Vala Reference Manual

Vala is a high level programming language that produces binaries for the native platform. The binaries maintain the C Application Binary Interface (ABI) and can be built as either an application or a library.

The Vala Reference Manual gives details on Vala's syntax and type system, including polymorphism using interfaces and type parameters (generics). Vala includes additional code generation routines, for example D-Bus inter-process communication and GTK+3 composite templates, these are only referred to in the Attributes section. The reference manual does not provide a tutorial for these features.

Vala is developed in a collaborative and self-supporting way by its users. They provide bug reports, documentation, patches, patch reviews and core development. Following this model of development the manual has an editable version at https://wiki.gnome.org/Projects/Vala/Manual .

Vala version: 0.50
Release: 0.50.2
Status: Draft, document contains omissions and might contain errors

1. Overview

1.1 Getting started

1.2 Documentation conventions

1.3 Vala source files

1.4 Vala conventions

1.5 Vala syntax

1.6 GType and GObject

1.7 Memory management

1.8 Vala compilation

1.9 Application entry point

2. Concepts

2.1 Variables, fields and parameters

2.2 Scope and naming

2.3 Object oriented programming

2.4 References and ownership

3. Types

3.1 Value types

3.2 Reference types

3.3 Parameterised types

3.4 Nullable types

3.5 Pointer types

3.6 Type conversions

4. Expressions

4.1 Literal expressions

4.2 Member access

4.3 Element access

4.4 Arithmetic operations

4.5 Relational operations

4.6 Increment/decrement operations

4.7 Logical operations

4.8 Bitwise operations

4.9 Assignment operations

4.10 Invocation expressions

4.11 Class instantiation
4.12 Struct instantiation
4.13 Array instantiation
4.14 Conditional expressions
4.15 Coalescing expressions
4.16 Flag operations
4.17 Type operations
4.18 Ownership transfer expressions
4.19 Lambda expressions
4.20 Pointer expressions

5. Statements

5.1 Simple statements
5.2 Variable declaration
5.3 Selection statements
5.4 Iteration statements
5.5 Jump Statements
5.6 Try Statement
5.7 Lock Statement
5.8 Unlock Statement
5.9 With Statement

6. Namespaces

6.1 The global namespace
6.2 Namespace declaration
6.3 Members
6.4 Fields
6.5 Constants
6.6 The "using" statement

7. Methods

7.1 Parameter directions
7.2 Method declaration
7.3 Invocation
7.4 Scope
7.5 Lambdas
7.6 Contract programming

8. Delegates

8.1 Types of delegate

8.2 Delegate declaration

8.3 Using delegates

8.4 Examples

9. Errors

9.1 Error throwing

9.2 Error catching

9.3 Examples

10. Classes

10.1 Types of class

10.2 Types of class members

10.3 Class scope

10.4 Class member visibility

10.5 Class declaration

10.6 Controlling instantiation

10.7 Construction

10.8 Class fields

10.9 Class constants

10.10 Class methods

10.11 Properties

10.12 Signals

10.13 Class enums

10.14 Class delegates

10.15 Examples

11. Interfaces

11.1 Interface declaration

11.2 Interface fields

11.3 Interface methods

11.4 Interface properties

11.5 Interface signals

11.6 Other interface members

11.7 Examples

12. Generics

12.1 Generics declaration

12.2 Instantiation

12.3 Examples

13. Structs

13.1 Struct declaration

13.2 Controlling instantiation

13.3 Struct fields

13.4 Struct constants

13.5 Struct methods

13.6 Examples

14. Enumerated types (Enums)

14.1 Enum declaration

14.2 Enum members

14.3 Methods

14.4 Flag types

14.5 Error domains

14.6 Examples

15. Attributes

15.1 Applying attributes

15.2 CCode Attribute

15.3 Version attribute

15.4 SimpleType attribute

15.5 BooleanType Attribute

15.6 IntegerType Attribute

15.7 FloatingType Attribute

15.8 Signal Attribute

15.9 Description Attribute

15.10 DBus Attribute

15.11 Gtk attributes

15.12 Other attributes

15.13 Deprecated Attributes

15.14 Examples

16. Preprocessor

16.1 Directives syntax

16.2 Defining symbols

16.3 Built-in defines

16.4 Examples

17. GIDL metadata format

17.1 Comments

17.2 Other Lines

17.3 Specifiers

17.4 Specifying Different Things

17.5 Properties Reference

17.6 Examples

18. GIR metadata format

18.1 Locating metadata

18.2 Comments

18.3 Syntax

18.4 Valid arguments

18.5 Examples

1. Overview

1.1 Getting started
1.2 Documentation conventions
1.3 Vala source files
1.4 Vala conventions
1.5 Vala syntax
1.6 GType and GObject
1.7 Memory management
1.8 Vala compilation
1.9 Application entry point

Vala is a programming language that aims to bring modern language features to GNOME developers without imposing any additional runtime requirements and without using a different ABI than applications and libraries written in C. It provides a concise way of using GLib and GObject features, but does not attempt to expose all possibilities. In particular, Vala is primarily a statically typed language - this is a design decision, not an oversight.

The only support that Vala applications require at runtime are the standard GLib and GObject libraries. It is possible to use any system library from Vala, provided that a VAPI file is written to describe the interface - Vala is distributed with VAPI descriptions of most libraries commonly used by GNOME applications, and many others as well.

Vala provides easy integration with DBus, by automatically writing boiler plate code where required, for exposing objects, dispatching methods, etc.

1.1 Getting started

The classic "Hello, world" example in Vala:

```
int main (string[] args) {
        stdout.printf ("hello, world\n");
        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`. "valac" is the Vala compiler; it will also allow you to take more control of the compile and link processes when required, but that is outside the scope of this introductory section.

1.2 Documentation conventions

A large amount of this documentation describes the language features precisely using a simple rule notation. The same notation is used to describe language syntax and semantics, with the accompanying text always explaining what is described. The following example shows the form:

rule-name:
    literalstring1
    literalstring2 [optional-section]

optional-section:
    literalstring3

Here, "rule-name" and "optional-section" describe rules, each of which can be expanded in a particular way. Expanding a rule means substituting one of the options of the rule into the place the rule is used. In the example, "optional-section" can be expanded into "literalstring3" or, in "rule-name", "optional-section" can also be substituted for nothing, as it is declared optional by the square brackets. Wherever "rule-name" is required, it can be substituted for either of the options declared in "rule-name". Anything highlighted, such as all **literalstrings** here are not rules, and thus cannot be expanded.

Example code is shown as follows. Example code will always be valid Vala code, but will not necessarily be usable out of context.

```
class MyClass : Object {
        int field = 1;
}
```

Some phrases are used in a specific ways in this documentation and it is often useful to recognise their precise meanings: that is, to create a method, you write a declaration for it. When the program is running and the method exists, it is then defined as per your declaration and can be invoked.

1.3 Vala source files

There are two types of Vala input files. Vala source files (with a ".vala" extension) contain compileable Vala code. VAPI files (with a ".vapi" extension) describe an interface to a library, which can be written in either Vala or C. VAPI files are not compileable, and cannot contain any executable code - they are used when compiling Vala source files.

There are no requirements for how Vala source files are named, although there are conventions that can be followed. VAPI files are usually named to matched the pkg-

config name of the library they relate to; they are described more fully in the documentation about bindings.

All Vala input files should be encoded in UTF-8.

1.4 Vala conventions

The logical structure of a Vala project is entirely based on the program text, not the file layout or naming. Vala therefore does not force particular naming schemes or file layouts. There are established conventions derived from how GNOME related applications are normally written, which are strongly encouraged. The choice of directory structure for a project is outside the scope of this documentation.

Vala source files usually contain one main public class, after which the source file is named. A common choice is to convert this main class' name to lowercase, and prefix with its namespace, also in lower case, to form the file name. In a small project the namespace may be redundant and so excluded. None of this is a requirement, it is just a convention.

It is not encouraged to include declarations in more than one namespace in a single Vala source file, simply for reasons of clarity. A namespace may be divided over any number of source files, but will normally not be used outside of one project. Each library or application will normally have one main namespace, with potentially others nested within.

In source code, the following naming conventions are normally followed:

- Namespaces are named in camel case: NameSpaceName
- Classes are named in camel case: ClassName
- Method names are all lowercase and use underscores to separate words: method_name
- Constants (and values of enumerated types) are all uppercase, with underscores between words: CONSTANT_NAME

Vala supports the notion of a package to conveniently divide program sections. A package is either a combination of an installed system library and its Vala binding, or else is a local directory that can be treated in a similar way. In the latter case it will contain all functionality related to some topic, the scope of which is up to the developer. All source files in package are placed within a directory named for package name. For details on using packages, see the Vala compiler documentation

1.5 Vala syntax

Vala's syntax is modelled on C#'s, and is therefore similar to all C-like languages. Curly braces are the basic delimiter, marking the start and end of a declaration or block of code.

There is no whitespace requirement, though this is a standard format that is used in Vala itself, and in many Vala projects. This format is a version of the coding style used for glib and gnome projects, but is not fully described in this document, other than being used for all examples.

There is flexibility in the order of declarations in Vala. It is not required to pre-declare anything in order to use it before its declaration.

Identifiers all follow the same rules, whether for local variables or class names. Legal identifiers must begin with one alphabetic character or underscore, followed by any number (zero or more) of alphanumerics or underscores (/[:alpha:_]([:alphanum:_])*/). It is also possible to use language keywords as identifiers, provided they are prefixed with a "@" when used in this way - the "@" is not considered a part of the identifier, it simply informs the compiler that the token should be considered as an identifier.

1.6 GType and GObject

Vala uses the runtime type system called GType. This system allows every type in Vala, including the fundamental types, to be identified at runtime. A Vala developer doesn't need to be aware of GType in most circumstances, as all interaction with the system is automatic.

GType provides Vala with a powerful object model called GObject. To all types descended from GLib.Object class, this model provides for features such as properties and signals.

GType and GObject are entirely runtime type systems, intended to be usable to dynamically typed languages. Vala is primarily a statically typed language, and so is designed not to provide access to all of GType and GObject's features. Instead Vala uses a coherent subset to support particular programming styles.

Vala is designed to use GType and GObject seamlessly. There are occasions, mostly when working with existing libraries, when you might need to circumvent parts of the system. These are all indicated in this documentation.

1.7 Memory management

Vala automatically uses the memory management system in GLib, which is a reference counting system. In order for this to work, the types used must support reference counting, as is the case with all GObject derived types and some others.

Memory is allocated and initialised by Vala when needed. The memory management scheme means it is also freed when possible. There is though no garbage collector, and currently reference cycles are not automatically broken. This can lead to memory being leaked. The main way to avoid this problem is to use weak references - these are not counted references and so cannot prevent memory being released, at the cost that they can be left referring to non existent data.

Vala also allows use of pointers in much the same way as C. An instance of a pointer type refers directly to an address in memory. Pointers are not references, and therefore the automatic memory management rules do not apply in the same way. See Types/Pointer types.

There are more details about memory management elsewhere, see Types, see Concepts.

1.8 Vala compilation

Vala programs and libraries are translated into C before being compiled into machine code. This stage is intended to be entirely transparent unless you request otherwise, as such it is not often required to know the details.

When performing a more complicated compile or link process than valac's default, valac can be instructed to simply output its intermediate C form of the program and exit. Each Vala source file is transformed into a C header and a C source file, each having the same name as the Vala source file except for the extension. These C files can be compiled without any help from any Vala utility or library.

The only times it is definitely required to be aware of the translation process is when a Vala feature cannot be represented in C, and so the generated C API will not be the same as the Vala one. For example, private struct members are meaningless in C. These issues are indicated in this documentation.

1.9 Application entry point

All Vala applications are executed beginning with a method called "main". This must be a non-instance method, but may exist inside a namespace or class. If the method takes a string array parameter, it will be passed the arguments given to the program on execution. If it returns an int type, this value will be passed to the user on the program's normal termination. The entry point method may not accept any other parameters, or return any other types, making the acceptable definitions:

```
void main () { ... }
int main () { ... }
void main (string[] args) { ... }
int main (string[] args) { ... }
```

The entry point can be implicit, in the sense that you can write the main code block directly in the file outside the `main` function.

2. Concepts

2.1 Variables, fields and parameters
2.2 Scope and naming
2.3 Object oriented programming
2.4 References and ownership

This pages describes concepts that are widely applicable in Vala. Specific syntax is not described here, see the linked pages for more details.

2.1 Variables, fields and parameters

Any piece of data in a Vala application is considered an instance of a data type. There are various different categories of data types, some being built into Vala, and others being user defined. Details about types are described elsewhere in this documentation, in particular see Types.

Instances of these types are created in various different ways, depending on the type. For example, fundamental types are instantiated with literal expressions, and classed types with the new operator.

In order to access data, the instance must be identifiable in some way, such as by a name. In Vala, there are broadly three ways that this is done, with similar but not identical semantics.

(All these subsections refer to ownership, so it may be useful to read the section on in conjunction with this section)

Variables

Within executable code in a method, an instance may be assigned to a variable. A variable has a name and is declared to refer to an instance of a particular data type. A typical variable declaration would be:

```
int a;
```

This declaration defines that "a" should become an expression that evaluates to an instance of the int type. The actual value of this expression will depend on which int instance is assigned to the variable. "a" can be assigned to more than once, with the most recent assignment being the only one considered when "a" is evaluated. Assignment to the variable is achieved via an assignment expression. Generally, the semantics of an assignment expression depends on the type of the variable.

A variable can take ownership of an instance, the precise meaning of which depends on the data type. In the context of reference types, it is possible to declare that a variable should not ever take ownership of an instance, this is done with the `unowned` keyword. See Types/Reference types.

If a type is directly instantiated in a variable declaration statement, then the variable will be created owning that new instance, for example:

```
string s = "stringvalue";
```

A variable ceases to exist when its scope is destroyed, that is when the code block it is defined in finishes. After this, the name can no longer be used to access the instance, and no new assignment to the variable is allowed. What happens to the instance itself is dependent on the type.

For more details of the concepts in this section, see Statements/Variable declaration and Expressions/Assignment operations.

Fields

A field is similar to a variable, except for the scope that it is defined in. Fields can be defined in namespaces, classes and structs. In the case of classes and structs, they may be either in the scope of the the class or struct, or in the scope of each instance of the class or struct.

A field is valid as long as its scope still exists - for non-instance fields, this is the entire life of the application; for instance fields, this is the lifetime of the instance.

Like variables, fields can take ownership of instances, and it is again possible to avoid this with the `unowned` keyword.

If a type is directly instantiated in the declaration of the field, then that field will be created owning that new instance.

For more details about fields, see Namespaces, Classes and Structs.

Parameters

Instances passed to methods are accessible within that method with names given in the method's parameter list.

They act like variables, except that they cannot, by default, take ownership of the first instance that is assigned to them, i.e. the instance passed to the method. This behaviour can be changed using explicit ownership transfer. When reassigning to a parameter, the result depends on the parameter direction. Assuming the parameter has no direction modifier, it will subsequently act exactly as a variable.

For more details of methods and parameters, see Methods and Expressions/Ownership transfer expressions.

2.2 Scope and naming

A "scope" in Vala refers to any context in which identifiers can be valid. Identifiers in this case refers to anything named, including class definitions, fields, variables, etc. Within a particular scope, identifiers defined in this scope can be used directly:

```
void main () {
        int a = 5;
        int b = a + 1;
}
```

Scopes in Vala are introduced in various different ways.

- Named scopes can be created directly with declarations like namespaces. These are always in existence when the program is running, and can be referred to by their name.
- Transient scopes are created automatically as the program executes. Every time a new code block is entered, a new scope is created. For example, a new scope is created when a method is invoked. There is no way to refer to this type of scope from outside.
- Instance scopes are created when a data type is instantiated, for example when a new instance of a classed type is created. These scopes can be accessed via identifiers defined in other scopes, e.g. a variable to which the new instance is assigned.

To refer to an identifier in another scope, you must generally qualify the name. For named scopes, the scope name is used; for instance scopes, any identifier to which the instance is assigned can be used. See Expressions/Member access for the syntax of accessing other scopes.

Scopes have parent scopes. If an identifier is not recognised in the current scope, the parent scope is searched. This continues up to the the global scope. The parent scope of any scope is inferred from its position in the program - the parent scope can easily be identified as it is the scope the current declaration is in.

For example, a namespace method creates a transient scope when it is invoked - the parent of this scope if the namespace which contains the definition of the method. There are slightly different rules applied when instances are involved, as are described at Classes/Class scope.

The ultimate parent of all other scopes is the global scope. The scope contains the fundamental data types, e.g. int, float, string. If a program has a declaration outside of any other, it is placed in this scope.

Qualifying names

The following rules describe when to qualify names:

- For names in the same scope as the current definition, just the name should be used.
- For names in scopes of which the current is parent, qualify with just the names of scopes that the current definition is not nested within.

- For names in other scopes entirely, or that are less deeply nested than the current, use the fully qualified name (starting from the global scope.)

There are some intricacies of scopes described elsewhere in this documentation. See Classes for how scopes are managed for inherited classes.

Vala will lookup names assuming first that they are not fully qualified. If a fully qualified name can be partially matched locally, or in a parent scope that is not the global scope, the compilation will fail. To avoid problems with this, do not reuse names from the global scope in other scopes.

There is one special scope qualifier that can be used to avoid the problem described in the previous paragraph. Prefixing an identifier with `global::` will instruct the compiler to only attempt to find the identifier in the global scope, skipping all earlier searching.

2.3 Object oriented programming

Vala is primarily an object oriented language. This documentation isn't going to describe object oriented programming in detail, but in order for other sections to make sense, some things need to be explained.

A class in Vala is a definition of a potentially polymorphic type. A polymorphic type is one which can be viewed as more than one type. The basic method for this is inheritance, whereby one type can be defined as a specialized version of another. An instance of a subtype, descended from a particular supertype, has all the properties of the supertype, and can be used wherever an instance of the supertype is expected. This sort of relationship is described as a "subtype instance is-a supertype instance." See Classes.

Vala provides inheritance functionality to any type of class (see Classes/Types of class). Given the following definition, every SubType instance is-a SuperType instance:

```
class SuperType {
        public int act() {
                return 1;
        }
}
class SubType : SuperType {
}
```

Whenever a SuperType instance is required, a SubType instance may be used. This is the extent of inheritance allowed to compact classes, but full classes are more featured. All classes that are not of compact type, can have virtual methods, and can implement interfaces.

To explain virtual functions, it makes sense to look at the alternative first. In the above example, it is legal for SubType to also define a method called "act" - this is called overriding. In this case, when a method called "act" is called on a SubType instance,

which method is invoked depends on what type the invoker believed it was dealing with. The following example demonstrates this:

```
SubType sub = new SubType();
SuperType super = sub;

sub.act();
super.act();
```

Here, when sub.act() is called, the method invoked will be SubType's "act" method. The call super.act() will call SuperType's "act". If the act method were virtual, the SubType.act method would have been called on both occasions. See Classes/Class methods for how to declare virtual methods.

Interfaces are a variety of non-instantiatable type. This means that it is not possible to create an instance of the type. Instead, interfaces are implemented by other types. Instances of these other types may then be used as though they were instances of the interface in question. See Interfaces.

2.4 References and ownership

Type instances in Vala are automatically managed to a large degree. This means that memory is allocated to store the data, and then deallocated when the data is no longer required. However, Vala does not have a runtime garbage collector, instead it applies rules at compile time that will predictably deallocate memory at runtime.

A central concept of Vala's memory management system is ownership. An instance is considered still in use as long as there is at least one way of accessing it, i.e. there is some field, variable or parameter that refers to the instance - one such identifier will be considered the instance's owner, and therefore the instance's memory will not be deallocated. When there is no longer any way to access the data instance, it is considered unowned, and its memory will be deallocated.

Value types

When dealing with instances of value types (see Types) knowledge of ownership is rarely important. This is because the instance is copied whenever it is assigned to a new identifier. This will cause each identifier to become owner of a unique instance - that instance will then be deallocated when the identifier ceases to be valid.

There is one exception to this rule: when a struct type instance is passed to a method, Vala will, by default, create the method parameter as a reference to the instance instead of copying the instance. This reference is a weak reference, as described in the following section. If the struct should instead be copied, and the parameter created as a standard value type identifier, the ownership transfer operator should be used (see Expressions/ Ownership transfer expressions).

Reference types

With reference types, it is possible for several identifiers to reference the same data instance. Not all identifiers that refer to reference type instance are capable of owning the instance, for reasons that will be explained. It is therefore often required to think about instance ownership when writing Vala code.

Most reference types support reference counting. This means that the instance internally maintains a count of how many references to it currently exist. This count is used to decide whether the instance is still in use, or if its memory can be deallocated. Each reference that is counted in this way is therefore a potential owner of the instance, as it ensures the instance continues to exist. There are situations when this is not desired, and so it is possible to define a field or variable as "weak". In this case the reference is not counted, and so the fact that the reference exists will not stop the instance being possibly deallocated, i.e. this sort of reference cannot take ownership of the instance.

When using reference counted types, the main use for weak references is to prevent reference cycles. These exist when a data instance contains internally a reference to another instance, which in turn contains a reference to the first. In this case it would not be possible to deallocate the instances, as each would be potentially owning the other. By ensuring that one of the references is weak, one of the instances can become unowned and be deallocated, and in the process the other will be dereferenced, and potentially become unowned and be deallocated also.

It is also possible to have reference types which are not reference counted; an example of this is the fundamental string type, others are compact classed types. If Vala were to allow several references to own such instances, it would not be able to keep track of when they all ceased to exist, and therefore would not be able to know when to deallocate the instance. Instead, exactly one or zero identifiers will own the instance - when it is zero, the instance is deallocated. This means that all references to an already owned instance must either be weak references, or ownership must be specifically passed to the new reference, using the ownership transfer operator (see Expressions/Ownership transfer expressions).

Pointer types

Pointer types are of great importance. Pointer types are value types, whose instances are references to some other data instance. They are therefore not actual references, and will never own the instance that they indirectly refer to. See Types/Pointer types.

3. Types

3.1 Value types
3.2 Reference types
3.3 Parameterised types
3.4 Nullable types
3.5 Pointer types
3.6 Type conversions

A "type", loosely described, is just an abstract set of 0 or more data fields. A type may be instantiated by creating an entity that contains values that map to the fields of the type. In Vala, a type generally consists of:

- A type name, which is used in various contexts in Vala code to signify an instance of the type.
- A data structure that defines how to represent an instance of the type in memory.
- A set of methods that can be called on an instance of the type.

These elements are combined as the definition of the type. The definition is given to Vala in the form of a declaration, for example a class declaration.

Vala supports three kinds of data types: value types, reference types, and meta 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 there is only ever one variable or field that refers to each instance, whereas variables or fields of the reference types store references to data which can also be referred to by other variable or fields. When two variables or fields of a reference type reference the same data, changes made using one identifier are visible when using the other. This is not possible with value types.

Meta types are created automatically from other types, and so may have either reference or value type semantics.

```
type:
        value-type
        reference-type
        meta-type

meta-type:
        parameterised-type
        nullable-type
```

pointer-type

3.1 Value types

Instances of value types are stored directly in variables or fields that represent them. Whenever a value type instance is assigned to another variable or field, the default action is to duplicate the value, such that each identifier refers to a unique copy of the data, over which it has ownership. When a value type is instantiated in a method, the instance is created on the stack.

value-type:
    fundamental-struct-type
    user-defined-struct-type
    enumerated-type

fundamental-struct-type:
    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
```

Where a literal is indicated, this means the actual type name of a built in struct type is given. The definition of these types is included in Vala, so these types are always available.

Struct types

A struct type is one that provides just a data structure and some methods that act upon it. Structs are not polymorphic, and cannot have advanced features such as signals or properties. See Structs for documentation on how to define structs and more details about them. See Expressions/Struct instantiation for how to instantiate structs.

Each variable or field to which a struct stype instance is assigned gains a copy of the data, over which it has ownership. However, when a struct type instance is passed to a method, a copy is not made. Instead a reference to the instance is passed. This behaviour can be changed by declaring the struct to be a simple type.

Fundamental types

In Vala, the fundamental types are defined as struct types whose data structure is known internally to Vala. They have one anonymous field, which is automatically accessed when required. All fundamental value types are defined as simple types, and so whenever the instance is assigned to a variable or field or passed as a function parameter, a copy of the data is made.

The fundamental value types fall into one of three categories: the boolean type, integral types, and floating point types.

Integral types

Integral types can contain only integers. They are either signed or unsigned, each of which is considered a different type, though it is possible to cast between them when needed.

Some types define exactly how many bits of storage are used to represent the integer, others depend of the environment. long, int short map to C data types and therefore depend on the machine architecture. char is 1 byte. unichar is 4 bytes, i.e. large enough to store any UTF-8 character.

All these types can be instantiated using a literal expression, see Expressions/Literal expressions.

Floating point types

Floating point types contain real floating point numbers in a fixed number of bits (see IEEE 754).

All these types can be instantiated using a literal expression, see Expressions/Literal expressions.

The bool type

Can have values of true of false. Although there are only two values that a bool instance can take, this is not an enumerated type. Each instance is unique and will be copied when required, the same as for the other fundamental value types.

This type can be instantiated using literal expressions, see Expressions/Literal expressions.

Enumerated types

An enumerated type is one in which all possible values that instances of the type can hold are declared with the type. In Vala enumerated types are real types, and will not be implicitly converted. It is possible to explicitly cast between enumerated types, but this is not generally advisable. When writing new code in Vala, don't rely on being able to cast in this way.

A variation on an enumerated type is a flag type. This represents a set of flags, any number of which can be combined in one instance of the flag type, in the same fashion as a bitfield in C.

See Enumerated types (Enums) for documentation on defining and using enumerated types.

3.2 Reference types

Instances of reference types are always stored on the heap. Variables of reference types contain references to the instances, rather than the instances themselves. Assigning an instance of a reference type to a variable or field will not make a copy of the data, instead only the reference to the data is copied. This means that both variables will refer to the same data, and so changes made to that data using one of the references will be visible when using the other.

Instances of any reference type can be assigned a variable that is declared "weak". This implies that the variable must not be known to the type instance. A reference counted type does not increase its reference count after being assigned to a weak variable: a weak variable cannot take ownership of an instance.

```
reference-type:
    classed-type
    array-type
    delegate-type
    error-type
```

```
    string

classed-type:
    simple-classed-type
    type-instance-classed-type
    object-classed-type

simple-classed-type:
    user-defined-simple-classed-type

type-instance-classed-type:
    user-defined-type-instance-classed-type

object-classed-type:
    user-defined-object-classed-type

array-type:
    non-array-type [ ]
    non-array-type [   dimension-separators ]

non-array-type:
    value-type
    classed-type
    delegate-type
    error-type

dimension-separators:
    ,
    dimension-separators ,

delegate-type:
    user-defined-delegate-type

error-type:
    user-defined-error-type
```

Classed types

A class definition introduces a new reference type - this is the most common way of creating a new type in Vala. Classes are a very powerful mechanism, as they have features such as polymorphism and inheritance. Full discussion of classes is found at Classes.

Most classed types in Vala are reference counted. This means that every time a classed type instance is assigned to a variable or field, not only is the reference copied, but the instance also records that another reference to it has been created. When a field or variable goes out of scope, the fact that a reference to the instance has been removed is also recorded. This means that a classed type instance can be automatically removed from memory when it is no longer needed. The only classed types that are not reference counted are compact classes.. Memory management is discussed at Overview/Memory management. If the instance is not of a reference counted type, then the ownership must be explicitly transferred using the # operator - this will cause the original variable to become invalid. When a classed-type instance is passed to a method, the same rules apply. The types of classes available are discussed at Classes/Types of class.

Array types

TODO: Check correctness.

An array is a data structure that can contains zero or more elements of the same type, up to a limit defined by the type. An array may have multiple dimensions; for each possible set of dimensions a new type is implied, but there is a meta type available that describes an array of any size with the same number of dimensions, i.e. int[1] is not the same type as int[2], while int[] is the same type as either.

A size can be retrieved from an array using the `length` member, this returns an int if the array has one dimension or an int[] if the array contains several dimensions.

You can also move or copy and array using respectively the `move` and `copy` members.

For single-dimension arrays, a `resize` member is also available to change the length of the array.

See Expressions/Array instantiation for how to instantiate an array type.

Delegate types

A delegate is a data structure that refers to a method. A method executes in a given scope which is also stored, meaning that for instance methods a delegate will contain also a reference to the instance.

Delegates are technically a referenced type, but since methods are immutable, this distinction is less important than for other types. Assigning a delegate to a variable or field cannot copy the method indicated, and no delegate is able to change the method in any way.

See Delegates for full documentation.

Error Types

Instances of error types represent recoverable runtime errors. All errors are described using error domains, a type of enumerated value, but errors themselves are not

enumerated types. Errors are discussed in detail in several sections of this documentation, see: Errors, Enumerated types (Enums)/Error domains and Methods.

Strings

Vala has built in support for Unicode strings, via the fundamental string type. This is the only fundamental type that is a reference type. Like other fundamental types, it can be instantiated with a literal expression (Expressions/Literal expressions.) Strings are UTF-8 encoded, the same as Vala source files, which means that they cannot be accessed like character arrays in C - each Unicode character is not guaranteed to be stored in just one byte. Instead the string fundamental struct type (which all strings are instances of) provides access methods along with other tools.

While strings are technically a reference type, they have the same default copy semantics as structs - the data is copied whenever a string value is assigned to a variable or field, but only a reference is passed as a parameter to a method. This is required because strings are not reference counted, and so the only way for a variable or field to be able to take ownership of a string is by being assigned a copy of the string. To avoid this behaviour, string values can be assigned to weak references (in such a case no copy is made).

The concept of ownership is very important in understanding string semantics. For more details see Concepts/References and ownership.

3.3 Parameterised types

TODO: Casting.

Vala allows definitions of types that can be customised at runtime with type parameters. For example, a list can be defined so that it can be instantiated as a list of ints, a list of Objects, etc. This is achieved using generic declarations. See Generics.

3.4 Nullable types

The name of a type can be used to implicitly create a nullable type related to that type. An instance of a nullable type `T?` can either be a value of type `T` or `null`.

A nullable type will have either value or reference type semantics, depending on the type it is based on.

3.5 Pointer types

The name of a type can be used to implicitly create a pointer type related to that type. The value of a variable declared as being of type T* represents the memory address of an instance of type T. The instance is never made aware that its address has been recorded, and so cannot record the fact that it is referred to in this way.

Instances of any type can be assigned to a variable that is declared to be a pointer to an instance of that type. For referenced types, direct assignment is allowed in either direction. For value types the pointer-to operator "&" is required to assign to a pointer, and the pointer-indirection operator "*" is used to access the instance pointed to. See Expressions/Pointer expressions.

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. See Expressions/Type operations.

A pointer type itself has value type semantics.

3.6 Type conversions

There are two types if type conversions possible in Vala, implicit conversions and explicit casts. In expressions, Vala will often convert fundamental types in order to make calculations possible. When the default conversion is not what you require, you can cast explicitly so that all operands are of compatible types. See Expressions for details of automatic conversions.

Vala will also automatically perform conversions related to polymorphism where the required cast is unambiguous and can be inferred from the context. This allows you to use a classed-type instance when an instance of any of its superclasses or implemented interfaces is required. Vala will never automatically cast to a subtype, as this must be done explicitly. See Concepts/Object oriented programming, see Classes.

For explicit casting expressions, see Expressions/Type operations.

4. Expressions

4.1 Literal expressions
4.2 Member access
4.3 Element access
4.4 Arithmetic operations
4.5 Relational operations
4.6 Increment/decrement operations
4.7 Logical operations
4.8 Bitwise operations
4.9 Assignment operations
4.10 Invocation expressions
4.11 Class instantiation
4.12 Struct instantiation
4.13 Array instantiation
4.14 Conditional expressions
4.15 Coalescing expressions
4.16 Flag operations
4.17 Type operations
4.18 Ownership transfer expressions
4.19 Lambda expressions
4.20 Pointer expressions

Expressions are short pieces of code that define an action that should be taken when they are reached during a program's execution. Such an operation can be arithmetical, calling a method, instantiating a type, and so on. All expressions evaluate to a single value of a particular type - this value can then be used in another expression, either by combing the expressions together, or by assigning the value to an identifier.

When expressions are combined together (e.g. add two numbers, then multiply the result by another: 5 + 4 * 3), then the order in which the sub-expressions are evaluated becomes significant. Parentheses are used to mark out which expressions should be nested within others, e.g. (5 + 4) * 3 implies the addition expression is nested inside the multiplication expression, and so must be evaluated first.

When identifiers are used in expressions they evaluate to their value, except when used in assignment. The left handed side of an assignment are a special case of expressions where an identifier is not considered an expression in itself and is therefore not evaluated. Some operations combine assignment with another operation (e.g. increment operations,) in which cases an identifier can be thought of as an expression initially, and then just an identifier for assignment after the overall expression has been evaluated.

```
primary-expression:
    literal
    template
    member-access-expression
```

```
        pointer-member-access-expression
        element-access-expression
        postfix-expression
        class-instantiation-expression
        array-instantiation-expression
        struct-instantiation-expression
        invocation-expression
        sizeof-expression
        typeof-expression

    unary-expression:
        primary-expression
        sign-expression
        logical-not-expression
        bitwise-not-expression
        prefix-expression
        ownership-transfer-expression
        cast-expression
        pointer-expression

    expression:
        conditional-expression
        assignment-expression
        lambda-expression
```

4.1 Literal expressions

Each literal expression instantiates its respective type with the value given.

Integer types... -?[:digit:]+

Floating point types... -?[:digit:]+(.[:digit:]+)?

Strings... "[^"\n]*". """.*"""

Booleans... true|false

A final literal expression is `null`. This expression evaluates to a non-typed data instance, which is a legal value for any nullable type (see Types/Nullable types.)

4.2 Member access

To access members of another scope.

member-access-expression:
    [primary-expression **.**] identifier

If no inner expression is supplied, then the identifier will be looked up starting from the current scope (for example a local variable in a method). Otherwise, the scope of the inner expression will be used. The special identifier **this** (without inner expression) inside an instance method will refer to the instance of the type symbol (class, struct, enum, etc.).

4.3 Element access

element-access-expression:
    container **[** indexes **]**

container:
    expression

indexes:
    expression [**,** indexes]

Element access can be used for:

- Accessing an element of a container at the given indexes
- Assigning an element to a container at the given indexes. In this case the element access expression is the left handed side of an assignment.

Element access can be used on strings, arrays and types that have **get** and/or **set** methods.

- On strings you can only access characters, it's not possible to assign any value to an element.
- On arrays, it's possible to both access an element or assign to an element. The type of the element access expression is the same as the array element type.

Element access can also be used with complex types (such as class, struct, etc.) as containers:

- If a **get** method exists accepting at least one argument and returning a value, then indexes will be used as arguments and the return value as element.

- If a **set** method exists accepting at least two arguments and returns **void**, then indexes will be used as arguments and the assigned value as last argument..

4.4 Arithmetic operations

Binary operators, taking one argument on each side. Each argument is an expression returning an appropriate type.

Applicable, unless said otherwise, where both operands evaluate to numeric types (integer or floating point).

Where at least one operand is a of floating point type, the result will be the same type as the largest floating point type involved. Where both operands are of integer types, the result will have the same type as the largest of the integer types involved.

Addition/Subtraction:

```
additive-expression:
    multiplicative-expression
    multiplicative-expression + multiplicative-expression
    multiplicative-expression - multiplicative-expression

sign-expression:
    + unary-expression
    - unary-expression
```

Adds/Subtracts the second argument to/from the first. Negations is equivalent to subtraction the operand from 0.

Overflow?

Multiplication/Division:

```
multiplicative-expression:
    unary-expression
    unary-expression * unary-expression
    unary-expression / unary-expression
    unary-expression % unary-expression
```

Multiplies/divides the first argument by the second.

If both operands are of integer types, then the result will be the quotient only of the calculation (equivalent to the precise answer rounded down to an integer value.) If either operand is of a floating point type, then the result will be as precise as possible within the boundaries of the result type (which is worked out from the basic arithmetic type rules.)

4.5 Relational operations

Result in a value of bool type.

Applicable for comparing two instances of any numeric type, or two instances of string type. Where numeric with at least one floating point type instance, operands are both converted to the largest floating point type involved. Where both operands are of integer type, both are converted to the largest integer type involved. When both are strings, they are lexically compared somehow.

```
equality-expression:
        relational-expression
        relational-expression == relational-expression
        relational-expression != relational-expression

relational-expression:
        shift-expression
        shift-expression <  relational-expression
        shift-expression <= relational-expression
        shift-expression >  relational-expression
        shift-expression >= relational-expression
        is-expression
        as-expression
```

4.6 Increment/decrement operations

```
postfix-expression:
        primary-expression ++
        primary-expression --

prefix-expression:
        ++ unary-expression
        -- unary-expression
```

Postfix and prefix expressions:

```
var postfix = i++;
var prefix = --j;
```

are equivalent to:

```
var postfix = i;
i += 1;

j -= 1;
var prefix = j;
```

4.7 Logical operations

Applicable to boolean type operands, return value is of boolean type. No non boolean type instances are automatically converted.

logical-or-expression:
    logical-and-expression **||** logical-and-expression

Documentation

logical-and-expression:
    contained-in-expression **&&** contained-in-expression

Documentation

logical-not-expression:
    ! expression

4.8 Bitwise operations

All only applicable to integer types.

```
bitwise-or-expression:
    bitwise-xor-expression | bitwise-xor-expression

bitwise-xor-expression:
    bitwise-and-expression ^ bitwise-and-expression

bitwise-and-expression:
    equality-expression & equality-expression

bitwise-not-expression:
    ~ expression
```

Documentation

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

Shifts the bits of the left argument left/right by the number represented by the second argument.

Undefined for shifting further than data size, e.g. with a 32 bit integer...

Documentation

4.9 Assignment operations

Value assigned to identifier on left. Type must match.

When assignment includes another operation natural result type must match the declared type of variable which is the left hand side of the expression. e.g. Let a be an int instance with the value 1, a += 0.5 is not allowed, as the natural result type of 1 + 0.5 is a float, not an int.

```
assignment-expression:
    simple-assignment-expression
    number-assignment-expression
```

simple-assignment-expression:

conditional-expression **=** expression

number-assignment-expression:

conditional-expression **+=** expression
conditional-expression **-=** expression
conditional-expression ***=** expression
conditional-expression **/=** expression
conditional-expression **%=** expression
conditional-expression **|=** expression
conditional-expression **&=** expression
conditional-expression **^=** expression
conditional-expression **<<=** expression
conditional-expression **>>=** expression

A simple assignment expression assigns the right handed side value to the left handed side. It is necessary that the left handed side expression is a valid lvalue. Other assignments:

```
result += value;
result <<= value;
...
```

Are equivalent to simple assignments:

```
result = result + value;
result = result << value;
...
```

4.10 Invocation expressions

invocation-expression:

[**yield**] primary-expression **(** [arguments] **)**

arguments:

```
    expression [ , arguments]
```

The expression can refer to any callable: a method, a delegate or a signal. The type of the expression depends upon the return type of the callable symbol. Each argument expression type must be compatible against the respective callable parameter type. If an argument is not provided for a parameter then:

If the callable has an ellipsis parameter, then any number of arguments of any type can be provided past the ellipsis.

Delegates... See Delegates

Firing a signal is basically the same. See Classes/Signals

4.11 Class instantiation

To instantiate a class (create an instance of it) use the new operator. This operator takes a the name of the class, and a list of zero or more arguments to be passed to the creation method.

```
class-instantiation-expression:
    new type-name ( arguments )

arguments:
    expression [ , arguments ]
```

4.12 Struct instantiation

```
struct-instantiation-expression:
    type-name ( arguments ) [ { initializer } ]

initializer:
    field-name = expression [ , initializer ]

arguments:
    expression [ , arguments ]
```

4.13 Array instantiation

This expression will create an array of the given size. The second approach shown below is a shorthand to the first one.

array-instantiation-expression:
    new type-name **[** sizes **]** [**{** [initializer] **}**]
    { initializer **}**

sizes:
    expression [**,** sizes]

initializer:
    expression [**,** initializer]

Sizes expressions must evaluate either to an integer type or an enum value. Initializer expressions type must be compatible with the array element type.

4.14 Conditional expressions

Allow a conditional in a single expression.

conditional-expression:
    boolean-expression [**?** conditional-true-clause **:** conditional-false-clause]

boolean-expression:
    coalescing-expression

conditional-true-clause:
    expression

conditional-false-clause
    expression

First boolean-expression is evaluated. If true, then the conditional-true-clause is evaluated, and its result is the result of the conditional expression. If the boolean expression evaluates to false, then the conditional-false-clause is evaluated, and its result becomes the result of the conditional expression.

4.15 Coalescing expressions

coalescing-expression:
    nullable-expression [**??** coalescing-expression]

nullable-expression:
    logical-or-expression

4.16 Flag operations

Flag types are a variation on enumerated types, in which any number of flag values can be combined in a single instance of the flag type. There are therefore operations available to combine several values in an instance, and to find out which values are represented in an instance.

flag-combination-expression:
    expression **|** expression

Where both expressions evaluate to instances of the same flag type, the result of this expression is a new instance of the flag type in which all values represented by either operand are represented.

flag-recombination-expression:
    expression **^** expression

Where both expressions evaluate to instances of the same flag type, the result of this expression is a new instance of the flag type in which all values represented by exactly one of the operands are represented.

flag-separation-expression:
    expression **&** expression

Where both expressions evaluate to instances of the same flag type, the result of this expression is a new instance of the flag type in which all values represented by both operands are represented.

flag-in-expression:
    expression **in** expression

Where both expressions evaluate to instances of the same flag type, the result of this expression is a boolean. The result will be true if the left-handed flag is set into the right-handed flags.

4.17 Type operations

is-expression:
    shift-expression **is** type-name

Performs a runtime type check on the instance resulting from evaluating the the nested expression. If the instance is an instance of the type described (with, for example, a class or interface name,) the overall expression evaluates to true.

Casting:

cast-expression:
    (!) unary-expression
    (type-name **)** unary-expression

A cast expression returns the instance created in the nested expression as an instance of the type described. If the nested expression evaluates to an instance of a type that is not also an instance of the given type, the expression is not valid. If you are not sure whether the cast is valid, instead use an "as" expression.

as-expression:
    shift-expression **as** type-name

An "as" expression combines an "is" expression and a cast operation, with the latter depending on the former. If the nested expression evaluates to an instance of the given type, then a cast is performed and the expression evaluates to the result of the nested expression cast as the given type. Otherwise, the result is null.

sizeof-expression:
    sizeof (type-name **)**

typeof-expression:
    typeof (type-name **)**

4.18 Ownership transfer expressions

ownership-transfer-expression:
    (owned) unary-expression

When an instance of a reference type is assigned to a variable or field, it is possible to request that the ownership of the instance is passed to the new field or variable. The precise meaning of this depends on the reference type, for an explanation of ownership, see Concepts/References and ownership. The identifier in this expression must refer to an instance of a reference type.

Note that similar syntax is used to define that a method parameter should take ownership of a value assigned to it. For this, see Methods.

4.19 Lambda expressions

lambda-expression:
    params **=>** body

params:
    [direction] identifier
    ([param-names] **)**

param-names:
    [direction] identifier [**,** param-names]

direction:
    out
    ref

body:

statement-block
expression

4.20 Pointer expressions

addressof-expression:
& unary-expression

The "address of" expression evaluates to a pointer to the inner expression. Valid inner expressions are:

- Variables (local variables, fields and parameters)
- Element access whose container is an array or a pointer

pointer-indirection-expression:

The pointer indirection evaluates to the value pointed to by the inner expression. The inner expression must be a valid pointer type and it must not be a pointer to a reference type (for example pointer indirection to a type `SomeClass*` is not possible).

pointer-member-access-expression:
primary-expression **->** identifier

This expression evaluates to the value of the member identified by the identifier. The inner expression must be a valid pointer type and the member must be in the scope of the base type of the pointer type.

5. Statements

5.1 Simple statements
5.2 Variable declaration
5.3 Selection statements
5.4 Iteration statements
5.5 Jump Statements
5.6 Try Statement
5.7 Lock Statement
5.8 Unlock Statement
5.9 With Statement

Statements define the path of execution within methods and similar constructions. They combine expressions together with structures for choosing between different code paths, repeating code sections, etc.

```
statement:
    empty-statement
    simple-statement
    statement-block
    variable-declaration-statement
    if-statement
    switch-statement
    while-statement
    do-statement
    for-statement
    foreach-statement
    return-statement
    throw-statement
    try-statement
    lock-statement
    unlock-statement
    with-statement

embedded-statement:
    statement
```

5.1 Simple statements

The Empty Statement does nothing, but is a valid statement nonetheless, and so can be used wherever a statement is required.

```
empty-statement:
    ;
```

A Simple Statement consists of one a subset of expressions that are considered free-standing. Not all expressions are allowed, only those that potentially have a useful side effect - for example, arithmetic expressions cannot form simple statements on their own, but are allowed as part of an assignment expressions, which has a useful side effect.

```
simple-statement:
    statement-expression ;

statement-expression:
    assignment-expression
    class-instantiation-expression
    struct instantiation-expression
    invocation-expression
```

A Statement Block allows several statements to be used in a context that would otherwise only allow one.

```
statement-block:
    {  [ statement-list ] }

statement-list:
    statement [ statement-list ]
```

Blocks create anonymous, transient scopes. For more details about scopes, see Concepts/ Scope and naming.

5.2 Variable declaration

Variable Declaration Statements define a local variable in current scope. The declaration includes a type, which signifies the variable will represent an instance of that type. Where

the type can be inferred by the compiler, the type-name can be replaced with the literal "var"

```
variable-declaration-statement:
        variable-declaration-with-explicit-type
        variable-declaration-with-explicit-type-and-initialiser
        variable-declaration-with-type-inference

variable-declaration-with-explicit-type:
        type-name identifier ;

variable-declaration-with-explicit-type-and-initialiser:
        type-name identifier =  expression ;

variable-declaration-with-type-inference:
        var identifier =  expression ;
```

Type inference is possible in any case where the variable is immediately assigned to. The type chosen will always be the type of the assigned expression, as decided by the rules described at Expressions. It is important to realise that the type of the variable will be fixed after the first assignment, and will not change on assigning another value to the variable. If the variable should be created with a type other than that of the assigned expression, the expression should be wrapped with a cast expression, provided that the cast is valid.

5.3 Selection statements

The If Statement decides whether to execute a given statement based on the value of a boolean expression. There are two possible extensions to this model:

An else clause declares that a given statement should be run if-and-only-if the condition in the the if statement fails.

Any number of "else if" clauses may appear between the "if" statement and its "else" clause (if there is one.) These are equivalent to:

FIXME: This doesn't work.

In simple terms, the program will test the conditions of the if statement and its "else if" clauses in turn, executing the statement belonging to the first that succeeds, or running the else clause if every condition fails.

if-statement:
    `if (` boolean-expression **`)`** embedded-statement [elseif-clauses] [**`else`** embedded-statement]

elseif-clauses:
    elseif-clause
    [elseif-clauses]

elseif-clause:
    `else if (` boolean-expression **`)`** embedded-statement

The switch statement decides which of a set of statements to execute based on the value of an expression. A switch statement will lead to the execution of one or zero statements. The choice is made by:

switch-statement:
    `switch (` expression **`) {`** [case-clauses] [default-clause] **`}`**

case-clauses:
    case-clause
    [case-clauses]

case-clause:
    `case` literal-expression **`:`** embedded-statement

default-clause:
    `default :` embedded-statement

5.4 Iteration statements

Iteration statements are used to execute statements multiple times based on certain conditions. Iteration Statements contain loop embedded statements - a superset of embedded statements which adds statements for manipulating the iteration.

loop-embedded-statement:
    loop-embedded-statement-block

embedded-statement
break-statement
continue-statement

loop-embedded-statement-block:
{ [loop-embedded-statement-list] **}**

loop-embedded-statement-list:
loop-embedded-statement [loop-embedded-statement-list]

Both break and continue statement are types of jump statement, described in .

The While Statement

The `while` statement conditionally executes an embedded statement zero or more times. When the while statement is reached, the boolean expression is executed. If the boolean value is true, the embedded statement is executed and execution returns to the `while` statement. If the boolean value is false, execution continues after the `while` statement.

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

The `do` statement conditionally executes an embedded statement one or more times. First the embedded statement is executed, and then the boolean expression is evaluated. If the boolean value is true, execution returns to the `do` statement. If the boolean value is false, execution continues after the `do` statement.

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

The For Statement

The `for` statement first evaluates a sequence of initialization expressions and then repeatedly executes an embedded statement. At the start of each iteration a boolean expression is evaluated, with a true value leading to the execution of the embedded statement, a false value leading to execution passing to the first statement following the `for` statement. After each iteration a sequence of iteration expressions are evaluated.

Executing this type of statement creates a new transient scope, in which any variables declared in the initializer are created.

for-statement:
    for ([for-initializer] **;** [for-condition] **;** [for-iterator] **)** loop-embedded-statement

for-initializer:
    variable-declaration [**,** expression-list]

for-condition:
    boolean-expression

for-iterator:
    expression-list

The Foreach Statement

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection. Each element in turn is assigned to a variable with the given identifier and the embedded statement is executed. Executing this type of statement creates a new transient scope in which the variable representing the collection element exists.

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

Foreach Statements are able to iterate over arrays and any class that implements the `Gee.Iterable` interface. This may change in future if an Iterable interface is incorporated into GLib.

5.5 Jump Statements

Jump statements move execution to an arbitrary point, dependent on the type of statement and its location. In any of these cases any transient scopes are ended appropriately: Concepts/Scope and naming and .

A `break` statement moves execution to the first statement after the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

break-statement:
break ;

A `continue` statement immediately moves execution the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

continue-statement:
continue ;

The `return` statement ends the execution of a method, and therefore completes the invocation of the method. The invocation expression has then been fully evaluated, and takes on the value of the expression in the `return` statement if there is one.

return-statement:
return [expression] **;**

The throw statement throws an exception.

throw-statement:
throw expression **;**

5.6 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.

For the syntax of the try statement, See Errors/Error catching.

5.7 Lock Statement

`lock` statements are the main part of Vala's resource control mechanism.

FIXME: Haven't actually written anything here about resource control.

lock-statement:
    `lock (` identifier **`)`** [embedded-statement] **`;`**

5.8 Unlock Statement

`unlock` statements are the main part of Vala's resource control mechanism.

FIXME: Haven't actually written anything here about resource control.

unlock-statement:
    `unlock (` identifier **`) ;`**

5.9 With Statement

The `with` statement creates data type scoped blocks which allow implicit member access to the given expression or declaration statement.

with_statement:
    `with (` [var | unowned var | type-name) identifier **`=`**] expression **`)`**
embedded_statement

6. Namespaces

6.1 The global namespace
6.2 Namespace declaration
6.3 Members
6.4 Fields
6.5 Constants
6.6 The "using" statement

Namespaces are named scopes (see Concepts/Scope and naming). Definitions in different namespaces can use the same names without causing conflicts. A namespace can be declared across any number of Vala source files, and there can be multiple namespaces defined in a single Vala source file. Namespaces can be nested to any depth.

When code needs to access definitions from other namespaces, it must either refer to them using a fully qualified name, or be written in a file with an appropriate using statement.

The simplest namespace declaration looks like this:

```
namespace NameSpaceName {
}
```

Namespace nesting is achieved either by nesting the declarations, or by providing multiple names in one declaration:

```
namespace NameSpaceName1 {
        namespace NameSpaceName2 {
        }
}

namespace NameSpaceName1.NameSpaceName2 {
}
```

6.1 The global namespace

Everything not declared within a particular namespace declaration is automatically in the global namespace. All defined namespaces are nested inside the global namespace at some depth. This is also where the fundamental types are defined.

If there is ever a need to explicitly refer to an identifier in the global namespace, the identifier can be prefixed with `global::`. This will allow you, for example, to refer to a namespace which has the same name as a local variable.

6.2 Namespace declaration

namespace-declaration:
    `namespace` qualified-namespace-name **`{`** [namespace-members] **`}`**

qualified-namespace-name:
    [qualified-namespace-name **`.`**] namespace-name

namespace-name:
    identifier

namespace-members:
    namespace-member [namespace-members]

namespace-member:
    class-declaration
    abstract-class-declaration
    constant-declaration
    delegate-declaration
    enum-declaration
    errordomain-declaration
    field-declaration
    interface-declaration
    method-declaration
    namespace-declaration
    struct-declaration

6.3 Members

Namespaces members exist in the namespace's scope. They fall into two broad categories: data and definitions. Data members are fields which contain type instances. Definitions are things that can be invoked or instantiated. Namespace members can be declared either private or public. Public data can be accessed from anywhere, whilst private data can only be accessed from inside the namespace. Public definitions are visible to code defined in a different namespace, and thus can be invoked or instantiated from anywhere, private definitions are only visible to code inside the namespace, and so can only be invoked or instantiated from there.

access-modifier:
    `public`

```
    private
```

For the types of namespace members that are not described on this page: see Classes, Structs, Delegates, Enumerated types (Enums), and Enumerated types (Enums)/Error domains.

6.4 Fields

Variables that exist directly in a namespace are known as namespace fields. These exist only once, and within the scope of the namespace which exists for the application's entire run time. They are therefore similar to global variables in C but without the risk of naming clashes.

```
field-declaration:
    [ access-modifier ] qualified-type-name field-name [ = expression ] ;

field-name:
    identifier
```

Fields in general are described at Concepts/Variables, fields and parameters.

6.5 Constants

Constants are similar to variables but can only be assigned to once. It is therefore required that the expression that initialises the constant be executable at the time the constant comes into scope. For namespaces this means that the expressions must be evaluable at the beginning of the application's execution.

```
constant-declaration:
    [ access-modifier ] const qualified-type-name constant-name =
expression ;

constant-name:
    identifier
```

6.6 The "using" statement

`using` statements can be used to avoid having to qualify names fully on a file-by-file basis. For all identifiers in the same file as the using statement, Vala will first try to

resolve them following the usual rules (see Concepts/Scope and naming). If the identifier cannot be resolved in any scope, each namespace that is referenced in a `using` will be searched in turn.

using-statement:
    `using` namespace-list **`;`**

namespace-list:
    qualified-namespace-name [**`,`** namespace-list]

There can be any number of using statements in a Vala source file, but they must all appear outside any other declarations. Note that `using` is not like import statements in other languages - it does not load anything, it just allows for automatic searching of namespace scopes, in order to allow frugal code to be written.

Most code depends on members of the GLib namespace, and so many source files begin with:

```
using GLib;
```

TODO: Include examples.

7. Methods

7.1 Parameter directions
7.2 Method declaration
7.3 Invocation
7.4 Scope
7.5 Lambdas
7.6 Contract programming

TODO: Do we really need this discussion? Are we introducing Vala, or general programming?

A method is an executable statement block that can be identified in one or more ways (i.e. by a name, or any number of delegate instances). A method can be invoked with an optional number of parameters, and may return a value. When invoked, the method's body will be executed with the parameters set to the values given by the invoker. The body is run in sequence until the end is reached, or a return statement is encountered, resulting in a return of control (and possibly some value, in the case of a return) to the invoker.

There are various contexts that may contain method declarations (see Namespaces, Classes, Interfaces, Structs). A method is always declared inside one of these other declarations, and that declaration will mark the parent scope that the method will be executed within. See Concepts/Scope and naming.

The Classes section of this documentation talks about both methods and abstract methods. It should be noted that the latter are not truly methods, as they cannot be invoked. Instead, they provide a mechanism for declaring how other methods should be defined. See Classes for a description of abstract methods and how they are used.

The syntax for invoking a method is described on the expressions page (see Expressions/Invocation expressions).

7.1 Parameter directions

The basics of method parameter semantics are described on the concepts page (see Concepts/Variables, fields and parameters). This basic form of parameter is technically an "in" parameter, which is used to pass data needed for the method to operate. If the parameter is of a reference type, the method may change the fields of the type instance it receives, but assignments to the parameter itself will not be visible to the invoking code. If the parameter is of a value type, which is not a fundamental type, the same rules apply as for a reference type. If the parameter is of a fundamental type, then the parameter will contain a copy of the value, and no changes made to it will be visible to the invoking code.

If the method wishes to return more than one value to the invoker, it should use "out" parameters. Out parameters do not pass any data to the method - instead the method may assign a value to the parameter that will be visible to the invoking code after the method has executed, stored in the variable passed to the method. If a method is invoked

passing a variable which has already been assigned to as an out parameter, then the value of that variable will be dereferenced or freed as appropriate. If the method does not assign a value to the parameter, then the invoker's variable will end with a value of "null".

The third parameter type is a "ref" argument (equivalent to "inout" in some other languages.) This allows the method to receive data from the invoker, and also to assign another value to the parameter in a way that will be visible to the invoker. This functions similarly to "out" parameters, except that if the method does not assign to the parameter, the same value is left in the invoker's variable.

7.2 Method declaration

The syntax for declaring a method changes slightly based on what sort of method is being declared. This section shows the form for a namespace method, Vala's closest equivalent to a global method in C. Many of the parts of the declaration are common to all types, so sections from here are referenced from class methods, interface methods, etc.

method-declaration:
    [access-modifier] return-type qualified-method-name ([params-list]) [**throws** error-list] method-contracts **{** statement-list **}**

return-type:
    type
    void

qualified-method-name:
    [qualified-namespace-name **.**] method-name

method-name:
    identifier

params-list:
    parameter [**,** params-list]

parameter:
    [parameter-direction] type identifier

parameter-direction:
    ref
    out

error-list:

```
    qualified-error-domain [ , error-list ]

method-contracts:
    [ requires ( expression ) ] [ ensures ( expression ) ]
```

For more details see , and Errors.

7.3 Invocation

See Expressions/Invocation expressions.

7.4 Scope

The execution of a method happens in a scope created for each invocation, which ceases to exist after execution is complete. The parent scope of this transient scope is always the scope the method was declared in, regardless of where it is invoked from.

Parameters and local variables exist in the invocation's transient scope. For more on scoping see Concepts/Scope and naming.

7.5 Lambdas

As Vala supports delegates, it is possible to have a method that is identified by a variable (or field, or parameter.) This section discusses a Vala syntax for defining inline methods and directly assigning them to an identifier. This syntax does not add any new features to Vala, but it is a lot more succinct than the alternative (defining all methods normally, in order to assign them to variables at runtime). See Delegates.

Declaring an inline method must be done with relation to a delegate or signal, so that the method signature is already defined. Parameter and return types are then learned from the signature. A lambda definition is an expression that returns an instance of a particular delegate type, and so can be assigned to a variable declared for the same type. Each time that the lambda expression is evaluated it will return a reference to exactly the same method, even though this is never an issue as methods are immutable in Vala.

```
lambda-declaration:
    ( [ lambda-params-list ] ) => { statement-list }

lambda-params-list:
    identifier [ , lambda-params-list ]
```

An example of lambda use:

```
delegate int DelegateType (int a, string b);

int use_delegate (DelegateType d, int a, string b) {
        return d (a, b);
}

int make_delegate () {
        DelegateType d = (a, b) => {
                return a;
        };
        use_delegate(d, 5, "test");
}
```

7.6 Contract programming

Vala supports basic contract programming features. A method may have preconditions (`requires`) and postconditions (`ensures`) that must be fulfilled at the beginning or the end of a method respectively:

```
double method_name (int x, double d)
                requires (x > 0 && x < 10)
                requires (d >= 0.0 && d <= 1.0)
                ensures (result >= 0.0 && result <= 10.0)
{
        return d * x;
}
```

`result` is a special variable representing the return value.

For example, if you call `method_name` with arguments `5` and `3.0`, it will output a CRITICAL message and return 0.

```
void main () {
        stdout.printf ("%i\n", method_name (5, 3.0));
}
```

Output:

```
CRITICAL **: 03:29:00.588: method_name: assertion 'd >= 0.0 && d <= 1
0
```

Vala allows you to manage the safety of issued messages at 6 levels: ERROR, CRITICAL, INFO, DEBUG, WARNING, MESSAGE. For example, the following code will cause a runtime error.

```
Log.set_always_fatal (LogLevelFlags.LEVEL_CRITICAL |
LogLevelFlags.LEVEL_WARNING);
stdout.printf ("%i\n", method_name (5, 3.0));
```

8. Delegates

8.1 Types of delegate
8.2 Delegate declaration
8.3 Using delegates
8.4 Examples

A delegate declaration defines a method type: a type that can be invoked, accepting a set of values of certain types, and returning a value of a set type. In Vala, methods are not first-class objects, and as such cannot be created dynamically; however, any method can be considered to be an instance of a delegate's type, provided that the method signature matches that of the delegate.

Methods are considered to be an immutable reference type. Any method can be referred to by name as an expression returning a reference to that method - this can be assigned to a field (or variable, or parameter), or else invoked directly as a standard method invocation (see Expressions/Invocation expressions).

8.1 Types of delegate

All delegate types in Vala are defined to be either static or instance delegates. This refers to whether the methods that may be considered instances of the delegate type are instance members of classes or structs, or not.

To assign an instance of an instance delegate, you must give the method name qualified with an identifier that refers to a class or struct instance. When an instance of an instance delegate is invoked, the method will act as though the method had been invoked directly: the "this" keyword will be usable, instance data will be accessible, etc.

Instance and static delegate instances are not interchangeable.

8.2 Delegate declaration

The syntax for declaring a delegate changes slightly based on what sort of delegate is being declared. This section shows the form for a namespace delegate. Many of the parts of the declaration are common to all types, so sections from here are referenced from class delegates, interface delegates, etc.

delegate-declaration:
    instance-delegate-declaration
    static-delegate-declaration

instance-delegate-declaration:
    [access-modifier] **`delegate`** return-type qualified-delegate-name **`(`** method-params-list **`)`** [**`throws`** error-list] **`;`**

static-delegate-declaration:
    [access-modifier] **`static delegate`** return-type qualified-delegate-name **`(`** method-params-list **`)`** [**`throws`** error-list] **`;`**

qualified-delegate-name:
    [qualified-namespace-name **`.`**] delegate-name

delegate-name:
    identifier

Parts of this syntax are based on the respective sections of the method declaration syntax (see Methods for details).

8.3 Using delegates

A delegate declaration defines a type. Instances of this type can then be assigned to variables (or fields, or parameters) of this type. Vala does not allow creating methods at runtime, and so the values of delegate-type instances will be references to methods known at compile time. To simplify the process, inlined methods may be written (see Methods/Lambdas).

To call the method referenced by a delegate-type instance, use the same notation as for calling a method; instead of giving the method's name, give the identifier of the variable, as described in Expressions/Invocation expressions.

8.4 Examples

Defining delegates:

```
// Static delegate taking two ints, returning void:
static void DelegateName (int a, int b);

// Instance delegate with the same signature:
void DelegateName (int a, int b);

// Static delegate which may throw an error:
static void DelegateName () throws GLib.Error;
```

Invoking delegates, and passing as parameters.

```
void f1(int a) { stdout.printf("%d\n", a); }
...
void f2(DelegateType d, int a) {
        d(a);
}
...
f2(f1, 5);
```

Instance delegates:

```
class Test : Object {
        private int data = 5;
        public void method (int a) {
                stdout.printf("%d %d\n", a, this.data);
        }
}

delegate void DelegateType (int a);

public static void main (string[] args) {
        var t = new Test();
        DelegateType d = t.method;

        d(1);
}
```

With Lambda:

```
f2(a => { stdout.printf("%d\n", a); }, 5);
```

9. Errors

9.1 Error throwing
9.2 Error catching
9.3 Examples

Vala Error handling is just for recoverable runtime errors, anything that can be reasonably foreseen should not be handled with errors, e.g. passing the wrong args to a method. In that example, a better action is to state that the method's result is undefined on illegal input, and use method contracts or assertions to catch potential problems during development: See Methods/Contract programming. A more suitable use for errors would be reporting missing files, which of course cannot be detected until the program is running.

A method may declare that it throws methods from any number of error domains. Error domains are groups of related errors, each of which is denoted by a unique symbol in much the same way an enumerated type, see Enumerated types (Enums)/Error domains for declaration syntax. In Vala it is not allowed to throw arbitrary data as in C++, and there is no class for errors, as in Java.

No error can be thrown must either be caught or declared as being thrown.

When a method declares it may thrown an error, the invoker may choose to either catch the error (should one be thrown), or ignore it, meaning it will be thrown on to that methods caller. In the latter case, the method failing to catch the error must also be declared to throw that type of error. Errors can only be caught when the method throwing it is invoked within the try block of a try statement. A try statement, with its associated catch blocks, can potentially catch all errors thrown in its scope, either with catch blocks for all error domains from which a thrown error might come, or with a generic catch block to catch any error.

When an error is first thrown, the "throw" statement is considered the same as a method which from which an error has been thrown. This means that it is possible to catch errors locally, but this is not good practise. The only proper use of this functionality is to use a finally block to free resources before the error is thrown from the method.

When an error is thrown, the following sequence of events happens:

NB: finally clauses are always run, regardless of if error is thrown and/or handled.

9.1 Error throwing

Throwing an error is done with the following syntax:

```
throw-statement:
    throw error-description ;
```

```
error-description:
    identifier
    error-creation-expression

error-creation-expression:
    new qualified-error-type ( message-expression )

qualified-error-type:
    qualified-error-domain . error-type

qualified-error-domain:
    [ qualified-namespace-name . ] error-domain-name
```

That is, throw an error that has already been created and can be identified by a name, or a new error created with a textual description. The message-expression is any expression that evaluates to a instance of the string type.

9.2 Error catching

The syntax of the try statement:

```
try-statement:
    try statement-block catch-clauses
    try statement-block [catch-clauses] finally-clause

catch-clauses:
    [ specific-catch-clauses ] general-catch-clause

specific-catch-clauses:
    specific-catch-clause
    [ specific-catch-clauses ]

specific-catch-clause:
    catch ( qualified-error-type identifier ) statement-block

general-catch-clause:
    catch statement-block

finally-clause:
```

```
    finally statement-block
```

In the statement block scope of each catch clause, the error is assigned to a variable with the identifier given.

9.3 Examples

Demonstrating...

```
errordomain ErrorType1 {
    CODE_1A
}

errordomain ErrorType2 {
    CODE_2A
}

public class Test : GLib.Object {
    public static void thrower() throws ErrorType1,
ErrorType2 {
        throw new ErrorType2.CODE_1A("Error");
    }

    public static void catcher() throws ErrorType2 {
        try {
            thrower();
        } catch (ErrorType1 ex) {
            // Deal with ErrorType1
        } finally {
            // Tidy up
        }
    }

    public static void main(string[] args) {
        try {
            catcher();
        } catch (ErrorType2 ex) {
            // Deal with ErrorType2
        }
```

```
        }
    }
```

10. Classes

10.1 Types of class
10.2 Types of class members
10.3 Class scope
10.4 Class member visibility
10.5 Class declaration
10.6 Controlling instantiation
10.7 Construction
10.8 Class fields
10.9 Class constants
10.10 Class methods
10.11 Properties
10.12 Signals
10.13 Class enums
10.14 Class delegates
10.15 Examples

A class is definition of a data type. A class 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.

The simplest class declaration looks like this:

```
class ClassName {
}
```

As class types support inheritance, you can specify a base class you want to derive from. A derived class is-a superclass. It gets access to some of its methods etc. It can always be used in place of a and so on....

No classes can have multiple base classes, however GObject subclasses may implement multiple interfaces. By implementing an interface, a classed type has an is-a relationship with an interface. Whenever an instance of that interface is expected, an instance of this class will do.

10.1 Types of class

Vala supports three different types of class:

- GObject subclasses are any classes derived directly or indirectly from GLib.Object. This is the most powerful type of class, supporting all features described in this page. This means signals, managed properties, interfaces and complex construction methods, plus all features of the simpler class types.

- Fundamental GType classes are those either without any superclass or that don't inherit at any level from GLib.Object. These classes support inheritance, interfaces,

virtual methods, reference counting, unmanaged properties, and private fields. They are instantiated faster than GObject subclasses but are less powerful - it isn't recommended in general to use this form of class unless there is a specific reason to.

- Compact classes, so called because they use less memory per instance, are the least featured of all class types. They are not registered with the GType system and do not support reference counting, virtual methods, or private fields. They do support unmanaged properties. Such classes are very fast to instantiate but not massively useful except when dealing with existing libraries. They are declared using the Compact attribute on the class, See

Any non-compact class can also be defined as abstract. An abstract class cannot be instantiated and is used as a base class for derived classes.

10.2 Types of class members

There are three fundamentally different types of class members, instance, class and static.

- Instance members are held per instance of the class. That is, each instance has its own copies of the members in its own instance scope. Changes to instance fields will only apply to that instance, calling instance methods will cause them to be executed in the scope of that instance.

- Class members are shared between all instances of a class. They can be accessed without an instance of the class, and class methods will execute in the scope of the class.

- Static members are shared between all instances of a class and any sub-classes of it. They can be accessed without an instance of the class, and static methods will execute in the scope of the class.

The distinction between class and static members is not common to other object models. The essential difference is that a sub-class will receive a copy of all its base classes' class members. This is opposed to static members, of which there is only one copy - sub classes access can their base classes' static members because they are automatically imported into the class' scope.

10.3 Class scope

Class scope is more complicated than other scopes, but conceptually the same. A class has a scope, which consists of its static and class members, as describe above. When an instance of the class is created, it is given its own scope, consisting of the defined instance members, with the class' scope as its parent scope.

Within the code of a class, the instance and class scopes are automatically searched as appropriate after the local scope, so no qualification is normally required. When there is a conflict with a name in the local scope, the `this` scope can be used, for example:

```
class ClassName {
        int field_name;
        void function_name(field_name) {
                this.field_name = field_name;
        }
}
```

When a name is defined in a class which conflicts with one in a subclass, the `base` scope can be used, to refer to the scope of the subclass.

10.4 Class member visibility

All class members have a visibility. Visibility is declared using the following mutually exclusive modifiers:

class-member-visibility-modifier:

`private`
`protected`
`internal`
`public`

This defines whether the member is visible to code in different locations:

- "private" asserts that the member will only be visible to code that is within this class declaration
- "protected" asserts that the member will be visible to any code within this class, and also to any code that is in a subclass of this class
- "internal" asserts that the member should be visible to any code in the project, but excludes the member from the public API of a shared object
- "public" asserts that the member should be visible to any code, including the public API of a shared object

10.5 Class declaration

class-declaration:

[access-modifier] **`class`** qualified-class-name [inheritance-list] **`{`**
[class-members] **`}`**

```
qualified-class-name:
    [ qualified-namespace-name . ] class-name

class-name:
    identifier

inheritance-list:
    : superclasses-and-interfaces

superclasses-and-interfaces:
    ( qualified-class-name | qualified-interface-name ) [ , superclasses-and-
interfaces ]

class-members:
    class-member [ class-members ]

class-member:
    class-creation-method-declaration
    class-constructor-declaration
    class-destructor-declaration
    class-constant-declaration
    class-delegate-declaration
    class-enum-declaration
    class-instance-member
    class-class-member
    class-static-member
    inner-class-declaration

class-constructor-declaration:
    class-instance-constructor-declaration
    class-class-constructor-declaration
    class-static-constructor-declaration

class-instance-member:
    class-instance-field-declaration
    class-instance-method-declaration
    class-instance-property-declaration
    class-instance-signal-declaration

class-class-member:
```

class-class-field-declaration
class-class-method-declaration
class-class-property-declaration

class-static-member:
class-static-field-declaration
class-static-method-declaration
class-static-property-declaration

inner-class-declaration:
[access-modifier] **`class`** class-name [inheritance-list] **{** [class-members] **}**

In Vala, a class must have either one or zero superclasses, where have zero superclasses has the result described in section. A class must meet all the prerequisites defined by the interfaces it wishes to implement, by implementing prerequisite interfaces or inheriting from a particular class. This latter requirement means it is potentially possible to have two interfaces that cannot be implemented by a single class.

When declaring which class, if any, a new class subclasses, and which interfaces it implements, the names of those other classes or interfaces can be qualified relative to the class being declared. This means that, for example, if the class is declared as "class foo.Bar" (class "Bar" in namespace "foo") then it may subclass class "Base" in namespace "foo" simply with "class foo.Bar : Base".

If an access modifier for the class is not given, the default "internal" is used.

It is possible to declare a class definition to be "abstract." An abstract class is one they may not be instantiated, instead it first be subclassed by a non-abstract ("concrete") class. An abstract class declaration may include abstract class instance members. These act as templates for methods or properties that must be implemented in all concrete subclasses of the abstract class. It is thus guaranteed that any instance of the abstract class (which must be in fact an instance of a concrete subclass) will have a method or property as described in the abstract class definition.

abstract-class-declaration:
[access-modifier] **`abstract class`** qualified-class-name [inheritance-list] **{** [abstract-class-members] **}**

abstract-class-members:
class-members
class-instance-abstract-method-declaration

class-instance-abstract-property-declaration

10.6 Controlling instantiation

When a class is instantiated, data might be required from the user to set initial properties. To define which properties should be or can be set at this stage, the class declaration should be written as:

```
class ClassName : GLib.Object {

        public ClassName() {
        }

        public ClassName.with_some_quality (Property1Type
property1value) {
                this.property1 = property1value;
        }
}
```

This example allows the `ClassName` class to be instantiated either setting no properties, or setting the property. The convention is to name constructors as "with_" and then a description of what the extra properties will be used for, though following this is optional.

class-creation-method-declaration:
    [class-member-visibility-modifier] class-name [**.** creation-method-name]
(param-list **)** **{** construction-assignments **}**

class-name:
    identifier

creation-method-name:
    identifier

construction-assignments:
    this **.** property-name **=** param-name **;**

class-name must be the same as the name of the class. If a creation method is given an extra name, this name is also used with instantiating the class, using the same syntax as for declaring the method, e.g. `var a = new Button.with_label ("text").`

If the property being set is construct type then assignment is made before construction, else afterwards.

Any number of these are allowed, but only one with each name (including null name.)

10.7 Construction

During instantiation, after construction properties have been set, a series of blocks of code are executed. This is the process that prepares the instance for use. There are three types of `construct` blocks that a class may define:

class-instance-constructor-declaration:
`construct {` statement-list **`}`**

Code in this block is executed on every instance of the class that is instantiated. It is run after construction properties have been set.

class-class-constructor-declaration:
`class construct {` statement-list **`}`**

This block will be executed once at the first use of its class, and once at the first use of each subclass of this class.

class-static-constructor-declaration:
`static construct {` statement-list **`}`**

The first time that a class, or any subclass of it, is instantiated, this code is run. It is guaranteed that this code will run exactly once in a program where such a class is used.

The order of execution for constructors:

class-instance-destructor-declaration:
    ~ class-name **( ) {** statement-list **}**

Destruction here. When does it happen? And when for each type of class?

10.8 Class fields

Fields act as variable with a scope of either the class or a particular instance, and therefore have names and types in the same way. Basic declarations are as:

class-instance-field-declaration:
    [class-member-visibility-modifier] qualified-type-name field-name [**=** expression] ;

class-class-field-declaration:
    [class-member-visibility-modifier] **class** qualified-type-name field-name [**=** expression] ;

class-static-field-declaration:
    [class-member-visibility-modifier] **static** qualified-type-name field-name [**=** expression] ;

Initial values are optional. FIXME: how much calculation can be done here? what are the defaults?

10.9 Class constants

Constants defined in a class are basically the same as those defined in a namespace. The only difference is the scope and the choice of visibilities available.

class-constant-declaration:
    [class-member-visibility-modifier] **const** qualified-type-name constant-name **=** expression ;

10.10 Class methods

Class methods are methods bound to a particularly class or class instance, i.e. they are executed within the scope of that class or class instance. They are declared the same way as other methods, but within the declaration of a class.

The same visibility modifiers can be used as for fields, although in this case they refer to what code can call the methods, rather than who can see or change values.

The `static` modifier is applicable to methods also. A static method is independent of any instance of the class. It is therefore only in the class scope, and may only access other `static` members.

class-instance-method-declaration:
    [class-member-visibility-modifier] [class-method-type-modifier] return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **{** statement-list **}**

class-class-method-declaration:
    [class-member-visibility-modifier] **class** return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **{** statement-list **}**

class-static-method-declaration:
    [class-member-visibility-modifier] **static** return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **{** statement-list **}**

class-method-type-modifier:
    virtual
    override

Methods can be virtual, as described in Concepts/Object oriented programming. Methods in Vala classes are not virtual automatically, instead the "virtual" modifier must be used when it is needed. Virtual methods will only chain up if overridden using the override keyword.

Vala classes may also define abstract methods, by writing the declaration with the "abstract" modifier and replacing the method body with an empty statement ";". Abstract methods are not true methods, as they do not have an associated statement block, and so cannot be invoked. Abstract methods can only exist in abstract classes, and must be overridden in derived classes. For this reason an abstract method is always virtual. The purpose of an abstract method is to define methods that all non-abstract

subclasses of the current definition must implement, it is therefore always allowable to invoke the method on an instance of the abstract class, because it is required that that instance must in fact be of a non-abstract subclass.

class-instance-abstract-method-declaration:
    [class-member-visibility-modifier] **abstract** return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **;**

10.11 Properties

Properties are an enhanced version of fields. They allow custom code to be called whenever the property is retrieved or assigned to, but may be treated as fields by external Vala code. Properties also function like methods to some extent, and so can be defined as virtual and overridden in subclasses. Since they are also allowed in interfaces, they allow interfaces to declare data members that implementing classes must expose (see Interfaces.)

Declaration

class-instance-property-declaration:
    [class-member-visibility-modifier] [class-method-type-modifier] qualified-type-name property-name **{** accessors [default-value] **} ;**

class-instance-abstract-property-declaration:
    [class-member-visibility-modifier] **abstract** qualified-type-name property-name **{** automatic-accessors **} ;**

class-class-property-declaration:
    [class-member-visibility-modifier] **class** qualified-type-name property-name **{** accessors [default-value] **} ;**

class-static-property-declaration:
    [class-member-visibility-modifier] **static** qualified-type-name property-name **{** accessors [default-value] **} ;**

property-name:
    identifier

accessors:
    automatic-accessors

```
    [ getter ] [ setter ] [ property-constructor ]

automatic-accessors:
    [ automatic-getter ] [ automatic-setter ] [ automatic-property-constructor ]

automatic-getter:
    [ class-member-visibility-modifier ] get ;

automatic-setter:
    [ class-member-visibility-modifier ] set [ construct ] ;

automatic-property-constructor:
    [ class-member-visibility-modifier ] construct ;

get-accessor:
    [ class-member-visibility-modifier ] get { statement-list }

set-accessor:
    [ class-member-visibility-modifier ] set [ construct ] { statement-
list }

property-constructor:
    [ class-member-visibility-modifier ] construct { statement-list }

default-value:
    default = expression ;
```

Execute Code on Setting/Getting Values

Properties can either be declared with code that will perform particular actions on get and set, or can simply declare which actions are allowed and allow Vala to implement simple get and set methods. This second pattern (automatic property) will result in fields being added to the class to store values that the property will get and set. If either get or set has custom code, then the other must either be also written in full, or omitted altogether.

When a value is assigned to a property, the **set** block is invoked, with a parameter called **value** of the same type as the property. When a value is requested from a property, the **get** block is invoked, and must return an instance of the same type of the property.

Construct / Set Construct Block

A property may have zero or one **construct** blocks. This means either a **set construct** block or a separate **construct** block. If this is the case that then the property becomes a construct property, meaning that if it is set in creation method, it will be set (using the construct block, as opposed to any simple **set** block, where there is a distinction) before class construct blocks are called.

Notify Changes Signals

Managed properties may be annotated with Notify, See Attributes. This will cause the class instance to emit a notify signal when the property has been assigned to.

Virtual Properties

Instance properties can be defined virtual with the same semantics as for virtual methods. If in an abstract class, an instance property can be defined as abstract. This is done using the "abstract" keyword on a declaration that is otherwise the same as an automatic property. It is then the responsibility of derived classes to implement the property by providing get or set blocks as appropriate. An abstract property is automatically virtual.

Abstract Properties

As with methods, it is possible to declare abstract properties. These have much the same semantics as abstract methods, i.e. all non-abstract subclasses will have to implement properties with at least the accessors defined in the abstract property. Any **set construct** or construct accessor must be defined too in non-abstract classes and use **override**.

class-instance-abstract-property-declaration:
    [class-member-visibility-modifier] **abstract** qualified-type-name property-name **{** automatic-accessors **} ;**

10.12 Signals

Signals are a system allowing a classed-type instance to emit events which can be received by arbitrary listeners. Receiving these events is achieved by connecting the signal to a handler, for which Vala has a specific syntax. Signals are integrated with the GLib MainLoop system, which provides a system for queueing events (i.e. signal emissions,) when needed - though this capability is not needed non-threaded applications.

class-instance-signal-declaration:
    [class-member-visibility-modifier] [class-method-type-modifier]
signal return-type signal-name **(** [params-list] **) ;**

signal-name:

identifier

Signals may also provide an extra piece of information called a signal detail. This is a single string, which can be used as an initial hint as to the purpose of the signal emission. In Vala you can register that a signal handler should only be invoked when the signal detail matches a given string. A typical use of signal details is in GObject's own "notify" signal, which says that a property of an object has changed - GObject uses the detail string to say which property has been changed.

To assign a handler to a signal, (or register to receive this type of event from the instance), use the following form of expression:

signal-connection-expression:
    qualified-signal-name [signal-detail] **+=** signal-handler

qualified-signal-name:
    [qualified-namespace-name **.**] variable-identifier **.** signal-name

signal-detail:
    [expression **]**

signal-handler:
    expression
    qualified-method-name
    lambda-expression

This expression will request that the signal handler given be invoked whenever the signal is emitted. In order for such a connection expression to be legal, the handler must have the correct signature. The handler should be defined to accept as parameters the same types as the signal, but with an extra parameter before. This parameter should have the type of the class in which the signal is declared. When a signal is emitted all handlers are called with this parameter being the object by which the signal was emitted.

The time that an arbitrary expression is acceptable in this expression is when that expression evaluates to an instance of a delegate type, i.e. to a method that is a legal handler for the signal. For details on delegates, see Delegates. For details on lambda expressions see Methods/Lambdas.

Note that optional signal detail should be directly appended to the signal name, with no white space, e.g. `o.notify["name"] += ...`

It is also possible to disconnect a signal handler using the following expression form:

signal-disconnection-expression:
    qualified-signal-name [signal-detail] **-=** connected-signal-handler

connected-signal-handler:
    expression
    qualified-method-name

Note that you cannot disconnect a signal handler which was defined inline as a lambda expression and then immediately connected to the signal. If this is the effect you really need to achieve, you must assign the lambda expression to an identifier first, so that the lambda can be referred to again at a later time.

10.13 Class enums

Enums defined in a class are basically the same as those defined in a namespace. The only difference is the scope and the choice of visibilities available. See Enumerated types (Enums).

class-enum-declaration:
    [class-member-visibility-modifier] **enum** enum-name **{** [enum-members] **}**

10.14 Class delegates

Delegates defined in a class are basically the same as those defined in a namespace. The only difference is the scope and the choice of visibilities available. See Delegates.

class-delegate-declaration:
    [class-member-visibility-modifier] return-type **delegate** delegate-name **(** method-params-list **) ;**

10.15 Examples

Demonstrating...

```
// ...
```

Using Properties

For more examples see: Samples for Class Properties

Virtual Properties

```
namespace Properties {
    class Base : Object {
        protected int _number;
        public virtual int number {
            get {
                return this._number;
            }
            set {
                this._number = value;
            }
        }
    }

    /*
     * This class just use Base class default handle
     * of number property.
     */
    class Subclass : Base {
        public string name { get; set; }
    }


    /**
     * This class override how number is handle
internally.
     *
     */
    class ClassOverride : Base {
        public override int number {
            get {
                return this._number;
```

```
            }
            set {
                this._number = value * 3;
            }
        }

        public static int main (string[] args) {
            stdout.printf ("Implementing Virtual
Properties...\n");
            var bc = new Base ();
            bc.number = 3;
            stdout.printf ("Class number = '" +
bc.number.to_string () + "'\n");
            var sc = new Subclass ();
            sc.number = 3;
            stdout.printf ("Class number = '" +
sc.number.to_string () + "'\n");
            var co = new ClassOverride ();
            co.number = 3;
            stdout.printf ("Class number = '" +
co.number.to_string () + "'\n");
            return 0;
        }
    }
}
```

Abstract Properties

```
namespace Properties {
    abstract class Base : Object {
        public abstract string name { get; set
construct; }

        construct {
            this.name = "NO_NAME";
        }
    }

    class Subclass : Base {
        private string _name;
```

```
        public override string name {
            get {
                return this._name;
            }

            set construct {
                this._name = value;
            }
        }

        /* This class have a default constructor that
initializes
         * name as the construct block on Base, and
a .with_name()
         * constructor where the user can set class
derived name
         * property.
        */
        public Subclass.with_name (string name) {
            Object (name:name);
            this._name = name;
        }

        public static int main (string[] args) {
            stdout.printf ("Implementing Abstract
Properties...\n");
            var sc = new Subclass.with_name
("TEST_CLASS");
            stdout.printf ("Class name = '" + sc.name +
"'\n");
            var sc2 = new Subclass ();
            stdout.printf ("Class name = '" + sc2.name +
"'\n");
            return 0;
        }
    }
}
```

Compile and run using:

```
# valac source.vala
# ./source
```

Using signals

```
public class Test : Object {
        public signal void test (int data);
}

void delegate TestHandler (Test t, int data);

public static void main (string[] args) {

        Test t = new Test();

        TestHandler h = (t, data) => {
                stdout.printf("Data: %d\n", d);
        }

        t.test ();
        t.test += h;
        t.test ();
        t.test -= h;
        t.test ();
}
```

11. Interfaces

11.1 Interface declaration
11.2 Interface fields
11.3 Interface methods
11.4 Interface properties
11.5 Interface signals
11.6 Other interface members
11.7 Examples

An interface in Vala is a non-instantiable type. A class may implement any number of interfaces, thereby declaring that an instance of that class should also be considered an instance of those interfaces. Interfaces are part of the GType system, and so compact classes may not implement interfaces (see Classes/Types of class.)

The simplest interface declaration looks like this:

```
interface InterfaceName {
}
```

Unlike C# or Java, Vala's interfaces may include implemented methods, and so provide premade functionality to an implementing class, similar to mixins in other languages. All methods defined in a Vala interface are automatically considered to be virtual. Interfaces in Vala may also have prerequisites - classes or other interfaces that implementing classes must inherit from or implement. This is a more general form of the interface inheritance found in other languages. It should be noted that if you want to guarantee that all implementors of an interface are GObject type classes, you should give that class as a prerequisite for the interface.

Interfaces in Vala have a static scope, identified by the name of the interface. This is the only scope associated with them (i.e. there is no class or instance scope created for them at any time.) Non-instance members of the interface (static members and other declarations,) can be identified using this scope.

For an overview of object oriented programming, see Concepts/Object oriented programming.

11.1 Interface declaration

interface-declaration:
    [access-modifier] **interface** qualified-interface-name [inheritance-list] **{** [interface-members] **}**

qualified-interface-name:

```
    [ qualified-namespace-name . ] interface-name

interface-name:
    identifier

inheritance-list:
    : prerequisite-classes-and-interfaces

prerequisite-classes-and-interfaces:
    qualified-class-name [ , prerequisite-classes-and-interfaces ]
    qualified-interface-name [ , prerequisite-classes-and-interfaces ]

interface-members:
    interface-member [ interface-members ]

interface-member:
    interface-constant-declaration
    interface-delegate-declaration
    interface-enum-declaration
    interface-instance-member
    interface-static-member
    interface-inner-class-declaration
    abstract-method-declaration

interface-instance-member:
    interface-instance-method-declaration
    interface-instance-abstract-method-declaration
    interface-instance-property-declaration
    interface-instance-signal-declaration

interface-static-member:
    interface-static-field-declaration
    interface-static-method-declaration
```

11.2 Interface fields

As an interface is not instantiable, it may not contain data on a per instance basis. It is though allowable to define static fields in an interface. These are equivalent to static

fields in a class: they exist exactly once regardless of how many instances there are of classes that implement the interface.

The syntax for static interface fields is the same as the static class fields: See Classes/Class fields. For more explanation of static vs instance members, see Classes/Types of class members.

11.3 Interface methods

Interfaces can contain abstract and non abstract methods. A non-abstract class that implements the interface must provide implementations of all abstract methods in the interface. All methods defined in an interface are automatically virtual.

Vala interfaces may also define static methods. These are equivalent to static methods in classes.

interface-instance-method-declaration:
    [class-member-visibility-modifier] return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **{** statement-list **}**

interface-instance-abstract-method-declaration:
    [class-member-visibility-modifier] **abstract** return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **;**

interface-static-method-declaration:
    [class-member-visibility-modifier] **static** return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **{** statement-list **}**

For discussion of methods in classes, see: Classes/Class methods. For information about methods in general, see Methods. Of particular note is that an abstract method of an interface defines a method that can always be called in an instance of an interface, because that instance is guaranteed to be of a non-abstract class that implements the interface's abstract methods.

11.4 Interface properties

Interfaces can contain properties in a similar way to classes. As interfaces can not contain per instance data, interface properties cannot be created automatically. This means that all properties must either be declared abstract (and implemented by implementing classes,) or have explicit get and set clauses as appropriate. Vala does not allow an abstract property to be partially implemented, instead it should just define which actions (get, set or both) should be implemented.

Interfaces are not constructed so there is no concept of an interface construction property.

interface-instance-property-declaration:
    [class-member-visibility-modifier] [class-method-type-modifier] qualified-type-name property-name **{** accessors [default-value] **}** **;**
    [class-member-visibility-modifier] **abstract** qualified-type-name property-name **{** automatic-accessors **}** **;**

For properties in classes see Classes/Properties.

11.5 Interface signals

Signals can be defined in interfaces. They have exactly the same semantics as when directly defined in the implementing class.

interface-instance-signal-declaration:
    class-instance-signal-declaration

11.6 Other interface members

Constants, Enums, Delegates and Inner Classes all function the same as when they are declared in a class. See Classes. When declared in an interface, all these members can be accessed either using the name of the interface (that is, of the static interface scope), or through and instance of an implementing class.

11.7 Examples

Here is an example implementing (and overriding) an **abstract** interface method,

```
/*
   This example gives you a simple interface, Speaker,
with
   - one abstract method, speak

   It shows you three classes to demonstrate how these
and overriding them behaves:
   - Fox, implementing Speaker
   - ArcticFox, extending Fox AND implementing Speaker
```

```
     (ArcticFox.speak () replaces superclasses' .speak())
   - RedFox, extending Fox BUT NOT implementing speaker
     (RedFox.speak () does not replace
superclasses' .speak())

   Important notes:
   - generally an object uses the most specific class's
implementation
   - ArcticFox extends Fox (which implements Speaker) and
implements Speaker itself,
     - ArcticFox defines speak () with new, so even
casting to Fox or Speaker still
       gives you ArcticFox.speak ()
   - RedFox extends from Fox, but DOES NOT implement
Speaker
     - RedFox speak () gives you RedFox.speak ()
     - casting RedFox to Speaker or Fox gives you
Fox.speak ()
*/

/* Speaker: extends from GObject */
interface Speaker : Object {
  /* speak: abstract without a body */
  public abstract void speak ();
}

/* Fox: implements Speaker, implements speak () */
class Fox : Object, Speaker {
  public void speak () {
    stdout.printf ("  Fox says Ow-wow-wow-wow\n");
  }
}

/* ArcticFox: extends Fox; must also implement Speaker to
re-define
 *            inherited methods and use them as Speaker */
class ArcticFox : Fox, Speaker {
  /* speak: uses 'new' to replace speak () from Fox */
  public new void speak () {
    stdout.printf ("  ArcticFox says Hatee-hatee-hatee-ho!
```

```
\n");
  }
}

/* RedFox: extends Fox, does not implement Speaker */
class RedFox : Fox {
  public new void speak () {
    stdout.printf ("  RedFox says Wa-pa-pa-pa-pa-pa-pow!
\n");
  }
}

public static int main () {
  Speaker f = new Fox ();
  Speaker a = new ArcticFox ();
  Speaker r = new RedFox ();

  stdout.printf ("\n\n// Fox implements Speaker, speak ()
\n");
  stdout.printf ("Fox as Speaker:\n");
  (f as Speaker).speak ();   /* Fox.speak () */
  stdout.printf ("\nFox as Fox:\n");
  (f as Fox).speak ();       /* Fox.speak () */

  stdout.printf ("\n\n// ArcticFox extends Fox, re-
implements Speaker and " +
                 "replaces speak ()\n");
  stdout.printf ("ArcticFox as Speaker:\n");
  (a as Speaker).speak ();   /* ArcticFox.speak () */
  stdout.printf ("\nArcticFox as Fox:\n");
  (a as Fox).speak ();       /* ArcticFox.speak () */
  stdout.printf ("\nArcticFox as ArcticFox:\n");
  (a as ArcticFox).speak (); /* ArcticFox.speak () */

  stdout.printf ("\n\n// RedFox extends Fox, DOES NOT re-
implement Speaker but" +
                 " does replace speak () for itself\n");
  stdout.printf ("RedFox as Speaker:\n");
  (r as Speaker).speak ();   /* Fox.speak () */
  stdout.printf ("\nRedFox as Fox:\n");
```

```
  (r as Fox).speak ();       /* Fox.speak () */
  stdout.printf ("\nRedFox as RedFox:\n");
  (r as RedFox).speak ();    /* RedFox.speak () */

  return 0;
}
```

Here is an example of implementing (and inheriting) a **virtual** interface method. Note that the same rules for subclasses re-implementing methods that apply to the **abstract** interface method above apply here.

```
/*
   This example gives you a simple interface, Yelper, with
   - one virtual default method, yelp

   It shows you two classes to demonstrate how these and
overriding them behaves:
   - Cat, implementing Yelper (inheriting yelp)
   - Fox, implementing Yelper (overriding yelp)

   Important notes:
   - generally an object uses the most specific class's
implementation
   - Yelper provides a default yelp (), but Fox overrides
it
     - Fox overriding yelp () means that even casting Fox
to Yelper still gives
       you Fox.yelp ()
   - as with the Speaker/speak() example, if a subclass
wants to override an
     implementation (e.g. Fox.yelp ()) of a virtual
interface method
     (e.g. Yelper.yelp ()), it must use 'new'
   - 'override' is used when overriding regular class
virtual methods,
     but not when implementing interface virtual methods.
*/

interface Yelper : Object {
  /* yelp: virtual, if we want to be able to override it
```

```
*/
  public virtual void yelp () {
    stdout.printf ("  Yelper yelps Yelp!\n");
  }
}

/* Cat: implements Yelper, inherits virtual yelp () */
class Cat : Object, Yelper {
}

/* Fox: implements Yelper, overrides virtual yelp () */
class Fox : Object, Yelper {
  public void yelp () {
    stdout.printf ("  Fox yelps Ring-ding-ding-ding-
dingeringeding!\n");
  }
}

public static int main () {
  Yelper f = new Fox ();
  Yelper c = new Cat ();

  stdout.printf ("// Cat implements Yelper, inherits
yelp\n");
  stdout.printf ("Cat as Yelper:\n");
  (c as Yelper).yelp ();  /* Yelper.yelp () */
  stdout.printf ("\nCat as Cat:\n");
  (c as Cat).yelp ();     /* Yelper.yelp () */

  stdout.printf ("\n\n// Fox implements Yelper, overrides
yelp ()\n");
  stdout.printf ("Fox as Yelper:\n");
  (f as Yelper).yelp ();  /* Fox.yelp () */
  stdout.printf ("\nFox as Fox:\n");
  (f as Fox).yelp ();     /* Fox.yelp () */

  return 0;
}
```

12. Generics

12.1 Generics declaration
12.2 Instantiation
12.3 Examples

Generic programming is a way of defining that something is applicable to a variety of potential types, without having to know these types before hand. The classic example would be a collection such as a list, which can be trivially customised to contain any type of data elements. Generics allow a Vala programmer to have these customisations done automatically.

Some of these are possible, which?

```
class Wrapper < T > : Object { ... }
new Wrapper < Object > ( ) ;
BUG: class StringWrapper : Wrapper < string > ( ) { ... }
FAIL: class WrapperWrapper < Wrapper < T > > : Object { ... }
FAIL: new WrapperWrapper < Wrapper < Object > > ( ) ;
interface IWrapper < T > { ... }
class ImpWrapper1 < T > : Object, IWrapper < T > { ... }
BUG: class ImpWrapper2 : Object, IWrapper < string > { ... }
```

12.1 Generics declaration

Some of the syntax could be best placed in the class/interface/struct pages, but that might overcomplicate them...

In class declaration - In struct declaration - In interface declaration - In base class declaration - In implemented interfaces declaration - In prerequesite class/interface declaration.

Declaration with type parameters introduces new types into that scope, identified by names given in declaration, e.g. T.

```
qualified-type-name-with-generic:
    qualified-class-name-with-generic
    qualified-interface-name-with-generic
    qualified-struct-name-with-generic

qualified-class-name-with-generic:
    [ qualified-namespace-name . ] class-name type-parameters
```

qualified-interface-name-with-generic:
    [qualified-namespace-name **.**] interface-name type-parameters

qualified-struct-name-with-generic:
    [qualified-namespace-name **.**] struct-name type-parameters

type-parameters:
    < generic-clause **>**

generic-clause:
    type-identifier [**,** generic-clause]
    qualified-type-name [**,** generic-clause]

type-identifier:
    identifier

type-identifier will be the type-name for the parameterised type.

Deal is: in the class/interface/struct sections, replace qualified-*-name with qualified-*-name-with-generic.

12.2 Instantiation

Only explanation here? Syntax should go with variable declaration statement?

When using generic for a type-name, only type-names can be used as type-parameters, not identifiers. NB. in scope of generic class, T etc. is a real type-name.

12.3 Examples

Demonstrating...

```
using GLib;

public interface With < T > {
        public abstract void sett(T t);
        public abstract T gett();
}
```

```
public class One : Object, With < int > {
        public int t;

        public void sett(int t) {
                this.t = t;
        }
        public int gett() {
                return t;
        }
}

public class Two < T, U > : Object, With < T > {
        public T t;

        public void sett(T t) {
                this.t = t;
        }
        public T gett() {
                return t;
        }

        public U u;
}

public class Test : GLib.Object {

        public static void main(string[] args) {
                var o = new One ();
                o.sett(5);
                stdout.printf("%d\n", o.t);

                var t = new Two < int, double? > ();
                t.sett(5);
                stdout.printf("%d\n", t.t);

                t.u = 5.0f;
                stdout.printf("%f\n", t.u);
        }
}
```

13. Structs

13.1 Struct declaration
13.2 Controlling instantiation
13.3 Struct fields
13.4 Struct constants
13.5 Struct methods
13.6 Examples

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

The simplest struct declaration looks like this:

```
struct StructName {
        int some_field;
}
```

A struct must have at least one field, except in either one of the following cases:

- It's external
- It has either one of `[BooleanType]`, `[IntegerType]` or `[FloatingType]` attributes
- It inherits from another struct

13.1 Struct declaration

struct-declaration:
    [access-modifier] **`struct`** qualified-struct-name [**`:`** super-struct] **`{`** [struct-members] **`}`**

qualified-struct-name:
    [qualified-namespace-name **`.`**] struct-name

struct-name:
    identifier

struct-members:
    struct-member [struct-members]

struct-member:

struct-creation-method-declaration:
struct-field-declaration
struct-constant-declaration
struct-method-declaration

If a super-struct is given, the struct-name becomes an alias for that struct.

13.2 Controlling instantiation

struct-creation-method-declaration:
[struct-access-modifier] struct-name [**.** creation-method-name] **(** param-list **)** **{** statement-list **}**

struct-name:
identifier

Unlike in a class, any code can go in this method.

13.3 Struct fields

Documentation

struct-field-declaration:
[access-modifier] [struct-field-type-modifier] qualified-type-name field-name [**=** expression] ;

struct-field-type-modifier:
`static`

13.4 Struct constants

class-constant-declaration:
[class-access-modifier] **`const`** qualified-type-name constant-name **=** expression ;

13.5 Struct methods

See Methods, See Classes/Class methods

struct-method-declaration:
    [access-modifier] [struct-method-type-modifier] return-type method-name **(** [params-list] **)** method-contracts [**throws** exception-list] **{** statement-list **}**

struct-method-type-modifier:
    static

13.6 Examples

Demonstrating...

```
// ...
```

14. Enumerated types (Enums)

14.1 Enum declaration
14.2 Enum members
14.3 Methods
14.4 Flag types
14.5 Error domains
14.6 Examples

Enumerated types declare all possible values that instances of the type may take. They may also define methods of the type, but an enumerated type has no data other than its value. Enumerated types are value types, and so each instantiation of the type is unique, even when they represent the same value. This distinction is not significant in practice because when instances are compared, it is always by value not identity.

Enumerated types are usually known as simply "enums".

14.1 Enum declaration

enum-declaration:
    [access-modifier] **`enum`** qualified-enum-name **`{`** [enum-members] **`}`**

qualified-enum-name:
    [qualified-namespace-name **`.`**] enum-name

enum-name:
    identifier

enum-members:
    [enum-values] [**`;`** enum-methods]

enum-values:
    enum-value [**`,`** enum-values]

enum-value:
    enum-value-name [**`=`** expression]

enum-value-name:
    identifier

enum-methods:

```
    enum-method [ enum-methods ]

enum-method:
    method-declaration
```

14.2 Enum members

Equivalent to constants, all have an integer value, either explicit or automatically assigned.

14.3 Methods

Are similar to static methods of classes, i.e. are not related to any particular instance, but can be invoked on either an instance or the enum itself.

14.4 Flag types

An enumerated type declaration can be converted into a flag type declaration by annotating the declaration with "Flags". A flag type represents a set of flags, any number of which can be combined in one instance of the flag type, in the same fashion as a bitfield in C. For an explanation of the operations that can be performed on flag types, see Expressions/Flag operations. For how to use attributes, see Attributes.

For example, say we want to draw the borders of a table cell:

```
[Flags]
enum Borders {
    LEFT,
    RIGHT,
    TOP,
    BOTTOM
}

void draw_borders (Borders selected_borders) {
    // equivalent to: if ((Borders.LEFT &
selected_borders) > 0)
    if (Borders.LEFT in selected_borders) {
        // draw left border
    }
    if (Borders.RIGHT in selected_borders) {
        // draw right border
    }
```

```
    ...
}
```

14.5 Error domains

Error domains are Vala's method for describing errors. An error domain is declared using a similar syntax to enumerated types, but this does not define a type - instead it defines a class of errors, which is used to implicitly create a new error type for the error domain. The error domain declaration syntax is effectively the same as for enumerated types, but the keyword `errordomain` is used instead of `enum`.

For more information about handling errors in Vala, see Errors.

14.6 Examples

Demonstrating...

```
// ...
```

15. Attributes

15.1 Applying attributes
15.2 CCode Attribute
15.3 Version attribute
15.4 SimpleType attribute
15.5 BooleanType Attribute
15.6 IntegerType Attribute
15.7 FloatingType Attribute
15.8 Signal Attribute
15.9 Description Attribute
15.10 DBus Attribute
15.11 Gtk attributes
15.12 Other attributes
15.13 Deprecated Attributes
15.14 Examples

Attributes are metadata information that is specified with regards to a symbol (a class, field, parameter, etc.).

Attributes provide extra information in order to:

- Integrate libraries more directly. These are the ones most often used in new Vala programs/libraries.
- Control C code generation, particularly with existing libraries. Mostly used in bindings.
- Give extra information to Vala that isn't included in code. Mostly used internally in Vala.

Most of these attributes are only useful within bindings. Some, however, are useful in normal code:

- `[DBus]`, `[Description]`, `[Version]`, `[Signal]`, `[ModuleInit]` (if you're writing a module).
- `CCode`'s `instance_pos` (if you're using `Gtk.Builder`'s signal auto-connection functionality).

15.1 Applying attributes

They are written as:

```
[ AnnotationName ( details-list ) ]
declaration
```

For example:

```
[ CCode ( cname = "var_c_name" ) ]
static int my_var;
```

15.2 CCode Attribute

This attribute influences the C code which is generated by Vala.

Name	Applies to	Type	Example
Description (optional)			
array_length	delegate, field, property, method, parameter	bool	
If the array length is unknown, setting array_length = false in the CCode attribute will cause Vala to set the array's .length property to -1 and not pass the length when used as a parameter.			
array_length_cname	field	string	
array_length_cexpr	field	string	
array_length_pos	constructor, delegate, method, parameter	double	0.9
The position of the argument which should be the length of the return array. Integers (such as 1.0, 2.0) specify arguments, so to place it before or after these arguments, use a value less (i.e. 0.9) or more (i.e. 1.1) than the argument.			
array_length_type	field, method	string	
array_null_terminated	constructor, method, delegate, field, parameter, property	bool	
cheader_filename	class, constant, constructor, delegate, enum, field, interface, method, namespace, struct	string - comma-separated list of headers	"glib.h"

The header file(s) which should be #included in the emitted C code, so that this symbol is usable. If more than one header file is needed, separate them by commas.			
cname	class, constant, constructor, delegate, enum, field, method, struct, propacc	string	"gboolean"
The name that this symbol will take when translated into C code. If this attribute is not specified, the symbol will get a name with the normal vala translation rules.			
const_cname	class, struct	string	
construct_function	constructor	string	
copy_function	class	string	
cprefix	class, enum, namespace, struct	string	
default_value	struct	string - C value expression	"FALSE"
A C expression representing this type's default value.			
delegate_target	field	bool	
delegate_target_pos	constructor, delegate, method, parameter	double	0.1
delegate_target_cname	delegate field/ parameter	string	"userdata"
A C expression representing the name of the target/userdata related to a delegate field/ parameter.			
destroy_function	struct	string	
destroy_notify_pos	parameter	double	
free_function	class	string	
free_function_address_of	class	bool	

generic_type_pos	method	double	
get_value_function	class, struct	string - function name	"g_value_get_boolean"
A function which will return an object when passed a GValue.			
gir_namespace	namespace	string	
gir_version	namespace	string	
has_construct_function	method	bool	
has_copy_function	struct	bool	
has_destroy_function	struct	bool	
has_new_function	method	bool	
has_target	delegate	bool	
has_type_id	class, struct, enum	bool	true
This is used to specify whether a corresponding GType must exists.			
instance_pos	constructor, delegate, method	double	
The argument position of the instance that will be used as `this` in methods.			
lower_case_cprefix	namespace	string	
lower_case_csuffix	class, enum, errordomain, interface	string	
marshaller_type_name	class, struct	string	"BOOLEAN"
notify	property	bool	
ordering	virtual method/ property/signal	int	
Specify the position of the vfunc in the vtable. Once one ordering has been specified in a class, it must be specified for all of the vfuncs.			

param_spec_function	class	string	
pos	parameter	double	
ref_function	class	string	
ref_function_void	class	bool	
Whether the ref function returns void. Default is `false`.			
ref_sink_function	class	string	
ref_sink_function_void	class	bool	
Whether the ref_sink function returns void. Default is `false`.			
returns_floating_reference	method	bool	
Whether the method returns a floating reference to an object.			
sentinel	constructor, method	string	
Sentinel value to use as the last of variadic arguments.			
scope	delegate, parameter	string	"async"
Scope of the delegate as in GIR notation.			
set_value_function	class, struct	string - function name	"g_value_set_boolean"
A function that will set a GValue with an object of this type.			
simple_generics	method	bool	
take_value_function	class	string	
type	class, interface, field, parameter, method	string	
type_check_function	class	string	
type_cname	interface	string	

type_id	class, enum, struct	string	"G_TYPE_BOOLEAN"
The GObject type system type that this object is registered with. If type_id is not specified, Vala uses a type ID based on the type's name.			
type_signature	class, interface, struct	string	
Will be soon moved to DBus.			
unref_function	class	string	
vfunc_name	constructor, method	string	

15.3 Version attribute

Used to annotate symbols with versioning information.

Available since Vala 0.31.1.

Name	Type	Description
since	string	Version number - if used will be checked against locally installed package version.
deprecated	bool	Was `[Deprecated]`
deprecated_since	string	Version number
replacement	string	Symbol name
experimental	bool	Was `[Experimental]`
experimental_until	string	Version number

15.4 SimpleType attribute

This attribute is applied to structs. Consider reading: Vala Manual - Value Types and Vala Tutorial - Value Types.

15.5 BooleanType Attribute

This attribute is applied to structs, combined with SimpleType. Marks the struct as being a boolean type.

15.6 IntegerType Attribute

This attribute is applied to structs, combined with SimpleType. Marks the struct as being an integer number type.

Name	Type
min	integer
max	integer
rank	integer
width	integer
signed	bool

15.7 FloatingType Attribute

This attribute is applied to structs, combined with SimpleType. Marks the struct as being a floating point number type.

Name	Type
decimal	bool
rank	integer
width	integer

15.8 Signal Attribute

This attribute influences the generation and usage of object signals, mostly for the GObject type system. The default is G_SIGNAL_RUN_LAST.

Name	Type
Description (optional)	
detailed	bool
Sets the G_SIGNAL_DETAILED flag.	
no_recurse	bool
Sets the G_SIGNAL_NO_RECURSE flag.	

run	string
Significant values are "first", "last" or "cleanup". Default is "last".	
action	bool
Sets the G_SIGNAL_ACTION flag.	
no_hooks	bool
Sets the G_SIGNAL_NO_HOOKS flag.	

15.9 Description Attribute

This attribute influences the generation and usage of object properties, mostly for the GObject type system.

Name	Type
nick	string
blurb	string

15.10 DBus Attribute

This attribute influences the generation of DBus interfaces (for servers) or DBus calls (for clients) which are generated by Vala.

Name	Applies to	Type	Example
Description (optional)			
name	class, interface, method, property, signal	string	"org.my.interface" or "MyMember"
signature	string		
This makes it possible to use GVariant in D-Bus clients and servers without automatic boxing/unboxing.			
use_string_marshalling	enum	bool	
Marshalling enum values as strings			
value	enumvalue	string	

Marshalling enum values as strings			
use_string_marshalling	enum	bool	
timeout (client only)	method, property	integer	
Timeout is specified in milliseconds			
no_reply	method		
Do not expect a reply from the server			
result (server only)	method	string	
visible (server only)	method, property, signal	bool	
By setting `visible = false` you can specify that the member should not be exported via D-Bus			

15.11 Gtk attributes

GtkTemplate attribute

Can only be applied to classes that inherit from Gtk.Widget. The "ui" argument is mandatory.

Name	Type	Example
Description (optional)		
ui	string (mandatory)	"/org/gnome/yourapp/main.ui"
Specifies the .ui gresource to be used for building the Gtk widget		

GtkChild attribute

Can only be applied to fields of classes being marked with [GtkTemplate]. It's used to connect a field with a child object in the Gtk builder definition.

Name	Type	Example
Description (optional)		
name	string	
Custom name being used in the Gtk builder ui definition. By default the name of the marked field is used.		

internal	bool	
Whether this child is internal or not in the Gtk builder ui definition.		

GtkCallback attribute

Can only be applied to methods of classes being marked with [GtkTemplate]. It's used to connect to a signal defined in the Gtk builder ui with the marked method.

Name	**Type**	**Example**
Description (optional)		
name	string	"on_button_clicked"
Custom name being used in the Gtk builder ui definition. By default the name of the marked method is used.		

15.12 Other attributes

Name	**Applies to**	**Description**
Assert	method	
Compact	class	
ConcreteAccessor	abstract property	Use get/set functions, but do not override them as they are not abstract.
DestroysInstance	method	
Diagnostics	method	
ErrorBase	class	Only use by GLib.Error at the moment
Flags	enum	Marks the enum values to be flags
FormatArg	parameter	specifies that the method takes and returns a printf or scanf format string
HasEmitter	signal	
Immutable	class, struct	

ModuleInit	method	Marks the associated type as dynamic, and marks the method as a TypeModule init function. See TypeModule example
NoAccessorMethod	property	
NoReturn	method	Once the method is called, it will never return
NoThrow	method	Marks methods that can never throw exceptions. Dova profile only
NoWrapper	method	
PointerType		
Print	method	Stringifies and concatenates all arguments you pass to the method
PrintfFormat	method	See also ScanfFormat attribute
ReturnsModifiedPointer	method	
ScanfFormat	method	See also PrintFormat attribute
SingleInstance	class	Makes the class a thread-safe singleton

15.13 Deprecated Attributes

Attributes that have been deprecated and should no longer be used in new code.

Name	Since	Use instead
Deprecated	0.31.1	[Version (deprecated = true, deprecated_since = "", replacement = "")]
Experimental	0.31.1	[Version (experimental = true, experimental_until = "")]
NoArrayLength	0.7.10	[CCode (array_length = false)]

15.14 Examples

TODO: write examples.

16. Preprocessor

16.1 Directives syntax
16.2 Defining symbols
16.3 Built-in defines
16.4 Examples

The Vala preprocessor is a particular part of Vala that acts at syntax level only, allowing you to conditionally write pieces of your software depending upon certain compile-time conditions. Preprocessor directives will never be generated in the resulting code.

16.1 Directives syntax

All preprocessor directives start with a hash (**#**), except for the first line of a file starting with **#!** (used for Vala scripts).

vala-code:
    [any vala code] [pp-condition] [any vala code]

pp-condition:
    `#if` pp-expression vala-code [pp-elif] [pp-else] **`#endif`**

pp-elif:
    `#elif` pp-expression vala-code [pp-elif]

pp-else:
    `#else` vala-code

pp-expression:
    pp-or-expression

pp-or-expression:
    pp-and-expression [**||** pp-and-expression]

pp-and-expression:
    pp-binary-expression [**&&** pp-binary-expression]

pp-binary-expression:
    pp-equality-expression
    pp-inequality-expression

```
pp-equality-expression:
    pp-unary-expression [ == pp-unary-expression ]

pp-inequality-expression:
    pp-unary-expression [ != pp-unary-expression ]

pp-unary-expression:
    pp-negation-expression
    pp-primary-expression

pp-negation-expression:
    ! pp-unary-expression

pp-primary-expression:
    pp-symbol
    ( pp-expression )
    true
    false

pp-symbol:
    identifier
```

The semantics of the preprocessor are very simple: if the condition is true then the Vala code surrounded by the preprocessor will be parsed, otherwise it will be ignored. A symbol evaluates to **true** if it is defined at compile-time. If a symbol in a preprocessor directive is not defined, it evaluates to **false**.

16.2 Defining symbols

It's not possible to define a preprocessor symbol inside the Vala code (like with C). The only way to define a symbol is to feed it through the `valac` option `-D`.

16.3 Built-in defines

Name	Description
POSIX	Set if the profile is posix
GOBJECT	Set if the profile is gobject
DOVA	Set if the profile is dova

VALA_X_Y	Set if Vala API version is equal or higher to version X.Y
DBUS_GLIB	Set if using dbus-glib-1 package

16.4 Examples

How to conditionally compile code based on a `valac` option `-D`.

Sample code:

```
// Vala preprocessor example
public class Preprocessor : Object {

    public Preprocessor () {
    }

    /* public instance method */
    public void run () {
#if PREPROCESSOR_DEBUG
        // Use "-D PREPROCESSOR_DEBUG" to run this code
path
        stdout.printf ("debug version \n");
#else
        // Normally, we run this code path
        stdout.printf ("production version \n");
#endif
    }

    /* application entry point */
    public static int main (string[] args) {
        var sample = new Preprocessor ();
        sample.run ();
        return 0;
    }
}
```

Compile and Run

Normal build/run:

```
$ valac -o preprocessor Preprocessor.vala
$ ./preprocessor
```

Debug build/run:

```
$ valac -D PREPROCESSOR_DEBUG -o preprocessor-debug Preprocessor.vala
$ ./preprocessor-debug
```

17. GIDL metadata format

17.1 Comments
17.2 Other Lines
17.3 Specifiers
17.4 Specifying Different Things
17.5 Properties Reference
17.6 Examples

This section describes the format of .metadata files as used by **vapigen** as additional information for .vapi file generation. Some of the information specified in the metadata can be used to set symbol attributes as well.

17.1 Comments

Comments start with a # and end at the end of a line. For example:

```
# this is a comment
```

17.2 Other Lines

Every non-comment line in the file is made of up two sections: the specifier, and the parameters.

The specifier is the first text to appear on the line, and it specifies what the rest of the line will be modifying.

The parameters are a space separated list of a parameter name, followed by an equals sign and the value enclosed in quotes.

For example, this line sets **parameter1** and **parameter2** on **foo.bar**:

```
foo.bar parameter1="value" parameter2="value"
```

17.3 Specifiers

Specifiers always use the C name for whatever it is you are modifying. For example if your namespace is **Foo**, and the Vala name for the type is **Bar**, then you would use `FooBar`.

Specifiers may also use wildcards, and all items that partially match the specifier will be selected. For example:

```
*.klass hidden="1"
```

will hide the **klass** field in all types.

17.4 Specifying Different Things

To specify a:

Function	`name_of_function`
Type	`Type`
Property	`Type:property_name`
Signal	`Type::signal_name`
Field	`Type.field_name`
Parameter (Function)	`name_of_function.param`
Parameter (Delegate)	`DelegateName.param`
Parameter (Signal)	`Type::signal_name.param`

For example, hiding a symbol:

Type	`Foo hidden="1"`
Function	`some_function hidden="1"`
Field	`Foo.bar hidden="1"`

17.5 Properties Reference

The format for the entries will be like so

Name	Applies To	Values	Description
`foobar`	Signal, Function, Class, Struct, etc	The acceptable values	The description goes here.

And in alphabetical order:

Name	Applies To	Values	Description
`abstract`	Class, Function	0, 1	
`accessor_method`	Property	0, 1	
`array_length_cname`	Field	C identifier	

`array_length_pos`	Parameter (Function)	Double (position between two Vala parameters)	Sets the position of the length for the parameter, length needs to be hidden separately.
`array_length_type`	Parameter (Function), Function (returning an array), Field	C type	
`array_null_terminated`	Function (returning an array), Parameter (Function), Field	0, 1	
`async`	Function	0, 1	Force async function, even if it doesn't end in _async
`base_class`	Class	C type	Marks the base class for the type
`base_type`	Struct	Vala type	Marks the struct as inheriting
`cheader_filename`	Anything (except parameters)	Header include path	Compiler will adds the specified header when thing is used.
`common_prefix`	Enum	String	Removes a common prefix from enumeration values
`const_cname`	Class (non-GObject)	C type	
`copy_function`	Class (non-GObject)	C function name	
`cprefix`	Module	String	
`ctype`	Parameter (Function), Field	C type	

`default_value`	Parameter (Function)	Any Vala value that would be valid for the type	Sets the default value for a parameter.
`delegate_target_pos`	Parameter (Function)	Double (position between two Vala parameters)	
`deprecated`	Anything (except parameters)	0, 1	Marks the thing as deprecated
`deprecated_since`	Anything (except parameters)	Version	Marks the thing as deprecated
`ellipsis`	Function	0, 1	Marks that the function has a variable argument list
`errordomain`	Enum	0, 1	Marks the enumeration as a GError domain
`finish_name`	Function	C function name	Sets custom asynchronous finish function
`free_function`	Class (non-GObject)	C function name	Sets a free function for the struct
`gir_namespace`	Module	String	
`gir_version`	Module	Version	
`has_copy_function`	Struct	0, 1	marks the struct as having a copy function
`has_destroy_function`	Struct	0, 1	
`has_emitter`	Signal	0, 1	
`has_target`	Delegate	0, 1	

has_type_id	Class, Enum, Struct	0, 1	Marks whether a GType is registered for this thing
hidden	Anything	0, 1	Causes the selected thing to not be output in the vapi file.
immutable	Struct	0, 1	Marks the struct as immutable
instance_pos	Delegate	Double (Position between two Vala parameters)	
is_array	Function (returning an array), Parameter, Field	0, 1	Marks the thing as an array
is_fundamental	Class (non-GObject)	0, 1	
is_immutable	Class (non-GObject)	0, 1	
is_out	Parameter	0, 1	Marks the parameter as "out"
is_ref	Parameter	0, 1	Marks the parameter as "ref"
is_value_type	Struct, Union	0, 1	Marks type as a value type (aka struct)
lower_case_cprefix	Module	String	
lower_case_csuffix	Interface	String	
name	Any Type, Function, Signal	Vala identifier	Changes the name of the thing, does not change namespace
namespace	Any Type	String	Changes the namespace of the thing

`namespace_name`	Signal Parameter	String	Specify the namespace of the parameter type indicated with type_name
`no_array_length`	Function (returning an array), Parameter (Function, Delegate)	0, 1	Does not implicitly pass/ return array length to/ from function
`nullable`	Function (having a return value), Parameter	0, 1	Marks the value as nullable
`owned_get`	Property	0, 1	
`parent`	Any module member	String (Namespace)	Strip namespace prefix from symbol and put it into given sub-namespace
`printf_format`	Function	0, 1	
`rank`	Struct	Integer	
`ref_function`	Class (non-GObject)	C function name	
`ref_function_void`	Class (non-GObject)	0, 1	
`rename_to`	Any Type	Vala identifier	Renames the type to something else, ie fooFloat to float (not exactly the same as `name`, AFAIK name changes both the vala name and the cname. rename_to adds the required code so that when the rename_to'ed type is used, the c type is used)
`replacement`	Anything (except parameters)	The thing that replaces this	Specifies a replacement for a deprecated symbol

`sentinel`	Function (with ellipsis)	C value	The sentinel value marking the end of the vararg list
`simple_type`	Struct	0, 1	Marks the struct as being a simple type, like int
`takes_ownership`	Parameter (Function, Delegate)	0, 1	
`throws`	Function	0, 1	Marks that the function should use an out parameter instead of throwing an error
`to_string`	Enum	C function name	
`transfer_ownership`	Function/ Delegate/ Signal (having a return value), Parameter (Function, Signal)	0, 1	Transfers ownership of the value
`type_arguments`	Function/ Delegate/ Signal (having a return value), Property, Field, Parameter	Vala types, comma separated	Restricts the generic type of the thing
`type_check_function`	Class (GObject)	C function/ macro name	
`type_id`	Struct, Class (GObject)	C macro	
`type_name`	Function (having a return value), Property, Parameter, Field	Vala type name	Changes the type of the selected thing. Overwrites old type, so "type_name" must be before any other type modifying metadata

`type_parameters`	Delegate, Class (non-GObject)	Vala generic type parameters, e.g. T, comma separated	
`unref_function`	Class (non-GObject)	C function name	
`value_owned`	Parameter (Function)	0, 1	
`vfunc_name`	Function	C function pointer name	
`virtual`	Function	0, 1	
`weak`	Field	0, 1	Marks the field as weak

17.6 Examples

Demonstrating...

```
// ...
```

18. GIR metadata format

18.1 Locating metadata
18.2 Comments
18.3 Syntax
18.4 Valid arguments
18.5 Examples

The `GIR` format actually has a lot of information for generating bindings, but it's a different language than Vala. Therefore, it's almost impossible to directly map a whole .gir file into a Vala tree, hence the need of metadata. On the other side we might want to use directly .gir + .metadata instead of generating a .vapi, but .vapi is more humanly readable and faster to parse than the GIR, hence the need of vapigen for generating a .vapi.

18.1 Locating metadata

The filename of a metadata for a `SomeLib.gir` must be `SomeLib.metadata`. By default Vala looks for .metadata into the same directory of the .gir file, however it's possible to specify other directories using the `--metadatadir` option.

18.2 Comments

Comments in the metadata have the same syntax as in Vala code:

```
// this is a comment
/*
 * multi-line comment
 */
```

18.3 Syntax

Metadata information for each symbol must provided on different lines:

```
rule:
      pattern [ arguments ] [  relative-rules ]

relative-rules:
      .  pattern [ arguments ] [  relative-rules ]

pattern:
      [ #  selector ] [ .  pattern ]
```

arguments:

[**=** expression] [arguments]

expression:

null
true
false
- expression
integer-literal
real-literal
string-literal
symbol

symbol:

identifier [**.** identifier]

- Patterns are tied to the GIR tree: if a class `FooBar` contains a method `baz_method` then it can be referenced in the metadata as `FooBar.baz_method`.
- Selectors are used to specify a particular element name of the GIR tree, for example `FooBar.baz_method#method` will only select method elements whose name is baz_method. Useful to solve name collisions.
- Given a namespace named `Foo` a special pattern `Foo` is available for setting general arguments.
- If a GIR symbol matches multiple rules then all of them will be applied: if there are clashes among arguments, last written rules in the file take precedence.
- If the expression for an argument is not provided, it's treated as **true** by default.
- A **relative rule** is relative to the nearest preceding **absolute rule**. Metadata must contain at least one absolute rule. It's not possible to make a rule relative to another relative rule.

18.4 Valid arguments

Name	Applies to	Type	Description
skip	all	bool	Skip processing the symbol
hidden	all	bool	Process the symbol but hide from output

type	method, parameter, property, field, constant, alias	string	Complete Vala type
type_arguments	method, parameter, property, field, constant, alias	string	Vala type parameters for generics, separated by commas
cheader_filename	all including namespace	string	C headers separated by commas
name	all including namespace	string	Vala symbol name
owned	parameter	bool	Whether the parameter value should be owned
unowned	method, property, field, constant	bool	Whether the symbol is unowned
parent	all	string	Move the symbol to the specified container symbol. If no container exists, a new namespace will be created.
nullable	method, parameter, property, field, constant, alias	bool	Whether the type is nullable or not
deprecated	all	bool	Whether the symbol is deprecated or not
replacement	all	string	Deprecation replacement, implies `deprecated=true`
deprecated_since	all	string	Deprecated since version, implies `deprecated=true`
array	method, parameter, property, field, constant, alias	bool	Whether the type is an array or not
array_length_idx	parameter	int	The index of the C array length parameter
default	parameter	any	Default expression for the parameter

out	parameter	bool	Whether the parameter direction is out or not
ref	parameter	bool	Whether the parameter direction is ref or not
vfunc_name	method	string	Name of the C virtual function
virtual	method, signal, property	bool	Whether the symbol is virtual or not
abstract	method, signal, property	bool	Whether the symbol is abstract or not
scope	parameter (async method)	string	Scope of the delegate, in GIR terms
struct	record (detected as boxed compact class)	bool	Whether the boxed type must be threaten as struct instead of compact class
printf_format	method	bool	Add the [PrintfFormat] attribute to the method if true
array_length_field	field (array)	string	The name of the length field
sentinel	method	string	C expression of the last argument for varargs
closure	parameter	int	Specifies the index of the parameter representing the user data for this callback
errordomain	enumeration	bool	Whether the enumeration is an errordomain or not
destroys_instance	method	bool	Whether the instance is owned by the method
throws	method	string	Type of exception the method throws

18.5 Examples

Demonstrating...

Overriding Types

When you have the following expression:

```
typedef GList MyList;
```

where `GList` will hold integers, use `type` metadata as follows:

```
MyList type="GLib.List<int>"
```

The above metadata will generate the following code:

```
public class MyList : GLib.List<int> {
    [CCode (has_construct_function = false)]
    protected MyList ();
    public static GLib.Type get_type ();
}
```

Then you can use `GLib.List` or `NameSpace.MyList` as if equal.

Skipping Simbols

```
MySimbol skip
```

More Examples

```
// ...
```