# The Nosica specification book

**David Jobet**

# The Nosica specification book

David Jobet

# Table of Contents

# List of Examples

# Chapter 1. Basic Concepts

## Application startup

An application starts by calling Nosica's entry point. Nosica's entry point is identified by the following signature :

**Example 1.1. Nosica's entry point signature**

```
static sub main(const string[] args);
```

- A Nosica application takes as input a list of string parameters. The size of the array is the number of arguments on the command line plus the name of the executable used to launch the application. Therefore,
  - args[0] returns the name of the executable
  - args[1] return the first argument (if any)
  - ...
  - args[N - 1] returns the last argument (if any)

  With N the size of the array (args.length)
- Before the actual main of a Nosica program is called, the static initializer of each classes get called. The order in which they get called is not specified and is implementation dependant.

## Application termination

- A Nosica application cannot return a value. It must use the System.exit method to do so. If it is not used at all, then the default return value is used (which is 0).
- After the actual main of a Nosica program is executed, the static deinitializer of each classes get called. The order in which they get called is not specified and is implementation dependant.

## Declarations

A Nosica source file is made of several parts :

- An optional package declaration
- An optional list of import declarations
- One or more top level Nosica declaration. A top level Nosica declaration is either a class, an interface, an enum, or a metadata type description.

Nosica's top level declarations get their namespaces name from the package in which they get defined, and their own name. Their complete name must match the filesystem's topology.

### Example 1.2. TypeName constitution

```
package package1;
class Test {}
```

In this example, the complete name of class Test is package1.Test.

It is possible to refer to an other type by their complete name, or by a shorter name provided they get imported. By default, when importing another type, the short name is the name of the type. It is possible to provides an alternate short name.

### Example 1.3. Importing a class

```
import package1.Test;
import package1.Test MyTest;
```

The first import form allows one to import package1.Test. The rest of the program can refer to it directly via the name "Test". The second import form allows one to import a type and provide its own custom short name. This is particularly usefull when several classes have same name in different packages.

It is possible to define several classes in the same Nosica source file, provided they bear the same name. This is only usefull with generic classes and partial specialisation. This is of no use for interfaces, enum and metadata.

### Example 1.4. Generic class declaration

```
class HashMap<T>
{
  // default implementation
}

class HashMap<string>
{
  // specific implementation for strings
}
```

The first form defines a class HashMap that can be applied on any type T. The second form specifies a custom implementation to use when T is in fact a string.

# Members

Classes, Interfaces, enums and Metadata have members.

# Interface members

Interfaces can contain :

- methods

- properties
- operators
- nested type declaration

# Class members

Classes can contain :

- at most static initializer
- at most static deinitializer
- constructors
- at most one destructor
- methods
- properties
- operators
- cast operators if type is primitive
- fields
- nested type declaration

# Enum members

An enum is composed of a list of symbolic typed constants. An enum provide a list of transformation functions to/from int/string.

### Example 1.5. Enum

```
enum Color {
  RED,
  GREEN,
  BLUE
}
```

The former example can be seen as syntactic sugar for the following :

```
class Color implements Enum<Color> {
  private int id;
  private constructor(int id) { this.id = id; }
  private static string[] strings = {"RED", "GREEN", "BLUE"};

  static public Color RED = new Color(0);
  static public Color GREEN = new Color(1);
  static public Color BLUE = new Color(2);

  string toString() { return strings[id]; }
  int toInt() { return id; }
```

```
 static Color fromString(string id) {
  if (id.equals("RED")) return RED;
  if (id.equals("GREEN")) return GREEN;
  if (id.equals("BLUE")) return BLUE;
  return null;
 }
 static Color fromInt(int id) {
  switch (id)
  {
  case 0 : return RED;
  case 1 : return GREEN;
  case 2 : return BLUE;
  default : return null;
  }
 }
}
```

## Metadata members

@TODO@

# Member access

By default a member has "package" access. That means, it is accessible only by members of the same package. All Nosica top declarations can have either package or public access.

Additionally, it is possible to define finer grain accessibility for classes members :

- public : accessible from anywhere
- protected : accessible only from derived classes
- private : accessible only by members of enclosing class

# Signatures and overloading

Uniqueness of a member (its signature) is defined by several properties :

- member's type
- member's name
- member's arguments
- member's modifiers

an argument is defined by its type and its varness. ("var" keyword)

member's modifier are staticness and varness. ("static" and "var" keywords)

It is possible to define several members having several same properties, provided at least one property is different. If two members have strictly same properties, this is a compil time error.

# Scopes

Basically, in a nested scope, you cannot hide a name of an enclosing scope. One exception to this is fields : as they can be accessed via the 'this' variable, you're allowed to hide them by a local variable or a method's argument.

The scope of a variable is its enclosing block.

The scope of an argument is the method.

The scope of a field is its class and all methods defined in the class and all inner sub classes (not nested classes).

The scope of variables defined inside a for or foreach statement is the block of the statement.

# Automatic memory management

The memory management for an object starts when the object is created via the "new" static method of the type

1. memory is allocated for it, and the constructor is run
2. when the object is no longer in used (the last reference to it reaches end of scope or is assigned another value), then the destructor must be run and the memory reclaimed

Note that the contraint "as soon as" imposes a reference counting algorithm. Thus it does not handle circle references. This may change in the future.

# Covariance

When implemening a method of an interface :

Result types are allowed to be covariant.

Input parameters are allowed to be covariant if and only if a method with invariant parameters is defined.

### Example 1.6. covariant example

```
class A {}
class B extends A {}

interface I {
  sub f(A a);
}

class IImpl1 implements I {
  public sub f(A a) {} // OK IImpl1.f really implements I.f (arguments are invariant)
}

class IImpl2 implements I {
  public sub f(B b) {} // Error IImpl2.f does not implement I.f (arguments are covariant)
}

class IImpl3 implements I {
  public sub f(A a) {} // OK IImpl1.f really implements I.f (arguments are invariant)
  public sub f(B b) {} // OK IImpl2.f can implements I.f with covariant arguments because IImpl3.f with invariant argu
}
```

# Dispatching

Dispatching is done on all arguments.

Most of the time, dispatching will be done only on the first argument (the 'this' argument), but provided several methods with covariant arguments exist, all necessary arguments will be taken into account to perform the dispatching.

### Example 1.7. Multiple dispatch

```
Multiple dispatchclass Toto {
  public static f(A a) {}
  public static f(B b) {}
}

A a = new B();
Toto.f(a); // will call Toto.f(B)
```

# Automatic delegation

Fields can be marked as proxy of a type.

The provided type must be a super type of the field's type.

All methods of the given interface are automatically "added" to the enclosing type, and the implementation consists of a delegation to the field.

### Example 1.8. Proxy example

```
interface I
{
  sub f();
  int g();
}

class IImpl implements I
{
  public () f() { Console.out << "IImpl.f\n"; }
  public int g() { Console.out << "IImpl.g\n"; return 1; }
}

class A
{
  private IImpl myField proxies IImpl;
}
```

is equivalent to :

```
class A
{
  private IImpl myField;

  public sub f() { myField.f(); }
  public int g() { return myField.g(); }
}
```

Therefore, this feature can be used to emulate multiple inheritance :

```
class A extends AbstractA implements I
{
  private IImpl myField proxies I;
}
```

It is always possible to explicitly implement a method delegated to the proxy.

```
class A extends AbstractA implements I
{
  private IImpl myField proxies I;

  public sub f() { Console.out << "A.f\n"; }
  // g is still forwarded to myField
}
```

# Anonymous classes

Their implementation is defined "inline" with the allocation statement. Anonymous classes are inner types.

```
interface I
{
  sub f();
}

class SomeClass
{
  void someMethod()
  {
    I i = new I() {
      public sub f() { Console.out << "Anonymous I.f\n"; }
    }
  }
}
```

This is equivalent to

```
class SomeClass
{
  void someMethod()
  {
    I i = new AnonymousClass1();
  }

  private inner class AnonymousClass1 implements I
  {
    public sub f() { Console.out << "Anonymous I.f\n"; }
  }
}
```

# Signal/slot, closure and continuation

Any method can act as a slot.

Any signal can be connected to any slots, provided their signature matches.

To define a signal, one has just to add the "signal" keyword in front of the method with an empty implementation.

A method must be seen as an implementation of an inner type whose interface is Method<TupleOut, TupleIn> where TupleOut is the result type of method and TupleIn is the list of parameter of the method.

As such, a method is a full closure, and can be used for continuation.

### Example 1.9. Methods and Method instantiation

```
class Foo
{
  public () bar(int i)
  {
    Console.out << "Coucou";
  }
}
```

is equivalent to

```
class Foo
{
  Method<(), (int)> bar = new Method<(), (int)>
  {
    () operator()(int i)
    {
      Console.out << "Coucou";
    }
  }
}
```

The interface Method is defined as

```
interface Method<Result, Parameter>
{
  Result operator()(Parameter p);
}
```

# Genericity

A class, an interface or a method can be made generic by adding a generic declaration following the name of the item.

```
class Vector<T>
{
}

interface Container<T>
{
}

class A
{
  public sub f<T>() {
  }
```

```
}
```

A generic declaration can provide one constraints over the generic type :

```
class HashMap<T inherits Hashable>
{
}
```

The constraints can be a class or an interface. By default, a generic parameter implements Object. That is, writing :

```
class Vector<T>
{
}
```

is the same as :

```
class Vector<T inherits Object>
{
}
```

# Chapter 2. Types

Types are divided into two main categories : value types and reference types.

# Value types

A value type is either a primitive type, a tuple or an enum.

## Default constructor

If the type defines no default constructor, a default is created for it. The purpose is to allow the value type to be instantiable by default.

- All reference fields composing the value type are initialised to zero.
- All value fields composing the value type get their default constructor called if it exists. If that's not the case, this is a compil time error.
- Native types of net.nosica.lang do not get initialised to zero.

## Primitive type

A primitive type is a class declared with the "primitive" keyword. It can contains

- at most one static initializer
- at most one static deinitializer
- zero, one or more constructors (if none are defined, a default is created)
- at most one destructor
- methods
- operators
- properties
- cast operators
- fields

### Example 2.1. Sample primitive type

```
primitive class complex
{
  public float32 real;
  public float32 imaginary;

  public complex operator +(complex c) {
    complex result;
    result.real = real + c.real;
    result.imaginary = imaginary + c.imaginary;
```

```
   return result;
  }

  public complex float32.operator +(complex c) {
   complex result;
   result.real = this + c.real;
   result.imaginary = c.imaginary;

   return result;
  }
}
```

# Enum types

Enum types contains a list of symbolic constants.

An enum types implements the interface Enum.

An enum has its proper type and can be converted to/from string.

# Tuple types

Tuple types are built-in types. They are generated on-the-fly by the compiler, pretty much like arrays. Tuples implements the Tuple interface.

A tuple is a lightweight value type holding one or more anonymous variables. The notation is :

```
(int, string, Object) tuple;
```

and this is equivalent to write :

```
 primitive class AnonymousTuple
 {
  public int    anonymous0;
  public string anonymous1;
  public Object anonymous2;
 }
```

each component of the tuple can be made mutable or not via the "var" modifier. Thus, one can write :

```
(var int, string, var Object) tuple;
```

## Tuple binding

it is possible to bind in a one to one relationship a list of variable and a tuple.

### Example 2.2. Binding example

```
(int i, float f) = (1, 2.30);
```

This is equivalent to :

```
int i;
float f;
(i, f) = (1, 2.30);
```

Thus, when calling a method, it is both possible to use normal parameter list (or variable list), or a tuple.

# References type

References type are either classes, interfaces or Arrays.

References types are garbage collected.

Default references types include

- Object
- Array
- None

# Classes type

A class type defines a data structure plus a set of methods working on the data structures.

Members of a class types are

- fields
- static initializer
- static deinitializer
- constructors
- destructor
- methods
- properties
- operators

A class can extends at most one other class (simple inheritance) and multiple interfaces.

If a class does not explicitly extends a class, it implicitly extends the class Object. Therefore, all classes directly or indirectly inherits from class Object.

There is a special class named None which implicitly extends all known classes of the compiled program. None is the type of the null literal.

# Interfaces type

An interface defines only a set of methods. There is no instance of an interface. There is only instances of classes that implements interfaces. Interfaces defines a kind of contract to which a class must adhere.

An interface can extends several other interfaces. A class can implements as much interfaces as it wishes.

# Arrays type

Arrays are built-in types. They are generated on the fly by the compiler. They implements the Array interface.

Arrays have at least one dimension but they are not limited to one dimension.

### Example 2.3. Array example

```
int[,] i = new int[10, 10];
```

# Boxing

Each value types can be boxed via the Box generic class.

The class Box is defined like :

```
class Box<T> {
  public T value proxies T;
}
```

### Example 2.4. Box example

```
sub f(Object o) {}

int i = 0;
f(i); // calls with f(new Box<int>(i));
```

There is no unboxing. The user must test the instance against the Box<T> type and access the underlying value.

Please note that the underlying value is immutable.

Ultimately, the Box type allows the type system to unify value types and references type because ultimately all types can be converted into an Object.

# Chapter 3. Variables

There are several different types of variables in Nosica : static fields, fields, parameters and local variables.

Variables have a type, possibly an array or a tuple type. Variables may have modifiers : the "var" modifier or the "static" modifier.

Initial value of array elements and tuple's members is the default value.

# Static fields

Static fields are defined with the "static" keyword. They exist before application startup and can be accessed at any time. They cease to exit after application shutdown.

Initial value of a static field is the default value.

# Fields

Fields are members of a class. They are defined inside a class without the "static" keyword.

Fields are created when the instance of the class is created. They cease to exist after the destructor has been executed.

Initial value of a field is the default value.

# Parameters

Parameters can be given in input or in output as in the following syntax :

### Example 3.1. Input and output parameters

```
(int i, Object o) someCall(float f, Array a) {
  i = f.narrow();
  o = a;
}
```

## Input parameters

There can be value parameters or references parameters.

A value parameter is a parameter defined without the "var" modifier. It means the parameter is immutable and cannot be modified.

A reference parameter is defined with the "var" modifier. A reference parameter does not create a new storage location. Thus the value of a reference parameter is always the same as the underlying variable used to perform the call.

### Example 3.2. Value and reference parameters

```
int i = 0;
int j = 0;
```

```
int[] a1 = int[].new(10);
int[] a2 = int[].new(10);

f(i, j, a1, a2);

sub f(int i, var int j, int[] a1, var int[] a2)
{
  j += i;
  for (int a = 0; a < a1.length; ++a)
  {
    a2[a] += a1[a];
  }
}
```

## Output parameters

Output parameters are always mutable.

Much like references parameters, output parameters do not create a new storage location. Instead, they are bound to the variable receiving the value in the caller. If no such variable exist, a new storage location is created in this sole purpose.

### Example 3.3. output parameters

```
(int i, float j) = f();

(int i, float j) f()
{
  i = 0;
  j = 0;
}
```

# Local variables

A local variables can be declared anywhere inside a block. Some special statements like the for or foreach statement allows the creation of a local variable inside their declaration.

The lifetime of a local variable is limited by the one of its enclosing block. When its enclosing block ends, the variable is said to have reached end of scope and is destroyed.

The initial value of a local variable is the default value.

# Chapter 4. Conversions

A conversion enables one type to be treated as another. Conversions can be implicit or explicit.

# Implicit conversions

Here are the classified implicit conversions :

- Implicit reference upcast conversion
- Implicit primitive cast conversion
- Implicit primitive boxing conversion
- Implicit reference immutable conversion

## Implicit reference upcast conversion

It is possible to convert any reference type to one of its super type. A super type being one of the reference type listed in the extends or implements declaration of the type, recursively.

It is possible to convert an array type TE with an element type E to an immutable array type TS with an element type S provided S is a super type of E.

It is possible to convert a generic type TE with a generic parameter E to an immutable generic type TS with a generic parameter S provided S is a super type of E.

As None is the type of the literal 'null' and None implictly inherits from all existing types of the program, it is therefore possible to assign any variables the 'null' literal.

As Object is the super type of all references types, it is possible to convert any reference type to Object.

## Implicit primitive cast conversion

A conversion is allowed from a primitive type P1 to another primitive type P2 provided that P1 defines a cast operator to P2.

### Example 4.1. Cast operator

```
primitive class int8
{
  int32 cast();
}
```

The user should not provide cast operators that loses data. Safe conversion are defined in the net.nosica.lang packages for the integral type via cast operators.

Unsafe conversion (conversion that loses data) should be declared via explicit narrow() methods.

### Example 4.2. Narrow method

```
primitive class int32
{
  int8 narrowToInt8();
}
```

## Implicit primitive boxing conversion

A primitive type can be converted into a reference type via the boxing conversion.

#### Example 4.3. Boxing conversion

```
Object o = 1; // is equivalent to Object o = new Box<int>(1);
```

## Implicit reference immutable conversion

A mutable reference variable is allowed to be converted into an immutable reference variable. The reverse is forbidden though.

Value types are always allowed to be converted from/to mutable/immutable variables as thet are copied.

# Explicit conversions

Here are the classified explicit conversions :

- explicit reference downcast conversion
- explicit reference upcast conversion
- explicit primitive cast conversion

## Explicit reference downcast conversion

explicit downcast conversion are allowed via the trycast statement.

#### Example 4.4. downcast explicit conversion

```
Object o = f();
trycast (o as A a)
{
}
else
{
}
```

## Explicit reference upcast conversion

explicit upcast conversion are allowed via the traditional cast statement.

### Example 4.5. upcast explicit conversion

```
A a = new A();
Object o = (Object)a;
```

## Explicit primitive cast conversion

It consists in manually calling the cast operator with the cast expression.

### Example 4.6. Explicit primitive cast conversion

```
int8 j = 0;
int32 i = (int32)j;
```

# Chapter 5. Expressions

There are unary, binary, ternary operators and N-ary operators Assignment : = ~ *= /= %= += -= @TODO@

- Primary : x.y f(x) a[x] T.new
- Unary : ++x --x +x -x !x (T)x
- Multiplicative : * / %
- Additive : + -
- Stream : << >>
- Relational : < > <= >=
- Equality : == != ~~ !~
- Conditional AND : &&
- Conditional OR : ||
- Implies : =>
- Conditional : ?:
- Assignment : = ~ *= /= %= += -=

@TODO@

# Chapter 6. Statements

- Statement list and blocks
- Labeled statements
- Local variable declaration
- Expression statement
- If statement
- switch statement
- while statement
- do statement
- for statement
- foreach statement
- break statement
- continue statement
- return statement
- throw and try statement

@TODO@

# Chapter 7. Namespaces

Namespaces are implicitly defined in Nosica using the package declaration.

The package of a nested type is the complete TypeName of the enclosing type.

It is possible to import an alias into a compilation unit using the import declaration.

# Compilation Unit

The compilation unit is the structure of a Nosica file. It consists of an optional top level package declaration, followed by a list of zero or more import declarations, followed by a list of one or more type declarations.

CompilationUnit ::=
  [PackageDeclaration]
  (ImportDeclaration)*
  (TypeDeclaration)+

# Package declaration

A package declaration defines the enclosing typename of a type's complete typename.

The syntax is as follows :

PackageDeclaration ::=
  "package" TypeName ";"

The compiler will check the file is effectively located into the package defined relatively to the given sourcepath.

### Example 7.1. file structure and package

As an example, suppose we have defined the sourcepath to contain the path

/home/joebar/project

and you define a file named Toto.nos in /home/joebar/project/net/myorg/Toto.nos, then the relative path between the sourcepath and the file location is net/myorg/Toto.nos. Therefore, the package declaration to use should be :

package net.myorg;

# Import declaration

An import declaration import a symbol from an outer package inside the current compilation unit. There are two forms of import package : the short and extended form. The syntax is as follows :

ImportDeclaration ::=
  "import" TypeName [Id];

The short form would be :

import net.myorg.Toto;

Whilst the extended form would be :

import net.myorg.Toto Toto;

The following two examples have exactly the same effect : the class Toto is now available with a short name "Toto", but the long full qualified name is always available : net.myorg.Toto. The difference between the short and the extended form is that in the short form, the chosen alias is always the last part of the fully qualified name, whilst with the extended form you are free to chose the name you want.

## Single import

If the import declaration specifies a class, an interface, an enum, or a metadata, only the specified entity is imported in the current compilation unit.

## Package import

It is possible to import a whole package at once. Just specify the package you want to import.

It is forbiddent to use the extended import form to specify an alternate name for the package.

The package import is equivalent to manually importing all elements of the package.

#### Example 7.2. Package import

import net.nosica.lang;

## Implicit imports

Each compilation unit implicitly imports two packages :

- net.nosica.lang package
- current package

The purpose is to simplify access to simple types like int, float, string and the likes and to allow the user to access easily related types defined in the same package as the current compilation unit.

Those default packages takes precedence over user defined imports. It is a compil time error to try to import a unit under an already defined import name.

# Type declaration

The type declaration can either be a ClassDeclaration, an InterfaceDeclaration, an Enum declaration or a MetaDataType declaration.

Each type declaration defines a name. That name added to the package in which the type is defined forms the fully qualified typename.

Additionnaly, the name of the declared type must match the one of the file in the sourcepath. The case is important.

# Class declaration

**Example 7.3. Class declaration**

```
package net.myorg;

class Toto
{
}
```

# Interface declaration

**Example 7.4. Interface declaration**

```
package net.myorg;

interface Totoable
{
}
```

# Enum declaration

**Example 7.5. Enum declaration**

```
package net.myorg;

enum Color
{
  RED, GREEN, BLUE
}
```

# MetadataType declaration

@TODO@

# Chapter 8. Classes declaration

The syntax is as follows :

```
ClassDeclaration ::=
  (AccessModifiers | ClassModifiers) "class" id [GenericDeclaration] [ "extends" TypeName ] [ "implements" TypeNa
    (ClassBodyDeclaration)*
  "}"

AccessModifiers ::=
  "public"
| "private"
| "protected"

ClassModifiers ::=
  "abstract"
| "final"
| "primitive"
| "native"

ClassBodyDeclaration ::=
  StaticInitializer
| StaticDeinitializer
| ConstructorDeclaration
| DestructorDeclaration
| MethodDeclaration
| PropertyDeclaration
| OperatorDeclaration
| FieldDeclaration
```

As a class can be generic, it is possible to define several classes in the same compilation unit. In that case, there must be one and only one complete generic declaration. The other classes must be generic specialisation classes.

Specialisation are allowed to be put in other files bearing the same fully qualified name. They must be defined in a distinct sourcepath.

# Field Declaration

The syntax is as follows

```
FieldDeclaration ::=
  (AccessModifiers | FieldModifiers)* TupleDeclaration id ";"

FieldModifiers ::=
  "static"
| "var"
| "mutable"
```

Fields are the constituent piece of classes.

Static fields are classes members available at anytime. They are created before program startup and are destroyed after program termination. Static fields are available and already initialised to their default values when the static initializer of the class is executed. Static fields are available when the static deinitializer of the class is executed and are destroyed after the static deinitializer is finished.

Instances fields (non static) are created and initialized to their default value before the instance constructor is run. Instances fields are available when the destructor of the instance is run. They are des-

troyed after the destructor's execution.

By default, fields are immutable. To make them mutable, one has to use the "var" keyword.

To make a field mutable in an immutable method, the fields has to be further marked as "mutable".

# Method Declaration

The syntax for method is the generatl syntax for other method-like entities :

```
MethodDeclaration ::=
  (AccessModifier* | MethodModifiers) ResultType [TypeName "."] Id Arguments ["var"] [ThrowsDeclaration] (";" | B

MethodModifiers ::=
  "static"
| "final"
| "signal"

ResultType ::=
  "sub"
| "(" [ResultTypeDeclaration ("," ResultTypeDeclaration)*] ")"

ResultTypeDeclaration ::=
  TupleDeclaration Id

Arguments ::=
  "(" [Argument ("," Argument)*] ")"

Argument ::=
  ["var"] TupleDeclaration Id
```

A method can be made "static". In that case, it is called a class method. If a method is not static, it is said to be an instance method.

By default, a method work on an immutable object. To make a method work on a mutable object you have to suffix it with the "var" modifier.

# Parameters

An argument with a "var" modifier is sait do be a reference parameter. An argument without a "var" modifier is said to be a value parameter.

## Value parameter

Value parameters are equivalent to local variable except that they get their values from the caller. Value parameters are immutable.

## Reference parameter

A reference parameters does not create a local storage. It represents the same storage as the one used to make the call. Reference parameters are always mutable.

## Output parameters

Output parameters are always mutable. They exist when entering the method and are initialised to their default value.

# Prefixing a method's name with a TypeName

Method can be prefixed with a TypeName. If the typename T is a super type of the enclosing type, then the method is overloading the enclosing type's method.

### Example 8.1. Specifically overloading a method

```
interface Base1 {
  sub f();
}

interface Base2 {
  sub f();
}

class Derived implements Base1, Base2 {
  public sub Base1.f() {}
  public sub Base2.f() {}
}
```

It allows one to specifically choose the overloading.

If the typename T is an unrelated type (not a super type), then the method is said to be added to the type T. Specifically :

- the method is directly accessible via the type T, but
- the method really belongs to E : that means normal access rules applied for private/public/protected access.

### Example 8.2. Adding a new method to an existing class

```
public class OStream {
  public OStream append(int i);
  public OStream append(float32 i);
  // ... other methods
}

public class Color {
  private int r;
  private int v;
  private int b;

  public OStream OStream.append(Color c) {
    this.append(c.r);
    this.append(c.v);
    this.append(c.b);

    return this;
  }
}
```

# Operator Declaration

The syntax is as follows

```
OperatorDeclaration ::=
  [AccessModifiers] [TypeName "."] "operator" OperatorName Arguments ["var"] [ThrowsDeclaration] (";" | Block)

OperatorName ::=
  UnaryOperators
| BinaryOperators
| NaryOperators

UnaryOperators ::= "-" | "+" | "--" | "++" | "!"

BinaryOperators ::= "-" | "+" | "*" | "/" | "%" | "^" | "~" | "-=" | "+=" | "*=" | "/=" | "%=" | "^=" | "~~" | "!~" | "<<" | ">>"

NaryOperators ::= "()"
```

Unary operators are always prefix.Binary operators are always infix.

There is no such things as postfix operator as this is handled by the more general method notation.

See the example to add methods in an existing class.

Unary operators have a ()->T signature.

## Example 8.3. Unary operator signature

```
class T {
  private int i;
  public constructor(int i) {this.i = i; }
  public T operator -() {
    return T(-i);
  }
}
```

Binary operators have generallt a T->T signature.

## Example 8.4. Binary operator signature

```
class T {
  private int i;
  public constructor(int i) {this.i = i; }
  public T operator -(T rhs) {
    return T(i - rhs.i);
  }
}
```

However, the copy operator has a special signature which is T->().

## Example 8.5. Copy operator signature

```
class T {
  private int i;
  public constructor(int i) {this.i = i; }
```

```
  public sub operator ~(T rhs) {
    this.i = rhs.i;
  }
}
```

# Property Declaration

The syntax is as follows

```
PropertyDeclaration ::=
  "property" TupleDeclaration id "{"
    [ [AccessModifiers] "get" (";" | Block) ]
    [ [AccessModifiers] "set" (";" | Block) ]
  "}"
```

A property declaration act as a Field, but it completes the field declaration with accessors : a get accessor if the equivalent field is to be readable, and a set accessor if the equivalent field is to be writable. If the PropertyDeclaration defines only a get accessor, the equivalent field is said to be read-only. If the PropertyDeclaration defines only a set accessor, the equivalent field is said to be write-only.

In the set form, the implicit argument is Id (the id used to define the property).

### Example 8.6. Set property

```
property int i {
  public get { return 1; }
  protected set { Console.out << i << "\n"; }
}
```

# Array properties

Array properties are like 'normal' properties except they modelize access to an array.

Array properties can have a name or they can be anonymous.

The syntax is as follows :

```
ArrayPropertyDeclaration ::=
  "property" TupleDeclaration [id] Arguments "{"
    [ [AccessModifiers] "get" (";" | Block) ]
    [ [AccessModifiers] "set" (";" | Block) ]
  "}"
```

When the array property is anonymous, then the implicit parameter is named "value".

### Example 8.7. Anonymous array access, or how to define a generic class "Map"

```
class Map<K, V> {
  property V[](K key) {
    public get { return redBlackTree.getValue(key); }
    public set { redBlackTree.setKeyValue(key, value); }
```

```
  }
}
```

# Constructor Declaration

The syntax is as follows

```
ConstructorDeclaration ::=
  [AccessModifiers] "constructor" Arguments [ThrowsDeclaration] (";" | ConstructorBlock)

ConstructorBlock ::=
  "{"
  [ExplicitConstructorInvocation]
  (BlockStatements)*
  "}"

ExplicitConstructorInvocation ::=
  "this" FormalParameters
|
  "super" FormalParameters
```

Fields are initialised to their default values when entering the ConstructorDeclaration.

# Destructor Declaration

The syntax is as follows

```
DestructorDeclaration ::=
  "destructor" "("")" (";" | Block)
```

# Static initializer Declaration

The syntax is as follows

```
"initializer" "("")" (";" | Block)
```

# Static deinitializer Declaration

The syntax is as follows

```
"deinitializer" "("")" (";" | Block)
```

# Chapter 9. Array declaration

Array can have multiple dimensions.

```
int[] i = int[].new(10);
int[,] j = int[,].new(10, 10);
int[][] k = int[][].new(10);
for (int l = 0; l < k.length; ++l)
  j[l] = new int[].new(10);
```

Arrays implement the Array interface.

```
public interface Array<T>
{
  word length {
    get;
  };
  word dimension {
    get;
  };
  word length(word dim);
  T[] {
    get;
    set;
  }
}
```

In this interface, only mono dimensional array get/set properties are declared. A real array type will have two set of array get/set accessors : a mono dimensional pair of accessors, and a multi dimensional pair of accessors if the type is multi dimensional.

This allows to represent all multi dimensional arrays with a mono dimensional representation suitable for iterations for example.

This also allow to have one super type for all arrays.

# Chapter 10. Interface declaration

Here's the syntax

```
InterfaceDeclaration ::=
  [AccessModifiers] "interface" id [GenericDeclaration] ["extends" TypeNameList] "{"
    (InterfaceMemberDeclaration)*
  "}"

InterfaceMemberDeclaration ::=
  MethodDeclaration
| PropertyDeclaration
| OperatorDeclaration
```

# Chapter 11. Enum declaration

Here's the syntax

```
EnumDeclaration ::=
  [AccessModifiers] "enum" Id "{"
    IdList
  "}"

IdList ::=
  [Id ("," Id)* ]An enum declaration contains one or more symbolic typed constant.All enums implements the inteface I
  string toString();
  sub fromString(string str);
  int toInt();
  sub fromInt(int id);
}
```

# Chapter 12. Tuple declaration

Here's the syntax

```
TupleDeclaration ::=
  TupleMember
| "(" TupleMembers ")"

TupleMembers ::=
  [TupleMember ("," TupleMember)* ]

TupleMember ::=
  TypeName
|  TupleDeclaration
```

Each tuple implements the tuple interface which is just empty.

A tuple is a primitive class with no methods and N anonymous fields. The Ith field has the type of Ith typename of the tuple.

There is no way to access directly the members of a tuple.

# Chapter 13. Exceptions

@TODO@

# Chapter 14. Attributes

@TODO@

# Appendix A. GNU Free Documentation License

Version 1.2, November 2002



0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sen

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in

We have designed this License in order to use it for manuals for free software, because free software needs free docu

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright hol

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied ve

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with th

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sect

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in th

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, La

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, leg

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains X

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Do

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a ma

It is requested, but not required, that you contact the authors of the Document well before redistributing any large nu

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, p

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of
* B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modificati

* C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
* D. Preserve all the copyright notices of the Document.
* E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
* F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Mod
* G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Docu
* H. Include an unaltered copy of this License.
* I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, n
* J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the D
* K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserv
* L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers
* M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
* N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Sect
* O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and con

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified V

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for public

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in secti

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be re

In the combination, you must combine any sections Entitled "History" in the various original documents, forming on

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replac

You may extract a single document from such a collection, and distribute it individually under this License, provide

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is les

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this Licen

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from tin

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numb