

Vala Reference Manual

- Overview
 - Getting started
- Types
 - Value types
 - Reference types
 - Nullable types
 - Pointer types
- Expressions
 - Primary expressions
 - Unary expressions
 - Arithmetic operations
 - Shift operations
 - Relational operations
 - Logical operations
 - Assignments
- Statements
 - Selection statements
 - Iteration statements
 - Jump statements
 - Try statement
- Namespaces
 - Namespace declarations
- Methods
 - Method declarations
- Classes
 - Class declarations
 - Fields
 - Methods
 - Properties
 - Signals
- Structs
 - Struct declarations
 - Fields
 - Methods
 - Properties
- Interfaces
 - Interface declarations
 - Methods
 - Delegates

- Signals
- Enums
 - Enum declarations
- Delegates
 - Delegate declarations
- Exceptions
 - Exception handling
- Attributes

Vala Reference Manual

- Overview
 - Getting started
- Types
 - Value types
 - Reference types
 - Nullable types
 - Pointer types
- Expressions
 - Primary expressions
 - Unary expressions
 - Arithmetic operations
 - Shift operations
 - Relational operations
 - Logical operations
 - Assignments
- Statements
 - Selection statements
 - Iteration statements
 - Jump statements
 - Try statement
- Namespaces
 - Namespace declarations
- Methods
 - Method declarations
- Classes
 - Class declarations
 - Fields
 - Methods
 - Properties
 - Signals
- Structs
 - Struct declarations
 - Fields
 - Methods
 - Properties
- Interfaces
 - Interface declarations
 - Methods
 - Delegates

- Signals
- Enums
 - Enum declarations
- Delegates
 - Delegate declarations
- Exceptions
 - Exception handling
- Attributes

Overview

Vala is a new programming language that aims to bring modern programming language features to GNOME developers without imposing any additional runtime requirements and without using a different ABI compared to applications and libraries written in C.

Getting started

The classic "Hello, world" example in Vala:

```
using GLib;

public class Sample : Object {
    public Sample () {
    }

    public void run () {
        stdout.printf ("Hello, world!\n");
    }

    static int main (string[] args) {
        var sample = new Sample ();
        sample.run ();
        return 0;
    }
}
```

Store the code in a file whose name ends in ".vala", such as `hello.vala`, and compile it with the command:

```
valac -o hello hello.vala
```

This will produce an executable file called `hello`.

Types

Vala supports four kinds of data types: value types, reference types, type parameters, and pointer types. Value types include simple types (e.g. char, int, and float), enum types, and struct types. Reference types include object types, array types, delegate types, and error types. Type parameters are parameters used in generic types.

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

type: value-type reference-type nullable-type type-parameter pointer-type

Value types

Instances of value types are stored directly in variables. They are duplicated whenever assigned to another variable (e.g. passed to a method). For local variables, value types are stored on the stack.

value-type: struct-type enum-type struct-type: type-name integral-type floating-point-type bool
integral-type: char uchar short ushort int uint long ulong size_t ssize_t int8 uint8 int16 uint16 int32
uint32 int64 uint64 unichar floating-point-type: float double enum-type: type-name

Struct types

Documentation

Simple types

Documentation

Integral types

Documentation

Floating point types

Documentation

The bool type

Documentation

Enumeration types

An enumeration type is a type containing named constants.

See enums.

Reference types

Instances of reference types are always stored on the heap. Variables contain references to them. Assigning to another variable duplicates reference, not object.

reference-type: object-type class-type array-type delegate-type error-type weak-reference-type
weak-reference-type: weak object-type weak class-type weak array-type weak delegate-type weak
error-type object-type: type-name string class-type: type-name . Class array-type: non-array-type []
non-array-type [dim-seperators] non-array-type: value-type object-type class-type delegate-type
error-type dim-seperators: , dim-seperators , delegate-type: type-name error-type: type-name

Weak reference types

Documentation

Array types

An array is a data structure that contains zero or more elements of the same type.

Delegate types

A delegate is a data structure that refers to a method, and for instance methods, it also refers to the corresponding object instance.

Error types

Instances of error types represent recoverable runtime errors.

Nullable types

An instance of a nullable type `T?` can either be a value of type `T` or `null`.

nullable-type: value-type ? reference-type ?

Pointer types

Unlike references, pointers are not tracked by the memory manager. The value of a pointer having type `T*` represents the address of a variable of type `T`. The pointer indirection operator `*` can be used to access this variable. Like a nullable object reference, a pointer can be null. The `void*` type represents a pointer to an unknown type. As the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa) and compared to values of other pointer types.

pointer-type: type-name * pointer-type * void*

Expressions

Primary expressions

primary-expression: literal simple-name (expression) member-access invocation-expression
element-access this base object-creation-expression array-creation-expression sizeof (type) typeof (type)

Unary expressions

unary-expression: primary-expression + unary-expression - unary-expression ! unary-expression ~ unary-expression cast-expression

Arithmetic operations

multiplicative-expression: unary-expression multiplicative-expression * unary-expression
multiplicative-expression / unary-expression multiplicative-expression % unary-expression
additive-expression: multiplicative-expression additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

Shift operations

shift-expression: additive-expression shift-expression << additive-expression shift-expression >> additive-expression

Relational operations

relational-expression: shift-expression relational-expression < shift-expression relational-expression
<= shift-expression relational-expression > shift-expression relational-expression >= shift-expression
equality-expression: relational-expression equality-expression == relational-expression
equality-expression != relational-expression

Logical operations

and-expression: equality-expression and-expression & equality-expression exclusive-or-expression:
and-expression exclusive-or-expression ^ and-expression inclusive-or-expression:
exclusive-or-expression inclusive-or-expression | exclusive-or-expression

Assignments

assignment: unary-expression = expression unary-expression += expression unary-expression -=
expression unary-expression *= expression unary-expression /= expression unary-expression %=
expression unary-expression &= expression unary-expression |= expression unary-expression ^=

expression unary-expression <<= expression unary-expression >>= expression

Statements

Selection statements

The if statement selects a statement for execution based on the value of a boolean expression.

if-statement: if (boolean-expression) embedded-statement if (boolean-expression)
embedded-statement else embedded-statement

Iteration statements

The while statement conditionally executes an embedded statement zero or more times.

while-statement: while (boolean-expression) embedded-statement

The do statement conditionally executes an embedded statement one or more times.

do-statement: do embedded-statement while (boolean-expression) ;

The for statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

for-statement: for ([for-initializer] ; [for-condition] ; [for-iterator]) embedded-statement
for-initializer: local-variable-declaration statement-expression-list for-condition: boolean-expression
for-iterator: statement-expression-list statement-expression-list: statement-expression
statement-expression-list , statement-expression

Within the embedded statement of a for statement, a break statement can be used to transfer control to the end point of the for statement (thus ending iteration of the embedded statement), and a continue statement can be used to transfer control to the end point of the embedded statement (thus executing another iteration of the for statement).

The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

foreach-statement: foreach (type identifier in expression) embedded-statement

Jump statements

The break statement exits the nearest enclosing switch, while, do, for, or foreach statement.

break-statement: break ;

The continue statement starts a new iteration of the nearest enclosing while, do, for, or foreach statement.

continue-statement: continue ;

When multiple while, do, for, or foreach statements are nested within each other, a continue statement applies only to the innermost statement. If a continue statement is not enclosed by a while, do, for, or foreach statement, a compile-time error occurs.

The return statement returns control to the caller of the function member in which the return statement appears.

return-statement: return [expression] ;

The throw statement throws an exception.

throw-statement: throw expression ;

Try statement

The try statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the try statement provides the ability to specify a block of code that is always executed when control leaves the try statement.

try-statement: try block catch-clauses try block [catch-clauses] finally-clause catch-clauses:
specific-catch-clause [specific-catch-clauses] general-catch-clause specific-catch-clause:
specific-catch-clause specific-catch-clauses specific-catch-clause specific-catch-clause: catch (
error-type identifier) block general-catch-clause: catch block finally-clause: finally block

Namespaces

Namespace declarations

namespace-declaration: namespace qualified-identifier { [namespace-members] }
qualified-identifier: [qualified-identifier .] identifier namespace-members: [namespace-members]
namespace-member namespace-member: namespace-declaration method-declaration
field-declaration constant-declaration class-declaration struct-declaration interface-declaration
enum-declaration delegate-declaration errordomain-declaration

Methods

Method declarations

Methods may be declared in namespaces, classes, interfaces, structs, enums, and error domains

method-declaration: [access-modifier] [member-modifiers] return-type qualified-identifier ([parameter-list]) method-contract [throws error-list] { statement-list } member-modifiers:
member-modifier [member-modifiers] member-modifier: abstract class extern inline override static
virtual return-type: type void parameter-list: [parameter-direction] type identifier [, parameter-list]
parameter-direction: ref out method-contract: [requires (expression)] [ensures (expression)]
error-list: error-type [, error-list]

Classes

A class is a data type that can contain fields, constants, methods, properties, and signals. Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class.

Class declarations

The simplest class declaration looks like this:

```
class ClassName {
    <class-member>
}
```

As class types support inheritance, you can specify a base class you want to derive from:

```
class ClassName : BaseClassName {
    <class-member>
}
```

GObject Note

It's recommended that you derive all your classes directly or indirectly from `GLib.Object`, unless you have a strong reason not to. Some class features are not supported for classes not deriving from `GLib.Object`.

Classes cannot have multiple base classes, however they may implement multiple interfaces:

```
class ClassName : BaseClassName, FirstInterfaceName, SecondInterfaceName {
    <class-member>
}
```

You may optionally specify an accessibility modifier. Classes support `public` and `private` accessibility and default to `private` if you don't specify one. Public classes may be accessed from outside the library or application they are defined in.

```
public class ClassName {
    <class-member>
}
```

The `abstract` modifier may be placed between the optional accessibility modifier and the class name to define an abstract class. An abstract class cannot be instantiated and is used as a base class for derived classes.

```
abstract class ClassName {
    <class-member>
}
```

The `static` modifier may be placed between the optional accessibility modifier and the class name to define a static class. A static class cannot be instantiated and may not have a base class. It can also not be used as a base class for derived classes and may only contain static members. Static classes are implicitly abstract, you may not use both modifiers, `abstract` and `static`, in the same class declaration.

```
static class ClassName {  
    <class-member>  
}
```

You may optionally prefix the class name with a namespace name. This places the class in the specified namespace without the need for a separate namespace declaration.

```
class NamespaceName.ClassName {  
    <class-member>  
}
```

Fields

Documentation

Methods

Documentation

Properties

property-declaration: [access-modifier] [member-modifiers] type identifier { accessor-declarations [default-value] } ; accessor-declarations: get-accessor [set-accessor] set-accessor [get-accessor]
get-accessor: [access-modifier] get ; [access-modifier] get { statement-list } set-accessor: [access-modifier] set [construct] ; [access-modifier] set [construct] { statement-list }
default-value: default = expression ;

Signals

The signal system allows objects to emit signals that can be handled by user-provided signal handlers.

signal-declaration: [access-modifier] signal return-type identifier ([parameter-list]) ;

Structs

A struct is a data type that can contain fields, constants, and methods.

Struct declarations

The simplest struct declaration looks like this:

```
struct StructName {  
    <struct-member>  
}
```

You may optionally specify an accessibility modifier. Structs support `public` and `private` accessibility and default to `private` if you don't specify one. Public structs may be accessed from outside the library or application they are defined in.

```
public struct StructName {  
    <struct-member>  
}
```

You may optionally prefix the struct name with a namespace name. This places the struct in the specified namespace without the need for a separate namespace declaration.

```
struct NamespaceName.StructName {  
    <struct-member>  
}
```

Fields

Documentation

Methods

Documentation

Properties

Documentation

Interfaces

Interfaces can be implemented by classes to provide functionality with a common interface. Each class can implement multiple interface and interfaces can require other interfaces to be implemented.

Interface declarations

```
public interface InterfaceName : RequiredInterface <optional>
{
    <interface members>
}
```

Methods

Interfaces can contain abstract and non abstract methods.

```
public abstract void MethodName ();
public void MethodName2 ()
{
    <Implementation>
}
```

Delegates

Interfaces can also contain delegates

```
public delegate void DelegateName (void* data);
```

Signals

Signals are also supported by interfaces

```
public signal void SignalName ();
```

Enums

Enumerated types represent a set of constant values.

Enum declarations

```
enum-declaration: [ access-modifier ] enum qualified-identifier { [ enum-members ] }  
enum-members: enum-values [ ; [ enum-methods ] ] enum-values: enum-value [ , enum-values ]  
enum-value: identifier [ = expression ] enum-methods: method-declaration [ enum-methods ]
```

Delegates

Delegate declarations

A delegate represents a callback supplied by the programmer.

delegate-declaration: [access-modifier] delegate return-type qualified-identifier (parameter-list) ;

Vala Reference Manual

Exceptions

Exception handling

Documentation

Vala Reference Manual

Attributes

Documentation