2b. Variables, Types, and Operators

Martin Alfaro
PhD in Economics

INTRODUCTION

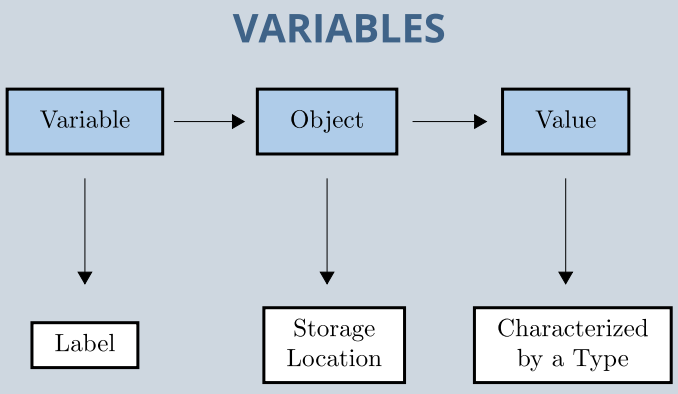
This section introduces the concepts of **variables** and **types**. We'll also present the notion of **operators**, focusing on their syntax. To ensure a smooth learning experience, I've minimized the reliance on objects that we haven't covered yet. The only one introduced is vectors, whose elements are enclosed in brackets (e.g., `[1,2,3]`).

VARIABLES

When a program is executed, the computer stores data in RAM (Random Access Memory). Each piece of data in RAM is referred to as an **object** and is assigned a unique memory address. These addresses are typically represented in hexadecimal format (e.g., `0x00007e0966dc0dd0`).

Furthermore, every object is associated with a value and a type. **Values** represent the actual data contained within the object. In turn, **types** define the nature of the data stored, providing the computer with critical information for handling the object internally.

Since directly referencing memory addresses would be impractical, we instead define **variables**. They act as human-readable **labels** for objects, simplifying our interaction with the data. Linking objects with a variable relies on the so-called **assignment operator** `=`, which creates a binding between the variable name and the object's memory location. This allows developers to interact with data through symbolic identifiers, rather than raw memory locations.

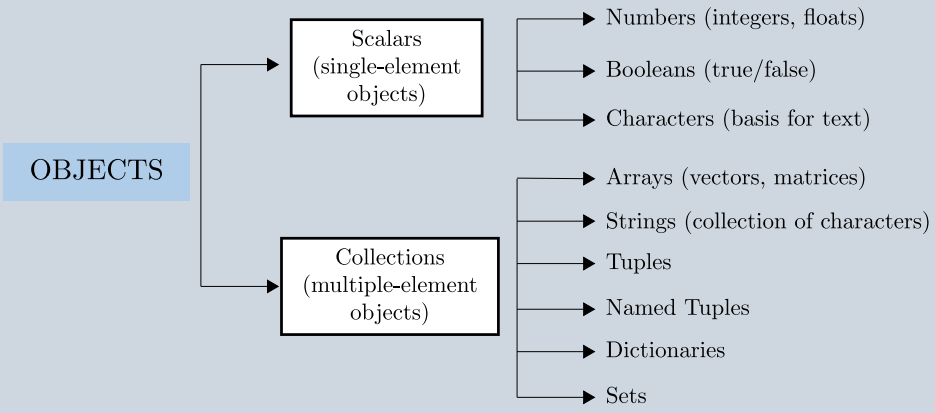


To illustrate this process, let's consider executing the command `x = "Hello"`. When this is run, several actions take place. First, the computer reserves a memory location to store the object (e.g., at address `0x1234`). This object is then assigned the specific value `"Hello"`, which in Julia is an instance of the type `String`.

At the same time, we're assigning the label `x` to this object, so that `x` points to the memory address `0x1234`. This means that, every time we use `x` in our code, we're actually accessing the object stored at memory address `0x1234`. It's important to note that `x` isn't the object or the value itself, but rather a reference to the memory allocation `0x1234`. This explains why we can define multiple labels to reference exactly the same object (i.e., the same memory address).

CLASSIFICATION OF OBJECTS

Objects are typically characterized according to the number of elements they contain: **scalars** refer to single-element objects, and **collections** refer to objects containing multiple elements. Below, we outline some objects encompassed in each category.



NAMES FOR VARIABLES

Variable names in Julia can be defined using Unicode characters, thus offering a wide range of possibilities. This feature enables you to use Greek letters, Chinese characters, symbols, and even emoticons. Underscores `_` are also permitted, which can be helpful for separating words within variable names (e.g., `intermediate_result`). Importantly, names are case-sensitive, so that `bar` and `Bar` are treated as two distinct variables.

```
a          = 2
A          = 2       # variable `A` is different from `a`

new_value = 2        # underscores allowed

β          = 2       # Greek letters allowed

中國        = 2       # Chinese characters allowed

x̄          = 2       # decorations allowed
x₁         = 2
ẋ          = 2

🐒         = 2       # emoticons allowed
```

> ⚠️ **Warning!**
>
> **Julia doesn't let you delete variables**. Once a variable is created, it remains in memory until the program terminates. If a variable is taking up too much memory, you can free up space by reassigning it to a smaller object.

> ❗ **Notation for Variable Names**
>
> Julia's developers adopt the convention of using **snake-case notation for variable names**. This format consists of lowercase letters and numbers, with words separated by underscores. (e.g., `snake_case_var1`). Note that this is only a convention, not a language's requirement.

UPDATING VARIABLES

It's possible to assign a new value to a variable using the variable itself. This approach is referred to as **updating a variable**.

```
x  = 2

x  = x + 3    # 'x' now equals 5
```

Julia offers a concise syntax for updating values, based on the so-called **update operators**. They're implemented by prefixing the assignment operator `=` with the operator to be applied, as demonstrated below.

```
x  = 2

x  = x + 3
x += 3        # equivalent

x  = x * 3
x *= 3        # equivalent

x  = x - 3
x -= 3        # equivalent
```

TYPES

Before diving into the intricacies of Julia, it's essential to familiarize yourself with the basics of its type system. This initial overview will only provide the minimum necessary for the upcoming chapters. A comprehensive treatment of types, including their role in performance optimization, will be deferred to Part II of this website. For now, the focus is on core definition and notation.

> **Notation for Types**
>
> Julia's developers adopt the convention of using **CamelCase** notation for denoting types, where every first letter is capitalized (e.g., `MyType`). Note that this is only a convention, not a language's requirement.

As previously mentioned, types define the nature of values, specifying all the information the computer needs for their storage and manipulation. To better illustrate types, let's split the discussion in terms of scalars and collections.

Common numeric types for scalars include `Int64` for integers, `Float64` for decimal numbers, and `Bool` for binary values (`true` and `false` values). Likewise, the type `Char` represents individual characters, serving as the building block for the `String` type. `String` is the standard type in Julia for representing text, and its values consist of sequences of characters.

Collections, on the other hand, often require **type parameters** for a full characterization of their types. These parameters can be incorporated into any type, and have the goal of providing additional information about its contents.

Type parameters are denoted using `{}` after the type's name. For instance, the type `Vector{Int64}` indicates that the collection represents a vector exclusively containing elements of type `Int64` (e.g., `[2, 4, 6]`). Here, `Int64` serves as a type parameter. Note that type parameters are optional and therefore can be omitted when not needed. Indeed, this is the case with the types for scalars mentioned above.

> **Type Annotations**
>
> You can explicitly declare the type of a variable by using **type annotations**, via the `::` operator. For example, `x::String` ensures

> that `x` can only store string values throughout the program, resulting in an error if you attempt to reassign `x` with a value of a different type.

CONCRETE TYPES AND ABSTRACT TYPES

In Julia, **types are organized hierarchically**, creating relations of supertypes and subtypes. This hierarchy gives rise to the notions of abstract and concrete types.

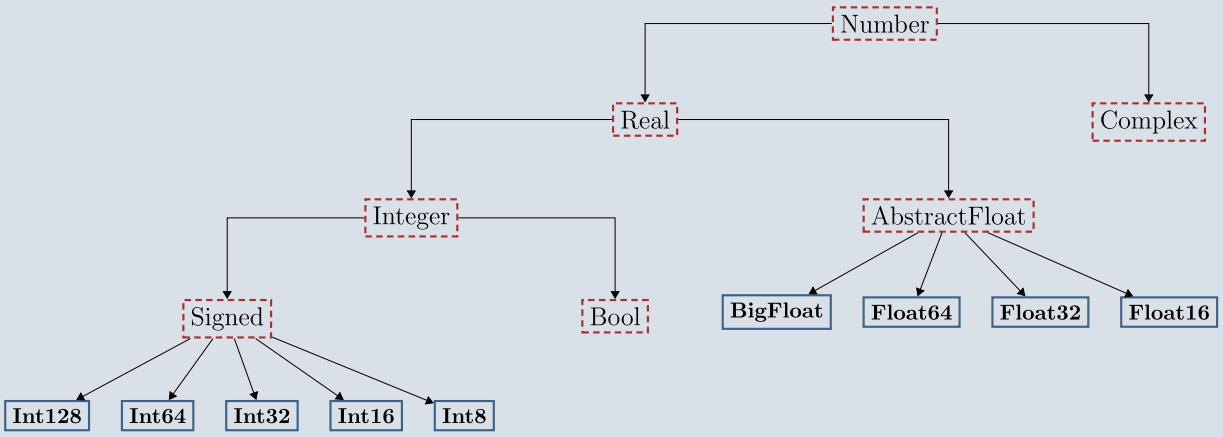
An **abstract type** is a set of types that serve as a parent to other types. The `Any` type in Julia is a prime example of abstract type. It acts as the root of the hierarchy, thus comprising all possible subtypes—by definition, every type in Julia is a subtype of `Any`.

In contrast, a **concrete type** is an irreducible unit, representing a terminal node in the hierarchy and therefore lacking subtypes. Concrete types include in particular primitive types, which represent the most fundamental types that computers use to perform calculations. Examples of primitive types are `Int64` and `Float64`, which directly map to low-level hardware representations.

Abstract types provide great flexibility for writing code. For example, the abstract type `Number` defined in Julia encompasses all possible numeric types (e.g., `Float64`, `Int64`, `Float32`). By declaring a variable as `Number`, programmers avoid unnecessarily constraining their programs to specific numeric representations or precision.

To demonstrate this hierarchy, we consider the concrete types comprised by `Number`. The names included in the table match the exact names in Julia. Note, nonetheless, that the full subtype hierarchy of `Number` is broader than the simplified representation presented.

EXAMPLE OF THE ABSTRACT TYPE "NUMBER"



Note: Dashed red borders indicate abstract types, while solid blue borders indicate concrete types.

OPERATORS

In programming, **operators** are symbols that represent operations performed on objects. They can be thought of as syntactic sugar for functions, as we'll see in the next chapters. In fact, almost all operators in Julia can be employed as functions.

For instance, the symbol `+` in `x + y` is an operator that performs the addition of `x` and `y`. Likewise, the symbols `x` and `y` are referred to as the **operands**, representing the operator's inputs to perform its calculation. *Operators follow specific syntax rules based on the number of operands they require*. Understanding this syntax will prove useful for several topics covered later on the website. Next, we define and illustrate the syntax through several examples. At this point, just focus on how operators are written, even if their specific functions are not yet clear.

- **Unary operators**: They take *one operand*, with the operator written to the left of it. Formally, their syntax is `<operator>x`, such as `√x` or `-x`.

- **Binary operators**: They take *two operands*, and the operator is written between them. Formally, their syntax is `x <operator> y`, such as `x + y` or `x^y` for x^y.

- **Ternary operators**: They take *three operands*. Formally, their syntax is `x <operator1> y <operator2> z`. Ternary operators are rare, which is why the specific operator `x ? y : z` is directly referred to as *the* ternary operator. We'll see that this operator performs a conditional evaluation, returning `y` if `x` is true and `z` returned otherwise.