Bamboo Programming Language

Table of Contents

Introduction

The Bamboo Programming Language is a new high-level programming language intended for smart contract development.

The language's goals are, in order of importance:

- *Safety and security*: Focus on safety, e.g. by providing a strong static type system, design by contract, and linear types; and security, by providing a capability system, and design by contract.
- *Auditability*: Focus on readability: make it easy to verify what the code is doing, and make intentions explicit, at a small cost of verbosity.
- *Simplicity*: Focus on developer productivity and usability: make it easy to write code, provide good tooling.

Syntax and Behavior

The programming language's syntax and behavior is inspired by Kotlin, Swift, Rust, TypeScript, and Solidity.

Comments

Comments can be used to document code. A comment is text that is not executed.

Single-line comments start with two slashes (`//`):

```
// This is a comment on a single line.
// Another comment line that is not executed.
```

Multi-line comments start with a slash and an asterisk (`/*`) and end with an asterisk and a slash (`*/`):

```
/* This is a comment which
spans multiple lines. */
```

Comments may be nested.

```
/* /* this */ is a valid comment */
```

Constants and Variable Declarations

Constants and variables are declarations that bind a value to a name. Constants can only be initialized with a value and cannot be reassigned afterwards. Variables can be initialized with a value and can be reassigned later. Declarations are valid in any scope, including the global scope.

Constant means that the *name* is constant, not the *value* – the value may still be changed if it allows it, i.e. is mutable.

Constants are declared using the `let` keyword. Variables are declared using the `var` keyword. The keywords are followed by the name, an optional type annotation, an equals sign `=`, and the initial value.

```
// Declare a constant named `a`
//
let a = 1

// Invalid: re-assigning to a constant
//
a = 2

// Declare a variable named `b`
//
var b = 3

// Assign a new value to the variable named `b`
//
b = 4
```

Variables and constants **must** be initialized.

```
// Invalid: the constant has no initial value
//
let a
```

Once a constant or variable is declared, it cannot be redeclared with the same name, with a different type, or changed into the corresponding other kind (variable to a constant and vice versa).

```
// Declare a constant named `a`
//
let a = 1

// Invalid: cannot re-declare a constant with name `a`,
// as it is already used in this scope
//
let a = 2

// Declare a variable named `b`
//
var b = 3
```

```
// Invalid: cannot re-declare a variable with name `b`,
// as it is already used in this scope
//
var b = 4

// Invalid: cannot declare a variable with the name `a`,
// as it is already used in this scope,
// and it is declared as a constant
//
var a = 5
```

Type Annotations

When declaring a constant or variable, an optional *type annotation* can be provided, to make it explicit what type the declaration has.

If no type annotation is provided, the type of the declaration is inferred from the initial value.

```
// Declare a variable named `initialized` which has an explicit type annotation.
//
// `Bool` is the type of booleans
//
var initialized: Bool = false

// Declare a constant named `inferred`, which has no type annotation
// and for which the type is inferred to be `Int`,
// based on the initial value
//
let inferred = 1
```

If a type annotation is provided, the initial value must be of this type, and new values assigned to variables must match the declaration's type. This type safety is explained in more detail in a separate section.

```
// Invalid: declare a variable with an explicit type `Bool`,
// but the initial value has type `Int`.
//
// `Int` is the type of arbitrary-precision integers
//
let booleanConstant: Bool = 1

// Declare a variable that has type `Bool`
//
var booleanVariable = false

// Invalid: assign a value with type `Int` to a variable which has type `Bool`
//
booleanVariable = 1
```

Naming

Names may start with any upper and lowercase letter (A-Z, a-z) or an underscore (`_`). This may be followed by zero or more upper and lower case letters, underscores, and numbers (0-9). Names may not begin with a number.

```
// Valid, title-case
//
PersonID

// Valid, with underscore
//
token_name

// Valid, leading underscore
//
_balance

// Valid, with number
//
account2

// Invalid, leading number
//
1something

// Invalid, various invalid characters
//
!@#$%^&*
```

Conventions

By convention, variables, constants, and functions have lowercase names; and types have title-case names.

Semicolons

Semicolons (;) are used as statement separators. Semicolons can be placed after any statement, but can be omitted if only one statemen appears on the line. Semicolons must be used to separate multiple statements if they appear on the same line.

```
// Declare a constant, without a semicolon
//
let a = 1

// Declare a variable, with a semicolon
//
var b = 2;

// Declare a constant and a variable on a single line, separated by semicolons
//
let d = 1; var e = 2
```

Values and Types

Values are objects, like for example booleans, integers, or arrays. Values are typed.

Booleans

The two boolean values `true` and `false` have the type `Bool`.

Numeric Literals

Numbers can be written in various bases. Numbers are assumed to be decimal by default. Non-decimal literals have a specific prefix.

Numeral system	Prefix	Characters
Decimal	*None*	one or more numbers (`0` to `9`)
Binary	`0b`	one or more zeros or ones (`0` or `1`)
Octal	`0o`	one or more numbers in the range `0` to `7`
Hexadecimal	`0x`	one or more numbers, or characters `a` to `f`, lowercase or uppercase

```
// A decimal number
//
1234567890

// A binary number
//
0b101010

// An octal number
//
0o12345670

// A hexadecimal number
//
0x1234567890ABCDEFabcdef

// Invalid: unsupported prefix 0z
//
0z0
```

Decimal numbers may contain underscores (`_`) to logically separate components.

```
let largeNumber = 1_000_000
```

Underscores are allowed for all numeral systems.

```
let binaryNumber = 0b10_11_01
```

Integers

Integers are whole numbers without a fractional part. They are either *signed* (positive, zero, or negative) or *unsigned* (positive or zero) an are either 8 bits, 16 bits, 32 bits, 64 bits or arbitrarily large.

The names for the integer types follow this naming convention: Signed integer types have an `Int` prefix, unsigned integer types have a `UInt` prefix, i.e., the integer types are named `Int8`, `Int16`, `Int32`, `Int64`, `UInt8`, `UInt16`, `UInt32`, and `UInt64`.

- **`Int8`**: -128 through 127
- **`Int16`**: -32768 through 32767
- **`Int32`**: -2147483648 through 2147483647
- **`Int64`**: -9223372036854775808 through 9223372036854775807
- **`UInt16`**: 0 through 65535
- **`UInt32`**: 0 through 4294967295
- **`UInt64`**: 0 through 18446744073709551615

```
// Declare a constant that has type `UInt8` and the value 10
let smallNumber: UInt8 = 10
```

```
// Invalid: negative literal cannot be used as an unsigned integer
//
let invalidNumber: UInt8 = -10
```

In addition, the arbitrary precision integer type `Int` is provided.

```
let veryLargeNumber: Int = 10000000000000000000000000000000
```

Negative integers are encoded in two's complement representation.

Floating-Point Numbers

There is no support for floating point numbers.

Contracts are not intended to work with values with error margins and therefore floating point arithmetic is not appropriate here. Fixed point numbers should be simulated using integers and a scale factor for now.

Addresses

The type `Address` represents an address. Addresses are unsigned integers with a size of 160 bits. Hexadecimal integer literals can be used to create address values.

```
// Declare a constant that has type `Address`
//
let someAddress: Address = 0x06012c8cf97bead5deae237070f9587f8e7a266d

// Invalid: Initial value is not compatible with type `Address`,
// it is not a number
//
let notAnAddress: Address = ""

// Invalid: Initial value is not compatible with type `Address`,
```

```
// it is a number, but larger than 160 bits
//
let alsoNotAnAddress: Address = 0x06012c8cf97bead5deae237070f9587f8e7a266d123456789
```

Integer literals are not inferred to be an address.

```
// Declare a number. Even though it happens to be a valid address,
// it is not inferred as it.
//
let aNumber = 0x06012c8cf97bead5deae237070f9587f8e7a266d
// `aNumber` has type `Int`
```

Any

`Any` is the top type, i.e., all types are a subtype of it.

```
// Declare a variable that has the type `Any`.
// Any value can be assigned to it, for example an integer.
//
var someValue: Any = 1

// Assign a value with a different type, `Bool`
someValue = true
```

However, using `Any` does not opt-out of type checking. It is invalid to access fields and call functions on `Any` typed values, as it has no fields and functions.

```
// Declare a variable that has the type `Any`. The initial value is an integer,
// but the variable  still has the explicit type `Any`.
//
let a: Any = 1

// Invalid: Operator cannot be used for an `Any` value (`a`, left-hand side)
// and an `Int` value (`2`, right-hand side)
//
a + 2
```

Never

`Never` is the bottom type, i.e., it is a subtype of all types. There is no value that has type `Never`. `Never` can be used as the return type for functions that never return normally. For example, it is the return type of the function `panic`.

```
// Declare a function named `crashAndBurn` which will never return,
// because it calls the function named `panic`, which never returns
//
fun crashAndBurn(): Never {
    panic("An unrecoverable error occurred")
}
```

Optionals

> 🚧 Status: Optionals are not implemented yet.

Optionals are values which can represent the absence of a value. Optionals have two cases: either there is a value, or there is nothing.

An optional type is declared using the `?` suffix for another type. For example, `Int` is a non-optional integer, and `Int?` is an optional integer, i.e. either nothing, or an integer.

The value representing nothing is `nil`.

```
// Declare a constant which has an optional integer type,
// with nil as its initial value
//
let a: Int? = nil

// Declare a constant which has an optional integer type,
// with 42 as its initial value
//
let b: Int? = 42
```

Nil-Coalescing Operator

The nil-coalescing operator `??` returns the value inside an optional if it contains a value, or returns an alternative value if the optional has no value, i.e., the optional value is `nil`.

If the left-hand side is non-nil, the right-hand side is not evaluated.

```
// Declare a constant which has an optional integer type
//
let a: Int? = nil

// Declare a constant with a non-optional integer type,
// which is initialized to b if it is non-nil, or 42 otherwise
//
let b: Int = a ?? 42
// `b` is 42, as `a` is nil
```

The nil-coalescing operator can only be applied to values which have an optional type.

```
// Declare a constant with a non-optional integer type
//
let a = 1

// Invalid: nil-coalescing operator is applied to a value which has a non-optional type
// (a has the non-optional type `Int`)
//
let b = a ?? 2
```

```
// Invalid: nil-coalescing operator is applied to a value which has a non-optional type
// (the integer literal is of type `Int`)
//
let c = 1 ?? 2
```

The alternative value, i.e. the right-hand side of the operator, must be the non-optional or optional type matching the type of the left-hand side.

```
// Declare a constant with a non-optional integer type
//
let a = 1

// Invalid: nil-coalescing operator is applied to a value of type `Int`,
// but the alternative has type `Bool`
//
let b = a ?? false
```

Conditional Downcasting Operator

The conditional downcasting operator `as?` can be used to type cast a value to a type. The operator returns an optional. If the value has a type that is a subtype of the given type that should be casted to, the operator returns the value as the given type, otherwise the result is `nil`.

```
// Declare a constant named `something` which has type `Any`,
// with an initial value which has type `Int`
//
let something: Any = 1

// Conditionally downcast the value of `something` to `Int`.
// The cast succeeds, because the value has type `Int`
//
let number = something as? Int
// `number` is 1 and has type `Int?`

// Conditionally downcast the value of `something` to `Bool`.
// The cast fails, because the value has type `Int`,
// and `Bool` is not a subtype of `Int`
//
let boolean = something as? Bool
// `boolean` is nil and has type `Bool?`
```

Strings and Characters

> 🚧 Status: Characters are not implemented yet.

Strings are collections of characters. Strings have the type `String`, and characters have the type `Character`. Strings can be used to work with text in a Unicode-compliant way. Strings are immutable.

String and character literals are enclosed in double quotation marks (`"`).

```
let someString = "Hello, world!"
```

String literals may contain escape sequences. An escape sequence starts with a backslash (`\`):

- `\0` : Null character
- `\\` : Backslash
- `\t` : Horizontal tab
- `\n` : Line feed
- `\r` : Carriage return
- `\"` : Double quotation mark
- `\'` : Single quotation mark
- `\u` : A Unicode scalar value, written as `\u{x}`, where `x` is a 1–8 digit hexadecimal number which needs to be a valid Unicode scalar value, i.e., in the range 0 to 0xD7FF and 0xE000 to 0x10FFFF inclusive

```
// Declare a constant which contains two lines of text
// (separated by the line feed character `\n`), and ends
// with a thumbs up emoji, which has code point U+1F44D (0x1F44D)
//
let thumbsUpText =
    "This is the first line.\nThis is the second line with an emoji: \u{1F44D}"
```

The type `Character` represents a single, human-readable character. Characters are extended grapheme clusters, which consist of one c more Unicode scalars.

For example, the single character `ü` can be represented in several ways in Unicode. First, it can be represented by a single Unicode scalar value `ü` ("LATIN SMALL LETTER U WITH DIAERESIS", code point U+00FC). Second, the same single character can be representec by two Unicode scalar values: `u` ("LATIN SMALL LETTER U", code point U+0075), and "COMBINING DIAERESIS" (code point U+0308). The combining Unicode scalar value is applied to the scalar before it, which turns a `u` into a `ü`.

Still, both variants represent the same human-readable character `ü`.

```
let singleScalar: Character = "\u{FC}"
// `singleScalar` is `ü`
let twoScalars: Character = "\u{75}\u{308}"
```

```
// `twoScalars` is `ü`
```

Another example where multiple Unicode scalar values are rendered as a single, human-readable character is a flag emoji. These emojis consist of two "REGIONAL INDICATOR SYMBOL LETTER" Unicode scalar values.

```
// Declare a constant for a string with a single character, the emoji
// for the Canadian flag, which consists of two Unicode scalar values:
// - REGIONAL INDICATOR SYMBOL LETTER C (U+1F1E8)
// - REGIONAL INDICATOR SYMBOL LETTER A (U+1F1E6)
//
let canadianFlag: Character = "\u{1F1E8}\u{1F1E6}"
// `canadianFlag` is `🇨🇦`
```

Arrays

Arrays are mutable, ordered collections of values. All values in an array must have the same type. Arrays may contain a value multiple times. Array literals start with an opening square bracket `[` and end with a closing square bracket `]`.

```
// An empty array
//
[]

// An array with integers
//
[1, 2, 3]

// Invalid: mixed types
//
[1, true, 2, false]
```

Array Types

Arrays either have a fixed size or are variably sized, i.e., elements can be added and removed.

Fixed-size arrays have the type suffix `[N]`, where `N` is the size of the array. For example, a fixed-size array of 3 `Int8` elements has the type `Int8[3]`.

Variable-size arrays have the type suffix `[]`. For example, the type `Int16[]` specifies a variable-size array of elements that have type `Int16`.

```
let array: Int8[2] = [1, 2]

let arrays: Int16[2][3] = [
    [1, 2, 3],
    [4, 5, 6]
]
```

Array Indexing

To get the element of an array at a specific index, the indexing syntax can be used: The array is followed by an opening square bracket `[` the indexing value, and ends with a closing square bracket `]`.

Accessing an element which is out of bounds results in a fatal error.

```
// Declare an array
let numbers = [42, 23]

// Get the first number
//
numbers[0] // is 42

// Get the second number
//
numbers[1] // is 23
```

```
// Error: Index 2 is out of bounds
//
numbers[2]
```

```
let arrays = [[1, 2], [3, 4]]

// Get the first number of the second array
//
arrays[1][0] // is 3
```

To set an element of an array at a specific index, the indexing syntax can be used as well.

```
let numbers = [42, 23]

// Change the second number
//
// NOTE: The declaration `numbers` is constant, which means that
// the *name* is constant, not the *value* – the value, i.e. the array,
// is mutable and can be changed
//
numbers[1] = 2

// `numbers` is [42, 2]
```

Dictionaries

> 🚧 Status: Dictionaries are not implemented yet.

Dictionaries are mutable, unordered collections of key-value associations. In a dictionary, all keys must have the same type, and all values must have the same type. Dictionaries may contain a key only once and may contain a value multiple times.

Dictionary literals start with an opening brace `{` and end with a closing brace `}`. Keys are separated from values by a colon, and key-value associations are separated by commas.

```
// An empty dictionary
//
{}

// A dictionary which associates integers with booleans
//
{
    1: true,
    2: false
}

// Invalid: mixed types
//
{
    1: true,
    false: 2
}
```

Dictionary Types

Dictionaries have the type suffix `[T]`, where `T` is the type of the key. For example, a dictionary with `Int` keys and `Bool` values has type `Bool[Int]`.

```
// Declare a constant that has type `Bool[Int]`,
// a dictionary mapping integers to booleans
//
let booleans = {
    1: true,
    0: false
}
```

```
// Declare a constant that has type `Int[Bool]`,
// a dictionary mapping booleans to integers
//
let integers = {
    true: 1,
    false: 0
}
```

Dictionary Access

To get the value for a specific key from a dictionary, the access syntax can be used: The dictionary is followed by an opening square bracket `[`, the key, and ends with a closing square bracket `]`.

Accessing a key returns an optional: If the key is found in the dictionary, the value for the given key is returned, and if the key is not found `nil` is returned.

```
// Declare a constant that has type `Bool[Int]`,
// a dictionary mapping integers to booleans
//
let booleans = {
    1: true,
    0: false
}

// The result of accessing a key has type `Bool?`
//
booleans[1] // is true
booleans[0] // is false
booleans[2] // is nil

// Invalid: Accessing a key which does not have type `Int`
//
booleans["1"]
```

```
// Declare a constant that has type `Int[Bool]`,
// a dictionary mapping booleans to integers
//
let integers = {
    true: 1,
    false: 0
}

// The result of accessing a key has type `Int?`
//
integers[true] // is 1
integers[false] // is 0
```

To set the value for a key of a dictionary, the access syntax can be used as well.

```
let booleans = {
    1: true,
    0: false
}
booleans[1] = false
booleans[0] = true
// `booleans` is `{1: false, 0: true}`
```

Dictionary Keys

Dictionary keys must be hashable and equatable, i.e., must implement the `Hashable` and `Equatable` interfaces.

Most of the built-in types, like booleans, integers, are hashable and equatable, so can be used as keys in dictionaries.

Operators

Operators are special symbols that perform a computation for one or more values. They are either unary, binary, or ternary.

- Unary operators perform an operation for a single value. The unary operator symbol appears before the value.
- Binary operators operate on two values. The binary operator symbol appears between the two values (infix).
- Ternary operators operate on three values. The operator symbols appear between the three values (infix).

Negation

The `-` unary operator negates an integer:

```
let a = 1
-a // is -1
```

The `!` unary operator logically negates a boolean:

```
let a = true
!a // is false
```

Assignment

The binary assignment operator `=` can be used to assign a new value to a variable. It is only allowed in a statement and is not allowed in expressions.

```
var a = 1
a = 2
// `a` is 2
```

The left-hand side of the assignment must be an identifier, followed by one or more index or access expressions.

```
let numbers = [1, 2]

// Change the first number
//
numbers[0] = 3

// `numbers` is [3, 2]
```

```
let arrays = [[1, 2], [3, 4]]

// Change the first number in the second array
//
arrays[1][0] = 5

// `arrays` is [[1, 2], [5, 4]]
```

```
let dictionaries = {
  true: {1: 2},
  false: {3: 4}
}

dictionaries[false][3] = 0

// `dictionaries` is {
//   true: {1: 2},
//   false: {3: 0}
//}
```

Arithmetic

There are four arithmetic operators:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Remainder: `%`

```
let a = 1 + 2
// `a` is 3
```

The arguments for the operators need to be of the same type.

Arithmetic operators do not cause values to overflow.

```
let a: Int8 = 100
let b: Int8 = 100
let c = a * b
// `c` is 10000, and has type `Int`
```

If overflow behavior is intended, overflowing operators are available, which are prefixed with an `&`:

- Overflow addition: `&+`
- Overflow subtraction: `&-`
- Overflow multiplication: `&*`

For example, the maximum value of an unsigned 8-bit integer is 255 (binary 11111111). Adding 1 results in an overflow, truncation to 8 bits and the value 0.

```
//     11111111 = 255
// &+         1
//  = 100000000 = 0
```

```
let a: UInt8 = 255
a &+ 1 // is 0
```

Similarly, for the minimum value 0, subtracting 1 wraps around and results in the maximum value 255.

```
//     00000000
// &-         1
//  =  11111111 = 255
```

```
let b: UInt8 = 0
b &- 1 // is 255
```

Signed integers are also affected by overflow. In a signed integer, the first bit is used for the sign. This leaves 7 bits for the actual value fo an 8-bit signed integer, i.e., the range of values is -128 (binary 10000000) to 127 (01111111). Subtracting 1 from -128 results in 127.

```
//    10000000 = -128
// &-        1
//  = 01111111 = 127
```

```
let c: Int8 = -128
c &- 1 // is 127
```

Logical Operators

Logical operators work with the boolean values `true` and `false`.

- Logical AND: `a && b`

  ```
  true && true
  // is true

  true && false
  // is false

  false && true
  // is false

  false && false
  // is false
  ```

  If the left-hand side is true, the right-hand side is not evaluated.

- Logical OR: `a || b`

  ```
  true || true
  // is true

  true || false
  // is true

  false || true
  // is true

  false || false
  // is false
  ```

  If the left-hand side is false, the right-hand side is not evaluated.

Comparison operators

Comparison operators work with boolean and integer values.

- Equality: `==`, for booleans and integers (possibly optional)

  ```
  1 == 1
  // is true

  1 == 2
  // is false
  ```

  ```
  true == true
  // is true

  true == false
  // is false
  ```

  ```
  let x: Int? = 1
  x == nil
  // is false
  ```

  ```
  let x: Int = 1
  x == nil
  // is false
  ```

  ```
  let x: Int? = 2
  let y: Int?? = nil
  x == y
  // is false
  ```

```
let x: Int? = 2
let y: Int?? = 2
x == y
// is true
```

- Inequality: `!=`, for booleans and integers (possibly optional)

  ```
  1 != 1
  // is false

  1 != 2
  // is true
  ```

  ```
  true != true
  // is false

  true != false
  // is true
  ```

  ```
  let x: Int? = 1
  x != nil
  // is true
  ```

  ```
  let x: Int = 1
  x != nil
  // is true
  ```

  ```
  let x: Int? = 2
  let y: Int?? = nil
  x != y
  // is true
  ```

  ```
  let x: Int? = 2
  let y: Int?? = 2
  x != y
  // is false
  ```

- Less than: `<`, for integers

  ```
  1 < 1
  // is false

  1 < 2
  // is true

  2 < 1
  // is false
  ```

- Less or equal than: `<=`, for integers

  ```
  1 <= 1
  // is true

  1 <= 2
  // is true

  2 <= 1
  // is false
  ```

- Greater than: `>`, for integers

  ```
  1 > 1
  // is false

  1 > 2
  // is false

  2 > 1
  // is true
  ```

- Greater or equal than: `>=`, for integers

  ```
  1 >= 1
  // is true

  1 >= 2
  // is false

  2 >= 1
  // is true
  ```

Ternary Conditional Operator

There is only one ternary conditional operator, the ternary conditional operator (`a ? b : c`).

It behaves like an if-statement, but is an expression: If the first operator value is true, the second operator value is returned. If the first operator value is false, the third value is returned.

```
let x = 1 > 2 ? 3 : 4
// `x` is 4
```

Precedence and Associativity

Operators have the following precedences, highest to lowest:

- Multiplication precedence: `*`, `&*`, `/`, `%`
- Addition precedence: `+`, `&+`, `-`, `&-`
- Relational precedence: `<`, `<=`, `>`, `>=`
- Equality precedence: `==`, `!=`
- Logical conjunction precedence: `&&`
- Logical disjunction precedence: `||`
- Ternary precedence: `? :`

All operators are left-associative, except for the ternary operator, which is right-associative.

Expressions can be wrapped in parentheses to override precedence conventions, i.e. an alternate order should be indicated, or when the default order should be emphasized, e.g. to avoid confusion. For example, `(2 + 3) * 4` forces addition to precede multiplication, and `5 + (6 * 7)` reinforces the default order.

Functions

Functions are sequences of statements that perform a specific task. Functions have parameters (inputs) and an optional return value (output). Functions are typed: the function type consists of the parameter types and the return type.

Functions are values, i.e., they can be assigned to constants and variables, and can be passed as arguments to other functions. This behavior is often called "first-class functions".

Function Declarations

Functions can be declared by using the `fun` keyword, followed by the name of the declaration, the parameters, the optional return type, and the code that should be executed when the function is called.

The parameters need to be enclosed in parentheses. The return type, if any, is separated from the parameters by a colon (`:`). The function code needs to be enclosed in opening and closing braces.

Each parameter can have a label, the name that a function call needs to use to provide an argument value for the parameter. Argument labels precede the parameter name. The special argument label `_` indicates that a function call can omit the argument label. If no argument label is provided, the function call must use the parameter name.

Each parameter needs to have a type annotation, which follows the parameter name after a colon.

```
// Declare a function named `double`, which multiples a number by two.
//
// The special argument label _ is specified for the parameter,
// so no argument label has to be provided in a function call
//
fun double(_ x: Int): Int {
    return x * 2
}

// Call the function named `double` with the value 4 for the first parameter.
//
// The argument label can be omitted in the function call as the declaration
// specifies the special argument label _ for the parameter
//
double(2) // returns 4
```

It is possible to require argument labels for some parameters, and not require argument labels for other parameters.

```
// Declare a function named `clamp`. The function takes an integer value,
// the lower limit, and the upper limit. It returns an integer between
// the lower and upper limit.
//
// For the first parameter the special argument label _ is used,
// so no argument label has to be given for it in a function call.
//
// For the second and third parameter no argument label is given,
// so the parameter names are the argument labels, i.e., the parameter names
// have to be given as argument labels in a function call.
//
fun clamp(_ value: Int, min: Int, max: Int): Int {
    if value > max {
        return max
    }

    if value < min {
        return min
    }

    return value
}

// Declare a constant which has the result of a call to the function
// named `clamp` as its initial value.
//
// For the first argument no label is given, as it is not required by
// the function declaration (the special argument label `_` is specified).
//
// For the second and this argument the labels must be provided,
// as the function declaration does not specify the special argument label `_`
// for these two parameters.
//
// As the function declaration also does not specify argument labels
// for these parameters, the parameter names must be used as argument labels.
//
let clamped = clamp(123, min: 0, max: 100)
// `clamped` is 100
```

Argument labels make code more explicit and readable. For example, they avoid confusion about the order of arguments when there are multiple arguments that have the same type.

Argument labels should be named so they make sense from the perspective of the function call.

```
// Declare a function named `send`, which transfers an amount
// from one account to another.
//
// The implementation is omitted for brevity.
//
// The first two parameters of the function have the same type, so there is
// a potential that a function call accidentally provides arguments in
// the wrong order.
//
// While the parameter names `sendingAccount` and `receivingAccount`
// are descriptive inside the function, they might be too verbose
// to require them as argument labels in function calls.
//
// For this reason the shorter argument labels `from` and `to` are specified,
// which still convey the meaning of the two parameters without being overly
// verbose.
//
// The name of the third parameter, `amount`, is both meaningful inside
// the function and also in a function call, so no argument label is given,
// and the parameter name is required as the argument label in a function call.
//
fun send(from sendingAccount: Account, to receivingAccount: Account, amount: Int) {
    // ...
}

// Declare a constant which refers to the sending account.
//
// The initial value is omitted for brevity
//
let sender: Account = // ...

// Declare a constant which refers to the receiving account.
//
// The initial value is omitted for brevity
//
let receiver: Account = // ...

// Call the function named `send`.
//
// The function declaration requires argument labels for all parameters,
// so they need to be provided in the function call.
//
// This avoids ambiguity. For example, in some languages (like C) it is
// a convention to order the parameters so that the receiver occurs first,
// followed by the sender. In other languages, it is common to have
// the sender be the first parameter, followed by the receiver.
//
// Here, the order is clear – send an amount from an account to another account.
//
send(from: sender, to: receiver, amount: 100)
```

The order of the arguments in a function call must match the order of the parameters in the function declaration.

```
// Declare a function named `test`, which accepts two parameters, named `first` and `second`
//
fun test(first: Int, second: Int) {
    // ...
}

// Invalid: the arguments are provided in the wrong order,
// even though the argument labels are provided correctly
//
test(second: 1, first: 2)
```

Functions can be nested, i.e., the code of a function may declare further functions.

```
// Declare a function which multiplies a number by two, and adds one
//
fun doubleAndAddOne(_ x: Int): Int {

    // Declare a nested function which multiplies a number by two
    //
    fun double(_ x: Int) {
        return x * 2
    }

    return double(x) + 1
}

doubleAndAddOne(2) // is 5
```

Function Expressions

Functions can be also used as expressions. The syntax is the same as for function declarations, except that function expressions have no name, i.e., it is anonymous.

```
// Declare a constant named `double`, which has a function as its value.
//
// The function multiplies a number by two when it is called
//
let double =
    fun (_ x: Int): Int {
        return x * 2
    }
```

Function Calls

Functions can be called (invoked). Function calls need to provide exactly as many argument values as the function has parameters.

```
fun double(_ x: Int): Int {
    return x * 2
}

// Valid: the correct amount of arguments is provided
//
double(2) // is 4

// Invalid: too many arguments are provided
//
double(2, 3)

// Invalid: too few arguments are provided
//
double()
```

Function Types

Function types consist of the function's parameter types and the function's return type.

The parameter types need to be enclosed in parentheses, followed by a colon (`:`), and end with the return type. The whole function type needs to be enclosed in parentheses.

```
// Declare a function named `add`, with the function type `((Int, Int): Int)`
//
fun add(a: Int, b: Int): Int {
    return a + b
}
```

```
// Declare a constant named `add`, with the function type `((Int, Int): Int)`
//
let add: ((Int, Int): Int) =
    fun (a: Int, b: Int): Int {
        return a + b
    }
```

If the function has no return type, it implicitly has the return type `Void`.

```
// Declare a constant named `doNothing`, which is a function
// that takes no parameters and returns nothing
//
let doNothing: ((): Void) =
    fun () {}
```

Parentheses also control precedence. For example, a function type `((Int): ((): Int))` is the type for a function which accepts one argument with type `Int`, and which returns another function, that takes no arguments and returns an `Int`.

The type `((Int): Int)[2]` specifies an array type of two functions, which accept one integer and return one integer.

Argument Passing Behavior

When arguments are passed to a function, they are copied. Therefore, values that are passed into a function are unchanged in the caller's scope when the function returns. This behavior is known as call-by-value.

```
// Declare a function that changes the first two elements
// of an array of integers
//
fun change(_ numbers: Int[]) {
    // Change the elements of the passed in array.
    // The changes are only local, as the array was copied
    //
    numbers[0] = 1
    numbers[1] = 2
    // `numbers` is [1, 2]
}

let numbers = [0, 1]

change(numbers)
// `numbers` is still [0, 1]
```

Parameters are constant, i.e., it is not allowed to assign to them.

```
fun test(x: Int) {
    // Invalid: cannot assign to a parameter (constant)
    //
    x = 2
}
```

Function Preconditions and Postconditions

Functions may have preconditions and may have postconditions. Preconditions and postconditions can be used to restrict the inputs (values for parameters) and output (return value) of a function.

Preconditions must be true right before the execution of the function. Preconditions are part of the function and introduced by the `pre` keyword, followed by the condition block.

Postconditions must be true right after the execution of the function. Postconditions are part of the function and introduced by the `post` keyword, followed by the condition block. Postconditions may only occur after preconditions, if any.

A conditions block consists of one or more conditions. Conditions are expressions evaluating to a boolean. They may not call functions, i.e., they cannot have side-effects and must be pure expressions. Also, conditions may not contain function expressions.

Conditions may be written on separate lines, or multiple conditions can be written on the same line, separated by a semicolon. This synta follows the syntax for statements.

Following each condition, an optional description can be provided after a colon.

In postconditions, the special constant `result` refers to the result of the function.

```
fun factorial(_ n: Int): Int {
    pre {
        // Require the parameter `n` to be greater than or equal to zero
        //
        n >= 0:
            "factorial is only defined for integers greater than or equal to zero"
    }
    post {
        // Ensure the result will be greater than or equal to 1
        //
        result >= 1:
            "the result must be greater than or equal to 1"
    }

    if n < 1 {
       return 1
    }

    return n * factorial(n - 1)
}

factorial(5) // returns 120

// Error: the given argument does not satisfy the precondition `n >= 0` of the function
//
factorial(-2)
```

In postconditions, the special function `before` can be used to get the value of an expression just before the function is called.

```
var n = 0

fun incrementN() {
    post {
        // Require the new value of `n` to be the old value of `n`, plus one
        //
        n == before(n) + 1:
            "n must be incremented by 1"
    }

    n = n + 1
}
```

Control flow

Control flow statements control the flow of execution in a function.

Conditional branching: if-statement

If-statements allow a certain piece of code to be executed only when a given condition is true.

The if-statement starts with the `if` keyword, followed by the condition, and the code that should be executed if the condition is true inside opening and closing braces. The condition must be boolean and the braces are required.

```
let a = 0
var b = 0

if a == 0 {
   b = 1
}
```

```
if a != 0 {
   b = 2
}

// `b` is 1
```

An additional else-clause can be added to execute another piece of code when the condition is false. The else-clause is introduced by the `else` keyword.

```
let a = 0
var b = 0

if a == 1 {
   b = 1
} else {
   b = 2
}

// `b` is 2
```

The else-clause can contain another if-statement, i.e., if-statements can be chained together.

```
let a = 0
var b = 0

if a == 1 {
   b = 1
} else if a == 2 {
   b = 2
} else {
   b = 3
}

// `b` is 3
```

Looping: while-statement

While-statements allow a certain piece of code to be executed repeatedly, as long as a condition remains true.

The while-statement starts with the `while` keyword, followed by the condition, and the code that should be repeatedly executed if the condition is true inside opening and closing braces. The condition must be boolean and the braces are required.

The while-statement will first evaluate the condition. If the condition is false, the execution is done. If it is true, the piece of code is executed and the evaluation of the condition is repeated. Thus, the piece of code is executed zero or more times.

```
var a = 0
while a < 5 {
    a = a + 1
}

// `a` is 5
```

The `continue` statement can be used to stop the current iteration of the loop and start the next iteration.

```
var i = 0
var x = 0
while i < 10 {
    i = i + 1
    if i < 5 {
        continue
    }
    x = x + 1
```

```
}

// `x` is 6
```

The `break` statement can be used to stop the loop.

```
var x = 0
while x < 10 {
    x = x + 1
    if x == 5 {
        break
    }
}

// `x` is 5
```

Immediate function return: return-statement

The return-statement causes a function to return immediately, i.e., any code after the return-statement is not executed. The return-statement starts with the `return` keyword and is followed by an optional expression that should be the return value of the function call.

Scope

Every function and block (`{` ... `}`) introduces a new scope for declarations. Each function and block can refer to declarations in its scope or any of the outer scopes.

```
let x = 10

fun f(): Int {
    let y = 10
    return x + y
}

f() // returns 20

// Invalid: the identifier `y` is not in scope
//
y
```

```
fun doubleAndAddOne(_ n: Int): Int {
    fun double(_ x: Int) {
        return x * 2
    }
    return double(n) + 1
}

// Invalid: the identifier `double` is not in scope
//
double(1)
```

Each scope can introduce new declarations, i.e., the outer declaration is shadowed.

```
let x = 2

fun test(): Int {
    let x = 3
    return x
}

test() // returns 3
```

Scope is lexical, not dynamic.

```
let x = 10

fun f(): Int {
   return x
}

fun g(): Int {
   let x = 20
   return f()
}

g() // returns 10, not 20
```

Declarations are **not** moved to the top of the enclosing function (hoisted).

```
let x = 2

fun f(): Int {
    if x == 0 {
        let x = 3
        return x
    }
    return x
}
f() // returns 2
```

Type Safety

The Bamboo programming language is a *type-safe* language.

When assigning a new value to a variable, the value must be the same type as the variable. For example, if a variable has type `Bool`, it can *only* be assigned a value that has type `Bool`, and not for example a value that has type `Int`.

```
// Declare a variable that has type `Bool`
var a = true

// Invalid: cannot assign a value that has type `Int` to a variable which has type `Bool`
//
a = 0
```

When passing arguments to a function, the types of the values must match the function parameters' types. For example, if a function expects an argument that has type `Bool`, *only* a value that has type `Bool` can be provided, and not for example a value which has type `Int`.

```
fun nand(_ a: Bool, _ b: Bool): Bool {
    return !(a && b)
}

nand(false, false) // returns true

// Invalid: integers are not booleans
//
nand(0, 0)
```

Types are **not** automatically converted. For example, an integer is not automatically converted to a boolean, nor is an `Int32` automaticall converted to an `Int8`, nor is an optional integer `Int?` automatically converted to a non-optional integer `Int`.

```
fun add(_ a: Int8, _ b: Int8): Int {
    return a + b
}

add(1, 2) // returns 3
```

```
// Declare two constants which have type `Int32`
//
let a: Int32 = 3_000_000_000
let b: Int32 = 3_000_000_000

// Invalid: cannot pass arguments which have type `Int32` to parameters which have type `Int8`
//
add(a, b)
```

Type Inference

> 🚧 Status: Only basic type inference is implemented.

If a variable or constant is not annotated explicitly with a type, it is inferred from the value.

Integer literals are inferred to type `Int`.

```
let a = 1

// `a` has type `Int`
```

Array literals are inferred based on the elements of the literal, and to be variable-size.

```
let integers = [1, 2]
// `integers` has type `Int[]`

// Invalid: mixed types
//
let invalidMixed = [1, true, 2, false]
```

Dictionary literals are inferred based on the keys and values of the literal.

```
let booleans = {
    1: true,
    2: false
}
// `booleans` has type `Bool[Int]`

// Invalid: mixed types
//
let invalidMixed = {
    1: true,
    false: 2
}
```

Functions are inferred based on the parameter types and the return type.

```
let add = (a: Int8, b: Int8): Int {
    return a + b
}

// `add` has type `((Int8, Int8): Int)`
```

Type inference is performed for eacg expression / statement, and not across statements.

There are cases where types cannot be inferred. In these cases explicit type annotations are required.

```
// Invalid: not possible to infer type based on array literal's elements
//
let array = []
```

```
// Invalid: not possible to infer type based on dictionary literal's keys and values
//
let dictionary = {}
```

Composite Data Types

> 🚧 Status: Resources are not implemented yet.

Composite data types allow composing simpler types into more complex types, i.e., they allow the composition of multiple values into one. Composite data types have a name and consist of zero or more named fields, and zero or more functions that operate on the data. Each field may have a different type.

There are two kinds of composite data types. The kinds differ in their usage and the behaviour when a value is used as the initial value for a constant or variable, when the value is assigned to a variable, when the value is passed as an argument to a function, and when the value is returned from a function:

- **Structures** are **copied**, i.e. they are value types. Structures are useful when copies with independent state are desired.
- **Resources** are **moved**, they are linear types and **must** be used **exactly once**.

  Resources are useful when it is desired to model ownership (a value exists exactly in one location and it should not be lost).

  Certain constructs in a blockchain represent assets of real, tangible value, as much as a house or car or bank account. We have to worry about literal loss and theft, perhaps even on the scale of millions of dollars.

  We think resources are a great way to represent such assets.

Two composite daya types are compatible if and only if they refer to the same declaration by name, i.e., nominal typing applies instead of structural typing.

Composite Data Type Declaration and Creation

Structures are declared using the `struct` keyword and resources are declared using the `resource` keyword. The keyword is followed by the name.

```
struct SomeStruct {
    // ...
}

resource SomeResource {
    // ...
}
```

Structures and resources are types.

Structures are created (instantiated) by calling the type like a function.

```
SomeStruct()
```

Resource are created (instantiated) by using the `create` keyword and calling the type like a function.

```
create SomeResource()
```

Composite data types can only be declared globally and not locally in functions. They can also not be nested.

Composite Data Type Fields

Fields are declared like variables and constants, however, they have no initial value. The initial values for fields are set in the initializer. All fields **must** be initialized in the initializer. The initializer is declared using the `init` keyword. Just like a function, it takes parameters. However, it has no return type, i.e., it is always `Void`. The initializer always follows any fields.

There are three kinds of fields.

Variable fields are stored in the composite value and can have new values assigned to them. They are declared using the `var` keyword.

Constant fields are also stored in the composite value, but they can **not** have new values assigned to them. They are declared using the `let` keyword.

Synthetic fields are **not** stored in the composite value, i.e. they are derived/computed from other values. They can have new values assigned to them and are declared using the `synthetic` keyword. Synthetic fields must have a getter and a setter. Getters and setters are explained in the next section. Synthetic fields are explained in a separate section.

Field Kind	Stored in memory	Assignable	Keyword
Variable field	Yes	Yes	`var`
Constant field	Yes	**No**	`let`
Synthetic field	**No**	Yes	`synthetic`

```
// Declare a structure named `Token`, which has a constant field
// named `id` and a variable field named `balance`.
//
// Both fields are initialized through the initializer
//
struct Token {
    let id: Int
    var balance: Int

    init(id: Int, balance: Int) {
        self.id = id
        self.balance = balance
    }
}
```

In initializers, the special constant `self` refers to the composite value that is to be initialized.

Fields can be read (if they are constant or variable) and set (if they are variable), using the access syntax: the composite value is followed by a dot (`.`) and the name of the field.

```
let token = Token(id: 42, balance: 1_000_00)

token.id // is 42
token.balance // is 1_000_000

token.balance = 1
// `token.balance` is 1

// Invalid: assignment to constant field
//
token.id = 23
```

Composite Data Type Field Getters and Setters

Fields may have an optional getter and an optional setter. Getters are functions that are called when a field is read, and setters are functions that are called when a field is written.

Getters and setters are enclosed in opening and closing braces, after the field's type.

Getters are declared using the `get` keyword. Getters have no parameters and their return type is implicitly the type of the field.

```
struct GetterExample {

    // Declare a variable field named `balance` with a getter
    // which ensures the read value is always positive
    //
    var balance: Int {
```

```
        get {
           post {
               result >= 0
           }

           if self.balance < 0 {
               return 0
           }

           return self.balance
        }
    }

    init(balance: Int) {
        self.balance = balance
    }
}

let example = GetterExample(balance: 10)
// `example.balance` is 10

example.balance = -50
// `example.balance` is 0. without the getter it would be -50
```

Setters are declared using the `set` keyword, followed by the name for the new value enclosed in parentheses. The parameter has implicitly the type of the field. Another type cannot be specified. Setters have no return type.

The types of values assigned to setters must always match the field's type.

```
struct SetterExample {

    // Declare a variable field named `balance` with a setter
    // which requires written values to be positive
    //
    var balance: Int {
        set(newBalance) {
            pre {
                newBalance >= 0
            }
            self.balance = newBalance
        }
    }

    init(balance: Int) {
        self.balance = balance
    }
}

let example = SetterExample(balance: 10)
// `example.balance` is 10

// error: precondition of setter for field balance failed
example.balance = -50
```

Synthetic Composite Data Type Fields

Fields which are not stored in the composite value are *synthetic*, i.e., the field value is computed. Synthetic can be either read-only, or readable and writable.

Synthetic fields are declared using the `synthetic` keyword.

Synthetic fields are read-only when only a getter is provided.

```
struct Rectangle {
    var width: Int
    var height: Int

    // Declare a synthetic field named `area`,
    // which computes the area based on the width and height
```

```
    //
    synthetic area: Int {
        get {
            return width * height
        }
    }

    init(width: Int, height: Int) {
        self.width = width
        self.height = height
    }
}
```

Synthetic fields are readable and writable when both a getter and a setter is declared.

```
// Declare a struct named `GoalTracker` which stores a number
// of target goals, a number of completed goals,
// and has a synthetic field to provide the left number of goals
//
// NOTE: the tracker only implements some functionality to demonstrate
// synthetic fields, it is incomplete (e.g. assignments to `goal` are not handled properly)
//
struct GoalTracker {

    var goal: Int
    var completed: Int

    // Declare a synthetic field which is both readable
    // and writable.
    //
    // When the field is read from (in the getter),
    // the number of left goals is computed from
    // the target number of goals and
    // the completed number of goals.
    //
    // When the field is written to (in the setter),
    // the number of completed goals is updated,
    // based on the number of target goals
    // and the new remaining number of goals
    //
    synthetic left: Double {
        get {
            return self.goal - self.completed
        }

        set(newLeft) {
            self.completed = self.goal - newLeft
        }
    }

    init(goal: Int, completed: Int) {
        self.goal = goal
        self.completed = completed
    }
}

let tracker = GoalTracker(goal: 10, completed: 0)
// `tracker.goal` is 10
// `tracker.completed` is 0
// `tracker.left` is 10

tracker.completed = 1
// `tracker.left` is 9

tracker.left = 8
// `tracker.completed` is 2
```

It is invalid to declare a synthetic field with only a setter.

Composite Data Type Functions

Composite data types may contain functions. Just like in the initializer, the special constant `self` refers to the composite value that the function is called on.

```
struct Rectangle {
    var width: Int
    var height: Int

    init(width: Int, height: Int) {
        self.width = width
        self.height = height
    }

    fun scale(factor: Int) {
        self.width = self.width * factor
        self.height = self.height * factor
    }
}

let rectangle = Rectangle(width: 2, height: 3)
rectangle.scale(factor: 4)
// `rectangle.width` is 8
// `rectangle.height` is 12
```

Composite Data Type Behaviour

Structures

Structures are **copied** when used as an initial value for constant or variable, when assigned to a different variable, when passed as an argument to a function, and when returned from a function:

```
// Declare a structure named `SomeStruct`, with a variable integer field
//
struct SomeStruct {
    var value: Int

    init(value: Int) {
        self.value = value
    }
}

// Declare a constant with value of structure type `SomeStruct`
//
let a = SomeStruct(value: 0)

// *Copy* the structure value into a new constant
//
let b = a

b.value = 1

a.value // is *0*
```

Resources

Resources are **moved** when used as an initial value for a constant or variable, when assigned to a different variable, when passed as an argument to a function, and when returned from a function.

When the resource was moved, the constant or variable that referred to the resource before the move becomes **invalid**.

To make the move explicit, the move operator `<-` must be used when the resource is the initial value of a constant or variable, when it moved to a different variable, or when it moved to a function.

```
// Declare a resource named `SomeResource`, with a variable integer field
//
resource SomeResource {
    var value: Int
```

```
    init(value: Int) {
        self.value = value
    }
}

// Declare a constant with value of resource type `SomeResource`
//
let a <- SomeResource(value: 0)

// *Move* the resource value to a new constant
//
let b <- a

// Invalid: Cannot use constant `a` anymore as the resource
// it referred to was moved to constant `b`
//
a.value

// Constant `b` owns the resource
//
b.value = 1

// Declare a function which accepts a resource.
//
// The parameter has a resource type, so the type name must be prefixed with `<-`
//
fun use(resource: <-SomeResource) {
    // ...
}

// Call function `use` and move the resource into it
//
use(<-b)

// Invalid: Cannot use constant `b` anymore as the resource
// it referred to was moved into function `foo`
//
b.value
```

Resources **must** be used **exactly once**. To destroy a resource, the `destroy` keyword must be used.

```
// Declare another, unrelated value of resource type `SomeResource`
//
let c = SomeResource(value: 10)

// Invalid: `c` is not used, but must be! `c` cannot be lost
```

```
// Declare another, unrelated value of resource type `SomeResource`
//
let d = SomeResource(value: 20)

// Destroy the resource referred to by constant `d`
//
destroy d

// Invalid: Cannot use constant `d` anymore as the resource
// it referred to was destroyed
//
d.value
```

To make it explicit that the type is moved, it must be prefixed with `<-` in type annotations.

```
// Declare a constant with an explicit type annotation.
//
// The constant has a resource type, so the type name must be prefixed with `<-`
//
let someResource: <-SomeResource <- create SomeResource()
```

```
// Declare a function which consumes a resource.
//
// The parameter has a resource type, so the type name must be prefixed with `<-`
//
fun use(resource: <-SomeResource) {
    destroy resource
}

// Declare a function which returns a resource.
//
// The return type is a resource type, so the type name must be prefixed with `<-`
//
fun get(): <-SomeResource {
    return create SomeResource()
}
```

Resources in Arrays and Dictionaries

Arrays and dictionaries behave differently when they contain resources: When a resource is **read** from the array at a certain index, or it is **read** from a dictionary by accessing a certain key, the resource is **moved** out of the array or dictionary.

```
let resources = [
    SomeResource(value: 1),
    SomeResource(value: 2),
    SomeResource(value: 3)
]

// **Move** the first resource into a new constant
//
let firstResource <- resources[0]

// **Move** the second resource into a new constant
//
let secondResource <- resources[1]

// `resources` only contains one element,
// the initial third resource!
//
// The first two resources were moved out of the array when
// they were read, i.e., the were removed from the array
//
// Accessing a field of a resource does not move the resource
//
resource[0].value // is 3

// Error: cannot access second element of `resources`,
// as it only has one element left after the first two elements
// were accessed above
//
resource[1]
```

Unbound References / Nulls

There is **no** support for nulls`.

Inheritance and Abstract Types

There is **no** support for inheritance. Inheritance is a feature common in other programming languages, that allows including the fields and functions of one type in another type.

Instead, follow the "composition over inheritance" principle, the idea of composing functionality from multiple individual parts, rather thar building an inheritance tree.

Furthermore, there is also **no** support for abstract types. An abstract type is a feature common in other programming languages, that prevents creating values of the type and only allows the creation of values of a subtype. In addition, abstract types may declare functions but omit the implementation of them and instead require subtypes to implement them.

Instead, consider using interfaces.

Access control

> 🚧 Status: Access control is not implemented yet.

Access control allows making certain parts of the program accessible/visible and making other parts inaccessible/invisible. Top-level declarations (variables, constants, functions, structures, resources, interfaces) and fields (in structures, and resources) are either private or public.

Private means the declaration is only accessible/visible in the current and inner scopes. For example, a private field can only be accessed by functions of the type is part of, not by code that uses an instance of the type in an outer scope.

Public means the declaration is accessible/visible in all scopes, the current and inner scopes like for private, and the outer scopes. For example, a private field in a type can be accessed using the access syntax on an instance of the type in an outer scope.

By default, everything is private. The `pub` keyword is used to make declarations public.

The `(set)` suffix can be used to make variables also publicly writable.

To summarize the behavior for variable declarations, constant declarations, and fields:

Declaration kind	Access modifier	Read scope	Write scope
`let`		Current and inner	*None*
`let`	`pub`	**All**	*None*
`var`		Current and inner	Current and inner
`var`	`pub`	**All**	Current and inner
`var`	`pub(set)`	**All**	**All**

To summarize the behavior for functions, structures, resources, and interfaces:

Declaration kind	Access modifier	Access scope
`fun`, `struct`, `resource`, `struct interface`, `resource interface`		Current and inner
`fun`, `struct`, `resource`, `struct interface`, `resource interface`	`pub`	**All**

```
// Declare a private constant, inaccessible/invisible in outer scope
//
let a = 1

// Declare a public constant, accessible/visible in all scopes
//
pub let b = 2
```

```
// Declare a public struct, accessible/visible in all scopes
//
pub struct SomeStruct {

    // Declare a private constant field,
    // only readable in the current and inner scopes
    //
    let a: Int

    // Declare a public constant field, readable in all scopes
    //
    pub let b: Int

    // Declare a private variable field,
    // only readable and writable in the current and inner scopes
    //
```

```
    var c: Int

    // Declare a public variable field, not settable,
    // only writable in the current and inner scopes,
    // readable in all scopes
    //
    pub var d: Int

    // Declare a public variable field, settable,
    // readable and writable in all scopes
    //
    pub(set) var e: Int

    // NOTE: initializer implementation skipped

    // Declare a private function,
    // only callable in the current and inner scopes
    //
    fun privateTest() {
        // ...
    }

    // Declare a public function,
    // callable in all scopes
    //
    pub fun privateTest() {
        // ...
    }

    // The initializer is omitted for brevity.

}

let some = SomeStruct()

// Invalid: cannot read private constant field in outer scope
//
some.a

// Invalid: cannot set private constant field in outer scope
//
some.a = 1

// Valid: can read public constant field in outer scope
//
some.b

// Invalid: cannot set public constant field in outer scope
//
some.b = 2

// Invalid: cannot read private variable field in outer scope
//
some.c

// Invalid: cannot set private variable field in outer scope
//
some.c = 3

// Valid: can read public variable field in outer scope
//
some.d

// Invalid: cannot set public variable field in outer scope
//
some.d = 4

// Valid: can read publicly settable variable field in outer scope
//
some.e
```

```
// Valid: can set publicly settable variable field in outer scope
//
some.e = 5
```

Interfaces

> 🚧 Status: Interfaces are not implemented yet.

An interface is an abstract type that specifies the behavior of types that *implement* the interface. Interfaces declare the required function and fields, as well as the access for those declarations, that implementing types need to provide.

There are two kinds of interfaces:

- **Structure interfaces**: implemented by structures
- **Resource interfaces**: implemented by resources

Structure and resource types may implement multiple interfaces.

Interfaces consist of the function and field requirements that a type implementing the interface must provide implementations for. Interface requirements, and therefore also their implementations, must always be at least public. Variable field requirements may be annotated to require them to be publicly settable.

Function requirements consist of the name of the function, parameter types, an optional return type, and optional preconditions and postconditions.

Field requirements consist of the name and the type of the field. Field requirements may optionally declare a getter requirement and a setter requirement, each with preconditions and postconditions.

Calling functions with preconditions and postconditions on interfaces instead of concrete implementations can improve the security of a program, as it ensures that even if implementations change, some aspects of them will always hold.

Interface Declaration

Interfaces are declared using the `struct` or `resource` keyword, followed by the `interface` keyword, the name of the interface, and the requirements, which must be enclosed in opening and closing braces.

Field requirements can be annotated to require the implementation to be a variable field, by using the `var` keyword; require the implementation to be a constant field, by using the `let` keyword; or the field requirement may specify nothing, in which case the implementation may either be a variable field, a constant field, or a synthetic field.

Field requirements and function requirements must specify the required level of access. The access must be at least be public, so the `pub` keyword must be provided. Variable field requirements can be specified to also be publicly settable by using the `pub(set)` keyword

The special type `Self` can be used to refer to the type implementing the interface.

```
// Declare a resource interface for a fungible token.
// Only resources can implement this resource interface
//
resource interface FungibleToken {

    // Require the implementing type to provide a field for the balance
    // that is readable in all scopes (`pub`).
    //
    // Neither the `var` keyword, nor the `let` keyword is used,
    // so the field may be implemented as either a variable field,
    // a constant field, or a synthetic field.
    //
    // The read balance must always be positive.
    //
    // NOTE: no requirement is made for the kind of field,
    // it can be either variable or constant in the implementation
    //
    pub balance: Int {
        get {
            post {
```

```
                result >= 0:
                    "Balances are always non-negative"
            }
        }
    }

    // Require the implementing type to provide an initializer that
    // given the initial balance, must initialize the balance field
    //
    init(balance: Int) {
        post {
            self.balance == balance:
                "the balance must be initialized to the initial balance"
        }

        // NOTE: no code
    }

    // Require the implementing type to provide a function that is
    // callable in all scopes, which withdraws an amount from
    // this fungible token and returns the withdrawn amount as
    // a new fungible token.
    //
    // The given amount must be positive and the function implementation
    // must add the amount to the balance.
    //
    // The function must return a new fungible token.
    //
    // NOTE: `<-Self` is the resource type implementing this interface
    //
    pub fun withdraw(amount: Int): <-Self {
        pre {
            amount > 0:
                "the amount must be positive"
            amount <= self.balance:
                "insufficient funds: the amount must be smaller or equal to the balance"
        }
        post {
            self.balance == before(self.balance) - amount:
                "the amount must be deducted from the balance"
        }

        // NOTE: no code
    }

    // Require the implementing type to provide a function that is
    // callable in all scopes, which deposits a fungible token
    // into this fungible token.
    //
    // The given token must be of the same type - a deposit of another
    // type is not possible.
    //
    // No precondition is required to check the given token's balance
    // is positive, as this condition is already ensured by
    // the field requirement.
    //
    // NOTE: the first parameter has the type `<-Self`,
    // i.e. the resource type implementing this interface
    //
    pub fun deposit(_ token: <-Self) {
        post {
            self.balance == before(self.balance) + token.balance:
                "the amount must be added to the balance"
        }

        // NOTE: no code
    }
}
```

Note that the required initializer and functions do not have any executable code.

Interface Implementation

Implementations for interfaces are declared using the `impl` keyword, followed by the name of interface, the `for` keyword, and the name of the composite data type (structure or resource) that provides the functionality required in the interface.

```
// Declare a resource named `ExampleToken` with a variable field named `balance`,
// that can be written by functions of the type, but outer scopes can only read it
//
resource ExampleToken {

    // Implement the required field `balance` for the `FungibleToken` interface.
    // The interface does not specify if the field must be variable, constant,
    // so in order for this type (`ExampleToken`) to be able to write to the field,
    // but limit outer scopes to only read from the field, it is declared variable,
    // and only has public access (non-settable).
    //
    pub var balance: Int

    // Implement the required initializer for the `FungibleToken` interface:
    // accept an initial balance and initialize the `balance` field.
    //
    // This implementation satisfies the required postcondition
    //
    // NOTE: the postcondition declared in the interface
    // does not have to be repeated here in the implementation
    //
    init(balance: Int) {
        self.balance = balance
    }
}


// Declare the implementation of the interface `FungibleToken`
// for the resource `ExampleToken`
//
impl FungibleToken for ExampleToken {

    // Implement the required function named `withdraw` of the interface
    // `FungibleToken`, that withdraws an amount from the token's balance.
    //
    // The function must be public.
    //
    // This implementation satisfies the required postcondition.
    //
    // NOTE: neither the precondition nor the postcondition declared
    // in the interface have to be repeated here in the implementation
    //
    pub fun withdraw(amount: Int): <-ExampleToken {
        self.balance = self.balance - amount
        return create ExampleToken(balance: amount)
    }

    // Implement the required function named `deposit` of the interface
    // `FungibleToken`, that deposits the amount from the given token
    // to this token.
    //
    // The function must be public.
    //
    // NOTE: the type of the parameter is `<-ExampleToken`,
    // i.e., only a token of the same type can be deposited.
    //
    // This implementation satisfies the required postconditions.
    //
    // NOTE: neither the precondition nor the postcondition declared
    // in the interface have to be repeated here in the implementation
    //
    pub fun deposit(_ token: <-ExampleToken) {
        self.balance = self.balance + amount
        destroy token
    }
}

// Declare a constant which has type `ExampleToken`,
// and is initialized with such an example token
```

```
//
let token <- create ExampleToken(balance: 100)

// Withdraw 10 units from the token.
//
// The amount satisfies the precondition of the `withdraw` function
// in the `FungibleToken` interface
//
let withdrawn <- token.withdraw(amount: 10)

// The postcondition of the `withdraw` function in the `FungibleToken`
// interface ensured the balance field of the token was updated properly
//
// `token.balance` is 90
// `withdrawn.balance` is 10

// Deposit the withdrawn token into another one.
let receiver: ExampleToken <- // ...
receiver.deposit(<-withdrawn)

// Error: precondition of function `withdraw` in interface
// `FungibleToken` is not satisfied: the parameter `amount`
// is larger than the field `balance` (100 > 90)
//
token.withdraw(amount: 100)
```

The access level for variable fields in an implementation may be less restrictive than the interface requires. For example, an interface may require a field to be at least public (i.e. the `pub` keyword is specified), and an implementation may provide a variable field which is public, but also publicly settable (the `pub(set)` keyword is specified).

```
struct interface AnInterface {
    // Require the implementing type to provide a publicly readable
    // field named `a` that has type `Int`. It may be a constant field,
    // a variable field, or a synthetic field.
    //
    pub a: Int
}

struct AnImplementation {
    // Declare a publicly settable variable field named `a`that has type `Int`.
    // This implementation satisfies the requirement for interface `AnInterface`:
    // The field is at least publicly readable, but this implementation also
    // allows the field to be written to in all scopes
    //
    pub(set) var a: Int

    init(a: Int) {
        self.a = a
    }
}

impl AnInterface for AnImplementation {
    // This implementation is empty, as the declaration
    // of the structure `AnImplementation` already fully satisfies
    // the requirements of the interface `AnInterface`,
    // i.e. a field named `a` that has type `Int` must be provided
}
```

Interface Type

Interfaces are types. Values implementing an interface can be used as initial values for constants and variables that have the interface as their type.

```
// Declare an interface named `Shape`.
//
// Require implementing types to provide a field which returns the area,
// and a function which scales the shape by a given factor.
//
struct interface Shape {
```

```
    pub area: Int
    pub fun scale(factor: Int)
}

// Declare a structure named `Square`
//
struct Square {
    pub var length: Int

    pub synthetic area: Int {
        get {
            return self.length * self.length
        }
    }

    pub init(length: Int) {
        self.length = length
    }
}

// Implement the interface `Shape` for the structure `Square`
//
impl Shape for Square {

    pub fun scale(factor: Int) {
        self.length = self.length * factor
    }
}

// Declare a structure named `Rectangle`
//
struct Rectangle {
    pub var width: Int
    pub var height: Int

    pub synthetic area: Int {
        get {
            return self.width * self.height
        }
    }

    pub init(width: Int, height: Int) {
        self.width = width
        self.height = height
    }
}

// Implement the interface `Rectangle` for the structure `Square`
//
impl Shape for Rectangle {

    pub fun scale(factor: Int) {
        self.width = self.width * factor
        self.height = self.height * factor
    }
}

// Declare a constant that has type `Shape`, which has a value that has type `Rectangle`
//
var shape: Shape = Rectangle(width: 10, height: 20)
```

Values implementing an interface are assignable to variables that have the interface as their type.

```
// Assign a value of type `Square` to the variable `shape` that has type `Shape`
//
shape = Square(length: 30)

// Invalid: cannot initialize a constant that has type `Rectangle`
// with a value that has type `Square`
//
let rectangle: Rectangle = Square(length: 10)
```

Fields declared in an interface can be accessed and functions declared in an interface can be called on values of a type that implements the interface.

```
// Declare a constant which has the type `Shape`
// and is initialized with a value that has type `Rectangle`
//
let shape: Shape = Rectangle(width: 2, height: 3)

// Access the field `area` declared in the interface `Shape`
//
shape.area // is 6

// Call the function `scale` declared in the interface `Shape`
//
shape.scale(factor: 3)
```

Interface Implementation Requirements

Interfaces can require implementing types to also implement other interfaces of the same kind. Interface implementation requirements can be declared by following the interface name with a colon (`:`) and one or more names of interfaces of the same kind, separated by commas.

```
// Declare a structure interface named `Shape`
//
struct interface Shape {}

// Declare a structure interface named `Polygon`.
// Require implementing types to also implement
// the structure interface `Shape`
//
struct interface Polygon: Shape {}

// Declare a structure named `Hexagon`
//
struct Hexagon {}

// Implement the structure interface `Polygon`
// for the structure `Hexagon`
//
impl Polygon for Hexagon {}

// Implement the structure interface `Shape`
// for the structure `Hexagon`.
//
// This is required, as the interface `Polygon`
// specified this implementation requirement.
//
impl Shape for Hexagon {}
```

Interface Nesting

Interfaces can be arbitrarily nested. Declaring an interface inside another does not require implementing types of the outer interface to provide an implementation of the inner interfaces.

```
// Declare a resource interface `OuterInterface`, which declares
// a nested structure interface named `InnerInterface`.
//
// Resources implementing `OuterInterface` do not need to provide
// an implementation of `InnerInterface`.
//
// Structures may just implement `InnerInterface`
//
resource interface OuterInterface {

    struct interface InnerInterface {}
}
```

```
// Declare a resource named `SomeOuter`
//
resource SomeOuter {}

// Implement the interface `OuterInterface` for the resource  `SomeOuter`.
//
// The resource is not required to implement `OuterInterface.InnerInterface`
//
impl OuterInterface for SomeOuter {}


// Declare a structure named `SomeInner`
//
struct SomeInner {}

// Implement the interface `InnerInterface` which is nested in
// interface `OuterInterface` for the structure `SomeInner`.
//
impl OuterInterface.InnerInterface for SomeInner {}
```

Nested Type Requirements

Interfaces can require implementing types to provide concrete nested types. For example, a resource interface may require an implementing type to provide a resource type.

```
// Declare a resource interface named `FungibleToken`.
//
// Require implementing types to provide a resource type named `Vault`
// which must have a field named `balance`
//
resource interface FungibleToken {

    pub resource Vault {
        pub balance: Int
    }
}

// Declare a resource named `ExampleToken`
//
resource ExampleToken {}

// Implement the resource interface `FungibleToken`
// for resource type `ExampleToken`.
//
// The nested type `Vault` must be provided
// to conform to the interface.
//
impl FungibleToken for ExampleToken {

    pub resource Vault {
        pub var balance: Int

        init(balance: Int) {
            self.balance = balance
        }
    }
}
```

`Equatable` Interface

> 🚧 Status: The `Equatable` interface is not implemented yet.

An equatable type is a type that can be compared for equality. Types are equatable when they implement the `Equatable` interface.

Equatable types can be compared for equality using the equals operator (`==`) or inequality using the unequals operator (`!=`).

Most of the built-in types are equatable, like booleans and integers. Arrays are equatable when their elements are equatable. Dictionaries are equatable when their values are equatable.

To make a type equatable the `Equatable` interface must be implemented, which requires the implementation of the function `equals`, which accepts another value that the given value should be compared for equality. Note that the parameter type is `Self`, i.e., the other value must have the same type as the implementing type.

```
struct interface Equatable {
    pub fun equals(_ other: Self): Bool
}
```

```
// Declare a struct named `Cat`, which has one field named `id`
// that has type `Int`, i.e., the identifier of the cat.
//
struct Cat {
    pub let id: Int

    init(id: Int) {
        self.id = id
    }
}

// Implement the interface `Equatable` for the type `Cat`,
// to allow cats to be compared for equality.
//
impl Equatable for Cat {

    pub fun equals(_ other: Self): Bool {
        // Cats are equal if their identifier matches.
        //
        return other.id == self.id
    }
}

Cat(1) == Cat(2) // is false
Cat(3) == Cat(3) // is true
```

`Hashable` Interface

> 🚧 Status: The `Hashable` interface is not implemented yet.

A hashable type is a type that can be hashed to an integer hash value, i.e., it is distilled into a value that is used as evidence of inequality. Types are hashable when they implement the `Hashable` interface.

Hashable types can be used as keys in dictionaries.

Hashable types must also be equatable, i.e., they must also implement the `Equatable` interface. This is because the hash value is only evidence for inequality: two values that have different hash values are guaranteed to be unequal. However, if the hash values of two value are the same, then the two values could still be unequal and just happen to hash to the same hash value. In that case equality still needs to be determined through an equality check. Without `Equatable`, values could be added to a dictionary, but it would not be possible to retrieve them.

Most of the built-in types are hashable, like booleans and integers. Arrays are hashable when their elements are hashable. Dictionaries ar hashable when their values are equatable.

Hashing a value means passing its essential components into a hash function. Essential components are those that are used in the type's implementation of `Equatable`.

If two values are equal because their `equals` function returns true, then the implementation must return the same integer hash value for each of the two values.

The implementation must also consistently return the same integer hash value during the execution of the program when the essential components have not changed. The integer hash value must not necessarily be the same across multiple executions.

```
struct interface Hashable: Equatable {
    pub hashValue: Int
}
```

```
// Declare a structure named `Point` with two fields
// named `x` and `y` that have type `Int`.
//
struct Point {

    pub(set) var x: Int
    pub(set) var y: Int

    init(x: Int, y: Int) {
        self.x = x
        self.y = y
    }
}

// Implement the interface `Equatable` for the type `Point`,
// to allow points to be compared for equality.
//
impl Equatable for Point {

    pub fun equals(_ other: Self): Bool {
        // Points are equal if their coordinates match.
        //
        // The essential components are therefore the fields
        // `x` and `y`, which must be used in the `Hashable`
        // implementation.
        //
        return other.x == self.x
            && other.y == self.y
    }
}

// Implement the interface `Equatable` for the type `Point`.
//
impl Hashable for Point {

    pub synthetic hashValue: Int {
        get {
            var hash = 7
            hash = 31 * hash + self.x
            hash = 31 * hash + self.y
            return hash
        }
    }
}
```

Attestations

> 🚧 Status: Attestations are not implemented yet.

Attestations are values that proof ownership. Attestations can be created for resources and reflect their current state, which is read-only. They cannot be stored.

Attestations are useful in cases where ownership of some asset/resource should be demonstrated to potentially untrusted code.

As an analogy, a bank statement is a proof of ownership of money. However, unlike a bank statement, an attestation is "live", i.e. it is not just a snapshot at the time it was created, but it reflects the current state of the underlying resource.

Attestations can only be created from resources, i.e., they cannot be forged by parties who do not have ownership of the resource, and can be safely handed to untrusted parties.

Attestations of resources are created using the `@` operator. Attestation types have the name of the resource type, prefixed with the `@` symbol.

```
// Declare a resource named `Token`
//
resource Token {}

// Create a new instance of the resource type `Token`.
```

```
//
let token <- create Token()

// Declare a constant named `attestation` that has the attestation type `@Token`,
// and has an attestation for the token value as its initial value
//
let attestation: @Token = @token
```

Like resources, attestations are associated with an account.

Accounts

> 🚧 Status: Accounts are not implemented yet.

```
struct interface Account {
    pub init(at address: Address)
}
```

Account Storage

Accounts have a `storage` object which contains the stored values of the account.

Only **resources** can be stored.

Stored values are keyed by a **type**, i.e., the access operator `[]` is used for both reading and writing stored values.

The stored value must be a subtype of the type it is keyed by.

```
// Declare a resource named `Counter`
//
resource Counter {
    pub var count: Int

    pub init(count: Int) {
        self.count = count
    }
}

let account: Account = // ...

// Create a new instance of the resource type `Counter` and move it
// into the storage of the account.
//
// The type `Counter` is used as the key to refer to the stored value
//
account.storage[Counter] <- create Counter(count: 0)
```

Usage of External Types

> 🚧 Status: The usage of external types is not implemented yet.

It is possible to use external types through the `using` keyword, followed by the type name, the `from` keyword, and the address literal where the declaration is deployed.

```
// Declaration for an interface named `Counter`,
// declared and deployed externally
//
resource interface Counter {
    pub count: Int
    pub fun increment(_ count: Int)
}

// Use the type `Counter` from address
```

```
// 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d.
//
using Counter from 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d
```

Transactions

Transactions are objects that are signed by one or more accounts and are sent to the chain to interact with it.

Transactions have three phases: Preparation, execution, and postconditions.

The preparer acts like the initializer in a composite data type, i.e., it initializes fields that can then be used in the execution phase. The preparer has the permissions to read and write to storage of all signer accounts.

Transactions are declared using the `transaction` keyword. The preparer is declared using the `prepare` keyword and the execution phase is declared using the `execute` keyword. The `post` section can be used to declare postconditions.

```
transaction {

    // Optional: fields, which must be initialized in `prepare`

    // The preparer needs to have as many account parameters
    // as there are signers for the transaction
    //
    prepare(signer1: Account) {
        // ...
    }

    execute {
        // ...
    }

    post {
        // ...
    }
}
```

Deployment

Transactions can deploy resources and resource interfaces.

```
// Declare a resource interface for a fungible token.
//
// It requires implementing types to provide a resource named `Vault`,
// which needs to implement the interfaces `Provider` and `Receiver`
//
resource interface FungibleToken {

    pub resource interface Provider {

        pub fun withdraw(amount: Int): <-Vault {
            pre {
                amount > 0:
                    "withdrawal amount must be positive"
            }
            post {
                result.balance == amount:
                    "incorrect amount returned"
            }
        }
    }

    pub resource interface Receiver {
        pub fun deposit(vault: <-Vault)
    }

    pub resource Vault: Provider, Receiver {

```

```
        pub balance: Int {
            get {
                post {
                    result >= 0:
                        "Balances are always non-negative"
                }
            }
        }

        init(balance: Int) {
            post {
                self.balance == balance:
                    "the balance must be initialized to the initial balance"
            }
        }

        pub fun withdraw(amount: Int): <-Self {
            pre {
                amount <= self.balance:
                    "insufficient funds: the amount must be smaller or equal to the balance"
            }
            post {
                self.balance == before(self.balance) - amount:
                    "Incorrect amount removed"
            }
        }

        pub fun deposit(vault: <-Self) {
            post {
                self.balance == before(self.balance) + vault.balance:
                    "the amount must be added to the balance"
            }
        }
    }
}
```

Transactions can refer to local code with the `using` keyword, followed by the name of the type, the `from` keyword, and the string literal for the path of the file which contains the code of the type.

The preparer can use the signing account's `deploy` function to deploy the resource interface.

Once deployed, the resource interfaces is available in the account's `types` object.

The `publish` operator is used to make the resource interface type publicly available.

```
// Execute a transaction which deploys the code for
// the resource interface `FungibleToken`, and makes
// the deployed type publicly available
//
transaction {

    // Refer to the resource interface type `FungibleToken`
    // in the local file "FungibleToken.bpl"
    //
    using FungibleToken from "FungibleToken.bpl"

    prepare(signer: Account) {
        // Deploy the  resource interface type `FungibleToken`
        // in the signing account
        signer.deploy(FungibleToken)

        // Make the deployed type publicly available
        //
        publish signer.types[FungibleToken]
    }
}
```

Just like resource interfaces it is possible to deploy resources.

```
// Declare a resource named `ExampleToken` which implements
// the resource interface `FungibleToken`
//
resource ExampleToken {}

impl FungibleToken for ExampleToken {

    resource Vault {
        pub var balance: Int

        init(balance: Int) {
            self.balance = balance
        }

        pub fun withdraw(amount: Int): <-Vault {
            self.balance = self.balance - amount
            return create Vault(balance: amount)
        }

        pub fun deposit(_ token: <-Vault) {
            self.balance = self.balance + amount
            destroy token
        }
    }

    impl Receiver for Vault {}
    impl Provider for Vault {}
}
```

```
// Execute a transaction which deploys the code for
// the resource `ExampleToken`, and makes the deployed
// type publicly available
//
transaction {

    using ExampleToken from "ExampleToken.bpl"

    prepare(signer: Account) {
        signer.deploy(ExampleToken)
        publish signer.types[ExampleToken]
    }
}
```

Interacting with Deployed Resources

Transactions can also refer to deployed code with the `using` keyword and the address of the account which contains the publicly available type.

In addition to storing resources it is also possible to store references to **stored** resources or even other references. References can only be keyed by (and therefore accessed through) **resource interfaces**.

References are created by using the `&` operator, followed by the stored resource or reference, the `as` operator, and the resource interface type.

```
// Execute a transaction which creates a new example token vault
// for the signing account
//
transaction {
    // Refer to the resource type `ExampleToken` deployed
    // at example address 0x42
    //
    using ExampleToken from 0x42

    prepare(signer: Account) {
        // Create a new example token vault for the signing account.
        //
        // NOTE: the vault is not publicly accessible
        //
```

```
        signer.storage[ExampleToken.Vault] <- create ExampleToken.Vault()

        // Store two storage references in the signing account:
        // One reference to the stored vault, keyed by the resource
        // interface `Provider`, and another reference to the stored vault,
        // keyed by the resource interface `Provider`
        //
        signer.storage[ExampleToken.Provider] =
            &signer.storage[ExampleToken.Vault] as ExampleToken.Provider

        signer.storage[ExampleToken.Receiver] =
            &signer.storage[ExampleToken.Vault] as ExampleToken.Receiver

        // Publish only the receiver so it can be accessed publicly.
        //
        // NOTE: neither the vault nor the publisher are published
        //
        publish signer.storage[ExampleToken.Receiver]
    }
}
```

```
// Execute a transaction which sends five coins from one account to another.
//
// The transaction fails unless there is a `ExampleToken.Provider` available
// for the sending account and there is a public `ExampleToken.Receiver`
// available for the recipient account.
//
// Only a signature from the sender is required.
// No signature from the recipient is required, as the receiver
// is published/publicly available (if it exists for the recipient)
//
transaction {
    // Refer to the resource type `ExampleToken` deployed
    // at example address 0x42
    //
    using ExampleToken from 0x42

    let sentFunds: ExampleToken.Vault

    prepare(signer: Account) {
        // Get the stored provider for the signing account.
        //
        // As the access is performed in the preparer,
        // the unpublished reference `ExampleToken.Provider`
        // can be accessed (if it exists)
        //
        let provider <- signer.storage[ExampleToken.Provider]

        // Withdraw five coins (as a vault) from the provider
        // and move it into the field `sentFunds`
        //
        self.sentFunds <- provider.withdraw(amount: 5)
    }

    execute {
        // The recipient account
        //
        let recipient: Account = // ...

        // Get the stored receiver for the recipient account
        //
        let receiver <- recipient.storage[ExampleToken.Receiver]

        // Deposit the amount withdrawn from the signer
        // in the recipient's vault through the receiver
        //
        receiver.deposit(vault: <-self.sentFunds)
    }
}
```

Built-in Functions

`panic`

```
fun panic(_ message: String): Never
```

Terminates the program unconditionally and reports a message which explains why the unrecoverable error occurred.

Example

```
let optionalAccount: Account? = // ...
let account = optionalAccount ?? panic("missing account")
```

`assert`

```
fun assert(_ condition: Bool, message: String)
```

Terminates the program if the given condition is false, and reports a message which explains how the condition is false. Use this function for internal sanity checks.

The message argument is optional.

Open questions

Shared Mutable State

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/264

Do we need a means to model shared mutable state, i.e. reference types?

Error Handling

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/65

How should errors be handled? Should we make the distinction between exceptional and unexceptional errors? Should we provide means to handle errors (exception handlers)? Should exceptions be values and typed? Would a value-less throw-catch/panic-recover be suitable for a first version?

On-chain Storage

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/101

How do we store programs on-chain?

Fixed-Point Numbers and Arithmetic

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/67

We do not allow floating-point numbers. Should we add fixed-point arithmetic to support fractional numbers?

Enums with Exhaustiveness Check

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/59

Should we allow the definition of enumerations? Is an exhaustiveness check in switch statements useful and feasible?

Switch-Case Statement

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/60

Should we add a switch-case statement? What kind of pattern matching should it support?

Distinct/New Types

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/45

Should we add support for distinct types, i.e., types that are derived from an existing type, but are not compatible with them?

Set Data Structure

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/266

Should the standard library provide a set data structure?

Generics

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/44

Should we add generics? In what form? Is it OK to add them in a later version?

Calls of Pure Functions in Preconditions and Postconditions

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/70

It might be useful to call pure functions preconditions and postconditons. How do we ensure preconditions and postconditions are side-effect free?

Late Initialization of Variables and Constants

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/71

Currently we do not allow variables and constants to be initialized after they are declared. This improves readability, as the reader of the code can always be sure the initial value can be found where the variable or constant was declared, not somewhere else in the code following the declaration.

Should we allow the late initialization of variables and constants?

Are there good examples for cases where late initialization would be useful or even essential?

Arbitrary Argument Order Based on Argument Labels

> ➡️ https://github.com/dapperlabs/bamboo-node/issues/76

Should arbitrary argument order be supported in function calls as long as the argument labels match?