lodge_spec

00.00 Preface

Specification Version: 0.0.1
Language Version: 0.0.1

Author: Dylan Carroll

Versioning

The versions of the specification and the language consist of three numbers. For the spec version and the language version, the first two numbers of each represent major and minor versions.

Major versions indicate major revisions to the language and large breaking changes.

Minor versions indicate minor revisions and compatibility-preserving changes.

The third digit of the specification represent sub-minor versions of the specification. These indicate changes to the specification that don't constitute an update to the language. Changes for clarity or organization would be considered sub-minor spec versions.

Likewise, the third digit of the language are sub-minor versions of the language. Sub-minor versions of the language do not reflect changes in the specification and thus do not correspond to new features or changes in behavior. Changes relating to fine implementation details, refactoring, and performance improvements may constitute sub-minor language versions.

Contents

01 Specification

This section describes how the lodge programming is intended to be used and behave

02 Implementation

This is a description of how Lodge is implemented. It is intended to function as documentation for the official implementation of lodge.

03 Style Guide

01.01.00 Basics

01.01.01 Statements

Statements

As lodge has no statement terminator, the end of statements is automatically inferred by newline characters in most cases. However, there are cases where multiple lines can be considered a single statement. This means that Lodge is not totally whitespace agnostic like other languages. It is still agnostic towards tab and space characters.

In Lodge, the statement termination happens at newlines by default, but statements may be joined together based on some rules.

1. If a line opens a bracket (such as: `( [ <`), all lines until that bracket is closed will be treated as a single statement. However, the closing bracket will only be searched for until the end of the current block.
2. If a line ends with an operator that requires another following operand, the following line will be treated as part of the same statement.
3. If a line begins with an operator that requires an operand beforehand, the previous line will be treated as part of the same statement.

```
!! Examples of rule 1
Int x := f(arg1,
               arg2,
               arg3)

List l := <val1, val2,
               val3, val4>

!! Example of rule 2
Int y := 10 +
         11 +
             12

!! Example of rule 3
Int z := 10
       + 11
       + 12
```

01.01.02 Comments

Comments

Comments are created with the `!!` token. Everything after the `!!` until the next newline will be considered a comment and completely ignored.

Bounded comments can be created with `!-` and `-!`. Everything between these two tokens will be ignored as comments.

01.01.03 Built-In Types

Integers

value type

size	signed	Unsigned
8	`i8`	`u8`
16	`i16`	`u16`
32	`i32`	`u32`
64	`i64`	`u64`

Floats

value type

size	name	
32	`f32`	
64	`f64`	

Booleans

value type

Size	name
8	bool

Strings

reference type

Str

- A reference type object much like strings in other languages
- Basically just an immutable array of chars
- Lodge Text Encoding
- String literals are created using text surrounded by double quotes like this:

```
Str aString := "A new string literal"
```

None

value type

Lodge None type can only have a single value: None
The None type has a totally empty interface.

Uses

- Indicate that something does not exist in cases where using null doesn't make sense.
- When a variable has a None in its union, the interface is empty and you would need to check that the value is not None in a type switch before you could do any operations with it.
- If that variable is always initialized to None, it could eliminate the chance of null pointer exceptions by forcing the None value check.
- Used as a default value for function arguments when that argument doesn't always need to be present. This is useful for variatic functions and optional parameters.

Data Structure Types

All data structure types are reference types

Tuple

- literal: `(value1, value2, value3)`
  - `(value,)` for a single value
- type definition: `(type1, type2, type3)`
  - `(type : size)` when it's all the same type
- immutable value types
- You must declare the type for each variable in the tuple

```
$(Int, Str) tuple1 := (10, "hello")
(Int,) tuple2 := (10,)
(Int: 5,) tuple3 := (1, 2, 3, 4, 5)
(Int: 2, Str) tuple3 := (1, 2, "a", b")
```

Array

- literal: `[value1, value2, value3]`
- type definition: `[type : size]`
  - note: arrays can only have one type (or union)
- mutable reference types
- Cannot grow or shrink
- The type (union) must be the same for each element in the array

```
[int] arr1 := [0, 1, 2, 3, 4]
var arr2 := [0, 1, 2, 3, 4]
```

List

- literal: `<value1, value2, value3>`
- type definition: `<type>`
- mutable reference type

- can grow and shrink

```
<int> list := <0, 1, 2, 3, 4>
var list2 := <5, 6, 7, 8, 9>
```

Dictionary

- literal: `{ key1 : value1, key2 : value2}`
- type definition: `{type1 : type2}`
- Mutable reference type
- Implemented as a hash table

```
{str : int} aDictionaty := {"A" : 1, "B" : 20, }
```

Set

- literal: `{value1, value2, value3}`
- type definition: `{type}`
- Implemented as a hash set

```
{int} set1 = {2, 4, 6, 8}
```

Value Types vs Reference Types

All values in rust are represented in memory by a thing.
The thing contains a field for the value and a field for type information.

For value types, the value field is the value itself, but for reference types the value is a reference to the object in memory.

For smaller value types like `i8`s, there are wasted bytes in the high end of the value portion of the thing. To combat this, some types such as arrays may be implemented in such a way as to store raw values rather than things when the type is restricted to that of a value type.

01.01.04 Operators

Mathematical Operators

op	action
+	addition
-	subtraction (and unary negation)
/	division

op	action
//	integer division
*	multiplication
%	modulus
**	power

Logical Operators

The operators here that are words are reserved keywords

op	action
and	Logical and
or	Logical or
not	unary logical negation operator
xor	exclusive or
==	value equivalence
is	identity equivalence
≠	not value equivalent
isnt	not identity equivalent
>	greater than
<	less than
≥	greater than or equal to
≤	less than or equal to

The value equivalence operator `==` should not be confused with the assignment operator `:=`, or the function keyword token `=`.

Bitwise Operators

op	action
&	bitwise and
|	bitwise or
^	bitwise xor
~	bitwise not (compliment)
>>	right shift
<<	left shift

Assignment Operators

op	action
:=	assignment
++	increment
--	decrement

There are also forms such as `+=`, `-=`, `*=`, `/=`, and `**=` that are shorthand for performing an operation and then assigning the result to the first operand (assuming that operand is an existing identifier). Any operator except assignment operators and unary operators followed by a = will be treated as being a part of this shorthand.

Other Operators

op	action
:	type conversion
()	call as a function
[]	indexing and slicing
->	Virtual field
!	Error Resolution
?	None Resolution Operator
#	hash
$	other
@	other

Type Conversion Operator

The type conversion operator `:` is the way to convert an object of one type into another.

It is particularly useful for conversion into strings for representational purposes:

```
Str myStrRepr := myObj:str
```

Or for conversions between numerical types

```
Int myInt := myFloat:int
```

Error Resolution Operator

Shorthand for resolving errors

Errable functions are functions that have the potential to return an error, meaning that the return type is the union of some type and `Err`. In order to use the return value of an errable function, you need to handle the possibility of an error.

Often times, handling an error just consists of passing it further up the call stack for the caller do handle. For this, the unary `!` operator is used. It will evaluate to a non-Err value or return Err from the function.

Here is a common pattern:

```
fun function() !Int {
        Int val := !errableFunction()

        !! Operations on val as a Str
}
```

Which is equivalent to:

```
fun function() (Err | Int) {
        (Err | Int) val := errableFunction()
        swype val {
                Err : { return Err }
                *   : { return val }
        }
}
```

Binary Error Resolution Operator

Sometimes, instead of passing up the error, it is better simply to have a default value in case of an error. For these cases, the error resolution operator can also be used as a binary operator.

```
fun function() !Err {
        Int val := errableFunction() ! -1
}
```

Is equivalent to

```
fun function() !Int {
        !Int result = errableFunction()
        Int val := swype result {
                Err : { -1 }
                Str : { result }
        }
}
```

In Type Definitions

The `!` token is also used to make an Errable type:

```
fun errableFunction() (Err | Str) {
        !! ...
}
```

```
fun errableFunction() !Str {
        !! ...
}
```

None Resolution Operator

The None resolution operator applies to variables whose type is a union that contains None. It is essentially a shorthand for an otherwise common type switch.

The following two code blocks are equivalent:

```
?Int temp = nonableFunction()

Int variable = swype variable {
        None : {-1}
        Int 2 : { temp }
}
```

```
Int variable = nonableFunction() ? -1
```

Operator Precedence

```
!! Assignment

:=  +=  -=  *=  /=  //=  ...

!! Logical

or
xor
and
not (unary)

==  !=  is  isnt



<  >  >= <=



!! Numerical
```

```
>>  <<



|
^
&



+ -



*  /  //   %



-  ~ (prefix)

**



!! Pre and post increment operators
++ --





!! Special Infix operators
!  ?



!! Special Prefix Operators
!

!! Special Postfix Operators
()  []   ->  :



.
```

01.01.05 Variables

Declaring and Assigning Variables

Lodge variable declarations work much like they do in C or Java. A variable declaration consists of a type followed by a variable name.

Assignment occurs using the `:=` operator, which often

```
Int value    !! Decalring a variable without defining it
Int value2 := 10    !! Declaring an int variable and assigning it a value
```

Assignment also does not need to be a standalone statement, but can be an expression that returns the new value, like in C.

```
!! Print out the characters from a file until the character 'a' is reached
Str c
loop while (c := file.getChar()) != 'a'
        { print(c) }
```

Lodge is garbage-collected, so created values do not need to be manually freed.

Var Keyword

When variables are declared with the `var` keyword instead of a type,

```
var value4 := "hello"    !! Declaring a variable as a string with an inferred type
var arr2 := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]    !! Declaring a type with an inferred type
variable
```

Variable Types

In Lodge, the type of a variable does not restrict the values that can be stored to just the type of the variable. Instead, any value with a compatible interface can be stored in that variable. For example, a variable with the default `Int` type can store any integer value.

```
!! Allowed
x := i8(-10)
x := u64(100)

!! Not Allowed
x := "10"
x := 10.5
```

The full behaviors of Lodge's type checking system is found in Chapter 04: Typing.

Union Types

Instead of creating a variable with just a single type, it is also possible in Lodge to define a variable with a composite type called a type union. This allows a variable to store any value that has a compatible interface with   type in the union.

```
(Str | Int) x

!! Allowed
x := 10
x := "Hello World"

x := i8(-10)
```

Composite types can also be given names with the `as` keyword

```
(Str | Int) as Strint
Strint x = "10"
```

Auto Keyword

Since variables in lodge are defined entirely based on interface compatibility than particular concrete types, it is possible to automatically declare a variable whose type interface is defined by what members are ac

The auto keyword will look at the entire scope that it's variable is being bound in to see what members of that variable's interface are being accessed. That then becomes the interface of that variable.

Take this example:

```
fun functionName(auto arg) {
        arg.name = "Name"
        Float result = arg.takeAction(10)
}
```

For this function the `arg` variable will have the following interface:

```
Interface {
        Str name
        fun takeAction(Int) Float
}
```

Literals

Numerical Literals

- numerical literals
  - `10`, `1.5`, `.5`, `0xFF`, `0b1010`, etc.
- string literals
  - "Text inside quotations"
- operators

    - `+`, `-`, `*`, `/`, etc.
- keywords
    - Like `if`, `loop`, `class`, `fun`, etc.
- identifiers
    - Like the names of variables, types, classes, etc.

Circular Type Definitions

There may be cases where a composite type declaration may want to refer to itself, which is allowed in Lodge.

Here's an example of how a tuple containing two values can be used to create something akin to a linked list

```
(int, listelement)|None as listElement

listElement list = (0 (1, (2, (3, None))))
```

01.01.06 Control Flow

Blocks

Blocks are opened and closed by curly brackets `{}`. Many statements such as if statements, loops, and class definitions require blocks.

Blocks in lodge create a new scope that can access variables from the outer scope, but whose variables only last during the scope of.

```
Int x := 0
loop while x <= 10 {
        if x%2 == 0
                { print(x) }
        x++
}
```

Naked blocks can also be created without an opening statement simply to create a new scope and group together a sequence of statements.

Block Expressions

If a block is placed in a location where it is evaluated as an expression rather than a standalone statement, the last statement of a block is treated as an expression that the entire block evaluates to.

For naked blocks, this can simply be used as a way to cluster a sequence of statements into a single unit that returns a value. It has essentially no impact on the way the code executes, but may be used for conceptual groupings of actions.

When an if statement is used as an expression, evaluating blocks can be used for constructs like ternaries.

When a loop is used in an assignment, each iteration of the loop is evaluated separately, and placed into a list that returns when the loop completes.

Conditionals

Conditionals use the `if` keyword, a boolean expression, and a block to create an if-statement. If the given boolean expression evaluates to true, the block is entered. Otherwise, it is skipped.

If

```
if boolean_expression {
        !! conditional body
}
```

If / Else

You can also specify as block to enter if the boolean is false using the `else` keyword.

```
if boolean_expression {
        !! Enter if true
} else {
        !! Enter if false
}
```

If / Else If / Else

And you can also chain if statements using `else if`

```
if exp1 {
        !! exp1 is true
}
else if exp2{
        !! exp1 is false and exp2 is true
}
else{
        !! exp1 is false and exp2 is false
}
```

If Expressions

You can use if statements like an expression and the selected block will be evaluated

```
Int x := 10
Int result := if (x < 0) {
        -1
} else if (x == 0) {
        0
} else {
        1
        }

!! result = 1
```

Loops

Lodge has four kinds of loops:

- plain loop
- while loops
- for loops
- over loops

Plain Loop

A plain loop is a loop without any parameters. It is essentially the same as a `loop while true`, but it does not perform a comparison. It is usually used for replicating a do-while type pattern using a `break if` statement.
Here is an example.

```
!! Read one charafter from the file onto the stack until a period is reached
loop {
        c := file.read()
        stack.push(c)

        break if c == '.'
}
```

While Loop

A while loop takes a boolean expression and will repeat its body code until the expression evaluates to false.

```
!! Print all the values on the stack
loop while not stack.is_empty() {
        ints val := stack.pop()
        print(val)
}
```

For Loop

This kind of loop is merely for looping over ranges of integers. Much like similar languages (besides python) this is essentially just syntactic sugar for a declaration, a while loop, and an increment. It uses the keywords `from`, `to`, and `by` to control this range. All of these except to are optional. If `from` is left out, the default is 0. If `by` is left out, the default is 1.

The type of the loop variable does not need to specified and will default to Int with a concrete type of i64.

If the variable exists already, its type will remain the same and its value will be assigned to the loop values.

Here is an example. The top of the range is exclusive.

```
!! Print all numbers less than 100 divisible by 3.
loop for x from 0 to 100 by 3
        print(x)
```

Over Loop

The over-loop works like Python's for-loop or for-each loops in other languages. It accepts any object that is compatible with the iterator interface and loops. It uses the `from` keyword to associate a name with the element from the iterator.

```
!! Print all of the characters in the list
loop over c from char_list
        print(c)
```

If you want to loop over multiple iterators of the same length at once, you can separate several associations with a comma as below.

```
loop over a, b from list1, list2
        list3.append(a + b)
```

You can also add an option that will keep track of the index with the `at` keyword.

```
!! Print all of the characters in the list
loop over c from char_list at i
        print(c)
```

Break and Continue

The keywords `break` and `continue` work as they do in similar languages.
`break` will exit the nearest loop that encapsulates it.
`continue` will skip to the beginning of the nearest loop that encapsulates it

Lodge also has the `break if` and `continue if` statements which act as conditional statements

For example, these two code blocks behave the same:

```
loop over value from someList {
        print(value)

        break if value == null
}

loop over value from someList {
        print(value)

        if value == null {
                break
        }
}
```

01.01.07 Functions

Declaring functions

Function declarations in lodge use the `fun` keyword. When declaring a function, the name, arguments, return type, and function body must be specified.

Synyax

Defining a function with a name

```
fun functionName(type1 arg1, type2 arg2, ... ) returnType {
        !! Function body
}
```

Defining an anonymous function

```
fun (args) returnType {
        !! Function body
}

!! Arrow notation for brevity
(args) => { !- Function body -! }
```

Declaring a variable with the type of a function.

```
fun (type1, type2, type3, ...) returnType variableName := !! ...

!! Howvever, it  is less verbose to use var
```

```
var variableName := fun (args) returnType { !-Function body-! }
```

Optional Arguments / Default Values

Normally, a function expects to be provided with all of its arguments whenever it is called. However, it is possible to give a default value for an argument that it will take when not provided.

- You can give a function optional arguments by giving those arguments default values
- When calling the function, those arguments don't need to be given
    - If you want to specify only an argument that comes after an optional argument, you must give that argument with the `arg = val` syntax.

```
fun f(int a = 10, ?int b = None) {
        !! Function body
}

f() !! case 1
f(1) !! case 2
f(b=2) !! case 3
f(1, 2) !! case 4
```

Providing Arguments by Keyword

In addition to passing positional arguments by their position, it is also possible to specify the name.

All non-optional arguments must be provided exactly once, whether that be positionally or by keyword. Failing to do so will result in a compiler error.

```
fun functionName(int x, int y=2, ?int z=None){
        !! Function body
}

!! Legal
functionName(1, 2, 3)
functionName(x=1, y=2, z=3)
functionName(1, z=3)

!! Not legal
functionName(y=2, z=3) !! x not provided
```

Capturing Extra Arguments

Capturing Extra Positional Arguments

- Define a function with variadic arguments by placing `...` after the last argument. That name will then be treated like an array of those values within the function body. Other arguments. The type of this value will be an array of values with the specified type.

```
fun f(int x, Int vargs...) returnType {}
        print(length(args))
        print(args[x])
}

f(1, 2, 3, 4, 5) !! prints 4, 3
f(4, 0, 0, 0, 0, 100, 0) !! prints 6, 100
```

Capturing Extra Keyword Arguments

It is also possible to give keyword arguments to a function that doesn't have those names as positional arguments as long as that function has somewhere to capture those keywords. This is done using the ** token.

```
fun functionName(**kwargs){
        print(kwarg["keyword"])
}

functionName(keyword="Hello World") !!Prints "Hello World"
```

Positional-Only Arguments

There are some cases in which it may be desirable to have positional arguments that cannot be provided through keywords or through dictionary expansion.

```
fun func(Int x | Int y, Int **kwargs){
        !! ...
}

func(1, 2)
func(1, y=2)
func(x=1, y=2) !! Compile error, x not provided
func(1, 2, x=3) !! Allowed, "x" will be a key in the kwargs dict


var dict =  Dict{"x":1, "y":2}

func(1, **dict) !! x=1, y=2, and "x" is a key in the kwargs dict
func(**dict) !! Compile error x not provided
```

Function/Method Overloading

Lodge doesn't have function or method overloading explicitly as type information could not be used to unambiguously select between functions based on their signature.

The same techniques used for optional parameters can be used to mimic the behavior of function and method overloading. Essentially, the behavior a function can be configured to depend on the types of the given arguments and which arguments are given.

Argument Expansion

List Expansion

You can pass sequences of values (lists, arrays, and tuples) into the arguments of a function with the `...` suffix. This only works if the types of the sequence match with the types of the corresponding arguments of the function. This also works for variadic functions. And the passing can be a subset of the arguments

```
fun func1(int a, int b, int c) {}
        !! ...
}

fun func2(int a, str b, int c) {
        !! ...
}

!! Expanding arrays
[int] arr1 := [1, 2, 3]
func1(*arr)
func2(*arr) !! type error because the second argument type does not match

!! Expanding a tuple with heterogenous types
(int, str, int) tuple := (1, "2", 3)
func2(*tuple)

!! Passing a subset
[int] arr2 := [1, 2]
func1(arr2..., 3)
```

Dictionary Expansion

It is also possible to do a similar action with keyword arguments using the `...` suffix. With this, it is possible to supply the same argument multiple times. When that is the case, the latest provided value will be the final value of the argument.

```
fun func1(Int a, Int b, int c){
        !! ...
}

fun func1(Int a, Int **kwargs){
        !! ...
}

{Str:Int} dict = {"a" : 1, "b": 2, "c":3}

func1(dict...) !! a=1, b=2, c=3

func1(dict...) !! a=1, b=2, c=3
```

Function argument Syntax

Bringing each piece together, functions are made up of positional arguments and keyword arguments. There are rules for how the function can be specified.

Generator Functions

generator function implementation

There is no special syntax for generator functions besides the yield keyword. Any function with the yield keyword in the function body can be a generator function.

When the yield keyword is encountered, that function instead returns a generator alongside the value being. This generator is a new function closure that represents the state of the function

When defining a generator function, the return type is expressed as the following generic: ```

```
Generator<type>
```

Here's an example of a generator

```
fun FibonacciGenerator() Generator<Int>{
        a, b := (0, 1)
        loop {
                a, b := (b, a+b)
                yield b
        }
}


Generator<int> gen = FibonacciGenerator()

print(gen.next()) !! 1
print(gen.next()) !! 1
print(gen.next()) !! 2
print(gen.next()) !! 3
print(gen.next()) !! 5
```

01.01.08 Scoping Rules

01.02.00 Classes and Interfaces

01.02.01 Classes

Class Definition Syntax

Basic Syntax

```
class ClassName  {
        promises Interface1, Class1
        uses ParentClass1, !ParentClass2

        Int field

        fun Method() ReturnType {
                !! Method Body
        }
}
```

Anonymous Classes

Classes can also be created without names in order to allow the creation of an object without needing to write a dedicated named class for it.

The anonymous class will participate in the typing system as normal, and can even be placed in other variables using the `var` keyword.

```
var anonObj := class {
        !! Class body
}()
```

Even though the class does not have a defined name, it will still have an internal name and interface like any other class.

Promising

When a class promises an interface or a class, all public fields and methods are required to be explicitly defined in the promising class.

If a promise results in a name collision with previous promises, this will be accompanied by an error unless the function signatures are identical / the field types are identical.

However, because of automatic interfacing, promising only makes a difference from the perspective the programmer and doesn't result in code that compiles any differently. The advantage to promising is that it becomes impossible to fail to implement ant of the promised methods or fields.

Even if a class/interface is not promised, it will be able to participate in polymorphisms via automatic interfacing as long as it matches the criteria to do so.

Promising simply ensures that the programmer is required to satisfy those criteria as well as providing more self-documenting code.

```
class ClassName  {
        promises Interface1, Class1

        !! Class Body
}
```

Using

When a class uses another class, all methods and fields (external and internal) from the superclass are implicitly copied into the subclass. Methods created in a class via using satisfy promise requirements.

Lodge allows for using multiple classes. In the case of namespace collisions, the value from the class listed second will be used. However, if a usage overwrites the implementation of the class or any previous uses, this will cause a compiler warning. This warning can be suppressed by placing a `!` before the class name that caused the collision as seen in the example above.

```
class ClassName  {
        promises Interface1, Class1
        uses ParentClass1, !ParentClass2

        !! Class Body
}
```

Public/Private Methods and Fields

```
class A {
        int _a := 0 !! Private/Internal field
        int a := 10 !! Public/External field

        fun F(int arg) int {
                !! External function
        }

        fun _F(int) int {
                !! Internal functions
        }

}
```

The difference between external and internal class members is that internal members will not be considered as part of the class's interface for the purposes of promising, automatic interfacing, and type union definition.

Internal members can be accessed from inside the class, but also from within concrete type switches that switch on the specific concrete class.

Properties

Defining properties is done using the get and set keywords

```
class A {
	get propertyName type {
		!! Code for getting the value
	}
	set propertyName type {
		!! Code for setting the value
	}
}
```

The get and set type for a given property do not have to be the same. In fact, the get and set properties act pretty much independently from eachother despite the fact that they have thee same name.

Behind the scenes, properties are compiled as functions. Properties cannot be internal, but they can still be accessed from within the class.

From the outside, properties act just like fields.

```
A a := new A()
print(a.propertyName)
a.properyName := 10
a.propertyName += 10
```

Generics

```
class A<T> {
	T field := ....
}

A<int> a := new A<int>()
int val := a.field
```

When the `<T>` is specified, all instances of the type name `T` found in the class definition will be replaced with whatever the type is at the object creation. `T` could be any identifier, but a single uppercase letter is standard.

This is implemented by simply using the broadest possible type union to replace T. If nothing is specified, T is equivalent to the 03 Built-In Interfaces.

The type of T is known at compile time for a given object, so the return types of methods can be checked statically and type errors are avoided.

You can also specify restrictions on the generic types with interfaces or type unions.

```
class A<Iterable T> {
        ...
}

class B<(int | str | list) T> {
        ...
}
```

Generics vs Unions

There is a small, yet important difference between creating non-generic class whose member types are a type union vs creating a generic class whose generic type is that same type union.
The difference arises from the binding time of the type.
With the generic, the single allowed type is specified at object creation, whereas the type union allows all the types in the union forever. This difference is demonstrated by the following example.

```
class A {
        (Int|Str) field
}

class B<(Int|Str) T> {
        T field
}

A a := new A()
B<int> bInt:= new B<Int>()
B<string> bStr := new B<Str>()

a.field := 10
a.field := "string"

b1.field := 10
b1.field := "string"    !!Compile error

b2.field := 10          !!Compile error
b2.field := "string"
```

In class A, both strings and integers can be placed into the field regardless, but with class B, only the generic type specified at the creation of the object can be placed in the field.

Nested Classes

Classes can be defined within other classes. This class can be accessed within the outer class and from the program at large through the outer class.

This is a purely static concept. Class instances (objects) can't and don't contain classes

```
class A {
        fun F() B {
                b := new B()
                return b
        }

        class B {
                int v := 10
        }
}

A a := new A()
A.B b := a.F()
```

Operator Methods

```
class A {
        fun + (auto other) int {
                !! Logic for adding objects
        }
}

A var1 := new A()
A var2 := new A()
A var3 := var1 + var2
```

The syntax of the format `fun + () ...` is only legal within classes.

Type switching inside operator methods

Because of the way that interface intersection works in lodge, it is best to use the broadest possible type for the operator method. These methods are a particularly good use case of the auto keyword as the type of the argument grows automatically as the ways that the variable is used grows.

Reverse operator methods

When an operator is acting on two objects, the compiler will first look if the first object implements the operator on the type of the second object. If not, it will look to see if the second object implements the   of that operator.

For example, take the expression `a / b` for objects of classes `A` and `B`.

If `A` implements `/` for the type of `B`, then that method will be called on `a` with `b` as an argument.

If `A` does   implement `/` for the type `B`, then the compiler will check for the `~/` method in `B` on the type of `A`.

The intention being that even if `A` was implemented with no knowledge of `B`, `B` can be implemented such that it is still compatible with `A`, by implementing the reciprocal of the operator.

Method Overloading

- Lodge does not have method overloading in the typical sense currently
  - Recent improvements in how method signatures are defined has made me rethink this
- Type unions make it less clear what a functions signature actually is as a functions argument types could refer to any number of combination of concrete argument types.
  - It might be possible to allow overloading but restrict it to strictly non-intersecting interfaces, which would allow method overloading in some cases
  - Though, this might get frustrating as this intersection behavior would likely be complex and unintuitive
  - Though, this also makes candidacy for interface intersection much more complicated and may not be worth it.
- Instead, the way Lodge is currently designed is to use optional parameters and type switching to achieve similar behavior.
  - This is similar to Python's behavior and people don't seem to mind that much
- Look at 07 Functions to see about optional parameters and type switching to mimic the behavior of method overloading

01.02.02 Interfaces

Interfaces

An Interfaces is a set of fields, properties, and methods. All classes have an interface, which is made up of the public members (methods and fields) of that class. Interfaces exist separate from the implementation behind those members.

An interface in Lodge is much more than it is in other programming languages and it is used throughout Lodge even when not explicitly defined by the programmer.

A lodge program consists of many concrete types and many interfaces. Interfaces may be created manually, implicitly by defining a class, or implicitly when creating a type union.

When a variable or container is declared, it automatically has some interface. Any concrete type that has a compatible interface can be placed in that container. See Type Checking for more detail.

Interface Creation

There are several ways that interfaces are created in Lodge

1. Explicitly with an interface definition
    - Using the interface definition syntax
2. Implicitly by creating a class
    - Every class definition will also create an interface of the same name out of that class's public members.
3. Using a type union to combine existing interfaces through interface intersection
4. Using a type intersection to combine existing interfaces through interface unions

Behind the scenes, the members that can make up an interface are getters, setters, and methods. This is distinct from how a programmer defines an interface or class, but the separation between getters and setters fields is important for interface intersection, type unions, and properties.

When a class or interface definition specifies a public field, both a getter and setter will be created of that type in the interface. When either a getter or a setter is specified, only that getter or setter is created in the interface. Getters and setters of a given name can have the same or different types. That means that, in an interface definition, getters and setters of the same type will have the same result as a field definition. Fields and properties are identical for both external access and interface satisfaction (assuming the getter and setters have the same type).

Interface Definition Syntax

The syntax for defining an interface is much like the class definition syntax, but without any values, method/property implementations, or private members.

Here is an example:

```
interface InterfaceName {
        int fieldName

        fun FunctionName(int arg1, str arg2) int

        get propertyName int
        set propertyName int
}
```

01.02.03 Built-In Interfaces

The broadest possible interface. Every single object in lodge can be placed inside of a variable defined as object.

Type Interfaces

Every concrete type in Lodge automatically has an associated interface.

Int interface

Float Interface

Err Interface

```
interface Err {
        get Str Type
        get Str Message

        fun $() T

        !! ...
}
```

Object Interface

```
interface Object {
        fun repr() Str

        !! ...
}
```

None Interface

The None interface is completely empty. Its used in situations where a value might not exist.

```
interface None {}
```

01.02.04 Interface Compatibility

Interface compatibility describes candidacy for polymorphism in Lodge.

The concept of a compatible interface refers to the ability for an interface to be substituted for another without issue. In order for an interface to be compatible with another, that interface must be a superset of the other interface. If we have two interfaces A and B, B is compatible with A, if there are no methods or fields in A that are missing from B. In other words, if B would satisfy promising A, then B is compatible with A.

For example, the Int interface is not compatible with the Str interface because it fails to implement several of the methods that the string class like length and lowercase. In other words,

01.02.05 Automatic Interfacing

Automatic Interfacing is the process by which a class that doesn't explicitly promise an interface can still be used with that interface if it happens to implement. If, at compile time, a class's public interface is a superset of an interface or the public interface of another class, interface, type union, or type intersection, that class will implicitly promise that class/interface.

The result of automatic interfacing is that, objects can be placed into any container whose interface is a subset of that objects interface without needing to be explicitly declared. This is conveniently similar to duck typing while maintaining static type safety.

See 04 Interface Compatibility\

01.03.00 Typing

This section discusses Lodge's typing system.

01.03.01 Type Checking

Lodge is a type type-safe language, meaning that all type checks occur statically at compile time and that no operation can cause a type error at runtime. However, unlike other statically typed languages, Lodge's type system allows more type flexibility while still ensuring type safety.

This means that the Lodge type checker is somewhat complex to match the complexity of the type system.

01.03.02 Type Set Operations

Type Unions and Type Intersections

Type unions and type intersections are two ways of creating composite types out of multiple other types.

Type unions are the most common and allow the programmer to create a flexible variable that hold multiple different concrete types, while still being fully type checked.

Type intersections allow the programmer to create a variable that can only contain values that satisfy multiple interfaces. This use case is less common, but allows variables to be entirely

This behavior is represented in the syntax of type unions and intersections:

- `(type1 | type2)` is reminiscent of the logical or, indicating that a type compatible with type1 or type2 would also be compatible with the type union.
- `(type1 & type2)` is likewise reminiscent of logical and, indicating that a type must be compatible wit both type1 and type2.

01.03.03 Type Unions

Type unions are a fundamental component of Lodge's typing system. A type union consists of one or more types that are combined into a single type that allows for the inclusion of any type. The interface for a given type union is the intersection of the public interface for all of the types in the union. Any value compatible with the intersected interface of the type un ion may be placed in any variable or argument whose type is the type of the union.

The only types that cannot be included in type unions are Concrete Variables.

Defining type unions

A type union can be defined by separating several types with the `|` (pipe) character. This is a special syntax for type unions and **not** an example of the `|` operator operating on type objects.

```
(int | string) variable := ...    !! Declaring a variable with an unnamed union
```

An unnamed type unions refers to all cases where the type union is not bound to a name such as the case below where the variable is being declared to have a type which is the union if int and string

It is also possible in lodge to bind a type union to a name using the `as` keyword. After defining a name for a type union, that name can be used as shorthand to refer to the union after the name definition statement and in all enclosed scopes. The name can be used in any case where the type union could have been used including variable declaration, type switching, and even the definition of further type unions.

```
(int | string) as intstr

intstr variable := ...
```

The Interface of a Type Union

The interface of a type union is the interface intersection of the interfaces of all of the types that make up the union.

01.03.04 Errable and Nonable Types

Errable and Nonable types simply refer to any type union that contains `Err` or `None` respectively. These use cases are so common that they have their own special syntax for declaration.

Errable

Types with the `!` token are unioned with `Err`

```
(Err | Str)

!! Is the same as

!Str
```

Nonable

Types with the `?` token are unioned with `None`

```
(None | Str)

!! Is the same as

?Str
```

01.03.05 Type Switching

For types inside a type union, it may sometimes be useful to treat different types differently. To achieve this while maintaining type safety, type switches can be used. The difference between a type switch and a simple if-statement is that, within the switch, the variable will be treated as that type for the purposes of type checking. Example of a basic type switch.

Inside each of those blocks, the programmer can reference the original variable as if it were the more specific type without breaking type safety.

```
swype variable {
        List : { print(length(variable)) }
        Int  : { print(variable) }
        *    : { !- If there were other possible cases, the * captures all of them -! }
}
```

Switching Concrete Types

By default, type switching simply tests if the object being switched is compatible with the interface.

Switching multiple types at once

This is particularly useful for imitating the behavior of Method Overloading

```
(int | str) a := 10
(int | str) b := "20"
```

```
swype a, b {
        int, int : { !- Code -! }
        str, str : { !- Code -! }
        str, *   : { !- Code -! }
        *, int   : { !- Code -! }
}
```

01.03.06 Type Intersections

A type intersection is the opposite of a type union.
It is an important part of promising an interface, as the required public interface of a class that promises multiple interfaces is equivalent to the type intersection of those interface types.

Syntax

`(type1 & type2)`

This form is most useful for combining component interfaces and requiring objects to promise several interfaces for compatibility with type intersections. For example, it might be the case that you want to require that a class promises an Iterable interface and also serializable with something like this: `(Iterable & Serializable)`

Thus, Lodge code is able to be fully agnostic to the concrete type of an object while still mandating the behavior it needs from the object.

Interface of a Type Intersection

The interface of a type intersection is the interface union of the participating types.

01.03.07 Interface Set Operations

Interface Intersection and Interface Union

These operations differ somewhat from strict definitions of mathematical set operations as the equivalence of interface members is depended on both the name and types of the members. Additionally, the merging of two members is more complex than simply checking for their presence. See the pages on interface intersection and interface union for more information on this behavior.

The naming of type type unions and type intersections might seem counter-intuitive as the result of a type union is an interface intersection and vice versa. However, this naming is sensible when you consider the set of compatible types for a give type union/intersection:

- When you create a type union out of several types, the set of interfaces that are compatible with that type union is the mathematical union of the sets of types that are compatible with the types that participate in the union.
- When you create a type intersection out of several types, the set of interfaces that are compatible with that type intersection is the mathematical intersection of the sets of types that are compatible with the types that participate in the intersection.

01.03.08 Interface Intersection

Interface intersection is the process by which multiple interfaces are combined into one for the purposes of type unioning.

When several types exist in a union together, that type union will have its own interface that is the intersection of the public interfaces of all participating types.

It is worth noting that not every intersection is possible. Since any setters or method arguments require their types to be intersected, there is the chance that a namespace collision results in that type intersection being impossible.

Process of interface intersection

Each interface contains a set of members that are either getters, setters, or methods with arguments.

During interface intersection, each member that has a corresponding member in every interface is a candidate for being included in the intersection. Candidate members do not need to have the same type, but they do need to be the same sort of member (getter / setter / method).

Notably, any member that doesn't have a matching name in all interfaces will not be included in the intersection at all.

Then, each candidate is evaluated to see how it can be merged together to form a member of the final intersected interface.

How this merging works depends on what kind of field, but the result of this merging is either an update of the member's set of types or a removal of the member from candidacy entirely.

Merging Candidate Getters

Any code dealing with the intersection of several getters must be able to handle the types from any of the getters matching the candidate. Thus, the type of the member in the final intersection is simply the type union made up of all of the members matching this candidate.

For example, when merging the two following interfaces:

```
interface A {
	get val int
}

interface B {
	get val string
}
```

The result of `(A | B) as C` would be the following:

```
interface C {
	get val (int | string)
}
```

In this case, when accessing the `val` getter from a variable of type C, the object stored behind the interface C could be either a B or an A, and `val` could either be an int or a string. This means that code that deals with C has to be able to handle the type of `val` regardless of what it is.

In the case where the type union of the getter types results in an empty interface, that getter will be removed from the union. If that getter refers to a field in the class, that field becomes, write-only.

Merging Candidate Setters

The type of a setter in the intersected interface must only be able to accept types that **all** setters matching the candidate can accept. Thus, the new type of the setter will be the type intersection (not interface intersection).

For example, when merging the two following interfaces:

```
interface A {
	set val int
}

interface B {
	set val string
}
```

The result of `(A | B) as C` would be the following:

```
interface C {
	get val (int & string)
}
```

Merging Candidate Methods

This section needs to be expanded

For merging methods, candidacy is still determined by the name of the method, but the type of a method is more complex.

For the sake of flexibility, the names of method arguments are not considered at all, only their types. Due to their similarities arguments are treated as setters, and return values are treated as getters.

The merging of method arguments begins at the first pair of positional arguments and merges their types, continuing until their is a failure or until one method is out of arguments.

If one function has more positional arguments than another, the merge will fail unless the excess arguments are optional (have default values)

If

Keyword arguments supplied statically in the code are compiled in the same way as arguments provided by keyword.

01.03.09 Interface Union

A union between multiple interfaces is the result of a 06 Type Intersections.

Interface unions are somewhat more complex than interface intersections, as there are cases where the union will fail due to incompatibility between

Process of interface union

Read 08 Interface Intersection first as that page has more details and examples of this process.

To start with, all members from every participant are candidates for inclusion in the union

However, there may be some elements with shared names, that require special consideration for their types. This is also the way in which some interface unions may fail.

Getters with shared names

An object that is compatible with the union of multiple interfaces must be able to properly behave like any of the of the types in the interface union. For example, consider the type intersection `(A & B)`. Any class that would be compatible with this interface union must be compatible with both A and B in every case. Thus, if both A and B have a getter of the same name, those two getters must be mutually compatible (A is compatible with B and B is compatible with A). Mutual compatibility requires that the two interfaces are identical.

If there are getters with shared names and all getters do not have the **same exact interface**, the interface union will **fail with a compiler error**.

For example, consider merging the two following interfaces:

```
interface A {
	set val int
}

interface B {
	set val string
}
```

The line `(A & B)` would result in a compiler error.

The failure of union is a non-ideal outcome, thus it is best practice in lodge to avoid constructing classes that are likely to result in failures when unioned with interfaces that they are likely to be unioned with.

- Reducing the public interfaces of classes as much as possible
- Using distinctive names for members to reduce collisions

Setters with shared names

Setters with shared names are somewhat simpler as the types of each setter is simply the type union of all the candidate setters.

Much like in interface intersection, when the interface of a member becomes empty as the result of type unioning, that member is removed from the resulting interface.

For example, when merging the two following interfaces:

```
interface A {
	get val int
}

interface B {
	get val string
}
```

The result of `(A & B) as C` would be the following:

```
interface C {
        get val (int | string)
}
```

Methods with shared names

The behavior of method merging is somewhat complex and I haven't worked it out all the way yet.

01.03.10 Types as Objects

There are several situations where treating classes as objects might be useful.

It can be useful in dependency injection where a given object still maintains the responsibility of instancing an object, but it shouldn't have to worry about what the specific class of that object is.

Additionally, it is useful in the definition of the Type Conversion Operator operator, which should be able to take a class as an argument.

Type of a Type object

This section needs to be expanded

From the perspective of interaction, types look like functions that are called to create instances, but also have static fields.

This makes type objects complex to put into the rest of the type system.

02.00 Introduction

This section is not complete and mostly just consists of a set of notes used to guide implementation.

Many of these decisions are deffered to the time at which implementation actually begins, so this section will be filled out as those decisions are made.

02.Co.Automatic Interfacing Implementation

#lodge_implementation

See: 05 Automatic Interfacing

Automatic interface is achieved by comparing each interface in a given lodge program to each concrete type to determine if that class is compatible with that interface. If the interface is compatible, an entry for that interface is placed into the ttable for that class.

The Lodge Intermediate Format (.lift) is an important component of this process, because each lift file contains a complete set of all interfaces defined in that code segment. Interfaces can come from explicit interface definition, but also through type unions When lift files are combined, those lists are joined together. During each LIFT merge, all typing information is joined together.

When a lift file is finally compiled into machine code, the process of automatic interfacing is finalized and types can be erased for the most part.

#lodge_implementation_decision

02.Co.Lodge Compilation

#lodge_implementation

Compiler Frontend

Lodge Parsing
Preprocessor
Tokenizer
LR Parser

The compiler frontend takes as input a lodge source code file, and outputs that `.lift` file representing that lodge file.

Intermediate Format

Lodge Intermediate Format (.lift)
The intermediate representation represents individual lodge files rather than entire programs.

The purpose of the intermediate format is to hold all of the information that the compiler backend needs to produce a complete lodge program such as relevant parse trees, classes, interfaces, type unions, and information necessary for performing name binding, automatic interfacing and multi-file static type analysis.

In the final compiler, this format will most likely not be saved to disk during the process. However, libraries will probably be stored and/or distributed in this format to reduce compilation overhead.

Interpreter Backend

During the development process, an interpreter for lodge will likely be implemented. The purpose of this

The design of the language, however, is careful in ensuring that the language   be required with minimal runtime processing and a minimal runtime package needing to be included in compiled files.

Some of the code for this Interpreter could potentially be reused for the final compiler as well.

Compiler Backend

An LLVM compiler backend is the goal of this project.
Because of the fact that LLVM includes well-developed optimization steps that I don't want to spend the time poorly recreating, the .lift format will most likely not include anything but the most basic optimizations.

02.Co.Lodge Intermediate Code Representation

The code format used by .lift

Philosophy

Most likely a bytecode format of a lower-level representation of Lodge code.

It will have all symbol names and type information preserved in order to allow further compilation with other .lift files

I want it to be a good target for interpretation as well as further compilation as I believe an interpreter backend will be a simpler than the compiler backend

Specification

Inspo

LLVM

https://llvm.org/docs/BitCodeFormat.html

Python

https://opensource.com/article/18/4/introduction-python-bytecode
https://aosabook.org/en/500L/a-python-interpreter-written-in-python.html

JVM

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html
https://docs.oracle.com/en/java/javase/21/vm/java-virtual-machine-technology-overview.html#GUID-982B244A-9B01-479A-8651-CB6475019281

C#

https://en.wikipedia.org/wiki/Common_Intermediate_Language

02.Co.Lodge Intermediate Format (

#lodge_implementation

Stored in `*.lift` files (Lodge Intermediate FormaT)

LIFT are a **cross-platform** mid-level representation of Lodge code intended to hold, in a minimal representation, all of the information required to compile the final program.

Each LIFT file represents a subset or entirety of a Lodge program. Lodge source code files can be compiled into lift files, and LIFT files can be merged together into a single a single file. This merging process is where external references and interface resolutions occur.

Merging LIFT files

The specifics of this process are still foggy.

LIFT Code Representation

Lodge Intermediate Code Representation

02.Co.Lodge Parsing

#lodge_implementation

Of course, we will need to know more about 01 Statements first.

Preprocessor

This step is in charge of cleaning up certain things that are easy to recognize at this stage that would otherwise make the tokenizer

Deals with:

- Removing Comments
- Compiler macros?
    - Don't know if Lodge will have this.

Tokenizer Based on Regular Expressions

Types of Tokens

- numerical literals
    - 10
    - 1.5
    - .5
    - 0xFF
    - 0b1010
- string literals
    - How to deal with string literals in the tokenizer?
- operators
- keywords
- identifiers
    - Identifiers are any alphanumerical tokens with a given format excluding keywords and alphabetic operators like not, and, or or.
    - The exact format has not been settled on yet

Parser Based on a Context-Free Grammar

Probably in LR form for more readable error messages.

02.Co.Untitled

02.Concrete Variables

Concrete Variables

In Lodge, concrete variables are variables/arguments/fields that do not participate in Lodge's typing system in the typical way including automatic interfacing, and inclusion in type unions.

Concrete variables for reference types

I'm not sure if it will be possible to make concrete variables for reference types.

There aren't a lot of scenarios under which you would want to create a concrete variable for a reference as it   always be the case that an object with a compatible interface has compatible behavior as well.

The other side of this is that it might be irritating to the programmer to not be allowed to do this: It's not generally the philosophy of lodge to disallow things that are bad practice, but rather to design the language such that best practices are intuitive.

02.In.Lodge Interpreter

The Lodge interpreter will take .lift files as input.

The Lodge Intermediate Code Representation will be designed in order to facilitate compilation natively, but also interpretation to provide flexibility.

Deciding on the exact format of the interpreter, but I am going to be looking at the python interpreter, the java virtual machine, and .NET

02.Lodge Function Implementation

#lodge_implementation

Function definition Locations

Regardless of where a function is defined, its code will be in the code segment of the compiled program.

- The main body of a code file
  - The function will be assessable from anywhere in the file or from any file that imports that one.
- Inside the body of a class/struct
  - It will be treated as a class/struct method and will be bond to
- Inside the body of another function.
  - A new function object will be created at the line where the function is defined and bound to the name it is defined as within the scope it is defined under.

Function Closures

When a function is defined inside of an existing scope, the values from that enclosing scope will be captured into the function. This will probably be implemented as some sort of function object that holds a pointer to the code section of that function as well as all of the values for the scope that it captures. So, although function code would exist statically in the code section of the compiled program, there can still exist multiple function objects that reference that code. That way, the function object can hold the captured scope, so multiple different instances of the function can be created with different values for the variables at different times.

It is known at compile time which variables inside of a function definition are captured, so the function object types can be defined at compile time to include fields for those objects. And when the function is called, those values are placed into the stack along with the other local variables to the function.

Generator Functions

Whenever a function in lodge is defined with a yield statement, it is a generator function.

Upon the first yield statement being encountered, the function state is copied into a newly created function closure, and the function returns that closure. That function closure also

When the generator is entered for the second time, the values of local variables are copied out of the closure and back onto the stack, then, the function will jump back to the location that it was most recently yielded out of.

I still need to make a decision about what generator scheme I'm going to use:``z
generator functions

There might also be a behind-the-scenes optimization possible where each generator function contains two code versions: the generator creator and the generator function. This way, the creator function would not need the labels or jump table. The generator function would not need to have the logic for creating a new closure and populating. It could also store its state directly in the closure through some pointer mechanism rather than having to copy it between the closure and the call stack each call.

02.Lodge Garbage Collection

Lodge will use a Garbage Collection system.

I haven't decided between a Reference-Counting Garbage Collector or a Tracing garbage collector. The decision may come down to trying both and profiling, but that would be a lot of work.

I am considering the possibility of using a Reference-counting garbage collector that occasionally employs a tracing algorithm to collect objects with cyclical references.

Alternatively, I could use an off-the-shelf garbage collector

02.Lodge Generics Implementation

It seems likely to me that the implementation of generics would mostly be static, and the same class body could be used for all generic values.

02.Lodge Text Encoding

#lodge_implementation

In theory, lodge Strings should have support for many various text-encoding schemes like ascii, ISO 8859, and utf-8, but that will probably come later.

The first few versions will probably only support ascii/ISO8859 strings.

I see no reason, however, not to support all of those things in the compiler, so lodge code will be able to be written in several encodings.

02.Lodge Thing

In Lodge, all values correspond to a simple structure in memory called a `thing` that holds that variable's value.

A thing is a pair of two fields. The first holds the actual value that the thing represents and the second field is a reference that refers to the type of the object.

The type field of the thing is also a pointer that points to a structure in memory that points to the ttable of that concrete type.

02.Lodge ttable

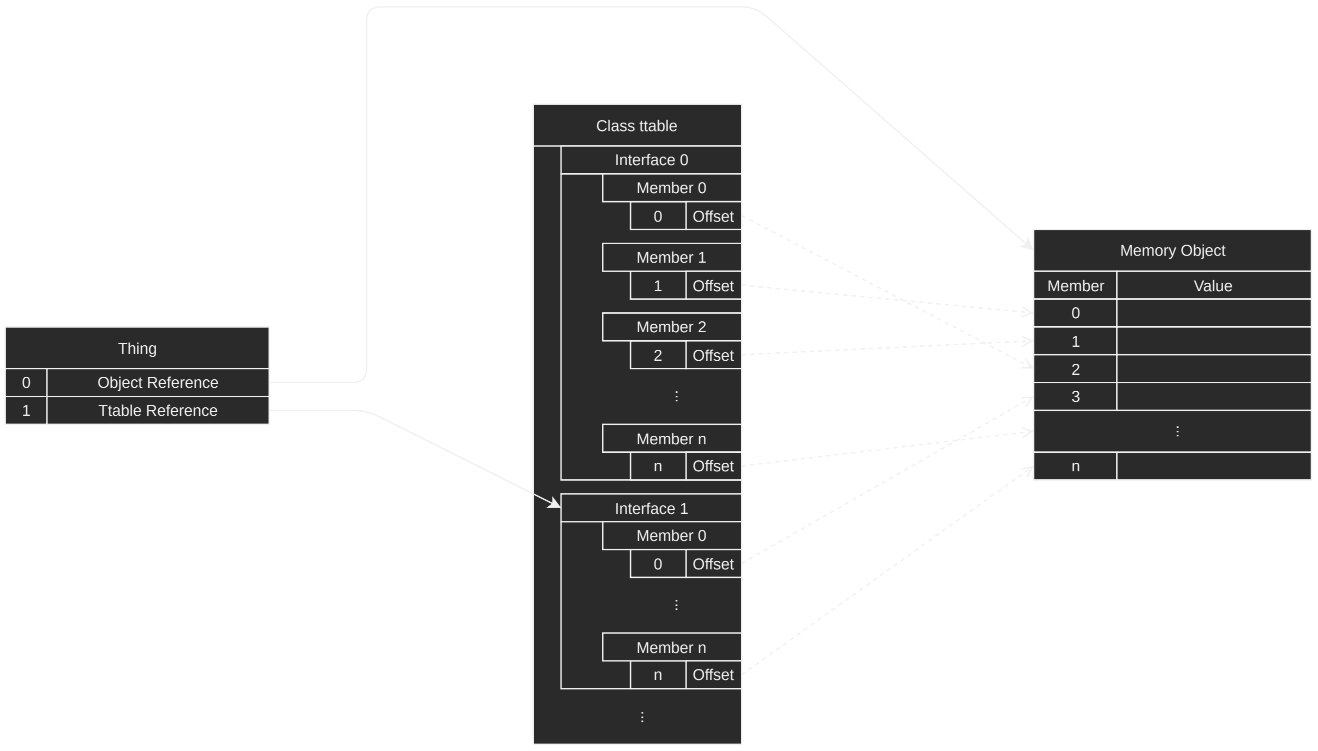
#lodge_implementation

Ttables in lodge are the fundamental way that it's typing system is implemented.

When a reference type is held inside of a variable it's thing points to that value in memory as well as pointing to the entry in its ttable that translates from the interface of that container to the fields and methods of that value's concrete type.

In compiled Lodge code, object members are accessed via field IDs that are specific to the given interface of the container that the value is in. The ttable is used at runtime to take these IDs and translate them into offsets within the structure in memory that represents that particular object
The result of this is that every single access to a field in Lodge results in an extra memory access required to get the offset from the ttable.

Here is a diagram depicting the relationship between thing, ttables, and the structure in memory that represents a given instance object:



02.Value Types

#lodge_implementation

There is a potential wrinkle in some cases where, despite compatible interfaces, the programmer would not want a type to work with another.

See 03 Built-In Types for more information on the usage of value vs reference types.

.lodge_spec

The Lodge Programming Language Specification

This is the first version of the specification document for the Lodge programming language (that is yet to be implemented).

This document is intended to describe the Lodge programming language as completely as possible as if it existed. This document acts as if the language has an existing implementation and takes the perspective of documentation/tutorial.

The purpose of this document was to create a concrete description of the language to aid in high-level design decisions before implementation work began. Additionally, this document will act as the location for further description of the language with the hopes that it can be complete documentation by the time that preliminary implementation is complete.

Currently, there are several important pieces of the specification missing and some remaining inconsistencies. Despite that, this document describes the fundamental concepts and syntax of the Lodge programming language in detail.

00.00 Preface

Specification Version: 0.0.1
Language Version: 0.0.1

Author: Dylan Carroll

Versioning

The versions of the specification and the language consist of three numbers. For the spec version and the language version, the first two numbers of each represent major and minor versions.

Major versions indicate major revisions to the language and large breaking changes.

Minor versions indicate minor revisions and compatibility-preserving changes.

The third digit of the specification represent sub-minor versions of the specification. These indicate changes to the specification that don't constitute an update to the language. Changes for clarity or organization would be considered sub-minor spec versions.

Likewise, the third digit of the language are sub-minor versions of the language. Sub-minor versions of the language do not reflect changes in the specification and thus do not correspond to new features or changes in behavior. Changes relating to fine implementation details, refactoring, and performance improvements may constitute sub-minor language versions.

Contents

01 Specification

This section describes how the lodge programming is intended to be used and behave

02 Implementation

This is a description of how Lodge is implemented. It is intended to function as documentation for the official implementation of lodge.

03 Style Guide

01.01.01 Statements

Statements

As lodge has no statement terminator, the end of statements is automatically inferred by newline characters in most cases. However, there are cases where multiple lines can be considered a single statement. This means that Lodge is not totally whitespace agnostic like other languages. It is still agnostic towards tab and space characters.

In Lodge, the statement termination happens at newlines by default, but statements may be joined together based on some rules.

1. If a line opens a bracket (such as: `( [ <`), all lines until that bracket is closed will be treated as a single statement. However, the closing bracket will only be searched for until the end of the current block.
2. If a line ends with an operator that requires another following operand, the following line will be treated as part of the same statement.
3. If a line begins with an operator that requires an operand beforehand, the previous line will be treated as part of the same statement.

```
!! Examples of rule 1
Int x := f(arg1,
               arg2,
               arg3)

List l := <val1, val2,
               val3, val4>

!! Example of rule 2
Int y := 10 +
         11 +
             12
```

```
!! Example of rule 3
Int z := 10
       + 11
       + 12
```

01.01.02 Comments

Comments

Comments are created with the `!!` token. Everything after the `!!` until the next newline will be considered a comment and completely ignored.

Bounded comments can be created with `!-` and `-!`. Everything between these two tokens will be ignored as comments.

01.01.03 Built-In Types

Integers

value type

size	signed	Unsigned
8	`i8`	`u8`
16	`i16`	`u16`
32	`i32`	`u32`
64	`i64`	`u64`

Floats

value type

size	name	
32	`f32`	
64	`f64`	

Booleans

value type

Size	name
8	bool

Strings

reference type

Str

- A reference type object much like strings in other languages
- Basically just an immutable array of chars
- Lodge Text Encoding
- String literals are created using text surrounded by double quotes like this:

```
Str aString := "A new string literal"
```

None

value type

Lodge None type can only have a single value: None
The None type has a totally empty interface.

Uses

- Indicate that something does not exist in cases where using null doesn't make sense.
- When a variable has a None in its union, the interface is empty and you would need to check that the value is not None in a type switch before you could do any operations with it.
- If that variable is always initialized to None, it could eliminate the chance of null pointer exceptions by forcing the None value check.
- Used as a default value for function arguments when that argument doesn't always need to be present. This is useful for variatic functions and optional parameters.

Data Structure Types

All data structure types are reference types

Tuple

- literal: `(value1, value2, value3)`
  - `(value,)` for a single value
- type definition: `(type1, type2, type3)`
  - `(type : size)` when it's all the same type
- immutable value types
- You must declare the type for each variable in the tuple

```
(Int, Str) tuple1 := (10, "hello")
(Int,) tuple2 := (10,)
```

```
(Int: 5,) tuple3 := (1, 2, 3, 4, 5)
(Int: 2, Str) tuple3 := (1, 2, "a", b")
```

Array

- literal: `[value1, value2, value3]`
- type definition: `[type : size]`
  - note: arrays can only have one type (or union)
- mutable reference types
- Cannot grow or shrink
- The type (union) must be the same for each element in the array

```
[int: 10] arr1 := [0, 1, 2, 3, 4]
var arr2 := [0, 1, 2, 3, 4]
```

List

- literal: `<value1, value2, value3>`
- type definition: `<type>`
- mutable reference type
- can grow and shrink

```
<int> list := <0, 1, 2, 3, 4>
var list2 := <5, 6, 7, 8, 9>
```

Dictionary

- literal: `{ key1 : value1, key2 : value2}`
- type definition: `{type1 : type2}`
- Mutable reference type
- Implemented as a hash table

```
{str : int} aDictionaty := {"A" : 1, "B" : 20, }
```

Set

- literal: `{value1, value2, value3}`
- type definition: `{type}`
- Implemented as a hash set

```
{int} set1 = {2, 4, 6, 8}
```

Value Types vs Reference Types

All values in rust are represented in memory by a thing.
The thing contains a field for the value and a field for type information.

For value types, the value field is the value itself, but for reference types the value is a reference to the object in memory.

For smaller value types like `i8`s, there are wasted bytes in the high end of the value portion of the thing. To combat this, some types such as arrays may be implemented in such a way as to store raw values rather than things when the type is restricted to that of a value type.

01.01.04 Operators

Mathematical Operators

op	action
+	addition
-	subtraction (and unary negation)
/	division
//	integer division
*	multiplication
%	modulus
**	power
/^	root

Logical Operators

The operators here that are words are reserved keywords

op	action
and	Logical and
or	Logical or
not	unary logical negation operator
xor	exclusive or
==	value equivalence
is	identity equivalence
≠	not value equivalent
isnt	not identity equivalent
>	greater than
<	less than
≥	greater than or equal to

op	action
≤	less than or equal to

The value equivalence operator `==` should not be confused with the assignment operator `:=`, or the function keyword token `=`.

Bitwise Operators

op	action
&	bitwise and
|	bitwise or
^	bitwise xor
~	bitwise not (compliment)
>>	right shift
<<	left shift

Assignment Operators

op	action
:=	assignment
++	increment
--	decrement

There are also forms such as `+=`, `-=`, `*=`, `/=`, and `**=` that are shorthand for performing an operation and then assigning the result to the first operand (assuming that operand is an existing identifier). Any operator except assignment operators and unary operators followed by a = will be treated as being a part of this shorthand.

Other Operators

op	action
:	type conversion
()	call as a function
[]	indexing and slicing
!	Error Resolution
?	None Resolution Operator
#	hash
$	other
@	other

Type Conversion Operator

The type conversion operator `:` is the way to convert an object of one type into another.

It is particularly useful for conversion into strings for representational purposes:

```
Str myStrRepr := myObj:str
```

Or for conversions between numerical types

```
Int myInt := myFloat:int
```

Error Resolution Operator

Shorthand for resolving errors

```
fun function (Err | Str) {
	(Err | Str) val := errableFunction()
	swype val Err {
		return Err
	} swype * {
		!! Operations on val as a Str
	}
}
```

Is equivalent to

```
fun function (Err | Str) {
	Str val := !errableFunction()

	!! Operations on val as a Str
}
```

In Type Definitions

The `!` token is also used to make an Errable type:

```
fun errableFunction() (Err | Str) {
	!! ...
}
```

```
fun errableFunction() $Str {
	!! ...
}
```

None Resolution Operator

The None resolution operator applies to variables whose type is a union that contains None. It is essentially a shorthand for an otherwise common type switch.

The following two code blocks are equivalent:

```
?Int temp = nonableFunction()

Int variable = swype variable {
        None : {-1}
        *    : { temp }
}
```

```
Int variable = nonableFunction() ? -1
```

01.01.05 Variables

Declaring and Assigning Variables

Lodge variable declarations work much like they do in C or Java. A variable declaration consists of a type followed by a variable name.

Assignment occurs using the `:=` operator, which often

```
Int value    !! Decalring a variable without defining it
Int value2 := 10    !! Declaring an int variable and assigning it a value
```

Assignment also does not need to be a standalone statement, but can be an expression that returns the new value, like in C.

```
!! Print out the characters from a file until the character 'a' is reached
Str c
loop while (c := file.getChar()) != 'a'
        { print(c) }
```

Lodge is garbage-collected, so created values do not need to be manually freed.

Var Keyword

When variables are declared with the `var` keyword instead of a type,

```
var value4 := "hello"    !! Declaring a variable as a string with an inferred type
var arr2 := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]    !! Declaring a type with an inferred type
variable
```

Variable Types

In Lodge, the type of a variable does not restrict the values that can be stored to just the type of the variable. Instead, any value with a compatible interface can be stored in that variable. For example, a variable with the default `Int` type can store any integer value.

```
!! Allowed
x := i8(-10)
x := u64(100)

!! Not Allowed
x := "10"
x := 10.5
```

The full behaviors of Lodge's type checking system is found in Chapter 04: Typing.

Union Types

Instead of creating a variable with just a single type, it is also possible in Lodge to define a variable with a composite type called a type union. This allows a variable to store any value that has a compatible interface with type in the union.

```
(Str | Int) x

!! Allowed
x := 10
x := "Hello World"

x := i8(-10)
```

Composite types can also be given names with the `as` keyword

```
(Str | Int) as Strint
Strint x = "10"
```

Auto Keyword

Since variables in lodge are defined entirely based on interface compatibility than particular concrete types, it is possible to automatically declare a variable whose type interface is defined by what members are ac

The auto keyword will look at the entire scope that it's variable is being bound in to see what members of that variable's interface are being accessed. That then becomes the interface of that variable.

Take this example:

```
fun functionName(auto arg) {
        arg.name = "Name"
        Float result = arg.takeAction(10)
}
```

For this function the `arg` variable will have the following interface:

```
Interface {
        Str name
        fun takeAction(Int) Float
}
```

Literals

Numerical Literals

- numerical literals
    - `10`, `1.5`, `.5`, `0xFF`, `0b1010`, etc.
- string literals
    - "Text inside quotations"
- operators
    - `+`, `-`, `*`, `/`, etc.
- keywords
    - Like `if`, `loop`, `class`, `fun`, etc.
- identifiers
    - Like the names of variables, types, classes, etc.

Circular Type Definitions

There may be cases where a composite type declaration may want to refer to itself, which is allowed in Lodge.

Here's an example of how a tuple containing two values can be used to create something akin to a linked list

```
(int, listelement)|None as listElement

listElement list = (0 (1, (2, (3, None))))
```

01.01.06 Control Flow

Blocks

Blocks are opened and closed by curly brackets `{}`. Many statements such as if statements, loops, and class definitions require blocks.

Blocks in lodge create a new scope that can access variables from the outer scope, but whose variables only last during the scope of.

```
Int x := 0
loop while x <= 10 {
        if x%2 == 0
                { print(x) }
        x++
}
```

Naked blocks can also be created without an opening statement simply to create a new scope and group together a sequence of statements.

Block Expressions

If a block is placed in a location where it is evaluated as an expression rather than a standalone statement, the last statement of a block is treated as an expression that the entire block evaluates to.

For naked blocks, this can simply be used as a way to cluster a sequence of statements into a single unit that returns a value. It has essentially no impact on the way the code executes, but may be used for conceptual groupings of actions.

When an if statement is used as an expression, evaluating blocks can be used for constructs like ternaries.

When a loop is used in an assignment, each iteration of the loop is evaluated separately, and placed into a list that returns when the loop completes.

Conditionals

Conditionals use the `if` keyword, a boolean expression, and a block to create an if-statement. If the given boolean expression evaluates to true, the block is entered. Otherwise, it is skipped.

If

```
if boolean_expression {
        !! conditional body
}
```

If / Else

You can also specify as block to enter if the boolean is false using the `else` keyword.

```
if boolean_expression {
        !! Enter if true
} else {
        !! Enter if false
}
```

If / Else If / Else

And you can also chain if statements using `else if`

```
if exp1 {
        !! exp1 is true
}
else if exp2{
        !! exp1 is false and exp2 is true
}
else{
        !! exp1 is false and exp2 is false
}
```

If Expressions

You can use if statements like an expression and the selected block will be evaluated

```
Int x := 10
Int result := if (x < 0) {
        -1
} else if (x == 0) {
        0
} else {
        1
        }

!! result = 1
```

Loops

Lodge has four kinds of loops:

- plain loop
- while loops
- for loops
- over loops

Plain Loop

A plain loop is a loop without any parameters. It is essentially the same as a `loop while true`, but it does not perform a comparison. It is usually used for replicating a do-while type pattern using a `break if` statement.
Here is an example.

```
!! Read one charafter from the file onto the stack until a period is reached
loop {
        c := file.read()
        stack.push(c)

        break if c == '.'
}
```

While Loop

A while loop takes a boolean expression and will repeat its body code until the expression evaluates to false.

```
!! Print all the values on the stack
loop while not stack.is_empty() {
        ints val := stack.pop()
        print(val)
}
```

For Loop

This kind of loop is merely for looping over ranges of integers. Much like similar languages (besides python) this is essentially just syntactic sugar for a declaration, a while loop, and an increment. It uses the keywords `from`, `to`, and `by` to control this range. All of these except to are optional. If `from` is left out, the default is 0. If `by` is left out, the default is 1.

The type of the loop variable does not need to specified and will default to Int with a concrete type of i64.

If the variable exists already, its type will remain the same and its value will be assigned to the loop values.

Here is an example. The top of the range is exclusive.

```
!! Print all numbers less than 100 divisible by 3.
loop for x from 0 to 100 by 3
        print(x)
```

Over Loop

The over-loop works like Python's for-loop or for-each loops in other languages. It accepts any object that is compatible with the iterator interface and loops. It uses the `from` keyword to associate a name with the element from the iterator.

```
!! Print all of the characters in the list
loop over c from char_list
        print(c)
```

If you want to loop over multiple iterators of the same length at once, you can separate several associations with a comma as below.

```
loop over a, b from list1, list2
        list3.append(a + b)
```

You can also add an option that will keep track of the index with the `at` keyword.

```
!! Print all of the characters in the list
loop over c from char_list at i
        print(c)
```

Break and Continue

The keywords `break` and `continue` work as they do in similar languages.
`break` will exit the nearest loop that encapsulates it.
`continue` will skip to the beginning of the nearest loop that encapsulates it

Lodge also has the `break if` and `continue if` statements which act as conditional statements

For example, these two code blocks behave the same:

```
loop over value from someList {
        print(value)

        break if value == null
}

loop over value from someList {
        print(value)

        if value == null {
                break
        }
}
```

01.01.07 Functions

Declaring functions

Function declarations in lodge use the `fun` keyword. When declaring a function, the name, arguments, return type, and function body must be specified.

Synyax

Defining a function with a name

```
fun functionName(type1 arg1, type2 arg2, ... ) returnType {
        !! Function body
}
```

Defining an anonymous function

```
fun (args) returnType {
        !! Function body
}

!! Arrow notation for brevity
(args) => { !- Function body -! }
```

Declaring a variable with the type of a function.

```
fun (type1, type2, type3, ...) returnType variableName := !! ...

!! Howvever, it  is less verbose to use var
var variableName := fun (args) returnType { !-Function body-! }
```

Variadic Functions (Variable number of arguments)

Optional arguments

- You can give a function optional arguments by giving those arguments default values
- When calling the function, those arguments don't need to be given
    - If you want to specify only an argument that comes after an optional argument, you must give that argument with the `arg = val` syntax.

```
fun f(?int a = None, ?int b = None) {
        swype a, b {
                None, None : { !- case 1: Called with no arguments -! }
                int, None  : { !- case 2: called with a only -! }
                None, int  : { !- case 3: called with b only -! }
                int, int   : { !- case 4 : called with both arguments -! }
        }
}

f() !! case 1
f(1) !! case 2
```

```
f(b=2) !! case 3
f(1, 2) !! case 4
```

<u>**Variadic Arguments**</u>

- Define a function with variadic arguments by placing `...` after the last argument. That name will then be treated like an array of those values within the function body. Other arguments

```
fun f(int a, ints vargs...) returnType {}
        print(length(args))
        print(args[a])
}

f(1, 2, 3, 4, 5) !! prints 4, 3
f(4, 0, 0, 0, 0, 100, 0) !! prints 6, 100
```

Function/Method Overloading

Lodge doesn't have function or method overloading explicitly as type information could not be used to unambiguously select between functions based on their signature.

The same techniques used for optional parameters can be used to mimic the behavior of function and method overloading. Essentially, the behavior a function can be configured to depend on the types of the given arguments and which arguments are given.

Argument Expansion

You can pass sequences of values (lists, arrays, and tuples) into the arguments of a function with the * prefix. This only works if the types of the sequence match with the types of the corresponding arguments of the function. This also works for variadic functions. And the passing can be a subset of the arguments

```
fun func1(int a, int b, int c) {}
        !! ...
}

fun func2(int a, str b, int c) {
        !! ...
}

!! Expanding arrays
[int] arr1 := [1, 2, 3]
func1(*arr)
func2(*arr) !! type error because the second argument type does not match

!! Expanding a tuple with heterogenous types
(int, str, int) tuple := (1, "2", 3)
func2(*tuple)
```

```
!! Passing a subset
[int] arr2 := [1, 2]
func1(*arr2, 3)
```

Generator Functions

generator function implementation

There is no special syntax for generator functions besides the yield keyword. Any function with the yield keyword in the function body can be a generator function.

When the yield keyword is encountered, that function instead returns a generator alongside the value being. This generator is a new function closure that represents the state of the function

When defining a generator function, the return type is expressed as the following generic: ```

```
Generator<type>
```

Here's an example of a generator

```
fun FibonacciGenerator() Generator<Int>{
        a, b := (0, 1)
        loop {
                a, b := (b, a+b)
                yield b
        }
}


Generator<int> gen = FibonacciGenerator()

print(gen.next()) !! 1
print(gen.next()) !! 1
print(gen.next()) !! 2
print(gen.next()) !! 3
print(gen.next()) !! 5
```

01.01.08 Scoping Rules

01.02.01 Classes

Class Definition Syntax

Basic Syntax

```
class ClassName  {
        promises Interface1, Class1
```

```
        uses ParentClass1, !ParentClass2

        Int field

        fun Method() ReturnType {
                !! Method Body
        }
}
```

Anonymous Classes

Classes can also be created without names in order to allow the creation of an object without needing to write a dedicated named class for it.

The anonymous class will participate in the typing system as normal, and can even be placed in other variables using the `var` keyword.

```
var anonObj := class {
        !! Class body
}()
```

Even though the class does not have a defined name, it will still have an internal name and interface like any other class.

Promising

When a class promises an interface or a class, all public fields and methods are required to be explicitly defined in the promising class.

If a promise results in a name collision with previous promises, this will be accompanied by an error unless the function signatures are identical / the field types are identical.

However, because of automatic interfacing, promising only makes a difference from the perspective the programmer and doesn't result in code that compiles any differently. The advantage to promising is that it becomes impossible to fail to implement ant of the promised methods or fields.

Even if a class/interface is not promised, it will be able to participate in polymorphisms via automatic interfacing as long as it matches the criteria to do so. Promising simply ensures that the programmer is required to satisfy those criteria as well as providing more self-documenting code.

```
class ClassName  {
        promises Interface1, Class1

        !! Class Body
}
```

Using

When a class uses another class, all methods and fields (external and internal) from the superclass are implicitly copied into the subclass. Methods created in a class via using satisfy promise requirements.

Lodge allows for using multiple classes. In the case of namespace collisions, the value from the class listed second will be used. However, if a usage overwrites the implementation of the class or any previous uses, this will cause a compiler warning. This warning can be suppressed by placing a `!` before the class name that caused the collision as seen in the example above.

```
class ClassName  {
        promises Interface1, Class1
        uses ParentClass1, !ParentClass2

        !! Class Body
}
```

Public/Private Methods and Fields

```
class A {
        int _a := 0 !! Private/Internal field
        int a := 10 !! Public/External field

        fun F(int arg) int {
                !! External function
        }

        fun _F(int) int {
                !! Internal functions
        }

}
```

The difference between external and internal class members is that internal members will not be considered as part of the class's interface for the purposes of promising, automatic interfacing, and type union definition.

Internal members can be accessed from inside the class, but also from within concrete type switches that switch on the specific concrete class.

Properties

Defining properties is done using the get and set keywords

```

class A {
        get propertyName type {
                !! Code for getting the value
        }
        set propertyName type {
                !! Code for setting the value
        }
}
```

The get and set type for a given property do not have to be the same. In fact, the get and set properties act pretty much independently from eachother despite the fact that they have thee same name.

Behind the scenes, properties are compiled as functions. Properties cannot be internal, but they can still be accessed from within the class.

From the outside, properties act just like fields.

```
A a := new A()
print(a.propertyName)
a.properyName := 10
a.propertyName += 10
```

Generics

```
class A<T> {
        T field := ....
}

A<int> a := new A<int>()
int val := a.field
```

When the `<T>` is specified, all instances of the type name `T` found in the class definition will be replaced with whatever the type is at the object creation. `T` could be any identifier, but a single uppercase letter is standard.

This is implemented by simply using the broadest possible type union to replace T. If nothing is specified, T is equivalent to the 03 Built-In Interfaces.

The type of T is known at compile time for a given object, so the return types of methods can be checked statically and type errors are avoided.

You can also specify restrictions on the generic types with interfaces or type unions.

```
class A<Iterable T> {
        ...
}

class B<(int | str | list) T> {
        ...
}
```

Generics vs Unions

There is a small, yet important difference between creating non-generic class whose member types are a type union vs creating a generic class whose generic type is that same type union.
The difference arises from the binding time of the type.
With the generic, the single allowed type is specified at object creation, whereas the type union allows all the types in the union forever. This difference is demonstrated by the following example.

```
class A {
        (Int|Str) field
}

class B<(Int|Str) T> {
        T field
}

A a := new A()
B<int> bInt:= new B<Int>()
B<string> bStr := new B<Str>()

a.field := 10
a.field := "string"

b1.field := 10
b1.field := "string"    !!Compile error

b2.field := 10          !!Compile error
b2.field := "string"
```

In class A, both strings and integers can be placed into the field regardless, but with class B, only the generic type specified at the creation of the object can be placed in the field.

Nested Classes

Classes can be defined within other classes. This class can be accessed within the outer class and from the program at large through the outer class.

This is a purely static concept. Class instances (objects) can't and don't contain classes

```
class A {
        fun F() B {
                b := new B()
                return b
        }

        class B {
                int v := 10
        }
}

A a := new A()
A.B b := a.F()
```

Operator Methods

```
class A {
        fun + (auto other) int {
                !! Logic for adding objects
        }
}

A var1 := new A()
A var2 := new A()
A var3 := var1 + var2
```

The syntax of the format `fun + () ...` is only legal within classes.

Type switching inside operator methods

Because of the way that interface intersection works in lodge, it is best to use the broadest possible type for the operator method. These methods are a particularly good use case of the auto keyword as the type of the argument grows automatically as the ways that the variable is used grows.

Reverse operator methods

When an operator is acting on two objects, the compiler will first look if the first object implements the operator on the type of the second object. If not, it will look to see if the second object implements the   of that operator.

For example, take the expression `a / b` for objects of classes `A` and `B`.

If `A` implements `/` for the type of `B`, then that method will be called on `a` with `b` as an argument.

If `A` does   implement `/` for the type `B`, then the compiler will check for the `~/` method in `B` on the type of `A`.

The intention being that even if `A` was implemented with no knowledge of `B`, `B` can be implemented such that it is still compatible with `A`, by implementing the reciprocal of the operator.

Method Overloading

- Lodge does not have method overloading in the typical sense currently

    - Recent improvements in how method signatures are defined has made me rethink this
- Type unions make it less clear what a functions signature actually is as a functions argument types could refer to any number of combination of concrete argument types.
    - It might be possible to allow overloading but restrict it to strictly non-intersecting interfaces, which would allow method overloading in some cases
    - Though, this might get frustrating as this intersection behavior would likely be complex and unintuitive
    - Though, this also makes candidacy for interface intersection much more complicated and may not be worth it.
- Instead, the way Lodge is currently designed is to use optional parameters and type switching to achieve similar behavior.
    - This is similar to Python's behavior and people don't seem to mind that much
- Look at 07 Functions to see about optional parameters and type switching to mimic the behavior of method overloading

01.02.02 Interfaces

Interfaces

An Interfaces is a set of fields, properties, and methods. All classes have an interface, which is made up of the public members (methods and fields) of that class. Interfaces exist separate from the implementation behind those members.

An interface in Lodge is much more than it is in other programming languages and it is used throughout Lodge even when not explicitly defined by the programmer.

A lodge program consists of many concrete types and many interfaces. Interfaces may be created manually, implicitly by defining a class, or implicitly when creating a type union.

When a variable or container is declared, it automatically has some interface. Any concrete type that has a compatible interface can be placed in that container. See Type Checking for more detail.

Interface Creation

There are several ways that interfaces are created in Lodge

1. Explicitly with an interface definition
    - Using the interface definition syntax
2. Implicitly by creating a class
    - Every class definition will also create an interface of the same name out of that class's public members.
3. Using a type union to combine existing interfaces through interface intersection
4. Using a type intersection to combine existing interfaces through interface unions

Behind the scenes, the members that can make up an interface are getters, setters, and methods. This is distinct from how a programmer defines an interface or class, but the separation between getters and setters fields is important for interface intersection, type unions, and properties.

When a class or interface definition specifies a public field, both a getter and setter will be created of that type in the interface. When either a getter or a setter is specified, only that getter or setter is created in the interface. Getters and setters of a given name can have the same or different types. That means that, in an interface definition, getters and setters of the same type will have the same result as a field definition. Fields and properties are identical for both external access and interface satisfaction (assuming the getter and setters have the same type).

Interface Definition Syntax

The syntax for defining an interface is much like the class definition syntax, but without any values, method/property implementations, or private members.

Here is an example:

```
interface InterfaceName {
        int fieldName

        fun FunctionName(int arg1, str arg2) int

        get propertyName int
        set propertyName int
}
```

01.02.03 Built-In Interfaces

The broadest possible interface. Every single object in lodge can be placed inside of a variable defined as object.

Type Interfaces

Every concrete type in Lodge automatically has an associated interface.

Int interface

Float Interface

Err Interface

```
interface Err {
        get Str Type
        get Str Message

        fun $() T
        !! ...
}
```

Object Interface

```
interface Object {
        fun repr() Str

        !! ...
}
```

None Interface

The None interface is completely empty. Its used in situations where a value might not exist.

```
interface None {}
```

01.02.04 Interface Compatibility

Interface compatibility describes candidacy for polymorphism in Lodge.

The concept of a compatible interface refers to the ability for an interface to be substituted for another without issue. In order for an interface to be compatible with another, that interface must be a superset of the other interface. If we have two interfaces A and B, B is compatible with A, if there are no methods or fields in A that are missing from B. In other words, if B would satisfy promising A, then B is compatible with A.

For example, the Int interface is not compatible with the Str interface because it fails to implement several of the methods that the string class like length and lowercase. In other words,

01.02.05 Automatic Interfacing

Automatic Interfacing is the process by which a class that doesn't explicitly promise an interface can still be used with that interface if it happens to implement. If, at compile time, a class's public interface is a superset of an interface or the public interface of another class, interface, type union, or type intersection, that class will implicitly promise that class/interface.

The result of automatic interfacing is that, objects can be placed into any container whose interface is a subset of that objects interface without needing to be explicitly declared. This is conveniently similar to duck typing while maintaining static type safety.

See 04 Interface Compatibility\

This section discusses Lodge's typing system.

01.03.01 Type Checking

Lodge is a type type-safe language, meaning that all type checks occur statically at compile time and that no operation can cause a type error at runtime. However, unlike other statically typed languages, Lodge's type system allows more type flexibility while still ensuring type safety.

This means that the Lodge type checker is somewhat complex to match the complexity of the type system.

01.03.02 Type Set Operations

Type Unions and Type Intersections

Type unions and type intersections are two ways of creating composite types out of multiple other types.

Type unions are the most common and allow the programmer to create a flexible variable that hold multiple different concrete types, while still being fully type checked.

Type intersections allow the programmer to create a variable that can only contain values that satisfy multiple interfaces. This use case is less common, but allows variables to be entirely

This behavior is represented in the syntax of type unions and intersections:

- `(type1 | type2)` is reminiscent of the logical or, indicating that a type compatible with type1 or type2 would also be compatible with the type union.
- `(type1 & type2)` is likewise reminiscent of logical and, indicating that a type must be compatible wit both type1 and type2.

01.03.03 Type Unions

Type unions are a fundamental component of Lodge's typing system. A type union consists of one or more types that are combined into a single type that allows for the inclusion of any type. The interface for a given type union is the intersection of the public interface for all of the types in the union. Any value compatible with the intersected interface of the type un ion may be placed in any variable or argument whose type is the type of the union.

The only types that cannot be included in type unions are Concrete Variables.

Defining type unions

A type union can be defined by separating several types with the `|` (pipe) character. This is a special syntax for type unions and **not** an example of the `|` operator operating on type objects.

```
(int | string) variable := ...    !! Declaring a variable with an unnamed union
```

An unnamed type unions refers to all cases where the type union is not bound to a name such as the case below where the variable is being declared to have a type which is the union if int and string

It is also possible in lodge to bind a type union to a name using the `as` keyword. After defining a name for a type union, that name can be used as shorthand to refer to the union after the name definition statement and in all enclosed scopes. The name can be used in any case where the type union could have been used including variable declaration, type switching, and even the definition of further type unions.

```
(int | string) as intstr

intstr variable := ...
```

The Interface of a Type Union

The interface of a type union is the interface intersection of the interfaces of all of the types that make up the union.

01.03.04 Errable and Nonable Types

Errable and Nonable types simply refer to any type union that contains `Err` or `None` respectively. These use cases are so common that they have their own special syntax for declaration.

Errable

Types with the `!` token are unioned with `Err`

```
(Err | Str)

!! Is the same as

!Str
```

Nonable

Types with the `?` token are unioned with `None`

```
(None | Str)

!! Is the same as

?Str
```

01.03.05 Type Switching

For types inside a type union, it may sometimes be useful to treat different types differently. To achieve this while maintaining type safety, type switches can be used. The difference between a type switch and a simple if-statement is that, within the switch, the variable will be treated as that type for the purposes of type checking. Example of a basic type switch.

Inside each of those blocks, the programmer can reference the original variable as if it were the more specific type without breaking type safety.

```
swype variable {
        List : { print(length(variable)) }
        Int  : { print(variable) }
        *    : { !- If there were other possible cases, the * captures all of them -! }
}
```

Switching Concrete Types

By default, type switching simply tests if the object being switched is compatible with the interface.

Switching multiple types at once

This is particularly useful for imitating the behavior of Method Overloading

```
(int | str) a := 10
(int | str) b := "20"

swype a, b {
        int, int : { !- Code -! }
        str, str : { !- Code -! }
        str, *   : { !- Code -! }
        *, int   : { !- Code -! }
}
```

01.03.06 Type Intersections

A type intersection is the opposite of a type union.
It is an important part of promising an interface, as the required public interface of a class that promises multiple interfaces is equivalent to the type intersection of those interface types.

Syntax

`(type1 & type2)`

This form is most useful for combining component interfaces and requiring objects to promise several interfaces for compatibility with type intersections. For example, it might be the case that you want to require that a class promises an Iterable interface and also serializable with something like this: `(Iterable & Serializable)`

Thus, Lodge code is able to be fully agnostic to the concrete type of an object while still mandating the behavior it needs from the object.

Interface of a Type Intersection

The interface of a type intersection is the interface union of the participating types.

01.03.07 Interface Set Operations

Interface Intersection and Interface Union

These operations differ somewhat from strict definitions of mathematical set operations as the equivalence of interface members is depended on both the name and types of the members. Additionally, the merging of two members is more complex than simply checking for their presence. See the pages on interface intersection and interface union for more information on this behavior.

The naming of type type unions and type intersections might seem counter-intuitive as the result of a type union is an interface intersection and vice versa. However, this naming is sensible when you consider the set of compatible types for a give type union/intersection:

- When you create a type union out of several types, the set of interfaces that are compatible with that type union is the mathematical union of the sets of types that are compatible with the types that participate in the union.
- When you create a type intersection out of several types, the set of interfaces that are compatible with that type intersection is the mathematical intersection of the sets of types that are compatible with the types that participate in the intersection.

01.03.08 Interface Intersection

Interface intersection is the process by which multiple interfaces are combined into one for the purposes of type unioning.

When several types exist in a union together, that type union will have its own interface that is the intersection of the public interfaces of all participating types.

It is worth noting that not every intersection is possible. Since any setters or method arguments require their types to be intersected, there is the chance that a namespace collision results in that type intersection being impossible.

Process of interface intersection

Each interface contains a set of members that are either getters, setters, or methods with arguments.

During interface intersection, each member that has a corresponding member in every interface is a candidate for being included in the intersection. Candidate members do not need to have the same type, but they do need to be the same sort of member (getter / setter / method).

Notably, any member that doesn't have a matching name in all interfaces will not be included in the intersection at all.

Then, each candidate is evaluated to see how it can be merged together to form a member of the final intersected interface.

How this merging works depends on what kind of field, but the result of this merging is either an update of the member's set of types or a removal of the member from candidacy entirely.

Merging Candidate Getters

Any code dealing with the intersection of several getters must be able to handle the types from any of the getters matching the candidate. Thus, the type of the member in the final intersection is simply the type union made up of all of the members matching this candidate.

For example, when merging the two following interfaces:

```
interface A {
	get val int
}

interface B {
	get val string
}
```

The result of `(A | B) as C` would be the following:

```
interface C {
	get val (int | string)
}
```

In this case, when accessing the `val` getter from a variable of type C, the object stored behind the interface C could be either a B or an A, and `val` could either be an int or a string. This means that code that deals with C has to be able to handle the type of `val` regardless of what it is.

In the case where the type union of the getter types results in an empty interface, that getter will be removed from the union. If that getter refers to a field in the class, that field becomes, write-only.

Merging Candidate Setters

The type of a setter in the intersected interface must only be able to accept types that **all** setters matching the candidate can accept. Thus, the new type of the setter will be the type intersection (not interface intersection).

For example, when merging the two following interfaces:

```
interface A {
	set val int
}

interface B {
	set val string
}
```

The result of `(A | B) as C` would be the following:

```
interface C {
	get val (int & string)
}
```

Merging Candidate Methods

This section needs to be expanded

For merging methods, candidacy is still determined by the name of the method, arguments are treated like setters, and return values are treated like getters.

For simplicity, method arguments are merged position-wise and their names are not considered at all.
However, if two methods have matching argument names that are not positionally aligned, a compiler warning will

This gets more complex when talking about optional parameters and variadic functions.

01.03.09 Interface Union

A union between multiple interfaces is the result of a 06 Type Intersections.

Interface unions are somewhat more complex than interface intersections, as there are cases where the union will fail due to incompatibility between

Process of interface union

Read 08 Interface Intersection first as that page has more details and examples of this process.

To start with, all members from every participant are candidates for inclusion in the union

However, there may be some elements with shared names, that require special consideration for their types. This is also the way in which some interface unions may fail.

Getters with shared names

An object that is compatible with the union of multiple interfaces must be able to properly behave like any of the of the types in the interface union. For example, consider the type intersection `(A & B)`. Any class that would be compatible with this interface union must be compatible with both A and B in every case. Thus, if both A and B have a getter of the same name, those two getters must be mutually compatible (A is compatible with B and B is compatible with A). Mutual compatibility requires that the two interfaces are identical.

If there are getters with shared names and all getters do not have the **same exact interface**, the interface union will **fail with a compiler error**.

For example, consider merging the two following interfaces:

```
interface A {
	set val int
}

interface B {
	set val string
}
```

The line `(A & B)` would result in a compiler error.

The failure of union is a non-ideal outcome, thus it is best practice in lodge to avoid constructing classes that are likely to result in failures when unioned with interfaces that they are likely to be unioned with.

- Reducing the public interfaces of classes as much as possible
- Using distinctive names for members to reduce collisions

Setters with shared names

Setters with shared names are somewhat simpler as the types of each setter is simply the type union of all the candidate setters.

Much like in interface intersection, when the interface of a member becomes empty as the result of type unioning, that member is removed from the resulting interface.

For example, when merging the two following interfaces:

```
interface A {
        get val int
}

interface B {
        get val string
}
```

The result of `(A & B) as C` would be the following:

```
interface C {
        get val (int | string)
}
```

Methods with shared names

The behavior of method merging is somewhat complex and I haven't worked it out all the way yet.

01.03.10 Types as Objects

There are several situations where treating classes as objects might be useful.

It can be useful in dependency injection where a given object still maintains the responsibility of instancing an object, but it shouldn't have to worry about what the specific class of that object is.

Additionally, it is useful in the definition of the Type Conversion Operator operator, which should be able to take a class as an argument.

Type of a Type object

This section needs to be expanded

From the perspective of interaction, types look like functions that are called to create instances, but also have static fields.

This makes type objects complex to put into the rest of the type system.