1 Variables types

Programming (often) consists in the definition of functions that handle variables. The type of a variable defines the operations one can perform on this variable. Hence, the choice of each variable type has some important consequences for the entire program. One must think about proper variable types *before* writing any code.

1.1 Primitive variable types

Primitive variable types are types already implemented by the language itself. Most common are:

- Int: integers can be signed (i.e. possibly negative) or unsigned (greater that only). In the second case they are often called UInt, for Unsigned Integer.
- Float: similare to integers, float values exist as different flavours: Float16, Float32, Float64 (sometimes also called Double). A float can store numbers with a decimal part.
- Char: character, a type than can store 1 byte.
- String: sequence of characters used to store text
- Bool: True/False

1.2 Containers

A container is a data structure that unites different variables for an easy reading and writing access. Most commons are:

- vectors/arrays: index-like access
- Stacks: LIFO access (*Last In First Out*)
- Queues: FIFO access (*First In First Out*)
- Dictionaries: key/values access. Each key is unique.
- set: set of unique values

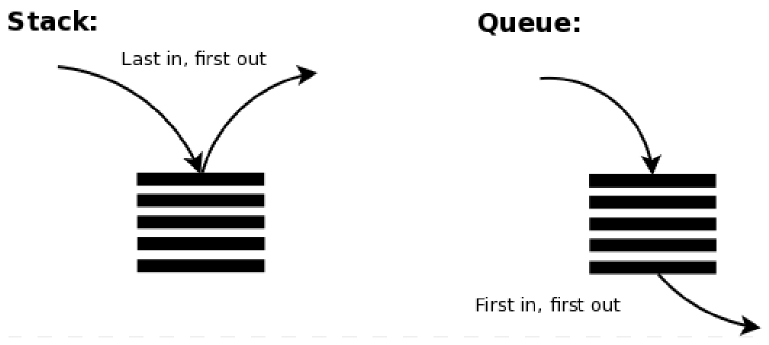

Figure 1: Stacks and queues (source: Martin Franěk, `quora.com`)

1.3 Object oriented programming

1.3.1 Definition

Some data types can handle references to different primitive variables in a common data structure. These variables are usually called **attributes**. **Object Oriented Programming** makes possible to define these types using **classes**. Each variable built from a class is called an **object**. It interacts with other variables through **methods**. **Inheritance** makes it possible for such a custom data type (*T2*) to have the properties (attributes, methods) of another one (*T1*).

1.3.2 Examples

First We can think about an **Official identity** type that stores the following information:

- Unique identifier: *Int*
- Name: *String*
- Height: *Float16*
- Parent(s) and siblings: containers of objects of type *Official identity*

One could also partly define **Official identity** from a type **Identity** (inheritance):

- Name: *String*

In that case, the "Name" attributed would be inherited from the parent object of type **Identity**. Then **Official identity** specifies **Identity**: every property of **Identity** is a property of **Official identity** (but not the other way).

Second In case of onheritance, one can **overload** some methods of the parent (see *park* in the example below)

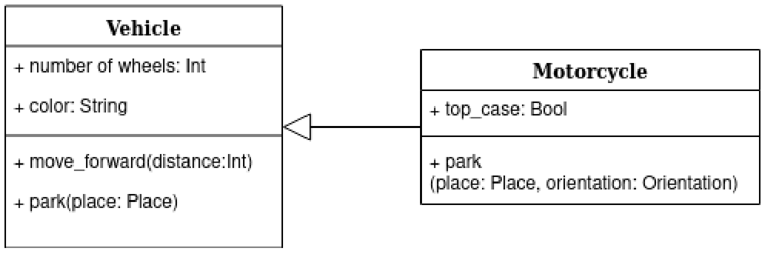

Figure 2: Inheritance as a UML description

2 Variable scope

2.1 Definition

Each language defines different variable scope rules. These rules state whether a variable created in a scope A can be accessed from a scope B. These rules are related to:

- Control flow structures
- Functions
- Local code imports

- ...

These rules makes it possible to use a common variable name in different places in the code, to refer to different variables. **Shadowing** refers to the fact of reusing for a different purpose an already defined variable name

2.2 Static and dynamic typing

Static typing consists in the manual assignment of a type to a variable. This variable will keep this type until the end of its use. Dynamic typing let the compiler (or interpreter) chose the variable type during run time.
In both cases, the variable has a known type during execution.

3 Mutability

The mutability of an object refers to the possibility of modifying its properties.

3.1 Copy and assignment

The assignment of a mutable object to another variable might lead to unexpected results. Indeed:

- The in-depth copy of an object makes a full duplicate of this object. (Figure 3).
- Conversely, an assignment of a mutable object to a variable makes the variable reference the memory content of the object (Figure 4).

In the first case, a modification on the first variable has no effect on the second. In the second case, if first is modified then second is modified too.
There exists an intermediate copy level between assignment and in-depth copy: shallow copy. If 'b' is a shallow copy of 'a', 'b' has a different memory address than 'a' (different objects), yet references of 'b' to mutables objects are the one of 'a'.

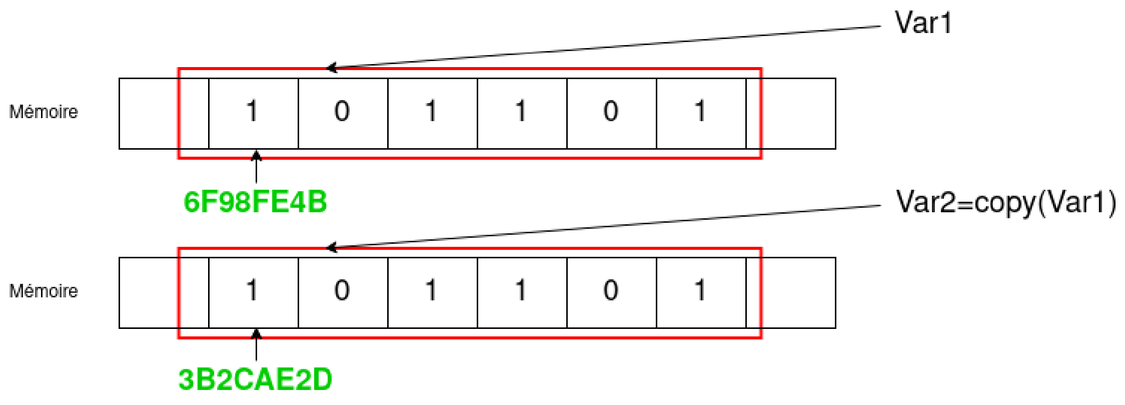

Figure 3: In-depth copy of a variable

3.2 Equality and identity

Some objects can be equal - i.e. same properties - withouth being the same - i.e. different memory address. The identity test makes it possible to check if 2 objets have the same identity. Using pseudo-code:

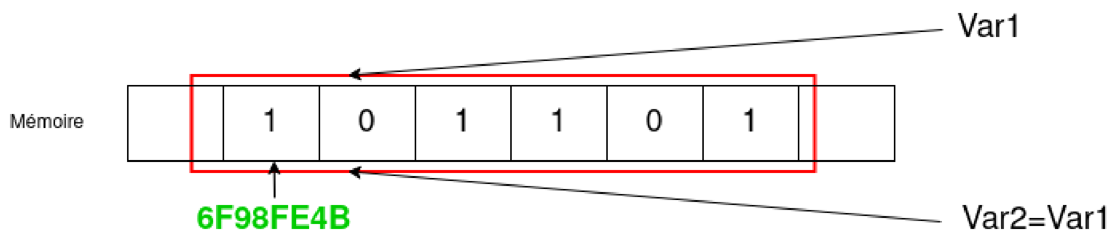

Figure 4: Reference to a variable

```
MyType var1 = MyType.constructor()
MyType var2 = InDepthCopy(var1)
AreIdentical(var1, var2)  # False
AreEqual(var1, var2)     # True
```