Error Handling

You represent errors using any type that adopts the `Error` protocol.

```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

Use `throw` to throw an error and `throws` to mark a function that can throw an error. If you throw an error in a function, the function returns immediately and the code that called the function handles the error.

```
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

There are several ways to handle errors. One way is to use `do-catch`. Inside the `do` block, you mark code that can throw an error by writing `try` in front of it. Inside the `catch` block, the error is automatically given the name `error` unless you give it a different name.

```
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
} catch {
    print(error)
}
// Prints "Job sent"
```

EXPERIMENT

Change the printer name to `"Never Has Toner"`, so that the `send(job:toPrinter:)` function throws an error.

You can provide multiple `catch` blocks that handle specific errors. You write a pattern after `catch` just as you do after `case` in a switch.

```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
// Prints "Job sent"
```

EXPERIMENT

Add code to throw an error inside the `do` block. What kind of error do you need to throw so that the error is handled by the first `catch` block? What about the second and third blocks?

Another way to handle errors is to use `try?` to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is `nil`. Otherwise, the result is an optional containing the value that the function returned.

```
let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")
let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")
```

Use `defer` to write a block of code that's executed after all other code in the function, just before the function returns. The code is executed regardless of whether the function throws an error. You can use `defer` to write setup and cleanup code 서로 옆에, 나란히 next to each other, even though they need to be executed at different times.

```swift
var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]

func fridgeContains(_ food: String) -> Bool {
    fridgeIsOpen = true
    defer {
        fridgeIsOpen = false
    }

    let result = fridgeContent.contains(food)
    return result
}
fridgeContains("banana")
print(fridgeIsOpen)
// Prints "false"
```