5b. Mutable and Immutable Objects

Martin Alfaro
PhD in Economics

INTRODUCTION

Objects in programming can be broadly classified into two categories: mutable and immutable. **Mutable objects** allow their elements to be modified, appended, or removed at will. They're designed for flexibility, with **vectors** constituting a prime example.

In contrast, **immutable objects** are inherently unchangeable: they prevent additions, removals, or modifications of their elements. A common example of immutable object is **tuples**. Immutability effectively locks variables into a read-only state, safeguarding against unintended changes. Moreover, it can result in potential performance gains, as we'll show in Part II of this website.

This section will be relatively brief, focusing solely on the distinctions between mutable and immutable objects. Subsequent sections will expand on their uses and properties.

> **Remark**
>
> A popular package called `StaticArrays` provides an implementation of **immutable vectors**. We'll explore this package in the context of high performance, as it greatly speeds up computations that involve small vectors.

EXAMPLES OF MUTABILITY AND IMMUTABILITY

To illustrate the consequences of immutability, the following examples attempt to modify existing elements of a collection. The examples rely on vectors as an example of a mutable object and tuples for immutable ones. Additionally, we present the case of strings as another example of immutable object. Recall that strings are essentially sequences of characters, usually employed to represent text.

```
x = [3,4,5]
```

```
julia> x[1] = 0

julia> x
3-element Vector{Int64}:
 0
 4
 5
```

```
x = (3,4,5)
```

```
julia> x[1] = 0
ERROR: MethodError: no method matching setindex!(::Tuple{Int64, Int64, Int64}, ::Int64,
::Int64)
```

```
x = "hello"
```

```
julia> x[1]
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)
julia> x[1] = "a"
ERROR: MethodError: no method matching setindex!(::String, ::Int64, ::Int64)
```

The key characteristic of mutable objects is their ability to modify existing elements. Moreover, mutability also commonly allows for the dynamic addition and removal of elements. In a subsequent section, we'll present various methods for implementing this functionality. For now, we simply demonstrate the concept by using the functions `push!` and `pop!`, which respectively add and remove an element at the end of a collection.

```
x = [3,4]

push!(x, 5)       # add element 5 at the end
```

```
julia> x
3-element Vector{Int64}:
 3
 4
 5
```

```
x = [3,4,5]

pop!(x)           # delete last element
```

```
julia> x
2-element Vector{Int64}:
 3
 4
```

```
x = (3,4,5)

pop!(x)           # error, just like push!(x, <some element>)
```

```
ERROR: MethodError: no method matching pop!(::Tuple{Int64, Int64, Int64})
```