The Golo Programming Language

Julien Ponge

The Golo Programming Language

Julien Ponge

Table of Contents

This is the documentation for the Golo programming language.

> This is a work in progress. Golo is even far from being feature-complete. You have been warned, but hey, you somehow knew it I'm pretty sure.

Copyright and License Notice

```
Copyright 2012-2014 Institut National des Sciences Appliquées de L

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, softwar
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or im
See the License for the specific language governing permissions an
limitations under the License.
```

Chapter 1. Basics

Let us start with the Golo basics.

1.1. Editor / IDE support

Editor and IDE support for Golo is available for:

- Vim
- Sublime Text 2
- IntelliJ IDEA (syntax highthing)
- Eclipse (contributed by Jeff Maury)
- Netbeans (contributed by Serli)

1.2. Hello world

Golo source code need to be placed in *modules*. Module names are separated with dots, as in:

```
Foo
foo.Bar
foo.bar.Baz
(...)
```

It is suggested yet not enforced that the first elements in a module name are in low-ercase, and that the last one have an uppercase first letter.

A Golo module can be executable if it has a *function* named `main` and that takes an argument for the JVM program arguments:

```
module hello.World

function main = |args| {
  println("Hello world!")
}
```

`println` is a predefined function that outputs a value to the standard console. As you can easily guess, here we output `Hello, world!` and that is an awesome achievement.

> Newlines are important in Golo, so make sure that your editor ends files with a newline.

1.3. Running *"Hello world"*

Of course, we need to run this incredibly complex application.

Golo comes with a `golo` script found in the distribution `bin/` folder. It provides several commands, notably:

- `version` to query the Golo version, and
- `compile` to compile some Golo code to JVM classes, and
- `run` to execute some already compiled Golo code, and
- `golo` to directly execute Golo code from source files, and
- `diagnose` to print compiler internal diagnosis information, and
- `doc` to generate module(s) documentation, and
- `new` to generate new project(s).

The complete commands usage instructions can be listed by running `golo --help`.

> The `golo` script comes with JVM tuning settings that may not be appropriate to your environment. We also provide a `vanilla-golo` script with no tuning. You may use the `$JAVA_OPTS` environment variable to provide custom JVM tuning to `vanilla-golo`.

Provided that `golo` is available from your current `$PATH`, you may run the program above as follows:

```
$ golo golo --files samples/helloworld.golo
Hello world!
$
```

`golo golo` takes several Golo source files as input. It expects the last one to have a `main` function to call. The Golo code is compiled on the fly and executed straight into a JVM.

You may also pass arguments to the `main` function by appending `--args` on the command line invocation. Suppose that we have a module `EchoArgs` as follows:

```
module EchoArgs

function main = |args| {
  foreach arg in args {
    println("->  " + arg)
  }
}
```

We may invoke it as follows:

```
$ golo golo --files samples/echo-args.golo --args plop da plop
-> plop
-> da
-> plop
```

```
$
```

Note that `args` is expected to be an array.

Finally, the `--classpath` flag allows to specify a list of classpath elements, which can be either directories or `.jar` files. See the `golo help` command for details on the various Golo commands.

1.4. Compiling Golo source code

Golo comes with a compiler that generates JVM bytecode in `.class` files. We will give more details in the chapter on interoperability with Java.

Compiling Golo files is straightforward:

```
$ golo compile --output classes samples/helloworld.golo
$
```

This compiles the code found in `samples/helloworld.golo` and outputs the generated classes to a `classes` folder (it will be created if needed):

```
$ tree classes/
classes/
### hello
    ### World.class

1 directory, 1 file
$
```

1.5. Running compiled Golo code

Golo provides a `golo` command for running compiled Golo code:

```
$ cd classes
$ golo run --module hello.World
Hello world!
$
```

Simple, isn't it?

1.6. Passing JVM-specific flags

Both `golo` and `run` commands can be given JVM-specific flags using the `JAVA_OPTS` environment variable.

As an example, the following runs `fibonacci.golo` and prints JIT compilation along the way:

```
# Exporting an environment variable
$ export JAVA_OPTS=-XX:+PrintCompilation
$ golo golo --files samples/fibonacci.golo

# ...or you may use this one-liner
$ JAVA_OPTS=-XX:+PrintCompilation golo golo --files samples/fibona
```

1.7. Bash autocompletion

A bash script can be found in `share/shell-completion/` called `golo-bash-completion` that will provide autocomplete support for the `golo` and `vanilla-golo` CLI scripts. You may either `source` the script, or drop the script into your `bash_completion.d/` folder and restart your terminal.

> Not sure where your `bash_completion.d/` folder is? Try `/etc/bash_completion.d/` on Linux or `/usr/local/etc/bash_completion.d/` for Mac Homebrew users.

1.8. Zsh autocompletion

A zsh script can be found in `share/shell-completion/` called `golo-zsh-completion` that works using the `golo-bash-completion` to provide autocomplete support using the bash autocomplete support provided by zsh. Place both files into the same directory and `source golo-zsh-completion` from your terminal or `.zshrc` to give it a try!

1.9. Comments

Golo comments start with a #, just like in Bash, Python or Ruby:

```
# This is a comment
println("WTF?") # it works here, too
```

1.10. Variable and constant references

Golo does not check for types at compile time, and they are not declared. Everything happens at runtime in Golo.

Variables are declared using the `var` keyword, while constant references are declared with `let`. It is strongly advised that you favour `let` over `var` unless you are certain that you need mutability.

Variables and constants need to be initialized when declared. Failing to do so results in a compilation error.

Here are a few examples:

```
# Ok
var i = 3
i = i + 1

# The assignment fails because truth is a constant
let truth = 42
truth = 666

# Invalid statement, variables / constants have to be initialized
var foo
```

Valid names contain upper and lower case letters within the `[a..z]` range, underscores (_), dollar symbols ($) and numbers. In any case, an identifier must not start with a number.

```
# Ok, but not necessarily great for humans...
let _$_f_o_$$666 = 666

# Wrong!
let 666_club = 666
```

1.11. Data literals

Golo supports a set of data literals. They directly map to their counterparts from the Java Standard API. We give them along with examples in the data literals table below.

Java type	Golo literals
`null`	`null`
`java.lang.Boolean`	`true` or `false`
`java.lang.String`	`"hello world"`
`java.lang.Character`	`'a'`, `'b'`, ...
`java.lang.Integer`	`123`, `-123`, `1_234`, ...
`java.lang.Long`	`123_L`, `-123_L`, `1_234_L`, ...
`java.lang.Double`	`1.234`, `-1.234`, `1.234e9`, ...
`java.lang.Float`	`1.234_F`, `-1.234_F`, `1.234e9_F`, ...

Java type	Golo literals

`java.lang.Class`	`String.class,`
	`java.lang.String.class,`
	`gololang.Predef.module,...`

`java.lang.invoke.MethodHandle`	`^foo, ^some.module::foo,...`

Speaking of strings, Golo also supports multi-line strings using the `"""` delimiters, as in:

```
let text = """This is
a multi-line string.
  How
    cool
      is
        that?"""
```

```
println(text)
```

This snippet would print the following to the standard console output:

```
This is
a multi-line string.
  How
    cool
      is
        that?
```

1.12. Collection literals

Golo support special support for common collections. The syntax uses brackets prefixed by a collection name, as in:

```
let s = set[1, 2, "a", "b"]
let v = vector[1, 2, 3]
let m = map[[1, "a"], [2, "b"]]
# (...)
```

The syntax and type matchings are the following:

Collection	Java type	Syntax
Tuple	`gololang.Tuple`	`tuple[1, 2, 3]`, or simply `[1, 2, 3]`
Array	`java.lang.Object[]`	`array[1, 2, 3]`
List	`java.util.LinkedList`	`list[1, 2, 3]`
Vector	`java.util.ArrayList`	`vector[1, 2, 3]`

Collection	Java type	Syntax
Set	`java.util.LinkedHashSet`	`set[1, 2, 3]`
Map	`java.util.LinkedHashMap`	`map[[1, "a"], [2, "b"]]`

1.12.1. A note on tuples

Tuples essentially behave as immutable arrays.

The `gololang.Tuple` class provides the following methods:

- a constructor with a variable-arguments list of values,
- a `get(index)` method to get the element at a specified index,
- `size()` and `isEmpty()` methods that do what their names suggest,
- an `iterator()` method because tuples are iterable, and
- `equals(other)`, `hashCode()` and `toString()` do just what you would expect.

1.12.2. A note on maps

The map collection literal expects entries to be specified as tuples where the first entry is the key, and the second entry is the value. This allows nested structures to be specified as in:

```
map[
  ["foo", "bar"],
  ["plop", set[1, 2, 3, 4, 5]],
  ["mrbean", map[
    ["name", "Mr Bean"],
    ["email", "bean@outlook.com"]
  ]]
]
```

There are a few rules to observe:

- not providing a series of tuples will yield class cast exceptions,
- tuples must have at least 2 entries or will yield index bound exceptions,
- tuples with more than 2 entries are ok, but only the first 2 entries matter.

Because of that, the following code compiles but raises exceptions at runtime:

```
let m1 = map[1, 2, 4, 5]
let m2 = map[
  [1],
  ["a", "b"]
]
```

The rationale for map literals to be loose is that we let you put any valid Golo expression, like functions returning valid tuples:

```
let a = -> [1, 'a']
let b = -> [2, 'b']
let m = map[a(), b()]
```

1.13. Operators

Golo supports the following set of operators.

Symbol(s)	Description	Examples
+	Addition on numbers and strings.	`1 + 2` gives 3. `"foo" + "bar"` gives `"foobar"`. `"foo" + something` where `something` is any object instance is equivalent to `"foo" + something.toString()` in Java.
-	Subtraction on numbers.	`4 - 1` gives 3.
*	Multiplication on numbers and strings.	`2 * 2` gives 4. `"a" * 3` gives `"aaa"`.
/	Division on numbers.	`4 / 2` gives 2.
%	Modulo on numbers.	`4 % 2` gives 0, `3 % 2` gives 1.
`"<"`, `"<="`, `"=="`, `"!="`, `">"`, `">="`	Comparison between numbers and objects that implement `java.lang.Comparable`. == is equivalent to calling `Object#equals(Object)` in Java.	`1 < 2` gives `true`.
`is`, `isnt`	Comparison of reference equality.	`a is b` gives `true` only if `a` and `b` reference the same object instance.
`and`, `or`, `not`	Boolean operators. `not` is of course a unary operator.	`true and true` gives `true`, `not(true)` gives `false`.
`oftype`	Checks the type of an object instance, equivalent to the `instanceof` operator in Java.	`("plop" oftype String.class)` gives `true`.

Symbol(s)	Description	Examples
`orIfNull`	Evaluates an expression and returns the value of another one if `null`.	`null orIfNull "a"` gives `"a"`. `foo() orIfNull 0` gives the value of calling `foo()`, or `0` if `foo()` returns `null`.

1.14. Calling a method

Although we will discuss this in more details later on, you should already know that `:` is used to invoke instance methods.

You could for instance call the `toString()` method that any Java object has, and print it out as follows:

```
println(123: toString())
println(someObject: toString())
```

1.15. Java / JVM arrays

As you probably know, arrays on the JVM are special objects. Golo deals with such arrays as being instances of `Object[]` and does not provide a wrapper class like many languages do. A Java / JVM array is just what it is supposed to be.

Golo adds some sugar to relieve the pain of working with arrays. Golo allows some special methods to be invoked on arrays:

- `get(index)` returns the value at `index`,
- `set(index, value)` sets `value` at `index`,
- `length()` and `size()` return the array length,
- `iterator()` returns a `java.util.Iterator`,
- `toString()` delegates to `java.util.Arrays.toString(Object[])`,
- `asList()` delegates to `java.util.Arrays.asList(Object[])`,
- `equals(someArray)` delegates to `java.util.Arrays.equals(this, someArray)`,
- `getClass()` return the array class.

Given a reference `a` on some array:

```
# Gets the element at index 0
a: get(0)

# Replaces the element at index 1 with "a"
a: set(1, "a")
```

```
# Nice print
println(a: toString())

# Convert to a real collection
let list = a: asList()
```

The methods above do **not** perform array bound checks.

Finally, arrays can be created with the `Array` function, as in:

```
let a = Array(1, 2, 3, 4)
let b = Array("a", "b")
```

You can of course take advantage of the `array` collection literal, too:

```
let a = array[1, 2, 3, 4]
let b = array["a", "b"]
```

Chapter 2. Creating new project(s)

The `golo new` command can create new Golo project(s):

```
$ golo new Foo
```

The command creates a new Golo module named `Foo` in a `main.golo` file with a simple *function* named `main` that takes an argument for the JVM program arguments.

By default we create a new free-form project but you can specify the type of project with the `--type` command argument. Three types of projects are currently available:

- Free-form project,
- Maven-driven project,
- Gradle-driven project.

As an example if you want to create a Maven-driven project, just add `--type maven`:

```
$ golo new Foo --type maven
```

By default we create the project directory where the `golo` command is run. If you need to create your project directory elsewhere you can use the `--path` command argument:

```
$ golo new Bar --path /opt/golo
```

This creates the project directory named `Bar` in `/opt/golo`.

2.1. Free-form project

The structure of a free-form project is as follows:

```
$ tree Foo
Foo
### imports
### jars
### main.golo
```

2.2. Maven-driven project

The structure of a Maven-driven project is as follows:

```
$ tree Foo
Foo
### pom.xml
```

```
### src
    ### main
        ### golo
            ### main.golo
```

The project can be built and packaged with Maven using the following command:

```
$ mvn package
```

You can now run the module `Foo` with:

- `mvn`

```
$ mvn exec:java
```

- `java`

```
$ java -jar target/Foo-*-jar-with-dependencies.jar
```

- `golo`

```
$ cd target/classes
$ golo run --module Foo
```

2.3. Gradle-driven project

The structure of a Gradle-driven project is as follows:

```
$ tree Foo
Foo
### build.gradle
### src
    ### main
        ### golo
            ### main.golo
```

The project can be built and packaged with Gradle using the following command:

```
$ gradle build
```

You can now run the module `Foo` with:

- `gradle`

```
$ gradle run
```

- `golo`

```
$ cd build/classes/main
$ golo run --module Foo
```

Chapter 3. Functions

Functions are first-class citizen in Golo. Here is how to define and call some.

3.1. Parameter-less functions

Golo modules can define functions as follows:

```
module sample

function hello = {
  return "Hello!"
}
```

In turn, you may invoke a function with a familiar notation:

```
let str = hello()
```

A function needs to return a value using the `return` keyword. Some languages state that the last statement is the return value, but Golo does not follow that trend. We believe that `return` is more explicit, and that a few keystrokes in favour of readability is still a good deal.

Still, you may omit `return` statements if your function does not return a value:

```
function printer = {
  println("Hey!")
}
```

If you do so, the function will actually return `null`, hence `result` in the next statement is `null`:

```
# result will be null
let result = printer()
```

3.2. Functions with parameters

Of course functions may take some parameters, as in:

```
function addition = |a, b| {
  return a + b
}
```

> Parameters are constant references, hence they cannot be reassigned.

Invoking functions that take parameters is straightforward, too:

```
let three = addition(1, 2)
let hello_world = addition("hello ", "world!")
```

3.3. Variable-arity functions

Functions may take a varying number of parameters. To define one, just add `...` to the last parameter name:

```
function foo = |a, b, c...| {
  # ...
}
```

Here, `c` catches the variable arguments in an array, just like it would be the case with Java. You can thus treat `c` as being a Java object of type `Object[]`.

Calling variable-arity functions does not requiring wrapping the last arguments in an array. While invoking the `foo` function above, the following examples are legit:

```
# a=1, b=2, c=[]
foo(1, 2)

# a=1, b=2, c=[3]
foo(1, 2, 3)

# a=1, b=2, c=[3,4]
foo(1, 2, 3, 4)
```

Because the parameter that catches the last arguments is an array, you may call array methods. Given:

```
function elementAt = |index, args...| {
  return args: get(index)
}
```

then:

```
# prints "2"
println(elementAt(1, 1, 2, 3))
```

3.4. Functions from other modules and imports

Suppose that we have a module `foo.Bar`:

```
module foo.Bar

function f = {
  return "f()"
```

```
}
```

We can invoke `f` from another module by prefixing it with its module name:

```
let r = foo.Bar.f()
```

Of course, we may also take advantage of an `import` statement:

```
module Somewhere.Else

import foo.Bar

function plop = {
  return f()
}
```

> Imports in Golo do not work as in Java. Golo is a dynamic language where symbols are being resolved at runtime. Module imports are **not** checked at compilation time, and their sole purpose is to help in dynamic resolution. Back to the previous example, `f` cannot be resolved from the current module, and the Golo runtime subsequently tries to resolve `f` from each `import` statement. Also, note that the order of `import` statements is important, as the resolution stops at the first module having the `f` function.

Last but not least, you may prepend the last piece of the module name. The following invocations are equivalent:

```
module Somewhere.Else

import foo.Bar

function plop = {
  let result = f()
  let result_bis = Bar.f()
  let result_full = foo.Bar.f()
  return result
}
```

Golo modules have a set of implicit imports:

- `gololang.Predefined,`
- `gololang.StandardAugmentations,`
- `gololang,`
- `java.lang.`

3.5. Local functions

By default, functions are visible outside of their module. You may restrict the visibility of a function by using the `local` keyword:

```
module Foo

local function a = {
  return 666
}

function b = {
  return a()
}
```

Here, b is visible while a can only be invoked from within the Foo module. Given another module called Bogus, the following would fail at runtime:

```
module Bogus

function i_will_crash = {
  return Foo.a()
}
```

Chapter 4. Java interoperability

Golo aims at providing a seamless 2-way interoperability with the Java programming language.

4.1. Main function Java compliance

If the Golo compiler find a unary function named `main`, it will be compiled to a `void(String[])` static method. This `main` method can servers as a JVM entry point.

Suppose that we have the following Golo module:

```
module mainEntryPoint


function main = |args| {
  println("-> " + args: get(0))
}
```

Once compiled, we may invoke it as follows:

```
$ golo compile mainEntryPoint.golo
$ java -cp ".:golo.jar" mainEntryPoint GoloRocks
-> GoloRocks
$
```

4.2. Calling static methods

Golo can invoke public Java static methods by treating them as functions:

```
module sample

import java.util.Arrays

function oneTwoThree = {
  return asList(1, 2, 3)
}
```

In this example, `asList` is resolved from the `java.util.Arrays` import and called as a function. Note that we could equivalently have written a qualified invocation as `Arrays.asList(1, 2, 3)`.

4.3. Calling instance methods

When you have an object, you may invoke its methods using the ：operator.

The following would call the `toString` method of any kind, then print it:

```
println(">>> " + someObject: toString())
```

Of course, you may chain calls as long as a method is not of a `void` return type. Golo converts Java `void` methods by making them return `null`. This is neither a bug or a feature: the **invokedynamic** support on the JVM simply does so.

4.4. `null`-safe instance method invocations

Golo supports **null-safe** methods invocations using the **"Elvis"** symbol: `?:`.

Suppose that we invoke the method `bar()` on some reference `foo`: `foo:    bar()`. If `foo` is `null`, then invoking `bar()` throws a `java.lang.NullPointerException`, just like you would expect in Java.

By contrast:

- `foo?: bar()` simply returns `null`, and
- `null?: anything()` returns `null`, too.

This is quite useful when querying data models where `null` values could be returned. This can be elegantly combined with the `orIfNull` operator to return a default value, as illustrated by the following example:

```
let person = dao: findByName("Mr Bean")
let city = person?: address()?: city() orIfNull "n/a"
```

This is more elegant than, say:

```
let person = dao: findByName("Mr Bean")
var city = "n/a"
if person isnt null {
  let address = person: address()
  if address isnt null {
    city = address: city() ofIfNull "n/a"
  }
}
```

> The runtime implementation of `null`-safe method invocations is **optimistic** as it behaves like a `try` block catching a `NullPointerException`. Performance is good unless most invocations happen to be on `null`, in which case using `?:` is probably not a great idea.

4.5. Creating objects

Golo doesn't have an instantiation operator like `new` in Java. Instead, creating an object and calling its constructor is done as if it was just another function.

As an example, we may allocate a `java.util.LinkedList` as follows:

```
module sample

import java.util

function aList = {
  return LinkedList()
}
```

Another example would be using a `java.lang.StringBuilder`.

```
function str_build = {
  return java.lang.StringBuilder("h"):
    append("e"):
    append("l"):
    append("l"):
    append("o"):
    toString()
}
```

As one would expect, the `str_build` function above gives the `"hello"` string.

4.6. Static fields

Golo treats public static fields as function, so one could get the maximum value for an `Integer` as follows:

```
module samples.MaxInt

local function max_int = {
  return java.lang.Integer.MAX_VALUE()
}

function main = |args| {
  println(max_int())
}
```

> Given than most static fields are used as constants in Java, Golo does not provide support to change their values. This may change in the future if compelling general-interest use-cases emerge.

4.7. Instance fields

Instance fields can be accessed as functions, both for reading and writing. Suppose that we have a Java class that looks as follows:

```
public class Foo {
  public String bar;
}
```

We can access the `bar` field as follows:

```
let foo = Foo()

# Write
foo: bar("baz")

# Read, prints "baz"
println(foo: bar())
```

An interesting behavior when writing fields is that the **"methods"** return the object, which means that you can chain invocations.

Suppose that we have a Java class as follows:

```
public class Foo {
  public String bar;
  public String baz;
}
```

We can set all fields by chaining invocations as in:

```
let foo = Foo(): bar(1): baz(2)
```

It should be noted that Golo won't bypass the regular Java visibility access rules on fields.

> ### What happens if there is both a field and a method with the same names?
>
> Back to the previous example, suppose that we have both a field and a method with the same name, as in:
>
> ```
> public class Foo {
>   public String bar;
>
>   public String bar() {
>     return bar;
>   }
> }
> ```
>
> **Golo resolves methods first, fields last.** Hence, the following Golo code will resolve the `bar()` method, not the `bar` field:
>
> ```
> let foo = Foo()
> ```

```
# Write the field
foo: bar("baz")

# Calls the bar() method
println(foo: bar())
```

4.8. Inner classes and enumerations

We will illustrate both how to deal with public static inner classes and enumerations at once.

The rules to deal with them in Golo are as follows.

1. Inner classes are identified by their real name in the JVM, with nested classes being separated by a $ sign. Hence, `Thread.State` in Java is written `Thread$State` in Golo.
2. Enumerations are just normal objects. They expose each entry as a static field, and each entry is an instance of the enumeration class.

Let us consider the following example:

```
module sample.EnumsThreadState

import java.lang.Thread$State

function main = |args| {

  # Call the enum entry like a function
  let new = Thread$State.NEW()
  println("name=" + new: name() + ", ordinal=" + new: ordinal())

  # Walk through all enum entries
  foreach element in Thread$State.values() {
    println("name=" + element: name() + ", ordinal=" + element: or
  }
}
```

Running it yields the following console output:

```
$ golo golo --files samples/enums-thread-state.golo
name=NEW, ordinal=0
name=NEW, ordinal=0
name=RUNNABLE, ordinal=1
name=BLOCKED, ordinal=2
name=WAITING, ordinal=3
name=TIMED_WAITING, ordinal=4
name=TERMINATED, ordinal=5
$
```

4.9. Clashes with Golo operators and escaping

Because Golo provides a few named operators such as `is`, `and` or `not`, they are recognized as operator tokens.

However, you may find yourself in a situation where you need to invoke a Java method whose name is a Golo operator, such as:

```
# Function call
is()

# Method call
someObject: foo(): is(): not(): bar()
```

This results in a parsing error, as `is` and `not` will be matched as operators instead of method identifiers.

The solution is to use **escaping**, by prefixing identifiers with a backtick, as in:

```
# Function call
`is()

# Method call
someObject: foo(): `is(): `not(): bar()
```

4.10. Golo class loader

Golo provides a class loader for directly loading and compiling Golo modules. You may use it as follows:

```
import fr.insalyon.citi.golo.compiler.GoloClassLoader;

public class Foo {

  public static void main(String... args) throws Throwable {
    GoloClassLoader classLoader = new GoloClassLoader();
    Class<?> moduleClass = classLoader.load("foo.golo", new FileIn
    Method bar = moduleClass.getMethod("bar", Object.class);
    bar.invoke(null, "golo golo");
  }
}
```

This would work with a Golo module defined as in:

```
module foo.Bar

function bar = |wat| -> println(wat)
```

Indeed, a Golo module is viewable as a Java class where each function is a static method.

> `GoloClassLoader` is rather dumb at this stage, and you will get an exception if you try to load two Golo source files with the same `module` name declaration. This is because it will attempt to redefine an already defined class.

> Later in the glorious and glamorous future, Golo will have objects and not just functions. Be patient, it's coming in!

Chapter 5. Control flow

Control flow in Golo is imperative and has the usual constructions found in upstream languages.

5.1. Conditional branching

Golo supports the traditional `if` / `else` constructions, as in:

```
if goloIsGreat() {
  println("Golo Golo")
}

if (someCondition) {
  doThis()
} else if someOtherCondition {
  doThat()
} else {
  doThatThing()
}
```

The condition of an `if` statement does not need parenthesis. You may add some to clarify a more elaborated expression, though.

5.2. `case` branching

Golo offers a versatile `case` construction for conditional branching. It may be used in place of multiple nested `if` / `else` statements, as in:

```
function what = |obj| {
  case {
    when obj oftype String.class {
      return "String"
    }
    when obj oftype Integer.class {
      return "Integer"
    }
    otherwise {
      return "alien"
    }
  }
}
```

A `case` statement requires at least 1 `when` clause and a mandatory `otherwise` clause. Each clause is being associated with a block. It is semantically equivalent to the corresponding `if` / `else` chain:

```
function what = |obj| {
  if obj oftype String.class {
    return "String"
  } else if obj oftype Integer.class {
    return "Integer"
  } else {
    return "alien"
  }
}
```

when clauses are being evaluated in the declaration order, and only the first satisfied one is being executed.

5.3. `match` statements

The `match` statement is a convenient shortcut for cases where a `case` statement would be used to match a value, and give back a result. While it may resemble **pattern matching** operators in some other languages it is not fully equivalent, as Golo does not support destructuring.

`match` is a great addition to the Golo programmer:

```
let item = "foo@bar.com"

let what_it_could_be = -> match {
  when item: contains("@") then "an email?"
  when item: startsWith("+33") then "a French phone number?"
  when item: startsWith("http://") then "a website URL?"
  otherwise "I have no clue, mate!"
}

# prints "an email?"
println(what_it_could_be(item))
```

The values to be returned are specified after a `then` keyword that follows a boolean expression to be evaluated.

Like `case` statements, a `match` construct needs at least one `when` clause and one `otherwise` clause.

5.4. `while` loops

While loops in Golo are straightforward:

```
function times = |n| {
```

```
    var times = 0
    while (times < n) { times = times + 1 }
    return times
}
```

The parenthesis in the `while` condition may be omitted like it is the case for `if` statements.

5.5. `for` loops

This is the most versatile loop construction, as it features:

1. a variable declaration and initialization (a Golo variable is always initialized anyway), and
2. a loop progress condition, and
3. a loop progress statement.

The following function shows a `for` loop:

```
function fact = |value, n| {
  var result = 1
  for (var i = 0, i < n, i = i + 1) {
    result = result * value
  }
  return result
}
```

As you can see, it is very much like a `for` loop in Java, except that:

- the `for` loop elements are separated by ',' instead of ';', and
- there cannot be multiple variables in the loop, and
- there cannot be multiple loop progress statements.

Again, this choice is dictated by the pursue of simplicity.

5.6. `foreach` loops

Golo provides a "for each" style of iteration over iterable elements. Any object that is an instance of `java.lang.Iterable` can be used in `foreach` loops, as in:

```
function concat_to_string = |iterable| {
  var result = ""
  foreach item in iterable {
    result = result + item
  }
  return result
```

}

In this example, `item` is a variable within the `foreach` loop scope, and `iterable` is an object that is expected to be iterable.

You may use parenthesis around a `foreach` expression, so `foreach (foo in bar)` is equivalent to `foreach foo in bar`.

> Although Java arrays (`Object[]`) are not real objects, they can be used with `foreach` loops. Golo provides a `iterator()` method for them.

5.7. `break` and `continue`

Although not strictly necessary, the `break` and `continue` statements can be useful to simplify some loops in imperative languages.

Like in Java and many other languages:

- `break` exits the current inner-most loop, and
- `continue` skips to the next iteration of the current inner-most loop.

Consider the following contrived example:

```
module test

function main = |args| {
  var i = 0
  while true {
    i = i + 1
    if i < 40 {
      continue
    } else {
      print(i + " ")
    }
    if i == 50 {
      break
    }
  }
  println("bye")
}
```

It prints the following output:

```
40 41 42 43 44 45 46 47 48 49 50 bye
```

Golo does not support `break` statements to labels like Java does. In fact, this is a `goto` statement in disguise.

5.8. Why no value from most control flow constructions?

Some programming languages return values from selected control flow constructions, with the returned value being the evaluation of the last statement in a block. This can be handy in some situations such as the following code snippet in Scala:

```
println(if (4 % 2 == 0) "even" else "odd")
```

The Golo original author recognizes and appreciates the expressiveness of such construct. However, he often finds it harder to spot the returned values with such constructs, and he thought that trading a few keystrokes for **explicitness** was better than shorter construct based in **implicitness**.

Therefore, most Golo control flow constructions do not return values, and programmers are instead required to extract a variable or provide an explicit return statement.

Chapter 6. Exceptions

Exception handling in Golo is simple. There is no distinction between **checked** and **unchecked** exceptions.

6.1. Raising exceptions

Golo provides 2 predefined functions for raising exceptions:

- `raise(message)` throws a `java.lang.RuntimeException` with a message given as a string, and
- `raise(message, cause)` does the same and specifies a cause which must be an instance of `java.lang.Throwable`.

Throwing an exception is thus as easy as:

```
if somethingIsWrong() {
  raise("Woops!")
}
```

6.2. Raising specialized exceptions

Of course not every exception shall be an instance of `java.lang.RuntimeException`. When a more specialized type is required, you may simply instantiate a Java exception and throw it using the `throw` keyword as in the following example:

```
module golotest.execution.Exceptions

import java.lang.RuntimeException

function runtimeException = {
  throw RuntimeException("w00t")
}
```

6.3. Exception handling

Exception handling uses the familiar `try / catch`, `try / catch / finally` and `try / finally` constructions. Their semantics are the same as found in other languages such as Java, especially regarding the handling of `finally` blocks.

The following snippets show each exception handling form.

```
# Good old try / catch
```

```
try {
  something()
} catch (e) {
  e: printStackTrace()
}

# A try / finally
try {
  doSomething()
} finally {
  cleanup()
}

# Full try / catch / finally construct
try {
  doSomething()
} catch (e) {
  e: printStackTrace()
  case {
    when e oftype IOException.class {
      println("Oh, an I/O exception that I was expecting!")
    }
    when e oftype SecurityException.class {
      println("Damn, I didn't expect a security problem...")
      throw e
    }
    otherwise {
      throw e
    }
  }
} finally {
  cleanup()
}
```

Because Golo is a weakly typed dynamic language, you need to check for the exception type with the `oftype` operator. In a statically typed language like Java, you would instead have several `catch` clauses with the exception reference given a specific type. We suggest that you take advantage of the `case` branching statement.

Chapter 7. Closures

Golo supports **closures**, which means that functions can be treated as first-class citizen.

7.1. Defining and using a closure

Defining a closure is straightforward as it derives from the way a function can be defined:

```
let adder = |a, b| {
  return a + b
}
```

At runtime, a closure is an instance of `java.lang.invoke.MethodHandle`. This means that you can do all the operations that method handles support, such as invoking them or inserting arguments as illustrated in the following example:

```
let adder = |a, b| {
  return a + b
}
println(adder: invokeWithArguments(1, 2))

let addToTen = adder: bindTo(10)
println(addToTen: invokeWithArguments(2))
```

As one would expect, this prints 3 and 12.

7.2. Compact closures

Golo supports a compact form of closures for the cases where their body consists of a single expression. The example above can be simplified as:

```
let adder = |a, b| -> a + b
```

You may also use this compact form when defining regular functions, as in:

```
module Foo

local function sayHello = |who| -> "Hello " + who + "!"

# Prints "Hello Julien!"
function main = |args| {
  println(sayHello("Julien"))
}
```

7.3. Calling closures

While you may take advantage of closures being method handles and call them using `invokeWithArguments`, there is a (much) better way.

When you have a reference to a closure, you may simply call it as a regular function. The previous `adder` example can be equivalently rewritten as:

```
let adder = |a, b| -> a + b
println(adder(1, 2))

let addToTen = adder: bindTo(10)
println(addToTen(2))
```

7.4. Limitations

Closures have access to the lexical scope of their defining environment. Consider this example:

```
function plus_3 = {
  let foo = 3
  return |x| -> x + foo
}
```

The `plus_3` function returns a closure that has access to the `foo` reference, just as you would expect. The `foo` reference is said to have been **captured** and made available in the closure.

It is important to **note that captured references are constants within the closure**. Consider the following example:

```
var a = 1
let f = {
  a = 2   # Compilation error!
}
```

The compilation fails because although `a` is declared using `var` in its original scope, it is actually passed as an argument to the `f` closure. Because function parameters are implicitly constant references, this results in a compilation error.

That being said, a closure has a reference on the same object as its defining environment, so a mutable object is a sensible way to pass data back from a closure as a side-effect, as in:

```
let list = java.util.LinkedList()
let pump_it = {
  list: add("I heard you say")
  list: add("Hey!")
```

```
    list: add("Hey!")
}
pump_it()
println(list)
```

which prints `[I heard you say, Hey!, Hey!]`.

7.5. Closures to single-method interfaces

The Java SE APIs have plenty of interfaces with a single method: `java.util.concurrent.Callable`, `java.lang.Runnable`, `javax.swing.ActionListener`, etc.

The predefined function `asInterfaceInstance` can be used to convert a method handle or Golo closure to an instance of a specific interface.

Here is how one could pass an action listener to a `javax.swing.JButton`:

```
let button = JButton("Click me!")
let handler = |event| -> println("Clicked!")
button: addActionListener(asInterfaceInstance(ActionListener.class
```

Because the `asInterfaceInstance` call consumes some readability budget, you may refactor it with a local function as in:

```
local function listener = |handler| -> asInterfaceInstance(ActionL

# (...)
let button = JButton("Click me!")
button: addActionListener(listener(|event| -> println("Clicked!"))
```

Here is another example that uses the `java.util.concurrent` APIs to obtain an executor, pass it a task, fetch the result with a `Future` object then shut it down:

```
function give_me_hey = {
  let executor = Executors.newSingleThreadExecutor()
  let future = executor: submit(asInterfaceInstance(Callable.class
  let result = future: get()
  executor: shutdown()
  return result
}
```

7.6. Direct closure passing works

When a function or method parameter of a Java API expects a single method interface type, you can pass a closure directly, as in:

```
# (...)
let button = JButton("Click me!")
button: addActionListener(|event| -> println("Clicked!"))
```

Note that this causes the creation of a method handle proxy object for each function or method invocation. For performance-sensitive contexts, we suggest that you use either `asInterfaceInstance` or the `to` conversion method described hereafter.

7.7. Conversion to single-method interfaces

Instead of using `asInterfaceInstance`, you may use a **class augmentation** which is described later in this documentation. In short, it allows you to call a `to` method on instances of `MethodHandle`, which in turn calls `asInterfaceInstance`. Back to the previous examples, the next 2 lines are equivalent:

```
# Calling asInterfaceInstance
future = executor: submit(asInterfaceInstance(Callable.class, -> "

# Using a class augmentation
future = executor: submit((-> "hey!"): to(Callable.class))
```

7.8. Getting a reference to a closure / Golo function

You may also take advantage of the predefined `fun` function to obtain a reference to a closure, as in:

```
import golotest.Closures

local function local_fun = |x| -> x + 1

function call_local_fun = {

  # local_fun, with a parameter
  var f = fun("local_fun", golotest.Closures.module, 1)

  # ...or just like this if there is only 1 local_fun definition
  f = fun("local_fun", golotest.Closures.module)

  return f(1)
}
```

Last but not least, we have an even shorter notation if function are not overridden:

```
import golotest.Closures

local function local_fun = |x| -> x + 1
```

```
function call_local_fun = {

  # In the current module
  var f = ^fun

  # ...or with a full module name
  f = ^golotest.Closures::fun

  return f(1)
}
```

7.9. Binding and composing

Because closure references are just instances of `java.lang.invoke.MethodHandle`, you can bind its first argument using the `bindTo(value)` method. If you need to bind an argument at another position than 0, you may take advantage of the `bindAt(position, value)` augmentation:

```
let diff = |a, b| -> a - b
let minus10 = diff: bindAt(1, 10)

# 10
println(minus10(20))
```

You may compose function using the `andThen` augmentation method:

```
let f = (|x| -> x + 1): andThen(|x| -> x - 10): andThen(|x| -> x *

# -500
println(f(4))
```

7.10. Calling functions that return functions

Given that functions are first-class objects in Golo, you may define functions (or closures) that return functions, as in:

```
let f = |x| -> |y| -> |z| -> -> x + y + z
```

You could use intermediate references to use the `f` function above:

```
let f1 = f(1)
let f2 = f1(2)
let f3 = f2(3)

# Prints '6'
```

```
println(f3())
```

Golo supports a nicer syntax if you don't need intermediate references:

```
# Prints '6'
println(f(1)(2)(3)())
```

> This syntax only works following a function or method invocation, not on expressions. This means that:
>
> ```
> foo: bar()("baz")
> ```
>
> is valid, while:
>
> ```
> (foo: bar())("baz")
> ```
>
> is not. Let us say that "It is not a bug, it is a feature".

Chapter 8. Predefined functions

Every Golo module definition comes with `gololang.Predefined` as a default import. It provides useful functions.

8.1. Console output

`print` and `println` do just what you would expect.

```
print("Hey")
println()

println("Hey")
```

8.2. Console input

`readln()` or `readln(strMessage)` reads a single line of text from the console. It always returns a string.

`readPassword()` or `readPassword(strPassword)` reads a password from the console with echoing disabled. It always returns a string. There are also `secureReadPassword()` and `secureReadPassword(strPassword)` variants that return a `char[]` array.

```
let name = readln("what's your name? ")
let value = readln()
let pwd = readpwd("type your password:")
```

8.3. Exceptions

`raise` can be used to throw a `java.lang.RuntimeException`. It comes in two forms: one with a message as a string, and one with a message and a cause.

```
try {
  ...
  raise("Somehow something is wrong")
} catch (e) {
  ...
  raise("Something was wrong, and here is the cause", e)
}
```

8.4. Preconditions

Preconditions are useful, especially in a dynamically-typed language.

`require` can check for a boolean expression along with an error message. In case of error, it throws an `AssertionError`.

```
function foo = |a| {
  require(a oftype String.class, "a must be a String")
  ...
}
```

You may also use `requireNotNull` that… well… checks that its argument is not null:

```
function foo = |a| {
  requireNotNull(a)
  ...
}
```

8.5. Arrays (deprecated)

Golo provides functions to deal with Java arrays (`Object[]`).

- the `Array` function takes a variable number of arguments and returns a Java array from them,
- the `aget` function takes an array and an index to return the element at that position,
- the `aset` function takes an array, an index and a value to set the element at that position,
- the `alength` function returns the length of an array,
- the `atoList` function calls the `java.util.Arrays.asList(values...)` method.

```
let a = Array(1, 2, 3)
require(alength(a) == 3, "a must be of length 3")
require(aget(a, 0) == 1, "the first element shall be 1")
aset(a, 0, 666)
require(aget(a, 0) == 666, "the new first element shall be 666")
```

Those functions were introduced for the needs of the early developments of Golo. They will be removed at some point before the release of version 0, so please use the corresponding array object methods instead: `get`, `set`, `length`, …

8.6. Ranges

The `range` function yields an iterable range over either `Integer` or `Long` bounds:

```
# Prints 1 2 (...) 100
```

```
foreach i in range(1, 101) {
  print(i + " ")
}

let r = range(0, 6): incrementBy(2)
println("Start: " + r: from())
println("End: " + r: to())
foreach i in r {
  println(i)
}

println("Increment: " + r: increment())
```

The lower bound is inclusive, the upper bound is exclusive.

8.7. Closures

Given a closure reference or a method handle, one can convert it to an instance of an interface with a single method declaration, as in:

```
local function listener = |handler| -> asInterfaceInstance(ActionL

# (...)
let button = JButton("Click me!")
button: addActionListener(listener(|event| -> println("Clicked!"))
```

It is possible to test if an object is a closure or not with the `isClosure` function. This is useful to support values and delayed evaluation, as in:

```
if isClosure(value) {
  map: put(key, value())
} else {
  map: put(key, value)
}
```

You can get a reference to a closure using the predefined `fun` function:

```
import golotest.Closures

local function local_fun = |x| -> x + 1

function call_local_fun = {
  let f = fun("local_fun", golotest.Closures.module)
  return f(1)
}
```

Because functions may be overloaded, there is a form that accepts an extra parameter for specifying the number of parameters:

```
import golotest.Closures

local function local_fun = |x| -> x + 1

function call_local_fun = {
  let f = fun("local_fun", golotest.Closures.module, 1)
  return f(1)
}
```

8.8. File I/O

Sometimes it is very desirable to read the content of a text file. The `fileToText` function does just that:

```
let text = fileToText("/some/file.txt", "UTF-8")
```

The first parameter is either a `java.lang.String`, a `java.io.File` or a `java.nio.file.Path`. The second parameter represents the encoding charset, either as a `java.lang.String` or a `java.nio.charset.Charset`.

We can write some text to a file, too:

```
textToFile("Hello, world!", "/foo/bar.txt")
```

The `textToFile` function overwrites existing files, and creates new ones if needed.

These functions are provided for convenience, so if you need more fine-grained control over reading and writing text then we suggest that you look into the `java.nio.file` package.

In addition, if you need to verify that a file exists, you can use the `fileExists` function.

```
if fileExists("/foo/bar.txt") {
  println("file found!")
}
```

As in the other File I/O methods, the parameter is either a `java.lang.String`, a `java.io.File` or a `java.nio.file.Path`. The `fileExists` function will return true if the file exists, false if it doesn't.

8.9. Array types

Golo does not provide a literal syntax for array types, such as `Object[].class` in Java.

Instead, we provide 3 helper functions.

- `isArray(object)`: returns a boolean if `object` is an array.
- `objectArrayType()`: returns `Object[].class`.
- `arrayTypeOf(type)`: given `type` as a `java.lang.Class`, returns an array of type `type[]`.

8.10. Misc.

`mapEntry` gives instances of `java.util.AbstractMap.SimpleEntry`, and is used as follows:

```
let e = mapEntry("foo", "bar")

# prints "foo => bar"
println(e: getKey() + " => " + e: getValue())
```

Chapter 9. Class augmentations

Many dynamic languages support the ability to extend existing classes by adding new methods to them. You may think of categories in Objective-C and Groovy, or **open classes** in Ruby.

This is generally implemented by providing **meta-classes**. When some piece of code adds a method `foo` to, say, `SomeClass`, then all instances of `SomeClass` get that new `foo` method. While very convenient, such an open system may lead to well-known conflicts between the added methods.

Golo provides a more limited but explicit way to add methods to existing classes in the form of **class augmentations**.

9.1. Wrapping a string with a function

Let us motivate the value of **augmentations** by starting with the following example. Suppose that we would like a function to wrap a string with a left and right string. We could do that in Golo as follows:

```
function wrap = |left, str, right| -> left + str + right

# (...)
let str = wrap("(", "foo", ")")
println(str) # prints "(abc)"
```

Defining functions for such tasks makes perfect sense, but what if we could just add the `wrap` method to all instances of `java.lang.String` instead?

9.2. Augmenting classes

Defining an augmentation is a matter of adding a `augment` block in a module:

```
module foo

augment java.lang.String {
  function wrap = |this, left, right| -> left + this + right
}

function wrapped = -> "abc": wrap("(", ")")
```

More specifically:

1. a `augment` definition is made on a fully-qualified class name, and
2. an augmentation function takes the receiver object as its first argument, followed by optional arguments, and

3. there can be as many augmentation functions as you want, and

4. there can be as many augmentations as you want.

It is a good convention to name the receiver `this`, but you are free to call it differently.

Also, augmentation functions can take variable-arity arguments, as in:

```
augment java.lang.String {

  function concatWith = |this, args...| {
    var result = this
    foreach(arg in args) {
      result = result + arg
    }
    return result
  }
}

# (...)
function varargs = -> "a": concatWith("b", "c", "d")
```

It should be noted that augmentations work with class hierarchies too. The following example adds an augmentation to `java.util.Collection`, which also adds it to concrete subclasses such as `java.util.LinkedList`:

```
augment java.util.Collection {
  function plop = |this| -> "plop!"
}

# (...)
function plop_in_a_list = -> java.util.LinkedList(): plop()
```

9.3. Augmentation scopes, reusable augmentations

By default, an augmentation is only visible from its defining module.

Augmentations are clear and explicit as they only affect the instances from which you have decided to make them visible.

It is advised to place reusable augmentations in separate module definitions. Then, a module that needs such augmentations can make them available through imports.

Suppose that you want to define augmentations for dealing with URLs from strings. You could define a `string-url-augmentations.golo` module source as follows:

```
module my.StringUrlAugmentations
```

```
import java.net

augment java.lang.String {

  function toURL = |this| -> URL(this)

  function httpGet = |this| {
    # Open the URL, get a connection, grab the body as a string, e
    # (...)
  }

  # (...)
}
```

Then, a module willing to take advantage of those augmentations can simply import their defining module:

```
module my.App

import my.StringUrlAugmentations

function googPageBody = -> "http://www.google.com/": httpGet()
```

> As a matter of style, we suggest that your module names end with `Augmentations`. Because importing a module imports **all** of its augmentation definitions, we suggest that you modularize them with **fine taste** (for what it means).

9.4. Standard augmentations

Golo comes with a set of pre-defined augmentations over collections, strings, closures and more.

These augmentation do not require a special import, and they are defined in the `gololang.StandardAugmentations` module.

Here is an example:

```
let odd = [1, 2, 3, 4, 5]: filter(|n| -> (n % 2) == 0)

let m = map[]
println(m: getOrElse("foo", -> "bar"))
```

The full set of standard augmentations is documented in the generated **golodoc** (hint: look for `doc/golodoc` in the Golo distribution).

Chapter 10. Structs

Golo allows the definition of simple structures using the `struct` keyword. They resemble structures in procedural languages such as C `struct` or Pascal **records**. They are useful to store data when the set of named entries is fixed.

10.1. Definition

Structures are defined at the module-level:

```
module sample

struct Person = { name, age, email }

function main = |args| {
  let p1 = Person("Mr Bean", 54, "bean@gmail.com")
  println(p1: name())
  let p2 = Person(): name("John"): age(32): email("john@b-root.com
  println(p2: age())
}
```

When declaring a structure, it also defines two factory functions: one with no arguments, and one with all arguments in their order of declaration in the `struct` statement. When not initialized, member values are `null`.

Each member yields a **getter** and a **setter** method: given a member `a`, the getter is method `a()` while the setter is method `a(newValue)`. It should be noted that setter methods return the structure instance which makes it possible to chain calls as illustrated in the previous example while building `p2`.

10.2. JVM existence

Each `struct` is compiled to a self-contained JVM class.

Given:

```
module sample

struct Point = { x, y }
```

a class `sample.types.Point` is being generated.

It is important to note that:

    1. each `struct` class is `final`,

2. each `struct` class inherits from `gololang.GoloStruct`,
3. proper definitions of `toString()`, `hashCode()` and `equals()` are being provided.

10.3. `toString()` behavior

The `toString()` method is being overridden to provide a meaningful description of a structure content.

Given the following program:

```
module test

struct Point = { x, y }

function main = |args| {
  println(Point(1, 2))
}
```

running it prints the following console output:

```
struct Point{x=1, y=2}
```

10.4. Immutable structs

Structure instances are mutable by default. Golo generates a factory function with the `Immutable` prefix to directly build immutable instances:

```
module test

struct Point = { x, y }

function main = |args| {

  let p = ImmutablePoint(1, 2)
  println(p)

  try {
    # Fails! (p is immutable)
    p: x(100)
  } catch (expected) {
    println(expected: getMessage())
  }
}
```

10.5. Copying

Instances of a structure provide copying methods:

- `copy()` returns a **shallow** copy of the structure instance, and
- `frozenCopy()` returns a read-only **shallow** copy.

Trying to invoke any setter methods on an instance obtained through `frozen-Copy()` raises a `java.lang.IllegalStateException`.

> The result of calling `copy()` on a frozen instance **is a mutable** copy, not a frozen copy.

10.6. `equals()` and `hashCode()` semantics

Golo structures honor the contract of Java objects regarding equality and hash codes.

By default, `equals()` and `hashCode()` are the ones of `java.lang.Object`. Indeed, structure members can be changed, so they cannot be used to compute stable values.

Nevertheless, structure instances returned by `frozenCopy()` have stable members, and members are being used.

Consider the following program:

```
module test

struct Point = { x, y }

function main = |args| {

  let p1 = Point(1, 2)
  let p2 = Point(1, 2)
  let p3 = p1: frozenCopy()
  let p4 = p1: frozenCopy()

  println("p1 == p2 " + (p1 == p2))
  println("p1 == p3 " + (p1 == p3))
  println("p3 == p4 " + (p3 == p4))

  println("#p1 " + p1: hashCode())
  println("#p2 " + p2: hashCode())
  println("#p3 " + p3: hashCode())
  println("#p4 " + p4: hashCode())
}
```

the console output is the following:

```
p1 == p2 false
p1 == p3 false
p3 == p4 true
#p1 1555845260
#p2 104739310
#p3 994
#p4 994
```

> It is recommended that you use `Immutable<name of struct>(...)` or `frozenCopy()` when you can, especially when storing values into collections.

10.7. Helper methods

A number of helper methods are being generated:

- `members()` returns a tuple of the member names,
- `values()` returns a tuple with the current member values,
- `isFrozen()` returns a boolean to check for frozen structure instances,
- `iterator()` provides an iterator over a structure where each element is a tuple `[member, value]`,
- `get(name)` returns the value of a member by its name,
- `set(name, value)` updates the value of a member by its name, and returns the same structure.

Chapter 11. Dynamic objects

Dynamic objects can have values and methods being added and removed dynamically at runtime. You can think of it as an enhancement over using hash maps and putting closures in them.

11.1. Creating dynamic objects

Creating a dynamic object is as simple as calling the `DynamicObject` function:

```
let foo = DynamicObject()
```

Dynamic objects have the following **reserved** methods, that is, methods that you cannot override:

- `define(name, value)` allows to define an object property, which can be either a value or a closure, and
- `get(name)` gives the value or closure for a property name, or `null` if there is none, and
- `undefine(name)` removes a property from the object, and
- `mixin(dynobj)` mixes in all the properties of the dynamic object `dynobj`, and
- `copy()` gives a copy of a dynamic object, and
- `freeze()` locks an object, and calling `define` will raise an `IllegalStateException`, and
- `isFrozen()` checks whether a dynamic object is frozen or not, and
- `properties()` gives the set of entries in the dynamic object, and
- `hasMethod(name)` checks if a method is defined or not in the dynamic object, and
- `invoker(name, type)` which is mostly used by the Golo runtime internals, and
- `fallback(handler)` defines a fallback behavior for property invocation.

11.2. Defining values

Defining values also defines getter and setter methods, as illustrated by the next example:

```
let person = DynamicObject():
  define("name", "MrBean"):
  define("email", "mrbean@gmail.com")

# prints "Mr Bean"
println(person: name())
```

```
# prints "Mr Beanz"
person: name("Mr Beanz")
println(person: name())
```

Calling a setter method for a non-existent property defines it, hence the previous
example can be rewritten as:

```
let person = DynamicObject(): name("MrBean"): email("mrbean@gmail.
```

```
# prints "Mr Bean"
println(person: name())
```

```
# prints "Mr Beanz"
person: name("Mr Beanz")
println(person: name())
```

11.3. Defining methods

Dynamic object methods are simply defined as closures. They must take the dynamic
object object as their first argument, and we suggest that you call it `this`. You can
then define as many parameters as you want.

Here is an example where we define a `toString`-style of method:

```
local function mrbean = -> DynamicObject():
  name("Mr Bean"):
  email("mrbean@gmail.com"):
  define("toString", |this| -> this: name() + " <" + this: email()

function main = |args| {

  let bean = mrbean()
  println(bean: toString())

  bean: email("mrbean@outlook.com")
  println(bean: toString())
}
```

> You cannot overload methods, that is, providing methods with the same
> name but different signatures.

> It is strongly recommended that you use `define` to create and update
> methods. Consider the following example:
>
> ```
> let obj = DynamicObject():
>   plop(|this| -> "Plop!")
> ```

Any call such as `obj: plop()` properly calls `plop()`. Because the dynamic object is fresh and new, the first call to `plop` creates a property since it is currently missing.

That being said, the following would fail:

```
obj: plop(|this| -> "Plop it up!")
```

Indeed, when the value of a dynamic object property is a function, it is understood to be a method, hence calling `plop` like it would be a setter method fails because there already exists a property that is a function, and it has a different signature. It needs to be updated as in:

```
obj: define('plop', |this| -> "Plop it up!")
```

As a rule of thumb, prefer named setters for values and `define` for methods. It is acceptable to have named definitions for methods if and only if a call happens after the object creation and before any call to `mixin` (remember that it injects properties from other objects, including methods).

11.4. Querying the properties

The `properties()` method returns a set of entries, as instances of `java.util.Map.Entry`. You can thus write code such as:

```
function dump = |obj| {
  foreach prop in obj: properties() {
    println(prop: getKey() + " -> " + prop: getValue())
  }
}
```

Because dynamic object entries mix both values and method handles, do not forget that the predefined `isClosure(obj)` function can be useful to distinguish them.

11.5. Defining a fallback behavior

The `fallback(handler)` method let's the user define a method that is invoked whenever the initial method dispatch fails. Here is an example of how to define a fallback.

> Calling a setter method for a non-existent property defines it, thus the fallback is not applicable for setters.

```
let dynob = DynamicObject():
```

```
    fallback(|this, method, args...| {
      return "Dispatch failed for method: " + method + ", with args:
    })

println(dynob: casperGetter())
println(dynob: casperMethod("foo", "bar"))

Dispatch failed for method: casperGetter, with args:
Dispatch failed for method: casperMethod, with args: foo bar
```

Chapter 12. Adapters

There is already much you can do while in Golo land using functions, closures, structs, augmentations and dynamic objects.

Yet, the JVM is a wider ecosystem and you will soon be tempted to integrate existing Java libraries into your code. Calling Java libraries from Golo is quite easy, but happens when you need to subclass classes or provide objects that implement specific interfaces?

As you can easily guess, this is all what adapters are about: they allow the definition of objects at runtime that can extend and inherit Java types.

12.1. A simple example

Let us get started with a simple example of a web application based on the nice Spark micro-framework.

Spark requires route handlers to extend an abstract base class called `spark.Route`. The following code snippet does just that:

```
module sparky

import spark
import spark.Spark

function main = |args| {
  let conf = map[        ❶
    ["extends", "spark.Route"],    ❷
    ["implements", map[      ❸
      ["handle", |this, request, response| {    ❹
        return "Golo, world!"
      }]
    ]]
  ]
  let fabric = AdapterFabric()  ❺
  let routeMaker = fabric: maker(conf)  ❻
  let route = routeMaker: newInstance("/hello")    ❼
  get(route)  ❽
}
```

❶   An adapter configuration is provided by a map object.
❷   The `extends` key allows specifying the name of the parent class (`java.lang.Object` by default).
❸   The `implements` provides a map of method implementations.
❹   The implementation is given by a closure whose signature matches the parent class definition, and where the first argument is the receiver object that is going to be the adapter instance.

❺ An adapter fabric provides context for creating adapters. It manages its own class loader.

❻ An adapter maker creates instances based on a configuration.

❼ The `newInstance()` method calls the right constructor based on the parent class constructors and provided argument types.

❽ The `spark.Spark.get()` static is method is happy as we feed it a subclass of `spark.Route`.

> Adapter objects implement the `gololang.GoloAdapter` marker interface, so you can do type checks on them a in: (`foo oftype gololang.GoloAdapter.class`).

12.2. Implementing interfaces

This is as easy as providing a `java.lang.Iterable` as part of the configuration:

```
let result = array[1, 2, 3]
let conf = map[
  ["interfaces", ["java.io.Serializable", "java.lang.Runnable"]],
  ["implements", map[
    ["run", |this| {
      for (var i = 0, i < result: length(), i = i + 1) {
        result: set(i, result: get(i) + 10)
      }
    }]
  ]]
]
let runner = AdapterFabric(): maker(conf): newInstance()
runner: run() ❶
```

❶ As you may guess, this changes the `result` array values to `[11, 12, 13]`.

12.3. Overrides

Implementations are great, but what happens if you need to call the parent class implementation of a method? In Java, you would use a `super` reference, but Golo does not provide that.

Instead, you can override methods, and have the parent class implementation given to you as a method handle parameter:

```
let conf = map[
  ["overrides", map[
    ["toString", |super, this| -> ">>> " + super(this)]
  ]]
```

```
]
println(AdapterFabric(): maker(conf): newInstance(): toString()) ❶
```

❶   This prints something like: >>> $Golo$Adapter$0@12fc7ceb.

> You can mix both implementations and overrides in an adapter configuration.

12.4. Star implementations and overrides

You can pass * as a name for implementations or overrides. In such cases, the provided closure become the dispatch targets for all methods that do not have an implementation or override. Note that providing both a star implementation and a star override is an error.

Let us see a concrete example:

```
let carbonCopy = list[] ❶
let conf = map[
  ["extends", "java.util.ArrayList"],
  ["overrides", map[
    ["*", |super, name, args| { ❷
      if name == "add" {
        if args: length() == 2 {
          carbonCopy: add(args: get(1))  ❸
        } else {
          carbonCopy: add(args: get(1), args: get(2)) ❹
        }
      }
      return super: spread(args)  ❺
    }
  ]]
]]
let list = AdapterFabric(): maker(conf): newInstance()
list: add("bar")
list: add(0, "foo")
list: add("baz")  ❻
```

❶   We create an empty list, more on that later.
❷   A star override takes 3 parameters: the parent class implementation, the method name and the arguments into an array (the element at index 0 is the receiver).
❸   We copy into carbonCopy.
❹   Same here, but we dispatch to a different method
❺   We just call the parent class implementation of whatever method it is. Note that spread allows to dispatch a closure call with an array of arguments.
❻   At this point carbonCopy contains ["foo", "bar", "baz"] (and so does list, too).

The case of star implementation is similar, except that the closure takes only 2 parameters: `|name, args|`.

12.5. Misc.

The `AdapterFabric` constructor can also take a class loader as a parameter. When none is provided, the current thread context class loader is being used as a parent for an `AdapterFabric`-internal classloader. There is also a static method `withParentClassLoader(classloader)` to obtain a fabric whose class loader is based on a provided parent.

As it is often the case for dynamic languages on the JVM, overloaded methods with the same name but different methods are painful. In such cases, we suggest that you take advantage of star-implementations or star-overrides as illustrated above on a `ArrayList` subclass where the 2 `add(obj)` and `add(index, obj)` methods are being intercepted.

Finally we do not encourage you to use adapters as part of Golo code outside of providing bridges to third-party APIs.

Chapter 13. Dynamic code evaluation

Golo provides facilities for dynamically evaluating code from strings in the form of the `gololang.EvaluationEnvironment` class. It provides an API that is useful both when used from Golo code, or when used from a polyglot JVM application that embeds Golo.

13.1. Loading a module

The code of a complete module can be evaluated by the `asModule` method:

```
let env = gololang.EvaluationEnvironment()
let code =
"""
module foo

function a = -> "a!"
function b = -> "b!"
"""
let mod = env: asModule(code)
let a = fun("a", mod)
let b = fun("b", mod)
println(a())
println(b())
```

It is important to note that an `EvaluationEnvironment` instance has a `Golo-Classloader`, and that attempting to evaluate module code with the same `module` declaration will cause an error. Indeed, a class loader cannot load classes with the same name twice.

13.2. Anonymous modules

The `anonymousModule` method is similar to `asModule`, except that the code to evaluate is free of `module` declaration:

```
let env = gololang.EvaluationEnvironment()
let code =
"""
function a = -> "a!"
function b = -> "b!"
"""
let mod = env: anonymousModule(code)
let a = fun("a", mod)
let b = fun("b", mod)
println(a())
```

```
println(b())
```

The modules that get evaluated through `anonymousModule` have unique names, hence this method is suitable in cases where the same code is to be re-evaluated several times.

13.3. Functions

The `asFunction` and `def` methods evaluate function code. Here is how `asFunction` can be used:

```
let env = gololang.EvaluationEnvironment()
let code = "return (a + b) * 2"
let f = env: asFunction(code, "a", "b")
println(f(10, 20))
```

It evaluates straight code as the body of a function. Note that `imports` can be used to specify `import` statements to be available while evaluation the code:

```
env:
  imports("java.util.LinkedList", "java.util.HashMap"):
  asFunction("""let l = LinkedList()
let m = HashMap()""")
```

The `def` method is similar, except that it has the parameters definition in the code to evaluate:

```
let env = gololang.EvaluationEnvironment()
let code = "|a, b| -> (a + b) * 2"
let f = env: def(code)
println(f(10, 20))
```

13.4. Running code

The first form of `run` method works as follows:

```
let env = gololang.EvaluationEnvironment()
let code = """println(">>> run")
foreach i in range(0, 3) {
  println("w00t")
}
return 666"""
println(env: run(code)) # => "w00t"x3 and "666"
```

The second form allows passing parameter values in a map:

```
let env = gololang.EvaluationEnvironment()
```

```
let code = """println(">>> run_map")
println(a)
println(b)
"""
let values = java.util.TreeMap(): add("a", 1): add("b", 2)
env: run(code, values)
```

It is important not to abuse `run`, as each invocation triggers the generation of a one-shot class. If the same code is to be run several times, we suggest that you take advantage of either `def` or `asFunction`.

Chapter 14. Concurrency with workers

Concurrency is **hard**. Fortunately for us the `java.util.concurrent` packages bring useful abstractions, data types and execution mechanisms to get concurrency **"a little bit better"**.

Golo doesn't provide a equivalent to the `synchronized` keyword of Java. This is on-purpose: when facing concurrency, we advise you to just use whatever is in `java.util.concurrent`.

That being said we provide a simple abstraction for concurrent executions in the form of **workers**. They pretty much resemble JavaScript web workers or **isolates** in Dart, albeit they do not really isolate the workers data space.

14.1. The big picture

A **worker** is simply a Golo function that can be executed concurrently. You can pass **messages** to a worker, and they are eventually received and handled by their target worker. In other words, workers react to messages in an asynchronous fashion.

Communications between a worker and some client code happens through **ports**. A **port** is simply an object that is responsible for dispatching a message to its worker.

Ports are obtained by **spawning** a worker function from a **worker environment**. Internally, a worker environment manages a `java.util.concurrent` executor, which means that you do not have to deal with thread management.

14.2. Worker environments

Worker environments are defined in the `gololang.concurrent.workers.WorkerEnvironment` class / module.

You can directly pass an instance of `java.util.concurrent.ExecutorService` to its constructor, or you may go through its builder object and call either of the following static methods:

- `withCachedThreadPool()` uses a cached thread pool,
- `withFixedThreadPool(size)` uses a fixed number of threads in a pool,
- `withFixedThreadPool()` uses a pool with 1 thread per processor core,
- `withSingleThreadExecutor()` uses a single executor thread.

In most scenarios `withCachedThreadPool()` is a safe choice, but as usual, your mileage varies. If you have many concurrent tasks to perform and they are not IO-bound, then `withFixedThreadPool()` is probably a better option. You should always measure, and remember that you can always pass a fine-tuned executor to the `WorkerEnvironment()` constructor.

Worker environments also provide delegate methods to their internal executor. It is important to call `shutdown()` to close the workers environment and release the threads pool. You can also call the `awaitTermination`, `is-Shutdown` and `isTerminated` methods whose semantics are exactly those of `java.util.concurrent.ExecutorService`.

14.3. Spawning a worker and passing messages

Worker functions take a single parameter which is the message to be received. To obtain a port, you need to call the `spawn(target)` function of a worker environment, as in:

```
let env = WorkerEnvironment.builder(): withFixedThreadPool()
let port = env: spawn(|message| -> println(">>> " + message))
```

A port provides a `send(message)` method:

```
port: send("hello"): send("world")
```

Messages are being put in a queue, and eventually dispatched to the function that we spawned.

14.4. A complete and useless example

To better understand how workers can be used, here is a (fairly useless) example:

```
module SampleWithWorkers

import java.lang.Thread
import java.util.concurrent
import gololang.concurrent.workers.WorkerEnvironment

local function pusher = |queue, message| -> queue: offer(message)

local function generator = |port, message| {      ❷
  foreach i in range(0, 100) {
    port: send(message)                            ❸
  }
}

function main = |args| {

  let env = WorkerEnvironment.builder(): withFixedThreadPool()
  let queue = ConcurrentLinkedQueue()

  let pusherPort = env: spawn(^pusher: bindTo(queue))
```

```
let generatorPort = env: spawn(^generator: bindTo(pusherPort))

let finishPort = env: spawn(|any| -> env: shutdown()) ❹

foreach i in range(0, 10) {
  generatorPort: send("[" + i + "]")
}
Thread.sleep(2000_L)
finishPort: send("Die!") ❺

env: awaitTermination(2000)
println(queue: reduce("", |acc, next| -> acc + " " + next))
}
```

In this example, we spawn 3 workers:

❷   the first repeats a message 100 times,
❸   …forwarding them to another one,
❶   …that ultimately pushes them to a concurrent queue.
❺   A message is sent to a final worker,
❹   …that shuts the workers environment down.

As an aside, the example illustrates that worker functions may take further dependencies as arguments. The `pusher` function takes a queue target and `generator` needs a port.

You can satisfy dependencies by pre-binding function arguments, all you need is to make sure that each function passed to `spawn` only expects a single message as its argument, as in:

- `^pusher: bindTo(queue)`, and
- `^generator: bindTo(pusherPort)`, and
- `env: spawn(|any| -> env: shutdown())` where the worker function is defined as a closure, and implicitly captures its `env` dependency from the surrounding context.

Chapter 15. Golo template engine

Golo comes with a built-in template engine that is reminiscent of Java Server Pages or Ruby ERB. It compiles template text into Golo functions.

15.1. Example

Consider the following example.

```
let template = """
<%@params posts %>
<!DOCTYPE html>
<html>
  <head>
    <title>Golo Chat</title>
  </head>
  <body>
  <form action="/" method="post">
    <input type="text" name="msg">
    <input type="submit" value="Send">
  </form>
  <div>
    <h3>Last posts</h3>
    <% foreach post in posts { %>
      <div>
        <%= post %>
      </div>
    <% } %>
  </div>
  </body>
</html>
"""
```

This multi-line string has a Golo template. It can be compiled into a function as follows:

```
let tpl = gololang.TemplateEngine(): compile(template)
println(tpl(someDataModel: posts()))
```

15.2. Directives

As you may have guess from the previous example:

- Golo code snippets are placed in `<%  %>` blocks, and
- expressions can output values using `<%=  %>`, and

- `<%@import foo.bar.Baz %>` causes `foo.bar.Baz` to be imported, and
- `<%@params foo, bar, baz %>` causes the template function to have 3 parameters, i.e., it is a `|foo, bar, baz| { ... }` function.

When no `<%@params ... %>` exists, the function is assumed to have a single `params` parameter.

> The template engine is a simple one and makes **no** verification either on the templates or the resulting Golo source code. The `compile` method may throw a `GoloCompilation` exception though, and you can query the exception `getSourceCode()` and `getProblems()` methods to obtain more details.

Chapter 16. Documenting Golo code

Of course you can document your code using comments (#), but who reads source code?

16.1. Documentation blocks

Golo provides a support for documentation blocks on modules, functions, augmentations and structs. Blocks are delimited by `----` and contain free-form Markdown text.

Here is a quick example:

```
----
This is a *nice* module that does a bunch of useless things.

See more at [our website](http://www.typeunsafe.org).
----
module Hello


----
Adds 2 elements, which is quite surprising given the name.

* `x` is the first argument,
* `y` is the second argument.

The following snipped prints `3`:

    let result = adder(1, 2)
    println(result)

Impressive!
----
function adder = |x, y| -> x + y
```

16.2. Rendering documentation

The `golo doc` command can render documentation in `html` (the default) or `markdown` format:

```
$ golo doc --output target/documentation src/**/*.golo
```

Please consult `golo help` for more details.

16.3. Alignment

It is sometimes necessary to indent documentation blocks to match the surrounding code format. Documentation blocks erase indentation based on the indentation level of the opening block:

```
----
The most useful augmentation *ever*.
----
augment java.lang.String {

  ----
  Creates a URL from a string, as in: `let url = "http://foo.bar/p
  ----
  function toURL = |this| -> java.net.URL(this)
}
```

When generating documentation from the code above, the documentation block of the `toURL` function is unindented of 2 spaces.

Chapter 17. Common pitfalls

Discovering a new programming language is fun. Yet, we all make mistakes in the beginning, as we idiomatically repeat habits from other languages.

Because Golo works closely with the Java programming language, it is likely that Java programmers will make some of the following mistakes early on.

17.1. `new`

Golo does not have a `new` operator for allocating objects. Instead, one should just call a constructor as a function:

```
# Good
let foo = java.util.LinkedList()

# Compilation fails
let foo = new java.util.LinkedList()
```

17.2. Imports

Golo does not have **star** imports like in Java. Imports are only used at runtime as Golo tries to resolve names of types, functions, and so on.

You must think of `import` statements as a notational shortcut, nothing else. Golo tries to resolve a name as-is, then tries to complete with every import until a match is found.

```
import java.util
import java.util.concurrent.AtomicInteger

# (...)

# Direct resolution at runtime
let foo = java.util.LinkedList()

# Resolution with the 1st import
let foo = LinkedList()

# Resolution with the 2nd import
let foo = AtomicInteger(666)
```

17.3. Method invocations

Keep in mind that instance methods are invoked using the ∶ operator, not with dots (.) like in many languages.

This is a common mistake!

```
# Calls toString() on foo
foo: toString()

# Looks for a function toString() in module foo
foo.toString()
```

17.4. `match` is not a closure

One thing to keep in mind is that `match` returns a value, and that it is not a closure unless you want it to.

```
let foo = match {
  case plop then 1
  case ploped then 2
  otherwise -1
}

# Ok
println(foo)

# Bad! foo is an integer!
println(foo("abc"))
```