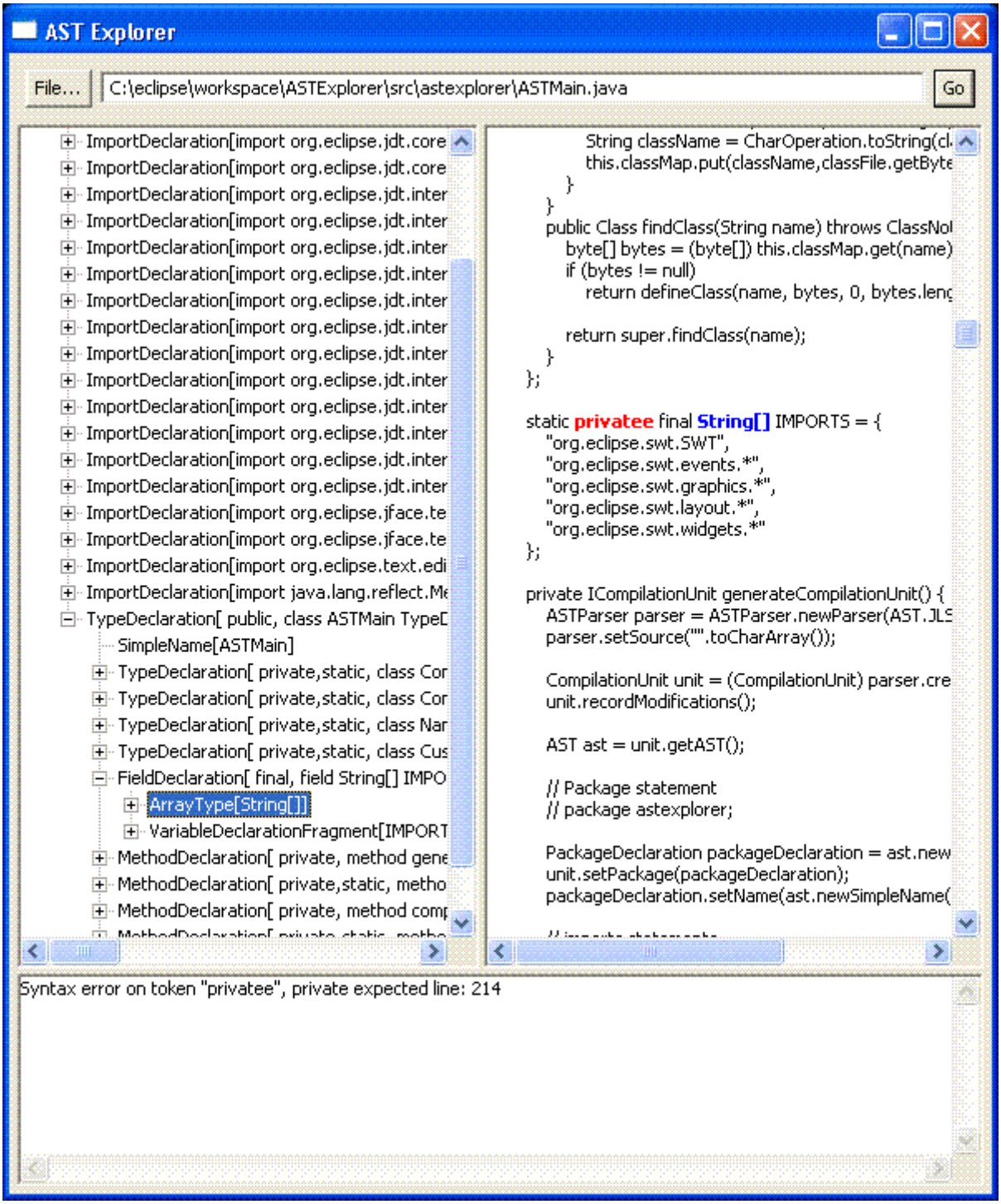

Figure 1. ASTExplorer in action

The samples were tested in Eclipse V3.0.1 and V3.0.2, Windows® XP Professional SP2, Sun J2SDK 1.4.2.05. The current project's classpath has entries for Eclipse V3.0.2. If you need it to run under Eclipse V3.0.1, just change the classpath to point to the correct plug-ins.

I recommend you download the sample applications before reading the rest of the article. The key word here is *exploration*. Your learning will improve if you run the samples as you read the article.

ASTParser and ASTNodes

At the top of the AST hierarchy is the **ASTNode**. Every Java construct is represented by it. Most of the node names are self-described, like **Comment**, **CastExpression**, etc. You create them by using methods of the AST class like **newBlock()**, **newBreakStatement()**, etc. A Java class is represented by the Compilation Unit node. Listing 1 shows how it is created.

```
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource("".toCharArray());
CompilationUnit unit = (CompilationUnit) parser.createAST(null);
unit.recordModifications();
AST ast = unit.getAST();
```

Listing 1. Creating a Compilation Unit

Notice how the **ASTParser** is configured for JLS2. The parser is then initialized with an empty array. If you don't do this, you'll get exceptions when trying to access the **Compilation Unit**.

These are the same steps you take to parse existing code. In this case, you may want to pass an instance of **org.eclipse.core.runtime.IProgressMonitor** to the **createAST()** method to provide feedback during a long parsing. I will demonstrate its use later.

The call to **recordModifications()** starts the monitoring of node changes. It is vital to call this method because it allows you to access the source later by retrieving the node modifications.

Finally, the AST owner is accessed from the Compilation Unit and will be used for all subsequent node creations. All the nodes in an AST Tree belong to the same owner. Any node not created by it needs to be imported first in order to be added to the tree. At that point, you are ready to start building your Java class. Listing 2 shows a package creation.

```
PackageDeclaration packageDeclaration = ast.newPackageDeclaration();
unit.setPackage(packageDeclaration);
packageDeclaration.setName(ast.newSimpleName("astexplorer"));
```

Listing 2. Creating a Package

Several of the nodes methods use a Name node. The Name node can be a **SimpleName** or a **QualifiedName**, which is a group of **SimpleNames**. The external representation of a **QualifiedName** is, for instance, **org.eclipse.swt.widgets**. So basically, whenever you have dots, you use a **QualifiedName**. The method **ast.newName()** lets you create Name nodes by accepting an array of strings. In the code sample, I provide a convenient method that parses a string with dots and creates this string array.

There are six major node groups: **BodyDeclaration**, **Comment**, **Expression**, **Statement**, **Type**, and **VariableDeclaration**. **BodyDeclarations** are any declaration that goes inside a class. For instance, the declaration **private Point minimumSize;** would be created as follows:

```
VariableDeclarationFragment vdf = ast.newVariableDeclarationFragment();
vdf.setName(ast.newSimpleName("minimumSize"));
FieldDeclaration fd = ast.newFieldDeclaration(vdf);
fd.setModifiers(Modifier.PRIVATE);
fd.setType(ast.newSimpleType(ast.newSimpleName("Point")));
```

Listing 3. Creating a VariableDeclaration

Notice how the **FieldDeclaration** is created from the

VariableDeclarationFragment. AST programming is about combining the different nodes. You don't use methods like **appendChild()** or **insertBefore()** found in the XML DOM. Instead, different node types have different ways of being created and initialized.

You just saw an example of a type of **VariableDeclaration**: the **VariableDeclarationFragment**. The other type, **SingleVariableDeclaration**, is used mostly for parameter lists. For instance, Listing 4 shows how to create the parameter size in **ControlAdapterImpl(Point size)**.

```
    SingleVariableDeclaration variableDeclaration =
ast.newSingleVariableDeclaration();
    variableDeclaration.setModifiers(Modifier.NONE);
    variableDeclaration.setType(ast.newSimpleType(ast.newSimpleName("Point"))
);
    variableDeclaration.setName(ast.newSimpleName("size"));
    methodConstructor.parameters().add(variableDeclaration);
```

Listing 4. Creating a method Parameter

There are three types of Comment nodes: **BlockComment**, **Javadoc**, and **LineComment**.

The AST Tree supports the creation and insertion of Javadoc nodes only. It considers the exact positioning of **BlockComment** and **LineComment** nodes problematic, so you will only see those when parsing an existing source. Listing 5 shows an example of a **Javadoc** node creation.

```
    Javadoc jc = ast.newJavadoc();
    TagElement tag = ast.newTagElement();
    TextElement te = ast.newTextElement();
    tag.fragments().add(te);
    te.setText("Sample SWT Composite class created using the ASTParser");
    jc.tags().add(tag);
    tag = ast.newTagElement();
    tag.setTagName(TagElement.TAG_AUTHOR);
    tag.fragments().add(ast.newSimpleName("Manoel Marques"));
    jc.tags().add(tag);
    classType.setJavadoc(jc);
```

Listing 5. Creating a Javadoc node

Expression and Statement nodes are the most used node types. There are several examples of its creation in the sample code. You can create a simple statement like **GridLayout gridLayout = new GridLayout()** with the following:

```
    VariableDeclarationFragment vdf = ast.newVariableDeclarationFragment();
    vdf.setName(ast.newSimpleName("gridLayout"));
    VariableDeclarationStatement                    vds                    =
ast.newVariableDeclarationStatement(vdf);
    vds.setType(ast.newSimpleType(ast.newSimpleName("GridLayout")));
    ClassInstanceCreation cc = ast.newClassInstanceCreation();
    cc.setName(ast.newSimpleName("GridLayout"));
    vdf.setInitializer(cc);
    constructorBlock.statements().add(vds);
```

Listing 6. Creating a Statement

Notice how the nodes are composed. The whole statement is a **VariableDeclarationStatement** of type **GridLayout**. It contains a **VariableDeclarationFragment**, which contains a **ClassInstanceCreation**.

The same statement can be created using an Assignment expression, as seen in Listing 7.

```
    Assignment a = ast.newAssignment();
    a.setOperator(Assignment.Operator.ASSIGN);
```

```
   VariableDeclarationFragment vdf = ast.newVariableDeclarationFragment();
   vdf.setName(ast.newSimpleName("gridLayout"));
   VariableDeclarationExpression                    vde                    =
ast.newVariableDeclarationExpression(vdf);
   vde.setType(ast.newSimpleType(ast.newSimpleName("GridLayout")));
   a.setLeftHandSide(vde);

   ClassInstanceCreation cc = ast.newClassInstanceCreation();
   cc.setName(ast.newSimpleName("GridLayout"));
   a.setRightHandSide(cc);
   constructorBlock.statements().add(ast.newExpressionStatement(a));
```

Listing 7. Alternate way of creating the same Statement

You can think of it as an **Assignment** expression with a **VariableDeclarationExpression** that contains a **VariableDeclarationFragment** on the left side and a **ClassInstanceCreation** on the right side. Notice that the Assignment expression is wrapped by a Statement with the method **newExpressionStatement()** before being added to the statements list.

Both ways will produce the same source code, but you should use the first method, though. If you parse existing code, you will see that the nodes created follow the first approach. That is why is so important to use the **ASTExplorer** sample. This way, you can visualize the nodes created by the parser for a particular code snippet and create yours the same way.

If you take a look at the **ASTMain** class sample, you'll have a pretty good idea of how to create the different nodes in several situations. I tried to include all the tricky constructs like inner classes, try blocks, array parameters, and much more. I touched on the areas where I had problems and where I believe you might need some help.

Getting the actual source code

Once you have a **Compilation Unit**, it is easy to get the actual source code.

You already did half the work by calling **recordModifications()**. Just call the method **rewrite()** in the **Compilation Unit**. It takes an instance of **org.eclipse.jface.text.IDocument** and a Map of formatting options. The **IDocument** instance should contain the original source -- in our case, none -- and the method will merge the changes in the **Compilation Unit** with the document text and return an instance of **org.eclipse.jface.text.edits.TextEdit** with all the changes.

The formatting options let you specify things like the positions of brackets and indentation. You can find a list of the possible options in the class **org.eclipse.jdt.core.formatter.DefaultCodeFormatterConstants**.

Once you get the **TextEdit** instance, you can use it to see what changed. But in our case, we just **apply** the changes to the original document.

All the code is now in the document ready to be extracted. You can see this process in Listing 8.

```
   public char[] getContents() {
     char[] contents = null;
     try {
        Document doc = new Document();
        TextEdit edits = unit.rewrite(doc, null);
        edits.apply(doc);
        String sourceCode = doc.get();
        if (sourceCode != null)
           contents = sourceCode.toCharArray();
```

```
      }
      catch (BadLocationException e) {
         throw new RuntimeException(e);
      }
      return contents;
   }
```

Listing 8. Accessing the Compilation Unit contents

Compiling the generated code

The Eclipse IDE works with projects in a workspace. You build the entire project and rely on JDT to check dependencies and compile all the classes. This is well explained in the Eclipse help, and it boils down to:

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

Listing 9. Building an Eclipse project

I won't spend more time on this, as it is well explained in the Eclipse help. The JDT provides another way that is more useful for our purposes through the use of the class **org.eclipse.jdt.internal.compiler.Compiler**. It is quite simple, actually: You instantiate a compiler object and call the method **compile()** on it.

```
   Compiler compiler = new Compiler(new NameEnvironmentImpl(unit),
                    DefaultErrorHandlingPolicies.proceedWithAllProblems(),
                    settings, requestor, new Default\
                    ProblemFactory(Locale.getDefault()));
   compiler.compile(new ICompilationUnit[] { unit });
```

Listing 10. Compiling a Compilation Unit

I will go over the constructor parameters first. You need the following:

- org.eclipse.jdt.internal.compiler.env.INameEnvironment

Connects the compiler with the outside environment. In simple terms, it represents the classpath. The compiler will use it to ask information about the types it may encounter.

- org.eclipse.jdt.internal.compiler.IErrorHandlingPolicy

Tells the compiler what to do when errors are encountered. I prefer to let the compiler proceed as much as possible before stopping, so I use the predefined instance **DefaultErrorHandlingPolicies.proceedWithAllProblems()**.

- Map Settings

Compiler settings that can be found in **org.eclipse.jdt.internal.compiler.impl.CompilerOptions**. It lets you specify if you need line numbers generated, deprecated method warnings, etc.

- org.eclipse.jdt.internal.compiler.ICompilerRequestor

Receives the compilation results and any errors encountered during the compilation process.

- org.eclipse.jdt.internal.compiler.IProblemFactory

Factory responsible for creating an instance of **org.eclipse.jdt.core.compiler.IProblem**. Useful to implement if you support some special type of problem handling or different languages for error messages. In the sample, I use the standard implementation

DefaultProblemFactory(Locale.getDefault()).

Finally, to compile, you need an array of **org.eclipse.jdt.internal.compiler.env.ICompilationUnit**. Please do not confuse this interface with **org.eclipse.jdt.core.ICompilationUnit**. Unfortunately, they have the same name, but the latter is useful only if your class is part of an Eclipse Java project.

It is easy to implement **org.eclipse.jdt.internal.compiler.env.ICompilationUnit.** It corresponds to the **CompilationUnit** node already created. Listing 11 shows a simple implementation.

```
static private class CompilationUnitImpl implements ICompilationUnit {
  private CompilationUnit unit;

  CompilationUnitImpl(CompilationUnit unit) {
    this.unit = unit;
  }
  public char[] getContents() {
    char[] contents = null;
    try {
      Document doc = new Document();
      TextEdit edits = unit.rewrite(doc,null);
      edits.apply(doc);
      String sourceCode = doc.get();
      if (sourceCode != null)
        contents = sourceCode.toCharArray();
    }
    catch (BadLocationException e) {
      throw new RuntimeException(e);
    }
    return contents;
  }
  public char[] getMainTypeName() {
    TypeDeclaration classType = (TypeDeclaration) unit.types().get(0);
    return classType.getName().getFullyQualifiedName().toCharArray();
  }
  public char[][] getPackageName() {
    String[] names =
getSimpleNames(this.unit.getPackage().getName().getFullyQualifiedName());
    char[][] packages = new char[names.length][];
    for (int i=0;i < names.length; ++i)
      packages[i] = names[i].toCharArray();
    return packages;
  }
  public char[] getFileName() {
    TypeDeclaration classType = (TypeDeclaration) unit.types().get(0);
    String name = classType.getName().getFullyQualifiedName() + ".java";
    return name.toCharArray();
  }
}
```

Listing 11. ICompilationUnit implementation

Checking for compilation errors

The first thing you should do after compiling is get any possible error from your **ICompilerRequestor** implementation. They could be just warnings or fatal errors. Here is a simple check:

```
List problems = requestor.getProblems();
boolean error = false;
for (Iterator it = problems.iterator(); it.hasNext();) {
  IProblem problem = (IProblem)it.next();
  StringBuffer buffer = new StringBuffer();
  buffer.append(problem.getMessage());
```

```
      buffer.append(" line: ");
      buffer.append(problem.getSourceLineNumber());
      String msg = buffer.toString();
      if(problem.isError()) {
         error = true;
         msg = "Error:\n" + msg;
      }
      else
         if(problem.isWarning())
            msg = "Warning:\n" + msg;

      System.out.println(msg);
   }
```

Listing 12. Handling compilation errors

Running the compiled application

If all goes well, it is time to instantiate the class and run its main method. This can be done easily using reflection on the bytecodes returned by the ICompilerRequestor implementation.

```
   try {
      ClassLoader loader = new CustomClassLoader(getClass().getClassLoader(),
      requestor.getResults());
      String className = CharOperation.toString(unit.getPackageName()) + "." +
      new String(unit.getMainTypeName());
      Class clazz = loader.loadClass(className);
      Method m = clazz.getMethod("main",new Class[] {String[].class});
      m.invoke(clazz,new Object[] { new String[0] });
   }
   catch (Exception e) {
      e.printStackTrace();
   }
```

Listing 13. Running the compiled application

Notice how I access the class from a custom class loader, which loads the compiled bytecodes upon request. A example of it is seen in Listing 14.

```
   static private class CustomClassLoader extends ClassLoader {
      private Map classMap;

      CustomClassLoader(ClassLoader parent,List classesList) {
         this.classMap = new HashMap();
         for (int i = 0; i < classesList.size(); i++) {
            ClassFile classFile = (ClassFile)classesList.get(i);
            String                          className                          =
CharOperation.toString(classFile.getCompoundName());
            this.classMap.put(className,classFile.getBytes());
         }
      }
      public Class findClass(String name) throws ClassNotFoundException {
         byte[] bytes = (byte[]) this.classMap.get(name);
         if (bytes != null)
            return defineClass(name, bytes, 0, bytes.length);

         return super.findClass(name);
      }
   }
```

Listing 14. Custom class loader

This is a very basic compiler. The **INameEnvironment** implementation is simplistic and expects that all the class dependencies are already loaded in the current class loader. A real implementation may need another custom class

loader that would search through some classpath provided just for the compilation.

Also, you may want to cache some of the information, especially what is returned from the **ICompilationUnit** implementation. The process of getting the source code, for instance, is time-consuming, so you should cache it.

Parsing existing code

Let's review the steps necessary for parsing, as shown in Listing 15.

```
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setKind(ASTParser.K_COMPILATION_UNIT);
parser.setSource(sourceString.toCharArray());
CompilationUnit node = (CompilationUnit) parser.createAST(null);
```

Listing 15. Parsing a Java class

I know that the outcome of the parsing should be a Compilation Unit, so I initialized the parser with **ASTParser.K_COMPILATION_UNIT**. You can use also **K_CLASS_BODY_DECKARATION**, **K_EXPRESSION** or **K_STATEMENTS**. You could have done this, instead:

```
parser.setKind(ASTParser.K_STATEMENTS);
parser.setSource(sourceString.toCharArray());
Block block =  (Block) parser.createAST(null);
```

Listing 16. Parsing Java statements

This is useful if you have a bunch of statements that will be inserted in an existing block later. Just don't forget to import your parsed node block before inserting it, as in: **block = (Block)ASTNode.copySubtree(unit.getAST(), block)**;.

The parameter to **createAST()** is **org.eclipse.core.runtime.IProgressMonitor**. It is not necessary when creating nodes, but it becomes important when parsing. It lets an external observer follow the task's progress and cancel it if necessary. The parsing can be done in a different thread while the UI thread receives notifications from the **IProgressMonitor**.

Any read-only tree operation is thread-safe as long as there isn't any thread modifying it. If other threads can modify a node, the recommended way is to synchronize the AST object that owns the tree **(synchronize (node.getAST()) {...}).**

The JFaces library provides a convenient dialog that encapsulates the IProgressMonitor in the **org.eclipse.jface.dialogs.ProgressMonitorDialog**. You can use it as shown in Listing 17.

```
ProgressMonitorDialog dialog = new ProgressMonitorDialog(getShell());
dialog.run(true, true, new IRunnableWithProgress() {
  public void run(final IProgressMonitor monitor)
        throws InvocationTargetException {
    try {
      ASTParser parser = ASTParser.newParser(AST.JLS2);
      if (monitor.isCanceled()) return;
      parser.setKind(ASTParser.K_COMPILATION_UNIT);
      if (monitor.isCanceled()) return;
      final String text = buffer.toString();
      parser.setSource(text.toCharArray());
      if (monitor.isCanceled()) return;
      final CompilationUnit node =
          (CompilationUnit) parser.createAST(monitor);
      if (monitor.isCanceled()) return;
      getDisplay().syncExec(new Runnable() {
         public void run() {
```

```
            // update the UI with the result of parsing
            ...
          }
        });
      }
      catch (IOException e) {
        throw new InvocationTargetException(e);
      }
    }
  });
```

Listing 17. Parsing with IProgressMonitor

The first two **boolean** parameters to the dialog run method indicate that the **IRunnableWithProgress** instance run method should be in a separate thread and it can be canceled. This way, the parsing runs in a different thread while the dialog is shown with a cancel button. If the user presses the button, the dialog will set the **IProgressMonitor** instance method **setCanceled()** to true. Since this instance was passed to the parser as well, the parser operation will stop.

After the parsing ends, we need to update the UI with the parsing results using its own UI thread, not the thread that did the parsing. In order to do this, use the method **org.eclipse.swt.widgets.Display.syncExec** that will run the code in the **runnable** instance in the UI thread.

Walking the tree

The ASTNode allows the walking of the node tree by the use of the visitor pattern (see Resources).

You create a class derived from o**rg.eclipse.jdt.core.dom.ASTVisitor** and pass an instance of it to node's method **accept()**. When this method is called, each node in the tree will be "visited," starting with the current node downward. The following methods are invoked per node:

1. preVisit(ASTNode node)
2. boolean visit(node)
3. endVisit(node)
4. postVisit(ASTNode node)

There is one method -- **visit()** and **endVisit()** -- for each node type. The type of the parameter node corresponds to the node being visited. If you return false from the **visit()** method, the children of this node won't be visited.

Comment nodes do not show in the AST Tree because they are not parented. The method **getParent()** returns null. It is possible to access them by calling the method **getCommentList()** in the Compilation Unit. If you need to show them, you should call this method and visit each comment node individually.

In the sample, the class **ASTExplorerVisitor** contains a block of commented code in the **preVisit()** method that, if uncommented, will show all comment nodes as children of the Compilation Unit.

Parser errors are returned in the **Compilation Unit** class instance. The method **getProblems()** returns an array of **IProblem** instances. This is the same **IProblem** class used for compilation errors.

It is important to notice that the **ASTParser** is not a compiler. It will flag errors only if there is something in the source file that will affect the AST Tree integrity.

For instance, if you type *classs* instead of *class*, it will affect the creation of

a **TypeDeclaration** node, and it will be an error. On the other hand, if you type **private Stringgg str;**, it will be valid because a class called **Stringgg** may exist somewhere. Only the compiler would be able to flag this as an error. Just be aware that *a valid tree does not mean a valid compilation*.

Conclusion

We have covered all the "gotchas" I encountered when using the **ASTParser**. You are now ready to add the JDT services in your own projects. There is a lot of power there, and you don't need to reinvent any wheels. Happy parsing!