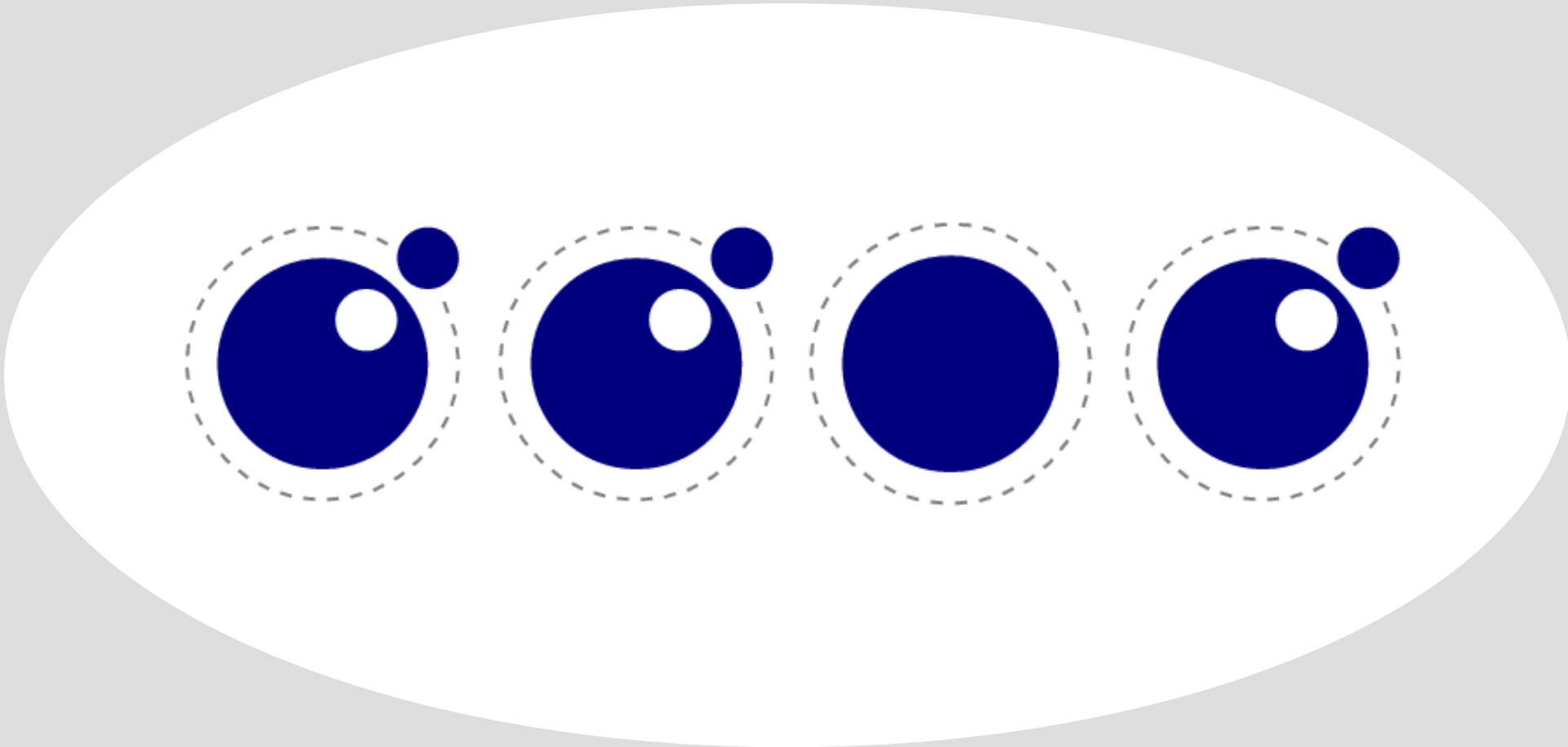
Exception patterns in Lua



John Belmonte

Lua Workshop 2006

Overview

- A reintroduction to exceptions
- Lua and exceptions
- A simple try-except construct
- Custom error objects

What problem do exceptions solve?

- Reasonable program behavior despite lack of error handling
- Error handling only where needed
- Consistency in raising and handling
- Simpler API's

(Good summary at http://digitalmars.com/d/errors.html)

Exception Concepts

- Raise
- Catch
- Re-raise
- Selective catch
  - can apply to any catch scenario
  - requires classification of errors
- Exceptions are part of an API

Usage Scenarios

- Quick scripting
  - let everything go unhandled
- Catching errors for:
  - suppression
  - alternate code path
  - cleanup (often re-raising)
  - retry
  - transformation (always re-raising)
    - add context
    - hide implementation

What should be an error?

- Obvious error: invalid arguments
- Usually not an error: string match failure
- What about file operation failures (open, delete, rename)?
- Criteria: If caller usually can't deal with the situation locally, it's an error
  - i.e. errors usually propagate up two or more stack frames

Lua and exceptions

- Raise with error(), assert(), lua_error()
- Catch with pcall()
- Implemented with C longjmp()
- Error object not limited to strings
- No try-except construct

Usage in core and standard library

- Exceptions mainly used for obvious programming errors
  - parse errors
  - type errors
  - invalid function arguments
- Notable departures: require(), dofile()
- Exclusively string error objects

The nil-error protocol

- On error, function returns [nil, error message] tuple
- Made popular by Lua standard libs
- Issues
  - not checking can result in delayed, secondary error
  - what if nil is a valid output?
- Can use assert() to convert to exception

A simple try-except construct

- Rationale
  - useful
  - familiar
  - encourages use of exceptions
- Requirements
  - usable without Lua changes
  - can be nested

Try-except definition

- try(f, catch_f)
  - Executes function `f`, and if an exception results then `catch_f` is called with the error object.
- Differs from xpcall()
  - propagates exceptions rather than returning nil-error
  - error handler supports nested errors

Try-except implementation and usage

```
function try(f, catch_f)
    local status, exception = pcall(f)
    if not status then
        catch_f(exception)
    end
end

try(function()
    -- Try block
    --
end, function(e)
    -- Except block.  E.g.:
    --   Use e for conditional catch
    --   Re-raise with error(e)
end)
```

Try-except issues

- Slightly verbose
  - use token filter: $try ... $catch(e) ... $end
- Functional implementation doesn't support direct return from try/catch
  - native implementation would solve this
- Coroutine yield cannot be used within a pcall
  - copcall() is a workaround
- Add finally block?
  - not as significant as for C
  - D's “scope hook” concept is better

Custom exception objects

```
error({code=121})
```

- What's wrong with strings?
  - selective catch is fragile at best
- Tables as errors
  - positive error identity
  - can attach arbitrary context
- Classes as errors
  - can employ inheritance testing

Sample error hierarchy

Excerpt from Python's built-in hierarchy:

```
Exception
    StandardError
        ArithmeticError
            FloatingPointError
            OverflowError
            ZeroDivisionError
        AssertionError
        ImportError
        KeyboardInterrupt
        RuntimeError
            NotImplementedError
        SyntaxError
        TypeError
        ValueError
```

Uncaught table

```
> error({code=121})
(error object is not a string)
```

- In the dark if table object is uncaught
  - what is the error?
  - where did it come from?
- Call stack should be displayed regardless of error type
  - lua.c should call tostring() on error objects
- All exceptions should have human-readable message

A better error object

- Set __tostring hook
- Make reference available

```
_exception_mt = { __tostring =
    function(e) return 'ERROR: '..e.msg end }
SomethingBad = {code=121, msg = 'Oops' }
setmetatable(SomethingBad, _exception_mt)
```

- Then, with patched lua.c:

```
> error(SomethingBad)
ERROR: Oops
stack traceback:
        [C]: in function 'error'
        ...
```

Still missing file and line number!

How error locations are conveyed in Lua

- Error system does not have concept of error location
- Convention is to pre-pend location to error string
- error() does this for you
- ... but only for string exceptions

Error location fix

- Ideal: lua_error() associates location with error object
  - possible efficiency concerns
- Compromise: error() sets location directly on object when it's a table
  - prototyped, works well

Conclusions

- Throw exceptions in situations which usually can't be handled locally by parent stack frame
- Use try-except construct for exception handling
- Throw tables instead of strings
- Enumerate errors as part of API
- Fixing pcall/coroutine problem is important
- Standard interface for inheritance testing would be useful

Resources

- See presentation source for ample notes, bonus slides
- Power patch for custom error object support coming soon