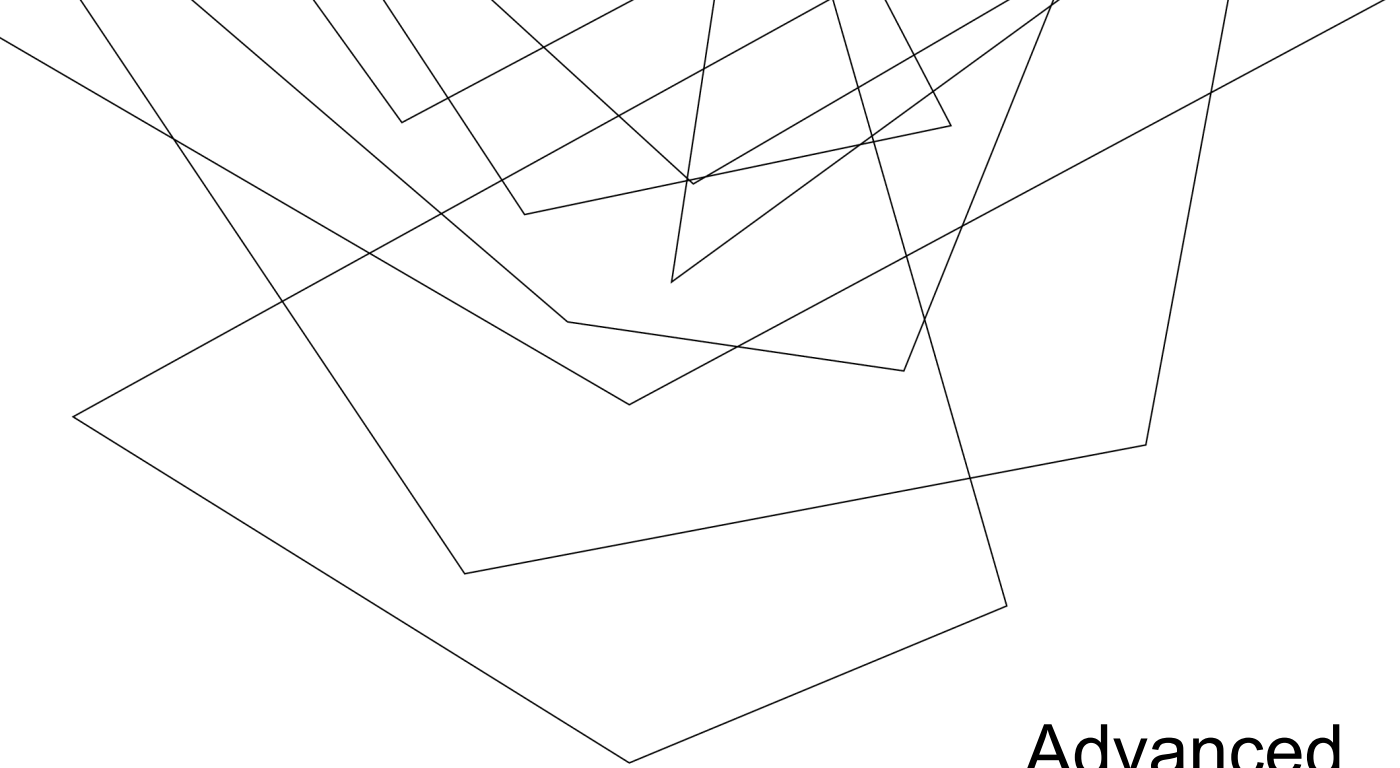
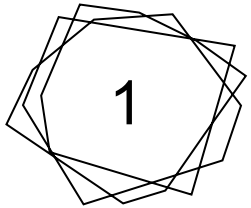


TÉCNICO LISBOA

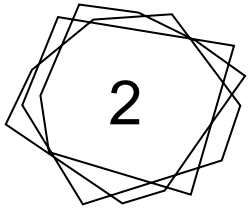
Advanced Programming

Ana Margarida Almeida - ist1102618
Tiago Farinha - ist1103327
Guilherme Marcondes - ist1104147

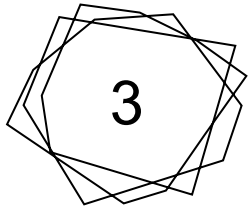
Implementations



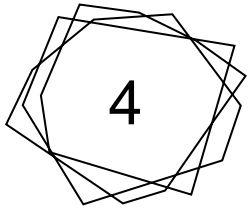
To Escape



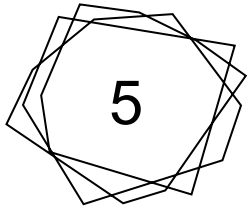
Handling



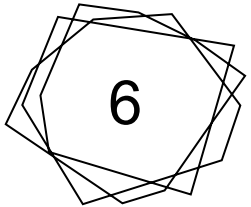
Invoke Restart



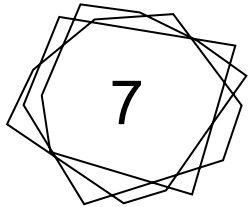
Available Restart



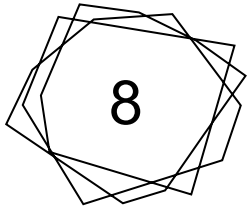
With Restart



Signal



Error



User-Handling Restarts

to_escape(func)

```
to_escape
```

```julia
struct Escape{T} <: Exception
    funcName::String
    result::T
end

function to_escape(func)
    methods = collect(Base.methods(func))
    escapeName = string(method_argnames(last(methods))[2])

    escape_func = function (args...)
        error(Escape(escapeName, args))
    end

    try
        return func(escape_func)
    catch e
        if e isa Escape && e.funcName == escapeName
            return length(e.result) == 1 ? e.result[1] : e.result
        else
            rethrow()
        end
    end
end

function method_argnames(m::Method)
    argnames = ccall(:jl_uncompress_argnames, Vector{Symbol}, (Any,), m.slot_syms)
    isempty(argnames) && return argnames
    return argnames[1:m.nargs]
end
```

Get the name of the argument associated with the function func.

Create an escape function that throws an `Escape` exception when called.

If, during the call to `func(escape_func)`, we catch an Escape exception and it matches the expected name we return the value stored in the exception.

- If not, we ignore it by rethrowing the obtained exception.

Note: The return value, stored in the exception, may be a tuple, which is why we first check if its size is 1. The function `method_argnames(m::Method)` is an auxiliary function to get the names of the arguments of a method.

handling(func, handlers...)

handling

```
currentHandlers = Vector{Vector{Pair{DataType, Function}}}()

function handling(func, handlers...)
    handlersList = Vector{Pair{DataType, Function}}()
    for handler in handlers
        push!(handlersList, handler)
    end

    global currentHandlers
    push!(currentHandlers, handlersList)

    res = func()

    pop!(currentHandlers)

    return res
end
```

The global variable `currentHandlers` is vector of vectors of handlers. The inner vectors contain the handers from each `handling` call, while the outer vector contains the inner vectors.

We are using a vector of vectors because we need to distinguish between handlers from the same `handling` function and handlers from different `handling` functions. If we were to use a vector of handlers we would lose this information and, as consequence, one exception could be handled by multiple handlers from the same `handling` function.

The `handling` function itself, will first store the handlers in the `currentHandlers` vector of vectors, then execute the invoke `func()` and remove the last handler vector from `currentHandlers`.

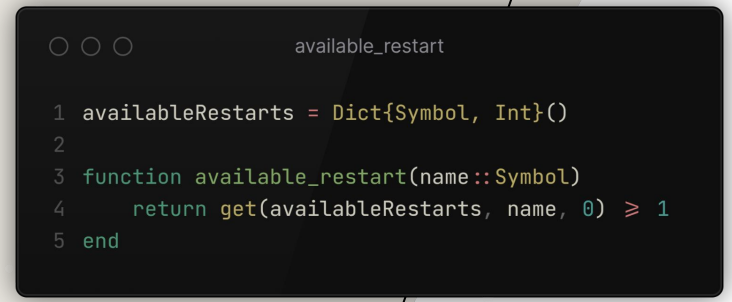
```
invoke_restart

1
2 struct UnavailableRestart{T} <: Exception
3     restart::Symbol
4     args::T
5 end
6
7 function invoke_restart(restart::Symbol, args...)
8     throw(UnavailableRestart(restart, args))
9 end
```

`invoke_restart(restart:Symbol, args...)`

To understand how we implemented restarts it is better to start explaining the `invoke_restart`.

The only thing it does is to throw - yes, throw and not error - an exception `UnavailableRestart`. It will store as arguments of the exception the restart symbol and the arguments used when the function was called.

`available_restart(name:Symbol)`



It's best to present the implementation of `available_restart` before introducing `with_restart`.

We'll use a dictionary where the key represents the restart symbol, and the value stores the current count of available restarts.

The `available_restart` function simply checks whether the value corresponding to the given key is greater than or equal to one.

with_restart(func, restarts...)

```
with_restart

function with_restart(func, restarts...)
    global availableRestarts
    for restart in restarts
        availableRestarts[restart.first] = get(availableRestarts, restart.first, 0) + 1
    end

    handlersLength = length(currentHandlers)

    try
        return func()
    catch e
        if e isa UnavailableRestart
            for restart in restarts
                if restart.first == e.restart
                    return restart.second(e.args...)
                end
            end
        end
        rethrow()
    finally
        for restart in restarts
        availableRestarts[restart.first] > 1 ?
            availableRestarts[restart.first] -= 1 :
            delete!(availableRestarts, restart.first)
        end

        while length(currentHandlers) ≠ handlersLength
            pop!(currentHandlers)
        end
    end
end
```

The first thing the `with_restart` function will do is to store the restarts in the `availableRestarts` dictionary.

If an `UnavailableRestart` exception is found while executing the `func` passed as argument, it will check if it matches some restart from `restarts`. If it does, then return the value defined in the corresponding restart definition.

- If no corresponding exception was found, ignore the exception, rethrowing it.

In the end, but before returning or rethrowing, it will remove the restarts from the `availableRestarts` dictionary.

It will also keep track of the current amount of vectors of handlers when this function is called. If we have a `handling` function inside a `with_restart` function and an exception is raised, then the removal of handlers from `currentHandlers` will not happen. We force that the size of `currentHandlers` is the same as when the `with_restart` function was invoked by popping elements from `currentHandlers` until it matches the initial size..

`signal(exception)`

signal

```
function signal(exception)
    global currentHandlers

    for handlerList in reverse(currentHandlers)
        for handler in handlerList
            if exception isa handler.first
                handler.second(exception)
                break
            end
        end
    end
end
```

The job of handling the exceptions is done in the `signal` function.

The signal is responsible applying the handlers based on the provided exception. Instead of throwing an exception, it sequentially invokes each handler for the given exception in reverse order to grant that the most specific handler is applied first.

It also makes sure that only one handler is applied per `handling` function by breaking the inner for loop, when a matching handler is found.

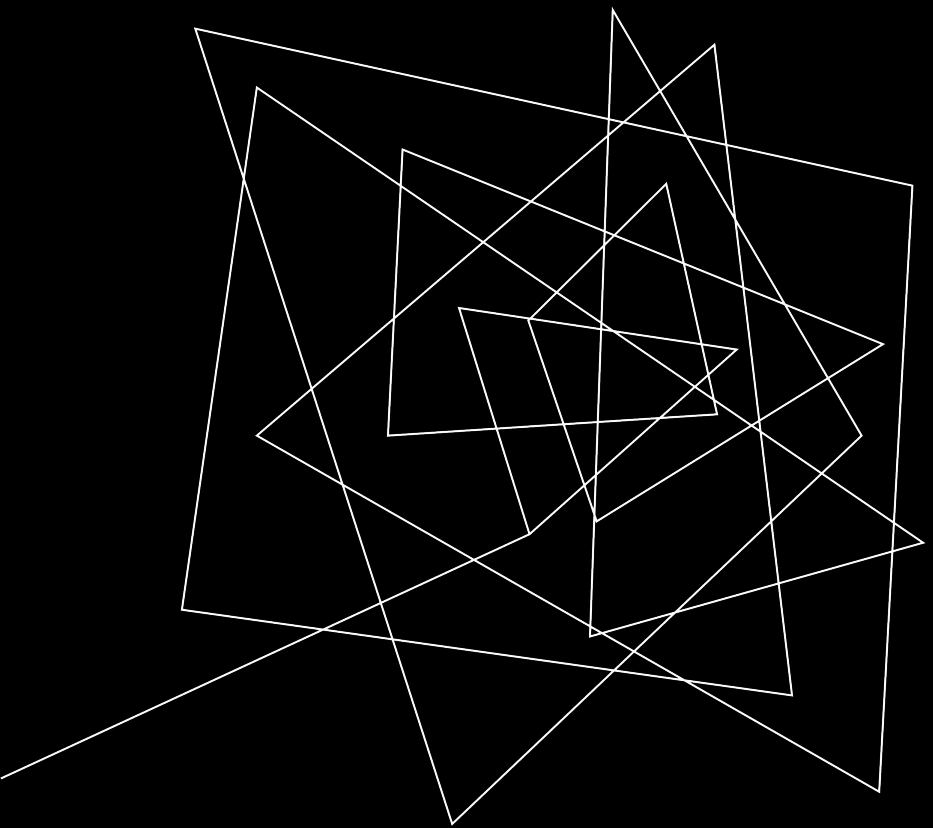
`error(exception)`

```
error

1 Base.error(exception) = begin
2     signal(exception)
3     throw(exception)
4 end
```

Overrides Julia's default error function.

First, it calls `signal(exception)` to process any defined handlers then it will throw the exception.

If the signal is handled by a `with_restart` function, the exception won't be thrown, as execution will be redirected to the catch block.



EXTENSION
User-Handling Restarts

```
An error occurred: DivisionByZero()
Available restarts:
 1) return_zero
 2) retry_using
 3) return_value

Select restart (1-3, q to quit): 1
You selected: 1
Write args (press Enter to skip):
0
```

`error(exception)`

```
error
1 Base.error(exception) = begin
2     interactive_signal(exception)
3     throw(exception)
4 end
```

Overrides `error` Exceptional.jl function.

It will do the same as `error` function from Exceptional.jl, but will invoke `interactive_signal` function instead of `signal` function.

interactive_signal

```
function interactive_signal(exception)
    global currentHandlers
    handled = false

    for handlerList in reverse(currentHandlers)
        for handler in handlerList
            if exception isa handler.first
                handler.second(exception)
                handled = true
                break
            end
        end
    end

    if !handled
        interactive_restart_prompt(exception)
    end
end
```

`interactive_signal(exception)`

We changed the the `signal` function to check if the exception was handled in the end.

If it was not handled the function `interactive_restart_prompt` will be called.

interactive_restart_prompt

```julia
function interactive_restart_prompt(exception)
    restarts = keys(availableRestarts)

    if isempty(restarts)
        # This print is commented to allow all the tests to pass
        # Uncomment for better user experience
        # println("\nNo restarts available. Rethrowing exception.")
        throw(exception)
    end

    println("\nAn error occurred: ", exception)
    println("Available restarts:")

    for (i, r) in enumerate(restarts)
        @printf "%2d) %-10s\n" i r
    end

    print("\nSelect restart (1-$(length(restarts)), q to quit): ")
    choice = readline()
    println("You selected: ", choice)

    if lowercase(choice) == "q"
        println("Quitting. Rethrowing exception.")
        throw(exception)
    end

    idx = parse(Int, choice)
    if 1 ⩽ idx ⩽ length(restarts)
        print("Write args (press Enter to skip): ")
        args = readline()
        parsed_args = tryparse(Int, args)
        if !isnothing(parsed_args)
            args = parsed_args
        end

        restart = collect(restarts)[idx]
        return invoke_restart(restart, args...)
    else
        println("Invalid choice. Rethrowing exception.")
        throw(exception)
    end
end
```

```
interactive_restart_prompt(
    exception
)
```

First of all we will get the symbols from all of the available restarts. If there are no available restarts then we don’t have any options to show to the user and just throw the exception.

If there are available restarts we will show them to the user and ask the user to choose one restart, while also providing the option to just throw the exception.

With the option selected, the user is then asked to write the arguments for the restart.

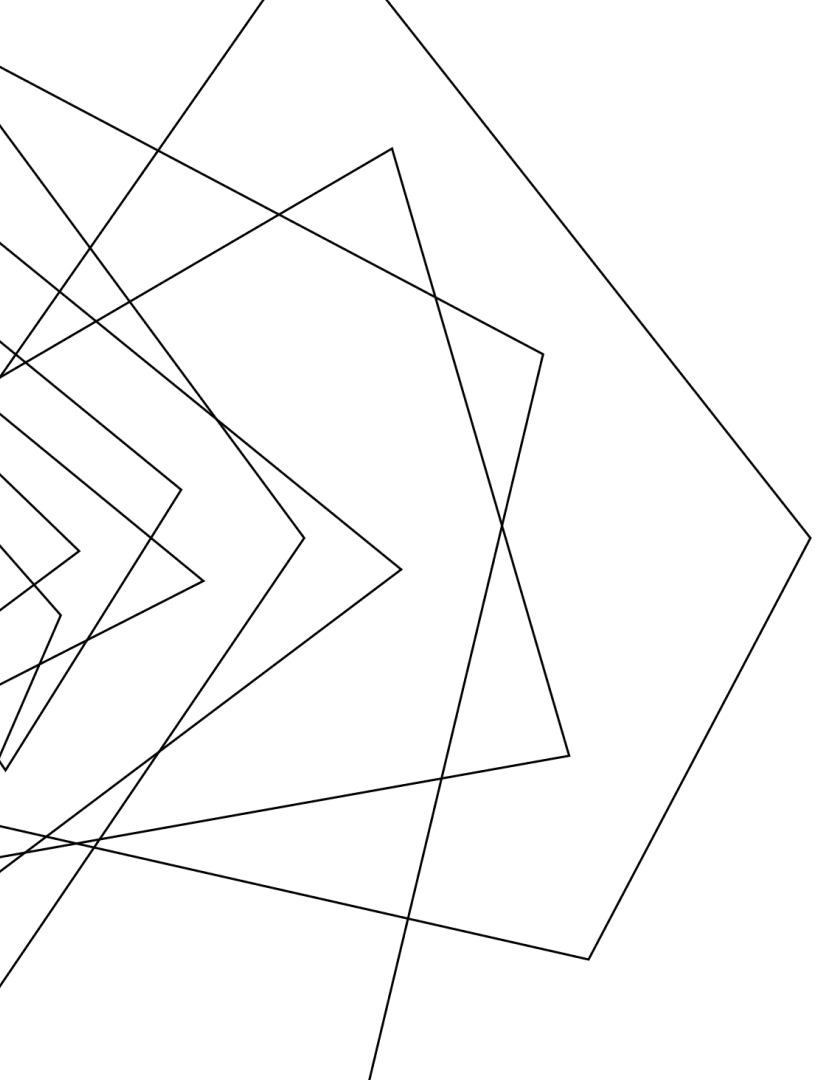
Finally, the function `invoke_restart` will be invoked with the chosen restart and arguments.

Tests

We have developed a total of 40 tests to thoroughly validate our project and ensure it meets the expected functionality.
These tests cover various aspects of signaling, handling, and restarts, verifying that exceptions are correctly processed and recovery mechanisms behave as intended.
By implementing a comprehensive test suite, we have strengthened the reliability of our solution and confirmed that it aligns with the expected behavior outlined in our design.

restart/test20.jl

```
# Test if outer with_restart returns to the right place after an error is found in inner with_restart

handling(DivisionByZero ⇒ (x) → invoke_restart(:return_one)) do
    println(
        with_restart(:return_one ⇒ () → 1) do
            with_restart(:return_zero ⇒ () → 0) do
                error(DivisionByZero())
            end + 3
        end
    )
end
```



Thank You For Your Attention!

Ana Margarida Almeida - ist1102618
Tiago Farinha - ist1103327
Guilherme Marcondes - ist1104147