Kira Language Specifications

Version 1.0

November 08, 2025

Contents

1 Hello World!

```
@trace("Hello World!")
```

2 FizzBuzz

```
for i in 0..101 {
    if i % 15 == 0 {
        @trace("FizzBuzz")
    } else if i % 3 == 0 {
        @trace("Fizz")
    } else if i % 5 == 0 {
        @trace("Buzz")
    } else {
        @trace(i)
    }
}
```

3 Comments

Kira supports comments via `//`. Anything after will be stripped out by the preprocessor.

```
// I won't appear in the generated output!
```

4 Variables

Kira is a statically-typed language meaning that all variables must declare a type.

```
name: Str = "John"
age: Int32 = 34
isOk: Bool = true
```

5 Control Flow Structures

These structures help determine which pieces of your code will run. Kira has the following control structures:

1. `if-else` selection statements
2. `while` loops
3. `do-while` loops
4. `for` loops

```
// If-Else Selection Statement
if name == "John" {
    @trace("Hi John")
} else if name == "Jamie" {
    @trace("Hi Jamie")
} else {
    @trace("Who are you?")
}

// While Loop
while true {
    @trace("Looping forever...")
}

// Do-While Loop
do {
    @trace("I will be printed once!")
} while false

// For Loop
for i: Int32 in 1..10 {
    @trace(i)
}
```

6 Functions

Kira treats functions as first-class citizens, meaning you can use them as values and pass them around.

To return a value, Kira uses the `return` keyword. If a function has no return, it must specify its return type as `Void` or `Never`.

```
fx sumOf(a: Int32, b: Int32 = 3): Int32 {
    return a + b
}
```

Function parameters can either be specified using their name or using position.

7 Module System

Kira uses a module system for you to organize your source structure. Each piece of Kira source file starts off with a module declaration:

```
module "author:module_name.submodule_name"
```

- `author` represents the organization or individual that this belongs to.
- `module_name` represents what project this source file falls under.
- `submodule_name` represents this individual source file's name

7.1 Using a submodule

Often times, you would include or import submodules, not entire modules themselves per submodule. This is to save on compile time and also potential bloating. To include another submodule:

```
use "author:module_name.submodule_name"

use "author:module_name" // makes all submodules available to the
    current context
```

For example, to use all builtin types from Kira (e.g. `Int32`, `Str`), you need to use the submodule `kira:lib.types`.

8 Classes

Kira supports classes, an object-oriented principle. Class in Kira are very easy and malleable to understand and lightweight; however, they do have some limitations:

1. No multi-inheritance (Diamond Inheritance Problem)
2. Only one default constructor
3. No nested structures (classes, enums, namespaces)
4. No companion or static members
5. All fields that are virtual can be provided through the constructor.
6. No direct abstract classes or interfaces

```
pub class Vector2
{
    require pub mut x: Float32
    require pub mut y: Float32

    pub fx dot(other: Vector2): Float32 {
        return (other.x * x) + (other.y * y)
    }

    pub mut fx toStr(): Str {
        return "( ${x}, ${y} )"
    }
}

a: Vector2 = Vector2 { 3, 3 }
b: Vector2 = Vector2 { 3, 3, toStr = fx() { return "< ${x}, ${y} >" } }
@trace(a.dot(b))
```

8.1 Inheritance

Multi-inheritance is not allowed, but to share common functions across multiple classes, Kira supports traits

Inheritance is very simple, there are only several types of allowed patterns:

1. **Concrete classes**
2. **(Semi-)Abstract classes**
3. **Interface-Like classes**

However, all of these utilize the format of classes meaning that you can only use ONE of these even if it is Interface-Like.

Here are some examples of the previously mentioned patterns:

8.1.1 Concrete Classes (Normal Classes)

All members are implemented, with the only exception being property fields.

```
pub class Student {
    require pub name: Str
    require pub mut gpa: Float32

    pub fx passing(): Bool {
        return gpa > 2.0
    }
}
```

8.1.2 (Semi-)Abstract Classes

Semi Abstract classes are created where you add unimplemented member function (methods) into the mix of concrete classes. However, they are "semi" abstract because Kira allows anonymous classes to be made everywhere by passing functions directly to the constructor.

```
pub class Human {
    pub scientificName: Str = "Homo Sapien"

    pub fx speak(): Void

    pub fx walk(): Void {
        @trace("Walking...")
    }
}
```

8.1.3 Interface-Like Classes

This pattern is the most redundant and should be avoided. Instead, prefer to use traits if you need to share common functionalities across multiple classes. In general, interface like classes define no property members and only abstract function members:

```
pub class Animalia {
    pub fx reproduce(): Animalia

    pub fx die(): Void

    pub fx eat(): Void

    pub fx isAlive(): Bool
}
```

9 Immutability By Default

Everything in Kira is immutable or closed by default. This means variables cannot be reassigned/mutated, classes cannot be inherited from, methods in classes cannot be overridden by default.

In order to allow for mutability, specify the `mut` modifier before the element you want to make mutable.

10 Visibility Modifiers

There are only 2 visibility levels allowed:

1. public - `pub`
2. internal - implicit (i.e. no keyword used)

10.1 Public Modifier

The `pub` modifier specifies that anything outside can look inside. For example, a submodule which has a class:

```
class A {
}

pub class B {
}
```

An external submodule that uses this cannot see `class A`, but can see `class B`. Additionally, `class B` can see `class A` and vice versa.

Within classes themselves, the modifier only serves as encapsulation purposes (i.e. hiding data and fields). If a member of a class does not have the `pub` modifier, that field can only be access through the setter during construction or within a neighboring method that can expose it (getter). Additionally, if the class is inheritable, it means that field is also private from the child and the child cannot access it.

Why no `protected`?

In other languages like Java, the `protected` keyword is another visibility layer that allows for only the members of the class and children of that class to view that field.
Kira does not utilize this layer simply because it is extra overhead and increases the learning curve. Creating a binary system where it is either the field is visible or not makes it not only easier, but gives more freedom to the programmer in designing their APIs.

11 Null Safety

Kira incorporates sound null-safety as one of its core pillars. It does so using a core type: `Maybe<A>`.

Important

Kira does not have a special `null` literal token in the language grammar. Instead, the core library exposes a global singleton value named `null` that represents an absent value (the canonical `Maybe` sentinel). The lexer/parser treat `null` as an ordinary identifier; the standard library provides the `null` binding and the semantic analyzer resolves it as the Maybe sentinel.

Without boxing a type in `Maybe`, you are not allowed to assign the `null` sentinel to anything of a non-`Maybe` type:

```
a: Int32 = null // error: cannot assign the ‘null‘ sentinel to a non-
    Maybe type

b: Maybe<Int32> = null // ok: ‘null‘ is the Maybe sentinel and can be
    assigned to Maybe<T>
```

`Maybe` also enables for field-valuation meaning you can achieve the following without using a getter for the internally held value:

```
mut a: Maybe<Int32> = null

@trace(a) // null (the sentinel)
@trace(a.value) // null

a = 32 // assignment to Maybe<T> will auto-box a value of type T into
    the Maybe<T>

@trace(a) // 32
@trace(a.value) // 32
```

11.1 Null Safety Operators

Null Safe Get Instance form:

```
// preferred instance-style helper
value: Int32 = a.unwrapOr(0)
```

Also, available as a helper function (static-style) for convenience:

```
value: Int32 = Maybe.unwrapOr(a, 0)
```

If the underlying value is the `null` sentinel, `unwrapOr` returns the provided `option` value, otherwise it returns the underlying value.

```
mut a: Maybe<Int32> = null

if a == null {
    @trace("Null")
} else {
    @trace(a)
}

// is akin to:

@trace(a.unwrapOr(0))
```

Sanity Checks If you do not like using == to check if the value is equal to the `null` sentinel, there are 2 methods to help you:

1. `Maybe<A>::isNull(): Bool` - returns `true` if the underlying value is the `null` sentinel else `false`
2. `Maybe<A>::isSome(): Bool` - returns `true` if the underlying value is not the `null` sentinel else `false`

12 Exceptions

Kira only has unchecked exceptions meaning that you do not need to catch everything. At the language core exceptions are represented as `Str` values (string messages):

```
throw "Something went wrong."
```

12.1 Exception Handling

Exceptions can be caught by using a `try-on` expression:

```
try {
    // ...
} on e: Str {
    @trace(e)
}
```

Note

The variable required after `on` is always guaranteed to be of `Str`.

If you need richer error information, prefer the ergonomic `Result<A, Str>` type (see `Ref` / `Result` in Reference Types) to encode error payloads and avoid relying solely on `throw`/`try-on` for control flow.

13 Memory Model

Kira uses an Automatic Reference Counting (ARC) memory model similar to Swift. Every object has a reference counter and when this counter drops to zero, the object is freed from memory. However, in certain scenarios like a circular reference where A and B reference each other, it can cause them to never be deallocated leading to undefined behavior and/or memory faults.

To mitigate this, Kira has 2 types that are intrinsified by the compiler:

1. `Weak<T>` - Used to denote a non-strong reference to mitigate circular references and allow for deallocation. Reading a `Weak<T>` requires an explicit upgrade/unwrap operation (for example `w.upgrade()` or `w.value`) which returns a `Maybe<T>`; if the target was already deallocated the upgrade returns `null`.
2. `Unsafe<T>` - Signifies a raw, non-refcounted reference. Dereferencing `Unsafe<T>` has no lifetime checks and may dangle; it is intended only for performance critical interop code.

14 Finalizers & Initializers Blocks

These are special functions that are only callable by the runtime and not from Kira itself. They are meant to be executed right at the time of object allocation and deallocation and are only permitted within classes.

14.1 Initializers

Initializers are ran right at the moment of object construction. It is accomplished by introducing the `initially` block:

```
class Person {
    require pub age: Int32

    initially {
        if age < 0 {
            @trace("Age must be >= 0!")
        }
    }
}
```

14.2 Finalizers

Finalizers are used to clean up external resources from things like File IO. It is accomplished by introducing the `finally` block:

```
class FileIO {

    finally {
        doCleanup()
    }
}
```

All code in the finally block is called and ran when the object is about to be deallocated.

15 Core Types

Core types are intrinsified by the compiler: they are recognized specially and the compiler may optimize for them. Kira treats all values as objects (no boxing distinction), but these core types form the language's primitives and commonly used reference helpers.

15.1 Numeric types

Integer types

1. `Int8` - 8-bit signed integer
2. `Int16` - 16-bit signed integer
3. `Int32` - 32-bit signed integer
4. `Int64` - 64-bit signed integer

Floating point types

1. `Float32` - 32-bit floating point
2. `Float64` - 64-bit floating point

15.2 Logical

1. `Bool` - Boolean type (`true` / `false`) backed by an 8-bit representation.

15.3 Unit / control types

1. `Void` - Indicates no return value.
2. `Never` - Indicates a function never returns (used to mark diverging code paths).

15.4 Reference / core helpers

These types are used for reference behavior, type representation, and core runtime helpers.

1. `Any` - Root of all classes; universal supertype for dynamic values.
2. `Type` - Runtime representation of a type.
3. `Ref<A>` - A boxed reference to an object of type `A`.
4. `Weak<T>` - Weak reference: non-owning reference that can be upgraded to `Maybe<T>` and may be `null` if target is deallocated.
5. `Unsafe<T>` - Raw, non-refcounted pointer-like reference (no lifetime checks; may dangle).
6. `Maybe<A>` - Nullability wrapper: use to allow `null` assignments and safe unwrap semantics.
7. `Result<A,B>` - Result type encoding success (`A`) or error (`B`), useful as an alternative to exceptions.
8. `Val<A>` - Trait indicating `A` can be referenced directly; used by `Maybe` and other helpers.
9. `Module` - Module metadata value, used to refer to module structures at runtime.
10. `Fx<P: Tuple, R>` - Function type representation where `P` is a `Tuple` of parameter types and `R` the return type (see "Variadic Generics & TupleN").

15.5 Sequence & collection types

1. `Str` - String / character sequence.
2. `Arr<A>` - Static immutable array structure (compiler-optimized for fixed arrays).
3. `List<A>` - Dynamic, mutable array structure (standard library implementation).
4. `Map<K,V>` - Dynamic hashmap (standard library implementation).
5. `Set<A>` - Dynamic hash set (standard library implementation).

16 Generics (Type Bounds)

Generics are a feature to allow for generalization of a feature/class towards other types while maintaining type safety. In Kira, they are implemented as compile-time features with runtime reification.

The syntax follows a very common format akin to C++, Java, and Kotlin:

```
class Pair<A, B> {
   require pub first: A
   require pub second: B
}

// Deprecated variadic-style example (old proposal)
// class Functor<A, [B]> {
//    require pub returnValue: A
//    require pub parameters: List<B>
// }

// Recommended tuple-based replacement:
class Functor<A, P: Tuple> {
   require pub returnValue: A
   require pub parameters: P // use a TupleN type for parameter lists,
       e.g. Tuple2<Int32, Str>
}
```

17 Variadic Generics & TupleN (design)

17.1 Problem summary

Variadic generics (e.g. `class Fx<A, [B]>`) aim to allow a trailing, variable-length list of type parameters. While convenient, treating the trailing variadic parameter as a raw list at the language level is ambiguous at source-level (is it a list/array? how are bounds applied?) and forces compiler magic to implement safely. This makes the feature hard to inspect, extend, and reason about for library authors and tooling.

17.2 Design overview

Kira resolves the ambiguity by modeling variadic parameter lists as tuples at the source level. The core library provides a small family of `TupleN` concrete classes (for example `Tuple1` .. `Tuple10`) and a minimal `Tuple` interface that they implement. The compiler and standard library use these `TupleN` types to represent variable parameter lists. This keeps the feature a source-level construct (no hidden compiler-only expansion required for everyday usage) while still enabling variadic-like behavior.

17.3 Core interface (source-level)

```
// minimal Tuple interface (core library / intrinsified by compiler for
    ergonomics)
pub @magic mut class Tuple {
    require pub fx size(): Int32
    require pub fx @get(index: Int): Any // accessor returns Any - see
       type-safety notes
}

// example concrete tuple with two elements
pub @magic Tuple2<A, B>: Tuple() {
    require pub first: A
    require pub second: B

    override pub fx size(): Int32 { return 2 }

    override pub fx @get(index: Int): Any {
        if index == 0 { return first }
        else if index == 1 { return second }
        throw "IndexNotFound: ${index}"
    }
}
```

17.4 Function types and Fx

Function types use a `Tuple` as their parameter bundle. For example:

```
pub @magic class Fx<P: Tuple, R> {
    pub fx (params: P): R
}
```

This lets a function type precisely declare the number and types of its parameters using `TupleN`:

```
// a function taking (Int32, Str) returning Bool
f: Fx<Tuple2<Int32, Str>, Bool>

// or explicitly as a function literal type in signatures
pub fx callMe(params: Tuple2<Int32, Str>): Bool { ... }
```

17.5 Usage

```
myTuple: Tuple2<Int32, Str> = Tuple2<Int32, Str> { 69, "Hello" }
for i in 0..(myTuple.size() - 1) {
    @trace(myTuple[i])
}
```

17.6 Extensibility and compiler behavior

- The core library ships `Tuple1` .. `Tuple10` as the standard set. These are implemented in source and are visible to users.
- Libraries may extend the `Tuple` interface to support larger tuples (e.g. `Tuple11`) without changing the compiler. Doing so is a normal library extension and remains source-visible.
- The compiler performs bound checking when matching a `TupleN` usage to a generic parameter constrained with `P: Tuple`.
- Tools (formatters, linters, IDEs) can inspect `TupleN` declarations directly; no hidden template magic is required.

17.7 Type-safety & ergonomics

- The `@get` accessor returns `Any` because the elements of a tuple may have heterogeneous types. Callers are expected to use `is`/`as` to narrow and safely cast where necessary.
- For common patterns, the standard library should provide typed helpers where appropriate. Examples:

  - `fn getAs<T>(tuple: Tuple, index: Int): Maybe<T>` — a generic helper that attempts to cast and returns `null` if the cast fails.
  - Small macros or intrinsics (editor/compile-time) can generate typed accessors for specific `TupleN` definitions.

17.8 Migration from previous variadic proposal

- Previously used variadic syntax (for example `class Fx<A, [B]>`) is deprecated in favor of the `TupleN` approach.
- Where the old syntax was used only to represent a function parameter list, replace it with `TupleN` bindings (for example `Fx<Tuple2<A,B>, R>` or `fx(params: Tuple2<A,B>): R`).

17.9 Rationale summary

- **Source-level visibility:** `TupleN` types live in the core library and are readable by developers and tooling.
- **Extensible without compiler changes:** users and libraries can extend `Tuple` to add support for larger tuples if needed.
- **Clear semantics:** tuples give a concrete representation for a fixed-length heterogenous sequence; they avoid the ambiguity of a "variadic list" type at the language surface.

18 Traits / Compile Time "Mixins"

Kira does not support multi-inheritance as previously seen; however, in order to suffice for allowing sharing common components across classes, **traits** are a good alternative.

Traits allow injecting a class with functions/methods at compile time directly. Additionally, it can also serve as a way to inject abstract methods or no-implementation functions that the target class must take care of.

However, traits differ from Mixins and Interfaces in that they are an entirely compile-time feature. This means that you cannot perform runtime checks for if a type has a trait to it (however, this can be implemented by directly checking if a certain method/function exists within).

Within Kira, you are only allowed to define functions within traits, and each function also implicitly points to the current instance like in classes (i.e. there is no `self` or `this` or `::` operands to access the current scope).

Mutability Note

All functions in a trait are mutable or overrideable, specifying the `mut` modifier will have no effect.

Visibility Note

You are able to enforce visibility modifiers on the trait itself and also functions. This is done by using the normal modifiers. However, when a function is marked with or without a modifier, it is not able to be changed by the implementing type.

Implementer Features

The implementer class is allowed to use specific functions in order ot refer

Implementing a trait:

```
trait Animal {
    fx makeNoise(): Void

    pub fx canConsume(items: Arr<Str>): Bool {
        return items.any([ "water", "air" ])
    }
}

class Dog: Animal {
    fx makeNoise(): Void {
        @trace("Woof")
    }

    pub fx canConsume(items: Arr<Str>): Bool {
        return
    }
}
```

19 Compile-Time Intrinsics

Kira supports compiler-integrated intrinsics for compile-time execution. These are not user-definable and are designed to simplify expressions, enable metaprogramming, and support DSL construction.

Properties:

- Prefixed with `@`
- Treated as standard identifiers
- Function-like or directive-like
- Executed during any compiler phase

```
a: Map<Str, Any> = @json_decode(`
  {
    "hello": 1,
    "world": 2
  }
`)

@trace(a["hello"]) // Outputs 1 to debugger
```

19.1 Standard Intrinsics

Table 1: Standard Intrinsics

`Intrinsic`	**Description**	**Example Usage**
`@trace(...)`	Outputs values to the debugger console at runtime.	
`@global`	Elevates a type, function, or variable to global scope.	
`@json_decode(...)`	Parses a JSON string at compile-time into a `Map<Str, Any>`.	
`@type_of(...)`	Returns the runtime `Type` representation of a value or type.	

20 Identifiers and naming

Kira prefers a consistent naming style to improve readability and to leave a syntactic distinction available for compiler-level constructs.

- Preferred styles for ordinary user-defined names (variables, functions, types, fields) are camelCase and PascalCase:
  - Variables and function names: camelCase (example: `myVariable`, `computeSum`).
  - Type and class names: PascalCase (example: `Int32`, `MyStruct`).
- Underscores (`_`) are explicitly disallowed inside ordinary identifiers (variables, functions, types, fields, enum members, etc.). The lexer will reject identifiers that contain underscores and will raise a diagnostic. This rule helps:
  - Reserve the underscore character for special/ambient language uses (see intrinsics below).
  - Reduce ambiguity and enforce a single, consistent naming style across codebases.
- Exceptions:
  - Module and file names may contain underscores. Those names are not subject to the same identifier restrictions as in-source identifiers.
  - Intrinsics are a language-level, compile-time feature and are lexed with the `@` prefix (for example `@trace_one`). Intrinsic names may include underscores (snake_case) to make them visually distinct from ordinary identifiers. Intrinsic tokens are lexed as a single `INTRINSIC_IDENTIFIER` token by the lexer.

Rationale Disallowing underscores for ordinary identifiers helps maintain a clear visual distinction between normal source-level names and compiler-level, intrinsic features. Intrinsics are intended to feel different from regular API or language constructs; allowing them to use snake_case (and the `@` prefix) keeps that separation obvious both in source and in compiled metadata.