



**Language Reference**

# Table of Contents

Introduction	1.1
Getting started	1.2
An HTTP Server	1.2.1
A Command Line Application	1.2.2
Using the compiler	1.3
The shards command	1.4
Syntax and semantics	1.5
Comments	1.5.1
Literals	1.5.2
Nil	1.5.2.1
Bool	1.5.2.2
Integers	1.5.2.3
Floats	1.5.2.4
Char	1.5.2.5
String	1.5.2.6
Symbol	1.5.2.7
Array	1.5.2.8
Hash	1.5.2.9
Range	1.5.2.10
Regex	1.5.2.11
Tuple	1.5.2.12
NamedTuple	1.5.2.13
Proc	1.5.2.14
Command	1.5.2.15
Assignment	1.5.3
Local variables	1.5.4
Control expressions	1.5.5
Truthy and falsey values	1.5.5.1
if	1.5.5.2
As a suffix	1.5.5.2.1
As an expression	1.5.5.2.2
Ternary if	1.5.5.2.3
if var	1.5.5.2.4
if var.is_a?(...)	1.5.5.2.5
if var.responds_to?(...)	1.5.5.2.6

if var.nil?	1.5.5.2.7
if !	1.5.5.2.8
unless	1.5.5.3
case	1.5.5.4
while	1.5.5.5
break	1.5.5.5.1
next	1.5.5.5.2
until	1.5.5.6
&&	1.5.5.7
	1.5.5.8
Requiring files	1.5.6
Types and methods	1.5.7
Everything is an object	1.5.7.1
The Program	1.5.7.2
Classes and methods	1.5.7.3
new, initialize and allocate	1.5.7.3.1
Methods and instance variables	1.5.7.3.2
Type inference	1.5.7.3.3
Union types	1.5.7.3.4
Overloading	1.5.7.3.5
Default values and named arguments	1.5.7.3.6
Splats and tuples	1.5.7.3.7
Type restrictions	1.5.7.3.8
Return types	1.5.7.3.9
Method arguments	1.5.7.3.10
Operators	1.5.7.3.11
Visibility	1.5.7.3.12
Inheritance	1.5.7.3.13
Virtual and abstract types	1.5.7.3.13.1
Class methods	1.5.7.3.14
Class variables	1.5.7.3.15
finalize	1.5.7.3.16
Modules	1.5.7.4
Generics	1.5.7.5
Structs	1.5.7.6
Constants	1.5.7.7
Enums	1.5.7.8
Blocks and Procs	1.5.7.9

Capturing blocks	1.5.7.9.1
Proc literal	1.5.7.9.2
Block forwarding	1.5.7.9.3
Closures	1.5.7.9.4
alias	1.5.7.10
Exception handling	1.5.8
Type grammar	1.5.9
Type reflection	1.5.10
is_a?	1.5.10.1
nil?	1.5.10.2
responds_to?	1.5.10.3
as	1.5.10.4
as?	1.5.10.5
typeof	1.5.10.6
Macros	1.5.11
Macro methods	1.5.11.1
Hooks	1.5.11.2
Fresh variables	1.5.11.3
Annotations	1.5.12
Built-in annotations	1.5.12.1
Low-level primitives	1.5.13
pointerof	1.5.13.1
sizeof	1.5.13.2
instance_sizeof	1.5.13.3
offsetof	1.5.13.4
Uninitialized variable declaration	1.5.13.5
Compile-time flags	1.5.14
Cross-compilation	1.5.14.1
C bindings	1.5.15
lib	1.5.15.1
fun	1.5.15.2
out	1.5.15.2.1
to_unsafe	1.5.15.2.2
struct	1.5.15.3
union	1.5.15.4
enum	1.5.15.5
Variables	1.5.15.6
Constants	1.5.15.7

type	1.5.15.8
alias	1.5.15.9
Callbacks	1.5.15.10
Unsafe code	1.5.16
Conventions	1.6
Coding style	1.6.1
Documenting code	1.6.2
Database	1.7
Connection	1.7.1
Connection pool	1.7.2
Transactions	1.7.3
Guides	1.8
Performance	1.8.1
Concurrency	1.8.2
Testing	1.8.3
Writing Shards	1.8.4
Hosting on GitHub	1.8.4.1
Hosting on GitLab	1.8.4.2
Continuous Integration	1.8.5
Using Travis CI	1.8.5.1
Using CircleCI	1.8.5.2

# Crystal Programming Language

Welcome to the language reference for the Crystal programming language!

Crystal is a programming language with the following goals:

- Have a syntax similar to Ruby (but compatibility with it is not a goal).
- Be statically type-checked, but without having to specify the type of variables or method arguments.
- Be able to call C code by writing bindings to it in Crystal.
- Have compile-time evaluation and generation of code, to avoid boilerplate code.
- Compile to efficient native code.

From here, you can jump to anywhere you want in this document. Although, if you are new to the Crystal Language, we suggest that you continue reading the [Getting started](#) section.

**Crystal's standard library is documented in the [API docs](#).**

## Getting started

Hi and welcome to Crystal's Reference Book!

First let's make sure to [install the compiler](#) correctly so that we may try all the examples listed in this book.

Once installed, the Crystal compiler should be available as `crystal` command.

Let's try it!

## Crystal version

We may check the Crystal compiler version. If Crystal is installed correctly then we should see something like this:

```
$ crystal --version
Crystal 0.34.0 (2020-04-07)

LLVM: 10.0.0
Default target: x86_64-apple-macosx
```

Great!

## Crystal help

Now, if we want to list all the options given by the compiler, we may run `crystal` program without any arguments:

```
$ crystal
Usage: crystal [command] [switches] [program file] [--] [arguments]

Command:
  init                generate a new project
  build               build an executable
  docs               generate documentation
  env                print Crystal environment information
  eval               eval code from args or standard input
  play               starts Crystal playground server
  run (default)      build and run program
  spec               build and run specs (in spec directory)
  tool               run a tool
  help, --help, -h  show this help
  version, --version, -v show version

Run a command followed by --help to see command specific information, ex:
crystal <command> --help
```

More details about using the compiler can be found on the manpage `man crystal` or in our [compiler manual](#).

# Hello Crystal

The following example is the classic Hello World. In Crystal it looks like this:

```
# hello_world.cr  
  
puts "Hello World!"
```

We may run our example like this:

```
$ crystal hello_world.cr  
Hello World!
```

**Note:** The main routine is simply the program itself. There's no need to define a "main" function or something similar.

Here we have two more examples to continue our first steps in Crystal:

- [HTTP Server](#)
- [Command Line Application](#)



## HTTP Server

A slightly more interesting example is an HTTP Server:

```
require "http/server"

server = HTTP::Server.new do |context|
  context.response.content_type = "text/plain"
  context.response.print "Hello world! The time is #{Time.local}"
end

address = server.bind_tcp 8080
puts "Listening on http://#{address}"
server.listen
```

The above code will make sense once you read the whole language reference, but we can already learn some things.

- You can [require](#) code defined in other files:

```
require "http/server"
```

- You can define [local variables](#) without the need to specify their type:

```
server = HTTP::Server.new ...
```

- The port of the HTTP server is set by using the method `bind_tcp` on the object `HTTP::Server` (the port set to 8080).

```
address = server.bind_tcp 8080
```

- You program by invoking [methods](#) (or sending messages) to objects.

```
HTTP::Server.new ...
...
Time.local
...
address = server.bind_tcp 8080
...
puts "Listening on http://#{address}"
...
server.listen
```

- You can use code blocks, or simply [blocks](#), which are a very convenient way to reuse code and get some features from the functional world:

```
HTTP::Server.new do |context|
  ...
end
```

- You can easily create strings with embedded content, known as string interpolation. The language comes with other [syntax](#) as well to create arrays,

Nil

hashes, ranges, tuples and more:

```
"Hello world! The time is #{Time.local}"
```

# Command Line Interface Application

Programming Command Line Interface applications (CLI applications) is one of the most entertaining tasks a developer may do. So let's have some fun building our first CLI application in Crystal.

There are two main topics when building a CLI application:

- [input](#)
- [output](#)

## Input

This topic covers all things related to:

- [options passed to the app](#)
- [request for user input](#)

## Options

It is a very common practice to pass options to the application. For example, we may run `crystal -v` and Crystal will display:

```
$ crystal -v
Crystal 0.31.1 (2019-10-02)

LLVM: 8.0.1
Default target: x86_64-apple-macosx
```

and if we run: `crystal -h`, then Crystal will show all the accepted options and how to use them.

So now the question would be: **do we need to implement an options parser?**

No need to, Crystal got us covered with the class `OptionParser`. Let's build an application using this parser!

At start our CLI application will have two options:

- `-v / --version` : it will display the application version.
- `-h / --help` : it will display the application help.

```
# file: help.cr
require "option_parser"

OptionParser.parse do |parser|
  parser.banner = "Welcome to The Beatles App!"

  parser.on "-v", "--version", "Show version" do
    puts "version 1.0"
    exit
  end
  parser.on "-h", "--help", "Show help" do
    puts parser
    exit
  end
end
```

So, how does all this work? Well ... magic! No, it's not really magic! Just Crystal making our life easy. When our application starts, the block passed to `OptionParser#parse` gets executed. In that block we define all the options. After the block is executed, the parser will start consuming the arguments passed to the application, trying to match each one with the options defined by us. If an option matches then the block passed to `parser#on` gets executed!

We can read all about `OptionParser` in [the official API documentation](#). And from there we are one click away from the source code ... the actual proof that it is not magic!

Now, let's run our application. We have two ways [using the compiler](#):

1. [Build the application](#) and then run it.
2. Compile and [run the application](#), all in one command.

We are going to use the second way:

```
$ crystal ./help.cr -- -h

Welcome to The Beatles App!
  -v, --version          Show version
  -h, --help            Show help
```

Let's build another *fabulous* application with the following feature:

By default (i.e. no options given) the application will display the names of the Fab Four. But, if we pass the option `-t / --twist` it will display the names in uppercase:

```

# file: twist_and_shout.cr
require "option_parser"

the_beatles = [
  "John Lennon",
  "Paul McCartney",
  "George Harrison",
  "Ringo Starr"
]
shout = false

option_parser = OptionParser.parse do |parser|
  parser.banner = "Welcome to The Beatles App!"

  parser.on "-v", "--version", "Show version" do
    puts "version 1.0"
    exit
  end
  parser.on "-h", "--help", "Show help" do
    puts parser
    exit
  end
  parser.on "-t", "--twist", "Twist and SHOUT" do
    shout = true
  end
end

members = the_beatles
members = the_beatles.map &.upcase if shout

puts ""
puts "Group members:"
puts "======"
members.each do |member|
  puts member
end

```

Running the application with the `-t` option will output:

```

$ crystal run ./twist_and_shout.cr -- -t

Group members:
=====
JOHN LENNON
PAUL MCCARTNEY
GEORGE HARRISON
RINGO STARR

```

## Parameterized options

Let's create another application: *when passing the option `-g / --goodbye_heLLo`, the application will say hello to a given name passed as a parameter to the option.*

```
# file: hello_goodbye.cr
require "option_parser"

the_beatles = [
  "John Lennon",
  "Paul McCartney",
  "George Harrison",
  "Ringo Starr"
]
say_hi_to = ""

option_parser = OptionParser.parse do |parser|
  parser.banner = "Welcome to The Beatles App!"

  parser.on "-v", "--version", "Show version" do
    puts "version 1.0"
    exit
  end
  parser.on "-h", "--help", "Show help" do
    puts parser
    exit
  end
  parser.on "-g NAME", "--goodbye_hello=NAME", "Say hello to whoever you want" do |name|
    say_hi_to = name
  end
end

unless say_hi_to.empty?
  puts ""
  puts "You say goodbye, and #{the_beatles.sample} says hello to #{say_hi_to}!"
end
```

In this case, the block receives a parameter that represents the parameter passed to the option.

Let's try it!

```
$ crystal ./hello_goodbye.cr -- -g "Penny Lane"

You say goodbye, and Ringo Starr say hello to Penny Lane!
```

Great! These applications look awesome! But, **what happens when we pass an option that is not declared?** For example -n

```
$ crystal ./hello_goodbye.cr -- -n
Unhandled exception: Invalid option: -n (OptionParser::InvalidOption)
from ...
```

Oh no! It's broken: we need to handle **invalid options** and **invalid parameters** given to an option! For these two situations, the `OptionParser` class has two methods: `#invalid_option` and `#missing_option`

So, let's add this option handlers and merge all this CLI applications into one fabulous CLI application!

## All My CLI: The complete application!

Nil

Here's the final result, with invalid/missing options handling, plus other new options:

```

# file: all_my_cli.cr
require "option_parser"

the_beatles = [
  "John Lennon",
  "Paul McCartney",
  "George Harrison",
  "Ringo Starr"
]
shout = false
say_hi_to = ""
strawberry = false

option_parser = OptionParser.parse do |parser|
  parser.banner = "Welcome to The Beatles App!"

  parser.on "-v", "--version", "Show version" do
    puts "version 1.0"
    exit
  end
  parser.on "-h", "--help", "Show help" do
    puts parser
    exit
  end
  parser.on "-t", "--twist", "Twist and SHOUT" do
    shout = true
  end
  parser.on "-g NAME", "--goodbye_hello=NAME", "Say hello to whoever you want" do |name|
    say_hi_to = name
  end
  parser.on "-r", "--random_goodbye_hello", "Say hello to one random member" do
    say_hi_to = the_beatles.sample
  end
  parser.on "-s", "--strawberry", "Strawberry fields forever mode ON" do
    strawberry = true
  end
  parser.missing_option do |option_flag|
    STDERR.puts "ERROR: #{option_flag} is missing something."
    STDERR.puts ""
    STDERR.puts parser
    exit(1)
  end
  parser.invalid_option do |option_flag|
    STDERR.puts "ERROR: #{option_flag} is not a valid option."
    STDERR.puts parser
    exit(1)
  end
end

members = the_beatles
members = the_beatles.map &.upcase if shout

puts "Strawberry fields forever mode ON" if strawberry

puts ""
puts "Group members:"
puts "======"
members.each do |member|
  puts "#{strawberry ? "🍓" : "-"} #{member}"
end

unless say_hi_to.empty?
  puts ""
  puts "You say goodbye, and I say hello to #{say_hi_to}!"
end

```



## Request for user input

Sometimes, we may need the user to input a value. How do we *read* that value? Easy, peasy! Let's create a new application: the Fab Four will sing with us any phrase we want. When running the application, it will request a phrase to the user and the magic will happen!

```
# file: let_it_cli.cr
puts "Welcome to The Beatles Sing Along version 1.0!"
puts "Enter a phrase you want The Beatles to sing"
print "> "
user_input = gets
puts "The Beatles are singing: 🎵#{user_input}🎵🎸🎤🎸"
```

The method `gets` will **pause** the execution of the application, until the user finishes entering the input (pressing the `Enter` key). When the user presses `Enter`, then the execution will continue and `user_input` will have the user value.

But what happen if the user doesn't enter any value? In that case, we would get an empty string (if the user only presses `Enter`) or maybe a `Nil` value (if the input stream id closed, e.g. by pressing `Ctrl+D`). To illustrate the problem let's try the following: we want the input entered by the user to be sang loudly:

```
# file: let_it_cli.cr
puts "Welcome to The Beatles Sing Along version 1.0!"
puts "Enter a phrase you want The Beatles to sing"
print "> "
user_input = gets
puts "The Beatles are singing: 🎵#{user_input.upcase}🎵🎸🎤🎸"
```

When running the example, Crystal will reply:

```
$ crystal ./let_it_cli.cr
Showing last frame. Use --error-trace for full trace.

In let_it_cli.cr:5:46

 5 | puts "The Beatles are singing: 🎵#{user_input.upper_case}
    |                                     ^-----
Error: undefined method 'upper_case' for Nil (compile-time type is (String | Nil))
```

Ah! We should have known better: the type of the user input is the [union type](#) `String | Nil`. So, we have to test for `Nil` and for `empty` and act naturally for each case:

```
# file: let_it_cli.cr
puts "Welcome to The Beatles Sing Along version 1.0!"
puts "Enter a phrase you want The Beatles to sing"
print "> "
user_input = gets

exit if user_input.nil? # Ctrl+D

default_lyrics = "Na, na, na, na-na-na na" \
  " / " \
  "Na-na-na na, hey Jude"

lyrics = user_input.presence || default_lyrics

puts "The Beatles are singing: 🎵#{lyrics.upcase}🎵🎸🎤"
```

## Output

Now, we will focus on the second main topic: our application's output. For starters, our applications already display information but (I think) we could do better. Let's add more *life* (i.e. colors!) to the outputs.

And to accomplish this, we will be using the `Colorize` module.

Let's build a really simple application that shows a string with colors! We will use yellow font on a black background:

```
# file: yellow_cli.cr
require "colorize"

puts "#{The Beatles".colorize(:yellow).on(:black)} App"
```

Great! That was easy! Now imagine using this string as the banner for our All My CLI application, it's easy if you try:

```
parser.banner = "#{The Beatles".colorize(:yellow).on(:black)} App"
```

For our second application, we will add a *text decoration* ( `blink` in this case):

```
# file: let_it_cli.cr
require "colorize"

puts "Welcome to The Beatles Sing Along version 1.0!"
puts "Enter a phrase you want The Beatles to sing"
print "> "
user_input = gets

exit if user_input.nil? # Ctrl+D

default_lyrics = "Na, na, na, na-na-na na" \
  " / " \
  "Na-na-na na, hey Jude"

lyrics = user_input.presence || default_lyrics

puts "The Beatles are singing: #{🎵#{user_input}🎵🎸🎤".colorize.mode(:blink)}"
```

Let's try the renewed application ... and *hear* the difference!! **Now** we have two fabulous apps!!

You may find a list of **available colors** and **text decorations** in the [API documentation](#).

## Testing

As with any other application, at some point we would like to [write tests](#) for the different features.

Right now the code containing the logic of each of the applications always gets executed with the `OptionParser`, i.e. there is no way to include that file without running the whole application. So first we would need to refactor the code, separating the code necessary for parsing options from the logic. Once the refactor is done, we could start testing the logic and including the file with the logic in the testing files we need. We leave this as an exercise for the reader.

## Using `Readline` and `NCurses`

In case we want to build richer CLI applications, there are libraries that can help us. Here we will name two well-known libraries: `Readline` and `NCurses`.

As stated in the documentation for the [GNU Readline Library](#), `Readline` is a library that provides a set of functions for use by applications that allow users to edit command lines as they are typed in. `Readline` has some great features: filename autocompletion out of the box; custom autocompletion method; keybinding, just to mention a few. If we want to try it then the [crystal-lang/crystal-readline](#) shard will give us an easy API to use `Readline`.

On the other hand, we have `NCurses` (New Curses). This library allows developers to create *graphical* user interfaces in the terminal. As its name implies, it is an improved version of the library named `Curses`, which was developed to support a text-based dungeon-crawling adventure game called Rogue! As you can imagine, there are already [a couple of shards](#) in the ecosystem that will allow us to use `NCurses` in Crystal!

And so we have reached The End 🤘🎵

## Using the compiler

### Compiling and running at once

To compile and run a program in a single shot, invoke `crystal run` with a single filename:

```
$ echo 'puts "Hello World!'" > hello_world.cr
$ crystal run hello_world.cr
Hello World!
```

The `run` command compiles the source file `hello_world.cr` to a binary executable in a temporary location and immediately executes it.

### Creating an executable

The `crystal build` command builds a binary executable. The output file has the same name as the source file minus the extension `.cr`.

```
$ crystal build hello_world.cr
$ ./hello_world
Hello World!
```

### Release builds

By default, the generated executables are not fully optimized. The `--release` flag can be used to enable optimizations.

```
$ crystal build hello_world.cr --release
```

Compiling without release mode is much faster and the resulting binaries still offer pretty good performance.

Building in release mode should be used for production-ready executables and when performing benchmarks. For simple development builds, there is usually no reason to do so.

To reduce the binary size for distributable files, the `--no-debug` flag can be used. This removes debug symbols reducing file size, but obviously making debugging more difficult.

### Creating a statically-linked executable

The `--static` flag can be used to build a statically-linked executable:

```
$ crystal build hello_world.cr --release --static
```

**NOTE:** Building fully static linked executables is currently only supported on Alpine Linux.

More information about statically linking [can be found on the wiki](#).

The compiler uses the `CRYSTAL_LIBRARY_PATH` environment variable as a first lookup destination for static and dynamic libraries that are to be linked. This can be used to provide static versions of libraries that are also available as dynamic libraries.

## Creating a Crystal project

The `crystal init` command helps to initialize a Crystal project folder, setting up a basic project structure. `crystal init app <name>` is used for an application, `crystal init lib <name>` for a library.

```
$ crystal init app myapp
  create  myapp/.gitignore
  create  myapp/.editorconfig
  create  myapp/LICENSE
  create  myapp/README.md
  create  myapp/.travis.yml
  create  myapp/shard.yml
  create  myapp/src/myapp.cr
  create  myapp/src/myapp/version.cr
  create  myapp/spec/spec_helper.cr
  create  myapp/spec/myapp_spec.cr
  Initialized empty Git repository in /home/crystal/myapp/.git/
```

Not all of these files are required for every project, and some might need more customization, but `crystal init` creates a good default environment for developing Crystal applications and libraries.

## Compiler commands

- `crystal init` : generate a new project
- `crystal build` : build an executable
- `crystal docs` : generate documentation
- `crystal env` : print Crystal environment information
- `crystal eval` : eval code from args or standard input
- `crystal play` : starts crystal playground server
- `crystal run` : build and run program
- `crystal spec` : build and run specs
- `crystal tool` : run a compiler tool
- `crystal help` : show help about commands and options
- `crystal version` : show version

To see the available options for a particular command, use `--help` after a command:

```
crystal run
```

The `run` command compiles a source file to a binary executable and immediately runs it.

```
crystal [run] [<options>] <programfile> [-- <argument>...]
```

Arguments to the compiled binary can be separated with double dash `--` from the compiler arguments. The binary executable is stored in a temporary location between compiling and running.

Example:

```
$ echo 'puts "Hello #{ARGV[0]?}!"' > hello_world.cr
$ crystal run hello_world.cr -- Crystal
Hello Crystal!
```

#### Common options:

- `--release` : Compile in release mode, doing extra work to apply optimizations to the generated code.
- `--progress` : Show progress during compilation.
- `--static` : Link statically.

More options are described in the integrated help: `crystal run --help` or man page `man crystal`.

## crystal build

The `crystal build` command builds a dynamically-linked binary executable.

```
crystal build [<options>] <programfile>
```

Unless specified, the resulting binary will have the same name as the source file minus the extension `.cr`.

Example:

```
$ echo 'puts "Hello #{ARGV[0]?}!"' > hello_world.cr
$ crystal build hello_world.cr
$ ./hello_world Crystal
Hello Crystal!
```

#### Common options:

- `--cross-compile` : Generate a `.o` file, and print the command to generate an executable to stdout.
- `-D FLAG, --define FLAG` : Define a compile-time flag.
- `-o <output_file>` : Define the name of the binary executable.
- `--release` : Compile in release mode, doing extra work to apply optimizations to the generated code.
- `--link-flags FLAGS` : Additional flags to pass to the linker.
- `--lto=thin` : Use ThinLTO, improving performance on release builds.
- `--no-debug` : Skip any symbolic debug info, reducing the output file size.

- `--progress` : Show progress during compilation.
- `--static` : Link statically.
- `--verbose` : Display executed commands.

More options are described in the integrated help: `crystal build --help` or man page `man crystal` .

## crystal eval

The `crystal eval` command reads Crystal source code from command line or stdin, compiles it to a binary executable and immediately runs it.

```
crystal eval [<options>] [<source>]
```

If no `source` argument is provided, the Crystal source is read from standard input. The binary executable is stored in a temporary location between compiling and running.

Example:

```
$ crystal eval 'puts "Hello World"'
Hello World!
$ echo 'puts "Hello World"' | crystal eval
Hello World!
```

NOTE: When running interactively, stdin can usually be closed by typing the end of transmission character ( `Ctrl+D` ).

### Common options:

- `-o <output_file>` : Define the name of the binary executable.
- `--release` : Compile in release mode, doing extra work to apply optimizations to the generated code.
- `--lto=thin` : Use ThinLTO, improves performance.
- `--no-debug` : Skip any symbolic debug info, reducing the output file size.
- `--progress` : Show progress during compilation.
- `--static` : Link statically.

More options are described in the integrated help: `crystal eval --help` or man page `man crystal` .

## crystal version

The `crystal version` command prints the Crystal version, LLVM version and default target triple.

```
crystal version
```

Example:

```
$ crystal version
Crystal 0.25.1 [b782738ff] (2018-06-27)

LLVM: 4.0.0
Default target: x86_64-unknown-linux-gnu
```

## crystal init

The `crystal init` command initializes a Crystal project folder.

```
crystal init (lib|app) <name> [<dir>]
```

The first argument is either `lib` or `app`. A `lib` is a reusable library whereas `app` describes an application not intended to be used as a dependency. A library doesn't have a `shard.lock` file in its repository and no build target in `shard.yml`, but instructions for using it as a dependency.

Example:

```
$ crystal init lib my_cool_lib
create my_cool_lib/.gitignore
create my_cool_lib/.editorconfig
create my_cool_lib/LICENSE
create my_cool_lib/README.md
create my_cool_lib/.travis.yml
create my_cool_lib/shard.yml
create my_cool_lib/src/my_cool_lib.cr
create my_cool_lib/spec/spec_helper.cr
create my_cool_lib/spec/my_cool_lib_spec.cr
Initialized empty Git repository in ~/my_cool_lib/.git/
```

## crystal docs

The `crystal docs` command generates API documentation from inline docstrings in Crystal files (see [documenting code](#)).

```
crystal docs [--output=<output_dir>] [--canonical-base-url=<url>] [<source_file>...]
```

The command creates a static website in `output_dir` (default `./docs`), consisting of HTML files for each Crystal type, in a folder structure mirroring the Crystal namespaces. The endpoint `docs/index.html` can be opened by any web browser. The entire API docs are also stored as a JSON document in

```
$output_dir/index.json .
```

By default, all Crystal files in `./src` will be appended (i.e. `src/**/*.cr`). In order to account for load-order dependencies, `source_file` can be used to specify one (or multiple) endpoints for the docs generator.

```
crystal docs src/my_app.cr
```

**Common options:**



- `--project-name=NAME` : Set the project name. The default value is extracted from `shard.yml` if available. In case no default can be found, this option is mandatory.
- `--project-version=VERSION` : Set the project version. The default value is extracted from current git commit or `shard.yml` if available. In case no default can be found, this option is mandatory.
- `--output=DIR, -o DIR` : Set the output directory (default: `./docs` )
- `--canonical-base-url=URL, -b URL` : Set the [canonical base url](#)

For the above example to output the docs at `public` with custom canonical base url, and entrypoint `src/my_app.cr` , the following arguments can be used:

```
crystal docs --output public --canonical-base-url http://example.com/ src/my_app.cr
```

## crystal env

The `crystal env` command prints environment variables used by Crystal.

```
crystal env [<var>...]
```

By default, it prints information as a shell script. If one or more `var` arguments are provided, the value of each named variable is printed on its own line.

Example:

```
$ crystal env
CRYSTAL_CACHE_DIR="/home/crystal/.cache/crystal"
CRYSTAL_PATH="/usr/bin/./share/crystal/src:lib"
CRYSTAL_VERSION="0.28.0"
CRYSTAL_LIBRARY_PATH="/usr/bin/./lib/crystal/lib"
$ crystal env CRYSTAL_VERSION
0.28.0
```

## crystal spec

The `crystal spec` command compiles and runs a Crystal spec suite.

```
crystal spec [<options>] [<file>...] [-- [<runner_options>]]
```

All `files` arguments are concatenated into a single Crystal source. If an argument points to a folder, all spec files inside that folder are appended. If no `files` argument is provided, the default is `./spec` . A filename can be suffixed by `:` and a line number, providing this location to the `--location` runner option (see below).

Run `crystal spec --options` for available options.

### Runner options:

`runner_options` are provided to the compiled binary executable which runs the specs. They should be separated from the other arguments by a double dash (`--`).

- `--verbose` : Prints verbose output, including all example names.
- `--profile` : Prints the 10 slowest specs.
- `--fail-fast` : Abort the spec run on first failure.
- `--junit_output <output_dir>` : Generates JUnit XML output.

The following options can be combined to filter the list of specs to run.

- `--example <name>` : Runs examples whose full nested names include `name` .
- `--line <line>` : Runs examples whose line matches `line` .
- `--location <file>:<line>` : Runs example(s) at `line` in `file` (multiple options allowed).
- `--tag <tag>` : Runs examples with the specified tag, or excludes examples by adding `~` before the tag (multiple options allowed).
  - `--tag a --tag b` will include specs tagged with `a` OR `b` .
  - `--tag ~a --tag ~b` will include specs not tagged with `a` AND not tagged with `b` .
  - `--tag a --tag ~b` will include specs tagged with `a` , but not tagged with `b` .

Example:

```
$ crystal spec
F

Failures:

  1) Myapp works
     Failure/Error: false.should eq(true)

       Expected: true
       got: false

     # spec/myapp_spec.cr:7

Finished in 880 microseconds
1 examples, 1 failures, 0 errors, 0 pending

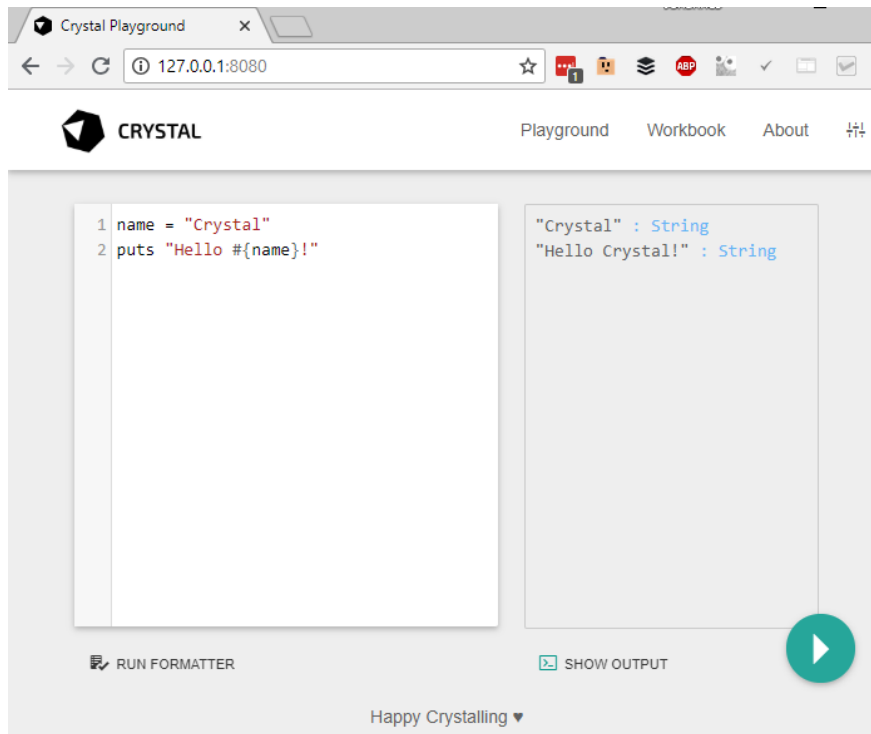
Failed examples:

crystal spec spec/myapp_spec.cr:6 # Myapp works
```

## crystal play

The `crystal play` command starts a webserver serving an interactive Crystal playground.

```
crystal play [--port <port>] [--binding <host>] [--verbose] [file]
```



## crystal tool

- `crystal tool context` : Show context for given location
- `crystal tool expand` : Show macro expansion for given location
- `crystal tool format` : Format Crystal files
- `crystal tool hierarchy` : Show type hierarchy
- `crystal tool implementations` : Show implementations for given call in location
- `crystal tool types` : Show types of main variables

## crystal tool format

The `crystal tool format` command applies default format to Crystal source files.

```
crystal tool format [--check] [<path>...]
```

`path` can be a file or folder name and include all Crystal files in that folder tree. Omitting `path` is equal to specifying the current working directory.

## Environment variables

The following environment variables are used by the Crystal compiler if set in the environment. Otherwise the compiler will populate them with default values. Their values can be inspected using `crystal env`.

- `CRYSTAL_CACHE_DIR` : Defines path where Crystal caches partial compilation results for faster subsequent builds. This path is also used to temporarily store executables when Crystal programs are run with `crystal run` rather than `crystal build`. Default value is the first directory that either exists or can

be created of `${XDG_CACHE_HOME}/crystal` (if `XDG_CACHE_HOME` is defined),  
`~/.cache/crystal`, `~/.crystal`, `./.crystal`. If `CRYSTAL_CACHE_DIR` is  
set but points to a path that is not writeable, the default values are used  
instead.

- `CRYSTAL_PATH` : Defines paths where Crystal searches for required files.
- `CRYSTAL_VERSION` is only available as output of `crystal env`. The compiler  
neither sets nor reads it.
- `CRYSTAL_LIBRARY_PATH` : The compiler uses the paths in this variable as a first  
lookup destination for static and dynamic libraries that are to be linked. For  
example, if static libraries are put in `build/libs`, setting the environment  
variable accordingly will tell the compiler to look for libraries there.

## The shards command

Crystal is typically accompanied by Shards, its dependency manager.

It manages dependencies for Crystal projects and libraries with reproducible installs across computers and systems.

## Installation

Shards is usually distributed with Crystal itself. Alternatively, a separate `shards` package may be available for your system.

To install from source, download or clone [the repository](#) and run `make CRFLAGS=--release`. The compiled binary is in `bin/shards` and should be added to `PATH`.

## Usage

`shards` requires the presence of a `shard.yml` file in the project folder (working directory). This file describes the project and lists dependencies that are required to build it. A default file can be created by running `shards init`. The file's contents are explained in the [Writing a Shard guide](#) and a detailed description of the file format is provided by the [shard.yml specification](#).

Running `shards install` resolves and installs the specified dependencies. The installed versions are written into a `shard.lock` file for using the exact same dependency versions when running `shards install` again.

If your shard builds an application, both `shard.yml` and `shard.lock` should be checked into version control to provide reproducible dependency installs. If it is only a library for other shards to depend on, `shard.lock` should *not* be checked in, only `shard.yml`. It's good advice to add it to `.gitignore` (the `crystal init` does this automatically when initializing a `lib` repository).

## Shards commands

```
shards [<options>...] [<command>]
```

If no command is given, `install` will be run by default.

- `shards build` : Builds an executable
- `shards check` : Verifies dependencies are installed
- `shards init` : Generates a new `shard.yml`
- `shards install` : Resolves and installs dependencies
- `shards list` : Lists installed dependencies
- `shards prune` : Removes unused dependencies
- `shards update` : Resolves and updates dependencies

- `shards version` : Shows version of a shard

To see the available options for a particular command, use `--help` after a command.

#### Common options:

- `--version` : Prints the version of `shards` .
- `-h, --help` : Prints usage synopsis.
- `--no-color` : Disabled colored output.
- `--production` : Runs in release mode. Development dependencies won't be installed and only locked dependencies will be installed. Commands will fail if dependencies in `shard.yml` and `shard.lock` are out of sync (used by `install` , `update` , `check` and `list` command)
- `-q, --quiet` : Decreases the log verbosity, printing only warnings and errors.
- `-v, --verbose` : Increases the log verbosity, printing all debug statements.

## shards build

```
shards build [<targets>] [<options>...]
```

Builds the specified targets in `bin` path. If no targets are specified, all are built. This command ensures all dependencies are installed, so it is not necessary to run `shards install` before.

All options following the command are delegated to `crystal build` .

## shards check

```
shards check
```

Verifies that all dependencies are installed and requirements are satisfied.

Exit status:

- `0` : Dependencies are satisfied.
- `1` : Dependencies are not satisfied.

## shards init

```
shards init
```

Initializes a shard folder and creates a `shard.yml` .

## shards install

```
shards install
```

Resolves and installs dependencies into the `lib` folder. If not already present, generates a `shard.lock` file from resolved dependencies, locking version numbers or Git commits.

Reads and enforces locked versions and commits if a `shard.lock` file is present. The install command may fail if a locked version doesn't match a requirement, but may succeed if a new dependency was added, as long as it doesn't generate a conflict, thus generating a new `shard.lock` file.

## shards list

```
shards list
```

Lists the installed dependencies and their versions.

## shards prune

```
shards prune
```

Removes unused dependencies from lib folder.

## shards update

```
shards update
```

Resolves and updates all dependencies into the lib folder again, whatever the locked versions and commits in the `shard.lock` file. Eventually generates a new `shard.lock` file.

## shards version

```
shards version [<path>]
```

Prints the version of the shard.

# Syntax and Semantics

The program's source code must be encoded in UTF-8.

- [Comments](#)
- [Literals](#)
- [Assignment](#)
- [Local Variables](#)
- [Control Expression](#)
- [Requiring Files](#)
- [Types and methods](#)
- [Exception Handling](#)
- [Type Grammar](#)
- [Type Reflection](#)
- [Macros](#)
- [Annotations](#)
- [Low Level Primitives](#)
- [Compile-time Flags](#)
- [C bindings](#)
- [Unsafe code](#)



## Comments

Comments start with the # character. Only one-line comments are supported for now.

```
# This is a comment
```

# Literals

Crystal provides several literals for creating values of some basic types.

Literal	Sample values
Nil	<code>nil</code>
Bool	<code>true</code> , <code>false</code>
Integers	<code>18</code> , <code>-12</code> , <code>19_i64</code> , <code>14_u32</code> , <code>64_u8</code>
Floats	<code>1.0</code> , <code>1.0_f32</code> , <code>1e10</code> , <code>-0.5</code>
Char	<code>'a'</code> , <code>'\n'</code> , <code>'あ'</code>
String	<code>"foo\tbar"</code> , <code>%("あ")</code> , <code>%q(foo #{foo})</code>
Symbol	<code>:symbol</code> , <code>:"foo bar"</code>
Array	<code>[1, 2, 3]</code> , <code>[1, 2, 3] of Int32</code> , <code>%w(one two three)</code>
Array-like	<code>Set{1, 2, 3}</code>
Hash	<code>{"foo" =&gt; 2}</code> , <code>{}</code> of <code>String =&gt; Int32</code>
Hash-like	<code>MyType{"foo" =&gt; "bar"}</code>
Range	<code>1..9</code> , <code>1...10</code> , <code>0..var</code>
Regex	<code>/(foo)?bar/</code> , <code>/foo #{foo}/imx</code> , <code>%r(foo/)</code>
Tuple	<code>{1, "hello", 'x'}</code>
NamedTuple	<code>{name: "Crystal", year: 2011}</code> , <code>{"this is a key": 1}</code>
Proc	<code>-&gt;(x : Int32, y : Int32) { x + y }</code>
Command	<code>`echo foo`</code> , <code>%x(echo foo)</code>

## Nil

The `Nil` type is used to represent the absence of a value, similar to `null` in other languages. It only has a single value:

```
nil
```

## Bool

`Bool` has only two possible values: `true` and `false`. They are constructed using the following literals:

```
true # A Bool that is true
false # A Bool that is false
```

# Integers

There are four signed integer types, and four unsigned integer types:

Type	Length	Minimum Value	Maximum Value
<code>Int8</code>	8	-128	127
<code>Int16</code>	16	-32,768	32,767
<code>Int32</code>	32	-2,147,483,648	2,147,483,647
<code>Int64</code>	64	$-2^{63}$	$2^{63} - 1$
<code>UInt8</code>	8	0	255
<code>UInt16</code>	16	0	65,535
<code>UInt32</code>	32	0	4,294,967,295
<code>UInt64</code>	64	0	$2^{64} - 1$

An integer literal is an optional `+` or `-` sign, followed by a sequence of digits and underscores, optionally followed by a suffix. If no suffix is present, the literal's type is the lowest between `Int32`, `Int64` and `UInt64` in which the number fits:

```

1      # Int32

1_i8   # Int8
1_i16  # Int16
1_i32  # Int32
1_i64  # Int64

1_u8   # UInt8
1_u16  # UInt16
1_u32  # UInt32
1_u64  # UInt64

+10    # Int32
-20    # Int32

2147483648      # Int64
9223372036854775808 # UInt64

```

The underscore `_` before the suffix is optional.

Underscores can be used to make some numbers more readable:

```
1_000_000 # better than 1000000
```

Binary numbers start with `0b`:

```
0b1101 # == 13
```

Octal numbers start with a `0o`:

Nil

```
0o123 # == 83
```

Hexadecimal numbers start with `0x` :

```
0xFE012D # == 16646445  
0xfe012d # == 16646445
```

# Floats

There are two floating point types, `Float32` and `Float64`, which correspond to the `binary32` and `binary64` types defined by IEEE.

A floating point literal is an optional `+` or `-` sign, followed by a sequence of numbers or underscores, followed by a dot, followed by numbers or underscores, followed by an optional exponent suffix, followed by an optional type suffix. If no suffix is present, the literal's type is `Float64`.

```
1.0      # Float64
1.0_f32  # Float32
1_f32    # Float32

1e10     # Float64
1.5e10   # Float64
1.5e-7   # Float64

+1.3     # Float64
-0.5     # Float64
```

The underscore `_` before the suffix is optional.

Underscores can be used to make some numbers more readable:

```
1_000_000.111_111 # a lot more readable than 1000000.111111, yet functionally the same
```

# Char

A `Char` represents a 32-bit [Unicode code point](#).

It is typically created with a char literal by enclosing an UTF-8 character in single quotes.

```
'a'
'z'
'0'
'_'
'あ'
```

A backslash denotes a special character, which can either be a named escape sequence or a numerical representation of a unicode codepoint.

Available escape sequences:

```
'\''      # single quote
'\\'      # backslash
'\a'      # alert
'\b'      # backspace
'\e'      # escape
'\f'      # form feed
'\n'      # newline
'\r'      # carriage return
'\t'      # tab
'\v'      # vertical tab
'\uFFFF'  # hexadecimal unicode character
'\u{10FFFF}' # hexadecimal unicode character
```

A backslash followed by a `u` denotes a unicode codepoint. It can either be followed by exactly four hexadecimal characters representing the unicode bytes ( `\u0000` to `\uFFFF` ) or a number of one to six hexadecimal characters wrapped in curly braces ( `\u{0}` to `\u{10FFFF}` ).

```
'\u0041' # => 'A'
'\u{41}' # => 'A'
'\u{1F52E}' # => '🔮'
```



## String

A `String` represents an immutable sequence of UTF-8 characters.

A `String` is typically created with a string literal enclosing UTF-8 characters in double quotes ( `"` ):

```
"hello world"
```

## Escaping

A backslash denotes a special character inside a string, which can either be a named escape sequence or a numerical representation of a unicode codepoint.

Available escape sequences:

```
"\"      # double quote
"\"      # backslash
"\a"    # alert
"\b"    # backspace
"\e"    # escape
"\f"    # form feed
"\n"    # newline
"\r"    # carriage return
"\t"    # tab
"\v"    # vertical tab
"\888"  # octal ASCII character
"\xFF"  # hexadecimal ASCII character
"\uFFFF" # hexadecimal unicode character
"\u{0}".." \u{10FFFF}" # hexadecimal unicode character
```

Any other character following a backslash is interpreted as the character itself.

A backslash followed by at most three digits ranging from 0 to 7 denotes a code point written in octal:

```
"\101" # => "A"
"\123" # => "S"
"\12"  # => "\n"
"\1"   # string with one character with code point 1
```

A backslash followed by a `u` denotes a unicode codepoint. It can either be followed by exactly four hexadecimal characters representing the unicode bytes ( `\u0000` to `\uFFFF` ) or a number of one to six hexadecimal characters wrapped in curly braces ( `\u{0}` to `\u{10FFFF}` ).

```
"\u0041" # => "A"
"\u{41}" # => "A"
"\u{1F52E}" # => "🔮"
```

One curly brace can contain multiple unicode characters each separated by a whitespace.

```
"\u{48 45 4C 4C 4F}" # => "HELLO"
```

## Interpolation

A string literal with interpolation allows to embed expressions into the string which will be expanded at runtime.

```
a = 1
b = 2
"sum: #{a} + #{b} = #{a + b}" # => "sum: 1 + 2 = 3"
```

String interpolation is also possible with `String#%`.

Any expression may be placed inside the interpolated section, but it's best to keep the expression small for readability.

Interpolation can be disabled by escaping the `#` character with a backslash or by using a non-interpolating string literal like `%q()`.

```
"\#{a + b}" # => "\#{a + b}"
%q(#{a + b}) # => "\#{a + b}"
```

Interpolation is implemented using a `String::Builder` and invoking `Object#to_s(IO)` on each expression enclosed by `#{...}`. The expression `"sum: #{a} + #{b} = #{a + b}"` is equivalent to:

```
String.build do |io|
  io << "sum: "
  io << a
  io << " + "
  io << b
  io << " = "
  io << a + b
end
```

## Percent string literals

Besides double-quotes strings, Crystal also supports string literals indicated by a percent sign (`%`) and a pair of delimiters. Valid delimiters are parentheses `()`, square brackets `[]`, curly braces `{}`, angles `<>` and pipes `||`. Except for the pipes, all delimiters can be nested meaning a start delimiter inside the string escapes the next end delimiter.

These are handy to write strings that include double quotes which would have to be escaped in double-quoted strings.

```
%(hello ("world")) # => "hello (\\"world\\")"
%[hello ["world"]] # => "hello [\\"world\\"]"
%{hello {"world"}} # => "hello {\\"world\\"}"
%<hello <"world"> # => "hello <\\"world\\">"
%|hello "world"| # => "hello \\"world\\""
```

A literal denoted by `%q` does not apply interpolation nor escapes while `%Q` has the same meaning as `%`.

```
name = "world"
%q(hello \n #{name}) # => "hello \\n \#{name}"
%Q(hello \n #{name}) # => "hello \n world"
```

## Percent string array literal

Besides the single string literal, there is also a percent literal to create an [Array](#) of strings. It is indicated by `%w` and a pair of delimiters. Valid delimiters are as same as [percent string literals](#).

```
%w(foo bar baz) # => ["foo", "bar", "baz"]
%w(foo\ncbar baz) # => ["foo\\ncbar", "baz"]
%w(foo(bar) baz) # => ["foo(bar)", "baz"]
```

Note that literal denoted by `%w` does not apply interpolation nor escapes expect spaces. Since strings are separated by a single space character ( ) which must be escaped to use it as a part of a string.

```
%w(foo\ bar baz) # => ["foo bar", "baz"]
```

## Multiline strings

Any string literal can span multiple lines:

```
"hello
 world" # => "hello\n world"
```

Note that in the above example trailing and leading spaces, as well as newlines, end up in the resulting string. To avoid this a string can be split into multiple lines by joining multiple literals with a backslash:

```
"hello " \
"world, " \
"no newlines" # same as "hello world, no newlines"
```

Alternatively, a backslash followed by a newline can be inserted inside the string literal:

```
"hello \
  world, \
  no newlines" # same as "hello world, no newlines"
```

In this case, leading whitespace is not included in the resulting string.

## Heredoc

A *here document* or *heredoc* can be useful for writing strings spanning over multiple lines. A heredoc is denoted by `<<-` followed by an heredoc identifier which is an alphanumeric sequence starting with a letter (and may include underscores). The heredoc starts in the following line and ends with the next line that starts with the heredoc identifier (ignoring leading whitespace) and is either followed by a newline or a non-alphanumeric character.

```
<<-XML
<parent>
  <child />
</parent>
XML
```

Leading whitespace is removed from the heredoc contents according to the number of whitespace in the last line before the heredoc identifier.

```
<<-STRING # => "Hello\n  world"
Hello
  world
STRING

<<-STRING # => "  Hello\n  world"
  Hello
    world
STRING
```

It is possible to directly call methods on heredoc string literals, or use them inside parentheses:

```
<<-SOME.upcase # => "HELLO"
hello
SOME

def upcase(string)
  string.upcase
end

upcase(<<-SOME) # => "HELLO"
hello
SOME
```

A heredoc generally allows interpolation and escapes.

To denote a heredoc without interpolation or escapes, the opening heredoc identifier is enclosed in single quotes:

Nil

```
<<-'HERE' # => "hello \n \#{world}"  
hello \n \#{world}  
HERE
```

# Symbol

A [Symbol](#) represents a unique name inside the entire source code.

Symbols are interpreted at compile time and cannot be created dynamically. The only way to create a Symbol is by using a symbol literal, denoted by a colon ( : ) followed by an identifier. The identifier may optionally be enclosed in double quotes ( " ).

```
:unquoted_symbol
:"quoted symbol"
:"a" # identical to :a
:あ
```

A double-quoted identifier can contain any unicode character including white spaces and accepts the same escape sequences as a [string literal](#), yet no interpolation.

For an unquoted identifier the same naming rules apply as for methods. It can contain alphanumeric characters, underscore ( \_ ) or characters with a code point greater than 159 ( 0x9F ). It must not start with a number and may end with an exclamation mark ( ! ) or question mark ( ? ).

```
:question?
:exclamation!
```

All [Crystal operators](#) can be used as symbol names unquoted:

```
:+
:-
:*
:/
:%
:&
:|
:^
:**
:>>
:<<
:==
:!=
:<
:<=
:>
:>=
:<=>
:===
:[]
:[]?
:[]=
:~
:~
:~
:~
```

Internally, symbols are implemented as constants with a numeric value of type

```
Int32 .
```

## Percent symbol array literal

Besides the single symbol literal, there is also a percent literal to create an [Array](#) of symbols. It is indicated by `%i` and a pair of delimiters. Valid delimiters are parentheses `()`, square brackets `[]`, curly braces `{}`, angles `<>` and pipes `||`. Except for the pipes, all delimiters can be nested; meaning a start delimiter inside the string escapes the next end delimiter.

```
%i(foo bar baz) # => [:foo, :bar, :baz]
%i(foo\nbar baz) # => [:"foo\nbar", :baz]
%i(foo(bar) baz) # => [:"foo(bar)", :baz]
```

Identifiers may contain any unicode characters. Individual symbols are separated by a single space character ( ) which must be escaped to use it as a part of an identifier.

```
%i(foo\ bar baz) # => [:"foo bar", :baz]
```

## Array

An [Array](#) is an ordered and integer-indexed generic collection of elements of a specific type `T`.

Arrays are typically created with an array literal denoted by square brackets (`[]`) and individual elements separated by a comma (`,`).

```
[1, 2, 3]
```

## Generic Type Argument

The array's generic type argument `T` is inferred from the types of the elements inside the literal. When all elements of the array have the same type, `T` equals to that. Otherwise it will be a union of all element types.

```
[1, 2, 3] # => Array(Int32)
[1, "hello", 'x'] # => Array(Int32 | String | Char)
```

An explicit type can be specified by immediately following the closing bracket with `of` and a type, each separated by whitespace. This overwrites the inferred type and can be used for example to create an array that holds only some types initially but can accept other types later.

```
array_of_numbers = [1, 2, 3] of Float64 | Int32 # => Array(Float64 | Int32)
array_of_numbers << 0.5 # => [1, 2, 3, 0.5]

array_of_int_or_string = [1, 2, 3] of Int32 | String # => Array(Int32 | String)
array_of_int_or_string << "foo" # => [1, 2, 3, "foo"]
```

Empty array literals always need a type specification:

```
[] of Int32 # => Array(Int32).new
```

## Percent Array Literals

[Arrays of strings](#) and [arrays of symbols](#) can be created with percent array literals:

```
%w(one two three) # => ["one", "two", "three"]
%i(one two three) # => [:one, :two, :three]
```

## Array-like Type Literal



Crystal supports an additional literal for arrays and array-like types. It consists of the name of the type followed by a list of elements enclosed in curly braces ( `{}` ) and individual elements separated by a comma ( `,` ).

```
Array{1, 2, 3}
```

This literal can be used with any type as long as it has an argless constructor and responds to `<<`.

```
IO::Memory{1, 2, 3}
Set{1, 2, 3}
```

For a non-generic type like `IO::Memory`, this is equivalent to:

```
array_like = IO::Memory.new
array_like << 1
array_like << 2
array_like << 3
```

For a generic type like `Set`, the generic type `T` is inferred from the types of the elements in the same way as with the array literal. The above is equivalent to:

```
array_like = Set(typeof(1, 2, 3)).new
array_like << 1
array_like << 2
array_like << 3
```

The type arguments can be explicitly specified as part of the type name:

```
Set(Int32){1, 2, 3}
```

## Hash

A `Hash` is a generic collection of key-value pairs mapping keys of type `κ` to values of type `v`.

Hashes are typically created with a hash literal denoted by curly braces (`{ }`) enclosing a list of pairs using `=>` as delimiter between key and value and separated by commas `,`.

```
{"one" => 1, "two" => 2}
```

## Generic Type Argument

The generic type arguments for keys `κ` and values `v` are inferred from the types of the keys or values inside the literal, respectively. When all have the same type, `κ / v` equals to that. Otherwise it will be a union of all key types or value types respectively.

```
{1 => 2, 3 => 4} # Hash(Int32, Int32)
{1 => 2, 'a' => 3} # Hash(Int32 | Char, Int32)
```

Explicit types can be specified by immediately following the closing bracket with `of` (separated by whitespace), a key type (`κ`) followed by `=>` as delimiter and a value type (`v`). This overwrites the inferred types and can be used for example to create a hash that holds only some types initially but can accept other types as well.

Empty hash literals always need type specifications:

```
{} of Int32 => Int32 # => Hash(Int32, Int32).new
```

## Hash-like Type Literal

Crystal supports an additional literal for hashes and hash-like types. It consists of the name of the type followed by a list of comma separated key-value pairs enclosed in curly braces (`{ }`).

```
Hash{"one" => 1, "two" => 2}
```

This literal can be used with any type as long as it has an argless constructor and responds to `[]=`.

```
HTTP::Headers{"foo" => "bar"}
```

For a non-generic type like `HTTP::Headers`, this is equivalent to:

```
headers = HTTP::Headers.new
headers["foo"] = "bar"
```

For a generic type, the generic types are inferred from the types of the keys and values in the same way as with the hash literal.

```
MyHash{"foo" => 1, "bar" => "baz"}
```

If `MyHash` is generic, the above is equivalent to this:

```
my_hash = MyHash(typeof("foo", "bar"), typeof(1, "baz")).new
my_hash["foo"] = 1
my_hash["bar"] = "baz"
```

The type arguments can be explicitly specified as part of the type name:

```
MyHash(String, String | Int32){"foo" => "bar"}
```

## Range

A `Range` represents an interval between two values. It is typically constructed with a range literal, consisting of two or three dots:

- `x..y`: Two dots denote an inclusive range, including `x` and `y` and all values in between (in mathematics:  $[x, y]$ ).
- `x...y`: Three dots denote an exclusive range, including `x` and all values up to but not including `y` (in mathematics:  $[x, y)$ ).

```
(0..5).to_a # => [0, 1, 2, 3, 4, 5]
(0...5).to_a # => [0, 1, 2, 3, 4]
```

**NOTE:** Range literals are often wrapped in parentheses, for example if it is meant to be used as the receiver of a call. `0..5.to_a` without parentheses would be semantically equivalent to `0..(5.to_a)` because method calls and other operators have higher precedence than the range literal.

An easy way to remember which one is inclusive and which one is exclusive is to think of the extra dot as if it pushes `y` further away, thus leaving it outside of the range.

The literal `x..y` is semantically equivalent to the explicit constructor `Range.new(x, y)` and `x...y` to `Range.new(x, y, true)`.

The begin and end values do not necessarily need to be of the same type:

`true..1` is a valid range, although pretty useless. `Enumerable` methods won't work with incompatible types. They need at least to be comparable.

Ranges with `nil` as begin are called begin-less and `nil` as end are called end-less ranges. In the literal notation, `nil` can be omitted: `x..` is an end-less range starting from `x`, and `..x` is a begin-less range ending at `x`.

```
numbers = [1, 10, 3, 4, 5, 8]
numbers.select(6..) # => [10, 8]
numbers.select(..6) # => [1, 3, 4, 5]

numbers[2..] = [3, 4, 5, 8]
numbers[..2] = [1, 10, 3]
```

A range that is both begin-less and end-less is valid and can be expressed as `..` or `...` but it's typically not very useful.

## Regular Expressions

Regular expressions are represented by the [Regex](#) class.

A `Regex` is typically created with a regex literal using [PCRE](#) syntax. It consists of a string of UTF-8 character enclosed in forward slashes ( `/` ):

```
/foo|bar/
/h(e+)llo/
/\d+/
/あ/
```

## Escaping

Regular expressions support the same [escape sequences as String literals](#).

```
\/      # slash
\\      # backslash
\b      # backspace
\e      # escape
\f      # form feed
\n      # newline
\r      # carriage return
\t      # tab
\v      # vertical tab
\NNN    # octal ASCII character
\xNN    # hexadecimal ASCII character
\x{FFFF} # hexadecimal unicode character
\x{10FFFF} # hexadecimal unicode character
```

The delimiter character `/` must be escaped inside slash-delimited regular expression literals. Note that special characters of the PCRE syntax need to be escaped if they are intended as literal characters.

## Interpolation

Interpolation works in regular expression literals just as it does in [string literals](#). Be aware that using this feature will cause an exception to be raised at runtime, if the resulting string results in an invalid regular expression.

## Modifiers

The closing delimiter may be followed by a number of optional modifiers to adjust the matching behaviour of the regular expression.

- `i` : case-insensitive matching ( `PCRE_CASELESS` ): Unicode letters in the pattern match both upper and lower case letters in the subject string.
- `m` : multiline matching ( `PCRE_MULTILINE` ): The *start of line* ( `^` ) and *end of line* ( `$` ) metacharacters match immediately following or immediately before

internal newlines in the subject string, respectively, as well as at the very start and end.

- `x`: extended whitespace matching ( `PCRE_EXTENDED` ): Most white space characters in the pattern are totally ignored except when ignore or inside a character class. Unescaped hash characters `#` denote the start of a comment ranging to the end of the line.

```
/foo/i.match("FOO")      # => #<Regex::MatchData "FOO">
/foo/m.match("bar\nfoo") # => #<Regex::MatchData "foo">
/foo /x.match("foo")     # => #<Regex::MatchData "foo">
/foo /imx.match("bar\nFOO") # => #<Regex::MatchData "FOO">
```

## Percent regex literals

Besides slash-delimited literals, regular expressions may also be expressed as a percent literal indicated by `%r` and a pair of delimiters. Valid delimiters are parentheses `()`, square brackets `[]`, curly braces `{}`, angles `<>` and pipes `||`. Except for the pipes, all delimiters can be nested; meaning a start delimiter inside the literal escapes the next end delimiter.

These are handy to write regular expressions that include slashes which would have to be escaped in slash-delimited literals.

```
%r(/) # => /(\/)/
%r[[]] # => /[[]]/
%r{ } # => /{ }/
%r<</> # => /<\>/
%r|/| # => /\/|/
```

# Tuple

A [Tuple](#) is typically created with a tuple literal:

```
tuple = {1, "hello", 'x'} # Tuple(Int32, String, Char)
tuple[0]           # => 1      (Int32)
tuple[1]           # => "hello" (String)
tuple[2]           # => 'x'    (Char)
```

To create an empty tuple use [Tuple.new](#).

To denote a tuple type you can write:

```
# The type denoting a tuple of Int32, String and Char
Tuple(Int32, String, Char)
```

In type restrictions, generic type arguments and other places where a type is expected, you can use a shorter syntax, as explained in the [type grammar](#):

```
# An array of tuples of Int32, String and Char
Array({Int32, String, Char})
```

## NamedTuple

A `NamedTuple` is typically created with a named tuple literal:

```
tuple = {name: "Crystal", year: 2011} # NamedTuple(name: String, year: Int32)
tuple[:name]                        # => "Crystal" (String)
tuple[:year]                         # => 2011      (Int32)
```

To denote a named tuple type you can write:

```
# The type denoting a named tuple of x: Int32, y: String
NamedTuple(x: Int32, y: String)
```

In type restrictions, generic type arguments and other places where a type is expected, you can use a shorter syntax, as explained in the [type grammar](#):

```
# An array of named tuples of x: Int32, y: String
Array({x: Int32, y: String})
```

A named tuple key can also be a string literal:

```
{"this is a key": 1}
```



## Proc

A `Proc` represents a function pointer with an optional context (the closure data). It is typically created with a proc literal:

```
# A proc without arguments
->{ 1 } # Proc(Int32)

# A proc with one argument
->(x : Int32) { x.to_s } # Proc(Int32, String)

# A proc with two arguments:
->(x : Int32, y : Int32) { x + y } # Proc(Int32, Int32, Int32)
```

The types of the arguments are mandatory, except when directly sending a proc literal to a lib `fun` in C bindings.

The return type is inferred from the proc's body.

A special `new` method is provided too:

```
Proc(Int32, String).new { |x| x.to_s } # Proc(Int32, String)
```

This form allows you to specify the return type and to check it against the proc's body.

## The Proc type

To denote a Proc type you can write:

```
# A Proc accepting a single Int32 argument and returning a String
Proc(Int32, String)

# A proc accepting no arguments and returning Void
Proc(Void)

# A proc accepting two arguments (one Int32 and one String) and returning a Char
Proc(Int32, String, Char)
```

In type restrictions, generic type arguments and other places where a type is expected, you can use a shorter syntax, as explained in the [type](#):

```
# An array of Proc(Int32, String, Char)
Array(Int32, String -> Char)
```

## Invoking

To invoke a Proc, you invoke the `call` method on it. The number of arguments must match the proc's type:

```
proc = ->(x : Int32, y : Int32) { x + y }  
proc.call(1, 2) # => 3
```

## From methods

A Proc can be created from an existing method:

```
def one  
  1  
end  
  
proc = ->one  
proc.call # => 1
```

If the method has arguments, you must specify their types:

```
def plus_one(x)  
  x + 1  
end  
  
proc = ->plus_one(Int32)  
proc.call(41) # => 42
```

A proc can optionally specify a receiver:

```
str = "hello"  
proc = ->str.count(Char)  
proc.call('e') # => 1  
proc.call('l') # => 2
```

## Command literal

A command literal is a string delimited by backticks ``` or a `%x` percent literal. It will be substituted at runtime by the captured output from executing the string in a subshell.

The same [escaping](#) and [interpolation rules](#) apply as for regular strings.

Similar to percent string literals, valid delimiters for `%x` are parentheses `()`, square brackets `[]`, curly braces `{}`, angles `<>` and pipes `||`. Except for the pipes, all delimiters can be nested; meaning a start delimiter inside the string escapes the next end delimiter.

The special variable  `$?`  holds the exit status of the command as a

`Process::Status`. It is only available in the same scope as the command literal.

```
`echo foo` # => "foo"
$?.success? # => true
```

Internally, the compiler rewrites command literals to calls to the top-level method

``()`:String-class-method) with a string literal containing the command as argument: ``echo #{argument}`` and `%x(echo #{argument})` are rewritten to ``("echo #{argument}")`.

## Security concerns

While command literals may prove useful for simple script-like tools, special caution is advised when interpolating user input because it may easily lead to command injection.

```
user_input = "hello; rm -rf *"
`echo #{user_input}`
```

This command will write `hello` and subsequently delete all files and folders in the current working directory.

To avoid this, command literals should generally not be used with interpolated user input. `Process` from the standard library offers a safe way to provide user input as command arguments:

```
user_input = "hello; rm -rf *"
process = Process.new("echo", [user_input], output: Process::Redirect::Pipe)
process.output.gets_to_end # => "hello; rm -rf *"
process.wait.success?      # => true
```

# Assignment

Assignment is done using the equals sign (`=`).

```
# Assigns to a local variable
local = 1

# Assigns to an instance variable
@instance = 2

# Assigns to a class variable
@@class = 3
```

Each of the above kinds of variables will be explained later on.

Some syntax sugar that contains the `=` character is available:

```
local += 1 # same as: local = local + 1

# The above is valid with these operators:
# +, -, *, /, %, |, &, ^, **, <<, >>

local ||= 1 # same as: local || (local = 1)
local &&= 1 # same as: local && (local = 1)
```

A method invocation that ends with `=` has syntax sugar:

```
# A setter
person.name=("John")

# The above can be written as:
person.name = "John"

# An indexed assignment
objects.[](2, 3)

# The above can be written as:
objects[2] = 3

# Not assignment-related, but also syntax sugar:
objects.[](2, 3)

# The above can be written as:
objects[2, 3]
```

The `=` operator syntax sugar is also available to setters and indexers. Note that `||` and `&&` use the `[]?` method to check for key presence.

```

person.age += 1 # same as: person.age = person.age + 1

person.name ||= "John" # same as: person.name || (person.name = "John")
person.name &&= "John" # same as: person.name && (person.name = "John")

objects[1] += 2 # same as: objects[1] = objects[1] + 2

objects[1] ||= 2 # same as: objects[1]? || (objects[1] = 2)
objects[1] &&= 2 # same as: objects[1]? && (objects[1] = 2)

```

## Chained assignment

You can assign the same value to multiple variables using chained assignment:

```

a = b = c = 123

# Now a, b and c have the same value:
a # => 123
b # => 123
c # => 123

```

The chained assignment is not only available to [local variables](#) but also to [instance variables](#), [class variables](#) and setter methods (methods that end with `=`).

## Multiple assignment

You can declare/assign multiple variables at the same time by separating expressions with a comma ( , ):

```

name, age = "Crystal", 1

# The above is the same as this:
temp1 = "Crystal"
temp2 = 1
name = temp1
age = temp2

```

Note that because expressions are assigned to temporary variables it is possible to exchange variables' contents in a single line:

```

a = 1
b = 2
a, b = b, a
a # => 2
b # => 1

```

If the right-hand side contains just one expression, the type is indexed for each variable on the left-hand side like so:

```
name, age, source = "Crystal, 123, GitHub".split(", ")

# The above is the same as this:
temp = "Crystal, 123, GitHub".split(", ")
name = temp[0]
age = temp[1]
source = temp[2]
```

Multiple assignment is also available to methods that end with `= :`

```
person.name, person.age = "John", 32

# Same as:
temp1 = "John"
temp2 = 32
person.name = temp1
person.age = temp2
```

And it is also available to [index assignments](#) (`[]=`):

```
objects[1], objects[2] = 3, 4

# Same as:
temp1 = 3
temp2 = 4
objects[1] = temp1
objects[2] = temp2
```

## Local variables

Local variables start with lowercase letters. They are declared when you first assign them a value.

```
name = "Crystal"  
age = 1
```

Their type is inferred from their usage, not only from their initializer. In general, they are just value holders associated with the type that the programmer expects them to have according to their location and usage on the program.

For example, reassigning a variable with a different expression makes it have that expression's type:

```
flower = "Tulip"  
# At this point 'flower' is a String  
  
flower = 1  
# At this point 'flower' is an Int32
```

Underscores are allowed at the beginning of a variable name, but these names are reserved for the compiler, so their use is not recommended (and it also makes the code uglier to read).

## Control expressions

Before talking about control expressions we need to know what *truthy* and *falsey* values are.



## Truthy and falsey values

A *truthy* value is a value that is considered true for an `if`, `unless`, `while` or `until` guard. A *falsey* value is a value that is considered false in those places.

The only falsey values are `nil`, `false` and null pointers (pointers whose memory address is zero). Any other value is truthy.

## if

An `if` evaluates the given branch if its condition is *truthy*. Otherwise, it evaluates the `else` branch if present.

```
a = 1
if a > 0
  a = 10
end
a # => 10

b = 1
if b > 2
  b = 10
else
  b = 20
end
b # => 20
```

To write a chain of if-else-if you use `elsif` :

```
if some_condition
  do_something
elsif some_other_condition
  do_something_else
else
  do_that
end
```

After an `if`, a variable's type depends on the type of the expressions used in both branches.

```
a = 1
if some_condition
  a = "hello"
else
  a = true
end
# a : String | Bool

b = 1
if some_condition
  b = "hello"
end
# b : Int32 | String

if some_condition
  c = 1
else
  c = "hello"
end
# c : Int32 | String

if some_condition
  d = 1
end
# d : Int32 | Nil
```

Note that if a variable is declared inside one of the branches but not in the other one, at the end of the `if` it will also contain the `Nil` type.

Inside an `if`'s branch the type of a variable is the one it got assigned in that branch, or the one that it had before the branch if it was not reassigned:

```
a = 1
if some_condition
  a = "hello"
  # a : String
  a.size
end
# a : String | Int32
```

That is, a variable's type is the type of the last expression(s) assigned to it.

If one of the branches never reaches past the end of an `if`, like in the case of a `return`, `next`, `break` or `raise`, that type is not considered at the end of the `if`:

```
if some_condition
  e = 1
else
  e = "hello"
  # e : String
  return
end
# e : Int32
```

## As a suffix

An `if` can be written as an expression's suffix:

```
a = 2 if some_condition

# The above is the same as:
if some_condition
  a = 2
end
```

This sometimes leads to code that is more natural to read.

## As an expression

The value of an `if` is the value of the last expression found in each of its branches:

```
a = if 2 > 1
  3
else
  4
end
a # => 3
```

If an `if` branch is empty, or it's missing, it's considered as if it had `nil` in it:

```
if 1 > 2
  3
end

# The above is the same as:
if 1 > 2
  3
else
  nil
end

# Another example:
if 1 > 2
else
  3
end

# The above is the same as:
if 1 > 2
  nil
else
  3
end
```

## Ternary if

The ternary `if` allows writing an `if` in a shorter way:

```
a = 1 > 2 ? 3 : 4

# The above is the same as:
a = if 1 > 2
    3
  else
    4
  end
```

## if var

If a variable is the condition of an `if`, inside the `then` branch the variable will be considered as not having the `Nil` type:

```
a = some_condition ? nil : 3
# a is Int32 or Nil

if a
  # Since the only way to get here is if a is truthy,
  # a can't be nil. So here a is Int32.
  a.abs
end
```

This also applies when a variable is assigned in an `if`'s condition:

```
if a = some_expression
  # here a is not nil
end
```

This logic also applies if there are ands (`&&`) in the condition:

```
if a && b
  # here both a and b are guaranteed not to be Nil
end
```

Here, the right-hand side of the `&&` expression is also guaranteed to have `a` as not `Nil`.

Of course, reassigning a variable inside the `then` branch makes that variable have a new type based on the expression assigned.

## Limitations

The above logic works **only for local variables**. It doesn't work with instance variables, class variables, or variables bound in a closure. The value of these kinds of variables could potentially be affected by another fiber after the condition was checked, rendering it `nil`. It also does not work with constants.

```

if @a
  # here `@a` can be nil
end

if @@a
  # here `@@a` can be nil
end

a = nil
closure = ->{ a = "foo" }

if a
  # here `a` can be nil
end

```

This can be circumvented by assigning the value to a new local variable:

```

if a = @a
  # here `a` can't be nil
end

```

Another option is to use `Object#try` found in the standard library which only executes the block if the value is not `nil`:

```

@a.try do |a|
  # here `a` can't be nil
end

```

## Method calls

That logic also doesn't work with proc and method calls, including getters and properties, because nilable (or, more generally, union-typed) procs and methods aren't guaranteed to return the same more-specific type on two successive calls.

```

if method # first call to a method that can return Int32 or Nil
  # here we know that the first call did not return Nil
  method # second call can still return Int32 or Nil
end

```

The techniques described above for instance variables will also work for proc and method calls.



## if var.is\_a?(...)

If an `if`'s condition is an `is_a?` test, the type of a variable is guaranteed to be restricted by that type in the `then` branch.

```
if a.is_a?(String)
  # here a is a String
end

if b.is_a?(Number)
  # here b is a Number
end
```

Additionally, in the `else` branch the type of the variable is guaranteed to not be restricted by that type:

```
a = some_condition ? 1 : "hello"
# a : Int32 | String

if a.is_a?(Number)
  # a : Int32
else
  # a : String
end
```

Note that you can use any type as an `is_a?` test, like abstract classes and modules.

The above also works if there are ands ( `&&` ) in the condition:

```
if a.is_a?(String) && b.is_a?(Number)
  # here a is a String and b is a Number
end
```

The above **doesn't** work with instance variables or class variables. To work with these, first assign them to a variable:

```
if @a.is_a?(String)
  # here @a is not guaranteed to be a String
end

a = @a
if a.is_a?(String)
  # here a is guaranteed to be a String
end

# A bit shorter:
if (a = @a).is_a?(String)
  # here a is guaranteed to be a String
end
```

## if var.responds\_to?(...)

If an `if`'s condition is a `responds_to?` test, in the `then` branch the type of a variable is guaranteed to be restricted to the types that respond to that method:

```
if a.responds_to?(:abs)
  # here a's type will be reduced to those responding to the 'abs' method
end
```

Additionally, in the `else` branch the type of the variable is guaranteed to be restricted to the types that don't respond to that method:

```
a = some_condition ? 1 : "hello"
# a : Int32 | String

if a.responds_to?(:abs)
  # here a will be Int32, since Int32#abs exists but String#abs doesn't
else
  # here a will be String
end
```

The above **doesn't** work with instance variables or class variables. To work with these, first assign them to a variable:

```
if @a.responds_to?(:abs)
  # here @a is not guaranteed to respond to `abs`
end

a = @a
if a.responds_to?(:abs)
  # here a is guaranteed to respond to `abs`
end

# A bit shorter:
if (a = @a).responds_to?(:abs)
  # here a is guaranteed to respond to `abs`
end
```

## if var.nil?

If an `if`'s condition is `var.nil?` then the type of `var` in the `then` branch is known by the compiler to be `Nil`, and to be known as non-`Nil` in the `else` branch:

```
a = some_condition ? nil : 3
if a.nil?
  # here a is Nil
else
  # here a is Int32
end
```

## if !

The `!` operator returns a `Bool` that results from negating the [truthiness](#) of a value.

When used in an `if` in conjunction with a variable, `is_a?`, `responds_to?` OR `nil?` the compiler will restrict the types accordingly:

```
a = some_condition ? nil : 3
if !a
  # here a is Nil because a is falsey in this branch
else
  # here a is Int32, because a is truthy in this branch
end
```

```
b = some_condition ? 1 : "x"
if !b.is_a?(Int32)
  # here b is String because it's not an Int32
end
```

## unless

An `unless` evaluates the then branch if its condition is *falsey*, and evaluates the `else` branch, if there's any, otherwise. That is, it behaves in the opposite way of an `if`:

```
unless some_condition
  then_expression
else
  else_expression
end

# The above is the same as:
if some_condition
  else_expression
else
  then_expression
end

# Can also be written as a suffix
close_door unless door_closed?
```

## case

A `case` is a control expression which functions a bit like pattern matching. It allows writing a chain of if-else-if with a small change in semantic and some more powerful constructs.

In its basic form, it allows matching a value against other values:

```
case exp
when value1, value2
  do_something
when value3
  do_something_else
else
  do_another_thing
end

# The above is the same as:
tmp = exp
if value1 === tmp || value2 === tmp
  do_something
elsif value3 === tmp
  do_something_else
else
  do_another_thing
end
```

For comparing an expression against a `case`'s value the *case equality operator* `===` is used. It is defined as a method on `Object` and can be overridden by subclasses to provide meaningful semantics in case statements. For example, `Class` defines case equality as when an object is an instance of that class, `Regex` as when the value matches the regular expression and `Range` as when the value is included in that range.

If a `when`'s expression is a type, `is_a?` is used. Additionally, if the case expression is a variable or a variable assignment the type of the variable is restricted:

```

case var
when String
  # var : String
  do_something
when Int32
  # var : Int32
  do_something_else
else
  # here var is neither a String nor an Int32
  do_another_thing
end

# The above is the same as:
if var.is_a?(String)
  do_something
elsif var.is_a?(Int32)
  do_something_else
else
  do_another_thing
end

```

You can invoke a method on the `case`'s expression in a `when` by using the implicit-object syntax:

```

case num
when .even?
  do_something
when .odd?
  do_something_else
end

# The above is the same as:
tmp = num
if tmp.even?
  do_something
elsif tmp.odd?
  do_something_else
end

```

You may use `then` after the `when` condition to place the body on a single line.

```

case exp
when value1, value2 then do_something
when value3          then do_something_else
else                  do_another_thing
end

```

Finally, you can omit the `case`'s value:

```

case
when cond1, cond2
  do_something
when cond3
  do_something_else
end

# The above is the same as:
if cond1 || cond2
  do_something
elsif cond3
  do_something_else
end

```

This sometimes leads to code that is more natural to read.

## Tuple literal

When a case expression is a tuple literal there are a few semantic differences if a

`when` condition is also a tuple literal.

## Tuple size must match

```

case {value1, value2}
when {0, 0} # OK, 2 elements
  # ...
when {1, 2, 3} # Syntax error: wrong number of tuple elements (given 3, expected 2)
  # ...
end

```

## Underscore allowed

```

case {value1, value2}
when {0, _}
  # Matches if 0 === value1, no test done against value2
when {_, 0}
  # Matches if 0 === value2, no test done against value1
end

```

## Implicit-object allowed

```

case {value1, value2}
when { .even?, .odd? }
  # Matches if value1.even? && value2.odd?
end

```

## Comparing against a type will perform an `is_a?` check



Nil

```
case {value1, value2}
when {String, Int32}
  # Matches if value1.is_a?(String) && value2.is_a?(Int32)
  # The type of value1 is known to be a String by the compiler,
  # and the type of value2 is known to be an Int32
end
```

## while

A `while` executes its body as long as its condition is *truthy*.

```
while some_condition
  do_this
end
```

The condition is first tested and, if *truthy*, the body is executed. That is, the body might never be executed.

A `while`'s type is always `Nil`.

Similar to an `if`, if a `while`'s condition is a variable, the variable is guaranteed to not be `nil` inside the body. If the condition is an `var.is_a?(Type)` test, `var` is guaranteed to be of type `Type` inside the body. And if the condition is a `var.responds_to?(:method)`, `var` is guaranteed to be of a type that responds to that method.

The type of a variable after a `while` depends on the type it had before the `while` and the type it had before leaving the `while`'s body:

```
a = 1
while some_condition
  # a : Int32 | String
  a = "hello"
  # a : String
  a.size
end
# a : Int32 | String
```

## Checking the condition at the end of a loop

If you need to execute the body at least once and then check for a breaking condition, you can do this:

```
while true
  do_something
  break if some_condition
end
```

Or use `loop`, found in the standard library:

```
loop do
  do_something
  break if some_condition
end
```

# break

You can use `break` to break out of a `while` loop:

```
a = 2
while (a += 1) < 20
  if a == 10
    break # goes to 'puts a'
  end
end
puts a # => 10
```

`break` can also take a parameter which will then be the value that gets returned:

```
def foo
  loop do
    break "bar"
  end
end

puts foo # => "bar"
```

## next

You can use `next` to try to execute the next iteration of a `while` loop. After executing `next`, the `while`'s condition is checked and, if *truthy*, the body will be executed.

```
a = 1
while a < 5
  a += 1
  if a == 3
    next
  end
  puts a
end

# The above prints the numbers 2, 4 and 5
```

`next` can also be used to exit from a block, for example:

```
def block
  yield
end

block do
  puts "hello"
  next
  puts "world"
end

# The above prints "hello"
```

Similar to `break`, `next` can also take a parameter which will then be returned by `yield`.

```
def block
  puts yield
end

block do
  next "hello"
end

# The above prints "hello"
```

## until

An `until` executes its body until its condition is *truthy*. An `until` is just syntax sugar for a `while` with the condition negated:

```
until some_condition
  do_this
end

# The above is the same as:
while !some_condition
  do_this
end
```

`break` and `next` can also be used inside an `until`.

## && - Logical AND Operator

An `&&` (and) evaluates its left hand side. If it's *truthy*, it evaluates its right hand side and has that value. Otherwise it has the value of the left hand side. Its type is the union of the types of both sides.

You can think an `&&` as syntax sugar of an `if` :

```
some_exp1 && some_exp2

# The above is the same as:
tmp = some_exp1
if tmp
  some_exp2
else
  tmp
end
```

## || - Logical OR Operator

An `||` (or) evaluates its left hand side. If it's *falsey*, it evaluates its right hand side and has that value. Otherwise it has the value of the left hand side. Its type is the union of the types of both sides.

You can think an `||` as syntax sugar of an `if` :

```
some_exp1 || some_exp2

# The above is the same as:
tmp = some_exp1
if tmp
  tmp
else
  some_exp2
end
```

## Requiring files

Writing a program in a single file is OK for little snippets and small benchmark code. Big programs are better maintained and understood when split across different files.

To make the compiler process other files you use `require "..."`. It accepts a single argument, a string literal, and it can come in many flavors.

Once a file is required, the compiler remembers its absolute path and later `require`s of that same file will be ignored.

### require "filename"

This looks up "filename" in the require path.

By default the require path is the location of the standard library that comes with the compiler, and the "lib" directory relative to the current working directory (given by `pwd` in a Unix shell). These are the only places that are looked up.

The lookup goes like this:

- If a file named "filename.cr" is found in the require path, it is required.
- If a directory named "filename" is found and it contains a file named "filename.cr" directly underneath it, it is required.
- If a directory named "filename" is found with a directory "src" in it and it contains a file named "filename.cr" directly underneath it, it is required.
- If a directory named "filename" is found with a directory "src" in it and it contains a directory named "filename" directly underneath it with a "filename.cr" file inside it, it is required.
- Otherwise a compile-time error is issued.

The second rule means that in addition to having this:

```
- project
  - src
    - file
      - sub1.cr
      - sub2.cr
    - file.cr (requires "./file/*")
```

you can have it like this:

```
- project
  - src
    - file
      - file.cr (requires "./*")
      - sub1.cr
      - sub2.cr
```

which might be a bit cleaner depending on your taste.



The third rule is very convenient because of the typical directory structure of a project:

```
- project
- lib
  - foo
    - src
      - foo.cr
  - bar
    - src
      - bar.cr
- src
- project.cr
- spec
- project_spec.cr
```

That is, inside "lib/{project}" each project's directory exists ( `src` , `spec` , `README.md` and so on)

For example, if you put `require "foo"` in `project.cr` and run `crystal src/project.cr` in the project's root directory, it will find `foo` in `lib/foo/foo.cr` .

The fourth rule is the second rule applied to the third rule.

If you run the compiler from somewhere else, say the `src` folder, `lib` will not be in the path and `require "foo"` can't be resolved.

## require "./filename"

This looks up "filename" relative to the file containing the require expression.

The lookup goes like this:

- If a file named "filename.cr" is found relative to the current file, it is required.
- If a directory named "filename" is found and it contains a file named "filename.cr" directly underneath it, it is required.
- Otherwise a compile-time error is issued.

This relative is mostly used inside a project to refer to other files inside it. It is also used to refer to code from [specs](#):

```
# in spec/project_spec.cr
require "../src/project"
```

## Other forms

In both cases you can use nested names and they will be looked up in nested directories:

- `require "foo/bar/baz"` will lookup "foo/bar/baz.cr", "foo/bar/baz/baz.cr", "foo/src/bar/baz.cr" or "foo/src/bar/baz/baz.cr" in the require path.
- `require "./foo/bar/baz"` will lookup "foo/bar/baz.cr" or "foo/bar/baz/baz.cr" relative to the current file.

You can also use `../` to access parent directories relative to the current file, so `require "../../foo/bar"` works as well.

In all of these cases you can use the special `*` and `**` suffixes:

- `require "foo/*"` will require all `.cr` files below the `"foo"` directory, but not below directories inside `"foo"`.
- `require "foo/**"` will require all `.cr` files below the `"foo"` directory, and below directories inside `"foo"`, recursively.

## Types and methods

The next sections will assume you know what [object oriented programming](#) is, as well as the concepts of [classes](#) and [methods](#).

## Everything is an object

In Crystal everything is an object. The definition of an object boils down to these points:

- It has a type
- It can respond to some methods

This is everything you can know about an object: its type and whether it responds to some method.

An object's internal state, if any, can only be queried by invoking methods.

## The Program

The program is a global object in which you can define types, methods and file-local variables.

```
# Defines a method in the program
def add(x, y)
  x + y
end

# Invokes the add method in the program
add(1, 2) # => 3
```

A method's value is the value of its last expression; there's no need for explicit `return` expressions. However, explicit `return` expressions are possible:

```
def even?(num)
  if num % 2 == 0
    return true
  end

  return false
end
```

When invoking a method without a receiver, like `add(1, 2)`, it will be searched for in the program if not found in the current type or any of its ancestors.

```
def add(x, y)
  x + y
end

class Foo
  def bar
    # invokes the program's add method
    add(1, 2)

    # invokes Foo's baz method
    baz(1, 2)
  end

  def baz(x, y)
    x * y
  end
end
```

If you want to invoke the program's method, even though the current type defines a method with the same name, prefix the call with `::`:

```
def baz(x, y)
  x + y
end

class Foo
  def bar
    baz(4, 2) # => 2
    ::baz(4, 2) # => 6
  end

  def baz(x, y)
    x - y
  end
end
```

Variables declared in a program are not visible inside methods:

```
x = 1

def add(y)
  x + y # error: undefined local variable or method 'x'
end

add(2)
```

Parentheses in method invocations are optional:

```
add 1, 2 # same as add(1, 2)
```

## Main code

Main code, the code that is run when you compile and run a program, can be written directly in a source file without the need to put it in a special "main" method:

```
# This is a program that prints "Hello Crystal!"
puts "Hello Crystal!"
```

Main code can also be inside type declarations:

```
# This is a program that prints "Hello"
class Hello
  # 'self' here is the Hello class
  puts self
end
```

## Classes and methods

A class is a blueprint from which individual objects are created. As an example, consider a `Person` class. You declare a class like this:

```
class Person  
end
```

Class names, and indeed all type names, begin with a capital letter in Crystal.

## new, initialize and allocate

You create an instance of a class by invoking `new` on that class:

```
person = Person.new
```

Here, `person` is an instance of `Person`.

We can't do much with `person`, so let's add some concepts to it. A `Person` has a name and an age. In the "Everything is an object" section we said that an object has a type and responds to some methods, which is the only way to interact with objects, so we'll need both `name` and `age` methods. We will store this information in instance variables, which are always prefixed with an *at* (`@`) character. We also want a `Person` to come into existence with a name of our choice and an age of zero. We code the "come into existence" part with a special `initialize` method, which is normally called a *constructor*:

```
class Person
  def initialize(name : String)
    @name = name
    @age = 0
  end

  def name
    @name
  end

  def age
    @age
  end
end
```

Now we can create people like this:

```
john = Person.new "John"
peter = Person.new "Peter"

john.name # => "John"
john.age # => 0

peter.name # => "Peter"
```

(If you wonder why we needed to specify that `name` is a `String` but we didn't need to do it for `age`, check the [global type inference algorithm](#))

Note that we create a `Person` with `new` but we defined the initialization in an `initialize` method, not in a `new` method. Why is this so?

The answer is that when we defined an `initialize` method Crystal defined a `new` method for us, like this:



```
class Person
  def self.new(name : String)
    instance = Person.allocate
    instance.initialize(name)
    instance
  end
end
```

First, note the `self.new` notation. This is a [class method](#) that belongs to the **class** `Person`, not to particular instances of that class. This is why we can do `Person.new`.

Second, `allocate` is a low-level class method that creates an uninitialized object of the given type. It basically allocates the necessary memory for the object, then `initialize` is invoked on it and finally the instance is returned. You generally never invoke `allocate`, as it is [unsafe](#), but that's the reason why `new` and `initialize` are related.

## Methods and instance variables

We can simplify our constructor by using a shorter syntax for assigning a method argument to an instance variable:

```
class Person
  def initialize(@name : String)
    @age = 0
  end

  def age
    @age
  end
end
```

Right now, we can't do much with a person aside from create it with a name. Its age will always be zero. So lets add a method that makes a person become older:

```
class Person
  def initialize(@name : String)
    @age = 0
  end

  def age
    @age
  end

  def become_older
    @age += 1
  end
end

john = Person.new "John"
peter = Person.new "Peter"

john.age # => 0

john.become_older
john.age # => 1

peter.age # => 0
```

Method names begin with a lowercase letter and, as a convention, only use lowercase letters, underscores and numbers.

## Getters and setters

The Crystal [Standard Library](#) provides macros which simplify the definition of getter and setter methods:

```

class Person
  property age
  getter name : String

  def initialize(@name)
    @age = 0
  end
end

john = Person.new "John"
john.age = 32
john.age # => 32

```

For more information on getter and setter macros, see the standard library documentation for [Object#getter](#), [Object#setter](#), and [Object#property](#).

As a side note, we can define `become_older` inside the original `Person` definition, or in a separate definition: Crystal combines all definitions into a single class. The following works just fine:

```

class Person
  def initialize(@name : String)
    @age = 0
  end
end

class Person
  def become_older
    @age += 1
  end
end

```

## Redefining methods, and `previous_def`

If you redefine a method, the last definition will take precedence.

```

class Person
  def become_older
    @age += 1
  end
end

class Person
  def become_older
    @age += 2
  end
end

person = Person.new "John"
person.become_older
person.age # => 2

```

You can invoke the previously redefined method with `previous_def` :

```
class Person
  def become_older
    @age += 1
  end
end

class Person
  def become_older
    previous_def
    @age += 2
  end
end

person = Person.new "John"
person.become_older
person.age # => 3
```

Without arguments or parentheses, `previous_def` receives the same arguments as the method's arguments. Otherwise, it receives the arguments you pass to it.

## Catch-all initialization

Instance variables can also be initialized outside `initialize` methods:

```
class Person
  @age = 0

  def initialize(@name : String)
  end
end
```

This will initialize `@age` to zero in every constructor. This is useful to avoid duplication, but also to avoid the `Nil` type when reopening a class and adding instance variables to it.

## Type inference

Crystal's philosophy is to require as few type restrictions as possible. However, some restrictions are required.

Consider a class definition like this:

```
class Person
  def initialize(@name)
    @age = 0
  end
end
```

We can quickly see that `@age` is an integer, but we don't know the type of `@name`. The compiler could infer its type from all uses of the `Person` class. However, doing so has a few issues:

- The type is not obvious for a human reading the code: they would also have to check all uses of `Person` to find this out.
- Some compiler optimizations, like having to analyze a method just once, and incremental compilation, are nearly impossible to do.

As a code base grows, these issues gain more relevance: understanding a project becomes harder, and compile times become unbearable.

For this reason, Crystal needs to know, in an obvious way (as obvious as to a human), the types of instance and `class` variables.

There are several ways to let Crystal know this.

## With type restrictions

The easiest, but probably most tedious, way is to use explicit type restrictions.

```
class Person
  @name : String
  @age : Int32

  def initialize(@name)
    @age = 0
  end
end
```

## Without type restrictions

If you omit an explicit type restriction, the compiler will try to infer the type of instance and class variables using a bunch of syntactic rules.

For a given instance/class variable, when a rule can be applied and a type can be guessed, the type is added to a set. When no more rules can be applied, the inferred type will be the [union](#) of those types. Additionally, if the compiler infers

that an instance variable isn't always initialized, it will also include the `Nil` type.

The rules are many, but usually the first three are most used. There's no need to remember them all. If the compiler gives an error saying that the type of an instance variable can't be inferred you can always add an explicit type restriction.

The following rules only mention instance variables, but they apply to class variables as well. They are:

## 1. Assigning a literal value

When a literal is assigned to an instance variable, the literal's type is added to the set. All `literals` have an associated type.

In the following example, `@name` is inferred to be `String` and `@age` to be `Int32`.

```
class Person
  def initialize
    @name = "John Doe"
    @age = 0
  end
end
```

This rule, and every following rule, will also be applied in methods other than `initialize`. For example:

```
class SomeObject
  def lucky_number
    @lucky_number = 42
  end
end
```

In the above case, `@lucky_number` will be inferred to be `Int32 | Nil : Int32` because 42 was assigned to it, and `Nil` because it wasn't assigned in all of the class' initialize methods.

## 2. Assigning the result of invoking the class method `new`

When an expression like `Type.new(...)` is assigned to an instance variable, the type `Type` is added to the set.

In the following example, `@address` is inferred to be `Address`.

```
class Person
  def initialize
    @address = Address.new("somewhere")
  end
end
```

This also is applied to generic types. Here `@values` is inferred to be `Array(Int32)`.

```
class Something
  def initialize
    @values = Array(Int32).new
  end
end
```

**Note:** a `new` method might be redefined by a type. In that case the inferred type will be the one returned by `new`, if it can be inferred using some of the next rules.

### 3. Assigning a variable that is a method argument with a type restriction

In the following example `@name` is inferred to be `String` because the method argument `name` has a type restriction of type `String`, and that argument is assigned to `@name`.

```
class Person
  def initialize(name : String)
    @name = name
  end
end
```

Note that the name of the method argument is not important; this works as well:

```
class Person
  def initialize(obj : String)
    @name = obj
  end
end
```

Using the shorter syntax to assign an instance variable from a method argument has the same effect:

```
class Person
  def initialize(@name : String)
  end
end
```

Also note that the compiler doesn't check whether a method argument is reassigned a different value:

```
class Person
  def initialize(name : String)
    name = 1
    @name = name
  end
end
```

In the above case, the compiler will still infer `@name` to be `String`, and later will give a compile time error, when fully typing that method, saying that `Int32` can't be assigned to a variable of type `String`. Use an explicit type restriction if `@name` isn't supposed to be a `String`.

## 4. Assigning the result of a class method that has a return type restriction

In the following example, `@address` is inferred to be `Address`, because the class method `Address.unknown` has a return type restriction of `Address`.

```
class Person
  def initialize
    @address = Address.unknown
  end
end

class Address
  def self.unknown : Address
    new("unknown")
  end

  def initialize(@name : String)
    end
end
```

In fact, the above code doesn't need the return type restriction in `self.unknown`. The reason is that the compiler will also look at a class method's body and if it can apply one of the previous rules (it's a `new` method, or it's a literal, etc.) it will infer the type from that expression. So, the above can be simply written like this:

```
class Person
  def initialize
    @address = Address.unknown
  end
end

class Address
  # No need for a return type restriction here
  def self.unknown
    new("unknown")
  end

  def initialize(@name : String)
    end
end
```

This extra rule is very convenient because it's very common to have "constructor-like" class methods in addition to `new`.

## 5. Assigning a variable that is a method argument with a default value

In the following example, because the default value of `name` is a string literal, and it's later assigned to `@name`, `String` will be added to the set of inferred types.

```
class Person
  def initialize(name = "John Doe")
    @name = name
  end
end
```



This of course also works with the short syntax:

```
class Person
  def initialize(@name = "John Doe")
  end
end
```

The default value can also be a `Type.new(...)` method or a class method with a return type restriction.

## 6. Assigning the result of invoking a `lib` function

Because a `lib function` must have explicit types, the compiler can use the return type when assigning it to an instance variable.

In the following example `@age` is inferred to be `Int32`.

```
class Person
  def initialize
    @age = LibPerson.compute_default_age
  end
end

lib LibPerson
  fun compute_default_age : Int32
end
```

## 7. Using an `out` `lib` expression

Because a `lib function` must have explicit types, the compiler can use the `out` argument's type, which should be a pointer type, and use the dereferenced type as a guess.

In the following example `@age` is inferred to be `Int32`.

```
class Person
  def initialize
    LibPerson.compute_default_age(out @age)
  end
end

lib LibPerson
  fun compute_default_age(age_ptr : Int32*)
end
```

## Other rules

The compiler will try to be as smart as possible to require less explicit type restrictions. For example, if assigning an `if` expression, type will be inferred from the `then` and `else` branches:

```
class Person
  def initialize
    @age = some_condition ? 1 : 2
  end
end
```

Because the `if` above (well, technically a ternary operator, but it's similar to an `if`) has integer literals, `@age` is successfully inferred to be `Int32` without requiring a redundant type restriction.

Another case is `||` and `||=`:

```
class SomeObject
  def lucky_number
    @lucky_number ||= 42
  end
end
```

In the above example `@lucky_number` will be inferred to be `Int32 | Nil`. This is very useful for lazily initialized variables.

Constants will also be followed, as it's pretty simple for the compiler (and a human) to do so.

```
class SomeObject
  DEFAULT_LUCKY_NUMBER = 42

  def initialize(@lucky_number = DEFAULT_LUCKY_NUMBER)
  end
end
```

Here rule 5 (argument's default value) is used, and because the constant resolves to an integer literal, `@lucky_number` is inferred to be `Int32`.

## Union types

The type of a variable or expression can consist of multiple types. This is called a union type. For example, when assigning to a same variable inside different `if` branches:

```
if 1 + 2 == 3
  a = 1
else
  a = "hello"
end

a # : Int32 | String
```

At the end of the `if`, `a` will have the `Int32 | String` type, read "the union of `Int32` and `String`". This union type is created automatically by the compiler. At runtime, `a` will of course be of one type only. This can be seen by invoking the `class` method:

```
# The runtime type
a.class # => Int32
```

The compile-time type can be seen by using `typeof`:

```
# The compile-time type
typeof(a) # => Int32 | String
```

A union can consist of an arbitrary large number of types. When invoking a method on an expression whose type is a union type, all types in the union must respond to the method, otherwise a compile-time error is given. The type of the method call is the union type of the return types of those methods.

```
# to_s is defined for Int32 and String, it returns String
a.to_s # => String

a + 1 # Error, because String#+(Int32) isn't defined
```

If necessary a variable can be defined as a union type at compile time

```
# set the compile-time type
a = 0.as(Int32 | Nil | String)
typeof(a) # => Int32 | Nil | String
```

## Union types rules

In the general case, when two types `T1` and `T2` are combined, the result is a union `T1 | T2`. However, there are a few cases where the resulting type is a different type.

## Union of classes and structs under the same hierarchy

If `T1` and `T2` are under the same hierarchy, and their nearest common ancestor `Parent` is not `Reference`, `Struct`, `Int`, `Float` nor `Value`, the resulting type is `Parent+`. This is called a virtual type, which basically means the compiler will now see the type as being `Parent` or any of its subtypes.

For example:

```
class Foo
end

class Bar < Foo
end

class Baz < Foo
end

bar = Bar.new
baz = Baz.new

# Here foo's type will be Bar | Baz,
# but because both Bar and Baz inherit from Foo,
# the resulting type is Foo+
foo = rand < 0.5 ? bar : baz
typeof(foo) # => Foo+
```

## Union of tuples of the same size

The union of two tuples of the same size results in a tuple type that has the union of the types in each position.

For example:

```
t1 = {1, "hi"} # Tuple(Int32, String)
t2 = {true, nil} # Tuple(Bool, Nil)

t3 = rand < 0.5 ? t1 : t2
typeof(t3) # Tuple(Int32 | Bool, String | Nil)
```

## Union of named tuples with the same keys

The union of two named tuples with the same keys (regardless of their order) results in a named tuple type that has the union of the types in each key. The order of the keys will be the ones from the tuple on the left hand side.

For example:

```
t1 = {x: 1, y: "hi"} # Tuple(x: Int32, y: String)
t2 = {y: true, x: nil} # Tuple(y: Bool, x: Nil)

t3 = rand < 0.5 ? t1 : t2
typeof(t3) # NamedTuple(x: Int32 | Nil, y: String | Bool)
```

# Overloading

We can define a `become_older` method that accepts a number indicating the years to grow:

```
class Person
  getter :age

  def initialize(@name : String, @age : Int = 0)
    end

  def become_older
    @age += 1
  end

  def become_older(years)
    @age += years
  end
end

john = Person.new "John"
john.age # => 0

john.become_older
john.age # => 1

john.become_older 5
john.age # => 6
```

That is, you can have different methods with the same name and different number of arguments and they will be considered as separate methods. This is called *method overloading*.

Methods overload by several criteria:

- The number of arguments
- The type restrictions applied to arguments
- The names of required named arguments
- Whether the method accepts a [block](#) or not

For example, we can define four different `become_older` methods:

```

class Person
  @age = 0

  # Increases age by one
  def become_older
    @age += 1
  end

  # Increases age by the given number of years
  def become_older(years : Int32)
    @age += years
  end

  # Increases age by the given number of years, as a String
  def become_older(years : String)
    @age += years.to_i
  end

  # Yields the current age of this person and increases
  # its age by the value returned by the block
  def become_older
    @age += yield @age
  end
end

person = Person.new "John"

person.become_older
person.age # => 1

person.become_older 5
person.age # => 6

person.become_older "12"
person.age # => 18

person.become_older do |current_age|
  current_age < 20 ? 10 : 30
end
person.age # => 28

```

Note that in the case of the method that yields, the compiler figured this out because there's a `yield` expression. To make this more explicit, you can add a dummy `&block` argument at the end:

```

class Person
  @age = 0

  def become_older(&block)
    @age += yield @age
  end
end

```

In generated documentation the dummy `&block` method will always appear, regardless of you writing it or not.

Given the same number of arguments, the compiler will try to sort them by leaving the less restrictive ones to the end:

```
class Person
  @age = 0

  # First, this method is defined
  def become_older(age)
    @age += age
  end

  # Since "String" is more restrictive than no restriction
  # at all, the compiler puts this method before the previous
  # one when considering which overload matches.
  def become_older(age : String)
    @age += age.to_i
  end
end

person = Person.new "John"

# Invokes the first definition
person.become_older 20

# Invokes the second definition
person.become_older "12"
```

However, the compiler cannot always figure out the order because there isn't always a total ordering, so it's always better to put less restrictive methods at the end.

## Default values

A method can specify default values for the last arguments:

```
class Person
  def become_older(by = 1)
    @age += by
  end
end

john = Person.new "John"
john.age # => 0

john.become_older
john.age # => 1

john.become_older 2
john.age # => 3
```

## Named arguments

All arguments can also be specified, in addition to their position, by their name.

For example:

```
john.become_older by: 5
```

When there are many arguments, the order of the names in the invocation doesn't matter, as long as all required arguments are covered:

```
def some_method(x, y = 1, z = 2, w = 3)
  # do something...
end

some_method 10 # x: 10, y: 1, z: 2, w: 3
some_method 10, z: 10 # x: 10, y: 1, z: 10, w: 3
some_method 10, w: 1, y: 2, z: 3 # x: 10, y: 2, z: 3, w: 1
some_method y: 10, x: 20 # x: 20, y: 10, z: 2, w: 3

some_method y: 10 # Error, missing argument: x
```

When a method specifies a splat (explained in the next section), named arguments can't be used. The reason is that understanding how arguments are matched becomes very difficult; positional arguments are easier to reason about in this case.



## Splats and tuples

A method can receive a variable number of arguments by using a *splat* (`*`), which can appear only once and in any position:

```
def sum(*elements)
  total = 0
  elements.each do |value|
    total += value
  end
  total
end

sum 1, 2, 3      # => 6
sum 1, 2, 3, 4.5 # => 10.5
```

The passed arguments become a [Tuple](#) in the method's body:

```
# elements is Tuple(Int32, Int32, Int32)
sum 1, 2, 3

# elements is Tuple(Int32, Int32, Int32, Float64)
sum 1, 2, 3, 4.5
```

Arguments past the splat argument can only be passed as named arguments:

```
def sum(*elements, initial = 0)
  total = initial
  elements.each do |value|
    total += value
  end
  total
end

sum 1, 2, 3      # => 6
sum 1, 2, 3, initial: 10 # => 16
```

Arguments past the splat method without a default value are required named arguments:

```
def sum(*elements, initial)
  total = initial
  elements.each do |value|
    total += value
  end
  total
end

sum 1, 2, 3      # Error, missing argument: initial
sum 1, 2, 3, initial: 10 # => 16
```

Two methods with different required named arguments overload between each other:

```
def foo(*elements, x)
  1
end

def foo(*elements, y)
  2
end

foo x: "something" # => 1
foo y: "something" # => 2
```

The splat argument can also be left unnamed, with the meaning "after this, named arguments follow":

```
def foo(x, y, *, z)
  end

foo 1, 2, 3 # Error, wrong number of arguments (given 3, expected 2)
foo 1, 2 # Error, missing argument: z
foo 1, 2, z: 3 # OK
```

## Splatting a tuple

A `Tuple` can be splat into a method call by using `*`:

```
def foo(x, y)
  x + y
end

tuple = {1, 2}
foo *tuple # => 3
```

## Double splats and named tuples

A double splat (`**`) captures named arguments that were not matched by other arguments. The type of the argument is a `NamedTuple`:

```
def foo(x, **other)
  # Return the captured named arguments as a NamedTuple
  other
end

foo 1, y: 2, z: 3 # => {y: 2, z: 3}
foo y: 2, x: 1, z: 3 # => {y: 2, z: 3}
```

## Double splatting a named tuple

A `NamedTuple` can be splat into a method call by using `**`:

Nil

```
def foo(x, y)
  x - y
end

tuple = {y: 3, x: 10}
foo **tuple # => 7
```

## Type restrictions

Type restrictions are applied to method arguments to restrict the types accepted by that method.

```
def add(x : Number, y : Number)
  x + y
end

# Ok
add 1, 2

# Error: no overload matches 'add' with types Bool, Bool
add true, false
```

Note that if we had defined `add` without type restrictions, we would also have gotten a compile time error:

```
def add(x, y)
  x + y
end

add true, false
```

The above code gives this compile error:

```
Error in foo.cr:6: instantiating 'add(Bool, Bool)'

add true, false
^~~

in foo.cr:2: undefined method '+' for Bool

  x + y
  ^
```

This is because when you invoke `add`, it is instantiated with the types of the arguments: every method invocation with a different type combination results in a different method instantiation.

The only difference is that the first error message is a little more clear, but both definitions are safe in that you will get a compile time error anyway. So, in general, it's preferable not to specify type restrictions and almost only use them to define different method overloads. This results in more generic, reusable code. For example, if we define a class that has a `+` method but isn't a `Number`, we can use the `add` method that doesn't have type restrictions, but we can't use the `add` method that has restrictions.

```

# A class that has a + method but isn't a Number
class Six
  def +(other)
    6 + other
  end
end

# add method without type restrictions
def add(x, y)
  x + y
end

# OK
add Six.new, 10

# add method with type restrictions
def restricted_add(x : Number, y : Number)
  x + y
end

# Error: no overload matches 'restricted_add' with types Six, Int32
restricted_add Six.new, 10

```

Refer to the [type grammar](#) for the notation used in type restrictions.

Note that type restrictions do not apply to the variables inside the actual methods.

```

def handle_path(path : String)
  path = Path.new(path) # *path* is now of the type Path
  # Do something with *path*
end

```

## self restriction

A special type restriction is `self` :

```

class Person
  def ==(other : self)
    other.name == name
  end

  def ==(other)
    false
  end
end

john = Person.new "John"
another_john = Person.new "John"
peter = Person.new "Peter"

john == another_john # => true
john == peter       # => false (names differ)
john == 1            # => false (because 1 is not a Person)

```

In the previous example `self` is the same as writing `Person` . But, in general, `self` is the same as writing the type that will finally own that method, which, when modules are involved, becomes more useful.

As a side note, since `Person` inherits `Reference` the second definition of `==` is not needed, since it's already defined in `Reference`.

Note that `self` always represents a match against an instance type, even in class methods:

```
class Person
  getter name : String

  def initialize(@name)
    end

  def self.compare(p1 : self, p2 : self)
    p1.name == p2.name
  end
end

john = Person.new "John"
peter = Person.new "Peter"

Person.compare(john, peter) # OK
```

You can use `self.class` to restrict to the `Person` type. The next section talks about the `.class` suffix in type restrictions.

## Classes as restrictions

Using, for example, `Int32` as a type restriction makes the method only accept instances of `Int32`:

```
def foo(x : Int32)
  end

foo 1 # OK
foo "hello" # Error
```

If you want a method to only accept the type `Int32` (not instances of it), you use

`.class`:

```
def foo(x : Int32.class)
  end

foo Int32 # OK
foo String # Error
```

The above is useful for providing overloads based on types, not instances:

```
def foo(x : Int32.class)
  puts "Got Int32"
end

def foo(x : String.class)
  puts "Got String"
end

foo Int32 # prints "Got Int32"
foo String # prints "Got String"
```

## Type restrictions in splats

You can specify type restrictions in splats:

```
def foo(*args : Int32)
  end

def foo(*args : String)
  end

foo 1, 2, 3 # OK, invokes first overload
foo "a", "b", "c" # OK, invokes second overload
foo 1, 2, "hello" # Error
foo() # Error
```

When specifying a type, all elements in a tuple must match that type. Additionally, the empty-tuple doesn't match any of the above cases. If you want to support the empty-tuple case, add another overload:

```
def foo
  # This is the empty-tuple case
end
```

A simple way to match against one or more elements of any type is to use `Object` as a restriction:

```
def foo(*args : Object)
  end

foo() # Error
foo(1) # OK
foo(1, "x") # OK
```

## Free variables

You can make a type restriction take the type of an argument, or part of the type of an argument, using `forall` :

```
def foo(x : T) forall T
  T
end

foo(1)      # => Int32
foo("hello") # => String
```

That is, `T` becomes the type that was effectively used to instantiate the method.

A free variable can be used to extract the type parameter of a generic type within a type restriction:

```
def foo(x : Array(T)) forall T
  T
end

foo([1, 2]) # => Int32
foo([1, "a"]) # => (Int32 | String)
```

To create a method that accepts a type name, rather than an instance of a type, append `.class` to a free variable in the type restriction:

```
def foo(x : T.class) forall T
  Array(T)
end

foo(Int32) # => Array(Int32)
foo(String) # => Array(String)
```

Multiple free variables can be specified too, for matching types of multiple arguments:

```
def push(element : T, array : Array(T)) forall T
  array << element
end

push(4, [1, 2, 3]) # OK
push("oops", [1, 2, 3]) # Error
```



## Return types

A method's return type is always inferred by the compiler. However, you might want to specify it for two reasons:

1. To make sure that the method returns the type that you want
2. To make it appear in documentation comments

For example:

```
def some_method : String
  "hello"
end
```

The return type follows the [type grammar](#).

## Nil return type

Marking a method as returning `Nil` will make it return `nil` regardless of what it actually returns:

```
def some_method : Nil
  1 + 2
end

some_method # => nil
```

This is useful for two reasons:

1. Making sure a method returns `nil` without needing to add an extra `nil` at the end, or at every return point
2. Documenting that the method's return value is of no interest

These methods usually imply a side effect.

Using `void` is the same, but `nil` is more idiomatic: `void` is preferred in C bindings.

## NoReturn return type

Some expressions won't return to the current scope and therefore have no return type. This is expressed as the special return type `NoReturn`.

Typical examples for non-returning methods and keywords are `return`, `exit`, `raise`, `next`, and `break`.

This is for example useful for deconstructing union types:

```
string = STDIN.gets
typeof(string) # => String?
typeof(raise "Empty input") # => NoReturn
typeof(string || raise "Empty input") # => String
```

The compiler recognizes that in case `string` is `Nil`, the right hand side of the expression `string || raise` will be evaluated. Since `typeof(raise "Empty input")` is `NoReturn` the execution would not return to the current scope in that case. That leaves only `String` as resulting type of the expression.

Every expression whose code paths all result in `NoReturn` will be `NoReturn` as well. `NoReturn` does not show up in a union type because it would essentially be included in every expression's type. It is only used when an expression will never return to the current scope.

`NoReturn` can be explicitly set as return type of a method or function definition but will usually be inferred by the compiler.

## Method arguments

This is the formal specification of method and call arguments.

## Components of a method definition

A method definition consists of:

- required and optional positional arguments
- an optional splat argument, whose name can be empty
- required and optional named arguments
- an optional double splat argument

For example:

```
def foo(
  # These are positional arguments:
  x, y, z = 1,
  # This is the splat argument:
  *args,
  # These are the named arguments:
  a, b, c = 2,
  # This is the double splat argument:
  **options
)
```

Each one of them is optional, so a method can do without the double splat, without the splat, without keyword arguments and without positional arguments.

## Components of a method call

A method call also has some parts:

```
foo(
  # These are positional arguments
  1, 2,
  # These are named arguments
  a: 1, b: 2
)
```

Additionally, a call argument can have a splat ( \* ) or double splat ( \*\* ). A splat expands a [Tuple](#) into positional arguments, while a double splat expands a [NamedTuple](#) into named arguments. Multiple argument splats and double splats are allowed.

## How call arguments are matched to method arguments

When invoking a method, the algorithm to match call arguments to method arguments is:

- First positional arguments are matched with positional method arguments. The number of these must be at least the number of positional arguments without a default value. If there's a splat method argument with a name (the case without a name is explained below), more positional arguments are allowed and they are captured as a tuple. Positional arguments never match past the splat method argument.
- Then named arguments are matched, by name, with any argument in the method (it can be before or after the splat method argument). If an argument was already filled by a positional argument then it's an error.
- Extra named arguments are placed in the double splat method argument, as a `NamedTuple`, if it exists, otherwise it's an error.

When a splat method argument has no name, it means no more positional arguments can be passed, and next arguments must be passed as named arguments. For example:

```
# Only one positional argument allowed, y must be passed as a named argument
def foo(x, *, y)
end

foo 1          # Error, missing argument: y
foo 1, 2       # Error: wrong number of arguments (given 2, expected 1)
foo 1, y: 10   # OK
```

But even if a splat method argument has a name, arguments that follow it must be passed as named arguments:

```
# One or more positional argument allowed, y must be passed as a named argument
def foo(x, *args, y)
end

foo 1          # Error, missing argument: y
foo 1, 2       # Error: missing argument; y
foo 1, 2, 3    # Error: missing argument: y
foo 1, y: 10   # OK
foo 1, 2, 3, y: 4 # OK
```

There's also the possibility of making a method only receive named arguments (and list them), by placing the star at the beginning:

```
# A method with two required named arguments: x and y
def foo(*, x, y)
end

foo          # Error: missing arguments: x, y
foo x: 1     # Error: missing argument: y
foo x: 1, y: 2 # OK
```

Arguments past the star can also have default values. It means: they must be passed as named arguments, but they aren't required (so: optional named arguments):

```
# A method with two required named arguments: x and y
def foo(*, x, y = 2)
  end

foo          # Error: missing argument: x
foo x: 1     # OK, y is 2
foo x: 1, y: 3 # OK, y is 3
```

Because arguments (without a default value) after the splat method argument must be passed by name, two methods with different required named arguments overload:

```
def foo(*, x)
  puts "Passed with x: #{x}"
end

def foo(*, y)
  puts "Passed with y: #{y}"
end

foo x: 1 # => Passed with x: 1
foo y: 2 # => Passed with y: 2
```

Positional arguments can always be matched by name:

```
def foo(x, *, y)
  end

foo 1, y: 2 # OK
foo y: 2, x: 3 # OK
```

## External names

An external name can be specified for a method argument. The external name is the one used when passing an argument as a named argument, and the internal name is the one used inside the method definition:

```
def foo(external_name internal_name)
  # here we use internal_name
end

foo external_name: 1
```

This covers two uses cases.

The first use case is using keywords as named arguments:

```
def plan(begin begin_time, end end_time)
  puts "Planning between #{begin_time} and #{end_time}"
end

plan begin: Time.now, end: 2.days.from_now
```

Nil

The second use case is making a method argument more readable inside a method body:

```
def increment(value, by)
  # OK, but reads odd
  value + by
end

def increment(value, by amount)
  # Better
  value + amount
end
```

## Operators

Crystal supports a number of operators, with one, two or three operands.

Operator expressions are actually parsed as method calls. For example `a + b` is semantically equivalent to `a.+(b)`, a call to method `+` on `a` with argument `b`.

There are however some special rules regarding operator syntax:

- The dot ( `.` ) usually put between receiver and method name (i.e. the *operator*) can be omitted.
- Chained sequences of operator calls are restructured by the compiler in order to implement [operator precedence](#). Enforcing operator precedence makes sure that an expression such as `1 * 2 + 3 * 4` is parsed as `(1 * 2) + (2 * 3)` to honour regular math rules.
- Regular method names must start with a letter or underscore, but operators only consist of special characters. Any method not starting with a letter or underscore is an operator method.
- Available operators are whitelisted in the compiler (see [List of Operators](#) below) which allows symbol-only method names and treats them as operators, including their precedence rules.

Operators are implemented like any regular method, and the standard library offers many implementations, for example for math expressions.

## Defining operator methods

Most operators can be implemented as regular methods.

One can assign any meaning to the operators, but it is advisable to stay within similar semantics to the generic operator meaning to avoid cryptic code that is confusing and behaves unexpectedly.

A few operators are defined directly by the compiler and cannot be redefined in user code. Examples for this are the inversion operator `!`, the assignment operator `=`, [combined assignment operators](#) such as `||=` and [range operators](#). Whether a method can be redefined is indicated by the column *Overloadable* in the below operator tables.

### Unary operators

Unary operators are written in prefix notation and have only a single operand. Thus, a method implementation receives no arguments and only operates on `self`.

The following example demonstrates the `Vector2` type as a two-dimensional vector with a unary operator method `-` for vector inversion.

```

struct Vector2
  getter x, y

  def initialize(@x : Int32, @y : Int32)
    end

  # Unary operator. Returns the inverted vector to `self`.
  def - : self
    Vector2.new(-x, -y)
  end
end

v1 = Vector2.new(1, 2)
-v1 # => Vector2(@x=-1, @y=-2)

```

## Binary operators

Binary operators have two operands. Thus, a method implementation receives exactly one argument representing the second operand. The first operand is the receiver `self`.

The following example demonstrates the `Vector2` type as a two-dimensional vector with a binary operator method `+` for vector addition.

```

struct Vector2
  getter x, y

  def initialize(@x : Int32, @y : Int32)
    end

  # Binary operator. Returns *other* added to `self`.
  def +(other : self) : self
    Vector2.new(x + other.x, y + other.y)
  end
end

v1 = Vector2.new(1, 2)
v2 = Vector2.new(3, 4)
v1 + v2 # => Vector2(@x=4, @y=6)

```

Per convention, the return type of a binary operator should be the type of the first operand (the receiver), so that `typeof(a <op> b) == typeof(a)`. Otherwise the assignment operator (`a <op>= b`) would unintentionally change the type of `a`. There can be reasonable exceptions though. For example in the standard library the float division operator `/` on integer types always returns `Float64`, because the quotient must not be limited to the value range of integers.

## Ternary operators

The [conditional operator](#) (`? :`) is the only ternary operator. It is not parsed as a method, and its meaning cannot be changed. The compiler transforms it to an `if` expression.



## Operator Precedence

This list is sorted by precedence, so upper entries bind stronger than lower ones.

Category	Operators
Index accessors	<code>[]</code> , <code>[]?</code>
Unary	<code>+</code> , <code>&amp;+</code> , <code>-</code> , <code>&amp;-</code> , <code>!</code> , <code>~</code> , <code>*</code> , <code>**</code>
Exponential	<code>**</code> , <code>&amp;**</code>
Multiplicative	<code>*</code> , <code>&amp;*</code> , <code>/</code> , <code>//</code> , <code>%</code>
Additive	<code>+</code> , <code>&amp;+</code> , <code>-</code> , <code>&amp;-</code>
Shift	<code>&lt;&lt;</code> , <code>&gt;&gt;</code>
Binary AND	<code>&amp;</code>
Binary OR/XOR	<code> </code> , <code>^</code>
Equality	<code>==</code> , <code>!=</code> , <code>~=</code> , <code>!~</code> , <code>===</code>
Comparison	<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> , <code>&lt;=&gt;</code>
Logical AND	<code>&amp;&amp;</code>
Logical OR	<code>  </code>
Range	<code>..</code> , <code>...</code>
Conditional	<code>?:</code>
Assignment	<code>=</code> , <code>[]=</code> , <code>+=</code> , <code>&amp;+=</code> , <code>-=</code> , <code>&amp;-=</code> , <code>*=</code> , <code>&amp;*=</code> , <code>/=</code> , <code>//=</code> , <code>%=</code> , <code> =</code> , <code>&amp; =</code> , <code>^=</code> , <code>**=</code> , <code>&lt;&lt;=</code> , <code>&gt;&gt;=</code> , <code>  =</code> , <code>&amp;&amp;=</code>

## List of operators

### Arithmetic operators

#### Unary

Operator	Description	Example	Overloadable
<code>+</code>	positive	<code>+1</code>	yes
<code>&amp;+</code>	wrapping positive	<code>&amp;+1</code>	yes
<code>-</code>	negative	<code>-1</code>	yes
<code>&amp;-</code>	wrapping negative	<code>&amp;-1</code>	yes

#### Multiplicative

Operator	Description	Example	Overloadable
<code>**</code>	exponentiation	<code>1 ** 2</code>	yes
<code>&amp;**</code>	wrapping exponentiation	<code>1 &amp;** 2</code>	yes
<code>*</code>	multiplication	<code>1 * 2</code>	yes
<code>&amp;*</code>	wrapping multiplication	<code>1 &amp;* 2</code>	yes
<code>/</code>	division	<code>1 / 2</code>	yes
<code>//</code>	floor division	<code>1 // 2</code>	yes
<code>%</code>	modulus	<code>1 % 2</code>	yes

## Additive

Operator	Description	Example	Overloadable
<code>+</code>	addition	<code>1 + 2</code>	yes
<code>&amp;+</code>	wrapping addition	<code>1 &amp;+ 2</code>	yes
<code>-</code>	subtraction	<code>1 - 2</code>	yes
<code>&amp;-</code>	wrapping subtraction	<code>1 &amp;- 2</code>	yes

## Other unary operators

Operator	Description	Example	Overloadable
<code>!</code>	inversion	<code>!true</code>	no
<code>~</code>	binary complement	<code>~1</code>	yes

## Shifts

Operator	Description	Example	Overloadable
<code>&lt;&lt;</code>	shift left, append	<code>1 &lt;&lt; 2</code> , <code>STDOUT &lt;&lt; "foo"</code>	yes
<code>&gt;&gt;</code>	shift right	<code>1 &gt;&gt; 2</code>	yes

## Binary

Operator	Description	Example	Overloadable
<code>&amp;</code>	binary AND	<code>1 &amp; 2</code>	yes
<code> </code>	binary OR	<code>1   2</code>	yes
<code>^</code>	binary XOR	<code>1 ^ 2</code>	yes

## Equality

Three base operators test equality:

- `==` : Checks whether the values of the operands are equal
- `=~`  : Checks whether the value of the first operand matches the value of the second operand with pattern matching.
- `===`  : Checks whether the left hand operand matches the right hand operand in [case equality](#). This operator is applied in `case ... when` conditions.

The first two operators also have inversion operators ( `!=` and `!~` ) whose semantical intention is just the inverse of the base operator: `a != b` is supposed to be equivalent to `!(a == b)` and `a !~ b` to `!(a =~ b)` . Nevertheless, these inversions can be defined with a custom implementation. This can be useful for example to improve performance (non-equality can often be proven faster than equality).

Operator	Description	Example	Overloadable
<code>==</code>	equals	<code>1 == 2</code>	yes
<code>!=</code>	not equals	<code>1 != 2</code>	yes
<code> =~ </code>	pattern match	<code>"foo" =~ /fo/</code>	yes
<code>!~</code>	no pattern match	<code>"foo" !~ /fo/</code>	yes
<code> === </code>	<a href="#">case equality</a>	<code>/foo/ === "foo"</code>	yes

## Comparison

Operator	Description	Example	Overloadable
<code>&lt;</code>	less	<code>1 &lt; 2</code>	yes
<code>&lt;=</code>	less or equal	<code>1 &lt;= 2</code>	yes
<code>&gt;</code>	greater	<code>1 &gt; 2</code>	yes
<code>&gt;=</code>	greater or equal	<code>1 &gt;= 2</code>	yes
<code>&lt;=&gt;</code>	comparison	<code>1 &lt;=&gt; 2</code>	yes

## Logical

Operator	Description	Example	Overloadable
<code>&amp;&amp;</code>	<a href="#">logical AND</a>	<code>true &amp;&amp; false</code>	no
<code>  </code>	<a href="#">logical OR</a>	<code>true    false</code>	no

## Range

The range operators are used in [Range](#) literals.

Operator	Description	Example	Overloadable
<code>..</code>	range	<code>1..10</code>	no
<code>...</code>	exclusive range	<code>1...10</code>	no

## Splats

Splat operators can only be used for destructuring tuples in method arguments. See [Splats and Tuples](#) for details.

Operator	Description	Example	Overloadable
*	splat	*foo	no
**	double splat	**foo	no

## Conditional

The [conditional operator](#) (`? :`) is internally rewritten to an `if` expression by the compiler.

Operator	Description	Example	Overloadable
? :	conditional	a == b ? c : d	no

## Assignments

The assignment operator `=` assigns the value of the second operand to the first operand. The first operand is either a variable (in this case the operator can't be redefined) or a call (in this case the operator can be redefined). See [assignment](#) for details.

Operator	Description	Example	Overloadable
=	variable assignment	a = 1	no
=	call assignment	a.b = 1	yes
[]=	index assignment	a[0] = 1	yes

## Combined assignments

The assignment operator `=` is the basis for all operators that combine an operator with assignment. The general form is `a <op>= b` and the compiler transform that into `a = a <op> b`.

Exceptions to the general expansion formula are the logical operators:

- `a ||= b` transforms to `a || (a = b)`
- `a &&= b` transforms to `a && (a = b)`

There is another special case when `a` is an index accessor (`[]`), it is changed to the nilable variant (`[]?` on the right hand side):

- `a[i] ||= b` transforms to `a[i] = (a[i]? || b)`
- `a[i] &&= b` transforms to `a[i] = (a[i]? && b)`

All transformations assume the receiver (`a`) is a variable. If it is a call, the replacements are semantically equivalent but the implementation is a bit more complex (introducing an anonymous temporary variable) and expects `a=` to be

callable.

The receiver can't be anything else than a variable or call.

Operator	Description	Example	Overloadable
<code>+=</code>	addition <i>and</i> assignment	<code>i += 1</code>	no
<code>&amp;+=</code>	wrapping addition <i>and</i> assignment	<code>i &amp;+= 1</code>	no
<code>-=</code>	subtraction <i>and</i> assignment	<code>i -= 1</code>	no
<code>&amp;-=</code>	wrapping subtraction <i>and</i> assignment	<code>i &amp;-= 1</code>	no
<code>*=</code>	multiplication <i>and</i> assignment	<code>i *= 1</code>	no
<code>&amp;*=</code>	wrapping multiplication <i>and</i> assignment	<code>i &amp;*= 1</code>	no
<code>/=</code>	division <i>and</i> assignment	<code>i /= 1</code>	no
<code>//=</code>	floor division <i>and</i> assignment	<code>i //= 1</code>	no
<code>%=</code>	modulo <i>and</i> assignment	<code>i %= 1</code>	yes
<code> =</code>	binary or <i>and</i> assignment	<code>i  = 1</code>	no
<code>&amp;=</code>	binary and <i>and</i> assignment	<code>i &amp;= 1</code>	no
<code>^=</code>	binary xor <i>and</i> assignment	<code>i ^= 1</code>	no
<code>**=</code>	exponential <i>and</i> assignment	<code>i **= 1</code>	no
<code>&lt;&lt;=</code>	left shift <i>and</i> assignment	<code>i &lt;&lt;= 1</code>	no
<code>&gt;&gt;=</code>	right shift <i>and</i> assignment	<code>i &gt;&gt;= 1</code>	no
<code>  =</code>	logical or <i>and</i> assignment	<code>i   = true</code>	no
<code>&amp;&amp;=</code>	logical and <i>and</i> assignment	<code>i &amp;&amp;= true</code>	no

## Index Accessors

Index accessors are used to query a value by index or key, for example an array item or map entry. The nilable variant `[]?` is supposed to return `nil` when the index is not found, while the non-nilable variant raises in that case.

Implementations in the standard-library usually raise `KeyError` or `IndexError`.

Operator	Description	Example	Overloadable
<code>[]</code>	index accessor	<code>ary[i]</code>	yes
<code>[]?</code>	nilable index accessor	<code>ary[i]?</code>	yes

## Visibility

Methods are public by default: the compiler will always let you invoke them. There is no `public` keyword for this reason.

Methods *can* be marked as `private` or `protected`.

## Private methods

A `private` method can only be invoked without a receiver, that is, without something before the dot. The only exception is `self` as a receiver:

```
class Person
  private def say(message)
    puts message
  end

  def say_hello
    say "hello" # OK, no receiver
    self.say "hello" # OK, self is a receiver, but it's allowed.

    other = Person.new "Other"
    other.say "hello" # Error, other is a receiver
  end
end
```

Note that `private` methods are visible by subclasses:

```
class Employee < Person
  def say_bye
    say "bye" # OK
  end
end
```

## Private types

Private types can only be referenced inside the namespace where they are defined, and never be fully qualified.

```
class Foo
  private class Bar
  end

  Bar # OK
  Foo::Bar # Error
end

Foo::Bar # Error
```

`private` can be used with `class`, `module`, `lib`, `enum`, `alias` and constants:

```
class Foo
  private ONE = 1

  ONE # => 1
end

Foo::ONE # Error
```

## Protected methods

A `protected` method can only be invoked on:

1. instances of the same type as the current type
2. instances in the same namespace (class, struct, module, etc.) as the current type

```
# Example of 1

class Person
  protected def say(message)
    puts message
  end

  def say_hello
    say "hello" # OK, implicit self is a Person
    self.say "hello" # OK, self is a Person

    other = Person.new "Other"
    other.say "hello" # OK, other is a Person
  end
end

class Animal
  def make_a_person_talk
    person = Person.new
    person.say "hello" # Error: person is a Person but current type is an Animal
  end
end

one_more = Person.new "One more"
one_more.say "hello" # Error: one_more is a Person but current type is the Program

# Example of 2

module Namespace
  class Foo
    protected def foo
      puts "Hello"
    end
  end

  class Bar
    def bar
      # Works, because Foo and Bar are under Namespace
      Foo.new.foo
    end
  end
end

Namespace::Bar.new.bar
```

A `protected` method can only be invoked from the scope of its class or its descendants. That includes the class scope and bodies of class methods and instance methods of the same type the protected method is defined on, as well as all types including or inheriting that type and all types in that namespace.

```
class Parent
  protected def self.protected_method
  end

  Parent.protected_method # OK

  def instance_method
    Parent.protected_method # OK
  end

  def self.class_method
    Parent.protected_method # OK
  end
end

class Child < Parent
  Parent.protected_method # OK

  def instance_method
    Parent.protected_method # OK
  end

  def self.class_method
    Parent.protected_method # OK
  end
end

class Parent::Sub
  Parent.protected_method # OK

  def instance_method
    Parent.protected_method # OK
  end

  def self.class_method
    Parent.protected_method # OK
  end
end
```

## Private top-level methods

A `private` top-level method is only visible in the current file.

```
# In file one.cr
private def greet
  puts "Hello"
end

greet # => "Hello"

# In file two.cr
require "./one"

greet # undefined local variable or method 'greet'
```



This allows you to define helper methods in a file that will only be known in that file.

## Private top-level types

A `private` top-level type is only visible in the current file.

```
# In file one.cr
private class Greeter
  def self.greet
    "Hello"
  end
end

Greeter.greet # => "Hello"

# In file two.cr
require "./one"

Greeter.greet # undefined constant 'Greeter'
```

## Inheritance

Every class except `Object`, the hierarchy root, inherits from another class (its superclass). If you don't specify one it defaults to `Reference` for classes and `Struct` for structs.

A class inherits all instance variables and all instance and class methods of a superclass, including its constructors (`new` and `initialize`).

```
class Person
  def initialize(@name : String)
    end

  def greet
    puts "Hi, I'm #{@name}"
  end
end

class Employee < Person
  end

employee = Employee.new "John"
employee.greet # "Hi, I'm John"
```

If a class defines a `new` or `initialize` then its superclass constructors are not inherited:

```
class Person
  def initialize(@name : String)
    end
end

class Employee < Person
  def initialize(@name : String, @company_name : String)
    end
end

Employee.new "John", "Acme" # OK
Employee.new "Peter" # Error: wrong number of arguments for 'Employee:Class#new'
```

You can override methods in a derived class:

```

class Person
  def greet(msg)
    puts "Hi, #{msg}"
  end
end

class Employee < Person
  def greet(msg)
    puts "Hello, #{msg}"
  end
end

p = Person.new
p.greet "everyone" # "Hi, everyone"

e = Employee.new
e.greet "everyone" # "Hello, everyone"

```

Instead of overriding you can define specialized methods by using type restrictions:

```

class Person
  def greet(msg)
    puts "Hi, #{msg}"
  end
end

class Employee < Person
  def greet(msg : Int32)
    puts "Hi, this is a number: #{msg}"
  end
end

e = Employee.new
e.greet "everyone" # "Hi, everyone"

e.greet 1 # "Hi, this is a number: 1"

```

## super

You can invoke a superclass' method using `super` :

```

class Person
  def greet(msg)
    puts "Hello, #{msg}"
  end
end

class Employee < Person
  def greet(msg)
    super # Same as: super(msg)
    super("another message")
  end
end

```

Without arguments or parentheses, `super` receives the same arguments as the method's arguments. Otherwise, it receives the arguments you pass to it.

## Covariance and Contravariance

One place inheritance can get a little tricky is with arrays. We have to be careful when declaring an array of objects where inheritance is used. For example, consider the following

```
class Foo
end

class Bar < Foo
end

foo_arr = [Bar.new] of Foo # => [#<Bar:0x10215bfe0>] : Array(Foo)
bar_arr = [Bar.new]      # => [#<Bar:0x10215bfd0>] : Array(Bar)
bar_arr2 = [Foo.new] of Bar # compiler error
```

A Foo array can hold both Foo's and Bar's, but an array of Bar can only hold Bar and its subclasses.

One place this might trip you up is when automatic casting comes into play. For example, the following won't work:

```
class Foo
end

class Bar < Foo
end

class Test
  @arr : Array(Foo)

  def initialize
    @arr = [Bar.new]
  end
end
```

we've declared `@arr` as type `Array(Foo)` so we may be tempted to think that we can start putting `Bar`s in there. Not quite. In the `initialize`, the type of the `[Bar.new]` expression is `Array(Bar)`, period. And `Array(Bar)` is *not* assignable to an `Array(Foo)` instance var.

What's the right way to do this? Change the expression so that it *is* of the right type: `Array(Foo)` (see example above).

```
class Foo
end

class Bar < Foo
end

class Test
  @arr : Array(Foo)

  def initialize
    @arr = [Bar.new] of Foo
  end
end
```

Nil

This is just one type (Array) and one operation (assignment), the logic of the above will be applied differently for other types and assignments, in general [Covariance and Contravariance](#) is not fully supported.

## Virtual and abstract types

When a variable's type combines different types under the same class hierarchy, its type becomes a **virtual type**. This applies to every class and struct except for

`Reference`, `Value`, `Int` and `Float`. An example:

```
class Animal
end

class Dog < Animal
  def talk
    "Woof!"
  end
end

class Cat < Animal
  def talk
    "Miau"
  end
end

class Person
  getter pet

  def initialize(@name : String, @pet : Animal)
  end
end

john = Person.new "John", Dog.new
peter = Person.new "Peter", Cat.new
```

If you compile the above program with the `tool hierarchy` command you will see this for `Person`:

```
- class Object
  |
+- class Reference
  |
+- class Person
    @name : String
    @pet : Animal+
```

You can see that `@pet` is `Animal+`. The `+` means it's a virtual type, meaning "any class that inherits from `Animal`, including `Animal`".

The compiler will always resolve a type union to a virtual type if they are under the same hierarchy:

```
if some_condition
  pet = Dog.new
else
  pet = Cat.new
end

# pet : Animal+
```

The compiler will always do this for classes and structs under the same hierarchy: it will find the first superclass from which all types inherit from (excluding `Reference`, `Value`, `Int` and `Float`). If it can't find one, the type union remains.

The real reason the compiler does this is to be able to compile programs faster by not creating all kinds of different similar unions, also making the generated code smaller in size. But, on the other hand, it makes sense: classes under the same hierarchy should behave in a similar way.

Lets make John's pet talk:

```
john.pet.talk # Error: undefined method 'talk' for Animal
```

We get an error because the compiler now treats `@pet` as an `Animal+`, which includes `Animal`. And since it can't find a `talk` method on it, it errors.

What the compiler doesn't know is that for us, `Animal` will never be instantiated as it doesn't make sense to instantiate one. We have a way to tell the compiler so by marking the class as `abstract`:

```
abstract class Animal
end
```

Now the code compiles:

```
john.pet.talk # => "Woof!"
```

Marking a class as `abstract` will also prevent us from creating an instance of it:

```
Animal.new # Error: can't instantiate abstract class Animal
```

To make it more explicit that an `Animal` must define a `talk` method, we can add it to `Animal` as an abstract method:

```
abstract class Animal
  # Makes this animal talk
  abstract def talk
end
```

By marking a method as `abstract` the compiler will check that all subclasses implement this method, even if a program doesn't use them.

Abstract methods can also be defined in modules, and the compiler will check that including types implement them.

## Class methods

Class methods are methods associated to a class or module instead of a specific instance.

```
module CaesarCipher
  def self.encrypt(string : String)
    string.chars.map { |char| ((char.upcase.ord - 52) % 26 + 65).chr }.join
  end
end

CaesarCipher.encrypt("HELLO") # => "URYYB"
```

Class methods are defined by prefixing the method name with the type name and a period.

```
def CaesarCipher.decrypt(string : String)
  encrypt(string)
end
```

When they're defined inside a class or module scope it is easier to use `self` instead of the class name.

Class methods can also be defined by [extending a Module](#).

A class method can be called under the same name as it was defined (`CaesarCipher.decrypt("HELLO")`). When called from within the same class or module scope the receiver can be `self` or implicit (like `encrypt(string)`).

## Constructors

Constructors are normal class methods which [create a new instance of the class](#). By default all classes in Crystal have at least one constructor called `new`, but they may also define other constructors with different names.



## Class variables

Class variables are associated to classes instead of instances. They are prefixed with two "at" signs ( @@ ). For example:

```
class Counter
  @@instances = 0

  def initialize
    @@instances += 1
  end

  def self.instances
    @@instances
  end
end

Counter.instances # => 0
Counter.new
Counter.new
Counter.new
Counter.instances # => 3
```

Class variables can be read and written from class methods or instance methods.

Their type is inferred using the [global type inference algorithm](#).

Class variables are inherited by subclasses with this meaning: their type is the same, but each class has a different runtime value. For example:

```
class Parent
  @@numbers = [] of Int32

  def self.numbers
    @@numbers
  end
end

class Child < Parent
end

Parent.numbers # => []
Child.numbers # => []

Parent.numbers << 1
Parent.numbers # => [1]
Child.numbers # => []
```

Class variables can also be associated to modules and structs. Like above, they are inherited by including/subclassing types.

# finalize

If a class defines a `finalize` method, when an instance of that class is garbage-collected that method will be invoked:

```
class Foo
  def finalize
    # Invoked when Foo is garbage-collected
    # Use to release non-managed resources (ie. C libraries, structs)
  end
end
```

Use this method to release resources allocated by external libraries that are not directly managed by Crystal garbage collector.

Examples of this can be found in `IO::FileDescriptor#finalize` or `OpenSSL::Digest#finalize`.

## Notes:

- The `finalize` method will only be invoked once the object has been fully initialized via the `initialize` method. If an exception is raised inside the `initialize` method, `finalize` won't be invoked. If your class defines a `finalize` method, be sure to catch any exceptions that might be raised in the `initialize` methods and free resources.
- Allocating any new object instances during garbage-collection might result in undefined behavior and most likely crashing your program.

# Modules

Modules serve two purposes:

- as namespaces for defining other types, methods and constants
- as partial types that can be mixed in other types

An example of a module as a namespace:

```
module Curses
  class Window
    end
  end

  Curses::Window.new
```

Library authors are advised to put their definitions inside a module to avoid name clashes. The standard library usually doesn't have a namespace as its types and methods are very common, to avoid writing long names.

To use a module as a partial type you use `include` or `extend`.

An `include` makes a type include methods defined in that module as instance methods:

```
module ItemsSize
  def size
    items.size
  end
end

class Items
  include ItemsSize

  def items
    [1, 2, 3]
  end
end

items = Items.new
items.size # => 3
```

In the above example, it is as if we pasted the `size` method from the module into the `Items` class. The way this really works is by making each type have a list of ancestors, or parents. By default this list starts with the superclass. As modules are included they are **prepended** to this list. When a method is not found in a type it is looked up in this list. When you invoke `super`, the first type in this ancestors list is used.

A `module` can include other modules, so when a method is not found in it it will be looked up in the included modules.

An `extend` makes a type include methods defined in that module as class methods:

```

module SomeSize
  def size
    3
  end
end

class Items
  extend SomeSize
end

Items.size # => 3

```

Both `include` and `extend` make constants defined in the module available to the including/extending type.

Both of them can be used at the top level to avoid writing a namespace over and over (although the chances of name clashes increase):

```

module SomeModule
  class SomeType
    end

    def some_method
      1
    end
  end

  include SomeModule

  SomeType.new # OK, same as SomeModule::SomeType
  some_method # OK, 1

```

## extend self

A common pattern for modules is `extend self`:

```

module Base64
  extend self

  def encode64(string)
    # ...
  end

  def decode64(string)
    # ...
  end
end

```

In this way a module can be used as a namespace:

```
Base64.encode64 "hello" # => "aGVsbG8="
```

But also it can be included in the program and its methods can be invoked without a namespace:

```
include Base64

encode64 "hello" # => "aGVsbG8="
```

For this to be useful the method name should have some reference to the module, otherwise chances of name clashes are high.

A module cannot be instantiated:

```
module Moo
end

Moo.new # undefined method 'new' for Moo:Module
```

## Module Type Checking

Modules can also be used for type checking.

If we define two modules with names `A` and `B`:

```
module A; end

module B; end
```

These can be included into classes:

```
class One
  include A
end

class Two
  include B
end

class Three < Two
  include A
end
```

We can then type check against instances of these classes with not only their class, but the included modules as well:

```
one = One.new
typeof(one) # => One
one.is_a?(A) # => true
one.is_a?(B) # => false

three = Three.new
typeof(three) # => Three
three.is_a?(A) # => true
three.is_a?(B) # => true
```

This allows you to define arrays and methods based on module type instead of class:

Nil

```
one = One.new
two = Two.new
three = Three.new

new_array = Array(A).new
new_array << one # Ok, One inherits module A
new_array << three # Ok, Three includes module A

new_array << two # Error, because Two does not inherit module A
```

## Generics

Generics allow you to parameterize a type based on other type. Consider a Box type:

```
class MyBox(T)
  def initialize(@value : T)
    end

  def value
    @value
  end
end

int_box = MyBox(Int32).new(1)
int_box.value # => 1 (Int32)

string_box = MyBox(String).new("hello")
string_box.value # => "hello" (String)

another_box = MyBox(String).new(1) # Error, Int32 doesn't match String
```

Generics are especially useful for implementing collection types. `Array`, `Hash`, `Set` are generic types, as is `Pointer`.

More than one type argument is allowed:

```
class MyDictionary(K, V)
  end
```

Any name can be used for type arguments:

```
class MyDictionary(KeyType, ValueType)
  end
```

## Type variables inference

Type restrictions in a generic type's constructor are free variables when type arguments were not specified, and then are used to infer them. For example:

```
MyBox.new(1) # : MyBox(Int32)
MyBox.new("hello") # : MyBox(String)
```

In the above code we didn't have to specify the type arguments of `MyBox`, the compiler inferred them following this process:

- `MyBox.new(value)` delegates to `initialize(@value : T)`
- `T` isn't bound to a type yet, so the compiler binds it to the type of the given argument

In this way generic types are less tedious to work with.

## Generic structs and modules

Structs and modules can be generic too. When a module is generic you include it like this:

```
module Moo(T)
  def t
    T
  end
end

class Foo(U)
  include Moo(U)

  def initialize(@value : U)
    end
end

foo = Foo.new(1)
foo.t # Int32
```

Note that in the above example `T` becomes `Int32` because `Foo.new(1)` makes `U` become `Int32`, which in turn makes `T` become `Int32` via the inclusion of the generic module.

## Generic types inheritance

Generic classes and structs can be inherited. When inheriting you can specify an instance of the generic type, or delegate type variables:

```
class Parent(T)
  end

class Int32Child < Parent(Int32)
  end

class GenericChild(T) < Parent(T)
  end
```



## Structs

Instead of defining a type with `class` you can do so with `struct` :

```
struct Point
  property x, y

  def initialize(@x : Int32, @y : Int32)
  end
end
```

Structs inherit from [Value](#) so they are allocated on the stack and passed by value: when passed to methods, returned from methods or assigned to variables, a copy of the value is actually passed (while classes inherit from [Reference](#), are allocated on the heap and passed by reference).

Therefore structs are mostly useful for immutable data types and/or stateless wrappers of other types, usually for performance reasons to avoid lots of small memory allocations when passing small copies might be more efficient (for more details, see the [performance guide](#)).

Mutable structs are still allowed, but you should be careful when writing code involving mutability if you want to avoid surprises that are described below.

## Passing by value

A struct is *always* passed by value, even when you return `self` from the method of that struct:

```
struct Counter
  def initialize(@count : Int32)
  end

  def plus
    @count += 1
    self
  end
end

counter = Counter.new(0)
counter.plus.plus # => Counter(@x=2)
puts counter     # => Counter(@x=1)
```

Notice that the chained calls of `plus` return the expected result, but only the first call to it modifies the variable `counter` , as the second call operates on the *copy* of the struct passed to it from the first call, and this copy is discarded after the expression is executed.

You should also be careful when working on mutable types inside of the struct:

```

class Klass
  property array = ["str"]
end

struct Strukt
  property array = ["str"]
end

def modify(object)
  object.array << "foo"
  object.array = ["new"]
  object.array << "bar"
end

klass = Klass.new
puts modify(klass) # => ["new", "bar"]
puts klass.array  # => ["new", "bar"]

strukt = Strukt.new
puts modify(strukt) # => ["new", "bar"]
puts strukt.array  # => ["str", "foo"]

```

What happens with the `strukt` here:

- `Array` is passed by reference, so the reference to `["str"]` is stored in the property of `strukt`
- When `strukt` is passed to `modify`, a *copy* of the `strukt` is passed with the reference to array inside it
- the array referenced by `array` is modified (element inside it is added) by `object.array << "foo"`
- this is also reflected in the original `strukt` as it holds reference to the same array
- `object.array = ["new"]` replaces the reference in the *copy* of `strukt` with the reference to the new array
- `object.array << "bar"` appends to this newly created array
- `modify` returns the reference to this new array and its content is printed
- the reference to this new array was held only in the *copy* of `strukt`, but not in the original, so that's why the original `strukt` only retained the result of the first statement, but not of the other two statements

`Klass` is a class, so it is passed by reference to `modify`, and `object.array = ["new"]` saves the reference to the newly created array in the original `klass` object, not in the copy as it was with the `strukt`.

## Inheritance

- A struct implicitly inherits from `Struct`, which inherits from `Value`. A class implicitly inherits from `Reference`.
- A struct cannot inherit from a non-abstract struct.

The second point has a reason to it: a struct has a very well defined memory layout. For example, the above `Point` struct occupies 8 bytes. If you have an array of points the points are embedded inside the array's buffer:

```
# The array's buffer will have 8 bytes dedicated to each Point  
ary = [] of Point
```

If `Point` is inherited, an array of such type should also account for the fact that other types can be inside it, so the size of each element should grow to accommodate that. That is certainly unexpected. So, non-abstract structs can't be inherited from. Abstract structs, on the other hand, will have descendants, so it is expected that an array of them will account for the possibility of having multiple types inside it.

A struct can also include modules and can be generic, just like a class.

## Constants

Constants can be declared at the top level or inside other types. They must start with a capital letter:

```
PI = 3.14

module Earth
  RADIUS = 6_371_000
end

PI          # => 3.14
Earth::RADIUS # => 6_371_000
```

Although not enforced by the compiler, constants are usually named with all capital letters and underscores to separate words.

A constant definition can invoke methods and have complex logic:

```
TEN = begin
  a = 0
  while a < 10
    a += 1
  end
  a
end

TEN # => 10
```

## Pseudo Constants

Crystal provides a few pseudo-constants which provide reflective data about the source code being executed.

`__LINE__` is the current line number in the currently executing crystal file. When `__LINE__` is declared as the default value to a method parameter, it represents the line number at the location of the method call.

`__END_LINE__` is the line number of the `end` of the calling block. Can only be used as a default value to a method parameter.

`__FILE__` references the full path to the currently executing crystal file.

`__DIR__` references the full path to the directory where the currently executing crystal file is located.

```
# Assuming this example code is saved at: /crystal_code/pseudo_constants.cr
#
def pseudo_constants(caller_line = __LINE__, end_of_caller = __END_LINE__)
  puts "Called from line number: #{caller_line}"
  puts "Currently at line number: #{__LINE__}"
  puts "End of caller block is at: #{end_of_caller}"
  puts "File path is: #{__FILE__}"
  puts "Directory file is in: #{__DIR__}"
end

begin
  pseudo_constants
end

# Program prints:
# Called from line number: 13
# Currently at line number: 5
# End of caller block is at: 14
# File path is: /crystal_code/pseudo_constants.cr
# Directory file is in: /crystal_code
```

## Dynamic assignment

Dynamically assigning values to constants using the [chained assignment](#) or the [multiple assignment](#) is not supported and results in a syntax error.

```
ONE, TWO, THREE = 1, 2, 3 # Syntax error: Multiple assignment is not allowed for const
```

## Enums

An enum is a set of integer values, where each value has an associated name. For example:

```
enum Color
  Red
  Green
  Blue
end
```

An enum is defined with the `enum` keyword, followed by its name. The enum's body contains the values. Values start with the value `0` and are incremented by one. The default value can be overwritten:

```
enum Color
  Red      # 0
  Green    # 1
  Blue    = 5 # overwritten to 5
  Yellow   # 6 (5 + 1)
end
```

Each constant in the enum has the type of the enum:

```
Color::Red # :: Color
```

To get the underlying value you invoke `value` on it:

```
Color::Green.value # => 1
```

The type of the value is `Int32` by default but can be changed:

```
enum Color : UInt8
  Red
  Green
  Blue
end

Color::Red.value # :: UInt8
```

Only integer types are allowed as the underlying type.

All enums inherit from [Enum](#).

## Flags enums

An enum can be marked with the `@[Flags]` attribute. This changes the default values:

```
@[Flags]
enum IOMode
  Read # 1
  Write # 2
  Async # 4
end
```

The `@[Flags]` attribute makes the first constant's value be `1`, and successive constants are multiplied by `2`.

Implicit constants, `None` and `All`, are automatically added to these enums, where `None` has the value `0` and `All` has the "or"ed value of all constants.

```
IOMode::None.value # => 0
IOMode::All.value # => 7
```

Additionally, some `Enum` methods check the `@[Flags]` attribute. For example:

```
puts(Color::Red) # prints "Red"
puts(IOMode::Write | IOMode::Async) # prints "Write, Async"
```

## Enums from integers

An enum can be created from an integer:

```
puts Color.new(1) # => prints "Green"
```

Values that don't correspond to an enum's constants are allowed: the value will still be of type `Color`, but when printed you will get the underlying value:

```
puts Color.new(10) # => prints "10"
```

This method is mainly intended to convert integers from C to enums in Crystal.

## Methods

Just like a class or a struct, you can define methods for enums:

```
enum Color
  Red
  Green
  Blue

  def red?
    self == Color::Red
  end
end

Color::Red.red? # => true
Color::Blue.red? # => false
```

Class variables are allowed, but instance variables are not.

## Usage

Enums are a type-safe alternative to [Symbol](#). For example, an API's method can specify a [type restriction](#) using an enum type:

```
def paint(color : Color)
  case color
  when Color::Red
    # ...
  else
    # Unusual, but still can happen
    raise "unknown color: #{color}"
  end
end

paint Color::Red
```

The above could also be implemented with a Symbol:

```
def paint(color : Symbol)
  case color
  when :red
    # ...
  else
    raise "unknown color: #{color}"
  end
end

paint :red
```

However, if the programmer makes a typo, say `:reed`, the error will only be caught at runtime, while attempting to use `Color::Reed` will result in a compile-time error.

The recommended thing to do is to use enums whenever possible, only use symbols for the internal implementation of an API, and avoid symbols for public APIs. But you are free to do what you want.



## Blocks and Procs

Methods can accept a block of code that is executed with the `yield` keyword. For example:

```
def twice
  yield
  yield
end

twice do
  puts "Hello!"
end
```

The above program prints "Hello!" twice, once for each `yield`.

To define a method that receives a block, simply use `yield` inside it and the compiler will know. You can make this more evident by declaring a dummy block argument, indicated as a last argument prefixed with ampersand (`&`):

```
def twice(&block)
  yield
  yield
end
```

To invoke a method and pass a block, you use `do ... end` or `{ ... }`. All of these are equivalent:

```
twice() do
  puts "Hello!"
end

twice do
  puts "Hello!"
end

twice { puts "Hello!" }
```

The difference between using `do ... end` and `{ ... }` is that `do ... end` binds to the left-most call, while `{ ... }` binds to the right-most call:

```
foo bar do
  something
end

# The above is the same as
foo(bar) do
  something
end

foo bar { something }
```

```
# The above is the same as
foo(bar { something })
```

The reason for this is to allow creating Domain Specific Languages (DSLs) using `do ... end` to have them be read as plain English:

```
open file "foo.cr" do
  something
end

# Same as:
open(file("foo.cr")) do
end
```

You wouldn't want the above to be:

```
open(file("foo.cr")) do
end)
```

## Overloads

Two methods, one that yields and another that doesn't, are considered different overloads, as explained in the [overloading](#) section.

## Yield arguments

The `yield` expression is similar to a call and can receive arguments. For example:

```
def twice
  yield 1
  yield 2
end

twice do |i|
  puts "Got #{i}"
end
```

The above prints "Got 1" and "Got 2".

A curly braces notation is also available:

```
twice { |i| puts "Got #{i}" }
```

You can `yield` many values:

```
def many
  yield 1, 2, 3
end

many do |x, y, z|
  puts x + y + z
end

# Output: 6
```

A block can specify less than the arguments yielded:

```
def many
  yield 1, 2, 3
end

many do |x, y|
  puts x + y
end

# Output: 3
```

It's an error specifying more block arguments than those yielded:

```
def twice
  yield
  yield
end

twice do |i| # Error: too many block arguments
end
```

Each block variable has the type of every yield expression in that position. For example:

```
def some
  yield 1, 'a'
  yield true, "hello"
  yield 2, nil
end

some do |first, second|
  # first is Int32 | Bool
  # second is Char | String | Nil
end
```

The block variable `second` also includes the `Nil` type because the last `yield` expression didn't include a second argument.

## Short one-argument syntax

If a block has a single argument and invokes a method on it, the block can be replaced with the short syntax argument.

This:

```
method do |argument|
  argument.some_method
end
```

and

```
method { |argument| argument.some_method }
```

can both be written as:

```
method &.some_method
```

Or like:

```
method(&.some_method)
```

In either case, `&.some_method` is an argument passed to `method`. This argument is syntactically equivalent to the block variants. It is only syntactic sugar and does not have any performance penalty.

If the method has other required parameters, the short syntax argument should also be supplied in the method's argument list.

```
["a", "b"].join(",", &.upcase)
```

Is equivalent to:

```
["a", "b"].join(",") { |s| s.upcase }
```

Arguments can be used with the short syntax argument as well:

```
["i", "o"].join(",", &.upcase(Unicode::CaseOptions::Turkic))
```

Operators can be invoked too:

```
method &.+{2}
method(&.[index])
```

## yield value

The `yield` expression itself has a value: the last expression of the block. For example:

```
def twice
  v1 = yield 1
  puts v1

  v2 = yield 2
  puts v2
end

twice do |i|
  i + 1
end
```

The above prints "2" and "3".

A `yield` expression's value is mostly useful for transforming and filtering values. The best examples of this are [Enumerable#map](#) and [Enumerable#select](#):

```
ary = [1, 2, 3]
ary.map { |x| x + 1 } # => [2, 3, 4]
ary.select { |x| x % 2 == 1 } # => [1, 3]
```

A dummy transformation method:

```
def transform(value)
  yield value
end

transform(1) { |x| x + 1 } # => 2
```

The result of the last expression is `2` because the last expression of the `transform` method is `yield`, whose value is the last expression of the block.

## Type restrictions

The type of the block in a method that uses `yield` can be restricted using the `&block` syntax. For example:

```
def transform_int(start : Int32, &block : Int32 -> Int32)
  result = yield start
  result * 2
end

transform_int(3) { |x| x + 2 } # => 10
transform_int(3) { |x| "foo" } # Error: expected block to return Int32, not String
```

## break

A `break` expression inside a block exits early from the method:

```
def thrice
  puts "Before 1"
  yield 1
  puts "Before 2"
  yield 2
  puts "Before 3"
  yield 3
  puts "After 3"
end

thrice do |i|
  if i == 2
    break
  end
end
```

The above prints "Before 1" and "Before 2". The `thrice` method didn't execute the `puts "Before 3"` expression because of the `break`.

`break` can also accept arguments: these become the method's return value. For example:

```
def twice
  yield 1
  yield 2
end

twice { |i| i + 1 }      # => 3
twice { |i| break "hello" } # => "hello"
```

The first call's value is 3 because the last expression of the `twice` method is `yield`, which gets the value of the block. The second call's value is "hello" because a `break` was performed.

If there are conditional breaks, the call's return value type will be a union of the type of the block's value and the type of the many `break` s:

```
value = twice do |i|
  if i == 1
    break "hello"
  end
  i + 1
end
value # :: Int32 | String
```

If a `break` receives many arguments, they are automatically transformed to a [Tuple](#):

```
values = twice { break 1, 2 }
values # => {1, 2}
```

If a `break` receives no arguments, it's the same as receiving a single `nil` argument:

```
value = twice { break }
value # => nil
```

## next

The `next` expression inside a block exits early from the block (not the method). For example:

```

def twice
  yield 1
  yield 2
end

twice do |i|
  if i == 1
    puts "Skipping 1"
    next
  end

  puts "Got #{i}"
end

# Output:
# Skipping 1
# Got 2

```

The `next` expression accepts arguments, and these give the value of the `yield` expression that invoked the block:

```

def twice
  v1 = yield 1
  puts v1

  v2 = yield 2
  puts v2
end

twice do |i|
  if i == 1
    next 10
  end

  i + 1
end

# Output
# 10
# 3

```

If a `next` receives many arguments, they are automatically transformed to a [Tuple](#). If it receives no arguments it's the same as receiving a single `nil` argument.

## with ... yield

A `yield` expression can be modified, using the `with` keyword, to specify an object to use as the default receiver of method calls within the block:

```

class Foo
  def one
    1
  end

  def yield_with_self
    with self yield
  end

  def yield_normally
    yield
  end
end

def one
  "one"
end

Foo.new.yield_with_self { one } # => 1
Foo.new.yield_normally { one } # => "one"

```

## Unpacking block arguments

A block argument can specify sub-arguments enclosed in parentheses:

```

array = [{1, "one"}, {2, "two"}]
array.each do |(number, word)|
  puts "#{number}: #{word}"
end

```

The above is simply syntax sugar of this:

```

array = [{1, "one"}, {2, "two"}]
array.each do |arg|
  number = arg[0]
  word = arg[1]
  puts "#{number}: #{word}"
end

```

That means that any type that responds to `[]` with integers can be unpacked in a block argument.

For [Tuple](#) arguments you can take advantage of auto-splatting and do not need parentheses:

```

array = [{1, "one", true}, {2, "two", false}]
array.each do |number, word, bool|
  puts "#{number}: #{word} #{bool}"
end

```

[Hash\(K, V\)#each](#):Nil-instance-method) passes `Tuple(K, V)` to the block so iterating key-value pairs works with auto-splatting:



```
h = {"foo" => "bar"}
h.each do |key, value|
  key # => "foo"
  value # => "bar"
end
```

## Performance

When using blocks with `yield`, the blocks are **always** inlined: no closures, calls or function pointers are involved. This means that this:

```
def twice
  yield 1
  yield 2
end

twice do |i|
  puts "Got: #{i}"
end
```

is exactly the same as writing this:

```
i = 1
puts "Got: #{i}"
i = 2
puts "Got: #{i}"
```

For example, the standard library includes a `times` method on integers, allowing you to write:

```
3.times do |i|
  puts i
end
```

This looks very fancy, but is it as fast as a C for loop? The answer is: yes!

This is `Int#times` definition:

```
struct Int
  def times
    i = 0
    while i < self
      yield i
      i += 1
    end
  end
end
```

Because a non-captured block is always inlined, the above method invocation is **exactly the same** as writing this:

Nil

```
i = 0
while i < 3
  puts i
  i += 1
end
```

Have no fear using blocks for readability or code reuse, it won't affect the resulting executable performance.

## Capturing blocks

A block can be captured and turned into a `Proc`, which represents a block of code with an associated context: the closed data.

To capture a block you must specify it as a method's block argument, give it a name and specify the input and output types. For example:

```
def int_to_int(&block : Int32 -> Int32)
  block
end

proc = int_to_int { |x| x + 1 }
proc.call(1) # => 2
```

The above code captures the block of code passed to `int_to_int` in the `block` variable, and returns it from the method. The type of `proc` is `Proc(Int32, Int32)`, a function that accepts a single `Int32` argument and returns an `Int32`.

In this way a block can be saved as a callback:

```
class Model
  def on_save(&block)
    @on_save_callback = block
  end

  def save
    if callback = @on_save_callback
      callback.call
    end
  end
end

model = Model.new
model.on_save { puts "Saved!" }
model.save # prints "Saved!"
```

In the above example the type of `&block` wasn't specified: this just means that the captured block doesn't have arguments and doesn't return anything.

Note that if the return type is not specified, nothing gets returned from the `proc` call:

```
def some_proc(&block : Int32 ->)
  block
end

proc = some_proc { |x| x + 1 }
proc.call(1) # void
```

To have something returned, either specify the return type or use an underscore to allow any return type:

```
def some_proc(&block : Int32 -> _)
  block
end

proc = some_proc { |x| x + 1 }
proc.call(1) # 2

proc = some_proc { |x| x.to_s }
proc.call(1) # "1"
```

## break and next

`return` and `break` can't be used inside a captured block. `next` can be used and will exit and give the value of the captured block.

## with ... yield

The default receiver within a captured block can't be changed by using `with ... yield`.

## Proc literal

A captured block is the same as declaring a [Proc literal](#) and [passing](#) it to the method.

```
def some_proc(&block : Int32 -> Int32)
  block
end

x = 0
proc = ->(i : Int32) { x += i }
proc = some_proc(&proc)
proc.call(1) # => 1
proc.call(10) # => 11
x           # => 11
```

As explained in the [proc literals](#) section, a Proc can also be created from existing methods:

```
def add(x, y)
  x + y
end

adder = ->add(Int32, Int32)
adder.call(1, 2) # => 3
```

## Block forwarding

To forward captured blocks, you use a block argument, prefixing an expression with `&`:

```
def capture(&block)
  block
end

def invoke(&block)
  block.call
end

proc = capture { puts "Hello" }
invoke(&proc) # prints "Hello"
```

In the above example, `invoke` receives a block. We can't pass `proc` directly to it because `invoke` doesn't receive regular arguments, just a block argument. We use `&` to specify that we really want to pass `proc` as the block argument. Otherwise:

```
invoke(proc) # Error: wrong number of arguments for 'invoke' (1 for 0)
```

You can actually pass a proc to a method that yields:

```
def capture(&block)
  block
end

def twice
  yield
  yield
end

proc = capture { puts "Hello" }
twice &proc
```

The above is simply rewritten to:

```
proc = capture { puts "Hello" }
twice do
  proc.call
end
```

Or, combining the `&` and `->` syntaxes:

```
twice &->{ puts "Hello" }
```

Or:

```
def say_hello
  puts "Hello"
end

twice &->say_hello
```

## Forwarding non-captured blocks

To forward non-captured blocks, you must use `yield` :

```
def foo
  yield 1
end

def wrap_foo
  puts "Before foo"
  foo do |x|
    yield x
  end
  puts "After foo"
end

wrap_foo do |i|
  puts i
end

# Output:
# Before foo
# 1
# After foo
```

You can also use the `&block` syntax to forward blocks, but then you have to at least specify the input types, and the generated code will involve closures and will be slower:

```
def foo
  yield 1
end

def wrap_foo(&block : Int32 -> _)
  puts "Before foo"
  foo(&block)
  puts "After foo"
end

wrap_foo do |i|
  puts i
end

# Output:
# Before foo
# 1
# After foo
```

Try to avoid forwarding blocks like this if doing `yield` is enough. There's also the issue that `break` and `next` are not allowed inside captured blocks, so the following won't work when using `&block` forwarding:

Nil

```
foo_forward do |i|  
  break # error  
end
```

In short, avoid `&block` forwarding when `yield` is involved.



## Closures

Captured blocks and proc literals closure local variables and `self`. This is better understood with an example:

```
x = 0
proc = ->{ x += 1; x }
proc.call # => 1
proc.call # => 2
x        # => 2
```

Or with a proc returned from a method:

```
def counter
  x = 0
  ->{ x += 1; x }
end

proc = counter
proc.call # => 1
proc.call # => 2
```

In the above example, even though `x` is a local variable, it was captured by the proc literal. In this case the compiler allocates `x` on the heap and uses it as the context data of the proc to make it work, because normally local variables live in the stack and are gone after a method returns.

## Type of closed variables

The compiler is usually moderately smart about the type of local variables. For example:

```
def foo
  yield
end

x = 1
foo do
  x = "hello"
end
x # : Int32 | String
```

The compiler knows that after the block, `x` can be `Int32` or `String` (it could know that it will always be `String` because the method always yields; this may improve in the future).

If `x` is assigned something else after the block, the compiler knows the type changed:

```
x = 1
foo do
  x = "hello"
end
x # : Int32 | String

x = 'a'
x # : Char
```

However, if `x` is closed by a proc, the type is always the mixed type of all assignments to it:

```
def capture(&block)
  block
end

x = 1
capture { x = "hello" }

x = 'a'
x # : Int32 | String | Char
```

This is because the captured block could have been potentially stored in a class or instance variable and invoked in a separate thread in between the instructions. The compiler doesn't do an exhaustive analysis of this: it just assumes that if a variable is captured by a proc, the time of that proc invocation is unknown.

This also happens with regular proc literals, even if it's evident that the proc wasn't invoked or stored:

```
def capture(&block)
  block
end

x = 1
->{ x = "hello" }

x = 'a'
x # : Int32 | String | Char
```

## alias

With `alias` you can give a type a different name:

```
alias PInt32 = Pointer(Int32)

ptr = PInt32.malloc(1) # : Pointer(Int32)
```

Every time you use an alias the compiler replaces it with the type it refers to.

Aliases are useful to avoid writing long type names, but also to be able to talk about recursive types:

```
alias RecArray = Array(Int32) | Array(RecArray)

ary = [] of RecArray
ary.push [1, 2, 3]
ary.push ary
ary # => [[1, 2, 3], [...]]
```

A real-world example of a recursive type is json:

```
module Json
  alias Type = Nil |
    Bool |
    Int64 |
    Float64 |
    String |
    Array(Type) |
    Hash(String, Type)
end
```

# Exception handling

Crystal's way to do error handling is by raising and rescuing exceptions.

## Raising exception

You raise exceptions by invoking a top-level `raise` method. Unlike other keywords, `raise` is a regular method with two overloads: [one accepting a String](#) and another [accepting an Exception instance](#):

```
raise "OH NO!"
raise Exception.new("Some error")
```

The String version just creates a new [Exception](#) instance with that message.

Only `Exception` instances or subclasses can be raised.

## Defining custom exceptions

To define a custom exception type, just subclass from [Exception](#):

```
class MyException < Exception
end

class MyOtherException < Exception
end
```

You can, as always, define a constructor for your exception or just use the default one.

## Rescuing exceptions

To rescue any exception use a `begin ... rescue ... end` expression:

```
begin
  raise "OH NO!"
rescue
  puts "Rescued!"
end

# Output: Rescued!
```

To access the rescued exception you can specify a variable in the `rescue` clause:

```
begin
  raise "OH NO!"
rescue ex
  puts ex.message
end

# Output: OH NO!
```

To rescue just one type of exception (or any of its subclasses):

```
begin
  raise MyException.new("OH NO!")
rescue MyException
  puts "Rescued MyException"
end

# Output: Rescued MyException
```

And to access it, use a syntax similar to type restrictions:

```
begin
  raise MyException.new("OH NO!")
rescue ex : MyException
  puts "Rescued MyException: #{ex.message}"
end

# Output: Rescued MyException: OH NO!
```

Multiple `rescue` clauses can be specified:

```
begin
  # ...
rescue ex1 : MyException
  # only MyException...
rescue ex2 : MyOtherException
  # only MyOtherException...
rescue
  # any other kind of exception
end
```

You can also rescue multiple exception types at once by specifying a union type:

```
begin
  # ...
rescue ex : MyException | MyOtherException
  # only MyException or MyOtherException
rescue
  # any other kind of exception
end
```

## else

An `else` clause is executed only if no exceptions were rescued:

```
begin
  something_dangerous
rescue
  # execute this if an exception is raised
else
  # execute this if an exception isn't raised
end
```

An `else` clause can only be specified if at least one `rescue` clause is specified.

## ensure

An `ensure` clause is executed at the end of a `begin ... end` OR `begin ... rescue ... end` expression regardless of whether an exception was raised or not:

```
begin
  something_dangerous
ensure
  puts "Cleanup..."
end

# Will print "Cleanup..." after invoking something_dangerous,
# regardless of whether it raised or not
```

Or:

```
begin
  something_dangerous
rescue
  # ...
else
  # ...
ensure
  # this will always be executed
end
```

`ensure` clauses are usually used for clean up, freeing resources, etc.

## Short syntax form

Exception handling has a short syntax form: assume a method or block definition is an implicit `begin ... end` expression, then specify `rescue`, `else`, and `ensure` clauses:

```
def some_method
  something_dangerous
rescue
  # execute if an exception is raised
end

# The above is the same as:
def some_method
  begin
    something_dangerous
  rescue
    # execute if an exception is raised
  end
end
```

With `ensure` :

```
def some_method
  something_dangerous
ensure
  # always execute this
end

# The above is the same as:
def some_method
  begin
    something_dangerous
  ensure
    # always execute this
  end
end

# Similarly, the shorthand also works with blocks:
(1..10).each do |n|
  # potentially dangerous operation

rescue
  # ..
else
  # ..
ensure
  # ..
end
```

## Type inference

Variables declared inside the `begin` part of an exception handler also get the `Nil` type when considered inside a `rescue` or `ensure` body. For example:

```
begin
  a = something_dangerous_that_returns_Int32
ensure
  puts a + 1 # error, undefined method '+' for Nil
end
```

The above happens even if `something_dangerous_that_returns_Int32` never raises, or if `a` was assigned a value and then a method that potentially raises is executed:

```
begin
  a = 1
  something_dangerous
ensure
  puts a + 1 # error, undefined method '+' for Nil
end
```

Although it is obvious that `a` will always be assigned a value, the compiler will still think `a` might never had a chance to be initialized. Even though this logic might improve in the future, right now it forces you to keep your exception handlers to their necessary minimum, making the code's intention more clear:

```
# Clearer than the above: `a` doesn't need
# to be in the exception handling code.
a = 1
begin
  something_dangerous
ensure
  puts a + 1 # works
end
```

## Alternative ways to do error handling

Although exceptions are available as one of the mechanisms for handling errors, they are not your only choice. Raising an exception involves allocating memory, and executing an exception handler is generally slow.

The standard library usually provides a couple of methods to accomplish something: one raises, one returns `nil`. For example:

```
array = [1, 2, 3]
array[4] # raises because of IndexError
array[4]? # returns nil because of index out of bounds
```

The usual convention is to provide an alternative "question" method to signal that this variant of the method returns `nil` instead of raising. This lets the user choose whether she wants to deal with exceptions or with `nil`. Note, however, that this is not available for every method out there, as exceptions are still the preferred way because they don't pollute the code with error handling logic.



## Type grammar

When:

- specifying [type restrictions](#)
- specifying [type arguments](#)
- [declaring variables](#)
- declaring [aliases](#)
- declaring [typedefs](#)
- the argument of an [is\\_a?](#) pseudo-call
- the argument of an [as](#) expression
- the argument of a [sizeof](#) expression
- the argument of an [instance\\_sizeof](#) expression
- a method's [return type](#)

a convenient syntax is provided for some common types. These are especially useful when writing [C bindings](#), but can be used in any of the above locations.

## Paths and generics

Regular types and generics can be used:

```
Int32
My::Nested::Type
Array(String)
```

## Union

```
alias Int32OrString = Int32 | String
```

The pipe ( | ) in types creates a union type. `Int32 | String` is read "Int32 or String". In regular code, `Int32 | String` means invoking the method `|` on `Int32` with `String` as an argument.

## Nilable

```
alias Int32OrNil = Int32?
```

is the same as:

```
alias Int32OrNil = Int32 | ::Nil
```

In regular code, `Int32?` is an `Int32 | ::Nil` union type itself.

## Pointer

```
alias Int32Ptr = Int32*
```

is the same as:

```
alias Int32Ptr = Pointer(Int32)
```

In regular code, `Int32*` means invoking the `*` method on `Int32`.

## StaticArray

```
alias Int32_8 = Int32[8]
```

is the same as:

```
alias Int32_8 = StaticArray(Int32, 8)
```

In regular code, `Int32[8]` means invoking the `[]` method on `Int32` with `8` as an argument.

## Tuple

```
alias Int32StringTuple = {Int32, String}
```

is the same as:

```
alias Int32StringTuple = Tuple(Int32, String)
```

In regular code, `{Int32, String}` is a tuple instance containing `Int32` and `String` as its elements. This is different than the above tuple **type**.

## NamedTuple

```
alias Int32StringNamedTuple = {x: Int32, y: String}
```

is the same as:

```
alias Int32StringNamedTuple = NamedTuple(x: Int32, y: String)
```

In regular code, `{x: Int32, y: String}` is a named tuple instance containing `Int32` and `String` for `x` and `y`. This is different than the above named tuple **type**.

## Proc

```
alias Int32ToString = Int32 -> String
```

is the same as:

```
alias Int32ToString = Proc(Int32, String)
```

To specify a Proc without arguments:

```
alias ProcThatReturnsInt32 = -> Int32
```

To specify multiple arguments:

```
alias Int32AndCharToString = Int32, Char -> String
```

For nested procs (and any type, in general), you can use parentheses:

```
alias ComplexProc = (Int32 -> Int32) -> String
```

In regular code `Int32 -> String` is a syntax error.

## self

`self` can be used in the type grammar to denote a `self` type. Refer to the [type restrictions](#) section.

## class

`class` is used to refer to a class type, instead of an instance type.

For example:

```
def foo(x : Int32)
  "instance"
end

def foo(x : Int32.class)
  "class"
end

foo 1 # "instance"
foo Int32 # "class"
```

`class` is also useful for creating arrays and collections of class type:

```
class Parent
end

class Child1 < Parent
end

class Child2 < Parent
end

ary = [] of Parent.class
ary << Child1
ary << Child2
```

## Underscore

An underscore is allowed in type restrictions. It matches anything:

```
# Same as not specifying a restriction, not very useful
def foo(x : _)
end

# A bit more useful: any two arguments Proc that returns an Int32:
def foo(x : _, _ -> Int32)
end
```

## typeof

`typeof` is allowed in the type grammar. It returns a union type of the type of the passed expressions:

```
typeof(1 + 2) # => Int32
typeof(1, "a") # => (Int32 | String)
```

## Type reflection

Crystal provides basic methods to do type reflection, casting and introspection.

## is\_a?

The pseudo-method `is_a?` determines whether an expression's runtime type inherits or includes another type. For example:

```
a = 1
a.is_a?(Int32)      # => true
a.is_a?(String)    # => false
a.is_a?(Number)    # => true
a.is_a?(Int32 | String) # => true
```

It is a pseudo-method because the compiler knows about it and it can affect type information, as explained in [if var.is\\_a?\(...\)](#). Also, it accepts a [type](#) that must be known at compile-time as its argument.

## nil?

The pseudo-method `nil?` determines whether an expression's runtime type is `Nil`. For example:

```
a = 1
a.nil? # => false

b = nil
b.nil? # => true
```

It is a pseudo-method because the compiler knows about it and it can affect type information, as explained in [if var.nil?\(...\)](#).

It has the same effect as writing `is_a?(Nil)` but it's shorter and easier to read and write.

## responds\_to?

The pseudo-method `responds_to?` determines whether a type has a method with the given name. For example:

```
a = 1
a.responds_to?(:abs) # => true
a.responds_to?(:size) # => false
```

It is a pseudo-method because it only accepts a symbol literal as its argument, and is also treated specially by the compiler, as explained in [if var.responds\\_to? \(...\)](#).



## as

The `as` pseudo-method restricts the types of an expression. For example:

```
if some_condition
  a = 1
else
  a = "hello"
end

# a : Int32 | String
```

In the above code, `a` is a union of `Int32 | String`. If for some reason we are sure `a` is an `Int32` after the `if`, we can force the compiler to treat it like one:

```
a_as_int = a.as(Int32)
a_as_int.abs # works, compiler knows that a_as_int is Int32
```

The `as` pseudo-method performs a runtime check: if `a` wasn't an `Int32`, an [exception](#) is raised.

The argument to the expression is a [type](#).

If it is impossible for a type to be restricted by another type, a compile-time error is issued:

```
1.as(String) # Compile-time error
```

**Note:** you can't use `as` to convert a type to an unrelated type: `as` is not like a `cast` in other languages. Methods on integers, floats and chars are provided for these conversions. Alternatively, use pointer casts as explained below.

## Converting between pointer types

The `as` pseudo-method also allows to cast between pointer types:

```
ptr = Pointer(Int32).malloc(1)
ptr.as(Int8*) # :: Pointer(Int8)
```

In this case, no runtime checks are done: pointers are unsafe and this type of casting is usually only needed in C bindings and low-level code.

## Converting between pointer types and other types

Conversion between pointer types and Reference types is also possible:

```
array = [1, 2, 3]

# object_id returns the address of an object in memory,
# so we create a pointer with that address
ptr = Pointer(Void).new(array.object_id)

# Now we cast that pointer to the same type, and
# we should get the same value
array2 = ptr.as(Array(Int32))
array2.same?(array) # => true
```

No runtime checks are performed in these cases because, again, pointers are involved. The need for this cast is even more rare than the previous one, but allows to implement some core types (like String) in Crystal itself, and it also allows passing a Reference type to C functions by casting it to a void pointer.

## Usage for casting to a bigger type

The `as` pseudo-method can be used to cast an expression to a "bigger" type. For example:

```
a = 1
b = a.as(Int32 | Float64)
b # :: Int32 | Float64
```

The above might not seem to be useful, but it is when, for example, mapping an array of elements:

```
ary = [1, 2, 3]

# We want to create an array 1, 2, 3 of Int32 | Float64
ary2 = ary.map { |x| x.as(Int32 | Float64) }

ary2 # :: Array(Int32 | Float64)
ary2 << 1.5 # OK
```

The `Array#map` method uses the block's type as the generic type for the Array. Without the `as` pseudo-method, the inferred type would have been `Int32` and we wouldn't have been able to add a `Float64` into it.

## Usage for when the compiler can't infer the type of a block

Sometimes the compiler can't infer the type of a block. This can happen in recursive calls that depend on each other. In those cases you can use `as` to let it know the type:

```
some_call { |v| v.method.as(ExpectedType) }
```

## as?

The `as?` pseudo-method is similar to `as`, except that it returns `nil` instead of raising an exception when the type doesn't match. It also can't be used to cast between pointer types and other types.

Example:

```
value = rand < 0.5 ? -3 : nil
result = value.as?(Int32) || 10

value.as?(Int32).try &.abs
```

## typeof

The `typeof` expression returns the type of an expression:

```
a = 1
b = typeof(a) # => Int32
```

It accepts multiple arguments, and the result is the union of the expression types:

```
typeof(1, "a", 'a') # => (Int32 | String | Char)
```

It is often used in generic code, to make use of the compiler's type inference capabilities:

```
hash = {} of Int32 => String
another_hash = typeof(hash).new # :: Hash(Int32, String)
```

Since `typeof` doesn't actually evaluate the expression, it can be used on methods at compile time, such as in this example, which recursively forms a union type out of nested type parameters:

```
class Array
  def self.elem_type(typ)
    if typ.is_a?(Array)
      elem_type(typ.first)
    else
      typ
    end
  end
end

nest = [1, ["b", [:c, ['d']]]]
flat = Array(typeof(Array.elem_type(nest))).new
typeof(nest) # => Array(Int32 | Array(String | Array(Symbol | Array(Char))))
typeof(flat) # => Array(String | Int32 | Symbol | Char)
```

This expression is also available in the [type grammar](#).

## Macros

Macros are methods that receive AST nodes at compile-time and produce code that is pasted into a program. For example:

```
macro define_method(name, content)
  def {{name}}
    {{content}}
  end
end

# This generates:
#
#   def foo
#     1
#   end
define_method foo, 1

foo # => 1
```

A macro's definition body looks like regular Crystal code with extra syntax to manipulate the AST nodes. The generated code must be valid Crystal code, meaning that you can't for example generate a `def` without a matching `end`, or a single `when` expression of a `case`, since both of them are not complete valid expressions. Refer to [Pitfalls](#) for more information.

## Scope

Macros declared at the top-level are visible anywhere. If a top-level macro is marked as `private` it is only accessible in that file.

They can also be defined in classes and modules, and are visible in those scopes. Macros are also looked-up in the ancestors chain (superclasses and included modules).

For example, a block which is given an object to use as the default receiver by being invoked with `with ... yield` can access macros defined within that object's ancestors chain:

```
class Foo
  macro emphasize(value)
    """##{ {{value}} }"""
  end

  def yield_with_self
    with self yield
    end
  end
end

Foo.new.yield_with_self { emphasize(10) } # => """10"""
```

Macros defined in classes and modules can be invoked from outside of them too:

```
class Foo
  macro emphasize(value)
    """#{ {{value}} }"""
  end
end

Foo.emphasize(10) # => "****10****"
```

## Interpolation

You use `{{...}}` to paste, or interpolate, an AST node, as in the above example.

Note that the node is pasted as-is. If in the previous example we pass a symbol, the generated code becomes invalid:

```
# This generates:
#
#   def :foo
#     1
#   end
define_method :foo, 1
```

Note that `:foo` was the result of the interpolation, because that's what was passed to the macro. You can use the method `ASTNode#id` in these cases, where you just need an identifier.

## Macro calls

You can invoke a **fixed subset** of methods on AST nodes at compile-time. These methods are documented in a fictitious `Crystal::Macros` module.

For example, invoking `ASTNode#id` in the above example solves the problem:

```
macro define_method(name, content)
  def {{name.id}}
    {{content}}
  end
end

# This correctly generates:
#
#   def foo
#     1
#   end
define_method :foo, 1
```

## Modules and classes

Modules, classes and structs can also be generated:

```

macro define_class(module_name, class_name, method, content)
  module {{module_name}}
    class {{class_name}}
      def initialize(@name : String)
        end

      def {{method}}
        {{content}} + @name
      end
    end
  end
end

# This generates:
#   module Foo
#     class Bar
#       def initialize(@name : String)
#         end
#
#       def say
#         "hi " + @name
#       end
#     end
#   end
define_class Foo, Bar, say, "hi "

p Foo::Bar.new("John").say # => "hi John"

```

## Conditionals

You use `{% if condition %} ... {% end %}` to conditionally generate code:

```

macro define_method(name, content)
  def {{name}}
    {% if content == 1 %}
      "one"
    {% elsif content == 2 %}
      "two"
    {% else %}
      {{content}}
    {% end %}
  end
end

define_method foo, 1
define_method bar, 2
define_method baz, 3

foo # => one
bar # => two
baz # => 3

```

Similar to regular code, `Nop`, `NilLiteral` and a false `BoolLiteral` are considered *falsey*, while everything else is considered *truthy*.

Macro conditionals can be used outside a macro definition:

```
{% if env("TEST") %}
  puts "We are in test mode"
{% end %}
```

## Iteration

You can iterate a finite amount of times:

```
macro define_constants(count)
  {% for i in (1..count) %}
    PI_{{i.id}} = Math::PI * {{i}}
  {% end %}
end

define_constants(3)

PI_1 # => 3.14159...
PI_2 # => 6.28318...
PI_3 # => 9.42477...
```

To iterate an `ArrayLiteral` :

```
macro define_dummy_methods(names)
  {% for name, index in names %}
    def {{name.id}}
      {{index}}
    end
  {% end %}
end

define_dummy_methods [foo, bar, baz]

foo # => 0
bar # => 1
baz # => 2
```

The `index` variable in the above example is optional.

To iterate a `HashLiteral` :

```
macro define_dummy_methods(hash)
  {% for key, value in hash %}
    def {{key.id}}
      {{value}}
    end
  {% end %}
end

define_dummy_methods({foo: 10, bar: 20})

foo # => 10
bar # => 20
```

Macro iterations can be used outside a macro definition:



```

{% for name, index in ["foo", "bar", "baz"] %}
  def {{name.id}}
    {{index}}
  end
{% end %}

foo # => 0
bar # => 1
baz # => 2

```

## Variadic arguments and splatting

A macro can accept variadic arguments:

```

macro define_dummy_methods(*names)
  {% for name, index in names %}
    def {{name.id}}
      {{index}}
    end
  {% end %}
end

define_dummy_methods foo, bar, baz

foo # => 0
bar # => 1
baz # => 2

```

The arguments are packed into an `ArrayLiteral` and passed to the macro.

Additionally, using `*` when interpolating an `ArrayLiteral` interpolates the elements separated by commas:

```

macro println(*values)
  print {{*values}}, '\n'
end

println 1, 2, 3 # outputs 123\n

```

## Type information

When a macro is invoked you can access the current scope, or type, with a special instance variable: `@type`. The type of this variable is `TypeNode`, which gives you access to type information at compile time.

Note that `@type` is always the *instance* type, even when the macro is invoked in a class method.

For example:

```
macro add_describe_methods
  def describe
    "Class is: " + {{ @type.stringify }}
  end

  def self.describe
    "Class is: " + {{ @type.stringify }}
  end
end

class Foo
  add_describe_methods
end

Foo.new.describe # => "Class is Foo"
Foo.describe    # => "Class is Foo"
```

## Method information

When a macro is invoked you can access the method, the macro is in with a special instance variable: `@def`. The type of this variable is `Def` unless the macro is outside of a method, in this case it's `NilLiteral`.

Example:

```
module Foo
  def Foo.boo(arg1, arg2)
    {% @def.receiver %} # => Foo
    {% @def.name %}     # => boo
    {% @def.args %}    # => [arg1, arg2]
  end
end

Foo.boo(0, 1)
```

## Constants

Macros can access constants. For example:

```
VALUES = [1, 2, 3]

{% for value in VALUES %}
  puts {{value}}
{% end %}
```

If the constant denotes a type, you get back a `TypeNode`.

## Nested macros

It is possible to define a macro which generates one or more macro definitions. You must escape macro expressions of the inner macro by preceding them with a backslash character `"\"` to prevent them from being evaluated by the outer macro.

```

macro define_macros(*names)
  {% for name in names %}
    macro greeting_for_{{name.id}}(greeting)
      {% if greeting == "hola" %}
        ¡hola {{name.id}}!
      {% else %}
        "\{{greeting.id}} {{name.id}}"
      {% end %}
    end
  {% end %}
end

# This generates:
#
# macro greeting_for_alice
#   {% if greeting == "hola" %}
#     ¡hola alice!
#   {% else %}
#     "\{{greeting.id}} alice"
#   {% end %}
# end
# macro greeting_for_bob
#   {% if greeting == "hola" %}
#     ¡hola bob!
#   {% else %}
#     "\{{greeting.id}} bob"
#   {% end %}
# end
define_macros alice, bob

greeting_for_alice "hello" # => "hello alice"
greeting_for_bob "hallo" # => "hallo bob"
greeting_for_alice "hej" # => "hej alice"
greeting_for_bob "hola" # => "¡hola bob!"

```

## verbatim

Another way to define a nested macro is by using the special `verbatim` call. Using this you will not be able to use any variable interpolation but will not need to escape the inner macro characters.

```

macro define_macros(*names)
  {% for name in names %}
    macro greeting_for_{{name.id}}(greeting)

      # name will not be available within the verbatim block
      \{% name = {{name.stringify}} %}

      {% verbatim do %}
        {% if greeting == "hola" %}
          ¡hola {{name.id}}!
        {% else %}
          "{{greeting.id}} {{name.id}}"
        {% end %}
      {% end %}
    end
  {% end %}
end

# This generates:
#
# macro greeting_for_alice
#   {% name = "alice" %}
#   {% if greeting == "hola" %}
#     ¡hola alice!
#   {% else %}
#     "{{greeting.id}} alice"
#   {% end %}
# end
# macro greeting_for_bob
#   {% name = "bob" %}
#   {% if greeting == "hola" %}
#     ¡hola bob!
#   {% else %}
#     "{{greeting.id}} bob"
#   {% end %}
# end
define_macros alice, bob

greeting_for_alice "hello" # => "hello alice"
greeting_for_bob "hallo" # => "hallo bob"
greeting_for_alice "hej" # => "hej alice"
greeting_for_bob "hola" # => "¡hola bob!"

```

Notice the variables in the inner macro are not available within the `verbatim` block. The contents of the block are transferred "as is", essentially as a string, until re-examined by the compiler.

## Comments

Macro expressions are evaluated both within comments as well as compilable sections of code. This may be used to provide relevant documentation for expansions:

```

{% for name, index in ["foo", "bar", "baz"] %}
  # Provides a placeholder {{name.id}} method. Always returns {{index}}.
  def {{name.id}}
    {{index}}
  end
{% end %}

```

This evaluation applies to both interpolation and directives. As a result of this, macros cannot be commented out.

```
macro a
  # {% if false %}
  puts 42
  # {% end %}
end

a
```

The expression above will result in no output.

## Pitfalls

When writing macros (especially outside of a macro definition) it is important to remember that the generated code from the macro must be valid Crystal code by itself even before it is merged into the main program's code. This means, for example, a macro cannot generate a one or more `when` expressions of a `case` statement unless `case` was a part of the generated code.

Here is an example of such an invalid macro:

```
case 42
  {% for klass in [Int32, String] %} # Syntax Error: unexpected token: {% (expecting whe
    when {{klass.id}}
      p "is {{klass}}"
    {% end %}
  end
```

Notice that `case` is not within the macro. The code generated by the macro consists solely of two `when` expressions which, by themselves, is not valid Crystal code. We must include `case` within the macro in order to make it valid by using `begin` and `end`:

```
{% begin %}
case 42
  {% for klass in [Int32, String] %}
    when {{klass.id}}
      p "is {{klass}}"
    {% end %}
  end
{% end %}
```

## Macro methods

Macro defs allow you to define a method for a class hierarchy which is then instantiated for each concrete subtype.

A `def` is implicitly considered a `macro def` if it contains a macro expression which refers to `@type`. For example:

```
class Object
  def instance_vars_names
    {{ @type.instance_vars.map &.name.stringify }}
  end
end

class Person
  def initialize(@name : String, @age : Int32)
  end
end

person = Person.new "John", 30
person.instance_vars_names # => ["name", "age"]
```

In macro definitions, arguments are passed as their AST nodes, giving you access to them in macro expansions ( `{{some_macro_argument}}` ). However that is not true for macro defs. Here the argument list is that of the method generated by the macro def. You cannot access their compile-time value.

```
class Object
  def has_instance_var?(name) : Bool
    # We cannot access name inside the macro expansion here,
    # instead we need to use the macro language to construct an array
    # and do the inclusion check at runtime.
    {{ @type.instance_vars.map &.name.stringify }}.includes? name
  end
end

person = Person.new "John", 30
person.has_instance_var?("name") # => true
person.has_instance_var?("birthday") # => false
```

## Hooks

Special macros exist that are invoked in some situations as hooks, at compile time:

- `inherited` is invoked when a subclass is defined. `@type` is the inheriting type.
- `included` is invoked when a module is included. `@type` is the including type.
- `extended` is invoked when a module is extended. `@type` is the extending type.
- `method_missing` is invoked when a method is not found.
- `method_added` is invoked when a new method is defined in the current scope.
- `finished` is invoked after instance variable types for all classes are known.

Example of `inherited` :

```
class Parent
  macro inherited
    def lineage
      "{{@type.name.id}} < Parent"
    end
  end
end

class Child < Parent
end

Child.new.lineage # => "Child < Parent"
```

Example of `method_missing` :

```
macro method_missing(call)
  print "Got ", {{call.name.id.stringify}}, " with ", {{call.args.size}}, " arguments"
end

foo      # Prints: Got foo with 0 arguments
bar 'a', 'b' # Prints: Got bar with 2 arguments
```

Example of `method_added` :

```
macro method_added(method)
  {% puts "Method added:", method.name.stringify %}
end

def generate_random_number
  4
end

# => Method added: generate_random_number
```

Both `method_missing` and `method_added` only apply to calls or methods in the same class that the macro is defined in, or only in the top level if the macro is defined outside of a class. For example:

```

macro method_missing(call)
  puts "In outer scope, got call: ", {{ call.name.stringify }}
end

class SomeClass
  macro method_missing(call)
    puts "Inside SomeClass, got call: ", {{ call.name.stringify }}
  end
end

class OtherClass
end

# This call is handled by the top-level `method_missing`
foo # => In outer scope, got call: foo

obj = SomeClass.new
# This is handled by the one inside SomeClass
obj.bar # => Inside SomeClass, got call: bar

other = OtherClass.new
# Neither OtherClass or its parents define a `method_missing` macro
other.baz # => Error: Undefined method 'baz' for OtherClass

```

`finished` is called once a type has been completely defined - this includes extensions on that class. Consider the following program:

```

macro print_methods
  {% puts @type.methods.map &.name %}
end

class Foo
  macro finished
    {% puts @type.methods.map &.name %}
  end

  print_methods
end

class Foo
  def bar
    puts "I'm a method!"
  end
end

Foo.new.bar

```

The `print_methods` macro will be run as soon as it is encountered - and will print an empty list as there are no methods defined at that point. Once the second declaration of `Foo` is compiled the `finished` macro will be run, which will print `[bar]` .



## Fresh variables

Once macros generate code, they are parsed with a regular Crystal parser where local variables in the context of the macro invocations are assumed to be defined.

This is better understood with an example:

```
macro update_x
  x = 1
end

x = 0
update_x
x # => 1
```

This can sometimes be useful to avoid repetitive code by deliberately reading/writing local variables, but can also overwrite local variables by mistake. To avoid this, fresh variables can be declared with `%name :`

```
macro dont_update_x
  %x = 1
  puts %x
end

x = 0
dont_update_x # outputs 1
x # => 0
```

Using `%x` in the above example, we declare a variable whose name is guaranteed not to conflict with local variables in the current scope.

Additionally, fresh variables with respect to some other AST node can be declared with `%var{key1, key2, ..., keyN}`. For example:

```
macro fresh_vars_sample(*names)
  # First declare vars
  {% for name, index in names %}
    print "Declaring: ", "%name{index}", '\n'
    %name{index} = {{index}}
  {% end %}

  # Then print them
  {% for name, index in names %}
    print "%name{index}: ", %name{index}, '\n'
  {% end %}
end

fresh_vars_sample a, b, c

# Sample output:
# Declaring: __temp_255
# Declaring: __temp_256
# Declaring: __temp_257
# __temp_255: 0
# __temp_256: 1
# __temp_257: 2
```

Nil

In the above example, three indexed variables are declared, assigned values, and then printed, displaying their corresponding indices.

## Annotations

Annotations can be used to add metadata to certain features in the source code. Types, methods and instance variables may be annotated. User-defined annotations, such as the standard library's `JSON::Field`, are defined using the `annotation` keyword. A number of [built-in annotations](#) are provided by the compiler.

Users can define their own annotations using the `annotation` keyword, which works similarly to defining a `class` or `struct`.

```
annotation MyAnnotation
end
```

The annotation can then be applied to various items, including:

- Instance and class methods
- Instance variables
- Classes, structs, enums, and modules

```
annotation MyAnnotation
end

@[MyAnnotation]
def foo
  "foo"
end

@[MyAnnotation]
class Klass
end

@[MyAnnotation]
module MyModule
end
```

## Applications

Annotations are best used to store metadata about a given instance variable, type, or method so that it can be read at compile time using macros. One of the main benefits of annotations is that they are applied directly to instance variables/methods, which causes classes to look more natural since a standard macro is not needed to create these properties/methods.

A few applications for annotations:

### Object Serialization

Have an annotation that when applied to an instance variable determines if that instance variable should be serialized, or with what key. Crystal's

`JSON::Serializable` and `YAML::Serializable` are examples of this.

## ORMs

An annotation could be used to designate a property as an ORM column. The name and type of the instance variable can be read off the `TypeNode` in addition to the annotation; removing the need for any ORM specific macro. The annotation itself could also be used to store metadata about the column, such as if it is nullable, the name of the column, or if it is the primary key.

## Fields

Data can be stored within an annotation.

```
annotation MyAnnotation
end

# The fields can either be a key/value pair
@[MyAnnotation(key: "value", value: 123)]

# Or positional
@[MyAnnotation("foo", 123, false)]
```

## Key/value

The values of annotation key/value pairs can be accessed at compile time via the `[]` method.

```
annotation MyAnnotation
end

@[MyAnnotation(value: 2)]
def annotation_value
  # The name can be a `String`, `Symbol`, or `MacroId`
  {{ @def.annotation(MyAnnotation)[:value] }}
end

annotation_value # => 2
```

The `named_args` method can be used to read all key/value pairs on an annotation as a `NamedTupleLiteral`. This method is defined on all annotations by default, and is unique to each applied annotation.

```
annotation MyAnnotation
end

@[MyAnnotation(value: 2, name: "Jim")]
def annotation_named_args
  {{ @def.annotation(MyAnnotation).named_args }}
end

annotation_named_args # => {value: 2, name: "Jim"}
```

Since this method returns a `NamedTupleLiteral`, all of the [methods](#) on that type are available for use. Especially `#double_splat` which makes it easy to pass annotation arguments to methods.

```

annotation MyAnnotation
end

class SomeClass
  def initialize(@value : Int32, @name : String); end
end

@[MyAnnotation(value: 2, name: "Jim")]
def new_test
  {% begin %}
    SomeClass.new { { @def.annotation(MyAnnotation).named_args.double_splat } }
  {% end %}
end

new_test # => #<SomeClass:0x5621a19ddf00 @name="Jim", @value=2>

```

## Positional

Positional values can be accessed at compile time via the `[]` method; however, only one index can be accessed at a time.

```

annotation MyAnnotation
end

@[MyAnnotation(1, 2, 3, 4)]
def annotation_read
  {% for idx in [0, 1, 2, 3, 4] %}
    {% value = @def.annotation(MyAnnotation)[idx] %}
    pp "{{ idx }} = {{ value }}"
  {% end %}
end

annotation_read

# Which would print
"0 = 1"
"1 = 2"
"2 = 3"
"3 = 4"
"4 = nil"

```

The `args` method can be used to read all positional arguments on an annotation as a `TupleLiteral`. This method is defined on all annotations by default, and is unique to each applied annotation.

```

annotation MyAnnotation
end

@[MyAnnotation(1, 2, 3, 4)]
def annotation_args
  { { @def.annotation(MyAnnotation).args } }
end

annotation_args # => {1, 2, 3, 4}

```

Since the return type of `TupleLiteral` is iterable, we can rewrite the previous example in a better way. By extension, all of the [methods](#) on `TupleLiteral` are available for use as well.

```

annotation MyAnnotation
end

@[MyAnnotation(1, "foo", true, 17.0)]
def annotation_read
  {% for value, idx in @def.annotation(MyAnnotation).args %}
    pp "{{ idx }} = #{{ value }}"
  {% end %}
end

annotation_read

# Which would print
"0 = 1"
"1 = foo"
"2 = true"
"3 = 17.0"

```

## Reading

Annotations can be read off of a `TypeNode`, `Def`, or `MetaVar` using the `.annotation(type : TypeNode)` method. This method return an `Annotation` object representing the applied annotation of the supplied type.

**NOTE:** If multiple annotations of the same type are applied, the `.annotation` method will return the *last* one.

The `@type` and `@def` variables can be used to get a `TypeNode` or `Def` object to use the `.annotation` method on. However, it is also possible to get `TypeNode / Def` types using other methods on `TypeNode`. For example `TypeNode.all_subclasses` or `TypeNode.methods`, respectively.

The `TypeNode.instance_vars` can be used to get an array of instance variable `MetaVar` objects that would allow reading annotations defined on those instance variables.

**NOTE:** `TypeNode.instance_vars` currently only works in the context of an instance/class method.

```

annotation MyClass
end

annotation MyMethod
end

annotation MyIvar
end

@[MyClass]
class Foo
  pp {{ @type.annotation(MyClass).stringify }}

  @[MyIvar]
  @num : Int32 = 1

  @[MyIvar]
  property name : String = "jim"

  def properties
    {% for ivar in @type.instance_vars %}
      pp {{ ivar.annotation(MyIvar).stringify }}
    {% end %}
  end
end

@[MyMethod]
def my_method
  pp {{ @def.annotation(MyMethod).stringify }}
end

Foo.new.properties
my_method
pp {{ Foo.annotation(MyClass).stringify }}

# Which would print
"@[MyClass]"
"@[MyIvar]"
"@[MyIvar]"
"@[MyMethod]"
"@[MyClass]"

```

## Reading Multiple Annotations

If there are multiple annotations of the same type applied to the same instance variable/method/type, the `.annotations(type : TypeNode)` method can be used. This will work on anything that `.annotation(type : TypeNode)` would, but instead returns an `ArrayLiteral(Annotation)`.

```
annotation MyAnnotation
end

@[MyAnnotation("foo")]
@[MyAnnotation(123)]
@[MyAnnotation(123)]
def annotation_read
  {% for ann, idx in @def.annotations(MyAnnotation) %}
    pp "Annotation {{ idx }} = {{ ann[0].id }}"
  {% end %}
end

annotation_read

# Which would print
"Annotation 0 = foo"
"Annotation 1 = 123"
"Annotation 2 = 123"
```



The Crystal standard library includes some pre-defined annotations:

- [Link](#)
- [Extern](#)
- [ThreadLocal](#)
- [Packed](#)
- [AlwaysInline](#)
- [NoInline](#)
- [ReturnsTwice](#)
- [Raises](#)
- [CallConvention](#)
- [Flags](#)

## Link

Tells the compiler how to link a C library. This is explained in the [lib](#) section.

## Extern

Marking a Crystal struct with this attribute makes it possible to use it in lib declarations:

```
@[Extern]
struct MyStruct
end

lib MyLib
  fun my_func(s : MyStruct) # OK (gives an error without the Extern attribute)
end
```

You can also make a struct behave like a C union (this can be pretty unsafe):

```
# A struct to easily convert between Int32 codepoints and Chars
@[Extern(union: true)]
struct Int32OrChar
  property int = 0
  property char = '\0'
end

s = Int32OrChar.new
s.char = 'A'
s.int # => 65

s.int = 66
s.char # => 'B'
```

## ThreadLocal

The `@[ThreadLocal]` attribute can be applied to class variables and C external variables. It makes them be thread local.

```
class DontUseThis
  # One for each thread
  @[ThreadLocal]
  @@values = [] of Int32
end
```

ThreadLocal is used in the standard library to implement the runtime and shouldn't be needed or used outside it.

## Packed

Marks a [C struct](#) as packed, which prevents the automatic insertion of padding bytes between fields. This is typically only needed if the C library explicitly uses packed structs.

## AlwaysInline

Gives a hint to the compiler to always inline a method:

```
@[AlwaysInline]
def foo
  1
end
```

## NoInline

Tells the compiler to never inline a method call. This has no effect if the method yields, since functions which yield are always inlined.

```
@[NoInline]
def foo
  1
end
```

## ReturnsTwice

Marks a method or [lib fun](#) as returning twice. The C `setjmp` is an example of such a function.

## Raises

Marks a method or [lib fun](#) as potentially raising an exception. This is explained in the [callbacks](#) section.

## CallConvention

Indicates the call convention of a [lib fun](#). For example:

```
lib LibFoo
  @[CallConvention("X86_StdCall")]
  fun foo : Int32
end
```

The list of valid call conventions is:

- C (the default)
- Fast
- Cold
- WebKit\_JS
- AnyReg
- X86\_StdCall
- X86\_FastCall

They are explained [here](#).

## Flags

Marks an [enum](#) as a "flags enum", which changes the behaviour of some of its methods, like `to_s`.

## Low-level primitives

Some low-level primitives are provided. They are mostly useful for interfacing with C libraries and for low-level code.

## pointerof

The `pointerof` expression returns a [Pointer](#) that points to the contents of a variable or instance variable.

An example with a variable:

```
a = 1

ptr = pointerof(a)
ptr.value = 2

a # => 2
```

An example with an instance variable:

```
class Point
  def initialize(@x : Int32, @y : Int32)
    end

  def x
    @x
  end

  def x_ptr
    pointerof(@x)
  end
end

point = Point.new 1, 2

ptr = point.x_ptr
ptr.value = 10

point.x # => 10
```

Because `pointerof` involves pointers, it is considered [unsafe](#).

## sizeof

The `sizeof` expression returns an `Int32` with the size in bytes of a given type.

For example:

```
sizeof(Int32) # => 4
sizeof(Int64) # => 8
```

For [Reference](#) types, the size is the same as the size of a pointer:

```
# On a 64 bits machine
sizeof(Pointer(Int32)) # => 8
sizeof(String)        # => 8
```

This is because a Reference's memory is allocated on the heap and a pointer to it is passed around. To get the effective size of a class, use [instance\\_sizeof](#).

The argument to `sizeof` is a [type](#) and is often combined with [typeof](#):

```
a = 1
sizeof(typeof(a)) # => 4
```

## instance\_sizeof

The `instance_sizeof` expression returns an `Int32` with the instance size of a given class. For example:

```
class Point
  def initialize(@x, @y)
    end
end

Point.new 1, 2

# 2 x Int32 = 2 x 4 = 8
instance_sizeof(Point) # => 12
```

Even though the instance has two `Int32` fields, the compiler always includes an extra `Int32` field for the type id of the object. That's why the instance size ends up being 12 and not 8.

## offsetof

An `offsetof` expression returns the byte offset of an instance variable in a struct or class type. It accepts the type as first argument and the instance variable name prefixed by an `@` as second argument:

```
offsetof(Type, @ivar)
```

This is a low-level primitive and only useful if a C API needs to directly interface with the data layout of a Crystal type.

Example:

```
struct Foo
  @x = 0_i64
  @y = 34_i32
end

offsetof(Foo, @y) # => 8
```



## Uninitialized variable declaration

Crystal allows declaring uninitialized variables:

```
x = uninitialized Int32
x # => some random value, garbage, unreliable
```

This is [unsafe](#) code and is almost always used in low-level code for declaring uninitialized [StaticArray](#) buffers without a performance penalty:

```
buffer = uninitialized UInt8[256]
```

The buffer is allocated on the stack, avoiding a heap allocation.

The type after the `uninitialized` keyword follows the [type grammar](#).

## Compile-time flags

Types, methods and generally any part of your code can be conditionally defined based on some flags available at compile time. These flags are by default read from the hosts [LLVM Target Triple](#), split on `-`. To get the target you can execute

```
llvm-config --host-target .
```

```
$ llvm-config --host-target
x86_64-unknown-linux-gnu

# so the flags are: x86_64, unknown, linux, gnu
```

To define a flag, simply use the `--define` or `-D` option, like so:

```
crystal some_program.cr -Dflag
```

Additionally, if a program is compiled with `--release`, the `release` flag will be set.

You can check if a flag is defined with the `flag?` macro method:

```
{% if flag?(:x86_64) %}
  # some specific code for 64 bits platforms
{% else %}
  # some specific code for non-64 bits platforms
{% end %}
```

`flag?` returns a boolean, so you can use it with `&&` and `||`:

```
{% if flag?(:linux) && flag?(:x86_64) %}
  # some specific code for linux 64 bits
{% end %}
```

These flags are generally used in C bindings to conditionally define types and functions. For example, the very well known `size_t` type is defined like this in Crystal:

```
lib C
  {% if flag?(:x86_64) %}
    alias SizeT = UInt64
  {% else %}
    alias SizeT = UInt32
  {% end %}
end
```

## Cross-compilation

Crystal supports a basic form of [cross compilation](#).

In order to achieve this, the compiler executable provides two flags:

- `--cross-compile` : When given enables cross compilation mode
- `--target` : the [LLVM Target Triple](#) to use and set the default [compile-time flags](#) from

To get the `--target` flags you can execute `llvm-config --host-target` using an installed LLVM 3.5. For example on a Linux it could say "x86\_64-unknown-linux-gnu".

If you need to set any compile-time flags not set implicitly through `--target`, you can use the `-D` command line flag.

Using these two, we can compile a program in a Mac that will run on that Linux like this:

```
crystal build your_program.cr --cross-compile --target "x86_64-unknown-linux-gnu"
```

This will generate a `.o` ([Object file](#)) and will print a line with a command to execute on the system we are trying to cross-compile to. For example:

```
cc your_program.o -o your_program -lpcrc -lrt -lm -lgc -lunwind
```

You must copy this `.o` file to that system and execute those commands. Once you do this the executable will be available in that target system.

This procedure is usually done with the compiler itself to port it to new platforms where a compiler is not yet available. Because in order to compile a Crystal compiler we need an older Crystal compiler, the only two ways to generate a compiler for a system where there isn't a compiler yet are:

- We checkout the latest version of the compiler written in Ruby, and from that compiler we compile the next versions until the current one.
- We create a `.o` file in the target system and from that file we create a compiler.

The first alternative is long and cumbersome, while the second one is much easier.

Cross-compiling can be done for other executables, but its main target is the compiler. If Crystal isn't available in some system you can try cross-compiling it there.

## C bindings

Crystal allows you to bind to existing C libraries without writing a single line in C.

Additionally, it provides some conveniences like `out` and `to_unsafe` so writing bindings is as painless as possible.

## lib

A `lib` declaration groups C functions and types that belong to a library.

```
@[Link("pcre")]
lib LibPCRE
end
```

Although not enforced by the compiler, a `lib`'s name usually starts with `Lib`.

Attributes are used to pass flags to the linker to find external libraries:

- `@[Link("pcre")]` will pass `-lpcre` to the linker, but the compiler will first try to use [pkg-config](#).
- `@[Link(ldflags: "...")]` will pass those flags directly to the linker, without modification. For example: `@[Link(ldflags: "-lpcre")]`. A common technique is to use backticks to execute commands: `@[Link(ldflags: "`pkg-config libpcre --libs`")]`.
- `@[Link(framework: "Cocoa")]` will pass `-framework Cocoa` to the linker (only useful in macOS).

Attributes can be omitted if the library is implicitly linked, as in the case of `libc`.

## fun

A `fun` declaration inside a `lib` binds to a C function.

```
lib C
  # In C: double cos(double x)
  fun cos(value : Float64) : Float64
end
```

Once you bind it, the function is available inside the `C` type as if it was a class method:

```
C.cos(1.5) # => 0.0707372
```

You can omit the parentheses if the function doesn't have arguments (and omit them in the call as well):

```
lib C
  fun getch : Int32
end

C.getch
```

If the return type is void you can omit it:

```
lib C
  fun srand(seed : UInt32)
end

C.srand(1_u32)
```

You can bind to variadic functions:

```
lib X
  fun variadic(value : Int32, ...) : Int32
end

X.variadic(1, 2, 3, 4)
```

Note that there are no implicit conversions (except `to_unsafe`, which is explained later) when invoking a C function: you must pass the exact type that is expected. For integers and floats you can use the various `to_...` methods.

## Function names

Function names in a `lib` definition can start with an upper case letter. That's different from methods and function definitions outside a `lib`, which must start with a lower case letter.

Function names in Crystal can be different from the C name. The following example shows how to bind the C function name `SDL_Init` as `LibSDL.init` in Crystal.

```
lib LibSDL
  fun init = SDL_Init(flags : UInt32) : Int32
end
```

The C name can be put in quotes to be able to write a name that is not a valid identifier:

```
lib LLVMIntrinsics
  fun ceil_f32 = "llvm.ceil.f32"(value : Float32) : Float32
end
```

This can also be used to give shorter, nicer names to C functions, as these tend to be long and are usually prefixed with the library name.

## Types in C Bindings

The valid types to use in C bindings are:

- Primitive types ( `Int8` , ..., `Int64` , `UInt8` , ..., `UInt64` , `Float32` , `Float64` )
- Pointer types ( `Pointer(Int32)` , which can also be written as `Int32*` )
- Static arrays ( `StaticArray(Int32, 8)` , which can also be written as `Int32[8]` )
- Function types ( `Function(Int32, Int32)` , which can also be written as `Int32 -> Int32` )
- Other `struct` , `union` , `enum` , `type` or `alias` declared previously.
- `Void` : the absence of a return value.
- `NoReturn` : similar to `Void` , but the compiler understands that no code can be executed after that invocation.
- Crystal structs marked with the `@[Extern]` attribute

Refer to the [type grammar](#) for the notation used in fun types.

The standard library defines the `LibC` lib with aliases for common C types, like `int` , `short` , `size_t` . Use them in bindings like this:

```
lib MyLib
  fun my_fun(some_size : LibC::SizeT)
end
```

**Note:** The C `char` type is `UInt8` in Crystal, so a `char*` or a `const char*` is `UInt8*` . The `char` type in Crystal is a unicode codepoint so it is represented by four bytes, making it similar to an `Int32` , not to an `UInt8` . There's also the alias `LibC::Char` if in doubt.

## out

Consider the `waitpid` function:

```
lib C
  fun waitpid(pid : Int32, status_ptr : Int32*, options : Int32) : Int32
end
```

The documentation of the function says:

```
The status information from the child process is stored in the object
that status_ptr points to, unless status_ptr is a null pointer.
```

We can use this function like this:

```
status_ptr = uninitialized Int32
C.waitpid(pid, pointerof(status_ptr), options)
```

In this way we pass a pointer of `status_ptr` to the function for it to fill its value.

There's a simpler way to write the above by using an `out` parameter:

```
C.waitpid(pid, out status_ptr, options)
```

The compiler will automatically declare a `status_ptr` variable of type `Int32`, because the argument is an `Int32*`.

This will work for any type, as long as the argument is a pointer of that type (and, of course, as long as the function does fill the value the pointer is pointing to).



## to\_unsafe

If a type defines a `to_unsafe` method, when passing it to C the value returned by this method will be passed. For example:

```
lib C
  fun exit(status : Int32) : NoReturn
end

class IntWrapper
  def initialize(@value)
  end

  def to_unsafe
    @value
  end
end

wrapper = IntWrapper.new(1)
C.exit(wrapper) # wrapper.to_unsafe is passed to C function which has type Int32
```

This is very useful for defining wrappers of C types without having to explicitly transform them to their wrapped values.

For example, the `String` class implements `to_unsafe` to return `UInt8*`:

```
lib C
  fun printf(format : UInt8*, ...) : Int32
end

a = 1
b = 2
C.printf "%d + %d = %d\n", a, b, a + b
```

## struct

A `struct` declaration inside a `lib` declares a C struct.

```
lib C
  # In C:
  #
  # struct TimeZone {
  #   int minutes_west;
  #   int dst_time;
  # };
  struct TimeZone
    minutes_west : Int32
    dst_time : Int32
  end
end
```

You can also specify many fields of the same type:

```
lib C
  struct TimeZone
    minutes_west, dst_time : Int32
  end
end
```

Recursive structs work just like you expect them to:

```
lib C
  struct LinkedListNode
    prev, _next : LinkedListNode*
  end

  struct LinkedList
    head : LinkedListNode*
  end
end
```

To create an instance of a struct use `new` :

```
tz = C::TimeZone.new
```

This allocates the struct on the stack.

A C struct starts with all its fields set to "zero": integers and floats start at zero, pointers start with an address of zero, etc.

To avoid this initialization you can use `uninitialized` :

```
tz = uninitialized C::TimeZone
tz.minutes_west # => some garbage value
```

You can set and get its properties:

```
tz = C::TimeZone.new
tz.minutes_west = 1
tz.minutes_west # => 1
```

If the assigned value is not exactly the same as the property's type, `to_unsafe` will be tried.

You can also initialize some fields with a syntax similar to [named arguments](#):

```
tz = C::TimeZone.new minutes_west: 1, dst_time: 2
tz.minutes_west # => 1
tz.dst_time     # => 2
```

A C struct is passed by value (as a copy) to functions and methods, and also passed by value when it is returned from a method:

```
def change_it(tz)
  tz.minutes_west = 1
end

tz = C::TimeZone.new
change_it tz
tz.minutes_west # => 0
```

Refer to the [type grammar](#) for the notation used in struct field types.

## union

A `union` declaration inside a `lib` declares a C union:

```
lib U
  # In C:
  #
  # union IntOrFloat {
  #   int some_int;
  #   double some_float;
  # };
  union IntOrFloat
    some_int : Int32
    some_float : Float64
  end
end
```

To create an instance of a union use `new` :

```
value = U::IntOrFloat.new
```

This allocates the union on the stack.

A C union starts with all its fields set to "zero": integers and floats start at zero, pointers start with an address of zero, etc.

To avoid this initialization you can use `uninitialized` :

```
value = uninitialized U::IntOrFloat
value.some_int # => some garbage value
```

You can set and get its properties:

```
value = U::IntOrFloat.new
value.some_int = 1
value.some_int # => 1
value.some_float # => 4.94066e-324
```

If the assigned value is not exactly the same as the property's type, `to_unsafe` will be tried.

A C union is passed by value (as a copy) to functions and methods, and also passed by value when it is returned from a method:

```
def change_it(value)
  value.some_int = 1
end

value = U::IntOrFloat.new
change_it value
value.some_int # => 0
```

Refer to the [type grammar](#) for the notation used in union field types.

## enum

An `enum` declaration inside a `lib` declares a C enum:

```
lib X
# In C:
#
# enum SomeEnum {
#   Zero,
#   One,
#   Two,
#   Three,
# };
enum SomeEnum
  Zero
  One
  Two
  Three
end
end
```

As in C, the first member of the enum has a value of zero and each successive value is incremented by one.

To use a value:

```
X::SomeEnum::One # => One
```

You can specify the value of a member:

```
lib X
enum SomeEnum
  Ten      = 10
  Twenty   = 10 * 2
  ThirtyTwo = 1 << 5
end
end
```

As you can see, some basic math is allowed for a member value: `+`, `-`, `*`, `/`, `&`, `|`, `<<`, `>>` and `%`.

The type of an enum member is `Int32` by default, even if you specify a different type in a constant value:

```
lib X
enum SomeEnum
  A = 1_u32
end
end

X::SomeEnum # => 1_i32
```

However, you can change this default type:

Nil

```
lib X
  enum SomeEnum : Int8
    Zero
    Two = 2
  end
end

X::SomeEnum::Zero # => 0_i8
X::SomeEnum::Two  # => 2_i8
```

You can use an enum as a type in a `fun` argument or `struct` or `union` members:

```
lib X
  enum SomeEnum
    One
    Two
  end

  fun some_fun(value : SomeEnum)
  end
```

## Variables

Variables exposed by a C library can be declared inside a `lib` declaration using a global-variable-like declaration:

```
lib C
  $errno : Int32
end
```

Then it can be get and set:

```
C.errno # => some value
C.errno = 0
C.errno # => 0
```

A variable can be marked as thread local with an attribute:

```
lib C
  @[ThreadLocal]
  $errno : Int32
end
```

Refer to the [type grammar](#) for the notation used in external variables types.

## Constants

You can also declare constants inside a `lib` declaration:

```
@[Link("pcre")]  
lib PCRE  
    INFO_CAPTURECOUNT = 2  
end  
  
PCRE::INFO_CAPTURECOUNT # => 2
```



## type

A `type` declaration inside a `lib` declares a kind of C `typedef`, but stronger:

```
lib X
  type MyInt = Int32
end
```

Unlike C, `Int32` and `MyInt` are not interchangeable:

```
lib X
  type MyInt = Int32

  fun some_fun(value : MyInt)
  end

X.some_fun 1 # Error: argument 'value' of 'X#some_fun' must be X::MyInt, not Int32
```

Thus, a `type` declaration is useful for opaque types that are created by the C library you are wrapping. An example of this is the C `FILE` type, which you can obtain with `fopen`.

Refer to the [type grammar](#) for the notation used in typedef types.

## alias

An `alias` declaration inside a `lib` declares a C `typedef` :

```
lib X
  alias MyInt = Int32
end
```

Now `Int32` and `MyInt` are interchangeable:

```
lib X
  alias MyInt = Int32

  fun some_fun(value : MyInt)
  end

X.some_fun 1 # OK
```

An `alias` is most useful to avoid writing long types over and over, but also to declare a type based on compile-time flags:

```
lib C
  {% if flag?(:x86_64) %}
    alias SizeT = Int64
  {% else %}
    alias SizeT = Int32
  {% end %}

  fun memcmp(p1 : Void*, p2 : Void*, size : C::SizeT) : Int32
  end
```

Refer to the [type grammar](#) for the notation used in alias types.

## Callbacks

You can use function types in C declarations:

```
lib X
  # In C:
  #
  # void callback(int (*f)(int));
  fun callback(f : Int32 -> Int32)
end
```

Then you can pass a function (a [Proc](#)) like this:

```
f = ->(x : Int32) { x + 1 }
X.callback(f)
```

If you define the function inline in the same call you can omit the argument types, the compiler will add the types for you based on the `fun` signature:

```
X.callback ->(x) { x + 1 }
```

Note, however, that functions passed to C can't form closures. If the compiler detects at compile-time that a closure is being passed, an error will be issued:

```
y = 2
X.callback ->(x) { x + y } # Error: can't send closure to C function
```

If the compiler can't detect this at compile-time, an exception will be raised at runtime.

Refer to the [type grammar](#) for the notation used in callbacks and procs types.

If you want to pass `NULL` instead of a callback, just pass `nil` :

```
# Same as callback(NULL) in C
X.callback nil
```

## Passing a closure to a C function

Most of the time a C function that allows setting a callback also provide an argument for custom data. This custom data is then sent as an argument to the callback. For example, suppose a C function that invokes a callback at every tick, passing that tick:

```
lib LibTicker
  fun on_tick(callback : (Int32, Void* ->), data : Void*)
end
```

To properly define a wrapper for this function we must send the Proc as the callback data, and then convert that callback data to the Proc and finally invoke it.

```

module Ticker
  # The callback for the user doesn't have a Void*
  @@box : Pointer(Void)?

  def self.on_tick(&callback : Int32 ->)
    # Since Proc is a {Void*, Void*}, we can't turn that into a Void*, so we
    # "box" it: we allocate memory and store the Proc there
    boxed_data = Box.box(callback)

    # We must save this in Crystal-land so the GC doesn't collect it (*)
    @@box = boxed_data

    # We pass a callback that doesn't form a closure, and pass the boxed_data as
    # the callback data
    LibTicker.on_tick(->(tick, data) {
      # Now we turn data back into the Proc, using Box.unbox
      data_as_callback = Box(typeof(callback)).unbox(data)
      # And finally invoke the user's callback
      data_as_callback.call(tick)
    }, boxed_data)
  end
end

Ticker.on_tick do |tick|
  puts tick
end

```

Note that we save the boxed callback in `@@box`. The reason is that if we don't do it, and our code doesn't reference it anymore, the GC will collect it. The C library will of course store the callback, but Crystal's GC has no way of knowing that.

## Raises attribute

If a C function executes a user-provided callback that might raise, it must be annotated with the `@[Raises]` attribute.

The compiler infers this attribute for a method if it invokes a method that is marked as `@[Raises]` or `raises` (recursively).

However, some C functions accept callbacks to be executed by other C functions. For example, suppose a fictitious library:

```

lib LibFoo
  fun store_callback(callback : ->)
  fun execute_callback
end

LibFoo.store_callback ->{ raise "OH NO!" }
LibFoo.execute_callback

```

If the callback passed to `store_callback` raises, then `execute_callback` will raise. However, the compiler doesn't know that `execute_callback` can potentially raise because it is not marked as `@[Raises]` and the compiler has no way to figure this out. In these cases you have to manually mark such functions:

Nil

```
lib LibFoo
  fun store_callback(callback : ->)

  @[Raises]
  fun execute_callback
end
```

If you don't mark them, `begin/rescue` blocks that surround this function's calls won't work as expected.

## Unsafe code

These parts of the language are considered unsafe:

- Code involving raw pointers: the [Pointer](#) type and [pointerof](#).
- The [allocate](#) class method.
- Code involving C bindings
- [Uninitialized variable declaration](#)

"Unsafe" means that memory corruption, segmentation faults and crashes are possible to achieve. For example:

```
a = 1
ptr = pointerof(a)
ptr[100_000] = 2 # undefined behaviour, probably a segmentation fault
```

However, regular code usually never involves pointer manipulation or uninitialized variables. And C bindings are usually wrapped in safe wrappers that include null pointers and bounds checks.

No language is 100% safe: some parts will inevitably be low-level, interface with the operating system and involve pointer manipulation. But once you abstract that and operate on a higher level, and assume (after mathematical proof or thorough testing) that the lower grounds are safe, you can be confident that your entire codebase is safe.

## Conventions

Follow these conventions so your code will be more accessible to other developers.

- Use [standard coding style](#) so your project will be navigable and readable to others.
- Write [documentation](#) to express the purpose of your code and support the `crystal docs` generator.

## Coding Style

This style is used in the standard library. You can use it in your own project to make it familiar to other developers.

## Naming

**Type names** are camelcased. For example:

```
class ParseError < Exception
end

module HTTP
  class RequestHandler
    end
  end
end

alias NumericValue = Float32 | Float64 | Int32 | Int64

lib LibYAML
end

struct TagDirective
end

enum Time::DayOfWeek
end
```

**Method names** are underscore-cased. For example:

```
class Person
  def first_name
  end

  def date_of_birth
  end

  def homepage_url
  end
end
```

**Variable names** are underscore-cased. For example:

```
class Greeting
  @@default_greeting = "Hello world"

  def initialize(@custom_greeting = nil)
  end

  def print_greeting
    greeting = @custom_greeting || @@default_greeting
    puts greeting
  end
end
```



**Constants** are screaming-cased. For example:

```
LUCKY_NUMBERS = [3, 7, 11]
DOCUMENTATION_URL = "http://crystal-lang.org/docs"
```

## Acronyms

In class names, acronyms are *all-uppercase*. For example, `HTTP`, and `LibXML`.

In method names, acronyms are *all-lowercase*. For example `#from_json`, `#to_io`.

## Libs

`Lib` names are prefixed with `Lib`. For example: `LibC`, `LibEvent2`.

## Directory and File Names

Within a project:

- `/` contains a readme, any project configurations (eg, CI or editor configs), and any other project-level documentation (eg, changelog or contributing guide).
- `src/` contains the project's source code.
- `spec/` contains the [project's specs](#), which can be run with `crystal spec`.
- `bin/` contains any executables.

File paths match the namespace of their contents. Files are named after the class or namespace they define, with *underscore-case*.

For example, `HTTP::WebSocket` is defined in `src/http/web_socket.cr`.

## Whitespace

Use **two spaces** to indent code inside namespaces, methods, blocks or other nested contexts. For example:

```
module Scorecard
  class Parser
    def parse(score_text)
      begin
        score_text.scan(SCORE_PATTERN) do |match|
          handle_match(match)
        end
      rescue err : ParseError
        # handle error ...
      end
    end
  end
end
```

Within a class, separate method definitions, constants and inner class definitions with **one newline**. For example:

```
module Money
  CURRENCIES = {
    "EUR" => 1.0,
    "ARS" => 10.55,
    "USD" => 1.12,
    "JPY" => 134.15,
  }

  class Amount
    getter :currency, :value

    def initialize(@currency, @value)
      end
    end

  class CurrencyConversion
    def initialize(@amount, @target_currency)
      end

    def amount
      # implement conversion ...
    end
  end
end
```

## Documenting code

Crystal can generate documentation from comments using a subset of [Markdown](#).

To generate documentation for a project, invoke `crystal docs`. By default this will create a `docs` directory, with a `docs/index.html` entry point. For more details see [Using the compiler – Creating documentation](#).

- Documentation should be positioned right above definitions of classes, modules, and methods. Leave no blanks between them.

```
# A unicorn is a legendary animal (see the `Legendary` module) that has been
# described since antiquity as a beast with a large, spiraling horn projecting
# from its forehead.
class Unicorn
end

# Bad: This is not attached to any class.

class Legendary
end
```

- The documentation of a method is included into the method summary and the method details. The former includes only the first line, the latter includes the entire documentation. In short, it is preferred to:
  1. State a method's purpose or functionality in the first line.
  2. Supplement it with details and usages after that.

For instance:

```
# Returns the number of horns this unicorn has.
#
# ---
# Unicorn.new.horns # => 1
# ---
def horns
  @horns
end
```

- Use the third person: `Returns the number of horns this unicorn has` instead of `Return the number of horns this unicorn has`.
- Parameter names should be *italicized* (surrounded with single asterisks `*` or underscores `_`):

```
# Creates a unicorn with the specified number of *horns*.
def initialize(@horns = 1)
  raise "Not a unicorn" if @horns != 1
end
```

- Code blocks that have Crystal code can be surrounded with triple backticks or indented with four spaces.

```
# ```
# unicorn = Unicorn.new
# unicorn.speak
# ```
```

or

```
# unicorn = Unicorn.new
# unicorn.speak
```

- Text blocks, for example to show program output, must be surrounded with triple backticks followed by the "text" keyword.

```
# ```text
# "I'm a unicorn"
# ```
```

- To automatically link to other types, enclose them with single backticks.

```
# the `Legendary` module
```

- To automatically link to methods of the currently documented type, use a hash like `#horns` or `#index(char)`, and enclose it with single backticks.
- To automatically link to methods in other types, do `OtherType#method(arg1, arg2)` or just `OtherType#method`, and enclose it with single backticks.

For example:

```
# Check the number of horns with `#horns`.
# See what a unicorn would say with `Unicorn#speak`.
```

- To show the value of an expression inside code blocks, use `# =>`.

```
1 + 2 # => 3
Unicorn.new.speak # => "I'm a unicorn"
```

- Use `:ditto:` to use the same comment as in the previous declaration.

```
# :ditto:
def number_of_horns
  horns
end
```

- Use `:nodoc:` to hide public declarations from the generated documentation. Private and protected methods are always hidden.

```
class Unicorn
  # :nodoc:
  class Helper
    end
end
```

## Inheriting Documentation

When an instance method has no doc comment, but a method with the same signature exists in a parent type, the documentation is inherited from the parent method.

For example:

```
abstract class Animal
  # Returns the name of `self`.
  abstract def name : String
end

class Unicorn < Animal
  def name : String
    "unicorn"
  end
end
```

The documentation for `Unicorn#name` would be:

```
Description copied from class `Animal`

Returns the name of `self`.
```

The child method can use `:inherit:` to explicitly copy the parent's documentation, without the `Description copied from ... text`. `:inherit:` can also be used to inject the parent's documentation into additional documentation on the child.

For example:

```
abstract class Parent
  # Some documentation common to every *id*.
  abstract def id : Int32
end

class Child < Parent
  # Some documentation specific to *id*'s usage within `Child`.
  #
  # :inherit:
  def id : Int32
    -1
  end
end
```

The documentation for `Child#id` would be:

```
Some documentation specific to *id*'s usage within `Child`.

Some documentation common to every *id*.
```

**NOTE:** Inheriting documentation only works on *instance*, non-constructor methods.

## Flagging Classes, Modules, and Methods

Given a valid keyword, Crystal will automatically generate visual flags that help highlight problems, notes and/or possible issues.

The supported flag keywords are:

- BUG
- DEPRECATED
- FIXME
- NOTE
- OPTIMIZE
- TODO

Flag keywords must be the first word in their respective line and must be in all caps. An optional trailing colon is preferred for readability.

```
# Makes the unicorn speak to STDOUT
#
# NOTE: Although unicorns don't normally talk, this one is special
# TODO: Check if unicorn is asleep and raise exception if not able to speak
# TODO: Create another `speak` method that takes and prints a string
def speak
  puts "I'm a unicorn"
end

# Makes the unicorn talk to STDOUT
#
# DEPRECATED: Use `speak`
def talk
  puts "I'm a unicorn"
end
```

## Use Crystal's code formatter

Crystal's built-in code formatter can be used not just to format your code, but also to format code samples included in documentation blocks.

This is done automatically when `crystal tool format` is invoked, which will automatically format all `.cr` files in current directory.

To format a single file:

```
$ crystal tool format file.cr
```

To format all `.cr` files within a directory:

```
$ crystal tool format src/
```

Use this tool to unify code styles and to submit documentation improvements to Crystal itself.

The formatter is also fast, so very little time is lost if you format the entire project instead of a single file.

## A Complete Example

```
# A unicorn is a legendary animal (see the `Legendary` module) that has been
# described since antiquity as a beast with a large, spiraling horn projecting
# from its forehead.
#
# To create a unicorn:
#
# ```
# unicorn = Unicorn.new
# unicorn.speak
# ```
#
# The above produces:
#
# ```text
# "I'm a unicorn"
# ```
#
# Check the number of horns with `#horns`.
class Unicorn
  include Legendary

  # Creates a unicorn with the specified number of *horns*.
  def initialize(@horns = 1)
    raise "Not a unicorn" if @horns != 1
  end

  # Returns the number of horns this unicorn has
  #
  # ```
  # Unicorn.new.horns # => 1
  # ```
  def horns
    @horns
  end

  # :ditto:
  def number_of_horns
    horns
  end

  # Makes the unicorn speak to STDOUT
  def speak
    puts "I'm a unicorn"
  end

  # :nodoc:
  class Helper
    end
end
```

## Database

To access a relational database you will need a shard designed for the database server you want to use. The package [crystal-lang/crystal-db](#) offers a unified api across different drivers.

The following packages are compliant with crystal-db

- [crystal-lang/crystal-sqlite3](#) for sqlite
- [crystal-lang/crystal-mysql](#) for mysql & mariadb
- [will/crystal-pg](#) for postgres

This guide presents the api of crystal-db, the sql commands might need to be adapted for the concrete driver due to differences between postgres, mysql and sqlite.

Also some drivers may offer additional functionality like postgres `LISTEN / NOTIFY` .

## Installing the shard

Choose the appropriate driver from the list above and add it as any shard to your application's `shard.yml`

There is no need to explicitly require `crystal-lang/crystal-db`

During this guide `crystal-lang/crystal-mysql` will be used.

```
dependencies:
  mysql:
    github: crystal-lang/crystal-mysql
```

## Open database

`DB.open` will allow you to easily connect to a database using a connection uri. The schema of the uri determines the expected driver. The following sample connects to a local mysql database named `test` with user `root` and password blank.

```
require "db"
require "mysql"

DB.open "mysql://root@localhost/test" do |db|
  # ... use db to perform queries
end
```

Other connection uris are

- `sqlite3:///path/to/data.db`
- `mysql://user:password@server:port/database`
- `postgres://server:port/database`



Alternatively you can use a non yielding `DB.open` method as long as `Database#close` is called at the end.

```
require "db"
require "mysql"

db = DB.open "mysql://root@localhost/test"
begin
  # ... use db to perform queries
ensure
  db.close
end
```

## Exec

To execute sql statements you can use `Database#exec`

```
db.exec "create table contacts (name varchar(30), age int)"
```

To avoid [SQL injection](#) values can be provided as query parameters. The syntax for using query parameters depends on the database driver because they are typically just passed through to the database. MySQL uses `?` for parameter expansion and assignment is based on argument order. PostgreSQL uses `$n` where `n` is the ordinal number of the argument (starting with 1).

```
# MySQL
db.exec "insert into contacts values (?, ?)", "John", 30
# Postgres
db.exec "insert into contacts values ($1, $2)", "Sarah", 33
```

## Query

To perform a query and get the result set use `Database#query`, arguments can be used as in `Database#exec`.

`Database#query` returns a `ResultSet` that needs to be closed. As in `Database#open`, if called with a block, the `ResultSet` will be closed implicitly.

```
db.query "select name, age from contacts order by age desc" do |rs|
  rs.each do
    # ... perform for each row in the ResultSet
  end
end
```

When reading values from the database there is no type information during compile time that crystal can use. You will need to call `rs.read(T)` with the type `T` you expect to get from the database.

```

db.query "select name, age from contacts order by age desc" do |rs|
  rs.each do
    name = rs.read(String)
    age = rs.read(Int32)
    puts "#{name} (#{age})"
    # => Sarah (33)
    # => John Doe (30)
  end
end

```

There are many convenient query methods built on top of `#query` .

You can read multiple columns at once:

```

name, age = rs.read(String, Int32)

```

Or read a single row:

```

name, age = db.query_one "select name, age from contacts order by age desc limit 1", &

```

Or read a scalar value without dealing explicitly with the ResultSet:

```

max_age = db.scalar "select max(age) from contacts"

```

All available methods to perform statements in a database are defined in

```

DB::QueryMethods .

```

## Connection

A connection is one of the key parts when working with databases. It represents the *runway* through which statements travel from our application to the database.

In Crystal we have two ways of building this connection. And so, coming up next, we are going to present examples with some advice on when to use each one.

## DB module

*Give me a place to stand, and I shall move the earth.*

Archimedes

The DB module, is our place to stand when working with databases in Crystal. As written in the documentation: *is a unified interface for database access*.

One of the methods implemented in this module is `DB#connect`. Using this method is the **first way** for creating a connection. Let's see how to use it.

## DB#connect

When using `DB#connect` we are indeed opening a connection to the database. The `uri` passed as the parameter is used by the module to determine which driver to use (for example: `mysql://`, `postgres://`, `sqlite://`, etc.) i.e. we do not need to specify which database we are using.

The `uri` for this example is `mysql://root:root@localhost/test`, and so the module will use the `mysql driver` to connect to the MySQL database.

Here is the example:

```
require "mysql"

cnn = DB.connect("mysql://root:root@localhost/test")
puts typeof(cnn) # => DB::Connection
cnn.close
```

It's worth mentioning that the method returns a `DB::Connection` object. Although more specifically, it returns a `MySQL::Connection` object, it doesn't matter because all types of connections should be polymorphic. So hereinafter we will work with a `DB::Connection` instance, helping us to abstract from specific issues of each database engine.

When creating a connection *manually* (as we are doing here) we are responsible for managing this resource, and so we must close the connection when we are done using it. Regarding the latter, this little details can be the cause of huge bugs! Crystal, being a *language for humans*, give us a more safe way of *manually* creating a connection using blocks, like this:

```
require "mysql"

DB.connect "mysql://root:root@localhost/test" do |cnn|
  puts typeof(cnn) # => DB::Connection
end # the connection will be closed here
```

Ok, now we have a connection, let's use it!

```
require "mysql"

DB.connect "mysql://root:root@localhost/test" do |cnn|
  puts typeof(cnn) # => DB::Connection
  puts "Connection closed: #{cnn.closed?}" # => false

  result = cnn.exec("drop table if exists contacts")
  puts result

  result = cnn.exec("create table contacts (name varchar(30), age int)")
  puts result

  cnn.transaction do |tx|
    cnn2 = tx.connection
    puts "Yep, it is the same connection! #{cnn == cnn2}"

    cnn2.exec("insert into contacts values ('Joe', 42)")
    cnn2.exec("insert into contacts values (?, ?)", "Sarah", 43)
  end

  cnn.query_each "select * from contacts" do |rs|
    puts "name: #{rs.read}, age: #{rs.read}"
  end
end
```

First, in this example, we are using a transaction (check the [transactions](#) section for more information on this topic) Second, it's important to notice that the connection given by the transaction **is the same connection** that we were working with, before the transaction begin. That is, there is only **one** connection at all times in our program. And last, we are using the method `#exec` and `#query`. You may read more about executing queries in the [database](#) section.

Now that we have a good idea about creating a connection, let's present the **second way** for creating one: `DB#open`

## DB#open

```
require "mysql"

db = DB.open("mysql://root:root@localhost/test")
puts typeof(db) # DB::Database
db.close
```

As with a connection, we should close the database once we don't need it anymore. Or instead, we could use a block and let Crystal close the database for us!

But, where is the connection? Well, we should be asking for the **connections**. When a database is created, a pool of connections is created with connections to the database prepared and ready to use! (Do you want to read more about **pool of connections**? In the [connection pool](#) section you may read all about this interesting topic!)

How do we use a connection from the `database` object? For this, we could ask the database for a connection using the method `Database#checkout`. But, doing this will require to explicitly return the connection to the pool using `Connection#release`. Here is an example:

```
require "mysql"

DB.open "mysql://root:root@localhost/test" do |db|
  cnn = db.checkout
  puts typeof(cnn)

  puts "Connection closed: #{cnn.closed?}" # => false
  cnn.release
  puts "Connection closed: #{cnn.closed?}" # => false
end
```

And we want a *safe* way (i.e. no need for us to release the connection) to request and use a connection from the `database`, we could use `Database#using_connection`:

```
require "mysql"

DB.open "mysql://root:root@localhost/test" do |db|
  db.using_connection do |cnn|
    puts typeof(cnn)
    # use cnn
  end
end
```

In the next example we will let the `database` object *manage the connections by itself*, like this:

```
require "mysql"

DB.open "mysql://root:root@localhost/test" do |db|
  db.exec("drop table if exists contacts")
  db.exec("create table contacts (name varchar(30), age int)")

  db.transaction do |tx|
    cnn = tx.connection
    cnn.exec("insert into contacts values ('Joe', 42)")
    cnn.exec("insert into contacts values (?, ?)", "Sarah", 43)
  end

  db.query_each "select * from contacts" do |rs|
    puts "name: #{rs.read}, age: #{rs.read}"
  end
end
```

As we may notice, the `database` is polymorphic with a `connection` object with regard to the `#exec` / `#query` / `#transaction` methods. The database is responsible for the use of the connections. Great!

## When to use one or the other?

Given the examples, it may come to our attention that **the number of connections is relevant**. If we are programming a short living application with only one user starting requests to the database then a single connection managed by us (i.e. a `DB::Connection` object) should be enough (think of a command line application that receives parameters, then starts a request to the database and finally displays the result to the user) On the other hand, if we are building a system with many concurrent users and with heavy database access, then we should use a `DB::Database` object; which by using a connection pool will have a number of connections already prepared and ready to use (no bootstrap/initialization-time penalizations). Or imagine that you are building a long-living application (like a background job) then a connection pool will free you from the responsibility of monitoring the state of the connection: is it alive or does it need to reconnect?

## Connection pool

When a connection is established it usually means opening a TCP connection or Socket. The socket will handle one statement at a time. If a program needs to perform many queries simultaneously, or if it handles concurrent requests that aim to use a database, it will need more than one active connection.

Since databases are separate services from the application using them, the connections might go down, the services might be restarted, and other sort of things the program might not want to care about.

To address this issues usually a connection pool is a neat solution.

When a database is opened with `crystal-db` there is already a connection pool working. `DB.open` returns a `DB::Database` object which manages the whole connection pool and not just a single connection.

```
DB.open("mysql://root@localhost/test") do |db|
  # db is a DB::Database
end
```

When executing statements using `db.query`, `db.exec`, `db.scalar`, etc. the algorithm goes:

1. Find an available connection in the pool.
  - i. Create one if needed and possible.
  - ii. If the pool is not allowed to create a new connection, wait a for a connection to become available.
    - i. But this wait should be aborted if it takes too long.
2. Checkout that connection from the pool.
3. Execute the SQL command.
4. If there is no `DB::ResultSet` yielded, return the connection to the pool. Otherwise, the connection will be returned to the pool when the `ResultSet` is closed.
5. Return the statement result.

If a connection can't be created, or if a connection loss occurs while the statement is performed the above process is repeated.

The retry logic only happens when the statement is sent through the `DB::Database`. If it is sent through a `DB::Connection` or `DB::Transaction` no retry is performed since the code will state that certain connection object was expected to be used.

## Configuration

The behavior of the pool can be configured from a set of parameters that can appear as query string in the connection URI.

Name	Default value
initial_pool_size	1
max_pool_size	0 (unlimited)
max_idle_pool_size	1
checkout_timeout	5.0 (seconds)
retry_attempts	1
retry_delay	1.0 (seconds)

When `DB::Database` is opened an initial number of `initial_pool_size` connections will be created. The pool will never hold more than `max_pool_size` connections. When returning/releasing a connection to the pool it will be closed if there are already `max_idle_pool_size` idle connections.

If the `max_pool_size` was reached and a connection is needed, wait up to `checkout_timeout` seconds for an existing connection to become available.

If a connection is lost or can't be established retry at most `retry_attempts` times waiting `retry_delay` seconds between each try.

## Sample

The following program will print the current time from MySQL but if the connection is lost or the whole server is down for a few seconds the program will still run without raising exceptions.

```
# file: sample.cr
require "mysql"

DB.open "mysql://root@localhost?retry_attempts=8&retry_delay=3" do |db|
  loop do
    pp db.scalar("SELECT NOW()")
    sleep 0.5
  end
end
```

```
$ crystal sample.cr
db.scalar("SELECT NOW()") # => 2016-12-16 16:36:57
db.scalar("SELECT NOW()") # => 2016-12-16 16:36:57
db.scalar("SELECT NOW()") # => 2016-12-16 16:36:58
db.scalar("SELECT NOW()") # => 2016-12-16 16:36:58
db.scalar("SELECT NOW()") # => 2016-12-16 16:36:59
db.scalar("SELECT NOW()") # => 2016-12-16 16:36:59
# stop mysql server for some seconds
db.scalar("SELECT NOW()") # => 2016-12-16 16:37:06
db.scalar("SELECT NOW()") # => 2016-12-16 16:37:06
db.scalar("SELECT NOW()") # => 2016-12-16 16:37:07
```



## Transactions

When working with databases, it is common to need to group operations in such a way that if one fails, then we can go back to the latest safe state. This solution is described in the **transaction paradigm**, and is implemented by most database engines as it is necessary to meet ACID properties (Atomicity, Consistency, Isolation, Durability) <sup>ACID</sup>

With this in mind, we present the following example:

We have two accounts (each represented by a name and an amount of money).

```
db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100
```

In one moment a transfer is made from one account to the other. For example, *John transfers \$50 to Sarah*

We have two accounts (each represented by a name and an amount of money).

```
deposit db, "Sarah", 50
withdraw db, "John", 50
```

It is important to have in mind that if one of the operations fails then the final state would be inconsistent. So we need to execute the **two operations** (deposit and withdraw) as **one operation**. And if an error occurs then we would like to go back in time as if that one operation was never executed.

```
db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100

db.transaction do |tx|
  cnn = tx.connection

  transfer_amount = 1000
  deposit cnn, "Sarah", transfer_amount
  withdraw cnn, "John", transfer_amount
end
```

In the above example, we start a transaction simply by calling the method `Database#transaction` (how we get the `database` object is encapsulated in the method `get_bank_db` and is out of the scope of this document). The `block` is the body of the transaction. When the `block` gets executed (without any error) then an **implicit commit** is finally executed to persist the changes in the database. If an exception is raised by one of the operations, then an **implicit rollback** is executed, bringing the database to the state before the transaction started.

## Exception handling and rolling back

As we mentioned early, an **implicit rollback** gets executed when an exception is raised, and it's worth mentioning that the exception may be rescued by us.

```
db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100

begin
  db.transaction do |tx|
    cnn = tx.connection

    transfer_amount = 1000
    deposit(cnn, "Sarah", transfer_amount)
    # John does not have enough money in his account!
    withdraw(cnn, "John", transfer_amount)
  end
rescue ex
  puts "Transfer has been rolled back due to: #{ex}"
end
```

We may also raise an exception in the body of the transaction:

```
db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100

begin
  db.transaction do |tx|
    cnn = tx.connection

    transfer_amount = 50
    deposit(cnn, "Sarah", transfer_amount)
    withdraw(cnn, "John", transfer_amount)
    raise Exception.new "Because ..."
  end
rescue ex
  puts "Transfer has been rolled back due to: #{ex}"
end
```

As the previous example, the exception cause the transaction to rollback and then is rescued by us.

There is one `exception` with a different behaviour. If a `DB::Rollback` is raised within the block, the implicit rollback will happen, but the exception will not be raised outside the block.

```

db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100

begin
  db.transaction do |tx|
    cnn = tx.connection

    transfer_amount = 50
    deposit(cnn, "Sarah", transfer_amount)
    withdraw(cnn, "John", transfer_amount)

    # rollback exception
    raise DB::Rollback.new
  end
rescue ex
  # ex is never a DB::Rollback
end

```

## Explicit commit and rollback

In all the previous examples, the rolling back is **implicit**, but we can also tell the transaction to rollback:

```

db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100

begin
  db.transaction do |tx|
    cnn = tx.connection

    transfer_amount = 50
    deposit(cnn, "Sarah", transfer_amount)
    withdraw(cnn, "John", transfer_amount)

    tx.rollback

    puts "Rolling Back the changes!"
  end
rescue ex
  # Notice that no exception is used in this case.
end

```

And we can also use the `commit` method:

```

db = get_bank_db

db.transaction do |tx|
  cnn = tx.connection

  transfer_amount = 50
  deposit(cnn, "Sarah", transfer_amount)
  withdraw(cnn, "John", transfer_amount)

  tx.commit
end

```

**NOTE:** After `commit` or `rollback` are used, the transaction is no longer usable. The connection is still open but any statement will be performed outside the context of the terminated transaction.

## Nested transactions

As the name suggests, a nested transaction is a transaction created inside the scope of another transaction. Here is an example:

```
db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100
create_account db, "Jack", amount: 0

begin
  db.transaction do |outer_tx|
    outer_cnn = outer_tx.connection

    transfer_amount = 50
    deposit(outer_cnn, "Sarah", transfer_amount)
    withdraw(outer_cnn, "John", transfer_amount)

    outer_tx.transaction do |inner_tx|
      inner_cnn = inner_tx.connection

      # John => 50 (pending commit)
      # Sarah => 150 (pending commit)
      # Jack => 0

      another_transfer_amount = 150
      deposit(inner_cnn, "Jack", another_transfer_amount)
      withdraw(inner_cnn, "Sarah", another_transfer_amount)
    end
  end
rescue ex
  puts "Exception raised due to: #{ex}"
end
```

Some observations from the above example: the `inner_tx` works with the values updated although the `outer_tx` is pending the commit. The connection used by `outer_tx` and `inner_tx` is **the same connection**. This is because the `inner_tx` inherits the connection from the `outer_tx` when created.

## Rollback nested transactions

As we've already seen, a rollback may be fired at any time (by an exception or by sending the message `rollback` explicitly)

So let's present an example with a **rollback fired by an exception placed at the outer-transaction**:

```

db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100
create_account db, "Jack", amount: 0

begin
  db.transaction do |outer_tx|
    outer_cnn = outer_tx.connection

    transfer_amount = 50
    deposit(outer_cnn, "Sarah", transfer_amount)
    withdraw(outer_cnn, "John", transfer_amount)

    outer_tx.transaction do |inner_tx|
      inner_cnn = inner_tx.connection

      # John => 50 (pending commit)
      # Sarah => 150 (pending commit)
      # Jack => 0

      another_transfer_amount = 150
      deposit(inner_cnn, "Jack", another_transfer_amount)
      withdraw(inner_cnn, "Sarah", another_transfer_amount)
    end

    raise Exception.new("Rollback all the things!")
  end
rescue ex
  puts "Exception raised due to: #{ex}"
end

```

The rollback place in the `outer_tx` block, rolled back all the changes including the ones in the `inner_tx` block (the same happens if we use an **explicit** rollback).

If the **rollback is fired by an exception at the inner\_tx block** all the changes including the ones in the `outer_tx` are rolled back.

```

db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100
create_account db, "Jack", amount: 0

begin
  db.transaction do |outer_tx|
    outer_cnn = outer_tx.connection

    transfer_amount = 50
    deposit(outer_cnn, "Sarah", transfer_amount)
    withdraw(outer_cnn, "John", transfer_amount)

    outer_tx.transaction do |inner_tx|
      inner_cnn = inner_tx.connection

      # John => 50 (pending commit)
      # Sarah => 150 (pending commit)
      # Jack => 0

      another_transfer_amount = 150
      deposit(inner_cnn, "Jack", another_transfer_amount)
      withdraw(inner_cnn, "Sarah", another_transfer_amount)

      raise Exception.new("Rollback all the things!")
    end
  end
rescue ex
  puts "Exception raised due to: #{ex}"
end

```

There is a way to rollback the changes in the `inner-transaction` but keep the ones in the `outer-transaction`. Use `rollback` in the `inner_tx` object. This will rollback **only** then inner-transaction. Here is the example:

```

db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100
create_account db, "Jack", amount: 0

begin
  db.transaction do |outer_tx|
    outer_cnn = outer_tx.connection

    transfer_amount = 50
    deposit(outer_cnn, "Sarah", transfer_amount)
    withdraw(outer_cnn, "John", transfer_amount)

    outer_tx.transaction do |inner_tx|
      inner_cnn = inner_tx.connection

      # John => 50 (pending commit)
      # Sarah => 150 (pending commit)
      # Jack => 0

      another_transfer_amount = 150
      deposit(inner_cnn, "Jack", another_transfer_amount)
      withdraw(inner_cnn, "Sarah", another_transfer_amount)

      inner_tx.rollback
    end
  end
rescue ex
  puts "Exception raised due to: #{ex}"
end

```

The same happens if a `DB::Rollback` exception is raised in the `inner-transaction` block.

```
db = get_bank_db

create_account db, "John", amount: 100
create_account db, "Sarah", amount: 100
create_account db, "Jack", amount: 0

begin
  db.transaction do |outer_tx|
    outer_cnn = outer_tx.connection


    transfer_amount = 50
    deposit(outer_cnn, "Sarah", transfer_amount)
    withdraw(outer_cnn, "John", transfer_amount)

    outer_tx.transaction do |inner_tx|
      inner_cnn = inner_tx.connection

      # John => 50 (pending commit)
      # Sarah => 150 (pending commit)
      # Jack => 0

      another_transfer_amount = 150
      deposit(inner_cnn, "Jack", another_transfer_amount)
      withdraw(inner_cnn, "Sarah", another_transfer_amount)

      # Rollback exception
      raise DB::Rollback.new
    end
  end
rescue ex
  puts "Exception raised due to: #{ex}"
end
```

ACID. Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. ACM Comput. Surv. 15, 4 (December 1983), 287-317. DOI=<http://dx.doi.org/10.1145/289.291> 



## **Guides**

Read these guides to get the best out of Crystal.

## Performance

Follow these tips to get the best out of your programs, both in speed and memory terms.

### Premature optimization

Donald Knuth once said:

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

However, if you are writing a program and you realize that writing a semantically equivalent, faster version involves just minor changes, you shouldn't miss that opportunity.

And always be sure to profile your program to learn what its bottlenecks are. For profiling, on macOS you can use [Instruments Time Profiler](#), which comes with XCode. On Linux, any program that can profile C/C++ programs, like [perf](#) or [Callgrind](#), should work.

Make sure to always profile programs by compiling or running them with the `--release` flag, which turns on optimizations.

### Avoiding memory allocations

One of the best optimizations you can do in a program is avoiding extra/useless memory allocation. A memory allocation happens when you create an instance of a **class**, which ends up allocating heap memory. Creating an instance of a **struct** uses stack memory and doesn't incur a performance penalty. If you don't know the difference between stack and heap memory, be sure to [read this](#).

Allocating heap memory is slow, and it puts more pressure on the Garbage Collector (GC) as it will later have to free that memory.

There are several ways to avoid heap memory allocations. The standard library is designed in a way to help you do that.

### Don't create intermediate strings when writing to an IO

To print a number to the standard output you write:

```
puts 123
```

In many programming languages what will happen is that `to_s`, or a similar method for converting the object to its string representation, will be invoked, and then that string will be written to the standard output. This works, but it has a flaw: it creates an intermediate string, in heap memory, only to write it and then discard it. This, involves a heap memory allocation and gives a bit of work to the GC.

In Crystal, `puts` will invoke `to_s(io)` on the object, passing it the IO to which the string representation should be written.

So, you should never do this:

```
puts 123.to_s
```

as it will create an intermediate string. Always append an object directly to an IO.

When writing custom types, always be sure to override `to_s(io)`, not `to_s`, and avoid creating intermediate strings in that method. For example:

```
class MyClass
  # Good
  def to_s(io)
    # appends "1, 2" to IO without creating intermediate strings
    x = 1
    y = 2
    io << x << ", " << y
  end

  # Bad
  def to_s(io)
    x = 1
    y = 2
    # using a string interpolation creates an intermediate string.
    # this should be avoided
    io << "#{x}, #{y}"
  end
end
```

This philosophy of appending to an IO instead of returning an intermediate string results in better performance than handling intermediate strings. You should use this strategy in your API definitions too.

Let's compare the times:

```
# io_benchmark.cr
require "benchmark"

io = IO::Memory.new

Benchmark.ips do |x|
  x.report("without to_s") do
    io << 123
    io.clear
  end

  x.report("with to_s") do
    io << 123.to_s
    io.clear
  end
end
```

Output:

```
$ crystal run --release io_benchmark.cr
without to_s 77.11M ( 12.97ns) (± 1.05%)   fastest
with to_s   18.15M ( 55.09ns) (± 7.99%) 4.25x slower
```

Always remember that it's not just the time that has improved: memory usage is also decreased.

## Use string interpolation instead of concatenation

Sometimes you need to work directly with strings built from combining string literals with other values. You shouldn't just concatenate these strings with `String#+(String)` but rather use [string interpolation](#) which allows to embed expressions into a string literal: `"Hello, #{name}"` is better than `"Hello, " + name.to_s`.

Interpolated strings are transformed by the compiler to append to a string IO so that it automatically avoids intermediate strings. The example above translates to:

```
String.build do |io|
  io << "Hello, " << name
end
```

## Avoid IO allocation for string building

Prefer to use the dedicated `String.build` optimized for building strings, instead of creating an intermediate `IO::Memory` allocation.

```

require "benchmark"

Benchmark.ips do |bm|
  bm.report("String.build") do
    String.build do |io|
      99.times do
        io << "hello world"
      end
    end
  end

  bm.report("IO::Memory") do
    io = IO::Memory.new
    99.times do
      io << "hello world"
    end
    io.to_s
  end
end

```

Output:

```

$ crystal run --release str_benchmark.cr
String.build 597.57k ( 1.67µs) (± 5.52%) fastest
IO::Memory 423.82k ( 2.36µs) (± 3.76%) 1.41× slower

```

## Avoid creating temporary objects over and over

Consider this program:

```

lines_with_language_reference = 0
while line = gets
  if ["crystal", "ruby", "java"].any? { |string| line.includes?(string) }
    lines_with_language_reference += 1
  end
end
puts "Lines that mention crystal, ruby or java: #{lines_with_language_reference}"

```

The above program works but has a big performance problem: on every iteration a new array is created for `["crystal", "ruby", "java"]`. Remember: an array literal is just syntax sugar for creating an instance of an array and adding some values to it, and this will happen over and over on each iteration.

There are two ways to solve this:

1. Use a tuple. If you use `{"crystal", "ruby", "java"}` in the above program it will work the same way, but since a tuple doesn't involve heap memory it will be faster, consume less memory, and give more chances for the compiler to optimize the program.

```

lines_with_language_reference = 0
while line = gets
  if {"crystal", "ruby", "java"}.any? { |string| line.includes?(string) }
    lines_with_language_reference += 1
  end
end
puts "Lines that mention crystal, ruby or java: #{lines_with_language_reference}"

```

2. Move the array to a constant.

```

LANGS = ["crystal", "ruby", "java"]

lines_with_language_reference = 0
while line = gets
  if LANGS.any? { |string| line.includes?(string) }
    lines_with_language_reference += 1
  end
end
puts "Lines that mention crystal, ruby or java: #{lines_with_language_reference}"

```

Using tuples is the preferred way.

Explicit array literals in loops is one way to create temporary objects, but these can also be created via method calls. For example `Hash#keys` will return a new array with the keys each time it's invoked. Instead of doing that, you can use

`Hash#each_key` , `Hash#has_key?` and other methods.

## Use structs when possible

If you declare your type as a **struct** instead of a **class**, creating an instance of it will use stack memory, which is much cheaper than heap memory and doesn't put pressure on the GC.

You shouldn't always use a struct, though. Structs are passed by value, so if you pass one to a method and the method makes changes to it, the caller won't see those changes, so they can be bug-prone. The best thing to do is to only use structs with immutable objects, especially if they are small.

For example:

```
# class_vs_struct.cr
require "benchmark"

class PointClass
  getter x
  getter y

  def initialize(@x : Int32, @y : Int32)
  end
end

struct PointStruct
  getter x
  getter y

  def initialize(@x : Int32, @y : Int32)
  end
end

Benchmark.ips do |x|
  x.report("class") { PointClass.new(1, 2) }
  x.report("struct") { PointStruct.new(1, 2) }
end
```

Output:

```
$ crystal run --release class_vs_struct.cr
class 28.17M (± 2.86%) 15.29x slower
struct 430.82M (± 6.58%)          fastest
```

## Iterating strings

Strings in Crystal always contain UTF-8 encoded bytes. UTF-8 is a variable-length encoding: a character may be represented by several bytes, although characters in the ASCII range are always represented by a single byte. Because of this, indexing a string with `String#[]` is not an  $O(1)$  operation, as the bytes need to be decoded each time to find the character at the given position. There's an optimization that Crystal's `String` does here: if it knows all the characters in the string are ASCII, then `String#[]` can be implemented in  $O(1)$ . However, this isn't generally true.

For this reason, iterating a `String` in this way is not optimal, and in fact has a complexity of  $O(n^2)$ :

```
string = "foo"
while i < string.size
  char = string[i]
  # ...
end
```

There's a second problem with the above: computing the `size` of a `String` is also slow, because it's not simply the number of bytes in the string (the `bytesize`). However, once a `String`'s size has been computed, it is cached.

Nil

The way to improve performance in this case is to either use one of the iteration methods ( `each_char` , `each_byte` , `each_codepoint` ), or use the more low-level

`Char::Reader` struct. For example, using `each_char` :

```
string = "foo"  
string.each_char do |char|  
  # ...  
end
```



# Concurrency

## Concurrency vs. Parallelism

The definitions of "concurrency" and "parallelism" sometimes get mixed up, but they are not the same.

A concurrent system is one that can be in charge of many tasks, although not necessarily it is executing them at the same time. You can think of yourself being in the kitchen cooking: you chop an onion, put it to fry, and while it's being fried you chop a tomato, but you are not doing all of those things at the same time: you distribute your time between those tasks. Parallelism would be to stir fry onions with one hand while with the other one you chop a tomato.

At the moment of this writing, Crystal has concurrency support but not parallelism: several tasks can be executed, and a bit of time will be spent on each of these, but two code paths are never executed at the same exact time.

A Crystal program executes in a single operating system thread, except the Garbage Collector (GC) which implements a concurrent mark-and-sweep (currently [Boehm GC](#)).

## Fibers

To achieve concurrency, Crystal has fibers. A fiber is in a way similar to an operating system thread except that it's much more lightweight and its execution is managed internally by the process. So, a program will spawn multiple fibers and Crystal will make sure to execute them when the time is right.

## Event loop

For everything I/O related there's an event loop. Some time-consuming operations are delegated to it, and while the event loop waits for that operation to finish the program can continue executing other fibers. A simple example of this is waiting for data to come through a socket.

## Channels

Crystal has Channels inspired by [CSP](#). They allow communicating data between fibers without sharing memory and without having to worry about locks, semaphores or other special structures.

## Execution of a program

When a program starts, it fires up a main fiber that will execute your top-level code. There, one can spawn many other fibers. The components of a program are:

- The Runtime Scheduler, in charge of executing all fibers when the time is right.
- The Event Loop, which is just another fiber, being in charge of async tasks, like for example files, sockets, pipes, signals and timers (like doing a `sleep`).
- Channels, to communicate data between fibers. The Runtime Scheduler will coordinate fibers and channels for their communication.
- Garbage Collector: to clean up "no longer used" memory.

## A Fiber

A fiber is an execution unit that is more lightweight than a thread. It's a small object that has an associated `stack` of 8MB, which is what is usually assigned to an operating system thread.

Fibers, unlike threads, are cooperative. Threads are pre-emptive: the operating system might interrupt a thread at any time and start executing another one. A fiber must explicitly tell the Runtime Scheduler to switch to another fiber. For example if there's I/O to be waited on, a fiber will tell the scheduler "Look, I have to wait for this I/O to be available, you continue executing other fibers and come back to me when that I/O is ready".

The advantage of being cooperative is that a lot of the overhead of doing a context switch (switching between threads) is gone.

A Fiber is much more lightweight than a thread: even though it's assigned 8MB, it starts with a small stack of 4KB.

On a 64-bit machine it lets us spawn millions and millions of fibers. In a 32-bit machine we can only spawn 512 fibers, which is not a lot. But because 32-bit machines are starting to become obsolete, we'll bet on the future and focus more on 64-bit machines.

## The Runtime Scheduler

The scheduler has a queue of:

- Fibers ready to be executed: for example when you spawn a fiber, it's ready to be executed.
- The event loop: which is another fiber. When there are no other fibers ready to be executed, the event loop checks if there is any async operation that is ready, and then executes the fiber waiting for that operation. The event loop is currently implemented with `libevent`, which is an abstraction of other event mechanisms like `epoll` and `kqueue`.
- Fibers that voluntarily asked to wait: this is done with `Fiber.yield`, which means "I can continue executing, but I'll give you some time to execute other fibers if you want".

## Communicating data

Because at this moment there's only a single thread executing your code, accessing and modifying a class variable in different fibers will work just fine. However, once multiple threads (parallelism) is introduced in the language, it might break. That's why the recommended mechanism to communicate data is using channels and sending messages between them. Internally, a channel implements all the locking mechanisms to avoid data races, but from the outside you use them as communication primitives, so you (the user) don't have to use locks.

## Sample code

### Spawning a fiber

To spawn a fiber you use `spawn` with a block:

```
spawn do
  # ...
  socket.gets
  # ...
end

spawn do
  # ...
  sleep 5.seconds
  # ...
end
```

Here we have two fibers: one reads from a socket and the other does a `sleep`. When the first fiber reaches the `socket.gets` line, it gets suspended, the Event Loop is told to continue executing this fiber when there's data in the socket, and the program continues with the second fiber. This fiber wants to sleep for 5 seconds, so the Event Loop is told to continue with this fiber in 5 seconds. If there aren't other fibers to execute, the Event Loop will wait until either of these events happen, without consuming CPU time.

The reason why `socket.gets` and `sleep` behave like this is because their implementations talk directly with the Runtime Scheduler and the Event Loop, there's nothing magical about it. In general, the standard library already takes care of doing all of this so you don't have to.

Note, however, that fibers don't get executed right away. For example:

```
spawn do
  loop do
    puts "Hello!"
  end
end
```

Running the above code will produce no output and exit immediately.

The reason for this is that a fiber is not executed as soon as it is spawned. So, the main fiber, the one that spawns the above fiber, finishes its execution and the program exits.

One way to solve it is to do a `sleep` :

```
spawn do
  loop do
    puts "Hello!"
  end
end

sleep 1.second
```

This program will now print "Hello!" for one second and then exit. This is because the `sleep` call will schedule the main fiber to be executed in a second, and then executes another "ready to execute" fiber, which in this case is the one above.

Another way is this:

```
spawn do
  loop do
    puts "Hello!"
  end
end

Fiber.yield
```

This time `Fiber.yield` will tell the scheduler to execute the other fiber. This will print "Hello!" until the standard output blocks (the system call will tell us we have to wait until the output is ready), and then execution continues with the main fiber and the program exits. Here the standard output *might* never block so the program will continue executing forever.

If we want to execute the spawned fiber for ever, we can use `sleep` without arguments:

```
spawn do
  loop do
    puts "Hello!"
  end
end

sleep
```

Of course the above program can be written without `spawn` at all, just with a loop. `sleep` is more useful when spawning more than one fiber.

## Spawning a call

You can also spawn by passing a method call instead of a block. To understand why this is useful, let's look at this example:

```

i = 0
while i < 10
  spawn do
    puts(i)
  end
  i += 1
end

Fiber.yield

```

The above program prints "10" ten times. The problem is that there's only one variable `i` that all spawned fibers refer to, and when `Fiber.yield` is executed its value is 10.

To solve this, we can do this:

```

i = 0
while i < 10
  proc = ->(x : Int32) do
    spawn do
      puts(x)
    end
  end
  proc.call(i)
  i += 1
end

Fiber.yield

```

Now it works because we are creating a `Proc` and we invoke it passing `i`, so the value gets copied and now the spawned fiber receives a copy.

To avoid all this boilerplate, the standard library provides a `spawn` macro that accepts a call expression and basically rewrites it to do the above. Using it, we end up with:

```

i = 0
while i < 10
  spawn puts(i)
  i += 1
end

Fiber.yield

```

This is mostly useful with local variables that change at iterations. This doesn't happen with block arguments. For example, this works as expected:

```

10.times do |i|
  spawn do
    puts i
  end
end

Fiber.yield

```

## Spawning a fiber and waiting for it to complete

We can use a channel for this:

```
channel = Channel(Nil).new

spawn do
  puts "Before send"
  channel.send(nil)
  puts "After send"
end

puts "Before receive"
channel.receive
puts "After receive"
```

This prints:

```
Before receive
Before send
After receive
```

First, the program spawns a fiber but doesn't execute it yet. When we invoke `channel.receive`, the main fiber blocks and execution continues with the spawned fiber. Then `channel.send(nil)` is invoked, and so execution continues at `channel.receive`, which was waiting for a value. Then the main fiber continues executing and finishes, so the program exits without giving the other fiber a chance to print "After send".

In the above example we used `nil` just to communicate that the fiber ended. We can also use channels to communicate values between fibers:

```
channel = Channel(Int32).new

spawn do
  puts "Before first send"
  channel.send(1)
  puts "Before second send"
  channel.send(2)
end

puts "Before first receive"
value = channel.receive
puts value # => 1

puts "Before second receive"
value = channel.receive
puts value # => 2
```

Output:

```
Before first receive
Before first send
1
Before second receive
Before second send
2
```

Note that when the program executes a `receive`, that fiber blocks and execution continues with the other fiber. When `send` is executed, execution continues with the fiber that was waiting on that channel.

Here we are sending literal values, but the spawned fiber might compute this value by, for example, reading a file, or getting it from a socket. When this fiber will have to wait for I/O, other fibers will be able to continue executing code until I/O is ready, and finally when the value is ready and sent through the channel, the main fiber will receive it. For example:

```
require "socket"

channel = Channel(String).new

spawn do
  server = TCPServer.new("0.0.0.0", 8080)
  socket = server.accept
  while line = socket.gets
    channel.send(line)
  end
end

spawn do
  while line = gets
    channel.send(line)
  end
end

3.times do
  puts channel.receive
end
```

The above program spawns two fibers. The first one creates a `TCPServer`, accepts one connection and reads lines from it, sending them to the channel. There's a second fiber reading lines from standard input. The main fiber reads the first 3 messages sent to the channel, either from the socket or stdin, then the program exits. The `gets` calls will block the fibers and tell the Event Loop to continue from there if data comes.

Likewise, we can wait for multiple fibers to complete execution, and gather their values:

```
channel = Channel(Int32).new

10.times do |i|
  spawn do
    channel.send(i * 2)
  end
end

sum = 0
10.times do
  sum += channel.receive
end

puts sum # => 90
```

You can, of course, use `receive` inside a spawned fiber:

```

channel = Channel(Int32).new

spawn do
  puts "Before send"
  channel.send(1)
  puts "After send"
end

spawn do
  puts "Before receive"
  puts channel.receive
  puts "After receive"
end

puts "Before yield"
Fiber.yield
puts "After yield"

```

Output:

```

Before yield
Before send
Before receive
1
After receive
After send
After yield

```

Here `channel.send` is executed first, but since there's no one waiting for a value (yet), execution continues in other fibers. The second fiber is executed, there's a value on the channel, it's obtained, and execution continues, first with the first fiber, then with the main fiber, because `Fiber.yield` puts a fiber at the end of the execution queue.

## Buffered channels

The above examples use unbuffered channels: when sending a value, if a fiber is waiting on that channel then execution continues on that fiber.

With a buffered channel, invoking `send` won't switch to another fiber unless the buffer is full:

```

# A buffered channel of capacity 2
channel = Channel(Int32).new(2)

spawn do
  puts "Before send 1"
  channel.send(1)
  puts "Before send 2"
  channel.send(2)
  puts "Before send 3"
  channel.send(3)
  puts "After send"
end

3.times do |i|
  puts channel.receive
end

```



## Output:

```
Before send 1
Before send 2
Before send 3
1
2
After send
3
```

Note that the first 2 sends are executed without switching to another fiber. However, in the third send the channel's buffer is full, so execution goes to the main fiber. Here the two values are received and the channel is depleted. At the third `receive` the main fiber blocks and execution goes to the other fiber, which sends more values, finishes, etc.

## Testing Crystal Code

Crystal comes with a fully-featured spec library in the `Spec` module. It provides a structure for writing executable examples of how your code should behave.

Inspired by `Rspec`, it includes a domain specific language (DSL) that allows you to write examples in a way similar to plain english.

A basic spec looks something like this:

```
require "spec"

describe Array do
  describe "#size" do
    it "correctly reports the number of elements in the Array" do
      [1, 2, 3].size.should eq 3
    end
  end

  describe "#empty?" do
    it "is true when no elements are in the array" do
      ([] of Int32).empty?.should be_true
    end

    it "is false if there are elements in the array" do
      [1].empty?.should be_false
    end
  end
end
```

## Anatomy of a spec file

To use the spec module and DSL, you need to add `require "spec"` to your spec files. Many projects use a custom `spec helper` which organizes these includes.

Concrete test cases are defined in `it` blocks. An optional (but strongly recommended) descriptive string states it's purpose and a block contains the main logic performing the test.

Test cases that have been defined or outlined but are not yet expected to work can be defined using `pending` instead of `it`. They will not be run but show up in the spec report as pending.

An `it` block contains an example that should invoke the code to be tested and define what is expected of it. Each example can contain multiple expectations, but it should test only one specific behaviour.

When `spec` is included, every object has the instance methods `#should` and `#should_not`. These methods are invoked on the value being tested with an expectation as argument. If the expectation is met, code execution continues. Otherwise the example has *failed* and other code in this block will not be executed.

In test files, specs are structured by example groups which are defined by `describe` and `context` sections. Typically a top level `describe` defines the outer unit (such as a class) to be tested by the spec. Further `describe` sections can be nested within the outer unit to specify smaller units under test (such as individual methods).

For unit tests, it is recommended to follow the conventions for method names: Outer `describe` is the name of the class, inner `describe` targets methods. Instance methods are prefixed with `#`, class methods with `..`.

To establish certain contexts - think *empty array* versus *array with elements* - the `context` method may be used to communicate this to the reader. It has a different name, but behaves exactly like `describe`.

`describe` and `context` take a description as argument (which should usually be a string) and a block containing the individual specs or nested groupings.

## Expectations

Expectations define if the value being tested (*actual*) matches a certain value or specific criteria.

### Equivalence, Identity and Type

There are methods to create expectations which test for equivalence (`eq`), identity (`be`), type (`be_a`), and nil (`be_nil`). Note that the identity expectation uses `.same?` which tests if `#object_id` are identical. This is only true if the expected value points to *the same object* instead of *an equivalent one*. This is only possible for reference types and won't work for value types like structs or numbers.

```
actual.should eq(expected) # passes if actual == expected
actual.should be(expected) # passes if actual.same?(expected)
actual.should be_a(expected) # passes if actual.is_a?(expected)
actual.should be_nil      # passes if actual.nil?
```

### Truthiness

```
actual.should be_true # passes if actual == true
actual.should be_false # passes if actual == false
actual.should be_truthy # passes if actual is truthy (neither nil nor false nor Pointer)
actual.should be_falsey # passes if actual is falsey (nil, false or Pointer.null)
```

### Comparisons

```
actual.should be < expected # passes if actual < expected
actual.should be <= expected # passes if actual <= expected
actual.should be > expected # passes if actual > expected
actual.should be >= expected # passes if actual >= expected
```

## Other matchers

```
actual.should be_close(expected, delta) # passes if actual is within delta of expected
                                         # (actual - expected).abs <= delta
actual.should contain(expected)         # passes if actual.includes?(expected)
actual.should match(expected)          # passes if actual =~ expected
```

## Expecting errors

These matchers run a block and pass if it raises a certain exception.

```
expect_raises(MyError) do
  # Passes if this block raises an exception of type MyError.
end

expect_raises(MyError, "error message") do
  # Passes if this block raises an exception of type MyError
  # and the error message contains "error message".
end

expect_raises(MyError, /error \w{7}/) do
  # Passes if this block raises an exception of type MyError
  # and the error message matches the regular expression.
end
```

They return the rescued exception so it can be used for further expectations, for example to verify specific properties of the exception.

## Focusing on a group of specs

`describe`, `context` and `it` blocks can be marked with `focus: true`, like this:

```
it "adds", focus: true do
  (2 + 2).should_not eq(5)
end
```

If any such thing is marked with `focus: true` then only those examples will run.

## Tagging specs

Tags can be used to group specs, allowing to only run a subset of specs when providing a `--tag` argument to the spec runner (see [Using the compiler](#)).

`describe`, `context` and `it` blocks can be tagged, like this:

```

it "is slow", tags: "slow" do
  sleep 60
  true.should be(true)
end

it "is fast", tags: "fast" do
  true.should be(true)
end

```

Tagging an example group ( `describe` or `context` ) extends to all of the contained examples.

Multiple tags can be specified by giving an `Enumerable` , such as `Array` or `Set` .

## Running specs

The Crystal compiler has a `spec` command with tools to constrain which examples get run and tailor the output. All specs of a project are compiled and executed through the command `crystal spec` .

By convention, specs live in the `spec/` directory of a project. Spec files must end with `_spec.cr` to be recognizable as such by the compiler command.

You can compile and run specs from folder trees, individual files, or specific lines in a file. If the specified line is the beginning of a `describe` or `context` section, all specs inside that group are run.

The default formatter outputs the file and line style command for failing specs which makes it easy to rerun just this individual spec.

You can turn off colors with the switch `--no-color` .

## Randomizing order of specs

Specs, by default, run in the order defined, but can be run in a random order by passing `--order random` to `crystal spec` .

Specs run in random order will display a seed value upon completion. This seed value can be used to rerun the specs in that same order by passing the seed value to `--order` .

## Examples

```

# Run all specs in files matching spec/**/*_spec.cr
crystal spec

# Run all specs in files matching spec/**/*_spec.cr without colors
crystal spec --no-color

# Run all specs in files matching spec/my/test/**/*_spec.cr
crystal spec spec/my/test/

# Run all specs in spec/my/test/file_spec.cr
crystal spec spec/my/test/file_spec.cr

# Run the spec or group defined in line 14 of spec/my/test/file_spec.cr
crystal spec spec/my/test/file_spec.cr:14

# Run all specs tagged with "fast"
crystal spec --tag 'fast'

# Run all specs not tagged with "slow"
crystal spec --tag '~slow'

```

## Spec helper

Many projects use a custom spec helper file, usually named `spec/spec_helper.cr`.

This file is used to require `spec` and other includes like code from the project needed for every spec file. This is also a good place to define global helper methods that make writing specs easier and avoid code duplication.

```

# spec/spec_helper.cr
require "spec"
require "../src/my_project.cr"

def create_test_object(name)
  project = MyProject.new(option: false)
  object = project.create_object(name)
  object
end

# spec/my_project_spec.cr
require "./spec_helper"

describe "MyProject::Object" do
  it "is created" do
    object = create_test_object(name)
    object.should_not be_nil
  end
end

```

# Writing Shards

How to write and release Crystal Shards.

## What's a Shard?

Simply put, a Shard is a package of Crystal code, made to be shared-with and used-by other projects.

See [the Shards command](#) for details.

## Introduction

In this tutorial, we'll be making a Crystal library called *palindrome-example*.

For those who don't know, a palindrome is a word which is spelled the same way forwards as it is backwards. e.g. racecar, mom, dad, kayak, madam

## Requirements

In order to release a Crystal Shard, and follow along with this tutorial, you will need the following:

- A working installation of the [Crystal compiler](#)
- A working installation of [Git](#)
- A [GitHub](#) or [GitLab](#) account

## Creating the Project

Begin by using [the Crystal compiler's](#) `init lib` command to create a Crystal library with the standard directory structure.

In your terminal: `crystal init lib <YOUR-SHARD-NAME>`

e.g.

```
$ crystal init lib palindrome-example
  create  palindrome-example/.gitignore
  create  palindrome-example/.editorconfig
  create  palindrome-example/LICENSE
  create  palindrome-example/README.md
  create  palindrome-example/.travis.yml
  create  palindrome-example/shard.yml
  create  palindrome-example/src/palindrome-example.cr
  create  palindrome-example/src/palindrome-example/version.cr
  create  palindrome-example/spec/spec_helper.cr
  create  palindrome-example/spec/palindrome-example_spec.cr
  Initialized empty Git repository in /<YOUR-DIRECTORY>/../palindrome-example/.git/
```

...and `cd` into the directory:

e.g.

```
cd palindrome-example
```

Then `add` & `commit` to start tracking the files with Git:

```
$ git add -A
$ git commit -am "First Commit"
[master (root-commit) 77bad84] First Commit
10 files changed, 102 insertions(+)
create mode 100644 .editorconfig
create mode 100644 .gitignore
create mode 100644 .travis.yml
create mode 100644 LICENSE
create mode 100644 README.md
create mode 100644 shard.yml
create mode 100644 spec/palindrome-example_spec.cr
create mode 100644 spec/spec_helper.cr
create mode 100644 src/palindrome-example.cr
create mode 100644 src/palindrome-example/version.cr
```

## Writing the Code

The code you write is up to you, but how you write it impacts whether people want to use your library and/or help you maintain it.

## Testing the Code

- Test your code. All of it. It's the only way for anyone, including you, to know if it works.
- Crystal has [a built-in testing library](#). Use it!

## Documentation

- Document your code with comments. All of it. Even the private methods.
- Crystal has [a built-in documentation generator](#). Use it!

Run `crystal docs` to convert your code and comments into interlinking API documentation. Open the files in the `/docs/` directory with a web browser to see how your documentation is looking along the way.

See below for instructions on hosting your compiler-generated docs on GitHub/GitLab Pages.

Once your documentation is ready and available, you can add a documentation badge to your repository so users know that it exists. In GitLab this badge belongs to the project so we'll cover it in the GitLab instructions below, for GitHub it is common to place it below the description in your `README.md` like so: (Be sure to replace `<LINK-TO-YOUR-DOCUMENTATION>` accordingly)

```
[![Docs](https://img.shields.io/badge/docs-available-brightgreen.svg)](<LINK-TO-YOUR-D
```



## Writing a README

A good README can make or break your project. [Awesome README](#) is a nice curation of examples and resources on the topic.

Most importantly, your README should explain:

1. What your library is
2. What it does
3. How to use it

This explanation should include a few examples along with subheadings.

NOTE: Be sure to replace all instances of `[your-github-name]` in the Crystal-generated README template with your GitHub/GitLab username. If you're using GitLab, you'll also want to change all instances of `github` with `gitlab`.

## Coding Style

- It's fine to have your own style, but sticking to [some core rubrics defined by the Crystal team](#) can help keep your code consistent, readable and usable for other developers.
- Utilize Crystal's [built-in code formatter](#) to automatically format all `.cr` files in a directory.

e.g.

```
crystal tool format
```

To check if your code is formatted correctly, or to check if using the formatter wouldn't produce any changes, simply add `--check` to the end of this command.

e.g.

```
crystal tool format --check
```

See the Travis CI section below to implement this in your build.

## Writing a `shard.yml`

The [spec](#) is your rulebook. Follow it.

### Name

Your `shard.yml`'s `name` property should be concise and descriptive.

- Search [crystalshards.xyz](#) to check if your name is already taken.

e.g.

```
name: palindrome-example
```

## Description

Add a `description` to your `shard.yml`.

A `description` is a single line description used to search for and find your shard.

A description should be:

1. Informative
2. Discoverable

## Optimizing

It's hard for anyone to use your project if they can't find it. [crystalshards.xyz](https://crystalshards.xyz) is currently the go-to place for Crystal libraries, so that's what we'll optimize for.

There are people looking for the *exact* functionality of our library and the *general* functionality of our library. e.g. Bob needs a palindrome library, but Felipe is just looking for libraries involving text and Susan is looking for libraries involving spelling.

Our `name` is already descriptive enough for Bob's search of "palindrome". We don't need to repeat the *palindrome* keyword. Instead, we'll catch Susan's search for "spelling" and Felipe's search for "text".

```
description: |  
  A textual algorithm to tell if a word is spelled the same way forwards as it is back
```

## Hosting

From here the guide differs depending on whether you are hosting your repo on GitHub or GitLab. If you're hosting somewhere else, please feel free to write up a guide and add it to this book!

- [Hosting on GitHub](#)
- [Hosting on GitLab](#)

## Hosting on GitHub

- Create a repository with the same `name` and `description` as specified in your `shard.yml`.

- Add and commit everything:

```
$ git add -A && git commit -am "shard complete"
```

- Add the remote: (Be sure to replace `<YOUR-GITHUB-USERNAME>` and `<YOUR-REPOSITORY-NAME>` accordingly)

NOTE: If you like, feel free to replace `public` with `origin`, or a remote name of your choosing.

```
$ git remote add public https://github.com/<YOUR-GITHUB-NAME>/<YOUR-REPOSITORY-NAME>.g
```

- Push it:

```
$ git push public master
```

## GitHub Releases

It's good practice to do GitHub Releases.

Add the following markdown build badge below the description in your README to inform users what the most current release is: (Be sure to replace `<YOUR-GITHUB-USERNAME>` and `<YOUR-REPOSITORY-NAME>` accordingly)

```
[![GitHub release](https://img.shields.io/github/release/<YOUR-GITHUB-USERNAME>/<YOUR-
```

Start by navigating to your repository's *releases* page.

- This can be found at `https://github.com/<YOUR-GITHUB-NAME>/<YOUR-REPOSITORY-NAME>/releases`

Click "Create a new release".

According to [the Crystal Shards README](#),

When libraries are installed from Git repositories, the repository is expected to have version tags following a semver-like format, prefixed with a `v`.  
Examples: `v1.2.3`, `v2.0.0-rc1` or `v2017.04.1`

Accordingly, in the input that says `tag version`, type `v0.1.0`. Make sure this matches the `version` in `shard.yml`. Title it `v0.1.0` and write a short description for the release.

Click "Publish release" and you're done!

You'll now notice that the GitHub Release badge has updated in your README.

Follow [Semantic Versioning](#) and create a new release every time your push new code to `master`.

## Travis CI and `.travis.yml`

If you haven't already, [sign up for Travis CI](#).

Insert the following markdown build badge below the description in your README.md: (be sure to replace `<YOUR-GITHUB-USERNAME>` and `<YOUR-REPOSITORY-NAME>` accordingly)

```
[![Build Status](https://travis-ci.org/<YOUR-GITHUB-USERNAME>/<YOUR-REPOSITORY-NAME>.s
```

Build badges are a simple way to tell people whether your Travis CI build passes.

Add the following lines to your `.travis.yml`:

```
script:
  - crystal spec
```

This tells Travis CI to run your tests. Accordingly with the outcome of this command, Travis CI will return a [build status](#) of "passed", "errored", "failed" or "canceled".

If you want to verify that all your code has been formatted with `crystal tool format`, add a script for `crystal tool format --check`. If the code is not formatted correctly, this will [break the build](#) just as failing tests would.

e.g.

```
script:
  - crystal spec
  - crystal tool format --check
```

Commit and push to GitHub.

Follow [these guidelines](#) to get your repo up & running on Travis CI.

Once you're up and running, and the build is passing, the build badge will update in your README.

## Hosting your `docs` on GitHub-Pages

Add the following `script` to your `.travis.yml`:

```
- crystal docs
```

This tells Travis CI to generate your documentation.

Next, add the following lines to your `.travis.yml`. (Be sure to replace all instances of `<YOUR-GITHUB-REPOSITORY-NAME>` accordingly)

```
deploy:
  provider: pages
  skip_cleanup: true
  github_token: $GITHUB_TOKEN
  project_name: <YOUR-GITHUB-REPOSITORY-NAME>
  on:
    branch: master
  local_dir: docs
```

Set the Environment Variable, `GITHUB_TOKEN`, with your [personal access token](#).

If you've been following along, your `.travis.yml` file should look something like this:

```
language: crystal
script:
  - crystal spec
  - crystal docs
deploy:
  provider: pages
  skip_cleanup: true
  github_token: $GITHUB_TOKEN
  project_name: <YOUR-GITHUB-REPOSITORY-NAME>
  on:
    branch: master
  local_dir: docs
```

[Click Here](#) for the official documentation on deploying to GitHub-Pages with Travis CI.

## Hosting on GitLab

- Go ahead and delete the default `travis.yml` that comes with the project. We won't be needing it.
- Add and commit everything:

```
$ git add -A && git commit -am "shard complete"
```

- Create a GitLab project with the same `name` and `description` as specified in your `shard.yml`.
- Add the remote: (Be sure to replace `<YOUR-GITLAB-USERNAME>` and `<YOUR-REPOSITORY-NAME>` accordingly)

```
$ git remote add origin https://gitlab.com/<YOUR-GITLAB-USERNAME>/<YOUR-REPOSIT
```

or if you use SSH

```
$ git remote add origin git@gitlab.com:<YOUR-GITLAB-USERNAME>/<YOUR-REPOSITORY-
```

- Push it:

```
$ git push origin master
```

## Pipelines

Next, let's setup a [GitLab Pipeline](#) that can run our tests and build/deploy the docs when we push code to the repo.

Simply, you can just add the following file to the root of the repo and name it `.gitlab-ci.yml`

```

image: "crystallang/crystal:latest"

before_script:
  - shards install

cache:
  paths:
  - lib/

spec & format:
  script:
  - crystal spec
  - crystal tool format --check

pages:
  stage: deploy
  script:
  - crystal docs -o public src/palindrome-example.cr
  artifacts:
    paths:
    - public
  only:
  - master

```

This creates two jobs. The first one is titled "spec & format" (you can use any name you like) and by default goes in the "test" stage of the pipeline. It just runs the array of commands in `script` on a brand new instance of the docker container specified by `image`. You'll probably want to lock that container to the version of crystal you're using (the one specified in your `shard.yml`) but for this example we'll just use the `latest` tag.

The test stage of the pipeline will either pass (each element of the array returned a healthy exit code) or it will fail (one of the elements returned an error).

If it passes, then the pipeline will move onto the second job we defined here which [we must name](#) "pages". This is a special job just for deploying content to your gitlab pages site! This one is executed after tests have passed because we specified that it should occur in the "deploy" stage. It again runs the commands in `script` (this time building the docs), but this time we tell it to preserve the path `public` (where we stashed the docs) as an artifact of the job.

The result of naming this job `pages` and putting our docs in the `public` directory and specifying it as an `artifact` is that GitLab will deploy the site in that directory to the default URL `https://<YOUR-GITLAB-USERNAME>.gitlab.io/<YOUR-REPOSITORY-NAME>`.

The `before_script` and `cache` keys in the file are for running the same script in every job ( `shards install` ) and for hanging onto the files that were created ( `cache` ). They're not necessary if your shard doesn't have any dependencies.

If you commit the above file to your project and push, you'll trigger your first run of the new pipeline.

```
$ git add -A && git commit -am 'Add .gitlab-ci.yml' && git push origin master
```

## Some Badges

While that pipeline is running, let's attach some badges to the project to show off our docs and the (hopefully) successful state of our pipeline. (You might want to read the [badges docs](#).)

A badge is just a link with an image. So let's create a link to our pipeline and fetch a badge image from the [Gitlab Pipeline Badges API](#).

In the *Badges* section of the *General* settings, we'll first add a release badge. The link is: `https://gitlab.com/<YOUR-GITLAB-USERNAME>/<YOUR-REPOSITORY-NAME>/pipelines` and the *Badge Image URL* is: `https://gitlab.com/<YOUR-GITLAB-USERNAME>/<YOUR-REPOSITORY-NAME>/badges/master/pipeline.svg` .

And now if the pipeline has finished we'll have docs and we can link to them with a generic badge from `shields.io` .

- Link: `https://<YOUR-GITLAB-USERNAME>.gitlab.io/<YOUR-REPOSITORY-NAME>`
- Image: `https://img.shields.io/badge/docs-available-brightgreen.svg`

## Releases

A release is just a special commit in your history with a name (see [tagging](#)).

According to [the Crystal Shards README](#),

When libraries are installed from Git repositories, the repository is expected to have version tags following a semver-like format, prefixed with a `v` .  
Examples: `v1.2.3`, `v2.0.0-rc1` or `v2017.04.1`

GitLab also has a [releases feature](#) that let's you associate files and a description with this tag. That way you can (for example) distribute binaries.

As you'll see from the [releases docs](#), you can either create an *annotated* tag along with release notes/files in the UI:

The screenshot shows the 'New Tag' interface in GitLab. It has three main sections: 'Tag name' with the value 'v0.1.0', 'Create from' with a dropdown menu set to 'master', and 'Message' with the text 'Release v0.1.0'. Below the message field is a note: 'Optionally, add a message to the tag.' The 'Release notes' section has a 'Write' tab selected, showing a text area with 'First release :tada:'. At the bottom of the form are two buttons: 'Create tag' (highlighted in green) and 'Cancel'.

or you can create the tag from the command line like so:

```
$ git tag -a v0.1.0 -m "Release v0.1.0"
```

push it up



```
$ git push origin master --follow-tags
```

and then use the UI to add/edit the release note and attach files.

### Best Practices

- Use the `-a` option to create an annotated tag for releases.
- Follow [Semantic Versioning](#).

## Release Badge

If you'd like you can also add a `shields.io` badge for the release. GitLab doesn't have full support for this kind of thing, and until someone adds a [version badge for gitlab](#) to shields.io, we'll have to just code in the version number in the URLs directly.

- Link: `https://img.shields.io/badge/release-<VERSION>-brightgreen.svg`
- Image: `https://img.shields.io/badge/release-<VERSION>-brightgreen.svg`

where `<VERSION>` is the version number prefixed with a `v` like this: `v0.1.0`.

## Mirror to GitHub

At the moment, `crystalshards.xyz` only uses the GitHub API, so if you want your library to be indexed on that service you can set up a "push mirror" from GitLab to GitHub.

1. Create a GitHub repository with the same name as your project.
2. Follow the instructions here: [https://docs.gitlab.com/ee/workflow/repository\\_mirroring.html#setting-up-a-push-mirror-from-gitlab-to-github-core](https://docs.gitlab.com/ee/workflow/repository_mirroring.html#setting-up-a-push-mirror-from-gitlab-to-github-core)
3. Edit your GitHub description. The first few words of this description will show up in the search results of `crystalshards.xyz` but not the whole string, so for example, you could use the following
  - Description: Words that are the same forwards and backwards. This is a mirror of:
  - Link: <https://gitlab.com///>

This is a push mirror and that means changes will only propagate one way. So be sure to let potential collaborators know that pull requests and issues should be submitted to your GitLab project.

## Continuous Integration

The ability of having immediate feedback on what we are working should be one of the most important characteristics in software development. Imagine making one change to our source code and having to wait 2 weeks to see if it broke something? oh! That would be a nightmare! For this, Continuous Integration will help a team to have immediate and frequent feedback about the status of what they are building.

Martin Fowler [defines Continuous Integration](#) as *a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.*

In the next subsections, we are going to present 2 continuous integration tools: [Travis CI](#) and [Circle CI](#) and use them with a Crystal example application.

These tools not only will let us build and test our code each time the source has changed but also deploy the result (if the build was successful) or use automatic builds, and maybe test against different platforms, to mention a few.

## The example application

We are going to use Conway's Game of Life as the example application. More precisely, we are going to use only the first iterations in [Conway's Game of Life Kata](#) solution using [TDD](#).

Note that we won't be using TDD in the example itself, but we will mimic as if the example code is the result of the first iterations.

Another important thing to mention is that we are using `crystal init` to [create the application](#).

And here's the implementation:

```

# src/game_of_life.cr
class Location
  getter x : Int32
  getter y : Int32

  def self.random
    Location.new(Random.rand(10), Random.rand(10))
  end

  def initialize(@x, @y)
  end
end

class World
  @living_cells : Array(Location)

  def self.empty
    new
  end

  def initialize(living_cells = [] of Location)
    @living_cells = living_cells
  end

  def set_living_at(a_location)
    @living_cells << a_location
  end

  def is_empty?
    @living_cells.size == 0
  end
end

```

And the specs:

```

# spec/game_of_life_spec.cr
require "./spec_helper"

describe "a new world" do
  it "should be empty" do
    world = World.new
    world.is_empty?.should be_true
  end
end

describe "an empty world" do
  it "should not be empty after adding a cell" do
    world = World.empty
    world.set_living_at(Location.random)
    world.is_empty?.should be_false
  end
end

```

And this is all we need for our continuous integration examples! Let's start!

## Continuous Integration step by step

Here's the list of items we want to achieve:

1. Build and run specs using 3 different Crystal's versions:

Nil

- latest
  - nightly
  - 0.31.1 (using a Docker image)
2. Install shards packages
  3. Install binary dependencies
  4. Use a database (for example MySQL)
  5. Cache dependencies to make the build run faster

From here choose your next steps:

- I want to use [Travis CI](#)
- I want to use [CircleCI](#)

## Travis CI

In this section we are going to use [Travis CI](#) as our continuous-integration service. Travis CI is [mostly used](#) for building and running tests for projects hosted at GitHub. It supports [different programming languages](#) and for our particular case, it supports the [Crystal language](#).

**Note:** If you are new to continuous integration (or you want to refresh the basic concepts) we may start reading the [core concepts guide](#).

Now let's see some examples!

## Build and run specs

### Using `latest` and `nightly`

A first (and very basic) Travis CI config file could be:

```
# .travis.yml
language: crystal
```

That's it! With this config file, Travis CI by default will run `crystal spec`. Now, we just need to go to Travis CI dashboard to [add the GitHub repository](#).

Let's see another example:

```
# .travis.yml
language: crystal

crystal:
  - latest
  - nightly

script:
  - crystal spec
  - crystal tool format --check
```

With this configuration, Travis CI will run the tests using both Crystal `latest` and `nightly` releases on every push to a branch on your Github repository.

**Note:** When [creating a Crystal project](#) using `crystal init`, Crystal creates a `.travis.yml` file for us.

### Using a specific Crystal release

Let's suppose we want to pin a specific Crystal release (maybe we want to make sure the shard compiles and works with that version) for example [Crystal 0.31.1](#).

Travis CI only provides *runners* to `latest` and `nightly` releases directly and so, we need to install the requested Crystal release manually. For this we are going to use [Docker](#).

First we need to add Docker as a service in `.travis.yml`, and then we can use `docker` commands in our build steps, like this:

```
# .travis.yml
language: minimal

services:
  - docker

script:
  - docker run -v $PWD:/src -w /src crystallang/crystal:0.31.1 crystal spec
```

**Note:** We may read about different (languages)[<https://docs.travis-ci.com/user/languages/>] supported by Travis CI, included `minimal`.

**Note:** A list with the different official [Crystal docker images](#) is available at [DockerHub](#).

## Using `latest`, `nightly` and a specific Crystal release all together!

Supported *runners* can be combined with Docker-based *runners* using a [Build Matrix](#). This will allow us to run tests against `latest` and `nightly` and pinned releases.

Here is the example:

```
# .travis.yml
matrix:
  include:
    - language: crystal
      crystal:
        - latest
      script:
        - crystal spec

    - language: crystal
      crystal:
        - nightly
      script:
        - crystal spec

    - language: bash
      services:
        - docker
      script:
        - docker run -v $PWD:/src -w /src crystallang/crystal:0.31.1 crystal spec
```

## Installing shards packages

In native *runners* (`language: crystal`), Travis CI already automatically installs shards dependencies using `shards install`. To improve build performance we may add [caching](#) on top of that.

## Using Docker

In a Docker-based *runner* we need to run `shards install` explicitly, like this:

```
# .travis.yml
language: bash

services:
  - docker

script:
  - docker run -v $PWD:/src -w /src crystallang/crystal:0.31.1 shards install
  - docker run -v $PWD:/src -w /src crystallang/crystal:0.31.1 crystal spec
```

**Note:** Since the shards will be installed in `./lib/` folder, it will be preserved for the second docker run command.

## Installing binary dependencies

Our application or maybe some shards may required libraries and packages. This binary dependencies may be installed using different methods. Here we are going to show an example using the [Apt](#) command (since the Docker image we are using is based on Ubuntu)

Here is a first example installing the `libsqlite3` development package using the [APT addon](#):

```
# .travis.yml
language: crystal
crystal:
  - latest

before_install:
  - sudo apt-get -y install libsqlite3-dev

addons:
  apt:
    update: true

script:
  - crystal spec
```

## Using Docker

We are going to build a new docker image based on [crystallang/crystal](#), and in this new image we will be installing the binary dependencies.

To accomplish this we are going to use a [Dockerfile](#):

```
# Dockerfile
FROM crystallang/crystal:latest

# install binary dependencies:
RUN apt-get update && apt-get install -y libsqlite3-dev
```

And here is the Travis CI configuration file:

```

# .travis.yml
language: bash

services:
  - docker

before_install:
  # build image using Dockerfile:
  - docker build -t testing .

script:
  # run specs in the container
  - docker run -v $PWD:/src -w /src testing crystal spec

```

**Note:** Dockerfile arguments can be used to use the same Dockerfile for latest, nightly or a specific version.

## Using services

Travis CI may start [services](#) as requested.

For example, we can start a [MySQL](#) database service by adding a `services:` section to our `.travis.yml` :

```

# .travis.yml
language: crystal
crystal:
  - latest

services:
  - mysql

script:
  - crystal spec

```

Here is the new test file for testing against the database:

```

# spec/simple_db_spec.cr
require "./spec_helper"
require "mysql"

it "connects to the database" do
  DB.connect ENV["DATABASE_URL"] do |cnn|
    cnn.query_one("SELECT 'foo'", as: String).should eq "foo"
  end
end

```

When pushing this changes Travis CI will report the following error: `Unknown database 'test' (Exception)` , showing that we need to configure the MySQL service **and also setup the database:**



```

# .travis.yml
language: crystal
crystal:
  - latest

env:
  global:
    - DATABASE_NAME=test
    - DATABASE_URL=mysql://root@localhost/$DATABASE_NAME

services:
  - mysql

before_install:
  - mysql -e "CREATE DATABASE IF NOT EXISTS $DATABASE_NAME;"
  - mysql -u root --password="" $DATABASE_NAME < db/schema.sql

script:
  - crystal spec

```

We are using a [schema.sql](#) script to create a more readable `.travis.yml`. The file `./db/schema.sql` looks like this:

```

-- schema.sql
CREATE TABLE ... etc ...

```

Pushing these changes will trigger Travis CI and the build should be successful!

## Caching

If we read Travis CI job log, we will find that every time the job runs, Travis CI needs to fetch the libraries needed to run the application:

```

Fetching https://github.com/crystal-lang/crystal-mysql.git
Fetching https://github.com/crystal-lang/crystal-db.git

```

This takes time and, on the other hand, these libraries might not change as often as our application, so it looks like we may cache them and save time.

Travis CI [uses caching](#) to improve some parts of the building path. Here is the new configuration file **with cache enabled**:

```

# .travis.yml
language: crystal
crystal:
  - latest

cache: shards

script:
  - crystal spec

```

Let's push these changes. Travis CI will run, and it will install dependencies, but then it will cache the shards cache folder which, usually, is `~/.cache/shards`. The following runs will use the cached dependencies.

Nil

## CircleCI

In this section we are going to use [CircleCI](#) as our continuous-integration service. In a [few words](#) CircleCI automates your software builds, tests, and deployments. It supports [different programming languages](#) and for our particular case, it supports the [Crystal language](#).

In this section we are going to present some configuration examples to see how CircleCI implements some [continuous integration concepts](#).

## CircleCI orbs

Before showing some examples, it's worth mentioning [CircleCI orbs](#). As defined in the official docs:

Orbs define reusable commands, executors, and jobs so that commonly used pieces of configuration can be condensed into a single line of code.

In our case, we are going to use [Crystal's Orb](#)

## Build and run specs

### Simple example using `latest`

Let's start with a simple example. We are going to run the tests **using latest** Crystal release:

```
# .circleci/config.yml
workflows:
  version: 2
  build:
    jobs:
      - crystal/test

orbs:
  crystal: manastech/crystal@1.0
version: 2.1
```

Yeah! That was simple! With Orbs an abstraction layer is built so that the configuration file is more readable and intuitive.

In case we are wondering what the job [crystal/test](#) does, we always may see the source code.

### Using `nightly`

Using nightly Crystal release is as easy as:

```
# .circleci/config.yml
workflows:
  version: 2
  build:
    jobs:
      - crystal/test:
          name: test-on-nightly
          executor:
            name: crystal/default
            tag: nightly

orbs:
  crystal: manastech/crystal@1.0
version: 2.1
```

## Using a specific Crystal release

```
# .circleci/config.yml
workflows:
  version: 2
  build:
    jobs:
      - crystal/test:
          name: test-on-0.30
          executor:
            name: crystal/default
            tag: 0.30.0

orbs:
  crystal: manastech/crystal@1.0
version: 2.1
```

## Installing shards packages

You need not worry about it since the `crystal/test` job runs the `crystal/shard-install` orb command.

## Installing binary dependencies

Our application or maybe some shards may require libraries and packages. This binary dependencies may be installed using the [Apt](#) command.

Here is an example installing the `libsqlite3` development package:

```
# .circleci/config.yml
workflows:
  version: 2
  build:
    jobs:
      - crystal/test:
          pre-steps:
            - run: apt-get update && apt-get install -y libsqlite3-dev

orbs:
  crystal: manastech/crystal@1.0
version: 2.1
```

## Using services

Now, let's run specs using an external service (for example MySQL):

```
# .circleci/config.yml
executors:
  crystal_mysql:
    docker:
      - image: 'crystallang/crystal:latest'
        environment:
          DATABASE_URL: 'mysql://root@localhost/db'
      - image: 'mysql:5.7'
        environment:
          MYSQL_DATABASE: db
          MYSQL_ALLOW_EMPTY_PASSWORD: 'yes'

workflows:
  version: 2
  build:
    jobs:
      - crystal/test:
          executor: crystal_mysql
          pre-steps:
            - run:
                name: Waiting for service to start (check dockerize)
                command: sleep 1m
            - checkout
            - run:
                name: Install MySQL CLI; Import dummy data
                command: |
                  apt-get update && apt-get install -y mysql-client
                  mysql -h 127.0.0.1 -u root --password="" db < test-data/setup.

orbs:
  crystal: manastech/crystal@1.0
version: 2.1
```

**Note:** The explicit `checkout` in the `pre-steps` is to have the `test-data/setup.sql` file available.

## Caching

Caching is enabled by default when using the job `crystal/test`, because internally it uses the command `with-shards-cache`