Building cool scripts, apps, and games
for Android Smartphones

Practical Android Projects

Lucas Jordan | Pieter Greyling

Apress®

Chapter 5

Introducing SL4A: The Scripting Layer for Android

The main objective of this chapter is to introduce you to the Scripting Layer for Android (SL4A) platform. Our aim is to give you enough basic understanding of how SL4A[1] works and to be able to use it to run your own scripts written in a number of high-level scripting languages.

As you will see, the design of SL4A enables it to support many scripting language interpreters. In order to make practical use of SL4A, you will need to understand at least the rudiments of one high-level scripting language such as Python, Ruby, Perl, Lua, JavaScript, or BeanShell.

For this book, we assume that you have a good level of Java programming language knowledge and this should serve you well if you intend to use a related language such as BeanShell or JavaScript with SL4A.

In this chapter, we will first give you some background about what SL4A is, what it can be used for, where to get it, and where to learn more about it for yourself. We will then show you how to install and run SL4A with small examples. After this, you will get a technical overview of how SL4A works and how its design relates to the scripting architectures we presented in the two previous chapters. To help you on your way to a deeper study of SL4A, we will then show you how to obtain a copy of the complete SL4A source code repository. In conclusion, we will present some equivalent SL4A "Hello World" example code snippets in various scripting languages.

[1] From now on, we will use the acronym "SL4A" instead of "Scripting Layer for Android."

What Is Scripting Layer for Android?

In a nutshell, SL4A is an infrastructure for enabling the interoperation of scripting language engines that have been ported to the Android platform with the Android application programming interface (API) via remote procedure calls (RPCs) to a server implemented as a standard Android Java application.

About SL4A

SL4A,[2] originally called Android Scripting Environment (ASE), was brought to us by Damon Kohler[3] and is hosted on Google Code as an open-source project.

From the user perspective, the SL4A Android application lets you edit and run scripts against multiple interactive script interpreters on your Android device. It also supports the ability to install script interpreters into the application directly from the SL4A home site.

In essence, SL4A is more than just a standard end-user Android application; it is also a platform for exposing Android functionality to custom client programs such as scripting engines.

The SL4A License

Like the Android platform, SL4A is open source and is released under the same Apache License Version 2.0,[4] as is most of Android.

Using SL4A

The SL4A system is suited for the following kinds of tasks:

- **RAD programming:** With SL4A it is possible to use a rapid application development (RAD) approach to quickly create a prototype application that allows you to test the feasibility of an idea. Once the practicality of the application is confirmed, you can create a full-blown Android application.
- **Writing test scripts:** Assuming that the supporting Android APIs are exposed to SL4A, it can be used to create test scripts for other functionality.

---

[2] http://code.google.com/p/android-scripting/

[3] http://www.damonkohler.com/search/label/sl4a

[4] http://www.apache.org/licenses/

- **Building utilities:** You can fairly easily and quickly write utility and tool scripts that do small jobs or automate certain aspects of repetitive tasks. These tools probably do not require a complicated user interface; they need just a simple dialog-based user interaction mode.

SL4A Resources

For more background on SL4A, we recommend consulting the following online resources:

- SL4A home:
  - http://code.google.com/p/android-scripting/
- SL4A downloads:
  - http://code.google.com/p/android-scripting/downloads/list
  - http://code.google.com/p/android-scripting/downloads/list?q=label:Featured
- SL4A FAQ:
  - http://code.google.com/p/android-scripting/wiki/FAQ
- SL4A Wiki:
  - http://code.google.com/p/android-scripting/w/list
- SL4A tutorials:
  - http://code.google.com/p/android-scripting/wiki/Tutorials
- SL4A source code:
  - http://code.google.com/p/android-scripting/source/checkout

The SL4A Code Repository

The SL4A project source code is hosted on Google Code in a Mercurial repository.[5]

Mercurial SCM is an open-source, distributed-source, control management tool. Mercurial is designed to be cross-platform and is written in the Python programming language.

[5] http://code.google.com/p/android-scripting/source/browse/

Later in the chapter, we will show you how to use the Mercurial client-side tools and integrated development environment (IDE) plugins to bring the SL4A code to your desktop in order to build and run SL4A.

Running SL4A in the Android Emulator

Before we dive into the details of SL4A, we will show you the quickest way to have a look at SL4A in action.

We will download the SL4A Android application package (APK) distribution and install it on a running Android emulator instance. This will allow us to use SL4A to install some scripting engines into the emulator from the SL4A site.

Development Environment Configuration

As a quick reference, Table 5–1 lists some development environment configuration settings and commands that you will find useful for working with the code in this chapter.

Table 5–1. *Development Environment Configuration Quick Reference*

Item	Value or Command
PATH	<Android SDK Directory>/tools
PATH	<Android SDK Directory>/platform-tools
PATH	<Apache Ant Directory>/bin
PATH	<Mercurial Directory>/
Create an AVD	android create avd -n android23api9_hvga_32mb -t android-9 -c 32M
Start the AVD on Linux / Mac OS X	emulator -avd android23api9_hvga_32mb &
Start the AVD on Windows	start emulator -avd android23api9_hvga_32mb

Even though the PATH entry for the Mercurial executable (hg or hg.exe) is not strictly necessary right now, it will become useful if you decide to get a snapshot of the SL4A code and build it yourself.

Downloading the SL4A APKs

First, we will download the SL4A application archive from the SL4A download site here:

- http://code.google.com/p/android-scripting/downloads/list

Or here:

- http://code.google.com/p/android-scripting/downloads
/list?q=label:Featured

Download the latest release of the SL4A APK. At the time of writing, this was the following:

```
sl4a_r3.apk
```

Save this to a directory on your file system. We have called ours `sl4a-apk`.

Also available on the SL4A site were the following scripting engine APKs:

```
beanshell_for_android_r1.apk
jruby_for_android_r1.apk
lua_for_android_r1.apk
perl_for_android_r1.apk
python_for_android_r1.apk
rhino_for_android_r1.apk
```

NOTE: The SL4A APKs are included in the book downloads, so you can get started right away.

Installing the SL4A APK on the Android Emulator

Ensure that you are running the Android emulator using a compatible AVD. For this, follow the instructions in Table 5–1.

Now, in a terminal command-line shell, navigate to the directory where you have saved the downloaded SL4A APK file and enter the following command:

```
adb install sl4a_r3.apk
```

This should produce output similar to the following:

```
adb install sl4a_r3.apk
364 KB/s (840827 bytes in 2.250s)
    pkg: /data/local/tmp/sl4a_r3.apk
Success
```

The SL4A launcher icon should now appear on the emulator, as illustrated in Figure 5–1.

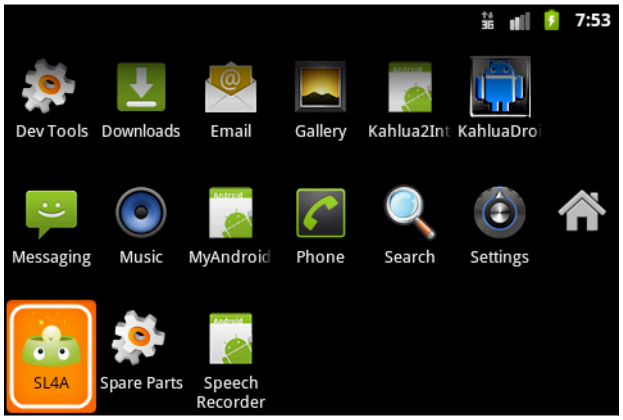
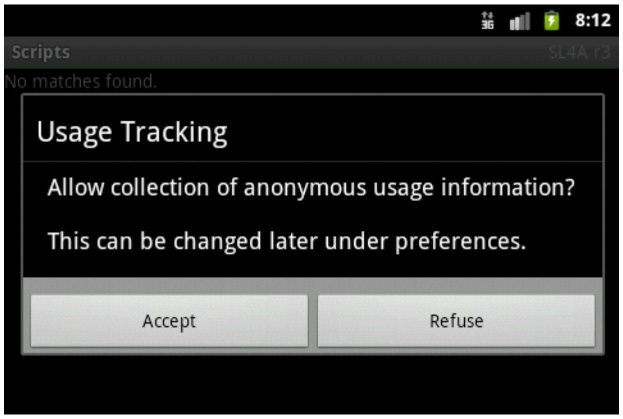


Figure 5–1. *SL4A Application Launcher icon*

Running SL4A on the Android Emulator

When we launch the application, you should see the SL4A primary Usage Tracking request, as shown in Figure 5–2.



No matches found.

Usage Tracking

Allow collection of anonymous usage information?

This can be changed later under preferences.

Figure 5–2. *SL4A Usage Tracking*

Simply click one of either the Accept or Refuse buttons. After invoking the SL4A application menu (press the F2 keyboard key while in the emulator), you should now see the image of the main SL4A application Activity screen, as shown in Figure 5–3.

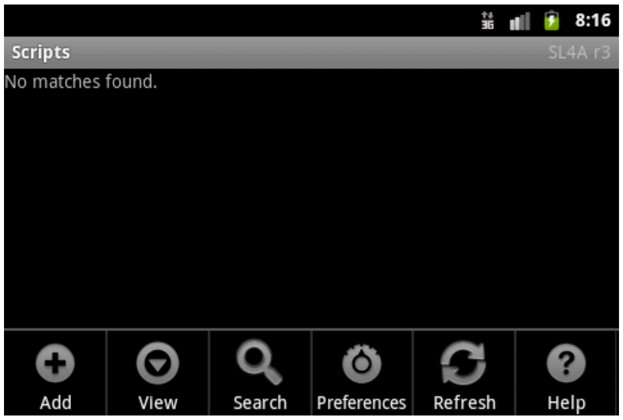


Figure 5–3. *SL4A application initial screen with Menu*

Now select the View menu item and choose the Interpreters entry, as shown in Figure 5–4.

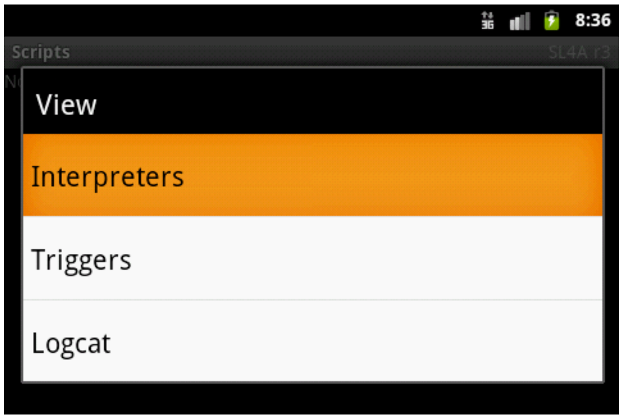


Figure 5–4. *SL4A view interpreters*

As illustrated in Figure 5–5, our SL4A emulator installation is currently equipped only with the Shell interpreter.

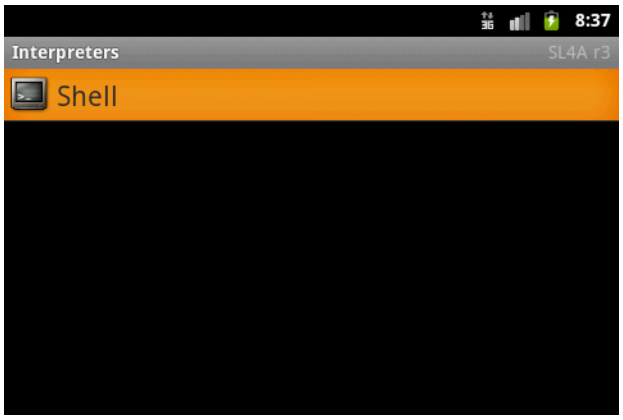
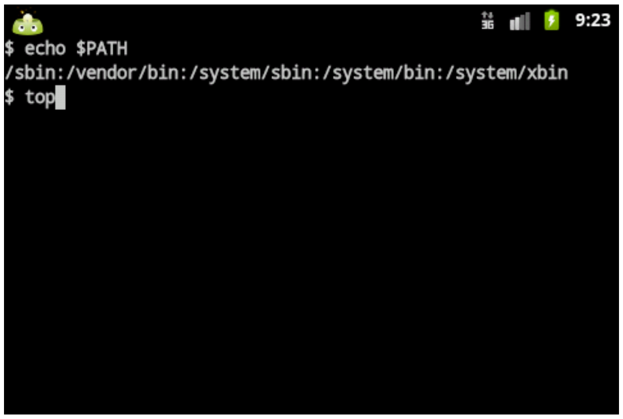


Figure 5–5. *SL4A interpreters: Shell*

If you run the Shell interpreter, you should be presented with Figure 5–6. As you can see, we have entered the following command into the shell:

```
echo $PATH
```

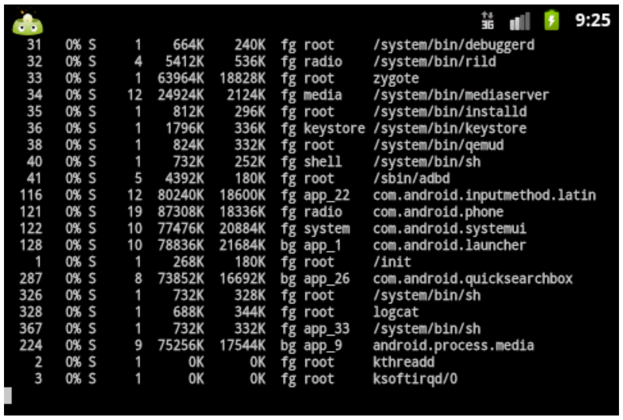
We are also about to execute the UNIX top command.



```
$ echo $PATH
/sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
$ top
```

Figure 5–6. *SL4A interpreters: Shell commands*

The result of running the top command is shown in Figure 5–7.

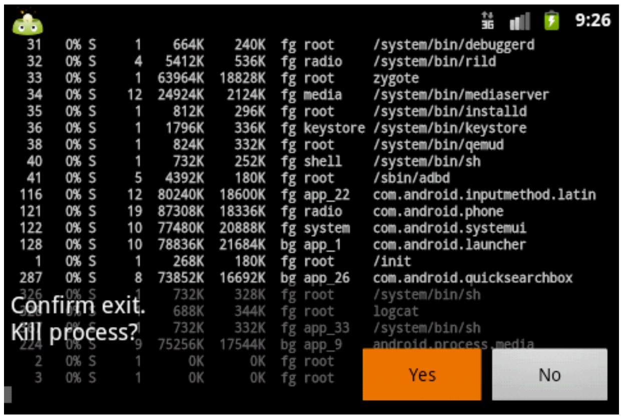


```
                                                     9:25
   31   0% S     1    664K    240K  fg root     /system/bin/debuggerd
   32   0% S     4   5412K    536K  fg radio    /system/bin/rild
   33   0% S     1  63964K  18828K  fg root     zygote
   34   0% S    12  24924K   2124K  fg media    /system/bin/mediaserver
   35   0% S     1    812K    296K  fg root     /system/bin/installd
   36   0% S     1   1796K    336K  fg keystore /system/bin/keystore
   38   0% S     1    824K    332K  fg root     /system/bin/qemud
   40   0% S     1    732K    252K  fg shell    /system/bin/sh
   41   0% S     5   4392K    180K  fg root     /sbin/adbd
  116   0% S    12  80240K  18600K  fg app_22   com.android.inputmethod.latin
  121   0% S    19  87308K  18336K  fg radio    com.android.phone
  122   0% S    10  77476K  20884K  fg system   com.android.systemui
  128   0% S    10  78836K  21684K  bg app_1    com.android.launcher
    1   0% S     1    268K    180K  fg root     /init
  287   0% S     8  73852K  16692K  bg app_26   com.android.quicksearchbox
  326   0% S     1    732K    328K  fg root     /system/bin/sh
  328   0% S     1    688K    344K  fg root     logcat
  367   0% S     1    732K    332K  fg app_33   /system/bin/sh
  224   0% S     9  75256K  17544K  bg app_9    android.process.media
    2   0% S     1      0K      0K  fg root     kthreadd
    3   0% S     1      0K      0K  fg root     ksoftirqd/0
```

Figure 5–7. *SL4A Interpreters: Shell top*

When you exit the Shell interpreter using the application menu, you are presented with the screen as depicted in Figure 5–8.

Click the Yes button.



```
                                                     9:26
   31   0% S     1    664K    240K  fg root     /system/bin/debuggerd
   32   0% S     4   5412K    536K  fg radio    /system/bin/rild
   33   0% S     1  63964K  18828K  fg root     zygote
   34   0% S    12  24924K   2124K  fg media    /system/bin/mediaserver
   35   0% S     1    812K    296K  fg root     /system/bin/installd
   36   0% S     1   1796K    336K  fg keystore /system/bin/keystore
   38   0% S     1    824K    332K  fg root     /system/bin/qemud
   40   0% S     1    732K    252K  fg shell    /system/bin/sh
   41   0% S     5   4392K    180K  fg root     /sbin/adbd
  116   0% S    12  80240K  18600K  fg app_22   com.android.inputmethod.latin
  121   0% S    19  87308K  18336K  fg radio    com.android.phone
  122   0% S    10  77480K  20888K  fg system   com.android.systemui
  128   0% S    10  78836K  21684K  bg app_1    com.android.launcher
    1   0% S     1    268K    180K  fg root     /init
  287   0% S     8  73852K  16692K  bg app_26   com.android.quicksearchbox
  326   0% S          732K    328K  fg root     /system/bin/sh
                      688K    344K  fg root     logcat
                1     732K    332K  fg app_33   /system/bin/sh
  224   0% S     9  75256K  17544K  bg app_9    android.process.media
    2   0% S     1      0K      0K  fg root
    3   0% S     1      0K      0K  fg root
```

Confirm exit.
Kill process?

Figure 5–8. *SL4A interpreters: Shell exit*

We are now done with confirming that our SL4A emulator installation is functioning correctly.

Installing SL4A Interpreters

Next we will add some new scripting language interpreters to the SL4A toolbox.

We will first show you how to do so using the SL4A application and then summarize how to install the interpreters from your computer using the downloadable[6] application packages (APKs). This will enable you to install the base interpreter engines slightly faster than doing it from the device using SL4A. It is best thereafter to let the relevant script engine fetch its current and compatible extra ZIP packages under its own control.

Adding Interpreters with SL4A

From within the `Interpreters` screen, select the `Add` menu item. After scrolling down to the Python entry in the list, you should see the image depicted in Figure 5–9.

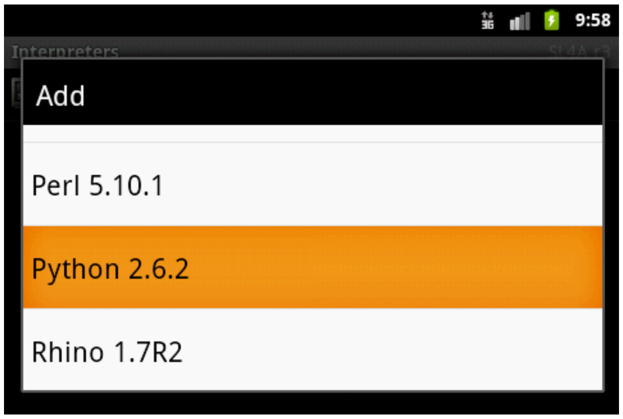
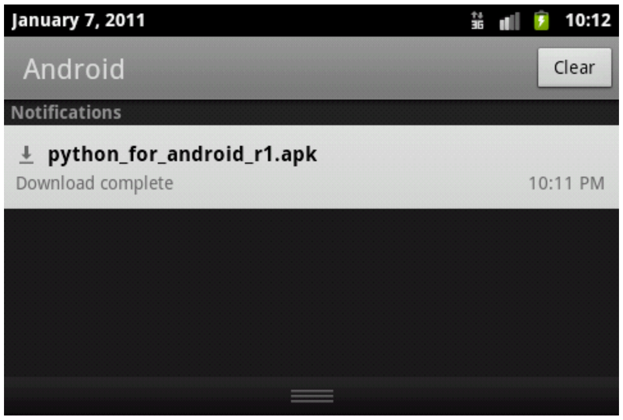


Figure 5–9. *SL4A Add interpreters*

Select the Python entry.

This will start a download of the Python APK from the SL4A site that should show the notification shown in Figure 5–10 when completed.

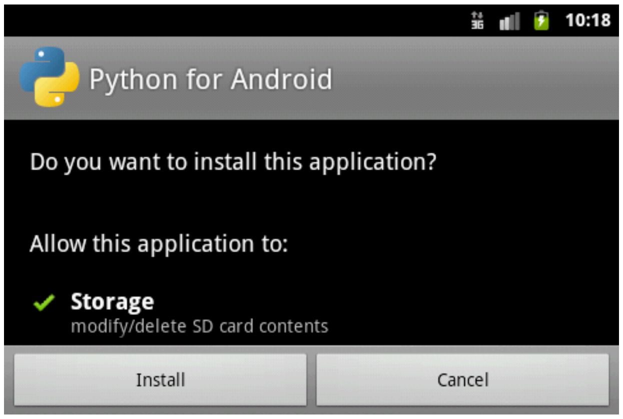
[6] http://code.google.com/p/android-scripting/downloads/list



python_for_android_r1.apk
Download complete 10:11 PM

Figure 5–10. *SL4A Add interpreters: Python downloaded*

Clicking the notification entry should display the dialog shown in Figure 5–11.



Do you want to install this application?

Allow this application to:

Storage
modify/delete SD card contents

Figure 5–11. *SL4A Add interpreters: Python install*

Now click the Install button.

The image shown in Figure 5–12 is what you should see during the download and installation process.

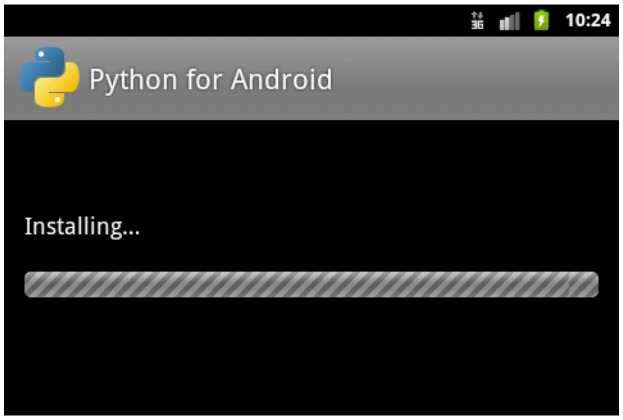


Figure 5–12. *SL4A Add interpreters: Python installing*

Once the installation is complete, you should see something similar to that shown in Figure 5–13.

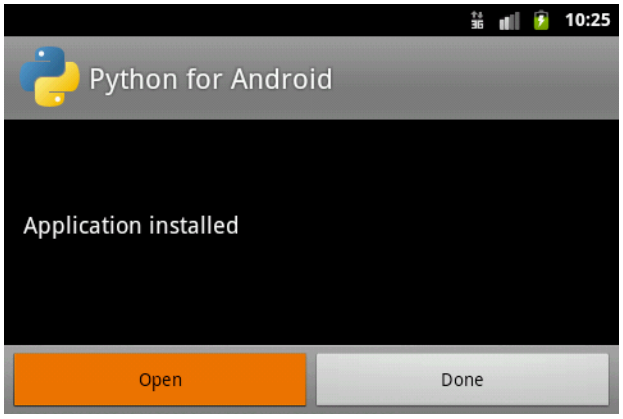


Figure 5–13. *SL4A Add interpreters: Python installed*

If you now click the `Open` button, you will be presented with a screen (see Figure 5–14) to initiate the secondary installation sequence. This will download all the supporting archives to the emulator instance.

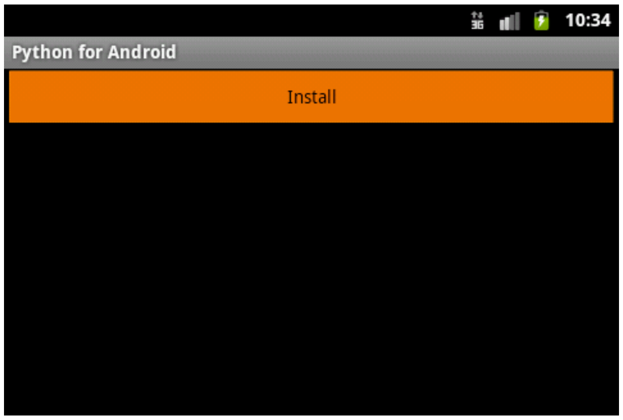


Figure 5–14. *SL4A Add interpreters: Install Python supporting files*

Figure 5–15 depicts the process of downloading the suite of Python interpreter files.

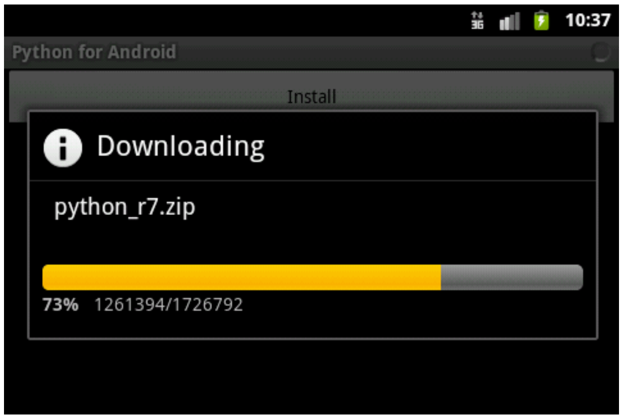


Figure 5–15. *SL4A Add interpreters: Install Python download supporting files*

Once the whole installation download process is complete, you should see the image illustrated in Figure 5–16.

Do not click the `Uninstall` button.

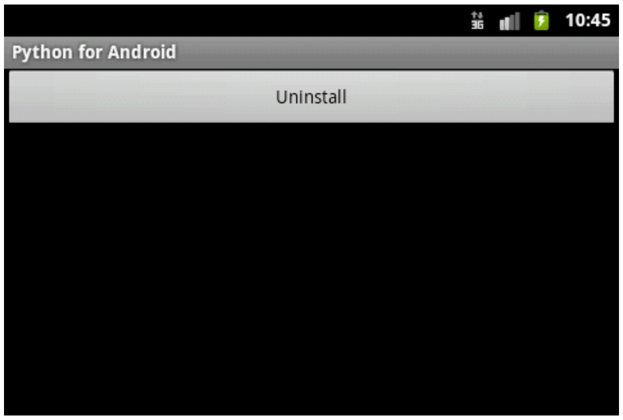


Figure 5–16. *SL4A Add interpreters: Install Python done*

Going back to the SL4A interpreters list, you should now see the Python script language entry, as shown in Figure 5–17.

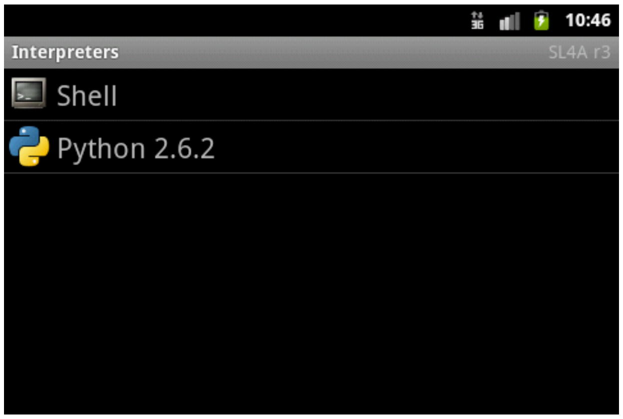
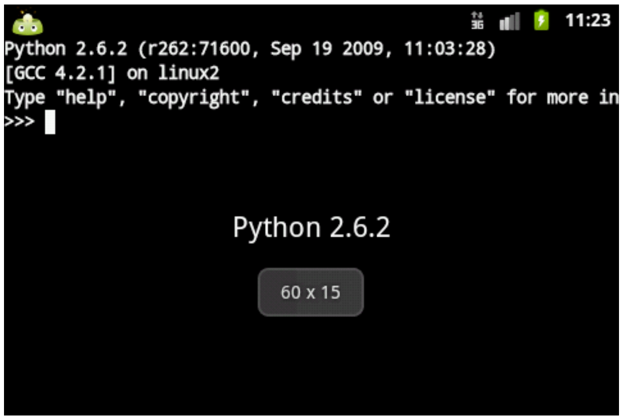


Figure 5–17. *SL4A Add interpreters: Python complete*

This means that you can run Python on Android! The result of clicking the Python interpreter entry is shown in Figure 5–18.



```
11:23
Python 2.6.2 (r262:71600, Sep 19 2009, 11:03:28)
[GCC 4.2.1] on linux2
Type "help", "copyright", "credits" or "license" for more in
>>> 
```

Figure 5–18. *SL4A: Run Python*

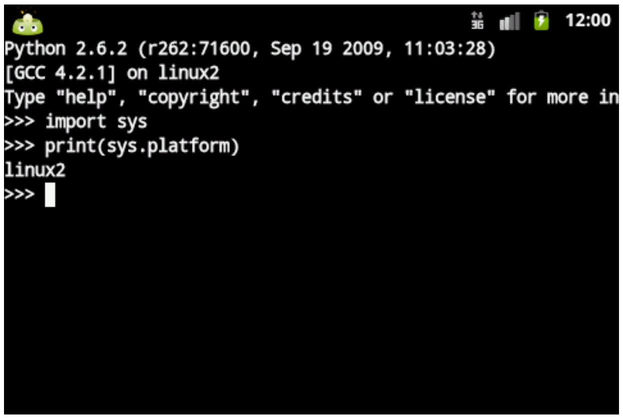
Let's enter some code into the interpreter. Try running the following Python statements:

```
import sys
print(sys.platform)
```

The result should be the following:

```
linux2
```

This short session is also shown in Figure 5–19.



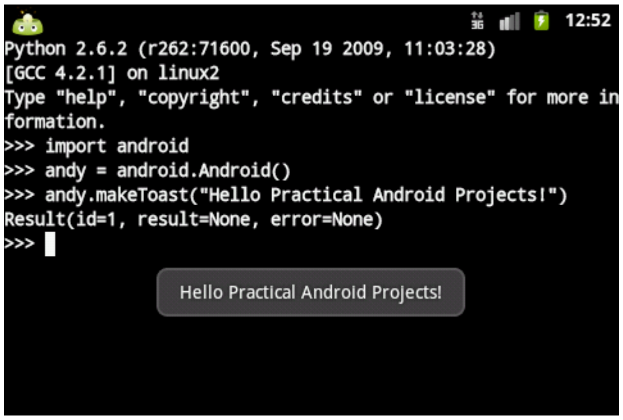
```
12:00
Python 2.6.2 (r262:71600, Sep 19 2009, 11:03:28)
[GCC 4.2.1] on linux2
Type "help", "copyright", "credits" or "license" for more in
>>> import sys
>>> print(sys.platform)
linux2
>>> 
```

Figure 5–19. *SL4A - Run Python Code*

To demonstrate interaction with the Android API, enter the following code into the interactive interpreter:

```
import android
andy = android.Android()
andy.makeToast('Hello Practical Android Projects!')
```

The result is shown in Figure 5–20.



```
Python 2.6.2 (r262:71600, Sep 19 2009, 11:03:28)
[GCC 4.2.1] on linux2
Type "help", "copyright", "credits" or "license" for more in
formation.
>>> import android
>>> andy = android.Android()
>>> andy.makeToast("Hello Practical Android Projects!")
Result(id=1, result=None, error=None)
>>>
```

Figure 5–20. *SL4A: Run Python Hello Android*

This concludes our demonstration of how to install SL4A interpreters using SL4A.

NOTE: The SL4A interpreter setup process that we have documented so far will also apply when using a physical device. So when you install SL4A and the supported interpreters on your phone, the same steps will be presented.

Adding Interpreters with Package Archives

As mentioned earlier, it is also possible to install the interpreters from your computer using the downloadable[7] application packages (APKs). This makes script interpreter installation slightly faster than only using the SL4A application as explained before.

Let's assume that you want to install the Perl scripting interpreter in this fashion. In a terminal command-line shell, navigate to the directory where you have saved the downloaded Perl APK file and enter the following command:

```
adb install perl_for_android_r1.apk
```

This should produce output similar to the following:

```
adb install perl_for_android_r1.apk
96 KB/s (33894 bytes in 0.343s)
      pkg: /data/local/tmp/perl_for_android_r1.apk
Success
```

As shown earlier in Figure 5–14 where we installed Python, you can now initiate the secondary installation sequence using the Perl for Android Launcher icon that should now be visible in the emulator or device. This will download all the supporting archives

[7] http://code.google.com/p/android-scripting/downloads/list

following on from Figure 5–14 with the same workflow as before. At the very least, it saves having the interpreter package being downloaded.

Understanding Scripting Layer for Android

As mentioned earlier, SL4A enables interoperation between Android agnostic scripting language engines and the Android API. As you will see later, this is not restricted to scripting languages only. Any program that implements a compatible JSON–based RPC[8] interfacing module or set of routines can potentially invoke the SL4A RPC Server.

Communicating Using JavaScript Object Notation (JSON)

Internally, SL4A uses the JavaScript Object Notation (JSON) data format for the interchange of messages and data between the SL4A RPC Server and its clients. This is fundamental to its workings so we will give a quick summary of JSON here.

The acronym JSON was originally specified by Douglas Crockford. The JSON data format is described in RFC 4627.[9]

To quote from the JSON specification:

> *"JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data. JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays)."*

To give you an idea of the data format, the specification also presents an example of a JSON object as follows:

```
{
  "Image": {
    "Width":  800,
    "Height": 600,
    "Title":  "View from 15th Floor",
    "Thumbnail": {
      "Url":    "http://www.example.com/image/481989943",
      "Height": 125,
      "Width":  "100"
    },
    "IDs": [116, 943, 234, 38793]
```

[8] Remote Procedure Call

[9] http://tools.ietf.org/html/rfc4627

```
  }
}
```

We will not go into more detail about JSON here but instead recommend that you follow up the online resources for more information if you are interested.

Summarizing the SL4A Architecture

SL4A exposes Android API functionality to its clients. It achieves this by implementing a scripting language–compatible module that marshals RPCs and their responses to and from a RPC server implemented as an Android Java application. This enables the RPC server to have direct access to the Android API and it behaves as a remote proxy using a façade that encapsulates and exposes selected Android APIs.

Scripting languages are ported, via cross-compilation or otherwise, to the Android platform in their purest form avoiding any source code changes. This implies that the scripting language has no knowledge of the Android platform at all. It gains access to the Android API using a special module generally implemented in the scripting language itself that accesses the Android API over the remote SL4A RPC server. The current implementation uses the JSON data format for its application layer network messaging package payload content.

The overall design of this infrastructure is illustrated in Figure 5–21.

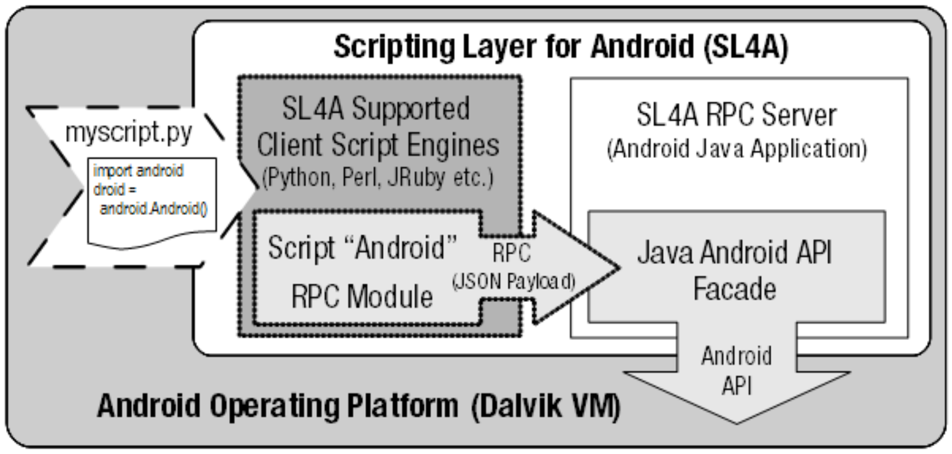


Figure 5–21. *SL4A architecture overview*

To explain things another way, a "generic" script engine, running in a self-contained process, accesses the Android API over bidirectional JSON-RPC via an API façade. The façade is serviced in a separate process, and implemented as a "standard," Java-based, Android server application. The latter has full access to the Android platform API and thus essentially serves as a remote API "proxy" for the scripting engine/interpreter. Each scripting language has a "wrapper" module providing an "Android object" that serves as an in-process, local, API proxy, with the task of packaging RPC calls as scripting methods in the spirit of the particular scripting engine.

To sum this up, an overview of the full-round trip invocation stack looks something like the following:

```
-- Script Interpreter
---- Client/Caller/Consumer Script
------ "Android" Script Object (locally wraps RPC calls) - Local Proxy
-------- Remote Procedure Calls (RPC) – Exchanges contain a JSON payload
------ Android API Java Facade - Remote Proxy
---- API Server/Provider - Android Java application
-- The Android Platform itself
```

NOTE: This decoupled architecture permits any compatible local or remote client to call into SL4A as long as it does so via the JSON RPC call interface.

Reviewing Local Proxy Implementations

We will not go into detailed explanations of the code, but to give you an idea of what is involved, here are some implementations of client RPC proxy wrapper modules.

In Python (see Listing 5–1):

Listing 5–1. *Python Module for Accessing the AndroidProxy (android-scripting/python/ase/android.py)*

```
# Copyright (C) 2009 Google Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License"); you may not
# use this file except in compliance with the License. You may obtain a copy of
# the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
# WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
# License for the specific language governing permissions and limitations under
# the License.

__author__ = 'Damon Kohler <damonkohler@gmail.com>'

import collections
import json
import os
import socket
import sys

PORT = os.environ.get('AP_PORT')
HOST = os.environ.get('AP_HOST')
HANDSHAKE = os.environ.get('AP_HANDSHAKE')
```

```
Result = collections.namedtuple('Result', 'id,result,error')


class Android(object):

  def __init__(self, addr=None):
    if addr is None:
      addr = HOST, PORT
    self.conn = socket.create_connection(addr)
    self.client = self.conn.makefile()
    self.id = 0
    if HANDSHAKE is not None:
      self._authenticate(HANDSHAKE)

  def _rpc(self, method, *args):
    data = {'id': self.id,
            'method': method,
            'params': args}
    request = json.dumps(data)
    self.client.write(request+'\n')
    self.client.flush()
    response = self.client.readline()
    self.id += 1
    result = json.loads(response)
    if result['error'] is not None:
      print result['error']
    # namedtuple doesn't work with unicode keys.
    return Result(id=result['id'], result=result['result'],
               error=result['error'], )

  def __getattr__(self, name):
    def rpc_call(*args):
      return self._rpc(name, *args)
    return rpc_call
```

The equivalent functionality is shown in the C programming language (see Listing 5–2) taken from the android-cruft project[10] on Google Code. This project demonstrates how to write C programs that can access the Android API by leveraging SL4A, using the same principles as the earlier Python proxy.

Listing 5–2 presents the C main function that is the meat of the code logic. It attempts to invoke the Android API over the SL4A RPC server in order to raise an Android Toast with the message “w00t!” It marshals the message character buffer into a json_array, which it then sends over to the SL4A RPC server with a call to the sl4a_rpc function.

Listing 5–2. ***C Main Module for Accessing SL4A (ndk-to-sl4a.c)***

[--code omitted--]

[10] http://code.google.com/p/android-cruft/wiki/SL4AC

```
main(int argc, char **argv) {
  int port = 0;
  if (argc != 2) {
    printf("Usage: %s port\n", argv[0]);
    return 1;
  }
  port = atoi(argv[1]);

  int socket_fd = init_socket("localhost", port);
  if (socket_fd < 0) return 2;

  json_t *params = json_array();
  json_array_append(params, json_string("w00t!"));
  sl4a_rpc(socket_fd, "makeToast", params);
}
[--code omitted--]
```

The supporting C functions are shown in Listing 5–3. We will not go into the details, but present the listing here for the sake of completeness.

Listing 5–3. *C Support Functions for Accessing SL4A (ndk-to-sl4a.c)*

```
// Released into the public domain, 15 August 2010
// This program demonstrates how a C application can access some of the Android
// API via the SL4A (Scripting Languages for Android, formerly "ASE", or Android
// Scripting Environment) RPC mechanism.  It works either from a host computer
// or as a native binary compiled with the NDK (rooted phone required, I think)
// SL4A is a neat Android app that provides support for many popular scripting
// languages like Python, Perl, Ruby and TCL.  SL4A exposes a useful subset of
// the Android API in a clever way: by setting up a JSON RPC server.  That way,
// each language only needs to implement a thin RPC client layer to access the
// whole SL4A API.
// The Android NDK is a C compiler only intended for writing optimized
// subroutines of "normal" Android apps written in Java.  So it doesn't come
// with any way to access the Android API.
// This program uses the excellent "Jansson" JSON library to talk to SL4A's
// RPC server, effectively adding native C programs to the list of languages
// supported by SL4A.
// To try it, first install SL4A: http://code.google.com/p/android-scripting/
//
// Start a private server with View->Interpreters->Start Server
//
// Note the port number the server is running on by pulling down the status
// bar and tapping "SL4A service".
// This program works just fine as either a native Android binary or from a
// host machine.
// ------------
// To compile on an ordinary linux machine, first install libjansson.  Then:
// $ gcc -ljansson ndk-to-sl4a.c -o ndk-to-sl4a
// To access SL4A on the phone use "adb forward tcp:XXXXX tcp:XXXXX" to port
// forward the SL4A server port from your host to the phone.  See this
// page for more details:
// http://code.google.com/p/android-scripting/wiki/RemoteControl
// ------------
// To compile using the NDK:
//   1. Make sure you can compile "Hello, world" using the NDK.  See:
//      http://credentiality2.blogspot.com/2010/08/native-android-c-program-➥
using-ndk.html
//
//   2. If you followed the above instructions, you have a copy of the agcc.pl
//      wrapper that calls the NDK's gcc compiler with the right options for
//      standalone apps.
//
//   3. Unpack a fresh copy of the jansson sources.  Tell configure to build for
//      Android:
//
// $ CC=agcc.pl ./configure --host=arm
// $ make
//
//   4. Cross your fingers and go!  (I'm quite certain there's a more elegant
//      way to do this)
//
// $ agcc.pl -I/path/to/jansson-1.3/src -o ndk-to-sl4a-arm ndk-to-sl4a.c➥
 /path/to/jansson-1.3/src/*.o
//
//   5. Copy to the phone and run it with the port of the SL4A server!

#include <stdio.h>
#include <jansson.h>
#include <unistd.h>
#include <string.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

// This mimics SL4A's android.py, constructing a JSON RPC object and
// sending it to the SL4A server.
int sl4a_rpc(int socket_fd, char *method, json_t *params) {
  static int request_id = 0; // monotonically increasing counter

  json_t *root = json_object();

  json_object_set(root, "id", json_integer(request_id));
  request_id++;

  json_object_set(root, "method", json_string(method));

  if (params == NULL) {
    params = json_array();
    json_array_append(params, json_null());
  }

  json_object_set(root, "params", params);
  char *command = json_dumps(root, JSON_PRESERVE_ORDER | JSON_ENSURE_ASCII);
  printf("command string:'%s'\n", command);

  write(socket_fd, command, strlen(command));
  write(socket_fd, "\n", strlen("\n"));

  // At this point we just print the response, but really we should buffer it
  // up into a single string, then pass it to json_loads() for decoding.
  printf("Got back:\n");
  while (1) {
    char c;
    read(socket_fd, &c, 1);
    printf("%c", c);
    if (c == '\n') {
      break;
    }
  }
  fflush(stdout);
  return 0;
}


// This function is just boilerplate TCP socket setup code
int init_socket(char *hostname, int port) {
  int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
  if (socket_fd == -1) {
    perror("Error creating socket");
    return 0;
  }

  struct hostent *host = gethostbyname(hostname);
  if (host == NULL) {
    perror("No such host");
    return -1;
  }

  struct sockaddr_in socket_address;

  int i;
  for (i=0; i < sizeof(socket_address); i++) {
    ((char *) &socket_address)[i] = 0;
  }

  socket_address.sin_family = AF_INET;

  for (i=0; i < host->h_length; i++) {
    ((char *) &socket_address.sin_addr.s_addr)[i] = ((char *) host->h_addr)[i];
  }

  socket_address.sin_port = htons(port);

  if (connect(socket_fd, (struct sockaddr *) &socket_address, sizeof(socket_address)) ➥
< 0) {
   perror("connect() failed");
   return -1;
 }

 return socket_fd;
}

[--code omitted--]
```

Getting the SL4A Source Code

To quote from the home site of SL4A:

> *"SL4A is designed for developers and is alpha quality software."*

This means that you can expect SL4A to go through relatively frequent changes and releases until it moves out of alpha. For this reason, we will present methods by which you can retrieve the SL4A source code.

Cloning the SL4A Source Code

We will show you several options for getting a local copy of the SL4A source code repository.

Installing Mercurial

We recommend that you install a local copy of Mercurial using the following resources:

- Mercurial source code management home:
  - http://mercurial.selenic.com/
  - http://mercurial.selenic.com/about/
- Mercurial downloads:
  - http://mercurial.selenic.com/downloads/
- Mercurial tools:
  - http://mercurial.selenic.com/wiki/OtherTools

NOTE: The Eclipse IDE plugin for Mercurial that we will present later also includes the option of installing the Mercurial executables and binaries for the Windows platform. We prefer to install Mercurial as a stand-alone application and add it to the system PATH variable.

To verify the Mercurial installation on your development computer, execute the following command from the terminal command line:

```
hg –v
```

This should result in output similar to the following:

```
Mercurial Distributed SCM (version 1.6.2)
Copyright (C) 2005–2010 Matt Mackall <mpm@selenic.com> and others
This is free software; see the source for copying conditions.
[--text omitted--]
use "hg help" for the full list of commands
```

Getting SL4A Using the Mercurial Hg Executable

We assume that you have installed a local copy of Mercurial and that it is on your system PATH variable. Enter into a file system directory of your choice using the terminal command line and execute the following command:

```
hg clone https://android-scripting.googlecode.com/hg/ android-scripting
```

This should result in output somewhat similar to the following:

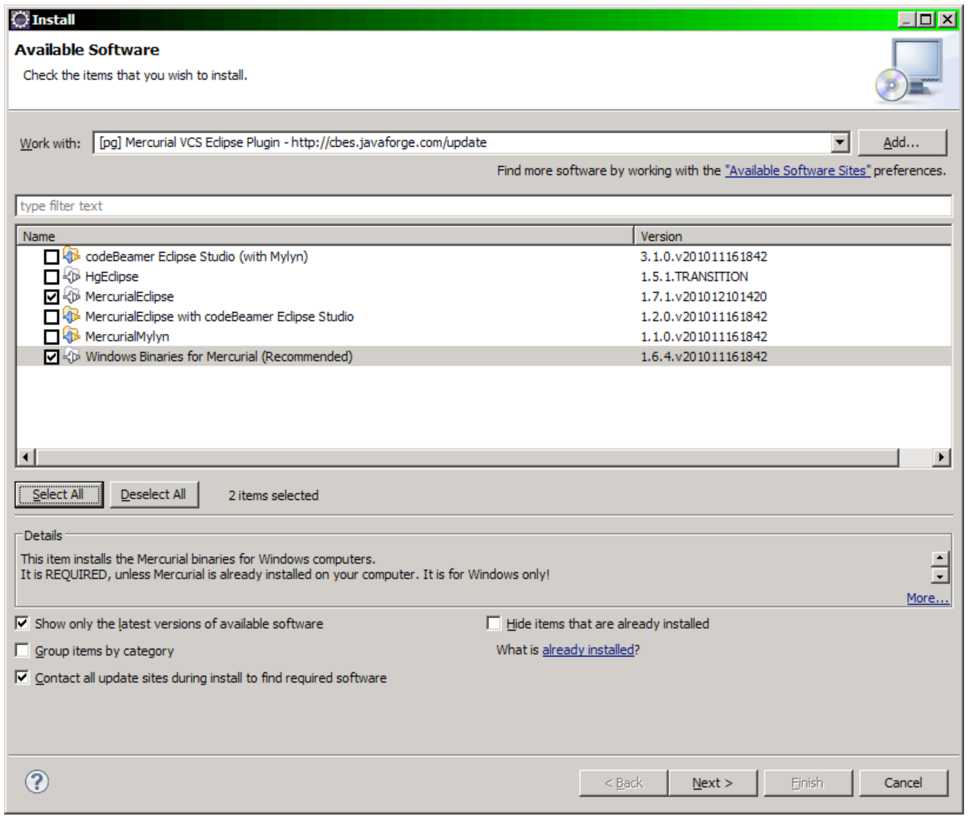
```
hg clone https://android-scripting.googlecode.com/hg/ android-scripting
requesting all changes
adding changesets
adding manifests
adding file changes
added 1066 changesets with 34881 changes to 28397 files
updating to branch default
11207 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Using Mercurial with Eclipse

You can install the Mercurial plugin for Eclipse from the update location below:

- Eclipse Mercurial plugin:
    - http://javaforge.com/project/HGE
- Eclipse Mercurial plugin update site:
    - http://cbes.javaforge.com/update

Figure 5–22 displays the form required for installing the Eclipse Mercurial plugin.



Install

Available Software

Check the items that you wish to install.

Work with: [pg] Mercurial VCS Eclipse Plugin - http://cbes.javaforge.com/update

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ codeBeamer Eclipse Studio (with Mylyn)	3.1.0.v201011161842
☐ HgEclipse	1.5.1.TRANSITION
☑ MercurialEclipse	1.7.1.v201012101420
☐ MercurialEclipse with codeBeamer Eclipse Studio	1.2.0.v201011161842
☐ MercurialMylyn	1.1.0.v201011161842
☑ Windows Binaries for Mercurial (Recommended)	1.6.4.v201011161842

2 items selected

Details

This item installs the Mercurial binaries for Windows computers.
It is REQUIRED, unless Mercurial is already installed on your computer. It is for Windows only!

☑ Show only the latest versions of available software
☐ Group items by category
☑ Contact all update sites during install to find required software
☐ Hide items that are already installed

What is already installed?

Figure 5–22. *Eclipse Mercurial plugin*

We will not follow the whole sequence here, but Figure 5–23 displays the form required for cloning a Mercurial repository from within Eclipse.

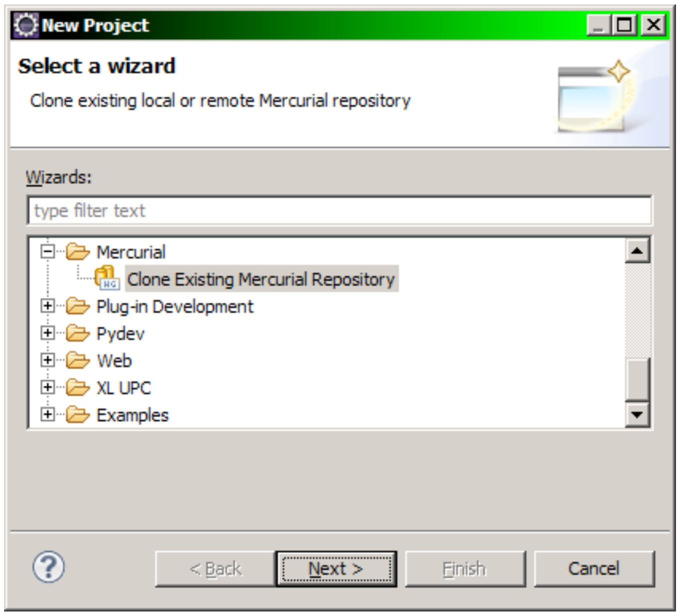


Figure 5–23. *Cloning a Mercurial repository with the Eclipse Mercurial plugin*

Using Mercurial with IntelliJ IDEA

You can install the Mercurial plugin for IntelliJ IDEA from the following location:

- IntelliJ IDEA Mercurial plugin:
  - http://plugins.intellij.net/plugin/?id=3370
  - https://bitbucket.org/willemv/hg4idea

Figure 5–24 displays the form required for using the IntelliJ IDEA Mercurial plugin. This will allow you to clone a Mercurial repository from the IDE.

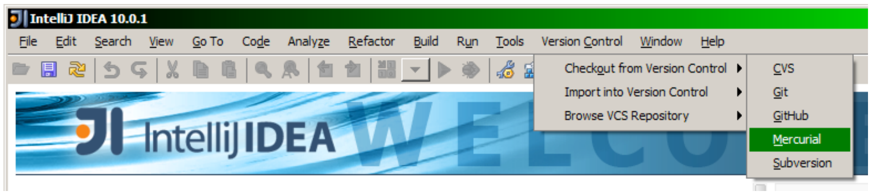

IntelliJ IDEA 10.0.1
File Edit Search View Go To Code Analyze Refactor Build Run Tools Version Control Window Help
Checkout from Version Control
Import into Version Control
Browse VCS Repository
CVS
Git
GitHub
Mercurial
Subversion
IntelliJ IDEA

Figure 5–24. *IntelliJ IDEA Mercurial plugin*

Again, we will not follow the whole sequence here, but Figure 5–25 displays the form required for cloning a Mercurial repository from within IntelliJ IDEA.

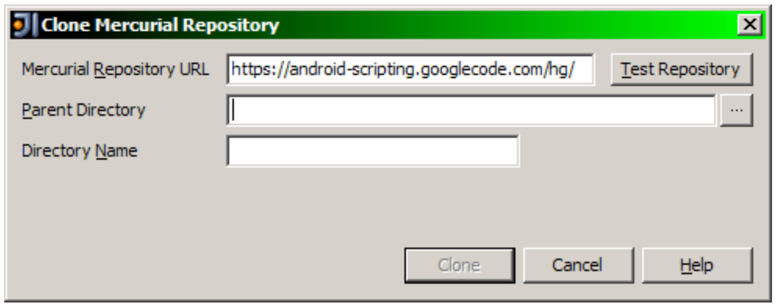


Figure 5–25. *Cloning a Mercurial repository with the IntelliJ IDEA Mercurial plugin*

SL4A Hello World Examples

To get an idea of the spirit of SL4A, here are some basic examples in different scripting languages. Install the interpreters and try them out.

- BeanShell:

```
source("/sdcard/com.googlecode.bshforandroid/extras/bsh/android.bsh");
droid = Android();
droid.call("makeToast", "Hello, Android!");
```

- JavaScript:

```
load("/sdcard/com.googlecode.rhinoforandroid/extras/rhino/android.js");
var droid = new Android();
droid.makeToast("Hello, Android!");
```

- Perl:

```
use Android;
my $a = Android->new();
$a->makeToast("Hello Practical Android Projects!");
```

- Python:

```
import android
andy = android.Android()
andy.makeToast("Hello Practical Android Projects!")
```

- Ruby:

```
droid = Android.new
droid.makeToast "Hello Practical Android Projects!"
```

- TCL:

```
package require android
set android [android new]
$android makeToast "Hello, Android!"
```

We hope that this will encourage you to study SL4A in more depth.

Summary

The main objective of this chapter was to introduce you to the Scripting Layer for Android (SL4A) platform. SL4A is a growing topic and is well worth investigating in depth.

We helped you get a basic understanding of how SL4A works and to be able to use it to run your own scripts on the Android platform.

You now have enough information to clone your own copy of the SL4A source code repository in order to build SL4A yourself.