Concorrência

Programa Exemplo

Faremos um programa que simula as operações de um *food truck* como apresentado na aula, como uma metáfora a programação concorrente.

Programação síncrona

Vamos iniciar a programação do código de pedidos de sanduíches, simulando uma nova ordem com os botões da parte superior da tela. Ambos estão associados com a *action order Sandwich*:

```
@IBAction func orderSandwich(_ sender: UIButton) {

    let type = SandwichType(rawValue: sender.tag)

    if let sandwichType = type {

    }

}
```

Este código obtém da tag associada a cada botão o tipo do sanduíche a ser preparado. Vamos agora incluir a lógica de "preparação" do sanduíche. Ta lógica está implementada na função *prepareOrder* de **KitchenServices** (que está basicamente buscando uma imagem de uma URL de forma síncrona, para representar algo demorado). Vamos utilizar este método e atualizar a *collection view*:

```
@IBAction func orderSandwich(_ sender: UIButton) {

    let type = SandwichType(rawValue: sender.tag)

    if let sandwichType = type {
        ticketNumber += 1
        let order = KitchenServices.prepareOrder(sandwichType,
                                                 ticketNumber: ticketNumber)
        if (order != nil) {
            orderList.insert(order!, at: 0)
            orderCollection.reloadData()
        }
    }

}
```

Este código, ao ser executado, gera uma UI não responsiva quando a operação longa de "preparação" do sanduíche é realizada. Isto gera a sensação de app "travada" e frustra o usuário!

Operation queue e *Operations*

Vamos agora implementar esta operação deixando não mais a *thread* de UI cuidando da tarefa de loga duração. Vamos criar uma fila para cuidar disto, e manter uma referência para a fila de UI (*main*):

```
//MARK: - Queues
let kitchenQueue = OperationQueue()
let mainQueue = OperationQueue.main
```

Com isto podemos colocar o bloco de código demorado dentro da fila da cozinha, liberando a *thread* de UI para continuar atendendo o usuário!

```
@IBAction func orderSandwich(_ sender: UIButton) {

    let type = SandwichType(rawValue: sender.tag)

    if let sandwichType = type {
        ticketNumber += 1

        kitchenQueue.addOperation {
            let order = KitchenServices.prepareOrder(sandwichType,
                                                     ticketNumber: self.ticketNumb
er)
            if (order != nil) {
                self.orderList.insert(order!, at: 0)
            }
        }
    }
}
```

Porém, após a execução da operação e inclusão da ordem na lista de pedidos já entregues, precisamos atualizar a interface. Isto **não** pode ser feito por outra *thread* que não a de UI. Por isso precisamos fazer esta operação na fila **main**, sincronizada ao fim da preparação:

```
@IBAction func orderSandwich(_ sender: UIButton) {

    let type = SandwichType(rawValue: sender.tag)

    if let sandwichType = type {
        ticketNumber += 1

        kitchenQueue.addOperation {
            let order = KitchenServices.prepareOrder(sandwichType,
                                                     ticketNumber: self.ticketNumb
er)
            if (order != nil) {
                self.orderList.insert(order!, at: 0)
                self.mainQueue.addOperation {
                    self.orderCollection.reloadData()
                }
            }
        }
    }
}
```

Veja que nisto sequer temos a garantia da ordem de entrega dos itens, se forem pedidos tipos diversos muito rápido.

Temos referência fortes a *self* no bloco, gerando um *retain cycle*. Para quebrar este ciclo a referência a *self* deve ser enfraquecida, utilizando as palavras chaves *unowned* ou *weak*:

```
@IBAction func orderSandwich(_ sender: UIButton) {

   let type = SandwichType(rawValue: sender.tag)

   if let sandwichType = type {
       ticketNumber += 1

       kitchenQueue.addOperation { [unowned self] in
           let order = KitchenServices.prepareOrder(sandwichType,
                                                    ticketNumber: self.ticketNumb
er)
           if (order != nil) {
               self.orderList.insert(order!, at: 0)
               self.mainQueue.addOperation { [unowned self] in
                   self.orderCollection.reloadData()
               }
           }
       }
   }
}
```

Operations

Nesta etapa criaremos operações não mais utilizando *closures* e sim uma classe que extende **Operation**, definindo as atividades que serão executadas e atributos para configurar seus parâmetros.

A estrutura de uma **Operation** lembra o padrão de projeto **Command**. Ele possui um método que executa o comando/ação (no caso *main*) e pode conter construtores ou atributos caso necessário.

Criaremos nesta etapa três operações: **KitchenOperation**, **ToyOperation** e **AssembleOperation**. A primeira cuidará do "preparo" do sanduíche, a segunda do "preparo" do brinde (simulada como a obtenção de uma imagem da internet e aplicação de uma máscara sobre ela) e a terceira de fazer a montagem da ordem final (no caso mesclando as imagens criadas).

O código que efetivamente executa estas operações, de forma síncrona, está nas classes de serviço. Vamos criar as operações que as utilizam, iniciando com a **KitchenOperation**:

```
class KitchenOperation: Operation {

    var order:Order!
    var sandwichType:SandwichType
    var ticketNumber:Int
    init (type:SandwichType, number:Int) {
        sandwichType = type
        ticketNumber = number
    }

    override func main() {
        order = KitchenServices.prepareOrder(sandwichType,ticketNumber: ticketNum
ber)
    }

}
```

Faremos algo similar com a classe **ToyOperation**:

```
class ToyOperation: Operation {

    var toyImage:UIImage!

    override func main() {
        toyImage = ToyServices.prepareToy()
    }
}
```

A última tarefa é mesclar o lanche e o brinde para gerar o pedido final. Temos que definir esta dependência, o que faremos na inicialização da operação, utilizando seu método *addDependency*. Podemos utilizar informações desats operações depois pela propriedade *dependencies*:

```
class AssemblyOperation: Operation {


    init(_ kitchen: KitchenOperation, _ toy: ToyOperation) {

        super.init()

        self.addDependency(kitchen)
        self.addDependency(toy)

    }

    override func main() {
        //Relavant info
        var originalOrder: Order!
        var toyImage:UIImage!
        //Get info from previous operation
        for operation in self.dependencies {
            let kitchenOp = operation as? KitchenOperation
            if let kitchen = kitchenOp {
                originalOrder = kitchen.order
            }
            let toyOp = operation as? ToyOperation
            if let toy = toyOp {
                toyImage = toy.toyImage
            }
        }
        //Assemble final order
        let order = ToyServices.assembleToy(originalOrder, toyImage: toyImage)
    }
}
```

Até este ponto fomos capazes de criar uma ordem mesclando o resultado das operações anteriores (após o término das mesmas). Porém precisamos notificar a aplicação do fim da operação desta montagem, para podermos utilizar o resultado obtido. Para tanto, vamos criar uma *callback*, associá-la no inicializador e chamá-la ao fim da execução que avisará o resultado para que este seja apresentado:

```
var completionCallbackFunction:(Order!)->Void


init(_ kitchen: KitchenOperation, _ toy: ToyOperation, completion: @escaping (Ord
er!) -> Void) {


   completionCallbackFunction = completion

   super.init()

   self.addDependency(kitchen)
   self.addDependency(toy)
}

override func main() {
   ...
   //Assemble final order
   let order = ToyServices.assembleToy(originalOrder, toyImage: toyImage)
   self.completionCallbackFunction(order)
}
```

Nota: A cláusula **@escaping** indica que uma *closure* passada como parâmetro em uma função será chamada após esta função ter retornado (no caso, o nosso inicializador) ou seja, que ele "escapa" do escopo do corpo desta função. Por padrão em Swift 3 é esperado que uma *closure* passada por parâmetro exista apenas no escopo da função (motivo pelo qual a clásula **@noescape** se tornou *deprecated*) e que avisemos quando isto não for verdade (e seremos alertados pelo compilador de tal).

Vamos criar agora uma fila a mais para as atividades associadas ao brinde na classe **FoodTruckViewController**:

```
//MARK: - Queues
let kitchenQueue = OperationQueue()
let mainQueue = OperationQueue.main

let toyQueue = OperationQueue()
```

E alterar o processamento do pedido de acordo com estas classes:

```
@IBAction func orderSandwich(_ sender: UIButton) {

   let type = SandwichType(rawValue: sender.tag)

   if let sandwichType = type {
       ticketNumber += 1

       let prepareOrderOperation = KitchenOperation(type: sandwichType, number: t
icketNumber)
       let toyOperation = ToyOperation()

       let assemblyOperation = AssemblyOperation(prepareOrderOperation,toyOperati
on) { [unowned self]  (order) in

           self.orderList.insert(order, at: 0)
           self.mainQueue.addOperation { [unowned self] in
               self.orderCollection.reloadData()
           }

       }

       kitchenQueue.addOperation(prepareOrderOperation)
       toyQueue.addOperation(toyOperation)
       toyQueue.addOperation(assemblyOperation)

   }
}
```

Activity Indicator

Para melhorar o *feedback* ao usuário, vamos utilizar o *activity indicator* para indicar que as tarefas demoradas (e sem duração prevista) estão ocorrendo.

Primeiramente precisamos fazer com que as atividades possam avisar seu término, utlizando o mesmo mecanismo de *callback* que fizemos para a **AssemblyOperation**:

```
class KitchenOperation: Operation {

    var completionCallbackFunction:()->Void

    var order:Order!
    var sandwichType:SandwichType
    var ticketNumber:Int
    init (type:SandwichType, number:Int, completion:@escaping () -> Void) {
        sandwichType = type
        ticketNumber = number
        completionCallbackFunction = completion
    }

    override func main() {
        order = KitchenServices.prepareOrder(sandwichType,ticketNumber: ticketNum
ber)
        completionCallbackFunction()
    }

}
```

```
class ToyOperation: Operation {

    var toyImage:UIImage!

    var completionCallBackFunction:()->Void

    init(completion:@escaping ()->Void) {
        completionCallBackFunction = completion
    }

    override func main() {
        toyImage = ToyServices.prepareToy()
        completionCallBackFunction()
    }

}
```

E podemos alterar a ação *orderSandwich* para atualizar a UI de acordo com estas *callbacks*:

```
@IBAction func orderSandwich(_ sender: UIButton) {

    let type = SandwichType(rawValue: sender.tag)

    if let sandwichType = type {
        ticketNumber += 1

        //Increment Counters
        kitchenOperationCounter += 1
        toyOperationCounter += 1
        assembleOperationCounter += 1
        updateOperationsOnUI()

        let prepareOrderOperation = KitchenOperation(type: sandwichType, number: t
icketNumber) { [unowned self] in
            self.kitchenOperationCounter -= 1
            self.mainQueue.addOperation {
                self.updateOperationsOnUI()
            }
        }

        let toyOperation = ToyOperation() { [unowned self] in
            self.toyOperationCounter -= 1
            self.mainQueue.addOperation {
                self.updateOperationsOnUI()
            }
        }

        let assemblyOperation = AssemblyOperation(prepareOrderOperation,toyOperati
on) { [unowned self]  (order) in
            self.assembleOperationCounter -= 1
            self.orderList.insert(order, at: 0)
            self.mainQueue.addOperation { [unowned self] in
                self.updateOperationsOnUI()
                self.orderCollection.reloadData()
            }

        }

        kitchenQueue.addOperation(prepareOrderOperation)
        toyQueue.addOperation(toyOperation)
        toyQueue.addOperation(assemblyOperation)

    }
}
```