Mir Language Reference

Yury V. Vishnevskiy

Version 1.0, 2023-07-19

Table of Contents

Introduction

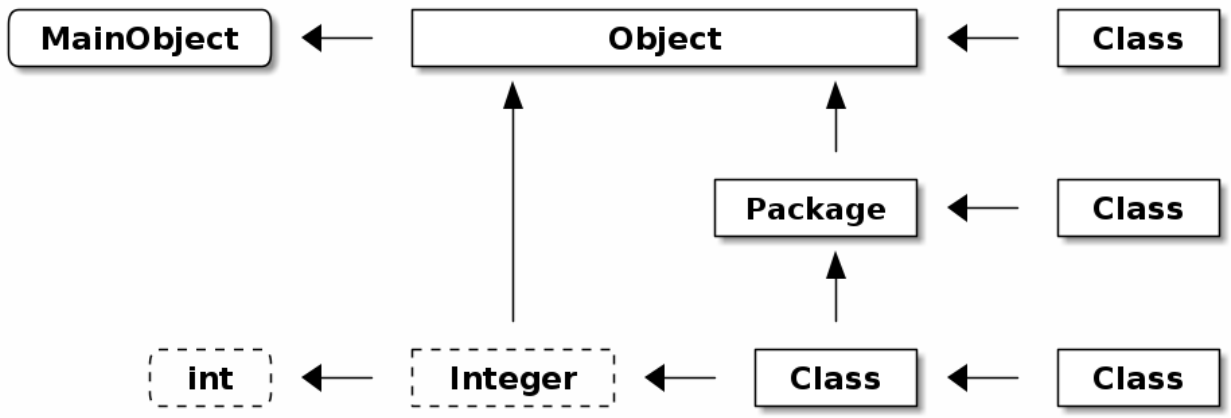
Mir (Rus: МИР, Мы Изучаем Реальность) is an object-oriented programming language. This manual documents basic aspects of Mir, the data and execution models, syntax (lexical structure and grammar) and core objects.

Data model

All units of data are objects, regardless whether they are classes or instances of classes. There are predefined most basic fundamental objects, built-in core objects (basic classes and some of their instances), objects from standard library and user-defined objects, which may be created during program execution.

Fundamental objects

Most important fundamental objects are their relationships determine functionality of Mir. Below is the diagram of these objects.

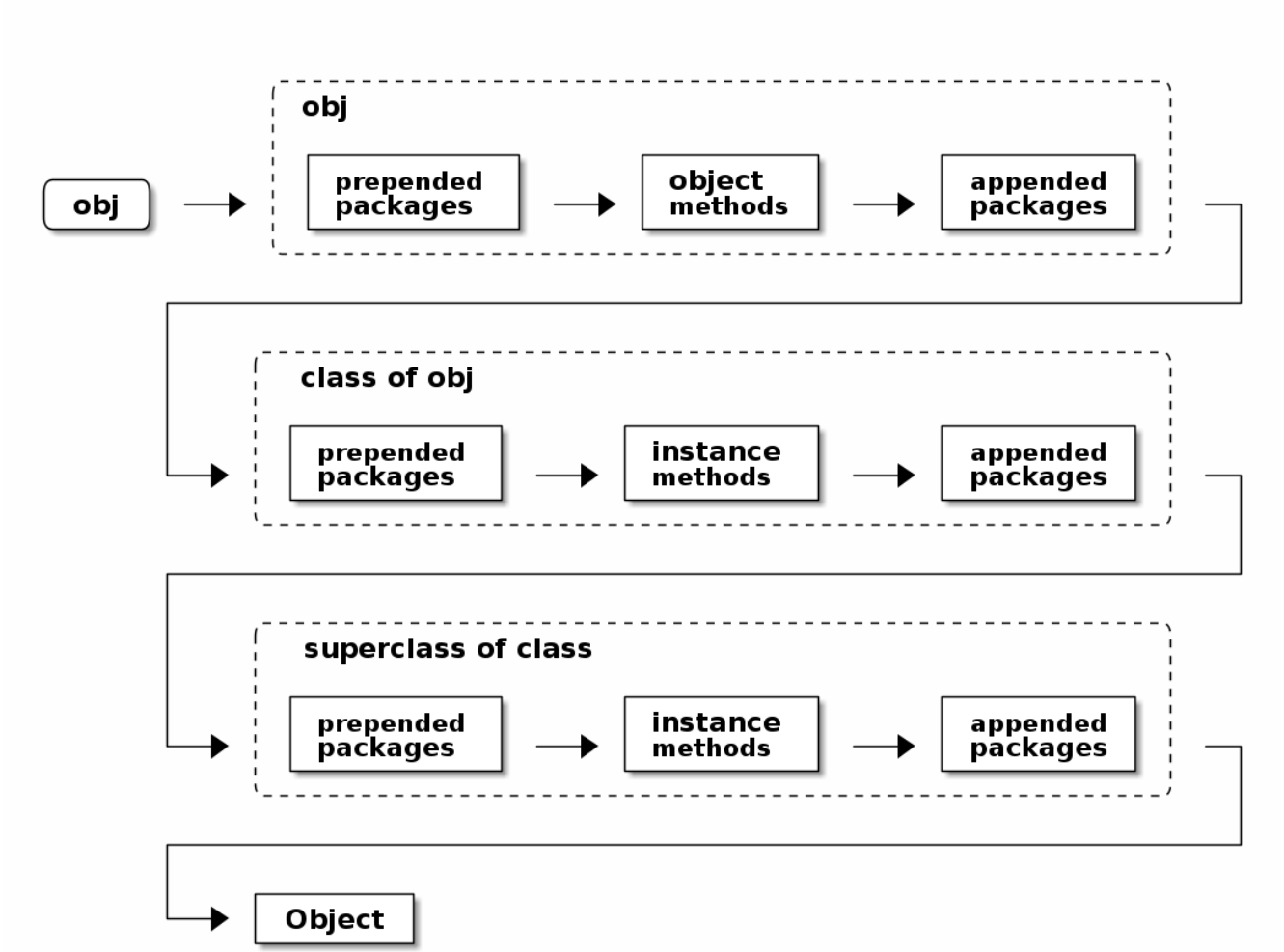


`Object`, `Package` and `Class` are the fundamental classes (shown in boxes) in Mir. `MainObject` is the fundamental object, which is not a class (shown in the box with rounded corners). In the dashed boxes the core class `Integer` and an integer number (denoted as `int`) are shown as examples of how other classes and their instances are related to the fundamental objects. Arrows show relationships of two kinds. Horizontal arrows show relationships between classes and their instances; vertical arrows are for classes and their superclasses, as depicted below.

In this diagram the instance of a class is shown as an object in the box with rounded corners. It is, however, possible that an instance is also a class, as shown above for the fundamental classes. Here there is no contradiction, since each class can be considered as an object from functional point of view.

Method lookup path

The standard method lookup procedure for objects is as in the diagram below.



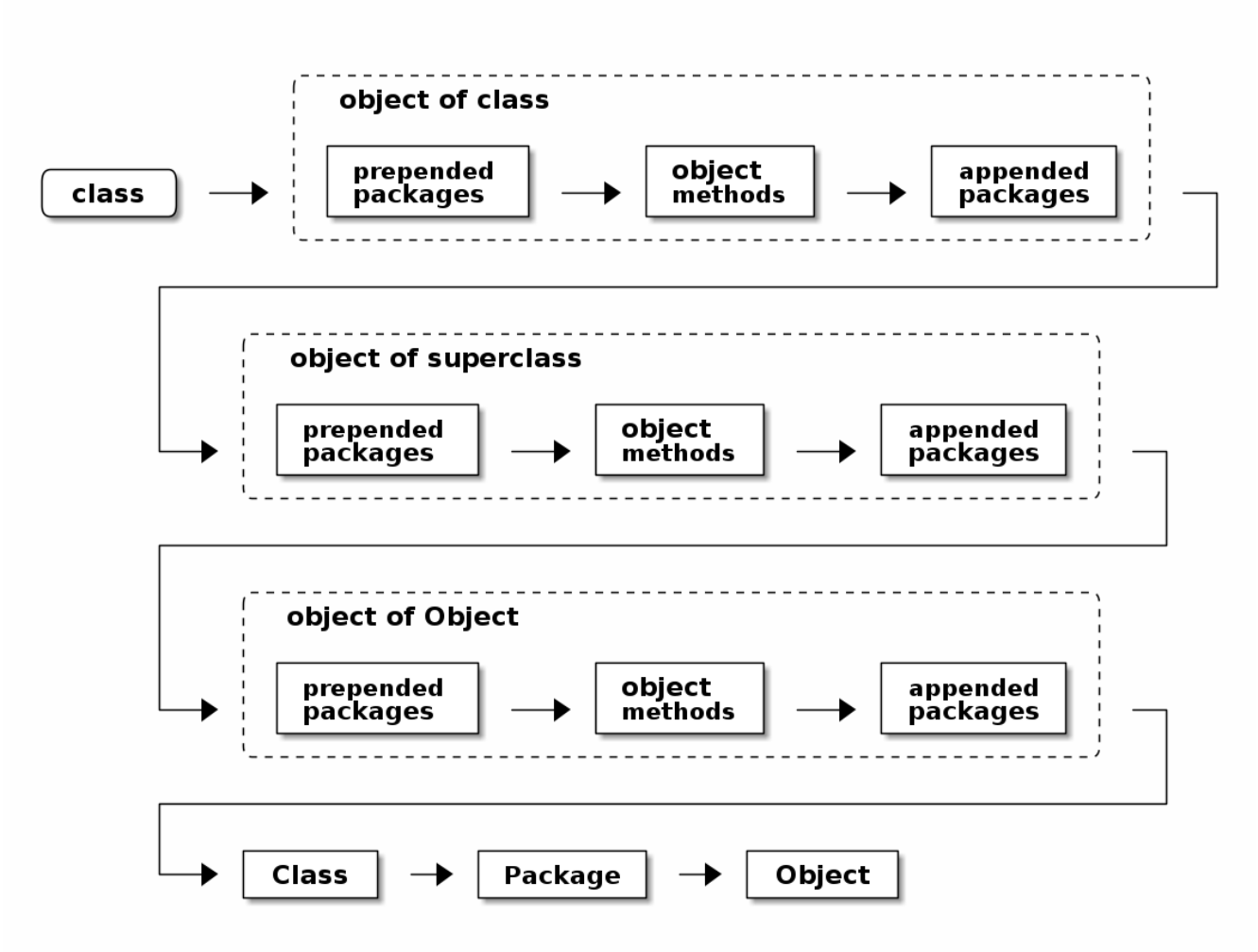
Any method for the object `obj` is first searched in this object itself in the order:

1. In prepended packages, if any.
2. In the list of methods for this particular object.
3. In appended packages, if any.

Next, the method is searched in the class of `obj`, taking into account possible included in this class packages as shown in the diagram. If the method has not been found in this class, then the searching is continued in the superclass of the current class. The procedure can be repeated in the complete chain of superclasses up to the class `Object`. This lookup path is used for all objects, which are not classes.

All classes are also objects, but for them the method lookup path includes an additional

step — searching in the object methods in the chain of superclasses.



The path in the diagram starts as usually with searching in the object itself. Note, the searching is done in packages, which are included in the *object*, not in the class. Next, the searching continued in the superclass object using the same principle. This procedure is continued in the chain of objects of superclasses up to `Object`. Only after that the usual part of the method lookup path starts — searching in the class `Class` and its chain of superclasses. Below is a small example code, demonstrating the difference in the method lookup paths for class and non-class objects.

```
class MyClass
end

object Object
    method mymethod()
        "Hello from object Object!"
    end
end

class Object
    method mymethod()
        "Hello from class Object!"
    end
end
```

```
MyClass.mymethod()       // => "Hello from object Object!"
MyClass.new().mymethod() // => "Hello from class Object!"
```

Syntax

Lexical elements

Identifiers are the names used for variables and constants, classes, functions, methods, etc. keywords: todo. Literals: these permit to define values in our languages. We can have string literals, numeric literal, char literals, boolean literals (but we could consider them keywords as well), array literals, map literals, and more, depending on the language. Separators and delimiters: like colons, semicolons, commas, parenthesis, brackets, braces. Whitespaces: spaces, tabs, newlines.... Comments: todo.

Mir has several types of variables and corresponding different scopes:

- Global variables.
- Local variables.
- Object variables.
- Class variables.

Grammar

> A compiler or interpreter could complain about syntax errors. Your co-workers will complain about semantics.
>
> — doctorlove, stackoverflow.com

Conditional structures

The most usual expression for conditional execution is

```
if condit1 [then]
   code1
elif condit2 [then]
   code2
else
   code3
end
```

Here the `condit1` conditional statement is evaluated. If it is neither `false` nor `nil`, then `code1` is executed. Otherwise the `condit2` is checked. If it is not `false` and not `nil`, then `code2` is executed. In the opposite case `code3` is executed. The keywords `then` in this construction are optional. The

sections `elif` and `else` are optional. There can be multiple `elif` sections.

```
a = Math::Random.new().uniform(min: -10.0, max: 10.0)
if a > 0 then
  print("Positive!")
elif a < 0
  print("Negative!")
else
  print("Zero?!")
end
```

Note, `if ⋯ elif ⋯ else ⋯ end` constructions are in fact expressions, which can returns objects. For example, the following is possible:

```
a = -3
var = if a < 0
-a
end
print(var) // => 3
```

The other type of conditional structure can be constructed using the `if` modifier:

```
code if condition
```

In this statement the `code` is executed only if `condition` is equivalent to `true`, which is any object except for `false` and `nil`. For example

```
func myf(arg)
  return "hi" if arg
end

myf(true) // => "hi"
myf(false) // => nil

raise() if not true // This will do nothing
```

Note, the `return` keyword without arguments cannot be combined with an if modifier. In this case the parser expects a normal `if ⋯ end` structure, whose result will be used as an argument for the return statement. For example

```
func myf(arg)
  return if arg then
    1
  else
    2
```

```
  end
  raise("WAT!?") // unreachable
end

myf(true)  // => 1
myf(false) // => 2
```

In Mir there is also ternary operator `?:` just like in C:

```
a ? b : c
```

which evaluates to `b` if `a` is equivalent to `true` (that is, neither `nil` nor `false`), otherwise to `c`. For example,

```
10**1000 > 33**580 ? "yes" : "no"  // => "yes"
```

Note, for `a`, `b` and `c` can be used functions or any other constructions, which evaluate to objects.

```
result = [].count() == 0 ? print("Yes") : print("No") // Prints Yes to output.
// Here result contains nil returned by the print() function.
```

Functions

Mir functions are also objects just like other instances of classes. This implies that all functions are lambda-functions, i.e. they can be assigned to variables. Functions are defined always in scope of instance variables.

There are three types of functions in respect to how they are related to objects:

- Class instance functions, also termed as methods. A method defined in a class is available for all instances of this class. Methods are defined in `class ... end` constructions.

  ```
  class Klass
    method ifunc(a,b)
      a+b
    end
  end
  Klass.new().ifunc(1,2)  // => 3
  ```

- Object methods. They are defined and available only for particular objects. Definition of object functions is done in `object ... end` constructions.

  ```
  var = 1
  object var
    method func(a)
  ```

```
    self+a
  end
end
var.func(2)  // => 3
```

- Normal functions. They can be defined everywhere using `func` keyword.

```
func myfunc(a,b)
  a+b
end

myfunc(1,2)        // => 3
```

It is possible to create an anonymous function using syntax with braces `{|arguments|: function_body}`

```
// Anonymous function definition without execution
{|x,y|: x+y}
// Define an anonymous function and assign it to a variable ...
var = {|x,y|: x+y}
// ... and call this function
var(1,2)         // => 3
// You can also call an anonymous function without assigning it to a variable
{|x,y|: x+y}(1,2) // => 3
```

In respect to how functions are related to scope of local variables there are two types of them:

- Normal functions. Defined and called as usually. See examples above.
- Closures. Upon creation, a closure captures current local scope and has access to it when called. Closures are created by applying the method `closure()` on functions.

```
// Define function
func times(a)
// An anonymous function is created and converted to a closure, which captures the
local variable 'a' and is returned by the function
{|x|: a*x}.closure()
end

// Create two closures
times2 = times(2)
times3 = times(3)

// Call them
times2(10)          // => 20
times3(10)          // => 30
```

Function arguments can be of two types:

- Normal argument. Defined by an identifier.

```
func myfunc(foo, bar)
  foo/bar
end
myfunc(10, 2)     // => 5
```

- Named argument. Defined using a label, which must be used upon calling the function.

```
func myfunc(foo:, bar:)
  foo/bar
end
myfunc(foo:10, bar:2)     // => 5
```

Both normal and named arguments can be used in the same function definition simultaneously and in any order.

```
func myfunc(foo, bar:)
  foo/bar
end
myfunc(10, bar:2)     // => 5

func myfunc(foo:, bar)
  foo/bar
end
myfunc(foo:10, 2)     // => 5
```

Both types of arguments can have default values and thus be optional.

```
func myfunc(foo=10, bar:2)
  foo/bar
end
myfunc()          // => 5
myfunc(6)         // => 3
myfunc(bar: 5)    // => 2
myfunc(8, bar: 2) // => 4
```

There is an additional special third type of arguments which utilizes the splat operator '*'. It is used for definition of functions, which can be called with variable number of arguments.

```
// Splat operator '*' is applied here onto the variable 'args'.
// Inside the function it is transformed into an array of argument values.
func myfunc(*args)
  return args
end
```

```
myfunc(1, 2.2, "str") // => [1, 2.2, "str"]
// A call without arguments creates an empty array 'args'.
myfunc()              // => []
```

A splated argument can be placed anywhere among all other arguments in function definition. The values of arguments will be accordingly assigned to the function variables upon calling.

```
func myfunc(a, *rest)
  print("a=", a, " rest=", rest)
end
myfunc(1,2,3)  // a=1 rest=[2, 3]

func myfunc(*rest, a)
  print("a=", a, " rest=", rest)
end
myfunc(1,2,3)  // a=3 rest=[1, 2]

func myfunc(a, *rest, b)
  print("a=", a, " rest=", rest, " b=", b)
end
myfunc(1,2,3)  // a=1 rest=[2] b=3
```

Functions can be defined as particular objects or as methods in classes. The former can be assigned to variables, which are stored in other objects. For example, "global functions" are stored in global variables, which reside in `MainObject`. In contrast, methods are just bound to identifiers. In Mir there is a convention on whether a function should be defined as a normal function or as a method in a package or in a class. If the internal logic and the result of the function does not depend on the particular implementation of the class or package, then in this case the class or package is used only as a namespace. Thus, the function is defined as a normal function. For example, in this way trigonometric functions are defined in the `Math` package and accessed using the scope resolution operator `::`.

```
Math::sin(1.0) // => 0.841470984807897
```

In contrast, if the result depends on the class, then this function should be defined as an object method of this class. Then this method can be inherited by a subclass. For example,

```
// FVector is one of the core classes.
FVector.range(0.0, 0.2, 1.0)  // => <F-vector: size=6>

// Define our child class
class MyVector < FVector
end                           // => MyVector

// The object method range() is inherited
MyVector.range(0.0, 0.2, 1.0) // => <F-vector: size=6>
```

Global objects

Functions

`exit()`

Forces termination of Mir script.

`import(libf) → nil`

`import_once(libf) → nil`

Imports and executes code from library file `libf`, which must be defined as a string. Information on each imported library is stored internally. The `import` command can load and execute code from the same library multiple times. The `import_once` command can process only libraries, which have not been imported before. Otherwise it does nothing.

`include(fname) → nil`

Load and execute code from the file `fname` (must be string). In contrast to `import()` this function does not track the content of the included file. It just unconditionally executes the code in the included file.

`print(obj1, obj2, ⋯, rank:int)`

This function applies method `to_s()` to each of input objects and prints resulted strings separated by spaces. In the end the newline symbol is printed. The optional argument `rank` defines rank of MPI process, which should print the strings. By default it is `0`. The value `-1` allows printing for all processes.

`print_table(obj1, obj2, ⋯, format:arr)`

Print objects as table. This function is used mostly for printing several vectors together. To each of input objects the method `to_s(sep:"|")` is applied and resulted strings are printed in form of table. Optional named argument `format` can be used to define custom format strings for input objects. This arguments must be array of strings. If format string `"fstr"` is defined then to the respective object method `to_s(sep:"|", format:"fstr")` is applied. The size of array must not necessarily correspond to the number of objects. The array may also contain `nil` elements. In this case default formats will be used for respective objects.

```
fvec1 = FVector.new(1.1,2.2,3.3)   // => <F-vector: size=3>
fvec2 = FVector.new(0.0,-1.0,-2.0) // => <F-vector: size=3>

print_table(fvec1, fvec2)  // 1.100000000000000e+00 0.000000000000000e+00
                           // 2.200000000000000e+00 -1.000000000000000e+00
                           // 3.300000000000000e+00 -2.000000000000000e+00

print_table(fvec1, fvec2, format:["%.2f", "% .3f"])  // 1.10  0.000
                                                     // 2.20 -1.000
                                                     // 3.30 -2.000

ivec1 = IVector.new(1,2,3) // => <I-vector: size=3>
print_table(fvec1, fvec2, ivec1, format:["% .2f","% .2f","%2ld"])
```

```
                                                    // 1.10  0.00  1
                                                    // 2.20 -1.00  2
                                                    // 3.30 -2.00  3

print_table(fvec1, fvec2, ivec1, format:["% .2f"])
                                          // 1.10 0.00000000000000e+00 1
                                          // 2.20 -1.00000000000000e+00 2
                                          // 3.30 -2.00000000000000e+00 3

print_table(fvec1, fvec2, ivec1, format:[nil,"% .1e","%04ld"])
                                          // 1.100000000000000e+00  0.0e+00 0001
                                          // 2.200000000000000e+00 -1.0e+00 0002
                                          // 3.300000000000000e+00 -2.0e+00 0003
```

`loadmod(modname) → nil`

`loadmod(modname,path) → nil`

Loads Mir module `modname` (must be string) from standard location or from `path` (string).

`unloadmod(modname) → nil`

Unloads Mir module `modname` (must be string).

`raise() → exception`

`raise(excclass) → exception`

`raise(str) → exception`

`raise(excclass,str) → exception`

Raises exception and returns it. Class of exception and custom error string can be explicitly defined. Otherwise default class and/or message will be used.

`global_variables() → array`

Returns array of symbols corresponding to all available global variables.

`add_reader(symb, ⋯) → nil`

`add_writer(symb, ⋯) → nil`

`add_accessors(symb, ⋯) → nil`

When called from a class definition scope these functions add methods for setting and/or getting of instance variables.

```
class MyClass
    method init()
        @foo = 1
        @bar = nil
        @test = nil
    end
    add_reader('foo')
    add_writer('bar')
    add_accessors('test')
end
```

```
a = MyClass.new()
var = a.foo          // => 1
a.bar = 10           // => 10
var = a.test()       // => nil
a.test = "test"      // => "test"
```

`append_pkg(pkg) → nil`

`prepend_pkg(pkg) → nil`

Appending and prepending package `pkg` to objects, classes, packages.

```
package PkgA
    method myfunc()
        "myfunc in PkgA"
    end
end

package PkgB
    method myfunc()
        "myfunc in PkgB"
    end
end

class ClassA
    method myfunc()
        "myfunc in ClassA"
    end
end

ClassA.new().myfunc() // => "myfunc in ClassA"

class ClassA
    append_pkg(PkgA)
end

ClassA.new().myfunc() // => "myfunc in ClassA"

class ClassA
    prepend_pkg(PkgB)
end

ClassA.new().myfunc() // => "myfunc in PkgB"

// Add method to PkgA
package PkgA
    method myfunc2()
        "myfunc2 in PkgA"
    end
end
```

```
ClassA.new().myfunc2() // => "myfunc2 in PkgA"
```

`system(str) → int`

Creates a child process that executes a shell command specified in `str` and returns after the command has been completed. The return integer value indicate status of the last executed command.

Other objects

`$0`

File name (string) of the currently processed script.

`$*`

Command line arguments (array) of the currently processed script.

`$$`

Process id (integer) of Mir.

Special objects

`$MIR_VERSION_MAJOR`

`$MIR_VERSION_MINOR`

Major and minor version numbers (int) of Mir.

`$MIR_VERSION_STRING`

Mir version as a string, which contains more detailed information.

Classes

Core classes are documented here.

Class

Class of all classes, i.e. all classes in Mir are instances of the class `Class`.

```
class MyClass
end

MyClass.class() // => Class
```

Class methods

`new() → object`

Creation of a new instance from class.

```
class Klass
end          // => Klass

Klass.new()  // => <Klass:0x558e91e71a30>
```

Func

Class of functions. Functions are defined using `func ... end` constructions or by placing a code between braces `{ |arguments|: code here }`.

```
func myfunc(a,b)
  a+b
end
myfunc(1,1)  // => 2
```

```
fvar = {|a,b|: a+b}
fvar(1,1)  // => 2
```

Instance methods

`closure() → func`

Conversion to a closure.

```
func times(a)
  {|x|: a*x}.closure()
end

times2 = times(2)
times3 = times(3)

times2(10)          // => 20
times3(10)          // => 30
```

Array

Class of arrays of objects. Indexing of elements starts from 0.

Arrays in Mir can be initialized by using square brackets.

```
var = [1,2,3]             // => [1, 2, 3]
```

Arrays can contain any kinds of objects.

```
var = [1,"a",3.4,Complex.new(1,1)]   // => [1, "a", 3.4, (1+1i)]
```

Instance methods

`inspect() → string`

Creates a string representation of array applying `inspect()` method to each object in array.

```
[1,"str", 2.5].inspect()  // => "[1, \"str\", 2.5]"
```

`to_s() → string`

Converts an array to a string applying `to_s()` method to each object in array.

```
[1,"str", 2.5].to_s()  // => "[1, str, 2.5]"
```

`count() → int`

Returns number of elements in array.

```
[1,"str", 2.5].count()  // => 3
```

`to_fv() → fvector`

Conversion to an object of class `FVector`, i.e. to a vector of floating-point numbers.

```
[1.1,2.2].to_fv()  // => <Vector: dtype=double size=2>
```

`to_iv() → ivector`

Conversion to an object of class `IVector`, i.e. to a vector of integer numbers.

```
[1,2].to_iv()  // => <Vector: dtype=int64 size=2>
```

`to_fm() → fmatrix`

Conversion to fmatrix, i.e. to a matrix of floating-point numbers. The array must contain subarrays or objects convertable to arrays containing floating-point numbers or objects convertable to floating-point numbers. Subarrays must not necessarily be of the same size.

```
[[1.1,2.2],[3.3,4.4]].to_fm().print()  // 1.1e+00  2.2e+00
                                       // 3.3e+00  4.4e+00
                                       // => nil

var = [[1.1,2.2],[3.3,4.4],[5.0],[6.6,7.7,8.9]]
var.to_fm().print()                          // 1.1e+00  2.2e+00  0.0e+00
                                             // 3.3e+00  4.4e+00  0.0e+00
```

```
                                                    // 5.0e+00  0.0e+00  0.0e+00
                                                    // 6.6e+00  7.7e+00  8.9e+00
                                                    // => nil
```

`*arr → obj1, obj2, ⋯, objn`

Applying the splat operator (*) to array. As the result the array is splitted to a series of objects.

```
// Define a function, which has 3 arguments
def func(a,b,c)
a+b+c
end

// Define an array of our arguments
arr = [1,2,3]

// Call the function with the array splitted to three arguments.
func(*arr)   // => 6
```

`arr << obj → arr`

Appends an object to the array.

```
[] << 1             // => [1]
["foo"] << "bar"    // => ["foo", "bar"]
```

`arr[idx1, idx2, ⋯, idxn] → obj1, obj2, ⋯, objn`

Provides access to elements of array. Returns objects corresponding to indices `idx1`, `idx2`, etc. For invalid indexes `nil` objects are returned.

```
var = [1,"a",3.4,Complex.new(1,1)]
var[1]                                  // => "a"
var[3]                                  // => (1+1i)
var[1,2]                                // => "a", 3.4
var[0,1,3]                              // => 1, "a", (1+1i)
var[4]                                  // => nil
var[3,4]                                // => (1+1i), nil
```

`[idx] = obj1, obj2, ⋯, objn → obj1, obj2, ⋯, objn`

Provides access to elements of array. Assigns object(s) to element of an array with index `idx`. If multiple objects assigned then they are converted to an array, which is assigned.

```
var = [1,2,3]     // => [1, 2, 3]
var[0] = "a"      // => "a"
var               // => ["a", 2, 3]
var[2] = "B", "C" // => "B", "C"
```

```
var               // => ["a", 2, ["B", "C"]]
```

`arr1 <=> arr2 → +1 or 0 or -1 or nil`

Comparison of two arrays. Returns 1, 0 or -1 if `arr1` is larger than, equal to or less than `arr2`. The pairwise comparison of objects in arrays is performed using the `<=>` method. First non-equal pair of objects determines the result of the comparison.

```
[1,2] <=> [1,2]       // =>  0
[2,2] <=> [1,2]       // =>  1
[0,2] <=> [1,2]       // => -1
[1,2,3] <=> [1,2]     // =>  1
[1,2] <=> [1,2,3]     // => -1
```

BigFloat

Class of floating point numbers with arbitrary precision.

Class methods

`new() → bigfloat`

`new(float) → bigfloat`

`new(bigfloat) → bigfloat`

`new(int) → bigfloat`

`new(bigint) → bigfloat`

Initialization.

```
BigFloat.new()                    // =>  0
BigFloat.new(1.0)                 // =>  1.0
BigFloat.new(1.0e3000)            // =>  1e+3000
BigFloat.new(-1)                  // => -1.0
BigFloat.new(BigInteger.new(5))   // =>  5.0
```

Instance methods

`to_s() → string`

Conversion to string.

```
BigFloat.new(53).to_s()    // => "53.0"
```

`to_i() → int or bigint`

Conversion to integer number. Warning: floating point (FP) numbers are ususally not exactly represented due to finite precision. Therefore, conversion of big FP numbers (like 1e4000 in the example below) to integers may be not exact.

```
1e4000.to_i()               // => 10000000000000000613363518972856100249503.....
BigFloat.new(1.1e20).to_i() // => 110000000000000000000
BigFloat.new(10.9).to_i()   // => 10
```

`to_f() → float` or `bigfloat`

Conversion to floating point number. First, this method tries to convert bigfloat to float and to return an object of class `Float`. If this is not possible, then the original object is returned.

```
var1 = BigFloat.new(3.0)       // => 3.0
var1.class()                   // => BigFloat
var2 = var1.to_f()             // => 3.0
var2.class()                   // => Float
```

```
var1 = BigFloat.new(-3.0e3000) // => -3e+3000
var1.class()                   // => BigFloat
var2 = var1.to_f()             // => -3e+3000
var2.class()                   // => BigFloat
```

`to_r() → rational`

Conversion to rational number.

```
BigFloat.new(0.25).to_r()    // => (1/4)
```

`to_c() → complex`

Conversion to complex number.

```
BigFloat.new(3.56).to_c()    // => (3.56+0i)
```

`bigfloat <=> number → 1` or `0` or `-1` or `nil`

Comparison of two numbers. Returns: 1 if `bigfloat` > `number`, 0 if `bigfloat` == `number`, -1 if `bigfloat` < `number`, nil in case of other objects. `number` can be float, int, bigfloat, bigint and rational.

```
BigFloat.new(20.0) <=> 10.0    // => 1
BigFloat.new(20.0) <=> 30.0    // => -1
BigFloat.new(20.0) <=> 20.0    // => 0
```

BigInteger

Class of integer numbers with arbitrary precision.

Class methods

`new() → bigint`

`new(int) → bigint`

`new(bigint) → bigint`

Initialization.

```
BigInteger.new()                    // => 0
BigInteger.new(1)                   // => 1
BigInteger.new(BigInteger.new(10)) // => 10
```

Instance methods

`to_s() → string`

Conversion to string.

```
BigInteger.new(10).to_s()    // => "10"
```

`to_i() → int or bigint`

Conversion to integer. First, this method tries to convert bigint to int and to return an object of class `Integer`. If this is not possible, then the original object is returned.

```
var1 = BigInteger.new(10)     // => 10
var1.class()                  // => BigInteger
var2 = var1.to_i()            // => 10
var2.class()                  // => Integer
```

```
var1 = BigInteger.new(9223372036854775808)     // => 9223372036854775808
var1.class()                                   // => BigInteger
var2 = var1.to_i()                             // => 9223372036854775808
var2.class()                                   // => BigInteger
```

`to_r() → rational`

Conversion to rational number.

```
BigInteger.new(10).to_r()    // => (10)
```

`to_c() → complex`

Conversion to complex number.

```
BigInteger.new(10).to_c()    // => (10+0i)
```

to_f() → float or **bigfloat**

Conversion to floating point number.

```
BigInteger.new(10).to_f()       // => 10.0
BigInteger.new(10**1000).to_f() // => 1e+1000
```

bigint + bigint → bigint

bigint + float → float or **bigfloat**

bigint + int → bigint

bigint + bigfloat → bigfloat

bigint + rational → rational

bigint + complex → complex

Addition operation.

```
BigInteger.new(10) + BigInteger.new(5)  // =>  15
BigInteger.new(3) + 1                   // =>  4
BigInteger.new(-3) + 1.1                // => -1.9
BigInteger.new(10**400) + 1.0e308       // =>  1e+400
BigInteger.new(10) + BigFloat.new(5.0)  // =>  15.0
BigInteger.new(10) + Rational.new(5,3)  // =>  (35/3)
BigInteger.new(10) + Complex.new(1,1)   // =>  (11+1i)
```

bigint - bigint → bigint

bigint - float → float or **bigfloat**

bigint - int → bigint

bigint - bigfloat → bigfloat

bigint - rational → rational

bigint - complex → complex

Subtraction operation.

```
BigInteger.new(10) - BigInteger.new(5)  // =>  5
BigInteger.new(3) - 1                   // =>  2
BigInteger.new(-3) - 1.1                // => -4.1
BigInteger.new(-(10**308)) - 1.0e308    // => -2e+308
BigInteger.new(10) - BigFloat.new(5.0)  // =>  5.0
BigInteger.new(10) - Rational.new(5,3)  // =>  (25/3)
BigInteger.new(10) - Complex.new(1,1)   // =>  (9-1i)
```

bigint * bigint → bigint

bigint * float → float or **bigfloat**

bigint * int → bigint

bigint * bigfloat → bigfloat

bigint * rational → rational

bigint * complex → complex

Multiplication operation.

```
BigInteger.new(2)*BigInteger.new(2)  // =>  4
BigInteger.new(2)*0.2                // =>  0.4
BigInteger.new(2)*1.7e308            // =>  3.4e+308
BigInteger.new(-2)*10                // => -20
BigInteger.new(2)*BigFloat.new(35.0) // =>  70.0
BigInteger.new(2)*Rational.new(1,3)  // =>  (2/3)
BigInteger.new(2)*Complex.new(-1,3)  // =>  (-2+6i)
```

bigint / bigint → bigint

bigint / float → float or **bigfloat**

bigint / int → bigint

bigint / bigfloat → bigfloat

bigint / rational → rational

bigint / complex → complex

Division operation.

```
BigInteger.new(20)/BigInteger.new(2) // =>  10
BigInteger.new(21)/BigInteger.new(2) // =>  10
BigInteger.new(2)/0.2                // =>  10.0
BigInteger.new(10)/3e-308            // =>  3.33333333333333e+308
BigInteger.new(10)/5                 // =>  2
BigInteger.new(20)/BigFloat.new(2.0) // =>  10.0
BigInteger.new(2)/Rational.new(3, 1) // =>  (2/3)
BigInteger.new(2)/Complex.new(1, 2)  // =>  (0.4-0.8i)
```

bigint1 <=> number2 → 1 or **0** or **-1** or **nil**

Comparison of two numbers. Returns: 1 if `bigint1` > `number2`, 0 if `bigint1` == `number2`, -1 if `bigint1` < `number2`. `number2` can be int, float, bigint, bigfloat and rational.

```
BigInteger.new(1) <=> 1                 // =>  0
BigInteger.new(-2) <=> 1                // => -1
BigInteger.new(2) <=> 1.0               // =>  1
BigInteger.new(1) <=> Rational.new(1,2) // =>  1
BigInteger.new(1) <=> BigInteger.new(10) // => -1
BigInteger.new(1) <=> BigFloat.new(-2.0) // =>  1
```

`bigint ** int → bigint or rational`

`bigint ** float → float or bigfloat or complex`

`bigint ** bigint → bigint or rational`

`bigint ** bigfloat → float or bigfloat or complex`

`bigint ** rational → bigint or rational or float or bigfloat or complex`

`bigint ** complex → complex`

Exponentiation.

```
BigInteger.new(3)**9                 // =>  19683
BigInteger.new(2)**(-36)             // =>  (1/68719476736)
BigInteger.new(7)**5.3               // =>  30131.4209000904
BigInteger.new(2)**3.0e8             // =>  5.00258364079601e+90308998
BigInteger.new(-2)**500.0            // =>  3.27339060789614e+150
BigInteger.new(-2)**501.0            // => -6.54678121579228e+150
BigInteger.new(-2)**5000.0           // =>  1.41246703213943e+1505
BigInteger.new(-2)**5001.0           // => -2.82493406427885e+1505
BigInteger.new(3)**3.0e-500          // =>  1.0
BigInteger.new(-2)**(0.5)            // =>  (8.65927e-17+1.41421i)
BigInteger.new(4)**Rational.new(1,2) // =>  2.0
BigInteger.new(4)**Rational.new(2,1) // =>  16
BigInteger.new(2)**Complex.new(1,2)  // =>  (0.366914+1.96606i)
```

BitArray

Class of bit arrays.

Instance methods

`to_s() → string`

Conversion to string.

```
BitArray.new(10).to_s() // => "0000000000"
```

`print(line:int) → nil`

Printing of bit array. There is one optional named argument `line`, which controls how many bits must be printed in one line. By default all bits are printed in one line.

```
BitArray.new(10).print() // 0000000000
                         // => nil
BitArray.new(100).print(line:50)
                         // 00000000000000000000000000000000000000000000000000
                         // 00000000000000000000000000000000000000000000000000
                         // => nil
```

`inspect() → string`

Creates a string representing bit array.

```
BitArray.new(100).inspect() // => "<Bit array: size=100>"
```

`size() → int`

Returns total number of bits in bit array.

```
BitArray.new(100).size() // => 100
```

`num_set() → int`

Returns number of set bits (hamming weight) in bit array.

```
BitArray.new(100).num_set() // => 0
```

`num_cleared() → int`

Returns number of not set (cleared) bits in bit array.

```
BitArray.new(100).num_cleared() // => 100
```

`bitarr | bitarr → bitarr`

Bitwise OR of two bit arrays.

```
a = BitArray.new(10)
a[1] = 1
a[3] = 1
a[9] = 1
b = BitArray.new(10)
b[0] = 1
b[2] = 1
b[9] = 1
c = a | b
a.print()    // 0101000001
b.print()    // 1010000001
c.print()    // 1111000001
```

`[idx] = int → int`

Set or clear bit with index `idx`.

```
a = BitArray.new(10)
a.print()             // 0000000000
a[1] = 1              // => 1
a.print()             // 0100000000
```

```
a[1] = 0              // => 0
a.print()             // 0000000000
```

Complex

Class of complex numbers.

Class methods

`new(obj) → complex`

`new(num1,num2) → complex`

Initialization. If one argument is given then Mir calls internally `to_c()` method for this object and returns result. If two arguments are given, they must be integer or floating point numbers.

```
Complex.new(1)              // => (1+0i)
Complex.new( 1.1, 2.2)      // => (1.1+2.2i)
Complex.new( 1.1,-2.2)      // => (1.1-2.2i)
Complex.new(-1.1,-2.2)      // => (-1.1-2.2i)
Complex.new(5, 6.5)         // => (5+6.5i)
```

Instance methods

`to_s() → string`

Conversion to string.

```
Complex.new(1.1, 2.2).to_s()  // => "(1.1+2.2i)"
```

`to_c() → complex`

Conversion to complex number. Returns the original object.

```
var1 = Complex.new(1,2) // => (1+2i)
var1.id()               // => 17086864
var2 = var1.to_c()      // => (1+2i)
var2.id()               // => 17086864
```

`abs() → float`

Absolute value of complex number.

```
var = (-27)**(0.33333333333333333333) // => (1.5+2.59808i)
var.abs()                             // => 3.0
```

complex ** int → complex

complex ** float → complex

complex ** bigint → complex

complex ** bigfloat → complex

complex ** rational → complex

complex ** complex → complex

Exponentiation.

```
Complex.new(2,1)**2                  // => (3+4i)
Complex.new(2,1)**(-2.1)             // => (0.10376-0.152602i)
Complex.new(-1,-1)**(3.0e2)          // => (-1.42725e+45-5.45377e+31i)
Complex.new(-1,-1)**Complex.new(5,3) // => (-1671.53+6430.17i)
Complex.new(-1,-1)**Rational.new(1,2) // => (0.45509-1.09868i)
```

complex + int → complex

complex + float → complex

complex + bigint → complex

complex + bigfloat → complex

complex + rational → complex

complex + complex → complex

Addition. Bigint, bigfloat and rational arguments should be convertable to complex number.

```
Complex.new(2,1) + 2                      // => (4+1i)
Complex.new(-4,1) + 2.99                  // => (-1.01+1i)
Complex.new(-4,1) + BigInteger.new(10)    // => (6+1i)
Complex.new(-4,1) + BigFloat.new(1.7e308) // => (1.7e+308+1i)
Complex.new(0.5,2) + Rational.new(-1,2)   // => (0+2i)
Complex.new(1,1) + Complex.new(1,1)       // => (2+2i)
Complex.new(1.7976e+308,0) + Complex.new(1.7976e+308,0) // => OverflowError
```

complex - int → complex

complex - float → complex

complex - bigint → complex

complex - bigfloat → complex

complex - rational → complex

complex - complex → complex

Subtraction operation. Bigint, bigfloat and rational arguments should be convertable to complex number.

```
Complex.new(2,1) - 2                      // => (0+1i)
Complex.new(-4,1) - 2.99                  // => (-6.99+1i)
Complex.new(4,1.2) - BigInteger.new(2)    // => (2+1.2i)
Complex.new(100,-1) - BigFloat.new(1.7e308) // => (-1.7e+308-1i)
```

```
Complex.new(0.5,2) - Rational.new(-1,2)     // => (1+2i)
Complex.new(1,1) - Complex.new(1,1)         // => (0+0i)
Complex.new(-1.7976e+308,0) - Complex.new(1.7976e+308,0) // => OverflowError
```

complex * int → complex

complex * float → complex

complex * bigint → complex

complex * bigfloat → complex

complex * rational → complex

complex * complex → complex

Multiplication. Bigint, bigfloat and rational arguments should be convertable to complex number.

```
Complex.new(2,1) * 2                        // => (4+2i)
Complex.new(-4,1) * 2.99                    // => (-11.96+2.99i)
Complex.new(4,1.2) * BigInteger.new(-2)     // => (-8-2.4i)
Complex.new(100,-1) * BigFloat.new(1.7)     // => (170-1.7i)
Complex.new(2,2) * Rational.new(-1,2)       // => (-1-1i)
Complex.new(0,-1) * Complex.new(0,-1)       // => (-1-0i)
Complex.new(0,1.0e+200) * Complex.new(0,1.0e+200) // => OverflowError
```

complex / int → complex

complex / float → complex

complex / bigint → complex

complex / bigfloat → complex

complex / rational → complex

complex / complex → complex

Division. Bigint, bigfloat and rational arguments should be convertable to complex number.

```
Complex.new(2,3) / 2                        // => (1+1.5i)
Complex.new(-4,1) / 2.99                    // => (-1.33779+0.334448i)
Complex.new(4,1.2) / BigInteger.new(-2)     // => (-2-0.6i)
Complex.new(170,-17) / BigFloat.new(1.7)    // => (100-10i)
Complex.new(2,2) / Rational.new(-1,2)       // => (-4-4i)
Complex.new(0,-1) / Complex.new(0,-1)       // => (1-0i)
Complex.new(1,1) / 0                        // => InvalidNumArgError
Complex.new(1,-1) / Complex.new(0,0)        // => InvalidNumArgError
Complex.new(1.0e300,1) / Complex.new(1.0e-300,1.0e-300) // => OverflowError
```

complex1 == complex2 → true or **false**

Testing for strict equality of two complex numbers.

```
Complex.new(1.0,2.0) == Complex.new(1.0,2.0)    // => true
```

```
Complex.new(1.001,2.0) == Complex.new(1.0,2.0)  // => false
```

`complex1 != complex2 → true` or `false`

Testing for strict inequality of two complex numbers.

```
Complex.new(1.0,2.0) != Complex.new(1.0,2.0)    // => flase
Complex.new(1.001,2.0) != Complex.new(1.0,2.0)  // => true
```

`eq_eps(complex, epsilon) → true` or `false`

Testing for equality with an other complex number using precision defined by epsilon.

```
Complex.new(1,2).eq_eps(Complex.new(1.00001, 2.00001), 0.0001)     // => true
Complex.new(1,2).eq_eps(Complex.new(1.00001, 2.00001), 0.000001)   // => false
```

Exception

Classes of exceptions. Here is the hierarchy of classes for built-in exceptions:

- Exception
  - InternalError
  - StandardError
    - NameError
      - NoMethodError
    - ArgumentError
    - RuntimeError
    - SysCallError
    - ScopeError
  - NumericError
    - ZeroDivisionError
    - ExpTooLongError
    - InvalidNumArgError
    - OverflowError
    - ArithmeticError

File

Class of files.

Functions

`exist?(file_name) → bool`

Checks if file exists. Returns true if exists, otherwise returns false.

```
File.exist?("readme.txt")       // => true
File.exist?("foobar.txt")       // => false
```

`basename(fpath) → str`

Returns a string corresponding to the basename of the file defined by the input string `fpath`.

```
File.basename("../../hello.txt") // => "hello.txt"
```

Float

Class of floating point numbers. Internally they are represented as 64-bit double numbers. Accordingly the number of significant digits for such numbers is about 15.

Instance methods

`bytes() → string`

Converts a floating-point value to a string of bytes.

```
0.0.bytes()    // => "00 00 00 00 00 00 00 00"
0.5.bytes()    // => "00 00 00 00 00 00 E0 3F"
(-0.3).bytes() // => "33 33 33 33 33 33 D3 BF"
```

`to_s() → string`

Conversion to string.

```
5.5.to_s()    // => "5.5"
```

`to_c() → complex`

Conversion to complex number.

```
2.0.to_c()    // => (2+0i)
```

`to_i() → int or bigint`

Conversion to integer number.

```
2.0.to_i()    // => 2
2.8.to_i()    // => 2
```

```
2e20.to_i()   // => 20000000000000000000
```

`to_r() → rational`

Conversion to rational number.

```
0.25.to_r()    // => (1/4)
```

`to_f() → float`

Conversion to floating point number. In class `Float` this method returns the original object.

```
5.8.to_f()    // => 5.8
```

`-float → float`

Unary negation.

```
var1 = 1.1        // =>  1.1
var2 = -var1      // => -1.1
```

`float + float → float` or `bigfloat`

`float + int → float` or `bigfloat`

`float + bigint → bigfloat`

`float + bigfloat → bigfloat`

`float + rational → bigfloat`

`float + complex → complex`

Addition operation. In case of possible overflow the result is bigfloat.

```
10.0+10.0               // =>  20.0
10.0+1                  // =>  11.0
1.7e308 + 1.7e308       // =>  3.4e+308
1.0 + Rational.new(1,2) // =>  1.5
1.0 + Complex.new(1,2)  // =>  (2+2i)
```

`float - float → float` or `bigfloat`

`float - int → float` or `bigfloat`

`float - bigint → bigfloat`

`float - bigfloat → bigfloat`

`float - rational → bigfloat`

`float - complex → complex`

Subtraction operation. In case of possible overflow the result is bigfloat.

```
10.0-1.0                // =>  9.0
```

```
10.0-1                  // =>  9.0
2.23e-308 - 2.24e-308   // => -0.01e-308
2.23e-308 - 1.23e-308   // =>  1e-308
1.0 - Rational.new(1,2) // =>  0.5
1.0 - Complex.new(2,2)  // =>  (-1-2i)
```

float * float → float or **bigfloat**

float * int → float or **bigfloat**

float * bigint → bigfloat

float * bigfloat → bigfloat

float * rational → bigfloat

float * complex → complex

Multiplication operation. In case of possible overflow the result is either bigint or bigfloat.

```
1.1*1.1                   // =>  1.21
1.1*11                    // =>  12.1
1.0e308*1.0e100           // =>  1e+408
0.1e-10*BigFloat.new(10.0) // =>  1e-10
1.1e-10*BigInteger.new(5)  // =>  5.5e-10
3.0*Complex.new(1, 1)     // =>  (3+3i)
3.0*Rational.new(1, 2)    // =>  1.5
```

float / float → float or **bigfloat**

float / int → float or **bigfloat**

float / bigint → bigfloat

float / bigfloat → bigfloat

float / rational → bigfloat

float / complex → complex

Division operation. In case of possible overflow the result is either bigint or bigfloat.

```
10.0/3.3                  // =>  3.03030303030303
45.0/5                    // =>  9.0
3.0e30/BigInteger.new(50) // =>  6e+28
3.0e30/BigFloat.new(50)   // =>  6e+28
3.0/Complex.new(1, 1)     // =>  (1.5-1.5i)
3.0/Rational.new(1, 2)    // =>  6.0
```

float1 <=> number2 → 1 or **0** or **-1** or **nil**

Strict comparison of two numbers. Returns: 1 if float1 > number2, 0 if float1 == number2, -1 if float1 < number2. number2 can be float, int, bigfloat, bigint and rational.

```
 2.0 <=> 1.0               // =>  1
-2.0 <=> 1                 // => -1
 1.0 <=> 1                 // =>  0
```

```
1.0 <=> Rational.new(1,2)  // =>  1
1.0 <=> BigInteger.new(10) // => -1
1.0 <=> BigFloat.new(-2.0) // =>  1
```

float ** float → float or **bigfloat** or **complex**

float ** int → float or **bigfloat** or **complex**

float ** bigint → float or **bigfloat** or **complex**

float ** bigfloat → float or **bigfloat** or **complex**

float ** rational → float or **bigfloat** or **complex**

float ** complex → complex

Exponentiation. In case of possible overflow the result is bigfloat.

```
3.0**9.0                // =>  19683.0
2.0**(-36)              // =>  1.45519152283668e-11
7.0**5.3                // =>  30131.4209000904
2.0**3.0e8              // =>  5.00258364079601e+90308998
(-2.0)**500.0           // =>  3.27339060789614e+150
(-2.0)**501.0           // => -6.54678121579228e+150
(-2.0)**5000.0          // =>  1.41246703213943e+1505
(-2.0)**5001.0          // => -2.82493406427885e+1505
3.0**3.0e-500           // =>  1.0
(-2.0)**(0.5)           // =>  (8.65927e-17+1.41421i)
4.0**Rational.new(1,2)  // =>  2.0
4.0**Rational.new(2,1)  // =>  16.0
2.0**Complex.new(1,2)   // =>  (0.366914+1.96606i)
```

eq_eps(number, epsilon) → true or **false** or **nil**

Testing for equality with another `number` using precision defined by `epsilon`. Returns `nil` if `number` is not an appropriate object.

```
var = 1.0
var.eq_eps(1.0, 0.00001)       // => true
var.eq_eps(0.9, 0.00001)       // => false
var.eq_eps(1.1, 0.00001)       // => false
var.eq_eps(-10, 0.00001)       // => false
var.eq_eps(10e3000, 0.00001)   // => false
var.eq_eps(-10e3000, 0.00001)  // => false
var.eq_eps(10**1000, 0.00001)  // => false
var.eq_eps(-10**1000, 0.00001) // => false
var.eq_eps(Rational.new(1,2), 0.00001) // => false
var.eq_eps(Rational.new(1,1), 0.00001) // => true
var.eq_eps("1", 0.00001)       // => nil
```

Other objects

`Pi`

Mathematical constant π (3.14159...).

`E`

Euler constant *e* (2.71828...)

`Epsilon`

Floating-point precision. Typical value is about 2.22e-16.

`Max`

Maximal floating point number. Typical value is about 1.8e+308.

FMatrix

Class of matrices for floating-point numbers. Indexing of matrix elements starts from 0.

Class methods

`new(m,n) → fmatrix`

Creates a new fmatrix with `m` rows and `n` columns

```
FMatrix.new(3,2)                // => <Matrix: dtype=double rows=3 columns=2>
```

`identity(dim) → fmatrix`

Creates square identity matrix of size `dim`.

```
var = FMatrix.identity(3)
var.print()  // 1.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
             // 0.000000000000000e+00  1.000000000000000e+00  0.000000000000000e+00
             // 0.000000000000000e+00  0.000000000000000e+00  1.000000000000000e+00
             // => nil
```

`diagonal(dim,val) → fmatrix`

`diagonal(fvec) → fmatrix`

`diagonal(val1,val2,val3,…) → fmatrix`

Create diagonal matrices.

```
var = FMatrix.diagonal(3,5.0)
var.print() // 5.00000000000000e+00  0.00000000000000e+00  0.00000000000000e+00
            // 0.00000000000000e+00  5.00000000000000e+00  0.00000000000000e+00
            // 0.00000000000000e+00  0.00000000000000e+00  5.00000000000000e+00
            // => nil
```

```
var = FVector.new(3.0, 2.0, 1.0)
fmat = FMatrix.diagonal(var)
fmat.print()    // 3.00000000000000e+00  0.00000000000000e+00  0.00000000000000e+00
                // 0.00000000000000e+00  2.00000000000000e+00  0.00000000000000e+00
                // 0.00000000000000e+00  0.00000000000000e+00  1.00000000000000e+00
                // => nil

var = FMatrix.diagonal(1.0, 2.0, 3.0)
var.print()   // 1.00000000000000e+00  0.00000000000000e+00  0.00000000000000e+00
              // 0.00000000000000e+00  2.00000000000000e+00  0.00000000000000e+00
              // 0.00000000000000e+00  0.00000000000000e+00  3.00000000000000e+00
              // => nil
```

`hilbert(m,n) → fmatrix`

Creates Hilbert matrix (m,n) whose element values are 1/(i+j+1), where i and j are row and column indices (starting from 0). A true Hilbert square matrix is created when m equals n.

```
mat = FMatrix.hilbert(10, 10)
mat.print()
```

`lehmer(m,n) → fmatrix`

Creates Lehmer matrix (m,n) whose element values are min(i+1,j+1)/max(i+1,j+1), where i and j are row and column indices (starting from 0). A true Lehmer square matrix is created when m equals n.

```
mat = FMatrix.lehmer(10, 10)
mat.print()
```

Instance methods

`to_s() → string`

Conversion to string.

```
FMatrix.new(2).to_s() // =>
"[[0.000000000000000e+00,0.000000000000000e+00],[0.000000000000000e+00,0.000000000000000e+00]]"
```

`print() → nil`

Formatted printing of matrix.

```
var = FMatrix.new(5,3)
var.print()  // 0.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
             // 0.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
             // 0.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
             // 0.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
```

```
            // 0.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00
            // => nil
```

`export_tiff(filename) → nil`

Converts matrix elements to grayscale values and writes a file in TIFF format with name `filename`.

```
var = FMatrix.new(5,3)
var.export_tiff("var.tif") // => nil
```

`[i,j] → float`

Returns matrix element corresponding to indices `i` and `j`.

```
mat = FMatrix.lehmer(10,10)
mat[1,1] // => 1.0
mat[2,2] // => 1.0
mat[2,3] // => 0.75
mat[1,3] // => 0.5
```

`[i,j] = float → float`

Assigns a floating point number to the element with indices `i,j`. Returns the same floating point number.

```
mat = FMatrix.new(2,2)  // => <FP-matrix: rows=2 columns=2>
mat.print()             // 0.00000000000000e+00  0.00000000000000e+00
                        // 0.00000000000000e+00  0.00000000000000e+00

mat[1,0] = 1.0/3.0      // => 0.333333333333333
mat.print()             // 0.00000000000000e+00  0.00000000000000e+00
                        // 3.33333333333333e-01  0.00000000000000e+00
```

FVector

Class of vectors for floating-point numbers. Indexing of vector elements starts from 0.

Object methods

`new() → fvector`

`new(num_1, num_2, ⋯, num_n) → fvector`

`new(size) → fvector`

`new(size, value) → fvector`

Creates a new fvector. Notes:

- Without argument a vector of size 0 is created.

- num_1, num_2, ..., num_n must be floating-point numbers.
- If one integer argument is given — a vector of the corresponding size and zeroed values is created.
- If two arguments, one integer and one floating point number, are given — a vector of the corresponding size with corresponding values is created.
- If one floating point number is given — a vector of size 1 and containing this value is created.

```
FVector.new(3)             // => <Vector: dtype=double size=3>
       // 0.00000000000000e+00,0.00000000000000e+00,0.00000000000000e+00
FVector.new(3, 5.0)        // => <Vector: dtype=double size=3>
       // 5.00000000000000e+00,5.00000000000000e+00,5.00000000000000e+00
FVector.new(1.1,2.2)       // => <Vector: dtype=double size=2>
       // 1.10000000000000e+00,2.20000000000000e+00
```

`range(min, step, max) → fvector`

`range(min, step, size) → fvector`

Creates a vector with values min, min+step, min+2*step, ... The number of vector elementes is determined by the last argument. This can be directly indicated as integer number size or by the maximal element value max.

```
FVector.range(0.0, 0.2, 1.0).print() // 0.00000000000000e+00
                                     // 2.00000000000000e-01
                                     // 4.00000000000000e-01
                                     // 6.00000000000000e-01
                                     // 8.00000000000000e-01
                                     // 1.00000000000000e+00

FVector.range(-0.3, 0.3, 3).print()  // -3.00000000000000e-01
                                     //  0.00000000000000e+00
                                     //  3.00000000000000e-01
```

Instance methods

`to_s(sep:string, format:string) → string`

Conversion to string. Optional named argument sep defines string for separating numbers. By default it is ",". Optional named argument format defines format string for the numbers. The rules for format are similar to those for printf function in C. In particular, it should match the following regular expression:

```
^([^%]|%%)*%[\+\-\ 0\\#]?[0-9]*([\.][0-9]+)?[aefgAEFG]([^%]|%%)*$
```

The default is the exponential format with maximal number of significant digits (determined automatically) after decimal points. Most usually this is "%.15e".

```
FVector.new(1,2).to_s()
       // => "1.000000000000000e+00,2.000000000000000e+00"
FVector.new(1.1,2.2,3.3).to_s(sep:"|")
       // => "1.100000000000000e+00|2.200000000000000e+00|3.300000000000000e+00"
FVector.new(1.1,2.2,3.3).to_s(sep:"")
       // => "1.100000000000000e+002.200000000000000e+003.300000000000000e+00"
FVector.new(1.1,2.2,-3.3e5).to_s(format:"%.2f")
       // => "1.10,2.20,-330000.00"
FVector.new(1.1,2.2,-3.3e40).to_s(format:"% .g")
       // => " 1, 2,-3e+40"
```

`print() → nil`

Printing of fvector.

```
FVector.new(3).print() // 0.000000000000000e+00
                       // 0.000000000000000e+00
                       // 0.000000000000000e+00
                       // => nil
```

`fill(func) → fvec`

`fill!(func) → fvec`

Fills elements of vector with values returned by function `func`, which should accept integer index as a parameter. The first method creates and returns a new fvector. The `fill!` method does the modification inplace and returns the original object.

```
var1 = FVector.new(3)
       // 0.000000000000000e+00,0.000000000000000e+00,0.000000000000000e+00

func f(idx)
    idx.to_f()
end

var2 = var1.fill(f)
       // 0.000000000000000e+00,1.000000000000000e+00,2.000000000000000e+00

var1.id()              // => 94359644505344
var2.id()              // => 94359644538528
```

`mean() → float`

Calculation of sample mean.

```
[1.1,1.2,1.3,1.4].to_fv().mean() // => 1.25
```

`variance() → float`

Calculation of sample variance.

```
[1.0e9+4.0,1.0e9+7.0,1.0e9+13.0,1.0e9+16].to_fv().variance() // => 30.0
```

`skewness() → float`

Calculation of sample skewness.

```
var = FVector.new(100000)

func f(gen)
  {|idx| gen.gauss()}.closure()
end

var.fill!(f(Math::Random.new()))

print(var.skewness())                // => 0.00796416783735292
```

`kurtosis() → float`

Calculation of sample excess kurtosis.

```
var = FVector.new(100000)

func f(gen)
  {|idx| gen.gauss()}.closure()
end

var.fill!(f(Math::Random.new()))

print(var.kurtosis())                // => 0.00985601954566118
```

`[idx1, idx2, ⋯, idxn] → num1, num2, ⋯, numn`

Provides access to elements of array. Returns floating point numbers corresponding to indices `idx1`, `idx2`, etc.

```
var = FVector.new(0.0, 1.1, 2.2, 3.3, 4.4)
var[1]           // => 1.1
var[3]           // => 3.3
var[1,2]         // => 1.1, 2.2
var[0,1,3]       // => 0.0, 1.1, 3.3
```

`[idx] = num → num`

Provides access to elements of fvector. Assigns floating point number to element with index `idx`.

```
var = FVector.new(9.0, 1.0)
var[0]                          // => 9.0
var[0] = -9.0                   // => -9.0
var[0]                          // => -9.0
```

`to_fm(nrows:int, ncols:int) → fmatrix`

Conversion to matrix of floating-point numbers. Either named argument `nrows` or named argument `ncols` should be defined. This determines sizes of new matrix and copying pattern from fvector to the fmatrix.

```
fvec = FVector.new(1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0)  // => <F-vector: size=8>
fvec.to_fm(nrows:4).print() // 1.000000000000000e+00  2.000000000000000e+00
                           // 3.000000000000000e+00  4.000000000000000e+00
                           // 5.000000000000000e+00  6.000000000000000e+00
                           // 7.000000000000000e+00  8.000000000000000e+00

fvec.to_fm(nrows:2).print() // 1.000000000000000e+00  2.000000000000000e+00
3.000000000000000e+00  4.000000000000000e+00
                           // 5.000000000000000e+00  6.000000000000000e+00
7.000000000000000e+00  8.000000000000000e+00

fvec.to_fm(ncols:4).print() // 1.000000000000000e+00  3.000000000000000e+00
5.000000000000000e+00  7.000000000000000e+00
                           // 2.000000000000000e+00  4.000000000000000e+00
6.000000000000000e+00  8.000000000000000e+00

fvec.to_fm(ncols:2).print() // 1.000000000000000e+00  5.000000000000000e+00
                           // 2.000000000000000e+00  6.000000000000000e+00
                           // 3.000000000000000e+00  7.000000000000000e+00
                           // 4.000000000000000e+00  8.000000000000000e+00
```

`eq_eps(fvec, epsilon) → true` or `false` or `nil`

Testing for equality to another fvector using precision defined by `epsilon`. Comparison is done elementwise. If all pairs of elements are equal, then `true` is returned. If sizes of fvectore are different, then `false` is returned. Returns `nil` if `fvec` is not a fvector.

```
vec1 = FVector.new(1.0, 2.0, 3.0, 4.0, 5.0)
vec2 = FVector.new(1.0, 2.0, 3.0, 4.0, 5.0)
vec1.eq_eps(vec2, Float::Epsilon) // => true
vec2 = FVector.new(1.0, 2.0, 3.0, 4.0, 5.00001)
vec1.eq_eps(vec2, Float::Epsilon) // => false
vec1.eq_eps(vec2, 0.0001)         // => true
vec1.eq_eps("abc", 0.1)           // => nil
vec2 = FVector.new(1.0, 2.0, 3.0, 4.0)
vec1.eq_eps(vec2, Float::Epsilon) // => false
```

Integer

Class of integer numbers. Used for signed integers which fit into 64 bits and are between −9,223,372,036,854,775,807 and 9,223,372,036,854,775,807.

Instance methods

`to_s(base: 10) → string`

Conversion to string. Optional (default value is 10) named argument `base` determines to which base the number should be converted. This must be between 2 and 62.

```
10.to_s()               // => "10"
2.to_s(base: 2)         // => "10"
(-456).to_s(base: 16)   // => "-1C8"
(31**12).to_s(base: 62) // => "wBVF0y3sFV"
```

`to_i() → int`

Conversion to integer. In class `Integer` this method returns the original object.

```
10.to_i()    // => 10
```

`to_r() → rational`

Conversion to rational number.

```
10.to_r()    // => (10)
```

`to_c() → complex`

Conversion to complex number.

```
1.to_c()    // => (1+0i)
```

`to_f() → float`

Conversion to floating point number.

```
2.to_f()    // => 2.0
```

`int + int → int or bigint`

`int + float → float or bigfloat`

`int + bigint → bigint`

`int + bigfloat → bigfloat`

`int + rational → rational`

`int + complex → complex`

Addition operation. In case of possible overflow the result is either bigint or bigfloat.

```
-2 + 1                   // =>  -1
-2 + 1.0                 // =>  -1.0
2 + 9223372036854775807  // =>  9223372036854775809
10 + 1.0e4000            // =>  1.0e+4000
1 + Rational.new(1,2)    // =>  (3/2)
1 + Complex.new(1,2)     // =>  (2+2i)
```

`int - int → int or bigint`

`int - float → float or bigfloat`

`int - bigint → bigint`

`int - bigfloat → bigfloat`

`int - rational → rational`

`int - complex → complex`

Subtraction operation. In case of possible overflow the result is either bigint or bigfloat.

```
2 - 1                    // =>  1
2 - 1.0                  // =>  1.0
-1 - 9223372036854775809 // =>  -9223372036854775810
1000 - 1.0e3000          // =>  -1.00000000000000e+3000
2 - Complex.new(1,2)     // =>  (1-2i)
1 - Rational.new(1,2)    // =>  (1/2)
```

`int * int → int or bigint`

`int * float → float or bigfloat`

`int * bigint → bigint`

`int * bigfloat → bigfloat`

`int * rational → rational`

`int * complex → complex`

Multiplication operation. In case of possible overflow the result is either bigint or bigfloat.

```
2 * 2                    // =>  4
2 * 2.0                  // =>  4.0
1000000000 * 10.0e300    // => 1e+310
10 * 10000000000000000000000000000000000000000000000000
                         // => 100000000000000000000000000000000000000000000000000
```

```
2 * 1.0e1000             // => 2.0e+1000
3 * Complex.new(1, 1)    // => (3+3i)
3 * Rational.new(1, 2)   // => (3/2)
```

`int / int → int`

`int / float → float` or `bigfloat`

`int / bigint → bigint`

`int / bigfloat → bigfloat`

`int / rational → rational`

`int / complex → complex`

Division. The result of integer division is always rounded towards zero. In case of possible overflow the result is either bigint or bigfloat.

```
28 / 5                // => 5
28 / 5.0              // => 5.6
28 / 5.0e500          // => 5.6e-500
2 / Rational.new(3, 2) // => (4/3)
3 / Complex.new(1, 1)  // => (1.5-1.5i)
```

`int % int → int`

`int % float → float`

`int % bigint → bigint`

`int % bigfloat → bigfloat`

Calculates remainder of division. If both values are positive then the result is the same as modulo.

```
11 % 5            // =>  1
11 % (-5)         // =>  1
(-11) % 5         // => -1
(-11) % (-5)      // => -1
11 % (-5.0e99999) // => 1.1e+01
```

`int ** int → int` or `rational`

`int ** float → float`

`int ** bigint → bigint` or `rational`

`int ** bigfloat → bigfloat`

`int ** rational → int` or `float`

`int ** complex → complex`

Exponentiation. In case of possible overflow the result is either bigint or bigfloat.

```
3**9                // =>  19683
2**(-36)            // =>  (1/68719476736)
7**5.3              // =>  30131.4209000904
```

```
2**3.0e8             // =>  5.00258364079601e+90308998
(-2)**500.0          // =>  3.27339060789614e+150
(-2)**501.0          // => -6.54678121579228e+150
(-2)**5000.0         // =>  1.41246703213943e+1505
(-2)**5001.0         // => -2.82493406427885e+1505
3**3.0e-500          // =>  1.0
(-2)**(0.5)          // =>  (8.65927e-17+1.41421i)
4**Rational.new(1,2) // =>  2.0
4**Rational.new(2,1) // =>  16
2**Complex.new(1,2)  // =>  (0.366914+1.96606i)
```

`int1 <=> num2 → 1 or 0 or -1 or nil`

Comparison of two numbers. Returns: 1 if `int1` > `num2`, 0 if `int1` == `num2`, -1 if `int1` < `num2`. `num2` can be int, float, bigint, bigfloat and rational. If `num2` is not a number then `nil` is returned.

```
 1 <=> 1                 // =>  0
-2 <=> 1                 // => -1
 2 <=> 1.0               // =>  1
 1 <=> Rational.new(1,2) // =>  1
 1 <=> BigInteger.new(10) // => -1
 1 <=> BigFloat.new(-2.0) // =>  1
```

`-int → int`

Unary negation. If the result does not fit into integer a bigint is created.

```
var1 = 1        // =>  1
var2 = -var1    // => -1
```

`import(int) → int`

Imports argument value and copies it to an object, to which the method is apllied. The original object is returned. The argument must be integer.

```
var = 15             // => 15
var.id()             // => 94594474701600
var.import(2)        // => 2
var                  // => 2
var.id()             // => 94594474701600
```

`int | int → int`

Bitwise OR of two integers.

```
(a = 18).to_s(2)     // => "10010"
(b = 20).to_s(2)     // => "10100"
(a | b).to_s(2)      // => "10110"
```

```
9|6                                      // => 15
5|3                                      // => 7
("1001".to_i(2) | "0110".to_i(2)).to_s(2) // => "1111"
```

`int & int → int`

Bitwise AND of two integers.

```
(a = 18).to_s(2) // => "10010"
(b = 20).to_s(2) // => "10100"
(a & b).to_s(2)  // => "10000"
```

`int ^ int → int`

Bitwise XOR of two integers.

```
(a = 18).to_s(2)  // => "10010"
(b = 20).to_s(2)  // => "10100"
(a ^ b).to_s(2)   // =>   "110"
```

`~int → int`

Bitwise NOT operation.

```
(a = 18).to_s(2)   // => "10010"
~a                 // => -19
(~a).to_s(2)       // => "-10011"
```

`int >> int → int`

`int << int → int`

Bitwise left and right shifts.

```
a = 18
a.to_s(2)            // =>   "10010"
(a >> 2).to_s(2)     // =>     "100"
(a >> 1).to_s(2)     // =>    "1001"
(a << 1).to_s(2)     // =>  "100100"
(a << 2).to_s(2)     // => "1001000"
```

IVector

Class of vectors of integer numbers in Mir. The limits for the numbers in these vectors are the same as for integers from the class `Integer`. Indexing of vector elements starts from 0.

Class methods

`new() → ivec`

`new(num_1, num_2, ⋯, num_n) → ivec`

`new(length) → ivec`

Creates a new ivector. Notes:

- Without arguments a vector of length 0 is created.
- num_1, num_2, ..., num_n must be integers.
- If one integer argument is given — a vector of the corresponding length and zeroed values is created.

```
IVector.new()              // => <Vector: dtype=int64 size=0>
IVector.new(3)             // => <Vector: dtype=int64 size=3>
IVector.new(1,2)           // => <Vector: dtype=int64 size=2>
```

Instance methods

`to_s(sep:string, format:string) → string`

Conversion to string. Optional named argument sep defines string for separating numbers. By default it is ",". Optional named argument format defines format string for the numbers. The rules for format are similar to those for printf function in C. In particular, it should match the following regular expression:

```
^([^%]|%%)*%[\+\-\ 0]?[0-9]*(l|ll)?[di]([^%]|%%)*$
```

The default format is "%ld".

```
str = IVector.new(10).to_s()                  // => "0,0,0,0,0,0,0,0,0,0"
str = IVector.new(-1,1,-2,3).to_s()           // => "-1,1,-2,3"
str = IVector.new(1,2,3).to_s(sep:" ")        // => "1 2 3"
str = IVector.new(1,-2,3).to_s(format:"% ld") // => " 1,-2, 3"
str = IVector.new(1,-2,3).to_s(format:"% 10ld")
                                        // => "         1,        -2,         3"
str = IVector.new(1,-2,3).to_s(format:"%010ld")
                                        // => "0000000001,-000000002,0000000003"
str = IVector.new(1,-2,3).to_s(format:"%5ld", sep:"|")
                                        // => "    1|   -2|    3"
```

`ivec << inum → ivec`

Appends integer number inum to ivector ivec. Returns modified vector.

```
var = IVector.new()
var << 1                // 1
```

```
var << 2                // 1,2
var << -1 << -2         // 1,2,-1,-2
```

`[idx1, idx2, ⋯, idxn] → num1, num2, ⋯, numn`

Provides access to elements of ivector. Returns numbers corresponding to indices `idx1`, `idx2`, etc.

```
ivec = IVector.new(4,5,6)
ivec[2] // => 6
```

`[idx] = num → num`

Provides access to elements of ivector. Assigns number `num` to element with index `idx`. Returns `num`.

```
ivec = IVector.new(4,5,6)
ivec[0] = -4                // => -4
ivec.to_s()                 // => "-4,5,6"
```

`to_fv() → fvector`

Conversion to fvector.

```
ivec = IVector.new(1,2,3)
ivec.to_fv()                  // => <F-vector: size=3>
fvec.to_s()
       // => "1.00000000000000e+00,2.00000000000000e+00,3.00000000000000e+00"
```

Object

`Object` is the most basic class in Mir.

Instance methods

`to_s() → string`

Returns a string representing object.

```
class MyClass
end                 // => MyClass
MyClass.new().to_s() // => "<MyClass:0x562df09d4cf0>"
```

`id() → int`

Returns an identificator of object. All simultaneously existing objects have unique ids. Note, an id can be zero, positive and negative number.

```
"test".id() // => 14908992
```

`instance_variables() → arr`

Returns an array with symbols corresponding to all variables in the object.

```
class MyClass
  def init(a,b)
    @var1 = a
    @var2 = b
  end
end                  // => MyClass
MyClass.new(1,'symb123').instance_variables() // => ['var1', 'var2']
```

`methods() → arr`

Returns an array with symbols corresponding to all available public methods of the object.

```
1.methods() // => ['to_s', 'to_i', 'to_r', 'to_c', 'to_f', '+', '-', '*',
            //     '/', '%', '**', '<=>', '-@', '<', '>', '<=', '>=', '==',
            //     '!=', 'class', 'to_s', 'id', 'inspect', 'print',
            //     'instance_variables', 'methods', 'object_methods']
```

`object_methods() → arr`

Returns an array with symbols of all available object methods in the object.

```
var = 1
var.object_methods()    // => []
object var
  method add(v)
    self+v
  end
end
var.object_methods()    // => ['add']
```

`class() → obj`

Returns class, to which the object belongs.

```
1.class() // => Integer
```

`del_model() → nil`

Delete model methods and parameters previously set by the `set_model_xxxx()` method(s).

`calc_model() → nil`

Calculates all model parameters and the object itself.

`hash() → str`

Calculates hash string for the object. Note, instances of different classes result in different hashes.

```
1.hash()     // => "618d64387e9229d31c0f5a3e673208f4ccb5af5f"
"1".hash()   // => "636aff38e474a5446ba091ce32f79d6920871b0f"
'1'.hash()   // => "cff1202428115288b1d2902ac5e6f764db1de4f1"
```

`obj1 == obj2 → true` or `false`

`obj1 != obj2 → true` or `false`

Comparison of two objects. When `obj1` and `obj2` are the same object then they are equal, otherwise they are not equal.

```
class MyClass
end                    // => MyClass

var = MyClass.new()    // => <MyClass:0x5576bb06d670>
var == 1               // => false
var == var             // => true
var != 5               // => true

var2 = MyClass.new()   // => <MyClass:0x5576bb071690>
var == var2            // => false
var != var2            // => true
```

Random

Class of random numbers. Available in `Math` package.

Instance methods

`uniform(min: 0.0, max: 1.0) → float`

Returns next random floating point number from uniform distribution in the range defined by two optional named arguments `min` and `max`, which have default values 0.0 and 1.0, respectively. Note, the return value is in the range [`min`, `max`).

```
generator = Math::Random.new()            // => <Random:0x56376a0b0720>
generator.uniform()                       // => 0.270667516629556
generator.uniform()                       // => 0.962210137298266
generator.uniform(max: 10.0)              // => 7.36400249690759
generator.uniform(min: -10.0, max: 10.0)  // => 1.92422736136211
```

`gauss(mu: 0.0, sigma: 1.0) → float`

Returns next random floating point number from Gaussian distribution with mean and standard deviations defined by optional named arguments `mu` and `sigma`. Their default values are

0.0 and 1.0, respectively.

```
generator = Math::Random.new()           // => <Random:0x56376a0b0720>
generator.gauss()                        // => 1.51906972236717
generator.gauss()                        // => 0.618238670944586
generator.gauss(mu: 10.0)                // => 11.1754892403026
generator.gauss(mu: 10.0, sigma: 1000.0) // => -448.534258387352
```

Rational

Class of rational numbers.

Class methods

`new() → rational`

`new(int) → rational`

`new(bigint) → rational`

`new(int,int) → rational`

Initialization.

```
Rational.new()                    // => (0)
Rational.new(5)                   // => (5)
Rational.new(BigInteger.new(5))   // => (5)
Rational.new(1, 5)                // => (1/5)
Rational.new(1, BigInteger.new(5)) // => (1/5)
Rational.new(BigInteger.new(1), 5) // => (1/5)
```

Instance methods

`to_s() → string`

Conversion to string.

```
Rational.new(1, 2).to_s()  // =>  "(1/2)"
```

`to_r() → rational`

Conversion to rational. Returns the original object.

```
var = Rational.new(1,2) // => (1/2)
var.id()                // => 24183744
var2 = var.to_r()       // => (1/2)
var2.id()               // => 24183744
```

`to_i() → int or bigint`

Conversion to an integer number.

```
Rational.new(1, 2).to_i()   // =>  0
Rational.new(2, 1).to_i()   // =>  2
Rational.new(-2, 1).to_i()  // => -2
Rational.new(-2, 1).to_i()  // => -2
Rational.new(6, 3).to_i()   // =>  2
Rational.new(6, 4).to_i()   // =>  1
```

`to_f() → float or bigfloat`

Conversion to a floating-point number.

```
Rational.new(6, 4).to_f()         // => 1.5
Rational.new(1, 10**10).to_f()    // => 1e-10
Rational.new(10**10000, 3).to_f() // => 3.33333333333333e+9999
```

`to_c() → complex`

Conversion to a complex number.

```
Rational.new(6, 4).to_c()         // => (1.5+0i)
Rational.new(-6, 4).to_c()        // => (-1.5+0i)
```

`rational + rational → rational`

Addition operation.

```
var1 = Rational.new(1, 2) // => (1/2)
var2 = Rational.new(3, 4) // => (3/4)
var1 + var2               // => (5/4)
```

`rational <=> number → 1 or 0 or -1 or nil`

Comparison of two numbers. Returns: 1 if `rational` > `number`, 0 if `rational` == `number`, -1 if `rational` < `number`. The `number` can be float, int, bigfloat, bigint and rational, otherwise `nil` is returned.

```
Rational.new(2,3) <=> 1.0/3.0     // => 1
Rational.new(1,3) <=> 2.0/3.0     // => -1
Rational.new(1,3) <=> 1.0/3.0     // => 0
```

String

Class of strings. Strings are initialized by using quotation marks.

```
"this is a string".class()  // => String
```

Instance methods

`to_s() → string`

Returns original string.

```
"AAAaaa".to_s()    // => "AAAaaa"
```

`string1 << string2 → string1`

String concatenation. Modifies original string.

```
var1 = "ABC"
var1.id()           // => 94619497105152
var1 << "DEF"       // => "ABCDEF"
var1.id()           // => 94619497105152
```

`copy() → string`

String copy.

```
var1 = "abc"       // => "abc"
var1.id()          // => 32875440
var2 = var1.copy() // => "abc"
var2.id()          // => 32867232
```

`inspect() → string`

Returns internal representation of the string.

```
"€".inspect()    // => "\"\\xE2\\x82\\xAC\""
```

`string1 + string2 → string3`

String concatenation. Returns a new string.

```
var1 = "ABC"
var2 = "DEF"
var1.id()           // => 32916480
var2.id()           // => 32924688
var3 = var1 + var2  // => "ABCDEF"
var3.id()           // => 32867232
```

`length() → int`

Returns number of characters in the string.

```
"abc".length()      // => 3
"①②③④".length()   // => 4
```

`to_i(base:10) → int`

Conversion to integer. Optional (default value is `10`) named argument `base` determines in which base the number is represented in the string. The base must be between `2` and `62`. The method skips whitespaces in the beginning and converts only characters until first invalid character. It will not raise exception if invalid characters have been met.

```
"333".to_i()             // => 333
"1000".to_i(base:2)      // => 8
"-FFFF".to_i(base:16)    // => -65535
"  1234 56".to_i()       // => 1234
"MIRmir".to_i(base:62)   // => 20427518501
```

`str1 <=> str2 → 1 or 0 or -1 or nil`

Alphabetical comparison of two strings. Returns `nil` if the argument (second object) is not a string.

```
"abc" <=> "abcd"         // => -1
"abce" <=> "abcd"        // => 1
"abce" <=> "abce"        // => 0
"abc" <=> 1              // => nil
```

Symbol

Class of symbols. Symbols are initialized by using single quotes.

```
var = 'symbol'
var.class()  // => Symbol
```

Mir symbols are similar to strings with one major difference: equal symbols represent the same unique Mir object, while equal strings are necessarily different objects.

```
// Initialize two strings with the same content.
str1 = "string"
str2 = "string"
// These two strings are in fact two different objects as can be seen using the id()
method.
str1.id()    // => 167380392
str2.id()    // => 167405256

// Now initialize two equal symbols ...
sym1 = 'symbol'
```

```
sym2 = 'symbol'
// ... and check their id.
sym1.id()    // => 167430536
sym2.id()    // => 167430536
```

Symbols are useful in time-critical parts of code. Once initialized a symbol lives forever in memory and should not be allocated and initialized next time when it is needed. In contrast, strings are deleted by garbage collector when not used anymore.

```
// Create string, assign it to a variable and check id.
var = "string"; var.id()    // => 167463560
// Create equal string and assign it to the same variable.
// After that the old string object with id 167463560 is not used anymore and
permanently deleted.
var = "string"; var.id()    // => 167488160

// In contrast, symbols are not deleted.
var = 'i_am_symbol'; var.id()    // => 167463560
var = 'i_am_symbol'; var.id()    // => 167463560
```

Class methods

`new(string) → symbol`

Creates a new symbol from respective `string`.

```
Symbol.new("symbol")    // => 'symbol'
```

Instance methods

`to_s() → string`

Conversion to string.

```
'symbol'.to_s()   // => "symbol"
```

`inspect() → string`

Creates a string representing the symbol.

```
'symbol'.inspect()   // => "\'symbol\'"
```

`symb1 <=> symb2 → 1 or 0 or -1 or nil`

Alphabetical comparison of two symbols. Returns `nil` if the argument (second object) is not a symbol.

```
'abc' <=> 'abcd'          // => -1
'abce' <=> 'abcd'         // => 1
'abce' <=> 'abce'         // => 0
'abc' <=> 1               // => nil
```

Time

Class of dates and time.

Class methods

`now() → time`

Returns instance of `Time` representing current time.

```
Time.now()   // => Mon, 22 Aug 2016 14:16:59 GMT
```

Instance methods

`inspect() → string`

Creates a string representing time.

```
Time.now().inspect() // => "Mon, 22 Aug 2016 14:16:59 GMT"
```

`to_i() → int`

Conversion to integer number, corresponding to the number of microseconds since 00:00:00 January 1, 1970 UTC.

```
Time.now().to_i()    // => 1471932988950839
```

`time1 <=> time2 → 1 or 0 or -1 or nil`

Comparison of two instances of `Time`. Returns 1 if `time1` > `time2`, 0 if `time1` equals `time2`, -1 if `time1` < `time2` or `nil` if second object for comparison in not a time.

```
t1 = Time.now() // => Tue, 23 Aug 2016 06:29:34 GMT
t2 = Time.now() // => Tue, 23 Aug 2016 06:29:39 GMT
t1 <=> t2       // => -1
```

`time1 + int → time2`

Addition operation. Second object is integer, representing number of microseconds. Returns a new instance of `Time`.

```
tim = Time.now()        // => Tue, 30 Aug 2016 20:31:37 GMT
```

```
tim + (2*60*60*1000000) // => Tue, 30 Aug 2016 22:31:37 GMT
```

`time1 - int → time2`

`time1 - time2 → int`

Subtraction of `int` number of microseconds from `time1` and returning a new instance of `Time`. Another option is the calculation of difference in microseconds between `time1` and `time2`.

```
t = Time.now()        // => Fri, 02 Sep 2016 14:30:48 GMT
t - 60*60*1000000     // => Fri, 02 Sep 2016 13:30:48 GMT

t1 = Time.now()       // => Fri, 02 Sep 2016 14:30:48 GMT
t2 = Time.now()       // => Fri, 02 Sep 2016 14:32:08 GMT
t2 - t1               // => 80241686
```

Packages

Core packages are documented here.

Comparable

Package for comparison of objects, whose classes define the method `<=>`. This method should return 1, 0, or -1 if `obj1` is greater, equal or less than `obj2`, respectively.

Methods

`num1 > num2 → true or false`

Greater than.

```
2 > 1    // => true
1 > 2    // => false
1 > 1    // => false
```

`num1 < num2 → true or false`

Less than.

```
1 < 10.0     // => true
20.0 < -3    // => false
1.5 < 1.5    // => false
```

`num1 >= num2 → true or false`

Greater than or equal to.

```
5 >= 3       // => true
```

```
5 >= 5       // => true
-3e-398 >= 1 // => false
```

`num1 <= num2 → true or false`

Less than or equal to.

```
3 <= 5                // => true
3 <= 3                // => true
4 <= BigInteger.new(5) // => true
BigInteger.new(5) <= Rational.new(1,2) // => false
```

`num1 == num2 → true or false`

Equal to.

```
3 == 3                // => true
5 == 3                // => false
4 == BigInteger.new(4) // => true
BigInteger.new(-5) == Rational.new(-5, 1) // => true
```

`num1 != num2 → true or false`

Not equal to.

```
3 != 3                 // => false
5 != 3                 // => true
4 != BigInteger.new(-4) // => true
BigInteger.new(5) != BigFloat.new(5.0) // => false
```

Math

Package for mathematical calculations. Contains trigonometric functions, etc.

Functions

`sin(num) → float or complex`

Calculation of sine. Accepts all types of numeric arguments.

```
Math::sin(1)                // =>  0.841470984807897
Math::sin(1.1)              // =>  0.891207360061435
Math::sin(Rational.new(1,2)) // =>  0.479425538604203
Math::sin(Complex.new(2,3))  // =>  (9.1545-4.16891i)
Math::sin(10**100)          // => -0.380637731005029
Math::sin(1e3000)           // => -0.97877565916552
```

`asin(num) → float or complex`

Calculation of arc sine. Accepts all types of numeric arguments.

```
Math::asin(1)                    // =>  1.5707963267949
Math::asin(0.5)                  // =>  0.523598775598299
Math::asin(Rational.new(-1,3))   // => -0.339836909454122
Math::asin(Complex.new(0.1,2.3)) // =>  (0.0398565+1.57101i)
Math::asin(BigFloat.new(-0.1))   // => -0.10016742116156
Math::asin(BigInteger.new(1))    // =>  1.5707963267949
```

`cos(num) → float or complex`

Calculation of cosine. Accepts all types of numeric arguments.

```
Math::cos(1)                 // => 0.54030230586814
Math::cos(1.1)               // => 0.453596121425577
Math::cos(Rational.new(1,2)) // => 0.877582561890373
Math::cos(Complex.new(2,3))  // => (-4.18963-9.10923i)
Math::cos(10**100)           // => 0.924724238751934
Math::cos(1e3000)            // => 0.204934645741275
```

`acos(num) → float or complex`

Calculation of arc cosine. Accepts all types of numeric arguments.

```
Math::acos(1)                    // => 0.0
Math::acos(0.5)                  // => 1.0471975511966
Math::acos(Rational.new(-1,3))   // => 1.91063323624902
Math::acos(Complex.new(0.1,2.3)) // => (1.53094-1.57101i)
Math::acos(BigFloat.new(-0.1))   // => 1.67096374795646
Math::acos(BigInteger.new(1))    // => 0.0
```

`tan(num) → float`

Calculation of tangent.

```
Math::tan(0)          // => 0.0
Math::tan(1.57)       // => 1255.76559150079
Math::tan(-1.57)      // => -1255.76559150079
```

`atan(num) → float`

Calculation of arctangent.

```
Math::atan(0)           // => 0.0
Math::atan(1.0e100)     // => 1.570796326794897
Math::atan(-1.0e100)    // => -1.570796326794897
```

`exp(num) → float or bigfloat or complex`

Base-e exponentiation. Accepts all types of numeric arguments.

```
Math::exp(1)                    // => 2.71828182845905
Math::exp(3.3)                  // => 27.1126389206579
Math::exp(-3.3)                 // => 0.03688316740124
Math::exp(Rational.new(1,2))    // => 1.64872127070013
Math::exp(9**9)                 // => 3.53964309320634e+168254580
Math::exp(BigInteger.new(50))   // => 5.18470552858707e+21
Math::exp(BigFloat.new(-50.0)) // => 1.92874984796392e-22
Math::exp(Complex.new(2,3))     // => (-7.31511+1.04274i)
```

`abs(num) → num`

Returns absolute value of number.

```
Math::abs(-1)                   // => 1
Math::abs(1)                    // => 1
Math::abs(-1.0)                 // => 1.0
Math::abs(1.0)                  // => 1.0
```

`sqrt(num) → float`

Square root.

```
Math::sqrt(4)                   // => 2.0
Math::sqrt(16.0)                // => 4.0
```

`log(num, base: e) → float`

Calculation of logarithm. By default a natural logarithm is calculated. A non-default base can be defined using the optional named argument `base`.

```
Math::log(1.0)              // => 0.0
Math::log(1)                // => 0.0
Math::log(Float::E)         // => 1.0
Math::log(10.0, base: 10.0) // => 1.0
```

MPI

Package with MPI functions, methods and other related objects.

`barrier()`

Blocks execution of the current process until all other processes in the communicator of the session also call this function. This operation performs a synchronization among the processes belonging to the session communicator.

```
// Each MPI process prints in order
i = 0
while i < MPI::Size
    if i == MPI::Rank
        print("I am process of rank " + i.to_s() + "\n", rank: i)
    end
    MPI::barrier()
    i += 1
end                 // I am process of rank 0
                    // I am process of rank 1
                    // I am process of rank 2
                    // I am process of rank 3
```

Other objects

`Size`

`Rank`

MPI size of 'world' and MPI rank of this process in 'world', both integer numbers.

```
// If starting in serial mode:
MPI::Size    // => 1
MPI::Rank    // => 0
```

```
// If starting in parallel mode on 4 nodes:
MPI::Size    // => 4
myrank = "My rank is " + MPI::Rank.to_s() + "\n"
print(rank:-1, myrank) // My rank is 1
                       // My rank is 0
                       // My rank is 3
                       // My rank is 2
```

Platform

This package contains various functions and constants related to the computing platform.

`scimark(size, args) → float`

Runs SciMark numerical benchmark [1], consisting of FFT, SOR, Monte Carlo, sparse matrix multiplication and LU decomposition, and returns the average score of the benchmark in approximate Mflops as a floating point number. The larger is the score the faster was the calculation. There must be defined a normal argument, one of symbols `'tiny'`, `'small'` or `'large'`, indicating the size of problems to be solved. Other named arguments are optional and can be used for fine-tuning of the problems (default values are shown):

- min_time: 2.0
- fft_size: 1048576

- sor_size: 100
- sparse_size_m: 100000
- sparse_size_nz: 1000000
- lu_size: 1000
- run_mc: true (false or nil switches off the Monte-Carlo test)
- verbose: false (use true to force printing additional info)
- singlify: false (use true to bind the process to one processing unit (core))

For the explanation of the parameters see [1]. Combinations of the normal argument with named arguments are possible. For example, setting the `'small'` size and switching off FFT and Monte-Carlo tests:

```
scimark('small', fft_size: 0, run_mc: false, verbose: true)
```

`mbandwidth(test_number, args) → float,float,symb`

Runs benchmark for memory bandwidth as implemented in the mirml_membench_bandwidth function of MIRML. Returns memory bandwidth and its standard deviation in MB/s (not MiB/s!) and a symbol with description of the test. Test number must be given as a normal argument, which must be in the range from 1 to `MBANDWIDTH_MAXTEST`. Optional named arguments can be defined (default values are shown):

- maxiter: 10 (maximal number of iterations)
- singlify: false (use true to bind the process to one processing unit (core))

```
Platform::mbandwidth(Platform::MBANDWIDTH_MAXTEST, maxiter: 20)
```

`mlatency(buf_size, args) → float,float,float,float`

Runs benchmark for memory latency as implemented in the mirml_membench_latency function of MIRML. Returns four floating point numbers, latency from the single read test, its standard deviation, latency from the double read test and the respective standard deviation. All values are in ns. The size of memory testing buffer must be given as a normal argument in bytes. Optional named arguments can also be defined (default values are shown):

- maxiter: 10 (maximal number of iterations)
- singlify: false (use true to bind the process to one processing unit (core))

```
Platform::mlatency(1 << 10, maxiter: 20)
```

`hwinfo() → nil`

Prints hardware information.

Other objects

`OS`

This constant contains a symbol with the type of operating system. In can be one of `'OS/2'`, `'Windows'`, `'Linux'`, `'Darwin'`, `'BSD'`, `'UNIX'`.

`OS_VERSION`

A symbol with detailed information on the version of the operating system.

`MBANDWIDTH_MAXTEST`

Maximal test number for the `mbandwidth` function.

References

[1] “SciMark 4.0.” https://math.nist.gov/scimark2/index.html , Accessed: 23.05.2022. [Online].