

# The Scala Plugin for Eclipse

Sean McDirmid and Martin Odersky

École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland  
{sean.mcdirmid, martin.odersky}@epfl.ch

## Extended Abstract

As a new programming language, Scala is designed as an improvement over Java with concise higher-order function syntax, type inference, mixin inheritance, and support for general typed-based pattern matching. However, many programmers are hesitant to try new languages such as Scala because their supporting tools are often not as polished as those of mainstream languages. As a result, it is very important to support Scala with good tool support, preferably in the form of an IDE. As an open platform, Eclipse can be extended to support new programming languages and provides many reusable components to support such extensions. Although documentation exists on how to use Eclipse's APIs, very little is documented on how to go about supporting new language such as Scala. This abstract reports on our experiences in extending Eclipse to support Scala.

Scala is a *Java-like language*: Scala code is imperative and organized into classes and Scala programs can execute on the Java virtual machine (JVM). Java-like languages are an important class of languages and include Kawa, JRuby, X10, and Groovy. Because Eclipse already supports Java very well, it should be easier to support Java-like languages than languages such as C and C++, whose programs do not execute on the JVM. Unfortunately, many of the components that can be reused to support a Java-like language are hidden inside Eclipse's Java Development Toolkit (JDT) and/or depend on specific details of the Java language. Although Java-like languages are based on the JVM, they are often not Java language extensions, and any component that depends on Java language details will be very difficult to reuse.

Given that we wanted to support for Scala a similar set of IDE features as are supported for Java, we were faced from the start with an important design decision: should we base the Scala plug-in on the JDT, where we could replace the Java language model with a Scala language model, or should we write the Scala plug-in from scratch? We found the former option to be unworkable: the Java language model is deeply entangled in the JDT and would have been very difficult to factor out. However, the latter option would require us to rewrite a lot of code to duplicate features that are already present in the JDT. In the end, we compromised: we would write new plug-ins that used undocumented internal APIs in the JDT to reuse many of its components. This approach worked out moderately well for us: using our recently overhauled Scala compiler, known as scalac, we were able to develop our first release of the Scala plug-in within about

four developer months. On the other hand, this approach has many drawbacks that we will discuss later in this abstract.

The remainder of this abstract describes our experience in building our plug-in implementation, which can be divided into three parts related to building, presentation (user interface), and launching. We then conclude the abstract with lessons learnt as well as some recommendations for future Eclipse development.

## **Building**

A builder in Eclipse is responsible for building the code of a project when a build is requested either implicitly on save or explicitly by the user. Our builder for Scala, which is based on scalac, is fully incremental and resident, meaning it can cache build information in-between build requests. Because resident compilation was already built into scalac when we started development on the plug-in, we were able to save a lot of time and development effort. The only thing we had to implement from scratch was our own dependency calculator to ensure that when the signature of a Scala file changed, dependent files would automatically be recompiled. Because scalac is completely re-entrant (no static variables), each open Scala project can have its own instance of the Scala compiler without resorting to class loader tricks. One feature currently missing from the scalac is a safe abort mechanism, meaning the cancel button in the build progress monitor does not currently work.

We completely reused the JDT's support for managing the classpath of a project. This was achieved by having Scala projects implement the Java nature and by representing the Scala library with a new classpath variable. When a Scala project is created, we modify the project's classpath with the Scala library variable in addition to the Java library variable that is installed by the Java nature. The Scala compiler is then initialized with the project's classpath, including its specification of the output and source directories. In this way, we conflict with the JDT: the Java builder assumes that it is the only builder generating class files in the output directory, and will delete all class files on a clean build request. The Scala builder necessarily exhibits similar behavior. As a result, the Java and Scala builders cannot coexist, and the Java builder must be uninstalled from a Scala project even though this builder is installed by the Java nature.

## **Presentation**

The presentation part of the Scala plug-in uses compiler output to augment how a programmer views and edits code. As with the building part, the presentation part is based on scalac. Rather than use the same instance of the compiler that is used to build the project, we create another instance of scalac per project that only performs type checking. Additionally, because position and comment information is not stored in class files, the presentation compiler always processes source code when available. The bulk of our effort in the presentation part implementation is devoted to translating the type-checked trees from the presentation

compiler into a flat structure that corresponds to the text in a source code buffer. We refer to this process as *re-sugaring* because it must undo any de-sugaring of source code performed by the compiler. The re-sugared compiler trees are then used to implement the following JDT-like features:

- Identifier highlighting based on the constructs they are bound to;
- Text hovering of identifier type information, which is very useful because of Scala's support for type inference;
- Hyperlinking of identifiers to their definitions;
- Searching for uses of an identifier;
- Override markers;
- Content assist (currently very limited); and
- A content outline view.

The translation of source code into trees via parsing is the most basic form of de-sugaring that eases compiler implementations. Compiler implementors are often tempted to perform more aggressive amounts of de-sugaring to simplify tree and program structure for later processing. For example, a Scala variable definition is de-sugared early on by scalac into a private field with public getter and setter methods. Re-sugaring is the process of mapping de-sugared trees back to the original source code. Command-line compilers already perform some amount of re-sugaring to report compilation error messages in useful to users; e.g., line numbers are associated with trees so they can be included in error messages. The re-sugaring needs of an IDE are more extreme; e.g., rather than line numbers the starting and ending position of a tree are used to create visual error markers. Our efforts at de-sugaring compiler trees have so far been mixed: we can de-sugar most but not all compiler trees, while we are still missing the ending positions of compiler trees. Because we currently cannot completely re-sugar compiler trees, some presentation features such as reliable refactoring cannot currently be supported.

Currently, our presentation compiler always runs in the foreground user-interface thread. We made this decision for two reasons: first, integrating background compilation results into the user interface is very complicated; and second, our compiler currently does not perform very much error recovery, so the results of a background compile are likely to be not usable. The presentation compiler is run in the foreground on an editor's buffer of source code in one of the following situations: when the editor obtains focus, when a presentation service is requested in the editor (text hover, hyperlinking), when the editor is saved, and on demand (ctrl-shift-Z). After the presentation compiler starts up, it runs fairly quickly and is not very noticeable to the programmer. However, initial start up time is very noticeable (around 20 seconds for a 10,000 LOC project), while we also cannot inform programmers of compiler errors interactively.

The Scala plug-in currently reuses many formatting components in the JDT such as the parenthesis matcher and auto-indenter. The use of the JDT auto-indenter, which performs heuristic parsing, works moderately well for Scala with some minor modifications. To achieve better results, we will have to write our

own auto-indenter and other Scala-specific formatting components from scratch. This is especially true for folding, where Scala block syntax is significantly different from Java.

## Launching

The final part deals with the launching and debugging of a Scala program. Because Scala programs are composed of class files that execute on the JVM, we can reuse most of the JDT's support for running and debugging programs. We modified the launch configuration user interface to be more Scala-like, which unfortunately required copying and modifying existing JDT code. Reusing the JDT debugger requires installing our own Java breakpoints that the Java Development Interface (JDI) can understand without referring to Java source code. Although this requires writing some of our own code, we are able to reuse a majority of the JDT debugging functionality.

## Lessons and Recommendations

We summarize here our experience in supporting a Java-like language in Eclipse. For those that are building compilers that they want to later integrate into an IDE, it is very useful to consider these IDE integration issues early. In this regard, we have learned the following lessons:

- Resident compilation is a sophisticated feature that is best designed into the compiler early on in its construction. Even then, consistency issues such as the use of stale compilation information make resident compilation tricky to get right.
- Be very careful about the level of de-sugaring that is performed before type checking is completed. De-sugaring requires re-sugaring, which may be impossible to perform if relevant information is not preserved. For example, type aliases in Scala are completely replaced with their bindings before type checking is completed, and there is currently no way to recover the type alias for presentation purposes.
- Try to preserve as much lexical information as possible. The offsets of identifiers as well as many keywords and operators will be needed presentation purposes. Likewise, comments must also be preserved.

The reuse of Eclipse components allowed us to develop a functionally complete plug-in in about a four month period of time. However, many reused components are hidden inside the JDT and were reused through either internal APIs or through copying and editing. This unfortunately means that we must either upgrade our copied code or carefully re-test and probably change our plug-in for every Eclipse release. Additionally, JDT components often depend on the existence of a Java model, which is derived from Java source code, in very minor ways. For example, the JDT's Package Explorer will not color orange packages that do not contain Java files. Because Scala files are not Java files, all Scala

packages are white. Additionally, the Package Explorer is integrated with the Java content outliner and does not know about the existence of the Scala content outliner. Conceivably, the Package Explorer as well as other JDT classpath and launching components do not need to depend on the Java model directly so this dependency could be refactored into separate plug-ins. Such plug-ins could then be reused by other plug-ins for Java-like languages, making such plug-ins easier to develop.

## **Conclusion**

Our Scala Eclipse plug-in is currently being used in a growing community of Scala programmers and by students in our university-level programming courses. Although the Scala plug-in still lacks many of the features and polish of the JDT, it has significantly reduced Scala's "tool-penalty" and increases the community of programmers who can explore Scala's language-based features. In the future, our goal is to eventually surpass the JDT using new incremental compiler technology that we are currently developing. However, the reuse of JDT components in our plug-in will always be important. We hope that in the future that the Java model can be better factored in the JDT. This would then make it easier to support Java-like languages such as Scala.