
Build an Eclipse development environment for Perl, Python, and PHP

Use the Dynamic Languages Toolkit (DLTK) to create your own IDE

Skill Level: Intermediate

[Matthew Scarpino](#)

Java Developer

Eclipse Engineering, LLC

[Nathan A. Good](#)

Senior Consultant and Freelance Developer

Freelance Developer

03 Feb 2009

Updated 27 Oct 2011

Eclipse presents a wealth of capabilities for building tools for compiled languages like C and the Java™ programming language but provides little support for scripting languages like Perl, Python, and PHP. For these and similar languages, the Eclipse Dynamic Languages Toolkit (DLTK) comes to the rescue. Walk through the process of building a DLTK-based IDE, and discover sample code for each step.

Section 1. Before you start

About this tutorial

This tutorial shows how Eclipse's Dynamic Languages Toolkit (DLTK) makes it possible to build development tools for scripting languages. In particular, it explains

how to implement syntax coloring, user preferences, and interpreter integration in a plug-in-based project.

Objectives

Frequently used acronyms

- **DLTK**: Eclipse Dynamic Languages Toolkit
- **GPL**: Gnu Public License
- **IDE**: Integrated Development Environment
- **JRE**: Java Runtime Environment
- **MVC**: Model-View-Controller
- **SWT**: Standard Widget Toolkit
- **UI**: User Interface

This tutorial explains—one step at a time—how to build a DLTK-based development environment. The discussion presents DLTK by focusing on a practical plug-in project based on the Octave numerical computation language. The topics covered include:

- Creating a plug-in project
- Configuring the editor and Eclipse DLTK text tools
- Adding classes to control syntax coloring in the text editor
- Enabling user preferences
- Integrating the script interpreter into the development environment
- Adding a custom console to communicate between the workbench and the interpreter

Prerequisites

This tutorial is written for Java™ developers familiar with Eclipse and interested in building tools for dynamic languages, such as PHP, Tcl, Ruby, and Python. It assumes that you have a basic understanding of plug-ins and Eclipse-based development tools in general.

System requirements

To build the example project in this tutorial, you need a computer with an Eclipse installation (the examples in this tutorial use version 3.7 or later) and a current JRE. You must install a recent version of the DLTK, which we describe in the section "Install the DLTK." (See [Resources](#) for links to download Eclipse.)

Section 2. Introducing DLTK and the DLTK editor

Eclipse offers many capabilities for building custom development tools, but they're not easy to use or understand. Many of the classes, such as `TextEditor`, require significant configuration to work properly. In contrast, the DLTK provides a prepackaged development environment that requires only minor tweaking to be operational. Building a development tool with the DLTK is like baking a cake from a mix: The output isn't completely original, but you get a maximum of quality and reliability with a minimum of effort.

A second important difference between DLTK and Eclipse involves the supported languages. Eclipse's build capabilities are based on compiled languages like Java, C, and C++. The DLTK is geared toward *dynamic languages*, which include scripting languages like PHP, Ruby, and Tcl. Ideally, a development tool for these languages supports easy integration of script interpreters, command evaluation on a line-by-line basis, and the ability to enter commands at a console. Eclipse doesn't provide these features, but the DLTK supports them all.

Install the DLTK

If you haven't installed the DLTK, do so now. Perform the following steps:

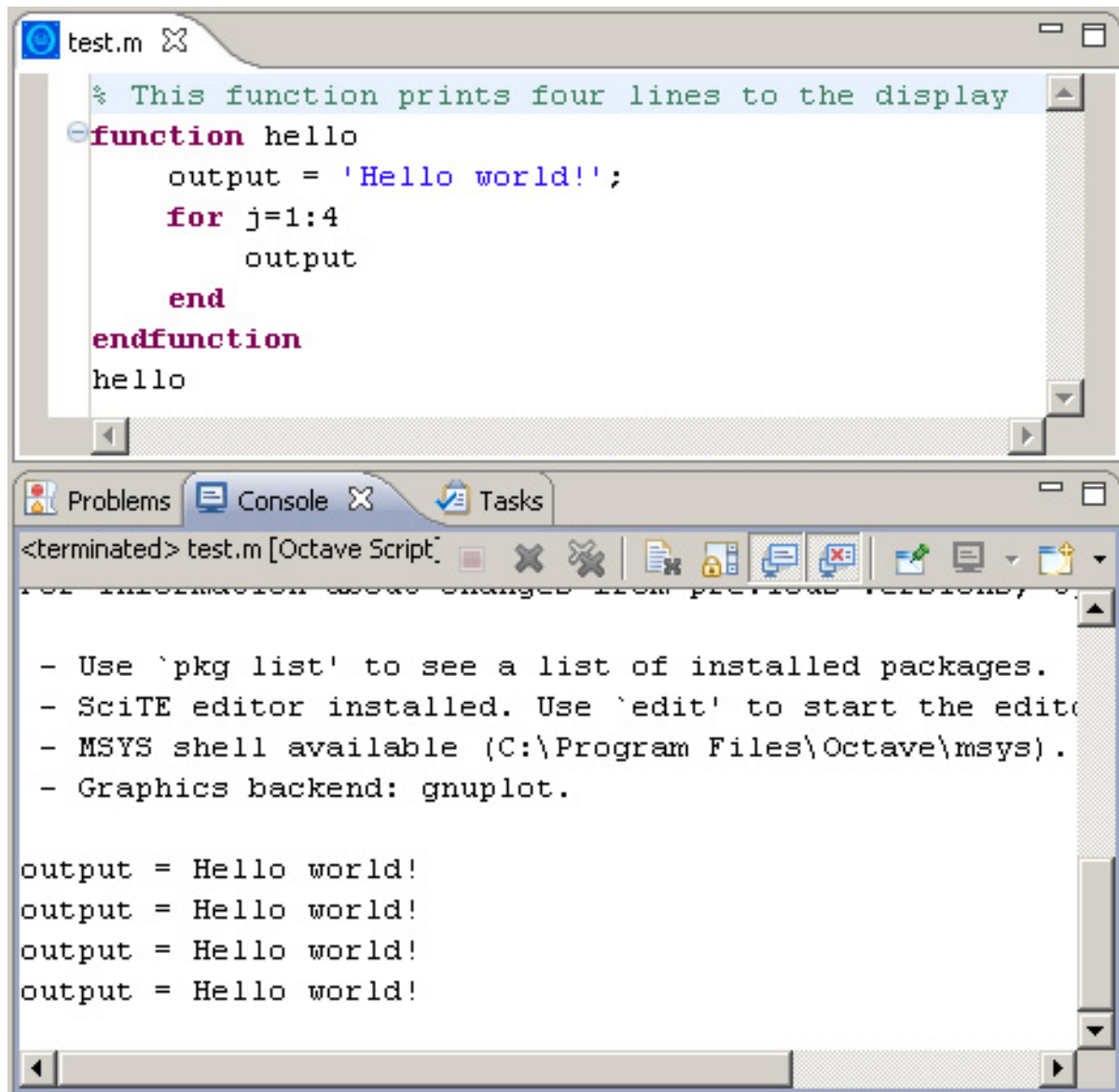
1. Open Eclipse, then click **Help > Install New Software**.
2. Select **Indigo** - <http://download.eclipse.org/releases/indigo> from the **Work with** list.
3. Select the **General Purpose Tools > Dynamic Languages Toolkit - Core Frameworks** check box, then click **Next**.
4. Click **Next** after reviewing the prerequisites that will also be installed.
5. If you accept the license agreements, click **Finish** to install the plug-in.
6. After installing the files, restart Eclipse.

The DLTK-based plug-in project: The Octave IDE

This tutorial does more than just discuss the DLTK and its features. It walks you through the steps of creating an DLTK-based IDE for the Octave scripting language. Octave is a GPL-licensed toolset for performing mathematical computation, particularly with matrices. Octave commands are easy to understand, and the language closely resembles that of the popular proprietary Matlab tool. You can download the example code from [Download](#).

The Octave IDE you'll be building won't have all of Eclipse's bells and whistles, but you will be able to create Octave scripts (*.m files), edit them in a multifeatured text editor, run the Octave interpreter with the click of a button, and see the result displayed in the IDE's console. [Figure 1](#) shows what the editor and console look like.

Figure 1. The Octave IDE



This tutorial explains the process of building the Octave IDE in four steps:

1. Create the Octave editor and add features.
2. Store and respond to user preferences.
3. Integrate the Octave interpreter.
4. Display the interpreter's output in the console.

The first task deals with the Octave script editor. There's a lot to learn about this subject, but before you start coding, let's take a step back and look at how DLTK

editors work. Much of this discussion applies to all Eclipse text editors.

A high-level introduction to the DLTk text editor

At its most basic level, the DLTk text editor is just a regular SWT `StyledText` widget with a lot of additional capabilities. These capabilities fall into one of two categories:

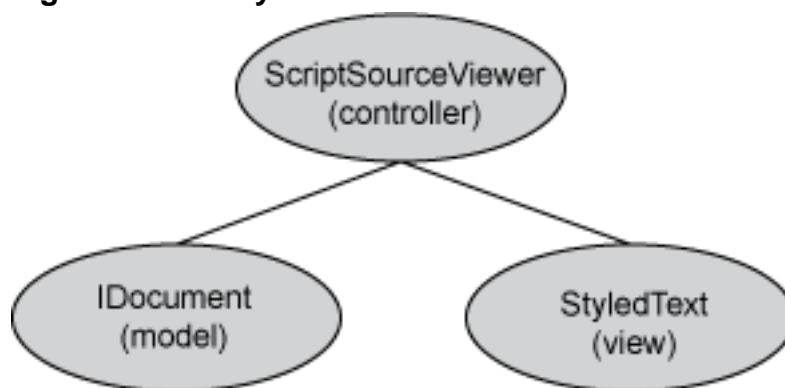
- **Responding to user events.** Changing the editor's display according to the user's keystrokes
- **File I/O.** Transferring character data between the IDE and files on the user's system

In DLTk editors, event handling is managed by a `ScriptSourceViewer` object. This object, created by the `ScriptEditor`, wraps around the `StyledText` widget and manages its response to user keystrokes. Typically, this response involves modifying the widget's presentation or updating the editor's internal data. This data is embodied in an `IDocument` object, which not only stores text but also information related to line numbers, positions, and regions, called *partitions*.

The `ScriptSourceViewer` updates the `IDocument` by accessing an object called a `SourceModuleDocumentProvider`. In addition to providing access to the `IDocument`, this object handles file operations. DLTk handles all the editor's file interactions for you, but if you're interested in learning more, read the documentation on the object's superclass: `TextFileDocumentProvider`.

If you're familiar with the MVC pattern, the relationship between the `ScriptSourceViewer`, `IDocument`, and `StyledText` widget should be clear. The `IDocument` serves as the editor's Model aspect, containing data independent of its presentation. The `StyledText` widget represents the editor's View aspect, and the `ScriptSourceViewer` serves as the Controller, managing communication among the Model, the View, and the user. [Figure 2](#) shows this relationship.

Figure 2. Primary elements of the DLTk editor



If the explanation thus far makes sense and Figure 2 seems reasonable, you should have no difficulty understanding the technical discussion that follows. In the next section, you start creating the Octave plug-in.

Section 3. Getting started with a DLTk project

For the sake of simplicity, this tutorial integrates all the Octave IDE functionality into a single plug-in. Generally, you implement a feature using multiple plug-ins: one containing core classes, one containing UI classes, one containing debug classes, and so on. But this tutorial's code only needs one plug-in, so if you have Eclipse running, click **File > New > Project**, select the **Plug-in Project** option, and click **Next** to begin creating the plug-in project. Follow the steps here to create a plug-in project that includes a sample editor, which is a good starting point for your new plug-in.

1. Enter the project name, which in this example is *org.dworks.octaveide*, as shown in [Figure 3](#).

Figure 3. Entering the project's name

New Plug-in Project

Plug-in Project
Create a new plug-in project

Project name:

☒ Use default location

Location:

Project Settings

☒ Create a Java project

Source folder:

Output folder:

Target Platform
This plug-in is targeted to run with:

☒ Eclipse version:

☐ an OSGi framework:

Working sets

☐ Add project to working sets

2. In the **Content** screen, shown in [Figure 4](#), enter `org.dworks.octaveide` as the **ID** if it's not already set, and click **Next**.

Figure 4. The Content screen

New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties

ID:

Version:

Name:

Provider:

Execution Environment: [Environments...](#)

Options

☒ Generate an activator, a Java class that controls the plug-in's life cycle
Activator:

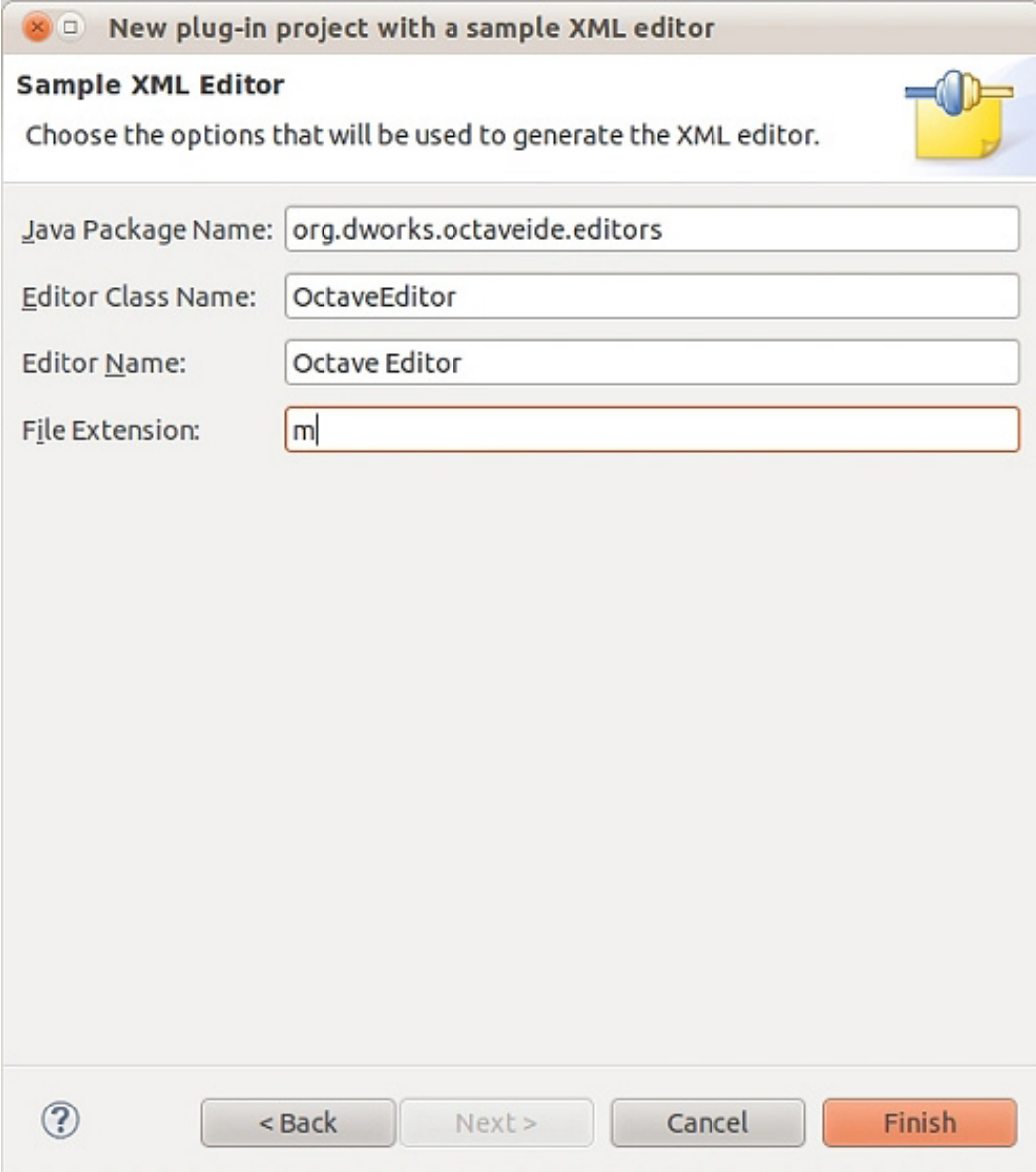
☒ This plug-in will make contributions to the UI

☐ Enable API Analysis

Rich Client Application
Would you like to create a rich client application? ☐ Yes ☒ No

[?](#) [< Back](#) [Next >](#) [Cancel](#) [Finish](#)

3. Select the **Create a plug-in using one of the templates** check box, select **Plug-in with an editor** from the list, and click **Next**.
 4. Enter `OctiveEditor`, as shown in [Figure 5](#), enter `m` for the **File Extension**, and then click **Finish** to create your plug-in project.
- Figure 5. Entering the information for the editor**



New plug-in project with a sample XML editor


Sample XML Editor
Choose the options that will be used to generate the XML editor.

Java Package Name:

Editor Class Name:

Editor Name:

File Extension:



Your new plug-in project is created with the `plugin.xml` configuration file already available in your project, and you're off to a good start. You can edit the `plugin.xml` configuration using the built-in editor, or you can edit the source of the file directly by clicking the editor's **plugin.xml** tab. This tutorial shows you the listing of the `plugin.xml` file itself.

Before continuing with the rest of the tutorial, you should add two dependencies that allow your plug-in code to find the DLTK classes. In the **Dependencies** tab of your `plugin.xml` editor, click **Add**, and add `org.eclipse.dltk.core` and `org.eclipse.dltk.ui`. The current version in this tutorial is 3.0.1.

Configure the Octave plug-in

The `plugin.xml` file that was started when you created your plug-in project contains information about the plug-in's extensions. These extensions tell Eclipse how the plug-in intends to contribute to the workbench. To start, the `plugin.xml` file needs two simple extensions: one that identifies the Octave editor and one that identifies the types of files to be edited. The first extension was already created when you created the plug-in project using the template.

[Listing 1](#) presents the first extension, which defines characteristics of the Octave editor.

Listing 1. The Octave IDE editor extension

```
<extension point="org.eclipse.ui.editors">
  <editor
    id="org.dworks.octaveide.editor.OctaveEditor"
    class="org.dworks.octaveide.editor.OctaveEditor"
    contributorClass="org.dworks.octaveide.actions.OctaveActionContributor"
    default="true"
    icon="icons/oct.gif"
    name="OctaveEditor">
    <contentTypeBinding
      contentTypeId="org.dworks.octaveide.octaveContentType">
    </contentTypeBinding>
  </editor>
</extension>
```

Thoughts on the DLTK

The more we learn about the DLTK, the more impressed we are. The DLTK creators thought of everything, and this tutorial barely scratches the surface. Whenever we find ourselves wondering whether the DLTK supports a feature we're curious about, it turns out that it does and that the DLTK creators were way ahead of us.

However, the DLTK has its shortcomings. It's a necessarily complicated system, and the documentation is sparse at best, nonexistent at worst. There are a few helpful comments in the source code, but not many. Thankfully, the DLTK examples—particularly for the Tcl editor—clear up a lot of confusion regarding how the DLTK is used in practice.

This extension contains several important attributes. The `class` attribute defines the primary class that embodies the editor: `OctaveEditor`, which you will create shortly. The `contributorClass` attribute identifies the class that provides actions for the editor. And the `default` attribute states that the `OctaveEditor` should be the default editor for associated files. The `name` attribute identifies the text that will be displayed for the editor. As defined in the `plugin.properties` file, the `OctaveEditor.name` property corresponds to "Octave Script Editor."

The last supplement in the extension, `contentTypeBinding`, requires explanation. Eclipse maintains a registry of content types that represent file formats and naming conventions. By adding a `contentTypeBinding` element to the editor's extension, you can associate the editor with file suffixes, file names, and even file aliases. But for your Octave editor, you're only interested in *.m files, so your extension of `org.eclipse.core.runtime.contentTypes` is simple. [Listing 2](#) shows this second extension.

Listing 2. The Octave IDE ContentType extension

```
<extension point="org.eclipse.core.runtime.contentTypes">
  <content-type
    id="org.dworks.octaveide.octaveContentType"
    base-type="org.eclipse.core.runtime.text"
    file-extensions=".m"
    name="%octaveContentType"
    priority="high">
  </content-type>
</extension>
```

Many more attributes are available for customizing content types, but this declaration identifies the `org.dworks.octaveide.octaveContentType` content type as being text based and associated with the file suffix `.m`. The `priority` attribute states that if a file becomes associated with multiple content types, this type should be given high priority.

Creating a DLTk language toolkit class

A principal advantage of using the DLTk is that it's simple to customize how IDEs work. This customization is made possible through DLTk's language toolkit interfaces (`IDLTkCoreLanguageToolkit`, `IDLTkDebugUILanguageToolkit`, `IDLTkUILanguageToolkit`, and so on). These interfaces present methods that, when implemented, configure many aspects of the IDE, such as parsing, color usage, user preferences, and accessing the interpreter.

This tutorial focuses primarily on the DLTk UI, so you'll only concern yourself with the `IDLTkUILanguageToolkit`. The methods in this interface perform one of two functions: They either return a class that handles editor configuration or provide the ID of a configuration object declared in `plugin.xml`. Seven of these methods are listed here in [Table 1](#).

Table 1. Methods on the `IDLTkUILanguageToolkit` interface

Method name	Description
<code>getEditorId(Object elementID)</code>	Returns the editor's ID
<code>getTextTools()</code>	Returns the <code>ScriptTextTools</code> object that will customize how text is presented in the editor

<code>createSourceViewerConfiguration()</code>	Returns the <code>ScriptSourceViewerConfiguration</code> object that will configure the operation of the editor's source viewer
<code>getPreferenceStore()</code>	Returns the <code>IPreferenceStore</code> object that contains user settings
<code>getEditorPreferencePages()</code>	Returns the IDs of the preference pages related to editor settings
<code>getInterpreterPreferencePage()</code>	Returns the IDs of the preference pages related to the script interpreter
<code>getIntepreterId()</code>	Returns the ID of the script interpreter

In the example code, the `OctaveUILanguageToolkit` class fleshes out these methods with objects and identifiers. The first two objects discussed relate to the first two methods in the toolkit: `OctaveEditor` and `OctaveTextTools`.

Create the Octave editor and the Octave text tools

Now that you've configured the editor's extension, the first order of business is to create the `OctaveEditor`. Thanks to the DLTK, it won't require much coding: The DLTK's `ScriptEditor` class (the superclass of `OctaveEditor`) handles most of the editor's operations on your behalf. All you need are a few configuration methods. In the `OctaveEditor`, the most important of these is `getTextTools()`, which returns a `ScriptTextTools` object.

The `ScriptTextTools` class is convenient. This central class accesses several other classes that provide the editor's capabilities. By subclassing `ScriptTextTools`, an editor can customize how these capabilities are performed. The Octave editor creates a subclass called `OctaveTextTools`, which provides access to the important objects shown in [Table 2](#).

Table 2. Objects made available by OctaveTextTools

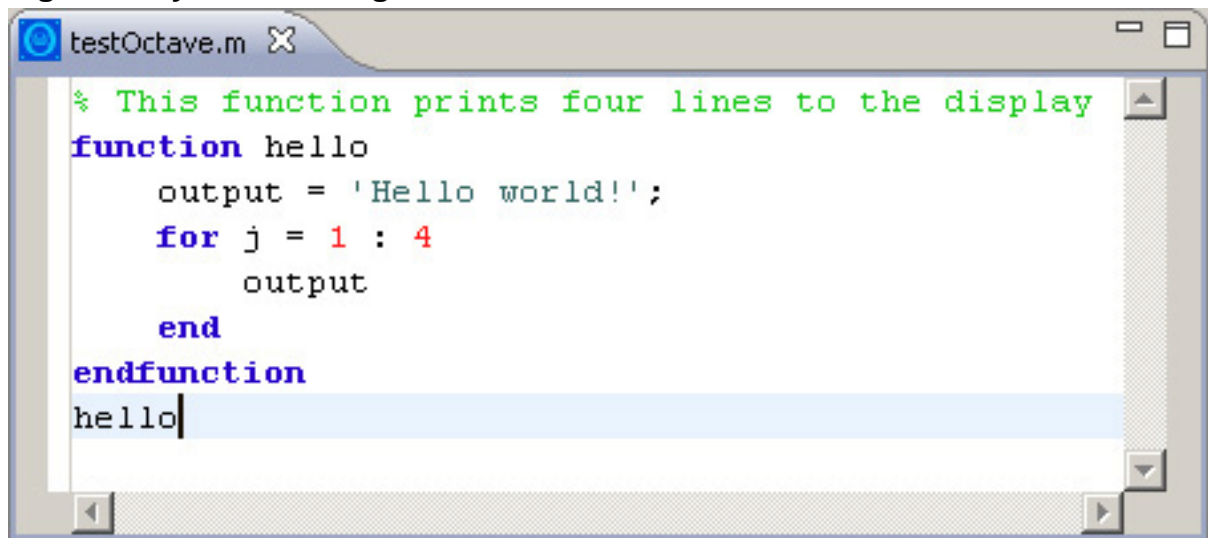
Object	Description
<code>OctavePartitionScanner</code>	Reads the editor text and determines partition boundaries
<code>OctaveSourceViewerConfiguration</code>	Configures the <code>ScriptSourceViewer</code> and the way it responds to user-generated events

These objects become particularly important in implementing *syntax coloring*—that is, changing the color of text depending on the text's syntactic function in code. The next topic explains this feature in greater detail.

Section 4. Syntax coloring in a DLTk editor

Syntax coloring is one of the most noticeable aspects of any professional source code editor. Not only do the colors make the code easier to read, but they also make typing errors readily apparent. [Figure 6](#) shows how comments, numbers, keywords, and strings are colored differently from regular code.

Figure 6. Syntax coloring in the Octave IDE



Configure syntax coloring in the Octave editor

The DLTk process of syntax coloring consists of four steps:

1. The `ScriptSourceViewer` responds to keystrokes by alerting the document's partitioner.
2. The partitioner splits the document's text into partitions according to rules.
3. When the partitioning is finished, the viewer alerts a presentation reconciler to analyze the modified partition.
4. The reconciler calls upon a damager/repairer to analyze the affected region and apply syntax coloring.

In this manner, the editor uses a divide-and-conquer strategy to color text. First, it splits the document into partitions. Then, it analyzes the modified partition and determines whether and where syntax coloring should be updated. The next two

sections explain this process in greater depth and show how to implement them in the Octave IDE.

Document partitioning

In addition to text, an `IDocument` stores an array of `Positions` that serve as boundaries of regions within text. These regions not only make syntax coloring possible (comments one color, code another color) but also let you apply different tools to different parts of the text. For example, there's no reason to tell your script parser to analyze commented lines. Using partitions, you can make sure your parser only operates on regions containing actual code.

Document partitioning is accomplished by two central objects: one that implements `IDocumentPartitioner` and one that implements `IPartitionTokenScanner`. The partitioner delivers text to the scanner, which analyzes the text and produces `ITokens`. The partitioner uses these tokens to set `Positions` within the document.

It would be nice if the `OctaveEditor` automatically initialized the partitioner and scanner whenever the user opens an *.m file. But the document's configuration must be explicitly defined in `plugin.xml` with an extension of the `org.eclipse.core.filebuffers.documentSetup` extension point. This extension identifies the class that should be called on to configure the document during its initialization. [Listing 3](#) presents the extension defined for the Octave IDE plug-in.

Listing 3. Configuring the document

```
<extension point="org.eclipse.core.filebuffers.documentSetup">
  id="org.dworks.octaveide.editor.OctaveDocumentSetup"
  name="%documentSetupName"
  <participant
    extensions="m"
    class="org.dworks.octaveide.editor.OctaveDocumentSetup">
  </participant>
</extension>
```

The `OctaveDocumentSetup` calls on the `DLTK ScriptTextTools` object to initialize *.m documents. By default, this creates a `FastPartitioner` to serve as the document's partitioner.

Each `DLTK` editor has to create its own `IPartitionTokenScanner` object to read characters and produce tokens corresponding to partitions. For example, if you want to create a partition for text starting with `/*` and ending with `*/`, you need to configure the scanner so that it forms the appropriate token when it encounters this text pattern. [Table 3](#) lists how the Octave plug-in divides text into three partitions.

Table 3. Partitions into which the Octave plug-in divides text

Partition	Description
Comments	Identified by <code>OCTAVE_COMMENT</code> , which equals <code>"__octave_comment__"</code>
Strings	Identified by <code>OCTAVE_STRING</code> , which equals <code>"__octave_string__"</code>
Regular script code	Identified by <code>IDocument.DEFAULT_CONTENT_TYPE</code> , which equals <code>"__dftl_partition_content_type"</code>

These partitions are defined in the `IOctavePartititons` interface. They are used to initialize the `FastPartitioner`, which sends them on to the partition scanner.

Partitions in the Octave editor can be identified using a set of logical rules, so your `OctavePartitionScanner` will extend the `RuleBasedPartitionScanner` class. The most important method in this class is `setPartitionRules`, which accepts an array of `IPredicateRule` objects. The goal of a rule is to convert sections of text into `ITokens`. Rules can be difficult to code from scratch, but thankfully, Eclipse provides many useful implementations of the `IPredicateRule` interface. The Octave IDE uses two of them to recognize partitions: `SingleLineRule` and `MultiLineRule`. Both classes have similar constructors, and they are initialized with the following information (at least):

- The character that begins the partition
- The character that ends the partition
- The `IToken` corresponding to the recognized partition

For example, say you want to create a rule that produces a string token when it finds text starting and ending with a single quotation mark (`'`). You do this with the following code:

```
IToken stringToken = new Token(OctavePartitions.OCTAVE_STRING);
new SingleLineRule("'", "'", stringToken, '\\');
```

[Listing 4](#) presents the entire code that `OctavePartitionScanner` uses to detect partitions in Octave scripts. The Octave editor creates comment partitions when it encounters text preceded by a percent sign (`%`) and followed by a new line. String partitions are delimited by single quotation marks.

Listing 4. The Octave IDE partition scanner

```
// Declare the different tokens
IToken string = new Token(IOctavePartitions.OCTAVE_STRING);
IToken comment = new Token(IOctavePartitions.OCTAVE_COMMENT);
```



```
// Create the list of rules that produce tokens
List<IPredicateRule> rules = new ArrayList<IPredicateRule>();
rules.add(new SingleLineRule("%", "\n", comment));
rules.add(new SingleLineRule("'", "'", string));
IPredicateRule[] result = new IPredicateRule[rules.size()];
rules.toArray(result);
setPredicateRules(result);
```

Having created the `OctavePartitionScanner` and initialized its rules, you've finished configuring the editor's partitioning. The next step is to tell the editor how to scan and color text inside the partitions.

Scan text inside partitions

As mentioned, DLTk editors rely on a `ScriptSourceViewer` to respond to user events like keystrokes. You can tailor the viewer's operation with a `ScriptSourceViewerConfiguration` object. Like the `ScriptTextTools` class, the configuration object does little work of its own. Instead, its methods identify classes that assist the source viewer's operation. When it comes to syntax coloring, two of the most important of these "assistance" methods are shown here in [Table 4](#).

Table 4. Assistance methods for syntax coloring

Method name	Description
<code>getPresentationReconciler()</code>	Returns an <code>IPresentationReconciler</code> that controls how text is presented when changes occur
<code>initializeScanners()</code>	Returns an array of <code>AbstractScriptScanners</code> that perform the same type of text scanning as the <code>RuleBasedPartitionScanners</code> described earlier

These two methods are closely related. The following discussion explains how the Octave IDE implements them to provide text coloring.

The `IPresentationReconciler`

In Eclipse parlance, a *reconciler* is an object that keeps track of a document's content and responds to changes. A *presentation reconciler* is a reconciler that responds to document changes by altering how text is presented. Specifically, the `IPresentationReconciler` responds to changes by calling on the objects shown in [Table 5](#).

Table 5. Objects used by `IPresentationReconciler`

Object name	Description
-------------	-------------

IPresentationDamager	Determines the extent of the document modification
IPresentationRepairer	Creates a TextPresentation object to respond to the modification identified by the IPresentationDamager

These two objects operate in sequence. When the user modifies text in a partition, the `IPresentationReconciler` gives the partition name to the `IPresentationDamager`. After the damager identifies the modified region, the `IPresentationRepairer` analyzes the region to see if, where, and how the partition's text presentation should be updated.

To keep matters simple, Eclipse provides a `DefaultDamagerRepairer` that serves as damager and repairer. The Octave IDE constructs a `DefaultDamagerRepairer` for each of the three partitions and initializes each with a different scanner. This reconciler is shown in [Listing 5](#).

Listing 5. Configuring the presentation reconciler

```
// Create a DefaultDamagerRepairer for the default partition (code)
DefaultDamagerRepairer dr = new DefaultDamagerRepairer(codeScanner);
reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

// Create a DefaultDamagerRepairer for string partitions
dr = new DefaultDamagerRepairer(stringScanner);
reconciler.setDamager(dr, OctavePartitions.OCTAVE_STRING);
reconciler.setRepairer(dr, OctavePartitions.OCTAVE_STRING);

// Create a DefaultDamagerRepairer for comment partitions
dr = new DefaultDamagerRepairer(commentScanner);
reconciler.setDamager(dr, OctavePartitions.OCTAVE_COMMENT);
reconciler.setRepairer(dr, OctavePartitions.OCTAVE_COMMENT);
```

Just as the partition scanner identifies partitions using rules, each of these scanners creates tokens according to similar rules. The next section discusses text scanners in detail.

Scanning text inside partitions

The constructor of the `ScriptSourceViewerConfiguration` calls `initializeScanners()` to create the scanners needed to analyze text. [Listing 6](#) presents the full implementation of `initializeScanners()` in the `OctaveSourceViewerConfiguration` class.

Listing 6. Creating the script scanners

```
private AbstractScriptScanner codeScanner, stringScanner, commentScanner;

protected void initializeScanners() {
```

```

// Create a scanner for the script code
codeScanner = new OctaveCodeScanner(getColorManager(), fPreferenceStore);

// Create a scanner for string partitions
stringScanner = new SingleTokenScriptScanner(getColorManager(),
fPreferenceStore, IOctavePartitions.OCTAVE_STRING);

// Create a scanner for comment partitions
commentScanner = new SingleTokenScriptScanner(getColorManager(),
fPreferenceStore, IOctavePartitions.OCTAVE_COMMENT);
}

```

Because the partition scanner already identified strings and comments, these partitions don't need to be reanalyzed. This is why the `stringScanner` and `commentScanner` are both `SingleTokenScriptScanners`. This class doesn't make decisions based on rules but always returns the same type of `IToken`. For example, the `stringScanner` returns a token initialized with `OctavePartitions.OCTAVE_STRING`, and the `commentScanner` returns a token initialized with `OctavePartitions.OCTAVE_COMMENT`. You'll see how these token properties determine text color shortly.

The `codeScanner` is more involved and performs the brunt of the IDE's text analysis. This scanner is an `OctaveCodeScanner` and, like the `PartitionTokenScanner`, creates tokens based on rules. But instead of the `MultiLineRule` or `SingleLineRule`, it relies on the `IRule` implementations shown in [Table 6](#).

Table 6. IRule implementations used by OctaveCodeScanner

Rule	Description
WordRule	Returns a token when the scanner encounters specific words
NumberRule	Returns a token when the scanner encounters numbers

These rules are created in the `createRules` method of the `OctaveCodeScanner`. [Listing 7](#) presents this method in full.

Listing 7. Creating the script scanners

```

protected List<IRule> createRules() {

    // Create tokens
    IToken keywordToken = getToken(DLTKColorConstants.DLTK_KEYWORD);
    IToken numberToken = getToken(DLTKColorConstants.DLTK_NUMBER);
    IToken defaultToken = getToken(DLTKColorConstants.DLTK_DEFAULT);

    // Create and populate list
    List<IRule> ruleList = new ArrayList<IRule>();

    // Create word rule to detect keywords
    WordRule wordRule = new WordRule(new OctaveWordDetector(), defaultToken);
}

```

```
for (int i = 0; i < IOctaveKeywords.keywords.length; i++)
    wordRule.addWord(IOctaveKeywords.keywords[i], keywordToken);
ruleList.add(wordRule);

// Create number rule to detect numbers
NumberRule numberRule = new NumberRule(numberToken);
ruleList.add(numberRule);

// Set the token returned for default text
setDefaultReturnToken(defaultToken);
return ruleList;
}
```

It's important to distinguish the tokens created in Listing 7 from those created in [Listing 4](#). The `Token` constructor accepts any `Object` as its argument, and this `Object` tells the token's recipient what action to take. In [Listing 4](#), `Tokens` are initialized with partition names that tell the `FastPartitioner` how to update the document's partitioning. In Listing 7, `Tokens` are initialized with `TextPresentation` objects, which tell the IDE how to present text in the editor.

Listing 7 calls `getToken` to associate each `Token` with a `TextPresentation`. This method searches through a `Map` populated with the user's preferences. To understand how this map is updated, it's important to understand how DLTK preferences work. This is the subject of the following discussion.

Section 5. Configuring DLTK preferences

The previous discussion showed how text presentation depends on the user's preferences. Now, we will go farther, showing how to create pages that receive the user's preferences. You can access these preferences by clicking **Window > Preferences** in Eclipse. At the conclusion of this discussion, we explain how the user's preferences make it possible to integrate the script interpreter into the IDE.

Preference stores

An Eclipse *preference store* is like a property file: It contains name-value pairs that identify the user's preferred settings. But there are two important differences. Each preference holds a default value in addition to its current value, and preference values must take one of six basic types:

- `Boolean`
- `int`

- `long`
- `float`
- `double`
- `String`

Preferences are stored and retrieved by calling methods defined by the `IPreferenceStore` interface. The `setValue` and `setDefaultValue` methods add and update preferences. To retrieve a preference value, you can call one of `getBoolean`, `getInt`, `getLong`, `getFloat`, `getDouble`, or `getString`. Similar methods access default values, such as `getDefaultDouble`.

To access preferences related to SWT objects (more particularly, `RGB`, `Rectangle`, `FontData`, and `FontData[]` objects), you can call methods in the `PreferenceConverter` class.

During initialization, a DLTK `ScriptEditor` creates an `ArrayList` of four `IPreferenceStore` objects:

- The `ScopedPreferenceStore` created for the Octave plug-in (`org.dworks.octaveide`)
- The `ScopedPreferenceStore` created for the Eclipse text editor plug-in (`org.eclipse.ui.editors.text`)
- An `EclipsePreferencesAdapter` that provides preferences related to the project containing the script being edited
- A `PreferencesAdapter` that contains the preferences associated with the DLTK core

These stores are combined into a `ChainedPreferenceStore`. If two or more preference stores contain identical preferences, the value is supplied by the store that added first.

Initialize preferences

Eclipse makes it possible to set preferences during initialization with the extension point `org.eclipse.core.runtime.preferences`. The Octave IDE extends this point and identifies the `OctavePreferenceInitializer` class as providing default values for the IDE's preferences, which in turn extends the abstract class `AbstractPreferenceInitializer`, whose only required method is `initializeDefaultPreferences`. The IDE invokes this method during initialization, and [Listing 8](#) shows how it's implemented in code.

Listing 8. Initializing default preferences

```
public void initializeDefaultPreferences() {  
    IPreferenceStore store = OctavePlugin.getDefault().getPreferenceStore();  
    OctavePreferenceConstants.initializeDefaultValues(store);  
}
```

This simple routine accesses the Octave plug-in's preference store and calls on the `OctavePreferenceConstants` class to initialize the store's values. This is a subclass of DLTK's `PreferenceConstants` class, which defines and initializes a wide range of general scripting preferences. In this tutorial, the only preferences that will be initialized are those dealing with syntax coloring. [Listing 9](#) shows how this initialization is accomplished.

Listing 9. Setting syntax-coloring preferences

```
public static void initializeDefaultValues(IPreferenceStore store) {  
    // Set default preferences for the editor  
    PreferenceConstants.initializeDefaultValues(store);  
  
    // Make keywords blue and bold  
    PreferenceConverter.setDefault(store, DLTKColorConstants.DLTK_KEYWORD,  
        new RGB(40, 0, 200));  
    store.setDefault(DLTKColorConstants.DLTK_KEYWORD +  
        PreferenceConstants.EDITOR_BOLD_SUFFIX, true);  
  
    // Set default values for other preferences  
    PreferenceConverter.setDefault(store,  
        DLTKColorConstants.DLTK_SINGLE_LINE_COMMENT, new RGB(25, 200, 25));  
    PreferenceConverter.setDefault(store, DLTKColorConstants.DLTK_NUMBER,  
        new RGB(255, 25, 25));  
    PreferenceConverter.setDefault(store, DLTKColorConstants.DLTK_STRING,  
        new RGB(50, 100, 100));  
}
```

The first of the Octave-specific preferences deals with keywords. The code configures both the color and the style (bold, italics, strikethrough) of keywords in the IDE and requires two steps. First, the `PreferenceConverter` is called on to associate `DLTKColorConstants.DLTK_KEYWORD` with blue. Next, the `store.setDefault` method associates `DLTKColorConstants.DLTK_KEYWORD + PreferenceConstants.EDITOR_BOLD_SUFFIX` with `true`. These two lines ensure that if any Token is initialized with `DLTKColorConstants.DLTK_KEYWORD`, the corresponding text is displayed in blue and boldface.

Preference pages and configuration blocks

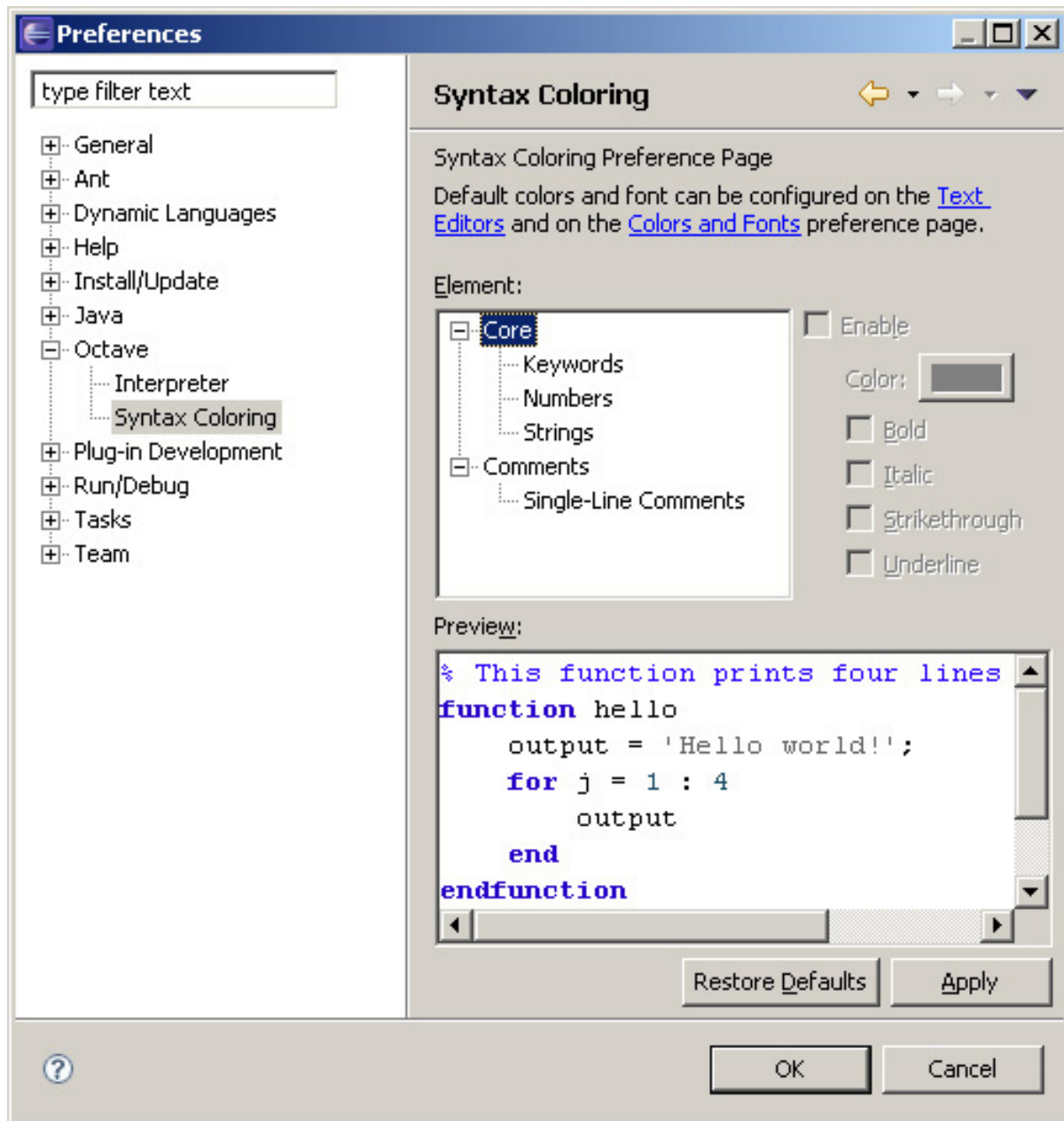
Now that you understand how to initialize preferences, it's important to give users the opportunity to configure preferences themselves. Eclipse makes this possible

through *preference pages*. DLTk simplifies the process of constructing these pages with objects called *configuration blocks*. This section describes both of these classes and how they work together to provide user preferences in the Octave IDE.

Eclipse preference pages

When you click **Window > Preferences** in Eclipse, the left pane of the **Preferences** window presents a hierarchy of user-configurable options. For the Octave IDE, you want a top-level preference called `Octave`, with sub-preferences for syntax coloring and the Octave interpreter. [Figure 7](#) shows what this window should look like.

Figure 7. The Octave Preferences window



In the Octave project, `plugin.xml` defines three preference topics by extending the point `org.eclipse.ui.preferencePages`. Each extension requires `id`, `name`, and `class` attributes. The two sub-preferences have an additional attribute—`category`—that names the `id` of the parent preference. These extensions are shown in [Listing 10](#).

Listing 10. Octave preference extensions

```
<extension point="org.eclipse.ui.preferencePages">
  <page
    class="org.dworks.octaveide.preferences.OctaveMainPreferencePage"
```



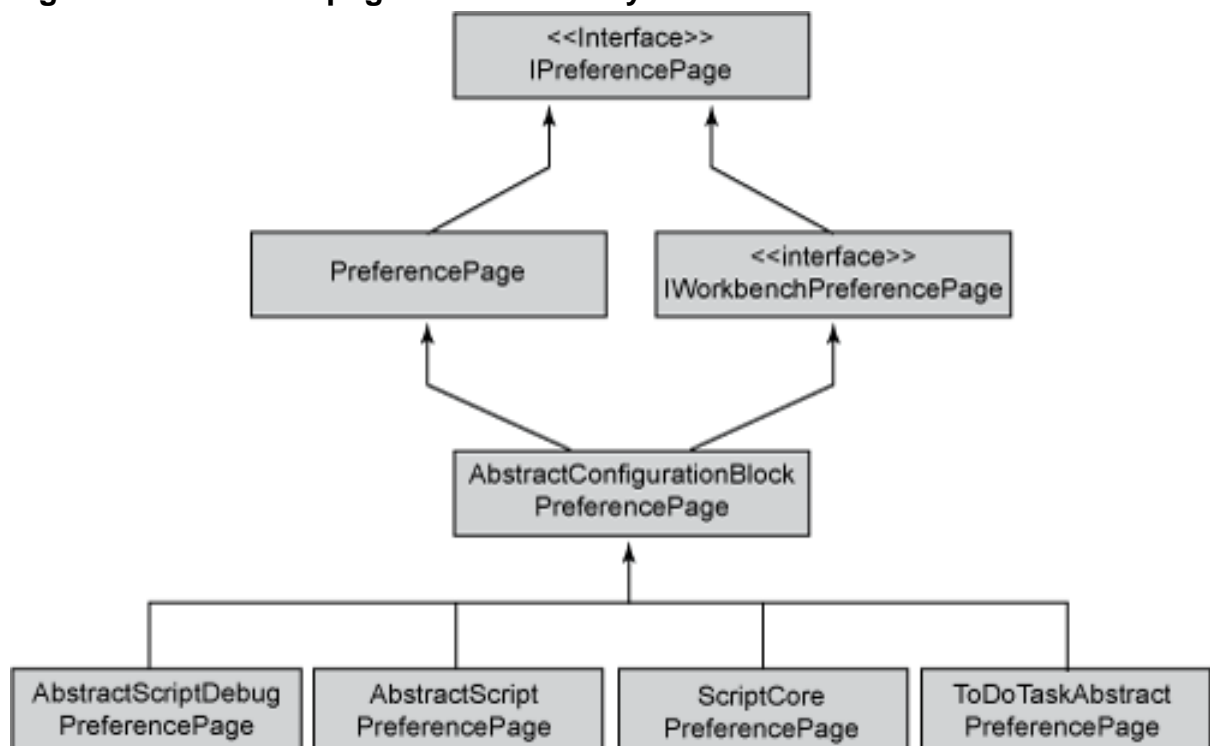
```

    id="org.dworks.octaveide.preferences.OctaveMainPreferencePage"
    name="%MainPreferencePage.name"/>
  <page
    category="org.dworks.octaveide.preferences.OctaveMainPreferencePage"
    class="org.dworks.octaveide.preferences.OctaveSyntaxColorPage"
    id="org.dworks.octaveide.preferences.OctaveSyntaxColorPage"
    name="%SyntaxColorPreferencePage.name"/>
  <page
    category="org.dworks.octaveide.preferences.OctaveMainPreferencePage"
    class="org.dworks.octaveide.preferences.OctaveInterpreterPreferencePage"
    id="org.dworks.octaveide.preferences.OctaveInterpreterPreferencePage"
    name="%InterpreterPreferencePage.name"/>
</extension>

```

The `class` attribute names a class that implements `IWorkbenchPreferencePage`, a subinterface of `IPreferencePage`. DLTK provides its own implementation of `IWorkbenchPreferencePage` called `AbstractConfigurationBlockPreferencePage`. DLTK also provides several subclasses for specific purposes. These class-interface relationships are depicted in [Figure 8](#).

Figure 8. Preference page class hierarchy



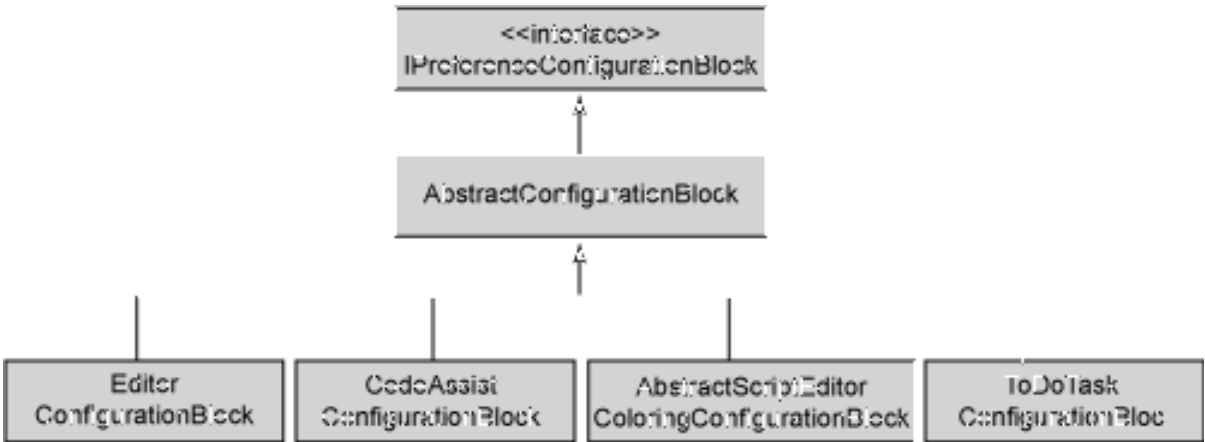
The three classes in Listing 10 are all direct subclasses of `AbstractConfigurationBlockPreferencePage`. If you look through their Java files, you'll see they don't contain much code. Each class provides an ID, a label, and access to the plug-in's preference store. Further, every `AbstractConfigurationBlockPreferencePage` must implement the method `createConfigurationBlock()`, which constructs the object that provides the page's graphical aspect. This object must implement the DLTK interface

`IPreferenceConfigurationBlock`. This important interface is discussed next.

DLTK configuration blocks

When it comes to presenting IDE preferences, the hardest part is building the UI. For example, the Octave syntax-coloring preference page not only allows the user to choose text colors and styles, it also provides an embedded text editor that shows how the styling actually looks. Creating these graphical controls from scratch can be time-consuming, but DLTK's `IPreferenceConfigurationBlocks` make your life much easier. [Figure 9](#) shows what the configuration block class hierarchy looks like.

Figure 9. Configuration block class hierarchy



The syntax-coloring configuration block

The first class of interest is `AbstractScriptEditorColoringConfigurationBlock`, which was created specifically for providing the user with syntax-coloring preferences. The `createSyntaxPage` method performs the main work, which is to create the graphical preference page. More specifically, it forms a `TreeViewer` that displays the different syntax types and a set of buttons that represent color and style preferences. Then, it calls `createPreviewer` to build the embedded text editor.

The Octave plug-in code contains a subclass of `AbstractScriptEditorColoringConfigurationBlock` called `OctaveSyntaxColorConfigurationBlock`. Besides the constructor, the methods listed in [Table 7](#) are implemented.

Table 7. The methods implemented by `OctaveSyntaxColorConfigurationBlock`

Method name	Description
<code>createPreviewViewer</code>	Returns a viewer to wrap around the embedded text editor
<code>createSimpleSourceViewerConfiguration</code>	Returns a basic object to configure the viewer in the embedded text editor

<code>setDocumentPartitioning</code>	Calls on an <code>IDocumentSetupParticipant</code> to set up the partitioning of the document
<code>getSyntaxColorListModel</code>	Returns a 2D array that identifies the different types of syntax that can be colored and the corresponding key in the preference store

The first three methods are straightforward, because the embedded text editor doesn't need a complicated viewer, viewer configuration object, or document setup participant. However, the last method, `getSyntaxColorListModel`, requires explanation. This method provides a 2D array whose elements correspond to the syntax types in the page's `TreeViewer`. Each element requires three pieces of information: a name, a category, and a corresponding key in the page's preference store.

Going back to the example, [Figure 10](#) shows that the Octave syntax color model consists of four different syntax types: keywords, numbers, strings, and single-line comments. The first three types fall under the `General` heading, while the last type falls under the `Comment` heading. The code in [Listing 11](#) shows how these types are configured in

`OctaveSyntaxColorConfigurationBlock.getSyntaxColorListModel`.

Listing 11. Creating the syntax color list model

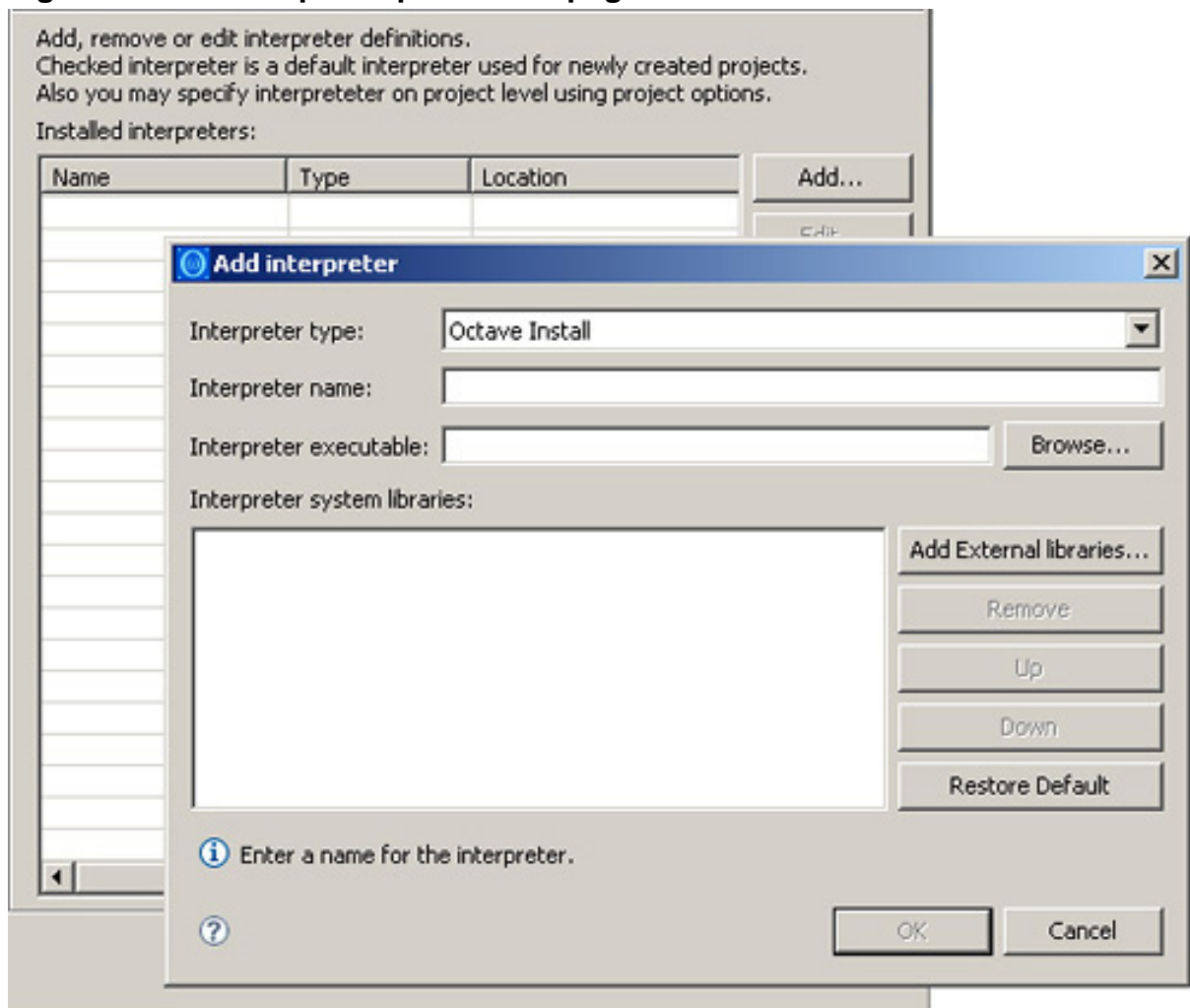
```
protected String[][] getSyntaxColorListModel() {
    return new String[][] {
        { "Single-Line Comments", DLTKColorConstants.DLTK_SINGLE_LINE_COMMENT,
          sCommentsCategory },
        { "Keywords", DLTKColorConstants.DLTK_KEYWORD, sCoreCategory },
        { "Strings", DLTKColorConstants.DLTK_STRING, sCoreCategory },
        { "Numbers", DLTKColorConstants.DLTK_NUMBER, sCoreCategory } };
}
```

When the user changes a preference setting in the page and clicks **OK**, the page updates the corresponding value in the preference store. The editor responds to the preference change by updating the presentation of its text.

Interpreter preferences

The last preference option in [Figure 10](#) provides settings for the IDE's script interpreter. DLTK provides two classes that make this possible:

`ScriptInterpreterPreferencePage` and `InterpretersBlock`. The `InterpretersBlock` creates the page's graphical control, which consists of a `Label` that provides instructions; a `CheckboxTableViewer` that lists available interpreters; and a series of buttons that allow the user to add, edit, copy, remove, or search for interpreters. [Figure 10](#) shows what this page looks like along with the window that appears when the user clicks **Add**.

Figure 10. The interpreter preference page and window

The abstract class `InterpretersBlock` requires that subclasses implement two methods: `getCurrentNature` and `createInterpreterDialog`. The first method returns a `String` that represents the interpreter, the source code language, and other associated objects. This nature is frequently identified in DLTk extension points, and it's important. If your Eclipse installation contains multiple DLTk-based features for different dynamic languages, you can easily distinguish them by their natures.

The second method, `createInterpreterDialog`, forms the window that appears when the user clicks **Add** on the preference page. This window, shown in [Figure 10](#), must furnish at least four pieces of information: the location of the interpreter executable, its name, its type, and the location of any libraries it requires to execute. To form this window, the `createInterpreterDialog` method requires an instance of the DLTk class `AddScriptInterpreterDialog`.

The `AddScriptInterpreterDialog` constructor accepts an array of interpreter

types and an object that implements `IInterpreterInstall`. Having multiple interpreter types allows the IDE to support multiple interpreters for the same language. The Octave IDE only supports one installation type, and it's identified in `plugin.xml` with an extension of `org.eclipse.dltk.launching.interpreterInstallTypes`. These preference extensions are shown in [Listing 12](#).

Listing 12. Octave preference extensions

```
<extension point="org.eclipse.dltk.launching.interpreterInstallTypes">
  <interpreterInstallType
    class="org.dworks.octaveide.launch.OctaveInterpreterInstallType"
    id="org.dworks.octaveide.launch.OctaveInterpreterInstallType">
  </interpreterInstallType>
</extension>
```

The `class` attribute identifies an object that implements `IInterpreterInstallType`. Each installation type is given a name, an ID, a nature (a `String` that identifies the interpreter and source type), and the locations of the interpreter and any required libraries. In addition, each `IInterpreterInstallType` must create one or more `IInterpreterInstall` objects.

Although an `IInterpreterInstallType` object defines a class of interpreter installations, an `IInterpreterInstall` object must be created for each particular installation. This object serves as the interpreter's primary data storage object, and it holds the information in [Table 8](#).

Table 8. The data stored by `IInterpreterInstall`

Field	Description
Name	A label for the interpreter installation
ID	A logical identifier for the interpreter installation
Nature	A logical identifier for the IDE (interpreter, source files, and so on)
Installation location	The path to the installation of the interpreter
Library locations	Paths to the interpreter's required libraries
Execution environment	Environment variables and process configuration
Interpreter arguments	Parameters directed to the interpreter

Finally, each `IInterpreterInstall` must provide access to an `IInterpreterRunner` object. This object is responsible for actually starting the interpreter, and the following discussion explains not only how it works but how it fits into the interpreter-IDE interaction as a whole.

Section 6. The DLTK interpreter and console

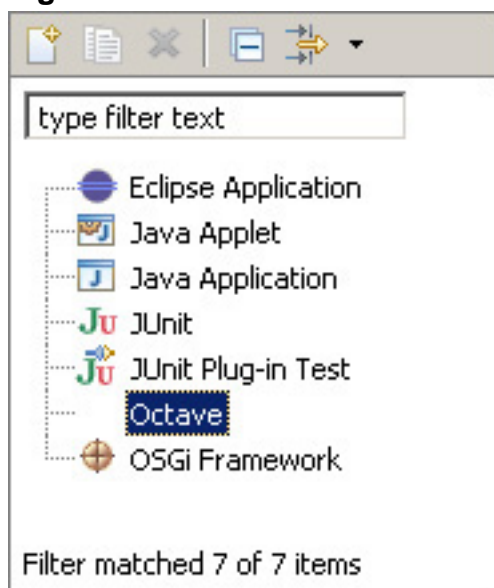
The most important feature of a DLTK application isn't the text editor or preferences but the ability to launch an interpreter with a button click and view its output in the IDE console. The DLTK interpreter and console are closely entwined, and managing them in code is a complicated process. Before delving into all the classes and interfaces, here's a high-level look at the events that take place when the user clicks **Run**:

1. The IDE responds to the button click by creating a launch object, collecting launch information, and adding a process to the launch.
2. The creation of the launch notifies all launch listeners, including the DLTK console manager.
3. The console manager acquires the launch information and uses it to create three objects: a console server, an interpreter object, and a console factory.
4. The console server creates a socket (by default, port 25000) and waits for incoming connections.
5. The console factory starts the interpreter and creates a console.
6. The console manager accesses the Eclipse Console plug-in and adds the new console.

DLTK classes handle much of this processing, but you still have to create many of your own. The Octave IDE needs to support Octave-related launches, then it needs Octave-specific classes to manage the console and interpreter. This discussion presents each of these subjects in turn.

Configure the DLTK launch

In Eclipse, the process of starting an application is called *launching the application in Run mode*. Because Eclipse supports many launching executables, you must make sure that the Octave interpreter launches when the user executes an Octave script. This association is made through *launch configuration types*, and you can see the available types in the **Run** window by clicking **Run > Run** in Eclipse. [Figure 11](#) shows what launch configuration types looks like.

Figure 11. The Octave launch configuration type

To create a launch configuration type for Octave, the first step is to add an extension of the `org.eclipse.debug.core.launchConfigurationTypes` to `plugin.xml`. In Figure 11, you can see the Octave configuration type selected in the window.

[Listing 13](#) presents the extension that defines this new type.

Listing 13. Defining the Octave launch configuration type

```
<extension point="org.eclipse.debug.core.launchConfigurationTypes">
  <launchConfigurationType
    id="org.dworks.octaveide.launch.OctaveLaunchConfigurationType"
    delegate="org.dworks.octaveide.launch.OctaveLaunchConfigurationDelegate"
    modes="run"
    public="true"
    name="%OctaveLaunchConfigurationType.name"
  </launchConfigurationType>
</extension>
```

The example project executes Octave scripts but doesn't debug them. Therefore, the `modes` attribute is set to `run` but not `run, debug`. The most important attribute is `delegate`, which identifies a class that implements `ILaunchConfigurationDelegate`. This interface defines a single method, `launch`, which is called when the user creates a launch configuration and clicks **Run**.

DLTK provides its own implementation of `ILaunchConfigurationDelegate` called `AbstractScriptLaunchConfigurationDelegate`. When this class's `launch` method is invoked, it calls the methods listed in [Table 9](#).

Table 9. The methods called during `AbstractScriptLaunchConfigurationDelegate.launch()`

Method	Description
<code>createInterpreterConfig</code>	This method constructs an <code>InterpreterConfig</code> object that receives all the information from the launch configuration.
<code>getInterpreterRunner</code>	This method accesses the <code>IInterpreterInstall</code> object created by the interpreter installation type and uses it to form an <code>IInterpreterRunner</code> .
<code>runRunner</code>	This method creates a process for the interpreter and adds it to the launch.

This process may sound complicated, but the first two methods simply organize information about the launch; the `InterpreterConfig` holds information taken from the launch configuration, and the `IInterpreterInstall` stores information taken from the preferences. It's important to understand that the third method does *not* actually start the interpreter. It creates an `IProcess` for the interpreter and adds it to the `Launch` object.

The DLTk console manager

Octclipse and Ryan Rusaw

We want to express our gratitude to Ryan Rusaw, who created the Octclipse development tool. It was his development that inspired the example project used in this tutorial. Not only does it provide syntax coloring, preferences, and interpreter integration, it also offers many additional features, including code parsing, debugging, semantic highlighting, and project-creation wizards. If you're interested in using the Octave interpreter, we strongly recommend that you download Ryan's Octclipse toolset (see [Resources](#)), released under the Eclipse Public License.

When the DLTk plug-in `org.eclipse.dltk.console.ui` starts, it accesses the Eclipse Debug UI plug-in and adds an instance of `ScriptConsoleManager` as a launch listener. These listeners are notified whenever a launch is added, changed, or removed. The `ScriptConsoleManager` only responds to added launches, and its first task is to check whether the launch has a recognizable script nature. If so, the manager creates three important objects: a console server, an interpreter object, and a console factory.

The manager creates a `ScriptConsoleServer` to manage communication between the interpreter and the console. During its initialization, the server binds a `ServerSocket` to port 25000 and tells it to wait for incoming connection requests. These requests are embodied by `ConsoleRequest` objects, and as the server receives new requests, it calls the `consoleConnected` method of each.

After creating the server, the `ScriptConsoleManager` accesses a `ConsoleRequest` representing the interpreter. It does this by searching for an extension of the point `org.eclipse.dltk.console.scriptInterpreter` in `plugin.xml`. This extension point identifies a nature designation and a class that implements `IScriptInterpreter`, a subinterface of `ConsoleRequest`. After it obtains the server and interpreter object, the console manager can start creating the console. To do this, it calls upon a console factory.

The console factory and the console

Ordinarily, if you want to access an Eclipse console from a plug-in, you need only extend one extension point: `org.eclipse.ui.console.consoleFactories`. This extension must identify a class that implements `IConsoleFactory`. Then, when the user clicks **Open Console** in the Console view, Eclipse calls the class's `openConsole` method to create a new console.

DLTK makes things slightly more complicated. The DLTK `ScriptConsoleManager` also needs to access a console factory, but it requires a class that implements `IScriptConsoleFactory`. Conveniently, `IConsoleFactory` and `IScriptConsoleFactory` both require the same `openConsole` method. However, you need to create two extension points: one for the Eclipse console factory and one for the DLTK console factory. [Listing 14](#) shows what these two points look like for the Octave IDE.

Listing 14. Identifying the Octave console factory

```
<extension point="org.eclipse.dltk.console.ui.scriptConsole">
  <scriptConsole
    class="org.dworks.octaveide.launch.OctaveConsoleFactory"
    natureID="org.dworks.octaveide.nature" />
</extension>

<extension point="org.eclipse.ui.console.consoleFactories">
  <consoleFactory
    class="org.dworks.octaveide.launch.OctaveConsoleFactory"
    icon="icons/oct.gif"
    label="%OctaveConsole.Console" />
</extension>
```

When the `ScriptConsoleManager` accesses a script console factory, it calls the factory's `openConsole` method, which creates the all-important `ScriptConsole` object. Like the Eclipse text editor, this console displays text using a document-based methodology; it has a `ConsoleDocument`, an `IConsoleDocumentPartitioner`, and arrays of `Positions` and `Regions`.

When the `ScriptConsole` is created, it calls `setInterpreter` with a reference to the `IScriptInterpreter` object that the console manager creates. This method creates a connection between the console and the interpreter in the form of an

`InitialStreamReader`. This stream reader attaches itself to the interpreter by calling `IScriptInterpreter.getInitialOutputStream`. Next, the `InitialStreamReader` reads each line of the interpreter's output and sends it to the console's display. When it's done displaying text, the console prints a prompt and places itself in editable mode.

After creating the `ScriptConsole`, the `ScriptConsoleManager` accesses the `Eclipse ConsoleManager` and adds the new object to the list of available consoles. Now the new console will be visible to the user and accessible by the interpreter.

Run the interpreter

Let's take a closer look at the `IScriptInterpreter` object created by the `ScriptConsoleManager`. This interface extends the `ConsoleRequest` interface, which means that it can be delivered to the `ScriptConsoleServer` for processing. The console manager handles this delivery, and when the server receives the interpreter's request, it calls

```
IScriptInterpreter.consoleConnected(IScriptConsoleIO protocol).
```

This method serves two important roles: It constructs the `InitialStreamReader` that enables data transfer between the console and the interpreter, and the method's argument identifies the protocol to be used for communication. The `IScriptConsoleIO` interface defines two important methods that receive data from the interpreter:

- **`InterpreterResponse execInterpreter(String command)`.**
Sends a script command to the interpreter, then receives a response
- **`ShellResponse execShell(String command, String[] args)`.**
Sends a shell command to the interpreter, then receives a response

These two functions direct commands to the interpreter and receive its output. Each `IScriptConsoleIO` protocol object is initialized with an `InputStream` and an `OutputStream`. When `execInterpreter` is called, the protocol sends the command to the interpreter to its `OutputStream`. When the interpreter responds, the protocol receives the `InterpreterResponse` (essentially, a `String`) and directs it to the console through the `InputStream`.

All communication to the interpreter relies on the protocol's methods. For example, the `IScriptInterpreter` interface contains the `exec(String command)` that sends commands to the interpreter. This command calls on `IScriptConsoleIO.execInterpreter` to deliver the command. Similarly, the interpreter's `close` method relies on `IScriptConsoleIO.execShell`.

Section 7. Summary

As this tutorial has shown, the DLTK provides myriad features for creating development environments for dynamic languages. Creating these environments doesn't require a lot of code, but you have to know exactly what you're doing. To configure text presentation, you must understand how the editor, viewer, and document interact as well as the rudiments of rule-based partitioning and scanning. To enable user preferences, you have to grasp the strange class hierarchy of Eclipse preference pages and DLTK configuration blocks. To enable launching of dynamic-language projects, you have to deal with threads, streams, and the protocol methods needed to communicate between the console and script interpreter.

The DLTK learning curve is steep and continues upward for a significant distance. But if you need to build a full-featured IDE for your customized script language, you'll find no better toolset. In addition to the material presented in this tutorial, DLTK provides many sample IDEs for languages like PHP and Tcl.

Downloads

Description	Name	Size	Download method
DLTK-based plug-in for Octave	os-eclipse-octave-example_plugin.zip	47 KB	HTTP

[Information about download methods](#)

Resources

Learn

- Explore the Eclipse [Dynamic Languages Toolkit](#).
- Visit the [GNU Octave site](#) for more information about this tool.
- Check out the [Recommended Eclipse reading list](#).
- Browse all the [Eclipse content](#) on developerWorks.
- New to Eclipse? Read the developerWorks article [Get started with the Eclipse Platform](#) (Chris Aniszczyk and David Gallardo, July 2007) to learn its origin and architecture and how to extend Eclipse with plug-ins.
- Expand your Eclipse skills by checking out IBM developerWorks' [Eclipse project resources](#).
- To listen to interesting interviews and discussions for software developers, check out [developerWorks podcasts](#).
- Stay current with developerWorks' [technical events and webcasts](#).
- Watch and learn about IBM and open source technologies and product functions with the no-cost [developerWorks on demand demos](#).
- Check out upcoming conferences, trade shows, webcasts, and other [events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Check out Ryan Rusaw's [Octclipse](#) toolset.
- Download the [Eclipse Platform and other projects](#) from the Eclipse Foundation.
- Download [IBM product evaluation versions](#), and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- The [Eclipse Platform newsgroups](#) should be your first stop for discussing questions regarding Eclipse. (Clicking this link launches your default Usenet news reader application and opens eclipse.platform.)

- The [Eclipse newsgroups](#) have many resources for people interested in using and extending Eclipse.
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

About the authors

Matthew Scarpino

Matthew Scarpino is a project manager and Java developer at Eclipse Engineering LLC. He is the lead author of *SWT/JFace in Action* and made a minor but important contribution to the Standard Widget Toolkit (SWT). He enjoys Irish folk music, marathon running, the poetry of William Blake, and the Graphical Editing Framework (GEF).

Nathan A. Good



Nathan A. Good lives in the Twin Cities area of Minnesota. Professionally, he does software development, software architecture, and systems administration. When he's not writing software, he enjoys building PCs and servers, reading about and working with new technologies, and trying to get his friends to make the move to open source software. He's written and co-written many books and articles, including *Professional Red Hat Enterprise Linux 3*, *Regular Expression Recipes: A Problem-Solution Approach*, and *Foundations of PEAR: Rapid PHP Development*.