Table of Contents

Error Package Documentation

This package provides utilities for working with MoonBit's error handling system
, including implementations of Show and ToJson traits for the built-in Error typ
e.

Basic Error Usage

MoonBit uses a structured error system with raise and try constructs:

```
1
2   test "basic error handling" {
3     fn divide(a : Int, b : Int) -> Int raise {
4       if b == 0 {
5         raise Failure("Division by zero")
6       } else {
7         a / b
8       }
9     }
10
11
12    let result1 = try! divide(10, 2)
13    inspect(result1, content="5")
14
15
16    let result2 = try? divide(10, 0)
17    inspect(result2, content="Err(Failure(\"Division by zero\"))")
18  }
```

Custom Error Types

Define custom error types using suberror:

```
suberror ValidationError String


suberror NetworkError String


test "custom errors" {
  fn validate_email(email : String) -> String raise ValidationError {
    if email.length() > 5 {
      email
    } else {
      raise ValidationError("Invalid email format")
    }
  }

  fn fetch_data(url : String) -> String raise NetworkError {
    if url.length() > 10 {
      "data"
    } else {
      raise NetworkError("Invalid URL")
    }
  }


  let email_result = try? validate_email("short")
  match email_result {
    Ok(_) => inspect(false, content="true")
    Err(_) => inspect(true, content="true")
  }


  let data_result = try? fetch_data("short")
  match data_result {
    Ok(_) => inspect(false, content="true")
    Err(_) => inspect(true, content="true")
  }
}
```

Error Display and JSON Conversion

The error package provides Show and ToJson implementations:

```

suberror MyError Int derive(ToJson)


test "error display and json" {
  let error : Error = MyError(42)


  let error_string = error.to_string()
  inspect(error_string.length() > 0, content="true")


  let error_json = error.to_json()
  inspect(error_json, content="Array([String(\"MyError\"), Number(42)])"
}
```

Error Propagation and Handling

Handle errors at different levels of your application:

```

suberror ParseError String


suberror FileError String


test "error propagation" {
  fn parse_number(s : String) -> Int raise ParseError {
    if s == "42" {
      42
    } else {
      raise ParseError("Invalid number: " + s)
    }
  }

  fn read_and_parse(content : String) -> Int raise {
    parse_number(content) catch {
      ParseError(msg) => raise FileError("Parse failed: " + msg)
    }
  }


  let result1 = try! read_and_parse("42")
  inspect(result1, content="42")


  let result2 = try? read_and_parse("invalid")
  match result2 {
    Ok(_) => inspect(false, content="true")
    Err(_) => inspect(true, content="true")
  }
}
```

Resource Management with Finally

Use protect functions for resource cleanup:

```

  suberror ResourceError String


  test "resource management" {
    fn risky_operation() -> String raise ResourceError {
      raise ResourceError("Something went wrong")
    }


    fn use_resource() -> String raise {

      risky_operation() catch {
        ResourceError(_) =>

          raise Failure("Operation failed after cleanup")
      }
    }

    let result = try? use_resource()
    match result {
      Ok(_) => inspect(false, content="true")
      Err(_) => inspect(true, content="true")
    }
  }
```

Error Composition

Combine multiple error-producing operations:

```
1
2   suberror ConfigError String
3
4
5   suberror DatabaseError String
6
7
8   test "error composition" {
9     fn load_config() -> String raise ConfigError {
10      if true {
11        "config_data"
12      } else {
13        raise ConfigError("Config not found")
14      }
15    }
16
17    fn connect_database(config : String) -> String raise DatabaseError {
18      if config == "config_data" {
19        "connected"
20      } else {
21        raise DatabaseError("Invalid config")
22      }
23    }
24
25    fn initialize_app() -> String raise {
26      let config = load_config() catch {
27        ConfigError(msg) => raise Failure("Config error: " + msg)
28      }
29      let db = connect_database(config) catch {
30        DatabaseError(msg) => raise Failure("Database error: " + msg)
31      }
32      "App initialized with " + db
33    }
34
35    let app_result = try! initialize_app()
36    inspect(app_result, content="App initialized with connected")
37  }
```

Best Practices

- **Use specific error types**: Create custom suberror types for different error categories
- **Provide meaningful messages**: Include context and actionable information in error messages
- **Handle errors at appropriate levels**: Don’t catch errors too early; let them propagate to where they can be properly handled
- **Use** try! for operations that should not fail: This will panic if an error occurs, making failures visible during development
- **Use** try? for recoverable errors: This returns a Result type that can be pattern matched
- **Implement proper cleanup**: Use the protect pattern or similar constructs for resource management

Performance Notes

- Error handling in MoonBit is zero-cost when no errors occur
- Error propagation is efficient and doesn’t require heap allocation for the error path
- Custom error types with derive(ToJson) automatically generate efficient JSON serialization