Orientação a Objetos:

• Encapsulamento: objetos "protegem" seu estado e as ações que podem ser realizadas sobre eles com o uso de modificadores de visibilidade. Em Swift, essa visibilidade pode ter 5 níveis de acesso: public, open, internal, fileprivate ou private. Dessa forma, outros objetos devem enviar mensagens a esse objeto e caso seja desejado, ele pode mudar seu estado interno.

• Herança: pode-se construir uma árvore hierárquica de comportamento com o uso de herança. Isso quer dizer que é possível definir uma classe base (pai) e caso especializações dessa classe (filhas) existam, pode-se utilizar comportamentos da base, sobrescrevê-los ou até mesmo complementá-los com comportamentos únicos à essas especializações. Importante destacar que a linguagem Swift não suporta herança múltipla, ou seja, uma classe filha pode herdar de apenas uma classe pai.

• Polimorfismo: expressão que significa múltiplas formas. Classes derivadas de uma única classe base são capazes de invocar métodos que se comportam de maneira diferente para cada uma das classes derivadas, apesar de possuírem a mesma assinatura. Sendo assim, para quem chama o método é indiferente a implementação interna que cada uma dessas classes derivadas fornece, o que interessa é o resultado que pode ser obtido com manipulações diferentes do estado do objeto para cada uma delas.

1) Classes E Objetos:

* Uma classe, na orientação a objetos, é um template para os nossos objetos. São as classes que definem o estado (propriedades) e as ações (métodos);

* Para definir uma classe basta utilizar a palavra-chave "class";

```
class FiguraGeometrica {
}

let quadrado = FiguraGeometrica()
var circulo = FiguraGeometrica()
```

2) Propriedades:

* As propriedades em Swift podem ser enxergadas como variáveis e constantes dentro das classes (e consequentemente dos objetos que são as instâncias dessas classes).

```
class Bicicleta {
    let rodas = 2
    var dono: String

    init(dono: String) {
        // utilizamos "self.dono" para se referir a propriedade
        // já que somente "dono" se refere ao parametro String
        // do construtor
        self.dono = dono
    }
}
```

```
class Bicicleta {
    let rodas = 2
    var dono: String

    init(dono: String) {
        // utilizamos "self.dono" para se referir a propriedade
        // já que somente "dono" se refere ao parametro String
        // do construtor
        self.dono = dono
    }
}

let bicicleta = Bicicleta(dono: "João") // Instanciamos a bicicleta de João.

print("A bicicleta de \(bicicleta.dono) tem \(bicicleta.rodas) rodas")

// Será impresso: "A bicicleta de João tem 2 rodas"

// Suponha que João venda sua bicicleta para Matheus, podemos representar
// isso em nosso programa alterando o dono de bicicleta. Perceba que não
// atribuímos uma nova bicicleta à constante, algo que ocasionaria um erro,
// apenas alteramos o estado do objeto bicicleta, alterando sua propriedade
// nome.

bicicleta.dono = "Matheus"

print("A bicicleta de \(bicicleta.dono) tem \(bicicleta.rodas) rodas")

// Será impresso: "A bicicleta de Matheus tem 2 rodas"
```

3) Métodos:

* Métodos são as formas como adicionamos comportamentos (ações) aos nossos objetos e classes.

```
class Bicicleta {
    let rodas = 2
    var dono: String

    init(dono: String) {
        // utilizamos "self.dono" para se referir a propriedade
        // já que somente "dono" se refere ao parametro String
        // do construtor
        self.dono = dono
    }

    func emprestar(para: String, horas: Int) {
        print("A bicicleta de \(dono) foi emprestada para \(para) por \(horas) horas")
    }
}

// Vamos instanciar uma bicicleta e emprestá-la
let b = Bicicleta(dono: "Matheus")
b.emprestar(para: "João", horas: 2)

// Será impresso: "A bicicleta de Matheus foi emprestada para João por 2 horas"
```

4) Herança:

* Com ela podemos ter classes que herdam comportamentos, propriedades e outras características de outras classes. Quando uma classe herda da outra, chamamos a classe filha de sub-classe e a classe pai de super-classe. Herança é uma das principais características que diferenciam as classes de outros tipos em Swift.

```
class FormaGeometrica {
    func descricao() {
        print("Decrição de uma forma geométrica")
    }
}

class Quadrado: FormaGeometrica {
    var tamanho: Int

    init(tamanho: Int) {
        self.tamanho = tamanho
    }

    func area() -> Int {
        return tamanho * tamanho
    }
}

let quadrado = Quadrado(tamanho: 2)
let area = quadrado.area()

print("Área do quadrado é \(area)")
// Será impresso: "Área do quadrado é 4"

quadrado.descricao()
// Será impresso: "Decricao de uma forma geométrica"
```

5) Protocolos:

* Sua proposta é estabelecer um contrato entre quem utiliza um determinado objeto de forma que o cliente não dependa do tipo, mas sim, do comportamento.

```
protocol OperacaoMatematica {
    func calcular(x: Double, y: Double) -> Double
}

class Soma: OperacaoMatematica {
    func calcular(x: Double, y: Double) -> Double {
        return x + y
    }
}

class Subtracao: OperacaoMatematica {
    func calcular(x: Double, y: Double) -> Double {
        return x - y
    }
}
```

6) Extensões:

* Tratam-se de estruturas que permitem que qualquer classe (seja ela definida pelo desenvolvedor ou pelos frameworks) do programa Swift seja "reaberta" e métodos sejam adicionados a ela.

```
extension String {
    func onlyVogals() -> String {
        var vogals = ""

        for c in self {
            let letter = "\(c)"
            if (letter == "a" || letter == "e" || letter == "i" ||
                letter == "o" || letter == "u") {
                vogals += letter
            }
        }

        return vogals
    }
}
```