ADVANCED EXCEPTION HANDLING IN JULIA

Torbjørn Rolstad (112882), Erlend Veire (112626), Petter Stokkeland (112675), and Ole Erik Nordtømme (112839)

Introduction

This project implements a flexible and powerful condition system in Julia, inspired by Common Lisp

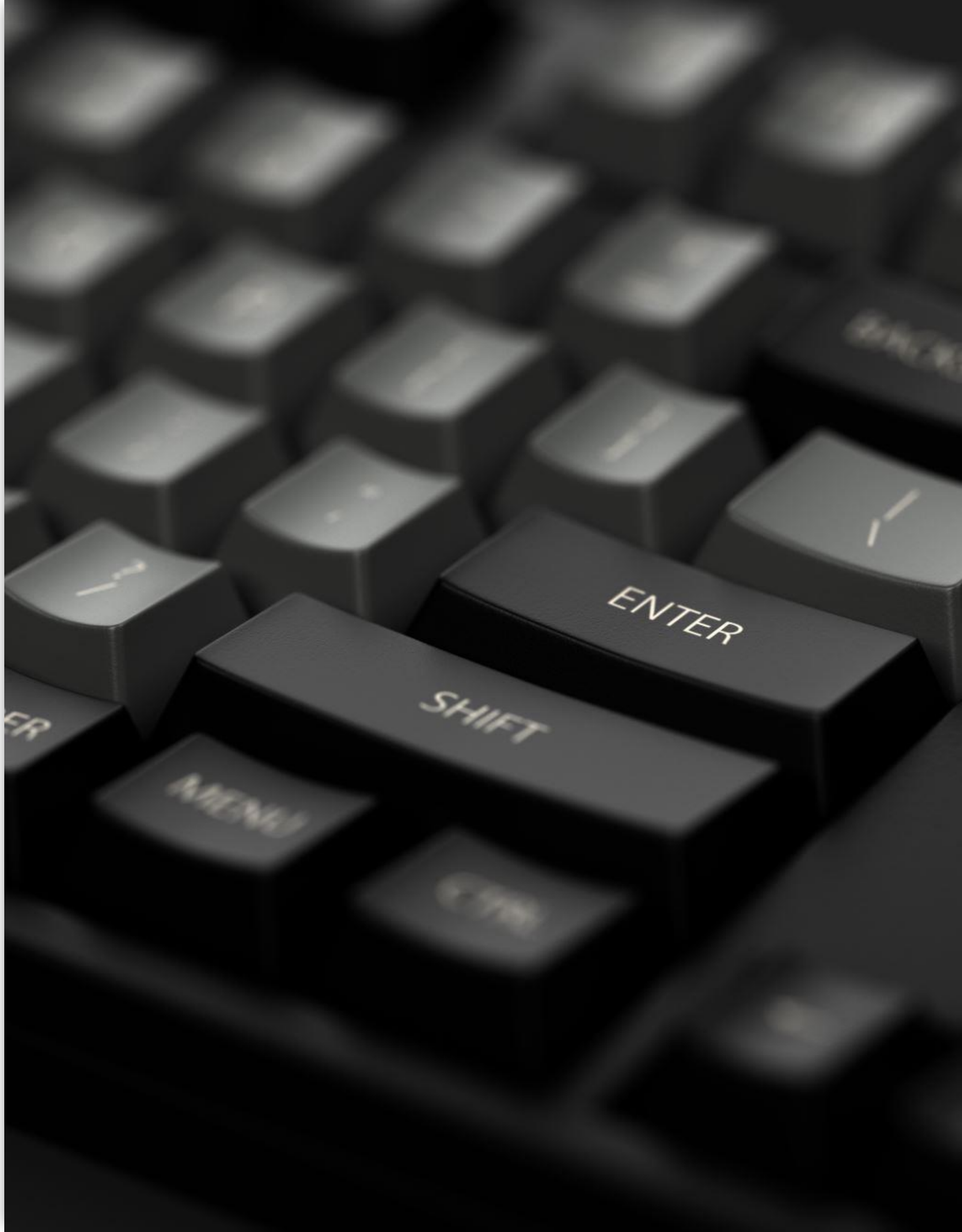
It goes beyond traditional try/catch by providing structured exception handling, non-local exits, and restartable recovery strategies

Julia’s built-in exception handling is effective but limited to propagation and catching

More sophisticated mechanisms (like restarts and signals) allow for customized, dynamic error handling that does not always terminate execution

Key Features

- Traditional exception handling with **handling**
- Error signaling using **error**
- Non-local exits via **to_escape**
- Named recovery strategies with **with_restart**
- Restart invocation using **invoke_restart**
- Introspection of restarts using **available_restarts**
- Signal processing with **signal**



Why these features?

- More control over error propagation
- Fine-grained recovery options based on context
- Separation of error detection and handling logic

Exception Handling

- Defines handlers dynamically
- Handlers can:
  - *Return a value and stop execution*
  - *Return nothing and allow propagation*
  - *Transfer control using to_escape or restarts*

```
handling(DivisionByZero => (c)->println("I saw it too")) do
    handling(DivisionByZero => (c)->println("I saw a division by zero")) do
        reciprocal(0)
    end
end
```

Function handling

- Uses a global signal_handlers registry, keyed by exception type
- handling dynamically registers handlers upon entry and deregisters them upon exit (using finally for robustness)
- Wraps provided handlers to integrate with the signal mechanism (checking for true return)
- Includes a standard try/catch block to intercept exceptions within its dynamic scope
- Separates handler definition from the main execution logic (func).
- Enables multiple, type-specific handlers within the same dynamic scope (dynamic dispatch)
- The register/deregister pattern ensures handlers are only active for the duration of the handling block, preventing interference across unrelated code sections
- The finally clause guarantees cleanup, avoiding dangling handlers in the global registry

```julia
function handling(func, handlers...)
    handler_ids = []
    for (exception_type, handler) in handlers
        wrapped_handler = (e) -> begin
            result = handler(e)
            return result !== nothing
        end
        id = register_signal_handler(exception_type, wrapped_handler)
        push!(handler_ids, (exception_type, id))
    end
    try
        return func()
    catch e
        for (exception_type, handler) in handlers
            if e isa exception_type
                result = handler(e)
                if result !== nothing
                    return result
                else
                    rethrow(e)
                end
            end
        end
        rethrow(e)
    finally
        for (exception_type, id) in handler_ids
            remove_signal_handler(exception_type, id)
        end
    end
end
```

Key benefits with handlers

More structured and reusable than try/catch

Allows multiple handlers for different error types

- and multiple handlers for the same error/exception types

Enables cleaner and more readable code

Error

- A built-in mechanism in Julia to signal exceptional conditions
- Immediately halts execution and raises an exception
- Can be handled dynamically using the handling system
- Instead of using throw, we use error(exception)
- Works with custom exception types
- Handlers can catch and process the error before propagating it

```julia
function reciprocal(x)
    x == 0 ? error(DivisionByZero()) : 1/x
end
```

Benefits with error

More structured and reusable than throw

Works seamlessly with the dynamic handler system

Ensures error conditions are properly reported and processed

Function error

- Overrides Base.error(exception)
  - Internally, it simply calls throw(exception)
- Acts as the primary way to trigger the mandatory handling flow
- Exceptions raised by error must be caught by an appropriate handling block or invoke a restart, if not, they propagate like standard unhandled exceptions

```
function Base.error(exception)
    throw(exception)
end
```

Non-local Exits

- Provides a controlled escape mechanism
- Allows jumping out of deeply nested functions
- Prevents unnecessary execution of code after an error
- Avoids messy and nested error handling
- Useful for early exits in error-prone computations
- Non-local transfer of control can stop propagation of signals, can define custom exit points
- Works well with exception handling and restart strategies

```
to_escape() do exit
    handling(DivisionByZero =>
            (c)->println("I saw it too")) do
        handling(DivisionByZero =>
                (c)->(println("I saw a division by zero");
                    exit("Done"))) do
            reciprocal(0)
        end
    end
end
```

Function to_escape

- Generates a unique context_id (Symbol) for each to_escape block
- Provides an escape_func to the user's code (func). This function closes over the context_id
- Calling escape_func throws a specific EscapeException containing the context_id and an optional return value
- The catch block specifically looks for EscapeException matching the unique context_id, preventing accidental capture by nested or unrelated to_escape blocks
- The context_id ensures the escape is caught only by its corresponding to_escape block, providing isolation
- Passing an explicit escape_func makes the capability clear and controlled within the user code
- Offers a structured alternative to goto or manually passing exit flags through multiple function layers

```julia
function to_escape(func)
    context_id = gensym("escape_context")
    escape_func = (value=nothing) -> throw(EscapeException(context_id, value))
    try
        return func(escape_func)
    catch e
        if e isa EscapeException && e.context_id == context_id
            return e.value
        else
            rethrow(e)
        end
    end
end
```

Restart System

- Separates error detection from recovery strategies
- Named restarts remain available after exceptions
- Allows retrying or choosing different recovery actions
- More flexibility than catch blocks
- Avoids rethrowing exceptions unnecessarily
- Separates "what can be done" (restarts defined) from "what went wrong" (restarts handled)

```
handling(DivisionByZero => (c)->invoke_restart(:return_zero)) do
    reciprocal(0)
end
```

Function with_restart

- Generates a unique context_id for the restart context
- Registers the provided (name, function) pairs in the global restart_registry, associated with the context_id
- Executes the user's code (func) within a try/catch block
- The try/catch block only rethrows the exception. Its purpose is solely to manage the lifetime of the restart registration
- Creates a dynamic environment where specific recovery options are registered
- Relies on global restart_registry to make these restarts discoverable by invoke_restart and available_restart

```
function with_restart(func, restarts...)
    context_id = gensym("restart_context")
    restart_registry[context_id] = [(name, restart_func) for (name, restart_func) in restarts]
    try
        return func()
    catch e
        rethrow(e)
    end
end
```

Function invoke_restart

- Searches the global restart_registry for the specified name
- If found, calls the associated restart_func with provided args. The execution continues from within the restart function, potentially never returning to the invoke_restart call site
- Throws an ArgumentError if no restart with the given name is found in any active context
- Completes the separation of concerns: handling detects/catches, invoke_restart chooses and triggers recovery defined by with_restart
- Performs a non-local control transfer, fundamentally altering the execution flow away from the error site towards the recovery code
- The ArgumentError signals a misuse of the system (trying to invoke something not available)
- Dynamic dispatch: appropriate restart function is determined at runtime based on the restart name and arguments

```
function invoke_restart(name, args...)
    for (context_id, restarts) in restart_registry
        for (restart_name, restart_func) in restarts
            if restart_name == name
                return restart_func(args...)
            end
        end
    end
    throw(ArgumentError("No restart named $name is available"))
end
```

Function available_restart

- Searches the global restart_registry across all active contexts
- Iterates through registered restarts, looking for a matching name
- Returns true if found, false otherwise
- Provides introspection capabilities to the error handling logic
- Enhances the robustness of handlers by allowing them to query the environment before committing to a recovery strategy (like invoking a restart)
- Further promotes the decoupling of handler logic from the specific implementation of recovery actions

```
function available_restart(name)
    for (context_id, restarts) in restart_registry
        for (restart_name, _) in restarts
            if restart_name == name
                return true
            end
        end
    end
    return false
end
```

Signal

- Signals an exceptional situation
- Allows handlers to respond without stopping execution, or ignore signal
- Decouples error detection from immediate handling
- Allows multiple handlers to react to an event
- Useful for logging, debugging, and notifications

```
handling(LineEndLimit => (c)->println()) do
    print_line("Hi, everybody! How are you feeling today?")
end
```

Function signal

- Looks up relevant handlers in the global signal_handlers registry based on the exception's type
- Iterates through registered handlers and calls them sequentially
- If any handler returns true, it signifies the signal has been sufficiently "handled," and processing stops for that signal
- Returns true if any handler returned true, false otherwise
- Implements an Observer-like pattern or Chain of Responsibility: multiple handlers can observe a signal, and the chain can be broken if one handler fully addresses it
- Decouples the signaler from the listeners (handlers)
- Provides a mechanism for broadcasting information about non-critical events within the system, managed by the same dynamic scoping rules as handling
- Dynamic dispatch: handlers are selected based on the runtime type of the exception, and each handler is called with the exception as an argument

```
function signal(exception)
    if haskey(signal_handlers, typeof(exception))
        for handler in signal_handlers[typeof(exception)]
            result = handler(exception)
            if result === true
                return true
            end
        end
    end
    return false
end
```

Implementation details - global variables

restart_registry

- a dictionary that maps context IDs to available restart strategies (tuple of name and function)
- with_restart creates an entry in restart_registry with restart strategies for a given context
- invoke_restart uses restart_registry to find and appply available restarts for a given context
- available_restarts can check restart_registry for available restarts

signal_handlers

- a dictionary mapping exception types to vectors of handler functions, allowing multiple handlers per exception type
- used in handler function to register a new handler for a specific exception type using helper function **register_signal_handler**
- will be cleaned up when out of scope in handling, using helper function **remove_signal_handler**
- used in signal function to invoke all relevant handlers for a given exception type

Architectural Decisions

Design choices

- Dynamic environment: Handlers and restarts are dynamically registered at runtime
  - allows error handling based on context from execution path
- Global state: Used for tracking handlers and restarts
- Custom exception types: Enables advanced flow control

Trade-offs

- Increased complexity compared to standard try/catch
- Potential concurrency challenges due to global state
- Requires careful design to avoid infinite loops or missing handlers

Why this design?

- Balances flexibility and control
- Inspired by Lisp's condition system but tailored for Julia
- Extends Julia's standard exception model

Design Patterns Used

Key design patterns implemented

- Command Pattern: Restart functions encapsulate recoverable actions
- Chain of Responsibility: Handlers process exceptions in sequence
- Observer Pattern: Signals notify multiple handlers
- Dynamic Dispatch: Handlers and restarts selected at runtime

Benefits of these patterns

- Makes the system modular and extensible
- Improves code reuse and separation of concerns
- Allows customizable error handling strategies

Conclusion

What have we achieved?

Extended Julia's exception handling beyond try/catch

Provided dynamic, recoverable error handling

Separated detection, handling, and recovery

Key takeaways

The condition system adds flexibility and power

Allows fine-grained control over exceptions and errors

Inspired by Lisp's condition system but adapted for Julia

Future improvements

Implement user handling of restarts

Implementing Common Lisp restart options, test, report, interactive

Implement macros handler_case and restart_case to simplify functions handling and with_restart