Deinitialization

A *deinitializer* is called immediately before a class instance is deallocated. You write deinitializers with the `deinit` keyword, similar to how initializers are written with the `init` keyword. Deinitializers are only available on class types.

How Deinitialization Works

Swift automatically deallocates your instances when they are no longer needed, to free up resources. Swift handles the memory management of instances through *automatic reference counting* (*ARC*), as described in Automatic Reference Counting. Typically you don't need to perform manual cleanup when your instances are deallocated. However, when you are working with your own resources, you might need to perform some additional cleanup yourself. For example, if you create a custom class to open a file and write some data to it, you might need to close the file before the class instance is deallocated.

Class definitions can have at most one deinitializer per class. The deinitializer does not take any parameters and is written without parentheses:

```
deinit {
    // perform the deinitialization
}
```

Deinitializers are called automatically, just before instance deallocation takes place. You are not allowed to call a deinitializer yourself. Superclass deinitializers are inherited by their subclasses, and the superclass deinitializer is called automatically at the end of a subclass deinitializer implementation. Superclass deinitializers are always called, even if a subclass does not provide its own deinitializer.

Because an instance is not deallocated until after its deinitializer is called, a deinitializer can access all properties of the instance it is called on and can modify its behavior based on those properties (such as looking up the name of a file that needs to be closed).

Deinitializers in Action

Here's an example of a deinitializer in action. This example defines two new types, `Bank` and `Player`, for a simple game. The `Bank` class manages a made-up currency, which can never have more than 10,000 coins in circulation. There can only ever be one `Bank` in the game, and so the `Bank` is implemented as a class with type properties and methods to store and manage its current state:

```
class Bank {
    static var coinsInBank = 10_000
    static func distribute(coins numberOfCoinsRequested: Int) -> Int {
        let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
        coinsInBank -= numberOfCoinsToVend
        return numberOfCoinsToVend
    }
    static func receive(coins: Int) {
        coinsInBank += coins
    }
}
```

`Bank` keeps track of the current number of coins it holds with its `coinsInBank` property. It also offers two methods—`distribute(coins:)` and `receive(coins:)`—to handle the distribution and collection of coins.

The `distribute(coins:)` method checks that there are enough coins in the bank before distributing them. If there are not enough coins, `Bank` returns a smaller number than the number that was requested (and returns zero if no coins are left in the bank). It returns an integer value to indicate the actual number of coins that were provided.

The `receive(coins:)` method simply adds the received number of coins back into the bank's coin store.

The `Player` class describes a player in the game. Each player has a certain number of coins stored in their purse at any time. This is represented by the player's `coinsInPurse` property:

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
}
```

Each `Player` instance is initialized with a starting allowance of a specified number of coins from the bank during initialization, although a `Player` instance may receive fewer than that number if not enough coins are available.

The `Player` class defines a `win(coins:)` method, which retrieves a certain number of coins from the bank and adds them to the player's purse. The `Player` class also implements a deinitializer, which is called just before a `Player` instance is deallocated. Here, the deinitializer simply returns all of the player's coins to the bank:

```
var playerOne: Player? = Player(coins: 100)
print("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
// Prints "A new player has joined the game with 100 coins"
print("There are now \(Bank.coinsInBank) coins left in the bank")
// Prints "There are now 9900 coins left in the bank"
```

A new `Player` instance is created, with a request for 100 coins if they are available. This `Player` instance is stored in an optional `Player` variable called `playerOne`. An optional variable is used here, because players can leave the game at any point. The optional lets you track whether there is currently a player in the game.

Because `playerOne` is an optional, it is qualified with an exclamation mark (`!`) when its `coinsInPurse` property is accessed to print its default number of coins, and whenever its `winCoins(_:)` method is called:

```
playerOne!.win(coins: 2_000)
print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
// Prints "PlayerOne won 2000 coins & now has 2100 coins"
print("The bank now only has \(Bank.coinsInBank) coins left")
// Prints "The bank now only has 7900 coins left"
```

Here, the player has won 2,000 coins. The player's purse now contains 2,100 coins, and the bank has only 7,900 coins left.

```
playerOne = nil
print("PlayerOne has left the game")
// Prints "PlayerOne has left the game"
print("The bank now has \(Bank.coinsInBank) coins")
// Prints "The bank now has 10000 coins"
```

The player has now left the game. This is indicated by setting the optional `playerOne` variable to `nil`, meaning "no `Player` instance." At the point that this happens, the `playerOne` variable's reference to the `Player` instance is broken. No other properties or variables are still referring to the `Player` instance, and so it is deallocated in order to free up its memory. Just before this happens, its deinitializer is called automatically, and its coins are returned to the bank.