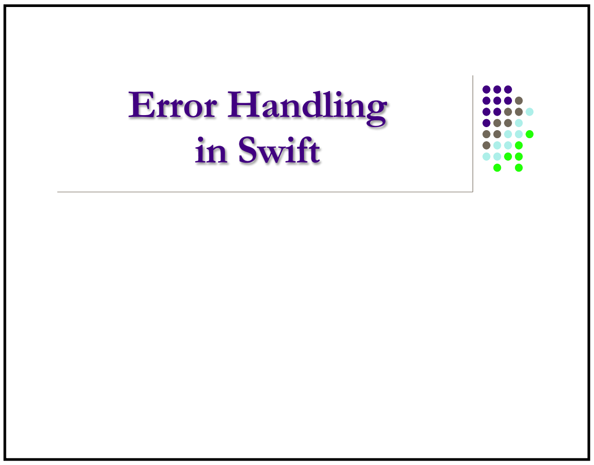
CSC 491 / 391 Mobile Application Development for iOS II

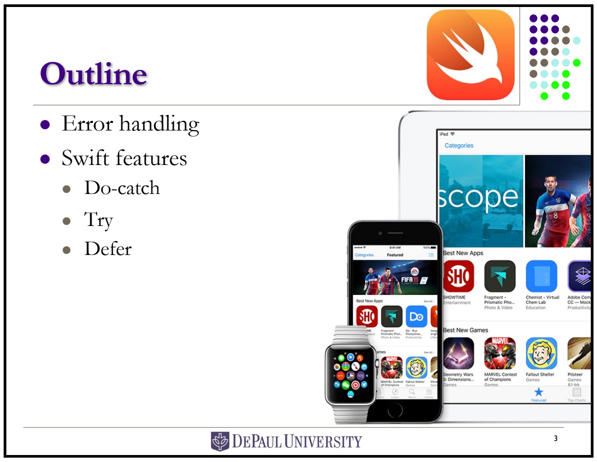
Prof. Xiaoping Jia
School of Computing, CDM
DePaul University
xjia@cdm.depaul.edu
@DePaulSWEng

1



Error Handling in Swift

2

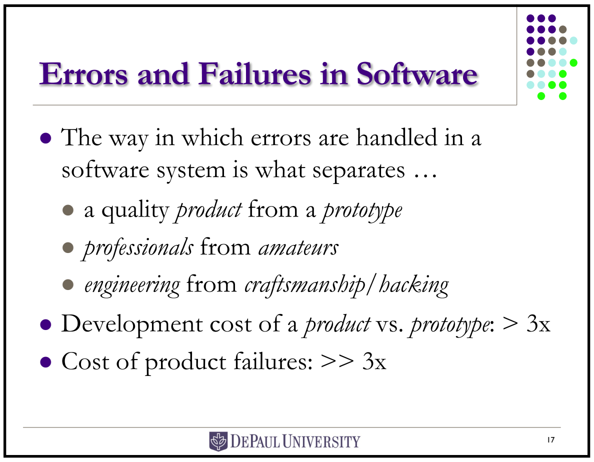


Outline

- Error handling
- Swift features
  - Do-catch
  - Try
  - Defer

DePaul University 3

3

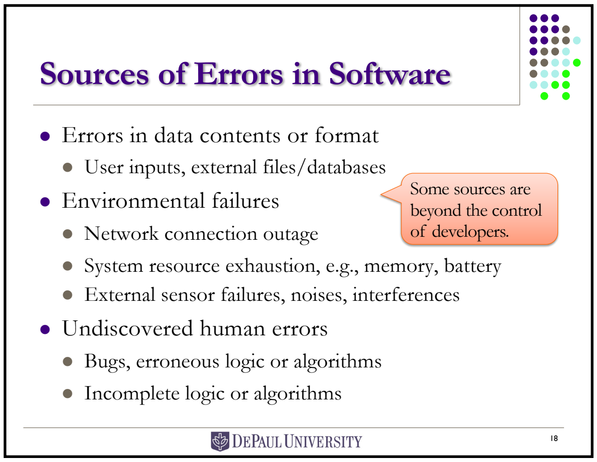


Errors and Failures in Software

- The way in which errors are handled in a software system is what separates …
  - a quality *product* from a *prototype*
  - *professionals* from *amateurs*
  - *engineering* from *craftsmanship/hacking*
- Development cost of a *product* vs. *prototype*: > 3x
- Cost of product failures: >> 3x

DePaul University 17

4



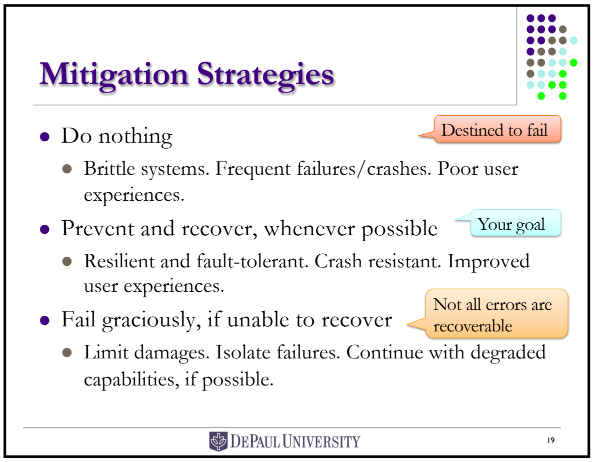
Sources of Errors in Software

- Errors in data contents or format
  - User inputs, external files/databases
- Environmental failures
  - Network connection outage
  - System resource exhaustion, e.g., memory, battery
  - External sensor failures, noises, interferences
- Undiscovered human errors
  - Bugs, erroneous logic or algorithms
  - Incomplete logic or algorithms

Some sources are beyond the control of developers.

DePaul University 18

5



Mitigation Strategies

- Do nothing
  - Brittle systems. Frequent failures/crashes. Poor user experiences.
- Prevent and recover, whenever possible
  - Resilient and fault-tolerant. Crash resistant. Improved user experiences.
- Fail graciously, if unable to recover
  - Limit damages. Isolate failures. Continue with degraded capabilities, if possible.

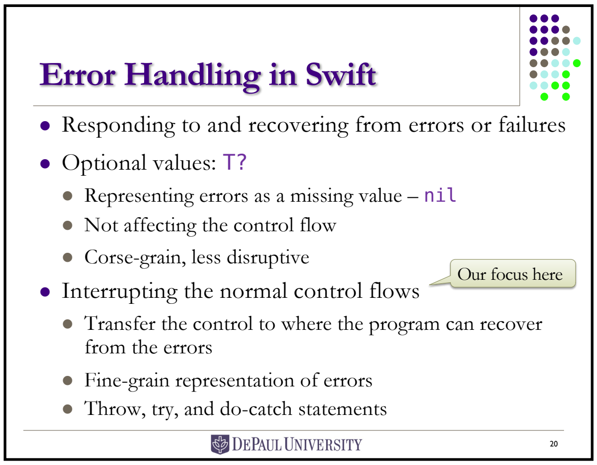
Destined to fail

Your goal

Not all errors are recoverable

DePaul University 19

11



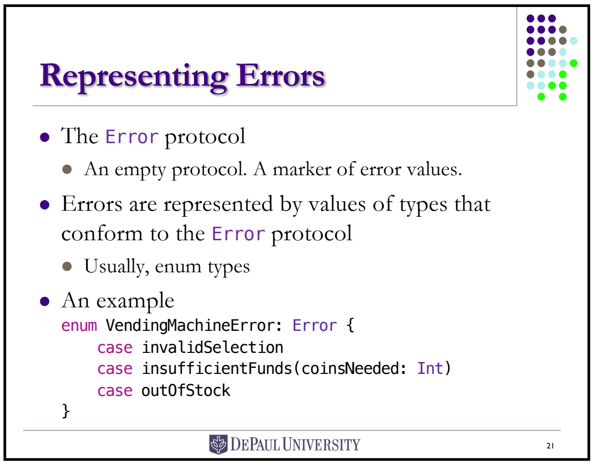
Error Handling in Swift

- Responding to and recovering from errors or failures
- Optional values: `T?`
  - Representing errors as a missing value – `nil`
  - Not affecting the control flow
  - Corse-grain, less disruptive
- Interrupting the normal control flows
  - Transfer the control to where the program can recover from the errors
  - Fine-grain representation of errors
  - Throw, try, and do-catch statements

Our focus here

DEPAUL UNIVERSITY 20

15



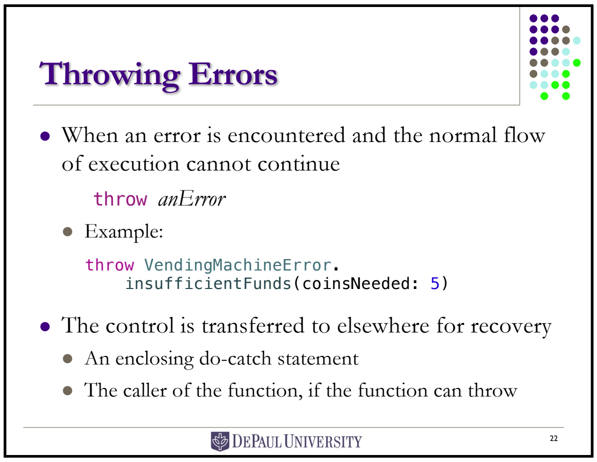
Representing Errors

- The `Error` protocol
  - An empty protocol. A marker of error values.
- Errors are represented by values of types that conform to the `Error` protocol
  - Usually, enum types
- An example

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

DEPAUL UNIVERSITY 21

16



Throwing Errors

- When an error is encountered and the normal flow of execution cannot continue

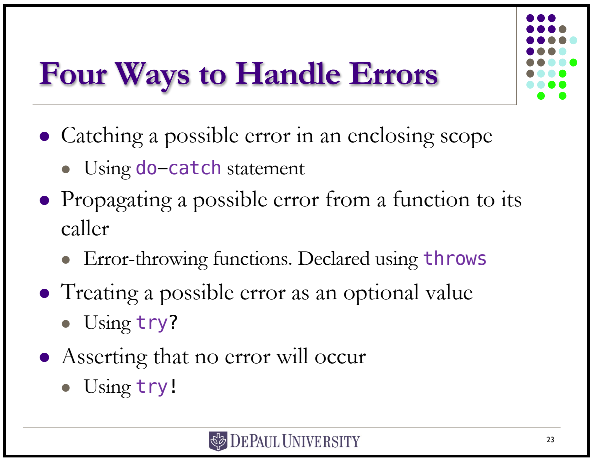
  `throw` *anError*
  - Example:

```
throw VendingMachineError.
    insufficientFunds(coinsNeeded: 5)
```

- The control is transferred to elsewhere for recovery
  - An enclosing do-catch statement
  - The caller of the function, if the function can throw

DEPAUL UNIVERSITY 22

18

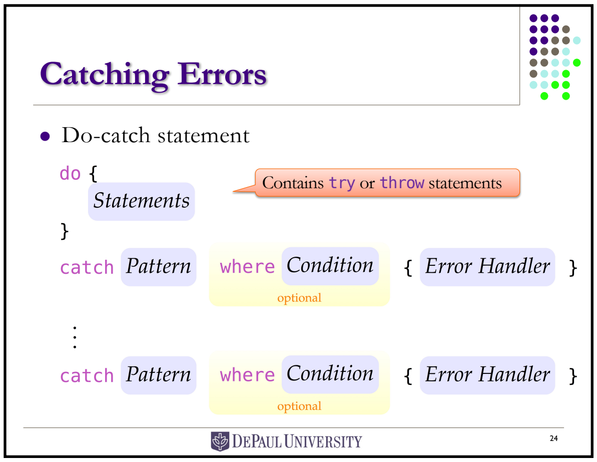


Four Ways to Handle Errors

- Catching a possible error in an enclosing scope
  - Using `do–catch` statement
- Propagating a possible error from a function to its caller
  - Error-throwing functions. Declared using `throws`
- Treating a possible error as an optional value
  - Using `try?`
- Asserting that no error will occur
  - Using `try!`

DEPAUL UNIVERSITY 23

19



Catching Errors

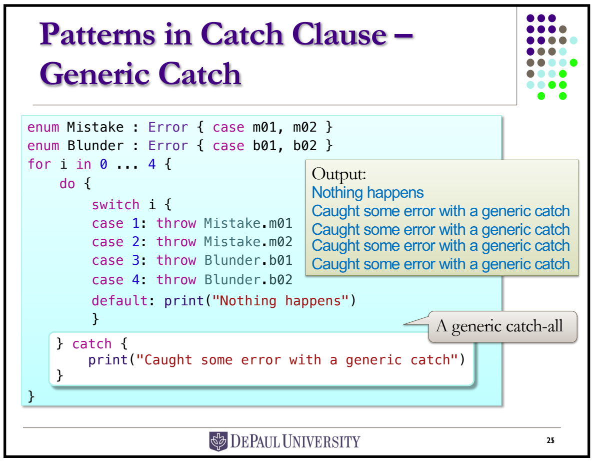
- Do-catch statement

`do` {
Statements
}
`catch` *Pattern* `where` *Condition* { *Error Handler* }
optional
⋮
`catch` *Pattern* `where` *Condition* { *Error Handler* }
optional

Contains `try` or `throw` statements

DEPAUL UNIVERSITY 24

20



Patterns in Catch Clause – Generic Catch

```
enum Mistake : Error { case m01, m02 }
enum Blunder : Error { case b01, b02 }
for i in 0 ... 4 {
    do {
        switch i {
        case 1: throw Mistake.m01
        case 2: throw Mistake.m02
        case 3: throw Blunder.b01
        case 4: throw Blunder.b02
        default: print("Nothing happens")
        }
    } catch {
        print("Caught some error with a generic catch")
    }
}
```

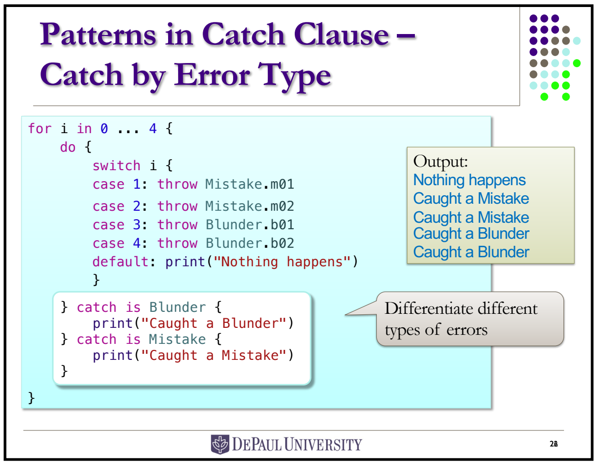
Output:

```
Nothing happens
Caught some error with a generic catch
Caught some error with a generic catch
Caught some error with a generic catch
Caught some error with a generic catch
```

A generic catch-all

DEPAUL UNIVERSITY 25

21



Patterns in Catch Clause – Catch by Error Type

```
for i in 0 ... 4 {
    do {
        switch i {
        case 1: throw Mistake.m01
        case 2: throw Mistake.m02
        case 3: throw Blunder.b01
        case 4: throw Blunder.b02
        default: print("Nothing happens")
        }
    } catch is Blunder {
        print("Caught a Blunder")
    } catch is Mistake {
        print("Caught a Mistake")
    }
}
```

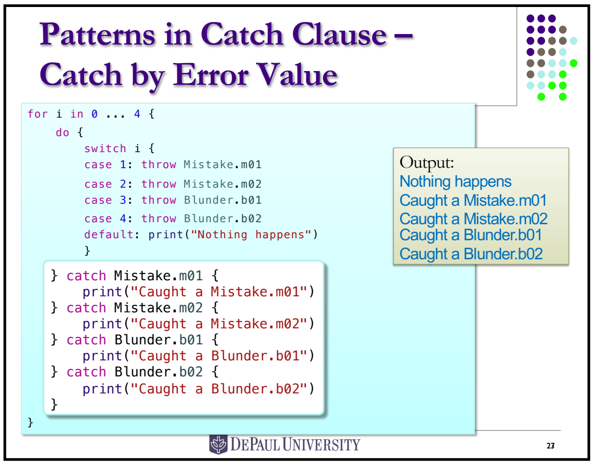
Output:

```
Nothing happens
Caught a Mistake
Caught a Mistake
Caught a Blunder
Caught a Blunder
```

Differentiate different types of errors

DePaul University 28

22



Patterns in Catch Clause – Catch by Error Value

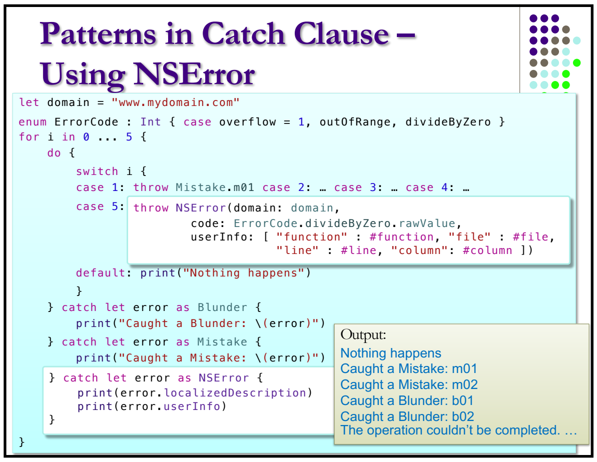
```
for i in 0 ... 4 {
    do {
        switch i {
        case 1: throw Mistake.m01
        case 2: throw Mistake.m02
        case 3: throw Blunder.b01
        case 4: throw Blunder.b02
        default: print("Nothing happens")
        }
    } catch Mistake.m01 {
        print("Caught a Mistake.m01")
    } catch Mistake.m02 {
        print("Caught a Mistake.m02")
    } catch Blunder.b01 {
        print("Caught a Blunder.b01")
    } catch Blunder.b02 {
        print("Caught a Blunder.b02")
    }
}
```

Output:

```
Nothing happens
Caught a Mistake.m01
Caught a Mistake.m02
Caught a Blunder.b01
Caught a Blunder.b02
```

DePaul University 23

23



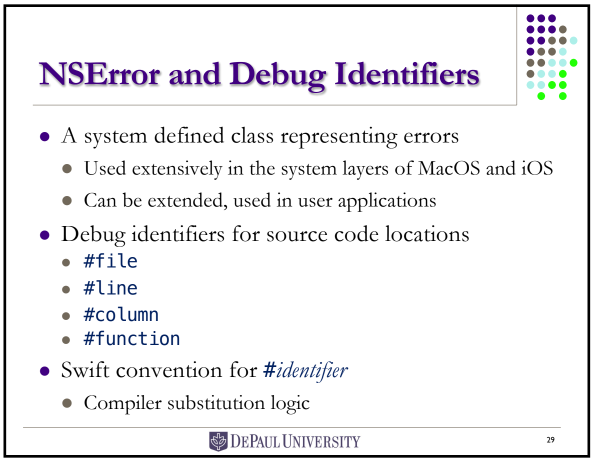
Patterns in Catch Clause – Using NSError

```
let domain = "www.mydomain.com"
enum ErrorCode : Int { case overflow = 1, outOfRange, divideByZero }
for i in 0 ... 5 {
    do {
        switch i {
        case 1: throw Mistake.m01 case 2: … case 3: … case 4: …
        case 5: throw NSError(domain: domain,
                        code: ErrorCode.divideByZero.rawValue,
                        userInfo: [ "function" : #function, "file" : #file,
                              "line" : #line, "column": #column ])
        default: print("Nothing happens")
        }
    } catch let error as Blunder {
        print("Caught a Blunder: \(error)")
    } catch let error as Mistake {
        print("Caught a Mistake: \(error)")
    } catch let error as NSError {
        print(error.localizedDescription)
        print(error.userInfo)
    }
}
```

Output:

```
Nothing happens
Caught a Mistake: m01
Caught a Mistake: m02
Caught a Blunder: b01
Caught a Blunder: b02
The operation couldn't be completed. …
```

24

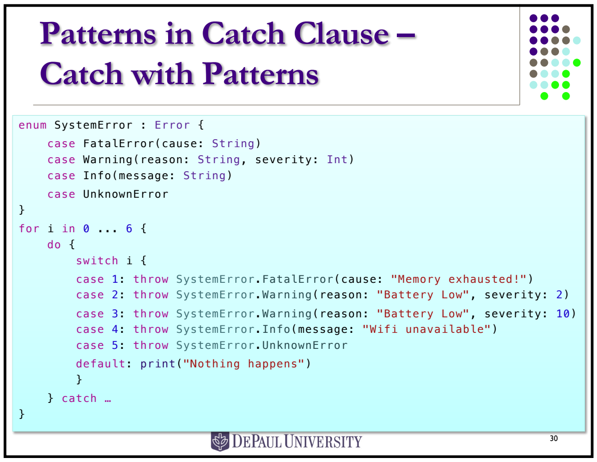


NSError and Debug Identifiers

- A system defined class representing errors
  - Used extensively in the system layers of MacOS and iOS
  - Can be extended, used in user applications
- Debug identifiers for source code locations
  - `#file`
  - `#line`
  - `#column`
  - `#function`
- Swift convention for *#identifier*
  - Compiler substitution logic

DePaul University 29

25

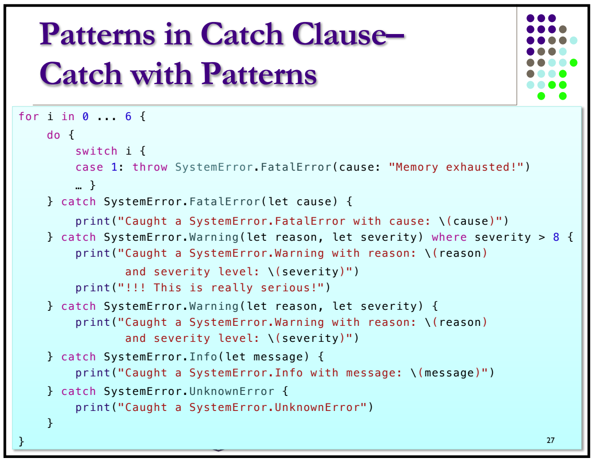


Patterns in Catch Clause – Catch with Patterns

```
enum SystemError : Error {
    case FatalError(cause: String)
    case Warning(reason: String, severity: Int)
    case Info(message: String)
    case UnknownError
}
for i in 0 ... 6 {
    do {
        switch i {
        case 1: throw SystemError.FatalError(cause: "Memory exhausted!")
        case 2: throw SystemError.Warning(reason: "Battery Low", severity: 2)
        case 3: throw SystemError.Warning(reason: "Battery Low", severity: 10)
        case 4: throw SystemError.Info(message: "Wifi unavailable")
        case 5: throw SystemError.UnknownError
        default: print("Nothing happens")
        }
    } catch …
}
```

DePaul University 30

26

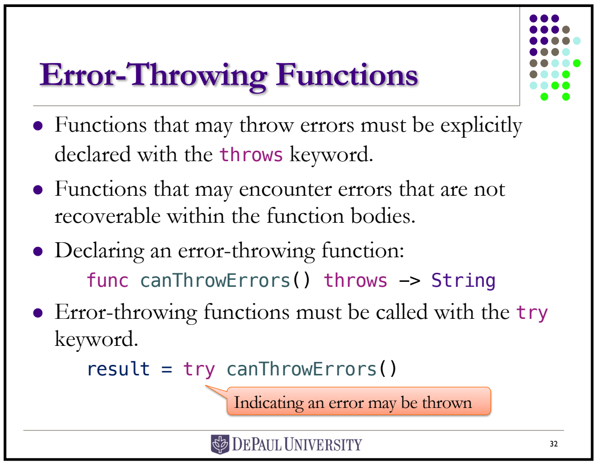


Patterns in Catch Clause– Catch with Patterns

```
for i in 0 ... 6 {
    do {
        switch i {
        case 1: throw SystemError.FatalError(cause: "Memory exhausted!")
        … }
    } catch SystemError.FatalError(let cause) {
        print("Caught a SystemError.FatalError with cause: \(cause)")
    } catch SystemError.Warning(let reason, let severity) where severity > 8 {
        print("Caught a SystemError.Warning with reason: \(reason)
              and severity level: \(severity)")
        print("!!! This is really serious!")
    } catch SystemError.Warning(let reason, let severity) {
        print("Caught a SystemError.Warning with reason: \(reason)
              and severity level: \(severity)")
    } catch SystemError.Info(let message) {
        print("Caught a SystemError.Info with message: \(message)")
    } catch SystemError.UnknownError {
        print("Caught a SystemError.UnknownError")
    }
}
```

27

27



Error-Throwing Functions

- Functions that may throw errors must be explicitly declared with the `throws` keyword.
- Functions that may encounter errors that are not recoverable within the function bodies.
- Declaring an error-throwing function:

```
func canThrowErrors() throws -> String
```

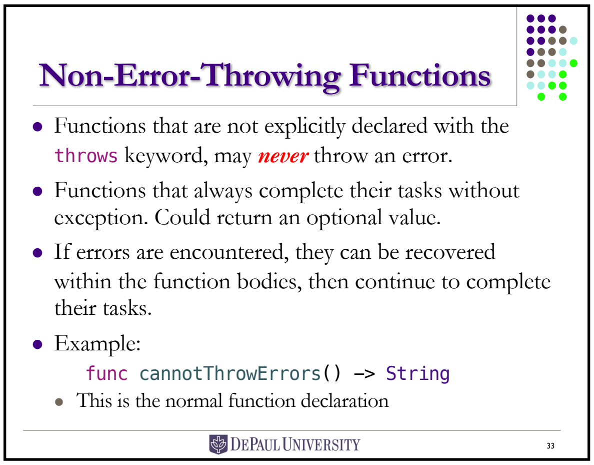
- Error-throwing functions must be called with the `try` keyword.

```
result = try canThrowErrors()
```

Indicating an error may be thrown

DePaul University 32

28



Non-Error-Throwing Functions

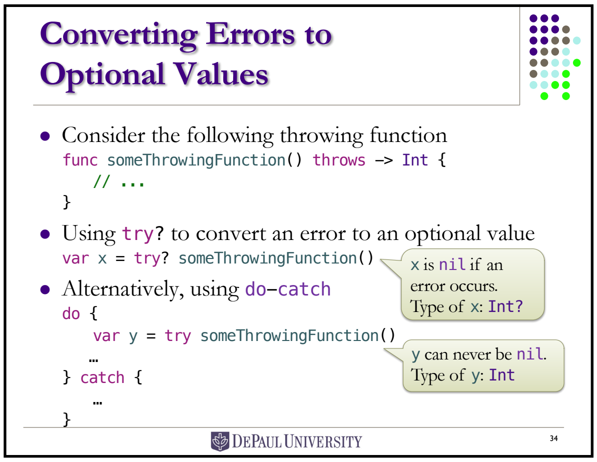
- Functions that are not explicitly declared with the `throws` keyword, may ***never*** throw an error.
- Functions that always complete their tasks without exception. Could return an optional value.
- If errors are encountered, they can be recovered within the function bodies, then continue to complete their tasks.
- Example:

```
func cannotThrowErrors() -> String
```

  - This is the normal function declaration

DePaul University 33

29



Converting Errors to Optional Values

- Consider the following throwing function

```
func someThrowingFunction() throws -> Int {
    // ...
}
```

- Using `try?` to convert an error to an optional value

```
var x = try? someThrowingFunction()
```

x is `nil` if an error occurs. Type of x: `Int?`

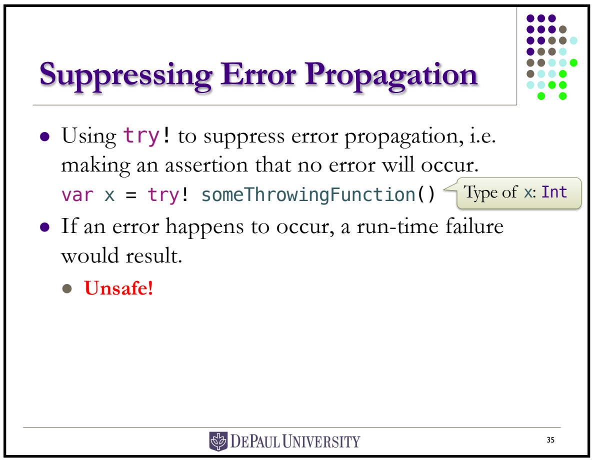
- Alternatively, using `do-catch`

```
do {
    var y = try someThrowingFunction()
    …
} catch {
    …
}
```

y can never be `nil`. Type of y: `Int`

DePaul University 34

30



Suppressing Error Propagation

- Using `try!` to suppress error propagation, i.e. making an assertion that no error will occur.

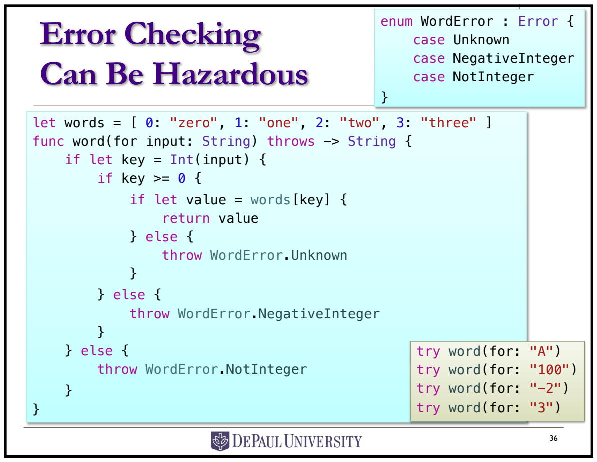
```
var x = try! someThrowingFunction()
```

Type of x: `Int`

- If an error happens to occur, a run-time failure would result.
  - **Unsafe!**

DePaul University 35

31



Error Checking Can Be Hazardous

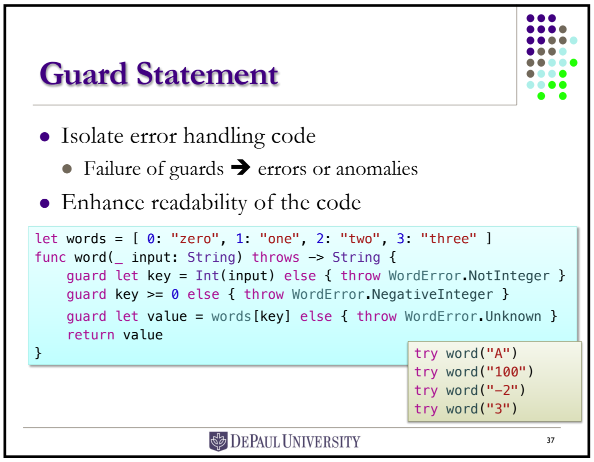
```
enum WordError : Error {
    case Unknown
    case NegativeInteger
    case NotInteger
}
```

```
let words = [ 0: "zero", 1: "one", 2: "two", 3: "three" ]
func word(for input: String) throws -> String {
    if let key = Int(input) {
        if key >= 0 {
            if let value = words[key] {
                return value
            } else {
                throw WordError.Unknown
            }
        } else {
            throw WordError.NegativeInteger
        }
    } else {
        throw WordError.NotInteger
    }
}
```

```
try word(for: "A")
try word(for: "100")
try word(for: "-2")
try word(for: "3")
```

DePaul University 36

32



Guard Statement

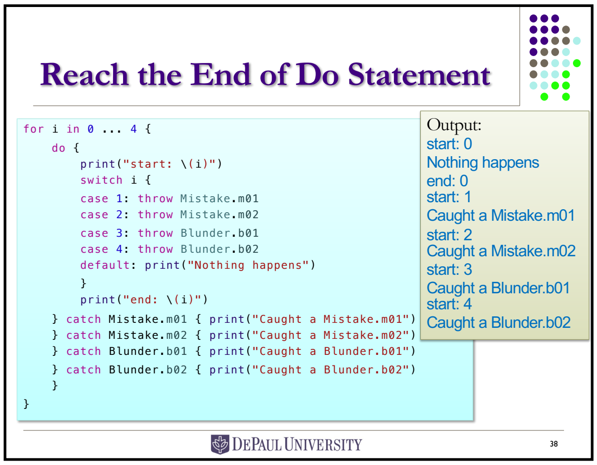
- Isolate error handling code
  - Failure of guards ➔ errors or anomalies
- Enhance readability of the code

```
let words = [ 0: "zero", 1: "one", 2: "two", 3: "three" ]
func word(_ input: String) throws -> String {
    guard let key = Int(input) else { throw WordError.NotInteger }
    guard key >= 0 else { throw WordError.NegativeInteger }
    guard let value = words[key] else { throw WordError.Unknown }
    return value
}
```

```
try word("A")
try word("100")
try word("-2")
try word("3")
```

DePaul University 37

33



Reach the End of Do Statement

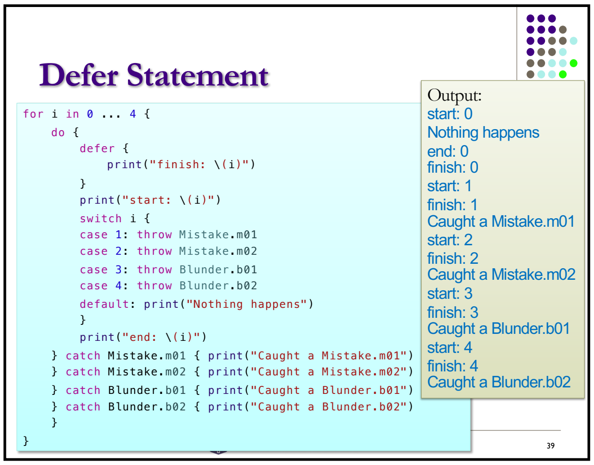
```
for i in 0 ... 4 {
    do {
        print("start: \(i)")
        switch i {
        case 1: throw Mistake.m01
        case 2: throw Mistake.m02
        case 3: throw Blunder.b01
        case 4: throw Blunder.b02
        default: print("Nothing happens")
        }
        print("end: \(i)")
    } catch Mistake.m01 { print("Caught a Mistake.m01")
    } catch Mistake.m02 { print("Caught a Mistake.m02")
    } catch Blunder.b01 { print("Caught a Blunder.b01")
    } catch Blunder.b02 { print("Caught a Blunder.b02")
    }
}
```

Output:

```
start: 0
Nothing happens
end: 0
start: 1
Caught a Mistake.m01
start: 2
Caught a Mistake.m02
start: 3
Caught a Blunder.b01
start: 4
Caught a Blunder.b02
```

DePaul University 38

34



Defer Statement

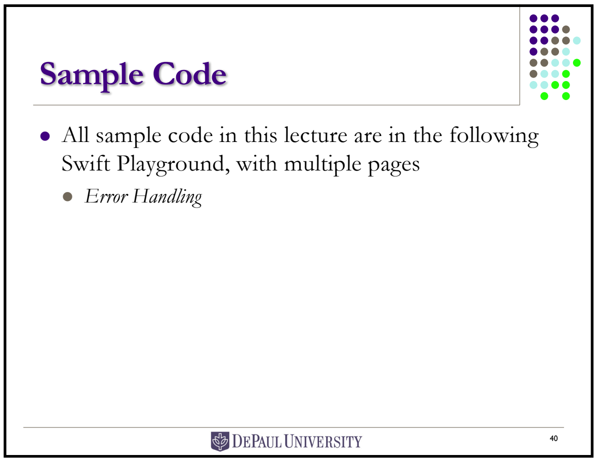
```
for i in 0 ... 4 {
    do {
        defer {
            print("finish: \(i)")
        }
        print("start: \(i)")
        switch i {
        case 1: throw Mistake.m01
        case 2: throw Mistake.m02
        case 3: throw Blunder.b01
        case 4: throw Blunder.b02
        default: print("Nothing happens")
        }
        print("end: \(i)")
    } catch Mistake.m01 { print("Caught a Mistake.m01")
    } catch Mistake.m02 { print("Caught a Mistake.m02")
    } catch Blunder.b01 { print("Caught a Blunder.b01")
    } catch Blunder.b02 { print("Caught a Blunder.b02")
    }
}
```

Output:

```
start: 0
Nothing happens
end: 0
finish: 0
start: 1
finish: 1
Caught a Mistake.m01
start: 2
finish: 2
Caught a Mistake.m02
start: 3
finish: 3
Caught a Blunder.b01
start: 4
finish: 4
Caught a Blunder.b02
```

39

35

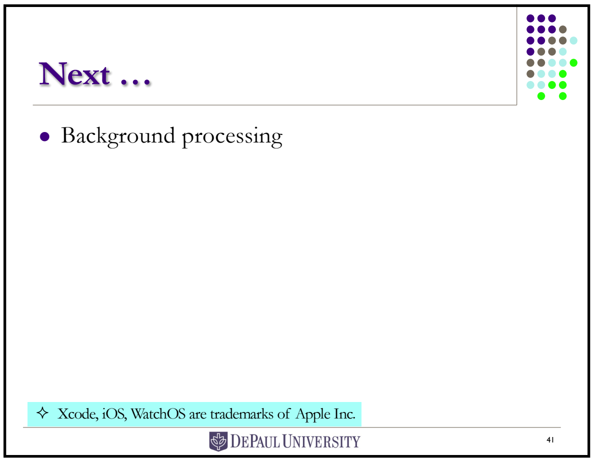


Sample Code

- All sample code in this lecture are in the following Swift Playground, with multiple pages
  - *Error Handling*

DePaul University 40

36



Next …

- Background processing

✧ Xcode, iOS, WatchOS are trademarks of Apple Inc.

DePaul University 41

37