Method Dispatch

As a developers, we should be very particular about what kind of function we should write which will be faster enough in terms of execution. We should not rely on and say that Swift is anyways faster than OBJ-C; then what else should we do? Well, here are topic that pops up.

As the definition says, Method dispatch is how a program selects which instructions to execute when invoking a method. It's something that happens every time a method is called and something that you tend to think a lot about.

Compiled programming languages have **three** primary methods of dispatch.

- Static or direct dispatch
- Table or dyanamic dispatch
- Message dispatch

Direct Dispatch:

Direct dispatch is the fastest style of method dispatch. Not only does it result in the fewest number of assembly instructions, but the compiler can perform all sorts of smart tricks, like inlining code, and many more things which are outside of the scope of article. This often referred to as static dispatch.

However, direct dispatch is also the most restrictive from a programming point of view, and it is not dynamic enough to support subclassing.

Structs (Value type Data type)

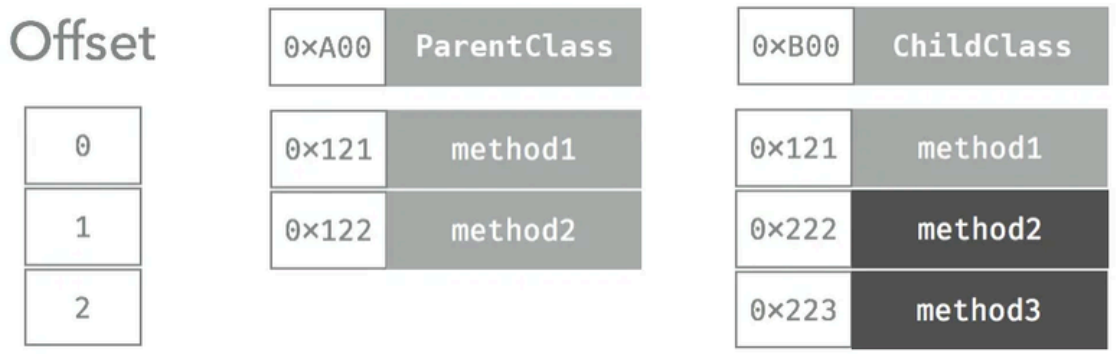
Static and final (Reference type with this keyword)

Table Dispatch or Dynamic Dispatch

Table dispatch is the most common implementation of dynamic dispatch behaviour in compiled languages. Table dispatch uses an **array** of function pointers for each method in the class declaration. Most languages refers to this as "**virtual table**", but Swift uses the terms "**witness table**". Every subclass has its own copy of the table with a different function pointer for every method that the class has overridden.

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let obj = ChildClass()
        obj.method2()
    }
}
class ParentClass {
    func method1() {}
    func method2() {}
}
class ChildClass: ParentClass {
    override func method2() {}
    func method3() {}
}
```

This code will create array of function address, as subclasses add new methods to the class, those methods are appended to the end of this array. This table is then consulted at runtime to determine the method to run.



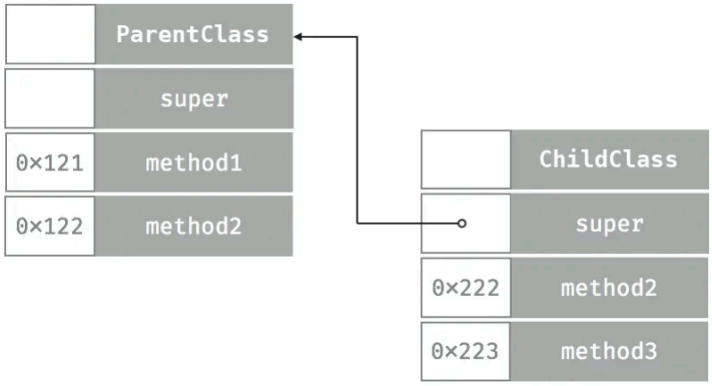
However this method of dispatch is still slow compared to direct dispatch. From a byte-code point of view, there are two additional reads and a jump, which contribute some overhead. However, another reason this is considered slow is that the compiler can't perform any optimisations based on what is occurring inside the method.

One downfall of this array-based implementation is that extensions cannot extend the dispatch table. Since subclasses add new methods to the end of the dispatch table, there's no index that an extension can safely add function pointer to.

Message Dispatch:

Though this Message dispatch technique is the most dynamic type, this is slowest of all other dispatch types. **In Table Dispatch the witness table is getting generated on compile time** but in message dispatch this not possible to decide which method to be called since it might have swizzled or new methods might have been added during runtime. However Cocoa frameworks use it inside a lot of its big players like KVO, Core Data, and other things.

Also, it enables method swizzling, which generally means that using this technique, we can change the functionality of method at runtime.



When a message is dispatched, the runtime will crawl the class hierarchy to determine which method to invoke. This is not really very slow as it has been implemented with high performance cache.

The compiler always tries to upgrade the dispatch technique to static dispatch unless we have explicitly marked it with the `dynamic`, `@objc` keyword.

let's understand this by below code.

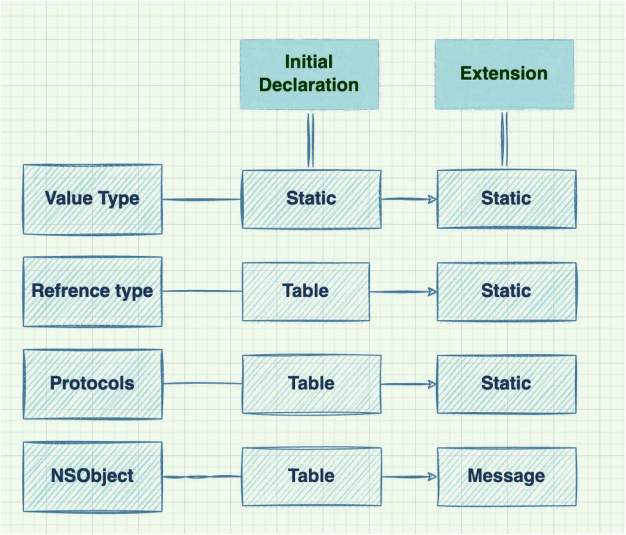
```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        hello(person: ParentClass())
        hello(person: ChildClass())

    }
}
class ParentClass: NSObject {
    func sayHi() {
        print("Hi From ParentClass")
    }
}
func hello(person: ParentClass) {
    person.sayHi()
}

class ChildClass: ParentClass {}
extension ChildClass {
    override func sayHi() {   2  Non-@objc instance method 'sayHi()' declared in 'ParentClass' cannot be overridden from extension
        print("Hi From ChildClass")
    }
}
```

In code we can see error saying the function cannot be overridden in the extension. This is because `sayHi()` is declared in extension meaning that method will be invoked with message dispatch. When `hello(person:)` is invoked, `sayHi()` is dispatched to the `ParentClass` object via table dispatch. Since the `ChildClass` override was added via message dispatch, the dispatch

table for `ChildClass` still has the `ParentClass` implementation in the dispatch table hence confusion happens and you will get an error. To make this work we can either move this extension method to `ChildClass` and not in extension `ChildClass`.



There are few things to note here:

- Value types always use direct dispatch. Nice and easy!
- Extensions of protocols and classes use direct dispatch.
- NSObject extensions use message dispatch.
- NSObject uses table dispatch for methods inside the initial declaration!
- Default implementation of methods in the initial protocol declaration use table dispatch.