```
//
// Copyright (C) 2015 Frank Steiler <frank@steilerdev.de>
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 2 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
//

//
//  MarshalOperator.swift
//  This file contains the definition of the Marshal operator, that is
    intended to simplify multi threading using GCD, by providing a thin
    abstraction layer, while still hiding the underlying C API.
//
//  The marshal operator (~>) is an in-, pre- and postfix operator used on
    closures. The operator was first suggested by Josh Smith, whose initial
    implementation is the foundation for this extension to the Swift
    programming language: http://ijoshsmith.com/2014/07/05/custom-threading-
    operator-in-swift/.
//
//  The idea is that everything in front of the operator is executed on a
    serial background queue, while everything behind the operator is executed
    on the main queue. As an infix operator the background task is executed
    first and the task run on the main queue afterwards. There are variations
    that offer the passing of a variable from one closure to the other, as
    well as a background closure provided with a ManagedObjectContext usable
    by the background queue.
//
//  Another operator is the inverted marshal operator (<~), used postfix only.
    As soon as the closure is finished executing (and therefore hitting the
    operator) it's execution is repeated after 5 minutes. With the infix
    operator '<~/' the user is able to specify the amount of time in seconds
    between executions through the right hand value.
//

import Foundation
import CoreData
import UIKit

// Defining the operators
infix operator ~> {}
prefix operator ~> {}
postfix operator ~> {}

postfix operator <~ {}
infix operator <~/ {}

// Defining a typealias to hide the C API on return values
typealias MarshalThreadingObject = dispatch_source_t

// The queue used by the marshal operator.
```

```
private let marshalQueue: dispatch_queue_t =
    dispatch_queue_create("de.steilerdev.myverein.marshal-queue",
    DISPATCH_QUEUE_SERIAL)

/// Using the Marshal operator as a prefix operator, means that the closure is
    executed on the main queue.
///
/// :param: mainClosure The closure executed on the main queue.
prefix func ~> (mainClosure: () -> ()) {
  dispatch_async(dispatch_get_main_queue(), mainClosure)
}

/// Using the Marshal operator as a postfix operator, means that the closure
    is executed on a background queue.
///
/// :param: backgroundClosure The closure executed on a background thread.
postfix func ~> (backgroundClosure: () -> ()) {
  dispatch_async(marshalQueue, backgroundClosure)
}

// Using the Marshal operator as a postfix operator, means that the closure is
    executed on a background queue. This operator provides a managed object
    context created on the background queue and can therefore be used without
    any concurency problems. The core data objects (especially the persistent
    store coordinator) need to be handled through the app delegate.
///
/// :param: backgroundClosure The closure executed on a background thread
    providing a managed context usable on this thread.
postfix func ~> (backgroundClosure: (NSManagedObjectContext) -> ()) {
  dispatch_async(marshalQueue) {
    var backgroundContext = NSManagedObjectContext()
    backgroundContext.persistentStoreCoordinator = (UIApplication.
        sharedApplication().delegate as! AppDelegate).
        persistentStoreCoordinator!
    backgroundClosure(backgroundContext)
  }
}

/// Executes the left-hand closure on the background queue, upon completion
    the right-hand closure is executed on the main queue. This operator
    provides a managed object context created on the background queue and can
    therefore be used without any concurency problems.
///
/// :param: backgroundClosure The closure executed on a background thread,
    providing a managed context usable on this queue.
/// :param: mainClosure The closure executed on the main thread, after the
    background thread is finished.
func ~> (backgroundClosure: (NSManagedObjectContext) -> (), mainClosure: () ->
    ()) {
  dispatch_async(marshalQueue) {
    var backgroundContext = NSManagedObjectContext()
    backgroundContext.persistentStoreCoordinator = (UIApplication.
        sharedApplication().delegate as! AppDelegate).
        persistentStoreCoordinator!
    backgroundClosure(backgroundContext)
    dispatch_async(dispatch_get_main_queue(), mainClosure)
  }
}

/// Executes the left-hand closure on the background queue, upon completion
```

```
    the right-hand closure is executed on the main queue.
///
/// :param: backgroundClosure The closure executed on a background thread.
/// :param: mainClosure The closure executed on the main thread, after the
    background thread is finished.
func ~> (backgroundClosure: () -> (), mainClosure: () -> ()) {
  dispatch_async(marshalQueue) {
    backgroundClosure()
    dispatch_async(dispatch_get_main_queue(), mainClosure)
  }
}

/// Executes the left-hand closure on the background queue, upon completion
    the right-hand closure is executed on the main queue using the return
    value of the left-hand closure.
///
/// :param: backgroundClosure The closure executed on a background thread.
/// :param: mainClosure The closure executed on the main thread.
func ~> <R> (backgroundClosure: () -> (R), mainClosure: (R) -> ()) {
  dispatch_async(marshalQueue) {
    let result = backgroundClosure()
    dispatch_async(dispatch_get_main_queue()) {
      mainClosure(result)
    }
  }
}

/// Executes the provided closure every 2 minutes. The execution might be
    defered by up to 30 seconds. The closure is executed as long as the
    returned object is referenced by the developer. If you want to stop the
    execution you therefore have to set the variable holding this object to
    nil.
///
/// :param: repeatedClosure The function that should be repeatedly executed.
///
/// :returns: The threading object, that needs to be referenced as long as the
    execution should continue.
postfix func <~ (repeatedClosure: () -> ()) -> MarshalThreadingObject {
  return repeatedClosure<~/50.0
  // TODO: Change back to 120.0!!
}

/// Executes the provided closure every x seconds. The execution might be
    defered by up to 30 seconds. The closure is executed as long as the
    returned object is referenced by the developer. If you want to stop the
    execution you therefore have to set the variable holding this object to
    nil.
///
/// :param: repeatedClosure The function that should be repeatedly executed.
/// :param: intervalBetweenExecutionInSeconds The amount of time between
    executions in seconds.
///
/// :returns: The threading object, that needs to be referenced as long as the
    execution should continue.
func <~/(repeatedClosure: () -> (), intervalBetweenExecutionInSeconds: Double)
    -> MarshalThreadingObject {
  // Amount of time the system can defer the execution of the closure
  let deferedTime = UInt64(30 * Double(NSEC_PER_SEC))
  // Amount of time between executions
  let intervalTime = UInt64(intervalBetweenExecutionInSeconds * Double
```

```
        (NSEC_PER_SEC))
    // Initial delay for first execution
    let startDelay: dispatch_time_t = dispatch_time(DISPATCH_TIME_NOW, 0)

    // Create a new timer on the marshal queue
    let timer: dispatch_source_t = dispatch_source_create
        (DISPATCH_SOURCE_TYPE_TIMER, 0, 0, marshalQueue)
    // Define the behaviour of the timer
    dispatch_source_set_timer(timer, startDelay, intervalTime, deferedTime)

    // Set the event handler (the closure that should be executed everytime the
        timer is firing)
    dispatch_source_set_event_handler(timer, repeatedClosure)

    // Starting the queue
    dispatch_resume(timer)
    return timer
}

/// Executes the right hand closure after waiting the amount of seconds
    defined by the left hand double. The closure is guaranteed to be executed
    on the main queue.
///
/// :param: waitInSeconds The amount of time, the execution gets delayed.
/// :param: mainClosure The closure that is going to be executed after the
    application waited the defined amount of time.
func ~>(waitInSeconds: Double, mainClosure: () -> ()) {
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(waitInSeconds *
        Double(NSEC_PER_SEC))), dispatch_get_main_queue()) {
        mainClosure()
    }
}
```