FREE eBook

LEARNING vala

#vala

Table of Contents

About

Chapter 1: Getting started with vala

Remarks

This section provides an overview of what vala is, and why a developer might want to use it.

It should also mention any large subjects within vala, and link out to the related topics. Since the Documentation for vala is new, you may need to create initial versions of those related topics.

Versions

Version	Release Date
0.36.4	2017-06-26
0.36.3	2017-05-02
0.36.2	2017-04-25
0.36.1	2017-04-03
0.36.0	2017-03-18
0.34.6	2017-03-02
0.34.5	2017-03-02
0.34.4	2016-12-05
0.34.3	2016-11-22
0.34.2	2016-10-23
0.34.1	2016-10-09
0.34.0	2016-09-19
0.32.1	2016-06-20
0.32.0	2016-03-21
0.31.1	2016-02-07
0.30.2	2016-06-20
0.30.1	2016-01-31
0.30.0	2015-09-18

Version	Release Date
0.29.3	2015-08-11
0.29.2	2015-06-22
0.29.1	2015-05-27
0.28.1	2015-08-11
0.28.0	2015-03-22
0.27.2	2015-03-18
0.27.1	2015-01-12
0.26.2	2015-01-12
0.26.1	2014-10-13
0.26.0	2014-09-22
0.25.4	2014-09-15
0.25.3	2014-09-01
0.25.2	2014-08-24
0.25.1	2014-07-23
0.24.0	2014-03-24
0.23.3	2014-02-18
0.23.2	2014-02-05
0.23.1	2013-12-22
0.22.1	2013-11-13
0.22.0	2013-09-23
0.21.2	2013-09-13
0.21.1	2013-08-02
0.20.1	2013-04-08
0.20.0	2013-03-26
0.19.0	2013-02-20

Version	Release Date
0.18.1	2012-11-13
0.18.0	2012-09-24
0.17.7	2012-09-16
0.17.6	2012-09-03
0.17.5	2012-08-20
0.17.4	2012-08-06
0.17.3	2012-07-16
0.17.2	2012-06-24
0.17.1	2012-06-02
0.17.0	2012-04-28
0.16.1	2012-06-23
0.16.0	2012-03-26
0.15.2	2012-02-25
0.15.1	2012-01-26
0.15.0	2011-12-05
0.14.2	2012-01-31
0.14.1	2011-11-30
0.14.0	2011-09-17
0.13.4	2011-09-07
0.13.3	2011-08-22
0.13.2	2011-08-16
0.13.1	2011-07-06
0.13.0	2011-06-17
0.12.1	2011-06-01
0.12.0	2011-04-03

Version	Release Date
0.11.7	2011-03-16
0.11.6	2011-02-14
0.11.5	2011-01-21
0.11.4	2011-01-15
0.11.3	2011-01-05
0.11.2	2010-11-08
0.11.1	2010-10-25
0.11.0	2010-10-04
0.10.4	2011-03-12
0.10.3	2011-01-22
0.10.2	2010-12-28
0.10.1	2010-10-26
0.10.0	2010-09-18
0.9.8	2010-09-04
0.9.7	2010-08-19
0.9.6	2010-08-18
0.9.5	2010-08-09
0.9.4	2010-07-27
0.9.3	2010-07-14
0.9.2	2010-06-20
0.9.1	2010-06-07
0.8.1	2010-04-21
0.8.0	2010-03-31
0.7.10	2010-02-04
0.7.9	2009-12-19

Version	Release Date
0.7.8	2009-11-04
0.7.7	2009-09-27
0.7.6	2009-09-18
0.7.5	2009-08-02
0.7.4	2009-06-28
0.7.3	2009-05-26
0.7.2	2009-05-07
0.7.1	2009-04-20
0.7.0	2009-04-05
0.6.1	2009-04-12
0.6.0	2009-03-30
0.5.7	2009-02-20
0.5.6	2009-01-18
0.5.5	2009-01-10
0.5.4	2009-01-07
0.5.3	2008-12-16
0.5.2	2008-12-01
0.5.1	2008-11-03
0.4.0	2008-10-20

Examples

Installation or Setup

The easiest way of installing Vala is to install your distribution-specific package.

On Ubuntu:

```
sudo apt install valac
```

On Fedora:

```
sudo dnf install vala
```

On Arch:

```
sudo pacman -S vala
```

On OS X, with Homebrew:

```
brew install vala
```

On Windows, you can get an installer for the latest version here.

You can also build it from sources, but you'll need to install `pkg-config`, a C compiler, a standard C library and GLib 2 before:

```
wget https://download.gnome.org/sources/vala/0.34/vala-0.34.4.tar.xz
tar xvf vala-0.34.4.tar.xz
cd vala-0.34.4
./configure
make
sudo make install
```

Hello world!

In `foo.vala`:

```
void main (string[] args) {
    stdout.printf ("Hello world!");
}
```

To compile the source into the `foo` binary:

```
valac foo.vala
```

To compile and run the source:

```
vala foo.vala
```

Read Getting started with vala online: https://riptutorial.com/vala/topic/9067/getting-started-with-vala

Chapter 2: Async and Yield

Introduction

Vala provide two syntax constructs to deal with asynchonous operations: `async` function and `yield` statement.

Examples

Declare an Asynchronous Function

```
public async int call_async () {
    return 1;
}

call_async.begin ((obj, res) => {
    var ret = call_async.end (res);
});
```

To call an asynchronous functions from a synchronous context, use the `begin` method and pass a callback to receive the result. The two arguments are:

- `obj` is a `GLib.Object` if this call was defined in a class
- `res` is a `GLib.AsyncResult` holding the result of the asynchronous operation

The `end` method extract the result of the operation.

Usage of GLib.Task to perform asynchronous operations

The `GLib.Task` provide low-level API for performing asynchronous operations.

```
var task = new GLib.Task (null, null, (obj, result) => {
    try {
        var ret = result.propagate_boolean ();
    } catch (Error err) {
        // handler err...
    }
});
```

Later in a thread or a callback:

```
task.return_boolean (true);
```

To use the `GLib.Task` internal thread pool:

```
task.run_in_thread (() => {
    task.return_boolean (true);
});
```

Yield from an Asynchronous Function

In order to chain asynchronous operations and avoid a callback hell, Vala supports the `yield` statement.

Used with an asynchronous invocation, it will pause the current coroutine until the call is completed and extract the result.

Used alone, `yield` pause the current coroutine until it's being woken up by invoking its source callback.

```
public async int foo_async () {
    yield; // pause the coroutine
    Timeout.add_seconds (5, bar_async.callback); // wakeup in 5 seconds
    return ret + 10;
}

public async int bar_async () {
    var ret = yield foo_async ();
}
```

Read Async and Yield online: https://riptutorial.com/vala/topic/9281/async-and-yield

Chapter 3: Classes

Introduction

Vala support various flavours of classes.

Remarks

Both `glib-2.0` and `gobject-2.0` dependencies are required unless `--nostdpkg` is explicitly given.

Examples

GObject Class

```
public class Foo : Object {
    public string prop { construct; get; }
}
```

It is meant for interospectable API using GObject Introspection. This is the recommended way for declaring classes.

Plain Class

```
public class Foo {
    public string prop { construct; get; }
}
```

Pure-Vala and lightweight class. This is useful if you need a compromise between efficiency of a `struct` and the feature of a full blown GObject class.

Compact Class

```
[Compact]
public class Foo {
    public string prop;
}
```

It is mainly used for writing bindings with specific memory management.

Read Classes online: https://riptutorial.com/vala/topic/9076/classes

Chapter 4: Functions

Introduction

Functions are pieces of code that can be executed by other functions of your program.

Your program always starts with the `main` function.

See also asynchronous functions.

Remarks

Methods are exactly the same as function, but they act on an object instance.

Examples

Basic functions

A function is defined by at least its return type and an unique name.

```
void say_hello () {
    print ("Hello, world!\n");
}
```

Then, to call it just use the name of the function followed by a parenthese.

```
say_hello ();
```

Functions can also have parameters between the parentheses, defined by their types and names and separated by commas. Then you can just use them as normal variables into your function.

```
int greet (string name, string family_name) {
    print ("Hello, %s %s!\n", name, family_name);
}
```

To call a function with parameters, just put a variable or a value between the parentheses.

```
string name = "John";
greet (name, "Doe");
```

You can also return a value which can be assigned to a variable with the `return` keyword.

```
int add (int a, int b) {
    return a + b;
}
```

```
int sum = add (24, 18);
```

All code path should end with a `return` statement. For instance, the following code is invalid.

```
int positive_sub (int a, int b) {
    if (a >= b) {
        return a - b;
    } else {
        // Nothing is returned in this case.
        print ("%d\n", b - a);
    }
}
```

Optional parameters

Parameters can be marked as optional by giving them a default value. Optional parameters can be omitted when calling the function.

```
string greet (string name, string language = "English") {
    if (language == "English") {
        return @"Hello, $name!";
    } else {
        return @"Sorry $name, I don't speak $language";
    }
}

greet ("John");
greet ("Jane", "Italian");
```

Out and Ref parameters

Value types (structures and enumerations) are passed by value to functions: a copy will be given to the function, not a reference to the variable. So the following function won't do anything.

```
void add_three (int x) {
    x += 3;
}

int a = 39;
add_three (a);
assert (a == 39); // a is still 39
```

To change this behavior you can use the `ref` keyword.

```
// Add it to the function declaration
void add_three (ref int x) {
    x += 3;
}

int a = 39;
add_three (ref a); // And when you call it
assert (a == 42); // It works!
```

`out` works the same way, but you are forced to set a value to this variable before the end of the function.

```
string content;
FileUtils.get_contents ("file.txt", out content);

// OK even if content was not initialized, because
// we are sure that it got a value in the function above.
print (content);
```

Contract programming

You can assert that parameters have certain values with `requires`.

```
int fib (int i) requires (i > 0) {
    if (i == 1) {
        return i;
    } else {
        return fib (i - 1) + fib (i - 2);
    }
}

fib (-1);
```

You won't get any error during the compilation, but you'll get an error when running your program and the function won't run.

You can also assert that the return value matches a certain condition with `ensures`

```
int add (int a, int b) ensures (result >= a && result >= b) {
    return a + b;
}
```

You can have as many `requires` and `ensures` as you want.

Variable arguments

```
int sum (int x, ...) {
    int result = x;
    va_list list = va_list ();
    for (int? y = list.arg<int?> (); y != null; y = list.arg<int?> ()) {
        result += y;
    }
    return result;
}

int a = sum (1, 2, 3, 36);
```

With this function, you can pass as many int as you want. If you pass something else, you'll either get an unexpected value or a segmentation fault.

Read Functions online: https://riptutorial.com/vala/topic/9319/functions

Chapter 5: Meson

Introduction

Meson is a next-generation build system designed with simplicity and explicitness in mind.

Examples

Basic project

```
project('Vala Project')

glib_dep = dependency('glib-2.0')
gobject_dep = dependency('gobject-2.0')

executable('foo', 'foo.vala', dependencies: [glib_dep, gobject_dep])
```

Note: both `glib-2.0` and `gobject-2.0` dependencies are required unless `--nostdpkg` is explicitly given.

Posix-based project (no GLib or GObject)

```
project('Posix-based Project', 'vala')

add_project_arguments(['--nostdpkg'], language: 'vala')

posix_dep = meson.get_compiler('vala').find_library('posix')

executable('foo', 'foo.vala', dependencies: [posix_dep])
```

Mixed sources

```
project('Mixed sources Project', 'vala')

glib_dep = dependency('glib-2.0')
gobject_dep = dependency('gobject-2.0')

executable('foo', 'foo.vala', 'bar.c', dependencies: [glib_dep, gobject_dep])
```

In `foo.vala`:

```
namespace Foo {
    public extern int bar ();

    public int main (string[] args) {
        return bar ();
    }
}
```

In `bar.c`:

```
int
bar ()
{
    return 0;
}
```

Read Meson online: https://riptutorial.com/vala/topic/9074/meson

Chapter 6: Ownership

Remarks

Note that the compiler will not prevent you from using variable for which its value ownership been transfeered.

Examples

Transfer Ownership

```
var foo = new uint8[12];
var bar = (owned) foo;
assert (foo == null);
```

The `bar` variable will own the value previously owned by `foo`.

Implicit Copy

```
var foo = new uint8[12];
var bar = foo;
assert (foo != bar);
```

In this example, the both `foo` and `bar` possess a strong reference, but since `uint8[]` only support single ownership, a copy is made.

Read Ownership online: https://riptutorial.com/vala/topic/9075/ownership

Chapter 7: Signals

Examples

Basic signal

Signals are only available to GObject classes. They can only be public, which means that any part of the code can connect handlers and trigger them.

```
public class Emitter : Object {
    // A signal is declared like a method,
    // but with the signal keyword.
    public signal void my_signal ();

    public void send_signal () {
        this.my_signal (); // Send a signal by calling it like a method.
    }
}

void main () {
    var emitter = new Emitter ();
    // Use the connect method of the signal to add an handler.
    emitter.my_signal.connect (() => {
        print ("Received the signal.\n");
    });
    emitter.send_signal ();
    emitter.my_signal (); // You can send a signal from anywhere.
}
```

You can also use normal functions as handlers if they have the same signature as the signal.

```
void main () {
    var emitter = new Emitter ();
    emitter.connect (my_handler);
    emitter.my_signal ();
}

void my_handler () {
    print ("Received the signal.\n");
}
```

Detailed signal

You can write detailed signals with the `[Signal (detailed = true)]` attribute.

```
public class Emitter : Object {
    [Signal (detailed = true)]
    public signal void detailed_signal ();

    public void emit_with_detail (string detail) {
        this.detailed_signal[detail] ();
    }
}

void main () {
    var emitter = new Emitter ();

    // Connect only when the detail is "foo".
    emitter.detailed_signal["foo"].connect (() => {
       print ("Received the signal with 'foo'.\n");
    });

    // Connect to the signal, whatever is the detail.
    emitter.detailed_signal.connect (() => {
        print ("Received the signal.\n");
    });

    emitter.emit_with_detail ("foo"); // Both handlers will be triggered.
    emitter.emit_with_detail ("bar"); // Only the general handler will be triggered.
}
```

This feature is often used with the `notify` signal, that any `Object` based class has, and which is sent when a property changes. The detail here is the name of the property, so you can choose to connect to this signal only for some of them.

```
public class Person : Object {
    public string name { get; set; }
    public int age { get; set; }
}

void main () {
    var john = new Person () { name = "John", age = 42 });
    john.notify["age"].connect (() => {
        print ("Happy birthday!");
    });
    john.age++;
}
```

Default handler and connect_after

Signals can have a default handler. All you need to do is to give it a body when you declare it.

```
public class Emitter : Object {
    public signal void my_signal () {
        print ("Hello from the default handler!\n");
    }
}
```

This handler will always be called after the `connect`ed ones. But you can use `connect_after` instead of `connect` if you want to add an handler after the default one.

```
var emitter = new Emitter ();
emitter.my_signal.connect_after (() => {
    print ("After the default handler!\n");
});
emitter.my_signal ();
```

Read Signals online: https://riptutorial.com/vala/topic/9791/signals

Chapter 8: Using GLib.Value

Examples

How to initialize it ?

To initialize struct, you can do like this :

```
public static void main (string[] args) {
    Value val = Value (typeof (int));
    val.set_int (33);
}
```

But Vala brings another way to initialize values :

```
public static void main (string[] args) {
    Value val = 33;
}
```

Your value is initialized with 'int' type and holds '33' int value.

How to use it ?

Use one of GLib.Value get methods (see valadoc documentation) or cast your value with the type of your value :

```
public static void main (string[] args) {
    Value val = 33;
    int i = val.get_int();
    int j = (int)val;
}
```

Note : if your current value doesn't contain desired type, GObject system will throw critical error :

```
public static void main (string[] args) {
    Value val = 33;
    string s = (string)val;
}
```

```
(process:5725): GLib-GObject-CRITICAL **: g_value_get_string: assertion 'G_VALUE_HOLDS_STRING
(value)' failed
```

Use GLib.Value in function parameters

This exemple shows how you can pass several types in function parameters :

```
static void print_value (Value val) {
    print ("value-type : %s\n", val.type().name());
    print ("value-content : %s\n\n", val.strdup_contents());
}

public static void main (string[] args) {
    print_value (33);
    print_value (24.46);
    print_value ("string");
}
```

```
value-type : gint
value-content : 33

value-type : gdouble
value-content : 24.460000

value-type : gchararray
value-content : "string"
```

Note : if GObject can transform your value with 'string' type (gchararray), 'strdup_contents' returns converted value, instead of pointer adress

```
static void print_value (Value val) {
    print ("value-type : %s\n", val.type().name());
    print ("value-content : %s\n\n", val.strdup_contents());
}

public static void main (string[] args) {
    print_value (new DateTime.now_local());
}
```

```
value-type : GDateTime
value-content : ((GDateTime*) 0x560337def040)
```

Register types for GLib.Value

In previous example, Value.strdup_contents prints GLib.DateTime as pointer address. You can register functions that will transform value to desired type. First, create a function that will have this signature :

```
static void datetime_to_string (Value src_value, ref Value dest_value) {
    DateTime dt = (DateTime)src_value;
    dest_value.set_string (dt.to_string());
}
```

then register this function with Value.register_transform_func :

```
Value.register_transform_func (typeof (DateTime), typeof (string), datetime_to_string);
```

now GObject can convert any DateTime object to string value.

the complete example :

```
static void datetime_to_string (Value src_value, ref Value dest_value) {
    DateTime dt = (DateTime)src_value;
    dest_value.set_string (dt.to_string());
}

static void print_value (Value val) {
    print ("value-type : %s\n", val.type().name());
    print ("value-content : %s\n\n", val.strdup_contents());
}

public static void main (string[] args) {
    print_value (new DateTime.now_local());
    Value.register_transform_func (typeof (DateTime), typeof (string), datetime_to_string);
    print_value (new DateTime.now_local());
}
```

```
value-type : GDateTime
value-content : ((GDateTime*) 0x560337def040)

value-type : GDateTime
value-content : 2017-04-20T18:40:20+0200
```

Read Using GLib.Value online: https://riptutorial.com/vala/topic/9777/using-glib-value

Chapter 9: Using Vala on Windows

Introduction

This topic focuses on how to get valac running on Windows.

Examples

Using msys2 (64 Bit)

1. Install msys2 (http://www.msys2.org/)

2. Install the required prerequisites for Vala

   ```
   pacman -S mingw64/mingw-w64-x86_64-gcc
   pacman -S mingw64/mingw-w64-x86_64-pkg-config
   pacman -S mingw64/mingw-w64-x86_64-vala
   ```

   and all the additional packages your code requires, i.e.

   ```
   pacman -S mingw64/mingw-w64-x86_64-libgee
   ...
   ```

3. Launch the correct msys2 shell

   ```
   C:\msys64\mingw64.exe
   ```

4. Check the `MSYSTEM` and `PKG_CONFIG_PATH` environment variables

   ```
   $ echo $MSYSTEM
   MINGW64

   $ echo $PKG_CONFIG_PATH
   /mingw64/lib/pkgconfig:/mingw64/share/pkgconfig
   ```

5. Run `valac` as usual, but make sure to always work in the correct environment (see steps 3 and 4)

   For example let's build the first GeeSample here:

   ```
   $ valac gee-list.vala --pkg gee-0.8
   ```

Read Using Vala on Windows online: https://riptutorial.com/vala/topic/9899/using-vala-on-windows

Credits

S. No	Chapters	Contributors
1	Getting started with vala	Adrià Arrufat, arteymix, avojak, Community, Günther Wutz
2	Async and Yield	arteymix
3	Classes	AlThomas, arteymix, Community, Günther Wutz
4	Functions	Community
5	Meson	arteymix
6	Ownership	arteymix
7	Signals	Community
8	Using GLib.Value	yannick inizan
9	Using Vala on Windows	Jens Mühlenhoff