Error Handling

On This Page

Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.

Some operations aren't guaranteed to always complete execution or produce a useful output. Optionals are used to represent the absence of a value, but when an operation fails, it's often useful to understand what caused the failure, so that your code can respond accordingly.

As an example, consider the task of reading and processing data from a file on disk. There are a number of ways this task can fail, including the file not existing at the specified path, the file not having read permissions, or the file not being encoded in a compatible format. Distinguishing among these different situations allows a program to resolve some errors and to communicate to the user any errors it can't resolve.

> NOTE
>
> Error handling in Swift interoperates with error handling patterns that use the `NSError` class in Cocoa and Objective-C. For more information about this class, see Error Handling in *Using Swift with Cocoa and Objective-C (Swift 4.1)*.

Representing and Throwing Errors

In Swift, errors are represented by values of types that conform to the `Error` protocol. This empty protocol indicates that a type can be used for error handling.

Swift enumerations are particularly well suited to modeling a group of related error conditions, with associated values allowing for additional information about the nature of an error to be communicated. For example, here's how you might represent the error conditions of operating a vending machine inside a game:

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

Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue. You use a `throw` statement to throw an error. For example, the following code throws an error to indicate that five additional coins are needed by the vending machine:

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

Handling Errors

When an error is thrown, some surrounding piece of code must be responsible for handling the error—for example, by correcting the problem, trying an alternative approach, or informing the user of the failure.

There are four ways to handle errors in Swift. You can propagate the error from a function to the code that calls that function, handle the error using a `do-catch` statement, handle the error as an optional value, or assert that the error will not occur. Each approach is described in a section below.

When a function throws an error, it changes the flow of your program, so it's important that you can quickly identify places in your code that can throw errors. To identify these places in your code, write the `try` keyword—or the `try?` or `try!` variation—before a piece of code that calls a function, method, or initializer that can throw an error. These keywords are described in the sections below.

> NOTE
>
> Error handling in Swift resembles exception handling in other languages, with the use of the `try`, `catch` and `throw` keywords. Unlike exception handling in many languages—including Objective-C—error handling in Swift does not involve unwinding the call stack, a process that can be computationally expensive. As such, the performance characteristics of a `throw` statement are comparable to those of a `return` statement.

Propagating Errors Using Throwing Functions

To indicate that a function, method, or initializer can throw an error, you write the `throws` keyword in the function's declaration after its parameters. A function marked with `throws` is called a *throwing function*. If the function specifies a return type, you write the `throws` keyword before the return arrow (`->`).

```
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

A throwing function propagates errors that are thrown inside of it to the scope from which it's called.

> NOTE
>
> Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.

In the example below, the `VendingMachine` class has a `vend(itemNamed:)` method that throws an appropriate `VendingMachineError` if the requested item is not available, is out of stock, or has a cost that exceeds the current deposited amount:

```
struct Item {
    var price: Int
    var count: Int
}

class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0

    func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price -
  coinsDeposited)
        }

        coinsDeposited -= item.price
```

```
28
29          var newItem = item
30          newItem.count -= 1
31          inventory[name] = newItem
32
33          print("Dispensing \(name)")
34      }
35  }
```

The implementation of the `vend(itemNamed:)` method uses `guard` statements to exit the method early and throw appropriate errors if any of the requirements for purchasing a snack aren't met. Because a `throw` statement immediately transfers program control, an item will be vended only if all of these requirements are met.

Because the `vend(itemNamed:)` method propagates any errors it throws, any code that calls this method must either handle the errors—using a `do-catch` statement, `try?`, or `try!`—or continue to propagate them. For example, the `buyFavoriteSnack(person:vendingMachine:)` in the example below is also a throwing function, and any errors that the `vend(itemNamed:)` method throws will propagate up to the point where the `buyFavoriteSnack(person:vendingMachine:)` function is called.

```
1  let favoriteSnacks = [
2      "Alice": "Chips",
3      "Bob": "Licorice",
4      "Eve": "Pretzels",
5  ]
6  func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
7      let snackName = favoriteSnacks[person] ?? "Candy Bar"
8      try vendingMachine.vend(itemNamed: snackName)
9  }
```

In this example, the `buyFavoriteSnack(person: vendingMachine:)` function looks up a given person's favorite snack and tries to buy it for them by calling the `vend(itemNamed:)` method. Because the `vend(itemNamed:)` method can throw an error, it's called with the `try` keyword in front of it.

Throwing initializers can propagate errors in the same way as throwing functions. For example, the initializer for the `PurchasedSnack` structure in the listing below calls a throwing function as part of the initialization process, and it handles any errors that it encounters by propagating them to its caller.

```
1  struct PurchasedSnack {
2      let name: String
3      init(name: String, vendingMachine: VendingMachine) throws {
4          try vendingMachine.vend(itemNamed: name)
5          self.name = name
6      }
7  }
```

Handling Errors Using Do-Catch

You use a `do-catch` statement to handle errors by running a block of code. If an error is thrown by the code in the `do` clause, it is matched against the `catch` clauses to determine which one of them can handle the error.

Here is the general form of a `do-catch` statement:

```
do {
    try expression
    statements
} catch pattern 1 {
```

```
    statements
} catch pattern 2 where condition {
    statements
} catch {
    statements
}
```

You write a pattern after `catch` to indicate what errors that clause can handle. If a `catch` clause doesn't have a pattern, the clause matches any error and binds the error to a local constant named `error`. For more information about pattern matching, see Patterns.

For example, the following code matches against all three cases of the `VendingMachineError` enumeration.

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \(error).")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

In the above example, the `buyFavoriteSnack(person:vendingMachine:)` function is called in a `try` expression, because it can throw an error. If an error is thrown, execution immediately transfers to the `catch` clauses, which decide whether to allow propagation to continue. If no pattern is matched, the error gets caught by the final `catch` clause and is bound to a local `error` constant. If no error is thrown, the remaining statements in the `do` statement are executed.

The `catch` clauses don't have to handle every possible error that the code in the `do` clause can throw. If none of the `catch` clauses handle the error, the error propagates to the surrounding scope. However, the propagated error must be handled by *some* surrounding scope. In a nonthrowing function, an enclosing `do-catch` clause must handle the error. In a throwing function, either an enclosing `do-catch` clause or the caller must handle the error. If the error propagates to the top-level scope without being handled, you'll get a runtime error.

For example, the above example can be written so any error that isn't a `VendingMachineError` is instead caught by the calling function:

```
func nourish(with item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch is VendingMachineError {
        print("Invalid selection, out of stock, or not enough money.")
    }
}

do {
    try nourish(with: "Beet-Flavored Chips")
} catch {
    print("Unexpected non-vending-machine-related error: \(error)")
```

```
}
// Prints "Invalid selection, out of stock, or not enough money."
```

In the `nourish(with:)` function, if `vend(itemNamed:)` throws an error that's one of the cases of the `VendingMachineError` enumeration, `nourish(with:)` handles the error by printing a message. Otherwise, `nourish(with:)` propagates the error to its call site. The error is then caught by the general `catch` clause.

Converting Errors to Optional Values

You use `try?` to handle an error by converting it to an optional value. If an error is thrown while evaluating the `try?` expression, the value of the expression is `nil`. For example, in the following code `x` and `y` have the same value and behavior:

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

let x = try? someThrowingFunction()

let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

If `someThrowingFunction()` throws an error, the value of `x` and `y` is `nil`. Otherwise, the value of `x` and `y` is the value that the function returned. Note that `x` and `y` are an optional of whatever type `someThrowingFunction()` returns. Here the function returns an integer, so `x` and `y` are optional integers.

Using `try?` lets you write concise error handling code when you want to handle all errors in the same way. For example, the following code uses several approaches to fetch data, or returns `nil` if all of the approaches fail.

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk() { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

Disabling Error Propagation

Sometimes you know a throwing function or method won't, in fact, throw an error at runtime. On those occasions, you can write `try!` before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you'll get a runtime error.

For example, the following code uses a `loadImage(atPath:)` function, which loads the image resource at a given path or throws an error if the image can't be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it is appropriate to disable error propagation.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Specifying Cleanup Actions

You use a `defer` statement to execute a set of statements just before code execution leaves the current block of code. This statement lets you do any necessary cleanup that should be performed regardless of *how*

execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as `return` or `break`. For example, you can use a `defer` statement to ensure that file descriptors are closed and manually allocated memory is freed.

A `defer` statement defers execution until the current scope is exited. This statement consists of the `defer` keyword and the statements to be executed later. The deferred statements may not contain any code that would transfer control out of the statements, such as a `break` or a `return` statement, or by throwing an error. Deferred actions are executed in the reverse of the order that they're written in your source code. That is, the code in the first `defer` statement executes last, the code in the second `defer` statement executes second to last, and so on. The last `defer` statement in source code order executes first.

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

The above example uses a `defer` statement to ensure that the `open(_:)` function has a corresponding call to `close(_:)`.

> NOTE
> You can use a `defer` statement even when no error handling code is involved.