HACKING WITH SWIFT

SWIFT INTERVIEW CHALLENGES

REAL PROBLEMS, REAL SOLUTIONS

Prepare for iOS interviews with graded challenges and comprehensive explanations

FREE SAMPLE

Paul Hudson

Chapter 1
Foundations

The least you need to know about computer science to do well in coding interviews.

Arrays

We're going to start with the most common collection type of all: arraya. You might think this is all very easy, but most people will learn something here – be patient!

Swift is a language that recognizes arrays are common, so we make them with special syntax:

```
let numbers = [10, 20, 30]
```

Placing them in square brackets is like we're visually putting them in a box, which is a good representation of how arrays work.

Arrays provide two important features for us compared to the other two:

1. They use sequential storage, which means 20 always come after 10 in our array. This makes them important for anywhere the *order* of items matters.
2. We get to access items by their index, meaning that we can read **numbers[2]** to get back 30. Accessing items in this way is extremely fast, because Swift can jump straight to index 2 and return whatever is there.

Both of those sound obvious when you've used arrays for a while, but they are surprisingly complicated.

Think about it for a moment: when we say **numbers[2]**, how can Swift return that value so quickly?

Well, in Swift the default **Int** type is 64-bit. "Bit" is short for "binary digit", meaning that **Int** holds 64 1s and 0s to represent its maximum number. Even if a number is small, it still uses 64 bits of storage.

So, to jump to index 2, Swift can more or less put itself at the start our array, then move forward 64 bits for each array item offset we're reading through. That means index 0 is at 0x64, so it's at the beginning. Index 1 is 1x64, so Swift needs to move 64 bits forward from the start to get to the item. And then index 2 is 2x64, so Swift needs to move 128 bits forward

from the start.

So, reading **numbers[2]** is really just a multiplication problem – Swift multiplies the size of each item by the index we're reading, and jumps straight there.

But consider this code:

```
protocol Readable {
  var pageCount: Int { get set }
}

struct Leaflet: Readable {
  var pageCount: Int
}

struct Magazine: Readable {
  var pageCount: Int
  var name: String
  var date: Date
}

struct Book: Readable {
  var pageCount: Int
  var name: String
  var author: String
  var isbn: String
}
```

That creates a protocol and two conforming types. We can then make an instance of each type:

```
let cheeseMonthly = Magazine(pageCount: 100, name: "Cheese
Monthly", date: .now)
let lotr = Book(pageCount: 1000, name: "Lord of the Rings",
```

```
author: "JRR Tolkien", isbn: "111-1111-11")
```

And then put them into an array together:

```
var material: [any Readable] = [cheeseMonthly, lotr]
```

We can read **material[1]** to get back the Lord of the Rings, but *how*? We've put two objects into the array that have different sizes: one has an integer, a string, and a date, and the other has an integer and three strings.

This means a simple multiplication doesn't seem possible: if we assumed every object was the same size then multiplied that size by our index, we could end up in the middle of some other object entirely!

But here's the problem: it is critically important that array access be lightning fast – vast swathes of code relies on it being fast, and breaking that promise would cause most if not all programs to grind to a halt.

So, Swift has a solution: and it's implicitly attached to the **any** keyword. They are called *existential containers*, and are special boxes that hold a fixed amount of data.

If our object fits wholly in that box, great! Things like **Leaflet** are trivial, so fit neatly into the box. But if our data *doesn't* fit into the box, then Swift puts what it can into the box, and stores the rest elsewhere in main memory.

That might sound complex, but it's important because it means every array element has a fixed size, so our index multiplication jump works here too – Swift hides the complexity of existential containers for us, but in doing so ensures it sticks to the contract that array indexing should always be very fast.

But there's another interesting complexity hiding in here: in order for this approach to work, we can't have gaps in our data.

So, if we remove the first item in our array, we would expect **material[1]** to become

material[0]:

```
material.removeFirst()
print(material[0])
```

There's no fancy logic here: when removing the first item in an array, we need to move all the other items down one place. Similarly, if we insert a new object in the middle of an array, we need to move subsequent items down one place to make space for it.

For two objects in an array this work is trivial, but what about 10,000 objects? Or a million objects? Now, Swift needs to move a whole lot of objects, which isn't fast. This is why removing from the end of an array is always preferable to removing from the start, because removing from the end has no effect on other objects.

Sets

Set are a bit like unordered arrays that don't allow duplicates. We create sets from arrays like this:

```
var names = Set(["John", "Paul", "George", "Ringo"])
```

Ensuring that every item is unique is an interesting and valuable feature of sets, but their real power feature is that they don't store items in any fixed order.

So, if you try printing the first item in the set, it's basically random what comes back:

```
print(names.first)
```

This is because sets store their objects according to their *hash* values.

To understand hash values, imagine you were sorting a group of toys: you might say "small, red, fluffy toys go there, large, red, fluffy toys go there, medium-sized, blue, fluffy toys go *there*," and so on. Each toy can be given an exact location baed on its size, color, and material, so when someone says "I'm looking for a large, green, wooden toy," you don't need to look through 1000 items to find what they want.

Of course, Swift wants something more precise, otherwise we'd spend time arguing over whether **Int** was fluffy or not. It uses *hash values*, which are integers that can be produced from our data.

For simple types, Swift can calculate hash values for us simply by conforming to the **Hashable** protocol:

```
struct User: Hashable {
  var name: String
  var age: Int
}
```

Because both **String** and **Int** already conform to **Hashable**, Swift generates the hash value of **User** to be the hash of its name followed by the hash of its age.

You can write it yourself if you want:

```
struct User: Hashable {
  var name: String
  var age: String

  func hash(into hasher: inout Hasher) {
    hasher.combine(name)
    hasher.combine(age)
  }
}
```

hasher.combine() takes care of converting **name** and **age** to their hash values, so we might end up with a long integer number that represents this user.

Like I said, sets store objects according to their hash values, which is why the **Hashable** requirement is there. Our **User** type conforms to **Hashable**, so we can use it immediately:

```
let character1 = User(name: "Bernard", age: 19)
let character2 = User(name: "Laverne", age: 17)
let character3 = User(name: "Hoagie", age: 21)
var users = Set([character1, character2])
users.insert(character3)
```

Every time Swift adds an object to a set, it calculates the object's hash value and uses that as its storage location. This is why adding an item to a set is called *inserting* rather than *appending*, because we aren't just adding the item to the end of a line of other objects.

We can then check whether a set contains a particular object:

```
print(users.contains(character2))
```

Internally, Swift calculates the hash value of **character2**, and looks in that location. Imagine it like rooms in a storage facility: if Swift calculates the hash value of an object and figures out it should be stored in room 805, then we can figure out whether the object exists in a facility by checking that one room.

In comparison, if our storage facility was an array, the only way to check if an object exists we'd need to start at room 1, then go to room 2, then room 3, etc, until we had either found the object or exhausted every room. We call this a *linear search*, and you can imagine how slow it is if your storage facility had 10,000 units!

So, the key thing to remember with sets is that they provide a lightning-fast **contains()** method – if they have one item or a million items, it runs at the same speed.

Now, the **Hashable** protocol inherits from **Equatable**, which means we can write this:

```
print(character1 == character2)
```

To understand why **Hashable** requires **Equatable**, think about this: what happens if two **User** objects evaluate to the same hash value? It's entirely possible, and actually surprisingly common. Fortunately, there's a simple solution that's invisible to us, and it's why **Hashable** requires **Equatable**.

You see, if two items need to be stored in the same place, there's not a great deal Swift can do: it needs to store *both* items there. So, the hash value takes us the roughly the correct place, but there might be three possible items to return – which is the correct item?

To find that out, Swift uses **Equatable** to compare each item it finds against the exact one you're looking for – it runs == against them all, until one matches.

That might seem like we're going back to a slow linear search, but it's still significantly better, because we're searching through two or three items in one location rather than every possible room.

One last thing about sets: the **insert()** method actually returns data that describes what happened. We don't need to care about it because the method is marked as having a discardable result, but you can read it if you want:

```
let result = users.insert(character2)
print(result.inserted)
print(result.memberAfterInsert)
```

The **inserted** value will be true if the **insert()** call added the item to the set. If it's false, it **memberAfterInsert** will be the object that there was previously. Trust me, this is helpful when solving interview challenges!

Dictionaries

Dictionaries are key-value stores, which means we can store values in locations of our choosing by specifying a unique key. This means they have a *lot* in common with sets:

- Reading a value in a dictionary is extremely fast, no matter how many items it has.
- This is possible because dictionary keys must conform to **Hashable**, so Swift can find them quickly.
- Dictionaries always have unique values.
- Dictionaries don't have strictly have an order.

They might seem straightforward enough, but there are three interesting complexities in dictionaries.

First, removing an item from a dictionary can be done using **removeValue(forKey:)**. For example, we could create a **[String: Int]** dictionary to track exam results like this:

```
var results = [String: Int]()
results["Taylor"] = 80
results["Adele"] = 95
print(results)
```

Then we could remove a result using **removeValue(forKey:)**:

```
results.removeValue(forKey: "Taylor")
print(results)
```

But we can also remove an item by assigning **nil** to its value, like this:

```
results["Taylor"] = nil
print(results)
```

So far that probably sounds simple enough, but what would happen if we *wanted* to store **nil** in

a dictionary? We might want scores from 0 through 100 for students who have taken the test, and a score of **nil** for students who have yet to take it.

You might try this:

```
var results = [String: Int?]()
results["Taylor"] = 80
results["Adele"] = nil
print(results)
```

But that's not going to work – it will say Taylor has a score of Optional(80), but won't say Adele exists at all. If you *want* to store a **nil** value you need to write it out explicitly:

```
results["Adele"] = Int?.none
print(results)
```

The second interesting complexity with dictionaries is about the order of its items, because Swift explicitly states that the order of dictionaries is stable *between mutations* – the order changes when you modify items, but will be fixed otherwise.

So, printing the **results** dictionary lots of times without changing it will show the same output repeatedly, but changing it will change the order too:

```
var items = ["One": 1, "Two": 2, "Three": 3]
print(items)
print(items)
print(items)
print(items)

items["Four"] = 4
print(items)
```

In my experience the same is true of sets too, but Swift doesn't make the same promise there.

The third interesting dictionary is its restriction that keys must be unique. It's an important feature of dictionaries, but it causes folks to trip up with code like this:

```
let mapped = items.map(\.self)
```

Mapping **\.self** should do nothing at all, but what we went sent back isn't a dictionary – **mapped** will actually be an array of tuples.

This happens because dictionaries must always have unique keys, and **map()** allows you to transform both the key and the value at the same time. So, there's nothing stopping you from applying a transform that returns conflicting keys, so should Swift just discard the duplicates, and which value should be retained from the duplicate keys?

Because of this risk, Swift provides a special method for dictionaries called **mapValues()**, which transforms values while leaving keys alone. This makes it safe to apply your transform because duplicate keys can't happen, which means this will result in a new dictionary:

```
let result = items.mapValues { $0 * 2 }
print(result)
```

Putting these complexities to one side, there is one thing dictionaries are excellent at, and you'll use a *lot*: counting items. This is an extremely common starting point in interviews, so it's something you'll definitely want to be familiar with.

Dictionaries work so well for counting for several reasons:

1. Their lightning-fast look up for keys means we can locate existing items instantly.
2. We can provide a default value to be given back if a key doesn't exist.
3. That default value is mutable, so we can modify it in place.

So, we can write count letters in a string like this:

```
var message = "Mississippi"
var counts = [Character: Int]()
```

```
for letter in message {
  counts[letter, default: 0] += 1
}

print(counts)
```

The mutability of the default value is really helpful for things like arrays. For example, if we had an array of cities like this:

```
let cities = ["London", "Lisbon", "Tokyo", "Toronto"]
```

We might want to group them into a dictionary like this one:

```
var grouped = [Character: [String]]()
```

That would store the first letter of each city as the dictionary key, and an array of all the matching cities for that letter as the value.

```
for city in cities {
  grouped[city.first!, default: []].append(city)
}
```

That uses the city's first letter as the key, and an empty array as the default value if the key doesn't exist. Thanks to the default value being mutable, we can call **append()** directly on the result – it will add the city to an existing array if there is one, or create an empty array first *then* add it there otherwise.

That particular form of counting is so popular that Swift even comes with a built-in version for easier access:

```
let grouped = Dictionary(grouping: cities) { $0.first! }
print(grouped)
```

So, that's dictionaries – like arrays and sets they might seem simple on the surface, but it's important to really understand them through and through.

Stacks, queues, and deques

There are three array-like data types that are common in computer science, but don't exist natively in Swift itself: they are stacks, queues, and deques, with the latter pronounced "deck".

Stacks are a simple structure that allow us to access only one item at a time, which is the last object that was added. If you want to access earlier objects, you need to remove later objects first.

Adding things to the stack is known as *pushing* an object onto the stack – you can imagine placing them one above the other as if you were stacking up building blocks. Removing items from the stack is called *popping*, and always removes whatever is the last or *top* item on the stack.

This approach means you can't get to an object that isn't at the top of the stack, so the first thing you add will be the last thing you remove. We call this as last-in first-out structure, or just LIFO for short.

We can implement a generic stack in Swift very easily:

```
struct Stack<T> {
  private var items: [T]

  init(_ items: [T] = []) {
    self.items = items
  }

  mutating func push(_ object: T) {
    items.append(object)
  }

  mutating func pop() -> T? {
    items.popLast()
  }
```

```
}
```

That can store any kind of items, but the internal array is marked private so users must access it using the **push()** and **pop()** methods.

We can use it like this:

```
var numbers = Stack([1, 3, 5, 7])
print(numbers.pop())
print(numbers.pop())
print(numbers.pop())
print(numbers.pop())
```

Although Swift doesn't have native support for a **Stack** type, they are extremely common in computer science:

- Function calls are arranged as a stack. If one function calls another, it gets pushed onto a stack and made the active function. When the new function finishes, it gets popped off the stack and now the active function is the previous one again.
- Apps that implement undo and redo can use stacks too – it wouldn't make sense to be able to leave a later change intact, while undoing an earlier change, because it could cause havoc!
- Web browsers can use stacks to implement a "Back" history and a "Forward" history – two stacks that move items between them as the user navigates.

Yes, they are just specialized arrays, but remember that removing items from the end of an array is really fast because nothing else needs to move around.

Alongside stacks are *queues*, which are first-in first-out structures, or FIFO for short. Queues are a bit like stacks except the item you remove must come from the front of the queue, like waiting in line at a grocery store checkout.

You might think queues are easy to implement in Swift:

```
struct Queue<T> {
  private var items: [T]

  init(_ items: [T]) {
    self.items = items
  }

  mutating func enqueue(_ object: T) {
    items.append(object)
  }

  mutating func dequeue() -> T? {
    if items.isEmpty { return nil }
    return items.removeFirst()
  }
}
```

We could then use it like this:

```
var queue = Queue<Int>()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.dequeue())
print(queue.dequeue())
print(queue.dequeue())
```

But remember: calling **removeFirst()** is really slow in Swift. A slightly more efficient approach is to have two arrays internally, like this:

```
struct Queue<T> {
  private var enqueueStack: [T]
  private var dequeueStack: [T]
```

```
  init(_ items: [T] = []) {
    enqueueStack = items
    dequeueStack = []
  }

  mutating func enqueue(_ element: T) {
    enqueueStack.append(element)
  }

  // more code to come
}
```

It's missing a **dequeue()** method, because that's where the cunning part comes in: rather than removing from the start of one array, we instead *reverse* our items and remove from the end instead.

If we only do this when our **dequeueStack** array is empty, then we avoid repeating work unnecessarily.

Here's how that looks:

```
mutating func dequeue() -> T? {
  if dequeueStack.isEmpty {
    dequeueStack = enqueueStack.reversed()
    enqueueStack.removeAll()
  }

  return dequeueStack.popLast()
}
```

The last array-like data type you'll commonly see is called a *deque* (pronounced "deck"), which is short for "double-ended queue." Deques can add or remove items from either end of

its array, and are useful when working with things like sliding windows – when you're working with a range of data, and need to know the largest and smallest value in that range.

I'm not going to implement a **Deque** here because it's one of the challenges later on, but hopefully you get the idea!

Trees and binary trees

Trees are a data structure where each node in the tree has child nodes, and each child can have its own children, and so on. The name comes from real-life trees, where there is one trunk with a handful of thick branches coming off it, and each branch has smaller branches, and each of those has smaller branches, and so on.

A simple tree implementation is really just a single struct, generic over some kind of value:

```
struct Node<Value> {
  var value: Value
  private(set) var children: [Node]

  init(_ value: Value, children: [Node] = []) {
    self.value = value
    self.children = children
  }

  mutating func add(child: Node) {
    children.append(child)
  }
}
```

As you can see, each **Node** has its own array of **Node** objects called **children**. I've marked it **private(set)** so that we ned to use the **add(child:)** method rather than poking around in the array directly.

We can then go ahead and make some **Node** instances:

```
var a = Node("a")
var b = Node("b")
var c = Node("c")
var d = Node("d")
```

```
var e = Node("e")
var f = Node("f")
```

Then we can start to link them up. For example, we could say that **d**, **e**, and **f** belong to **c**:

```
c.add(child: d)
c.add(child: e)
c.add(child: f)
```

Then we could make **b** and **c** belong to **a**:

```
a.add(child: b)
a.add(child: c)
```

You can see how they fit together using **dump()**, which shows indenting for the children:

```
dump(a)
```

Trees are a neat way to organize structured data, such as filesystems: you can have folders inside folders inside folders, and they are easy to navigate around.

Many tree implementations have a second type to wrap around individual nodes, which at the very least would contain the root of the tree:

```
class Tree<T> {
  var root: Node<T>?
}
```

Having this extra class is a great place to add common methods such as traversing the whole tree, but a lot of those things could equally well go into the nodes instead. For example, if you were to make tree traversal a method on your nodes rather than on the whole tree, it would allow you to traverse any subtree.

Binary trees are a specialized form of trees, and are particularly when it comes to interview

challenges. Rather than having an array of children, nodes in in a binary tree have **left** and `right properties that store one node each.

If we wanted to convert our current **Node** into a binary tree, we'd use this:

```
class Node<Value> {
  var value: Value
  var left: Node?
  var right: Node?

  init(_ value: Value) {
    self.value = value
  }
}
```

That needs to be a class now, because it directly references itself – there isn't an array of **Node** children, but instead 0, 1, or 2 depending on the values. If we didn't make this a class, each **Node** would need to allocate enough space for two more nodes, each of which would need to allocate space for two more nodes, and so on forever. Classes break that chain, because we're really just storing a memory reference to the **left** and **right** child nodes.

Now we can piece together our nodes like this:

```
c.left = b
b.left = a

c.right = e
e.left = d
e.right = f

dump(c)
```

You might think that ordering is quite arbitrary, but it isn't: I've placed them in such a way that

what we *actually* have is called a binary search tree, or just BST for short.

Regular binary trees have two children, but items can be placed anywhere in the tree. In a binary search tree, the left child should be sorted equal to or before the parent, and the right child should be sorted after the parent.

So, in our example we have **c** being the root node, meaning that **b** is placed to its left. To the left of **b** comes **a**, and to the right of **c** comes **e**. Finally, we have **d** and **f** to the left and right or **c** respectively – all sorted correctly.

These BSTs extremely efficient, because you can find data through a series of comparisons. If we want to know whether "a" exists in the tree, we'd start at **c**, the root node, then move to the left because we know that "a" comes before "c". We'd reach "b", and again move to the left because we know that "a" comes before "a". And then we'd find our answer – we wouldn't need to go near **d**, **e**, and **f**, because those are all on the wrong side of the tree.

This approach is called a *binary search*, because we discard half the possibilities every time. It's extremely efficient because it discards impossible options very quickly.

In an ideal binary tree each possibility is balanced, so each time we choose a path we eliminate half the options. In practice, binary trees are often *unbalanced* because one side contains more nodes than the other.