# Multithreading on iOS

# Why Multithreading?

# Why Multithreading?

- Keep the main thread free for UI rendering
- Use all available CPU cores
- Save energy and reduce battery consumption
- Prioritise tasks to enable the system to optimise the workload

# How to do concurrent work?

# How to do concurrent work?

Available concepts

- NSThread
- NSRunLoop
- NSOperation / NSOperationQueue
- Grand Central Dispatch

# How to do concurrent work?

Available concepts

- ~~NSThread~~
- ~~NSRunLoop~~
- NSOperation / NSOperationQueue
- Grand Central Dispatch

# How to do concurrent work?

## Threads vs. Queues

- Queues are an abstraction on top of threads
- Queues can have parent queues
- Multiple queues can run on a single thread
- A single queue can run on multiple threads
- Multiple queues use a thread pool

# How to do concurrent work?

NSOperation / NSOperationQueue

- Configurable number of parallel executions
- Cancellable
- Dependencies, priorities, QoS classes
- Based on GCD
- Main queue and custom queues

# How to do concurrent work?

Grand Central Dispatch

- Serial and concurrent queues
- QoS classes
- Main queue, default queues and custom queues

# How to do concurrent work?

## NSOperationQueue vs. Grand Central Dispatch

- NSOperationQueue can control maximum parallel execution
- GCD has default queues
- GCD has groups and semaphores

# How to do concurrent work?

## NSOperationQueue vs. Grand Central Dispatch

- NSOperationQueue is more convenient
- GCD has all the fancy shit
- GCD is much more than just getting stuff off the main queue

# Living in threaded environments

# Living in threaded environments

## Models

- Make your models immutable
- Have a single source of truth
- Make your data source thread safe
- Never rely on messages arriving on a certain queue unless explicitly documented
- Explicitly document if you rely on being run on a certain queue

# Living in threaded environments

## API Design

- Design your APIs for asynchronous use
- Ensure completion handler are called on a well defined queue
- Never use mutable data types on public APIs
- Ensure mutating methods are thread safe

# Making use of queues

# Making use of queues

User initiated actions

```
__weak typeof(self) weakSelf = self;
qos_class qosClass = QOS_CLASS_USER_INITIATED;
dispatch_queue_t queue = dispatch_get_global_queue(qosClass, 0);
dispatch_async(queue, ^{
  NSArray<Location *> *locations = // do some heavy work here
  dispatch_async(dispatch_get_main_queue(), ^{
    typeof(weakSelf) self = weakSelf;
    [self.mapView addAnnotations:locations];
    [self.mapView showAnnotations:locations animated:YES];
  });
});
```

# Making use of queues

User initiated actions

```
__weak typeof(self) weakSelf = self;
qos_class qosClass = QOS_CLASS_USER_INITIATED;
dispatch_queue_t queue = dispatch_get_global_queue(qosClass, 0);
dispatch_async(queue, ^{
  NSArray<Location *> *locations = // do some heavy work here
  dispatch_async(dispatch_get_main_queue(), ^{
    typeof(weakSelf) self = weakSelf;
    [self.mapView addAnnotations:locations];
    [self.mapView showAnnotations:locations animated:YES];
  });
});
```

# Making use of queues

User initiated actions

```
__weak typeof(self) weakSelf = self;
qos_class qosClass = QOS_CLASS_USER_INITIATED;
dispatch_queue_t queue = dispatch_get_global_queue(qosClass, 0);
dispatch_async(queue, ^{
  NSArray<Location *> *locations = // do some heavy work here
  dispatch_async(dispatch_get_main_queue(), ^{
    typeof(weakSelf) self = weakSelf;
    [self.mapView addAnnotations:locations];
    [self.mapView showAnnotations:locations animated:YES];
  });
});
```

# Making use of queues

User initiated actions

```
__weak typeof(self) weakSelf = self;
qos_class qosClass = QOS_CLASS_USER_INITIATED;
dispatch_queue_t queue = dispatch_get_global_queue(qosClass, 0);
dispatch_async(queue, ^{
  NSArray<Location *> *locations = // do some heavy work here
  dispatch_async(dispatch_get_main_queue(), ^{
    typeof(weakSelf) self = weakSelf;
    [self.mapView addAnnotations:locations];
    [self.mapView showAnnotations:locations animated:YES];
  });
});
```

# Making use of queues

User initiated actions

```
__weak typeof(self) weakSelf = self;
qos_class qosClass = QOS_CLASS_USER_INITIATED;
dispatch_queue_t queue = dispatch_get_global_queue(qosClass, 0);
dispatch_async(queue, ^{
  NSArray<Location *> *locations = // do some heavy work here
  dispatch_async(dispatch_get_main_queue(), ^{
    typeof(weakSelf) self = weakSelf;
    [self.mapView addAnnotations:locations];
    [self.mapView showAnnotations:locations animated:YES];
  });
});
```

# Making use of queues

User initiated actions

```
__weak typeof(self) weakSelf = self;
qos_class qosClass = QOS_CLASS_USER_INITIATED;
dispatch_queue_t queue = dispatch_get_global_queue(qosClass, 0);
dispatch_async(queue, ^{
  NSArray<Location *> *locations = // do some heavy work here
  dispatch_async(dispatch_get_main_queue(), ^{
    typeof(weakSelf) self = weakSelf;
    [self.mapView addAnnotations:locations];
    [self.mapView showAnnotations:locations animated:YES];
  });
});
```

# Making use of queues

User initiated actions

```
__weak typeof(self) weakSelf = self;
qos_class qosClass = QOS_CLASS_USER_INITIATED;
dispatch_queue_t queue = dispatch_get_global_queue(qosClass, 0);
dispatch_async(queue, ^{
  NSArray<Location *> *locations = // do some heavy work here
  dispatch_async(dispatch_get_main_queue(), ^{
    typeof(weakSelf) self = weakSelf;
    [self.mapView addAnnotations:locations];
    [self.mapView showAnnotations:locations animated:YES];
  });
});
```

# Making use of queues

## Data Source

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>

+ (NSProgress *)getWithCoordinate:(CLLocationCoordinate2D)coordinate
                completionHandler:(void(^)(
                                     NSArray<Event *> *events,
                                     NSError *error)
                                   )completionHandler;

@end
```

# Being thread safe

# Being thread safe

## Immutability

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>
@property (nonatomic, strong, readonly) NSNumber *identifier;
@property (nonatomic, strong, readonly) NSDate *startDate;
@property (nonatomic, strong, readonly) NSString *name;
@end


@interface MutableEvent : Event
@property (nonatomic, strong, readwrite) NSNumber *identifier;
@property (nonatomic, strong, readwrite) NSDate *startDate;
@property (nonatomic, strong, readwrite) NSString *name;
@end
```

# Being thread safe

## Immutability

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>
@property (nonatomic, strong, readonly) NSNumber *identifier;
@property (nonatomic, strong, readonly) NSDate *startDate;
@property (nonatomic, strong, readonly) NSString *name;
@end


@interface MutableEvent : Event
@property (nonatomic, strong, readwrite) NSNumber *identifier;
@property (nonatomic, strong, readwrite) NSDate *startDate;
@property (nonatomic, strong, readwrite) NSString *name;
@end
```

# Being thread safe

## Immutability

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>
@property (nonatomic, strong, readonly) NSNumber *identifier;
@property (nonatomic, strong, readonly) NSDate *startDate;
@property (nonatomic, strong, readonly) NSString *name;
@end


@interface MutableEvent : Event
@property (nonatomic, strong, readwrite) NSNumber *identifier;
@property (nonatomic, strong, readwrite) NSDate *startDate;
@property (nonatomic, strong, readwrite) NSString *name;
@end
```

# Being thread safe

## Immutability

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>
@property (nonatomic, strong, readonly) NSNumber *identifier;
@property (nonatomic, strong, readonly) NSDate *startDate;
@property (nonatomic, strong, readonly) NSString *name;
@end


@interface MutableEvent : Event
@property (nonatomic, strong, readwrite) NSNumber *identifier;
@property (nonatomic, strong, readwrite) NSDate *startDate;
@property (nonatomic, strong, readwrite) NSString *name;
@end
```

# Being thread safe

## Immutability

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>
@property (nonatomic, strong, readonly) NSNumber *identifier;
@property (nonatomic, strong, readonly) NSDate *startDate;
@property (nonatomic, strong, readonly) NSString *name;
@end


@interface MutableEvent : Event
@property (nonatomic, strong, readwrite) NSNumber *identifier;
@property (nonatomic, strong, readwrite) NSDate *startDate;
@property (nonatomic, strong, readwrite) NSString *name;
@end
```

# Being thread safe

## Immutability

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>
@property (nonatomic, strong, readonly) NSNumber *identifier;
@property (nonatomic, strong, readonly) NSDate *startDate;
@property (nonatomic, strong, readonly) NSString *name;
@end


@interface MutableEvent : Event
@property (nonatomic, strong, readwrite) NSNumber *identifier;
@property (nonatomic, strong, readwrite) NSDate *startDate;
@property (nonatomic, strong, readwrite) NSString *name;
@end
```

# Being thread safe

## Immutability

```
@interface Event : NSObject <NSCopying, NSMutableCopying, NSCoding>
@property (nonatomic, strong, readonly) NSNumber *identifier;
@property (nonatomic, strong, readonly) NSDate *startDate;
@property (nonatomic, strong, readonly) NSString *name;
@end


@interface MutableEvent : Event
@property (nonatomic, strong, readwrite) NSNumber *identifier;
@property (nonatomic, strong, readwrite) NSDate *startDate;
@property (nonatomic, strong, readwrite) NSString *name;
@end
```

# Being thread safe

## Thread safety

```
@interface FavoriteEventsController : NSObject

@property (nonatomic, strong, readonly) NSArray<Event *> *events;

- (void)addEvent:(Event *)event;
- (void)removeEvent:(Event *)event;

@end
```

# Being thread safe

## Thread safety

```
@interface FavoriteEventsController : NSObject

@property (nonatomic, strong, readonly) NSArray<Event *> *events;

- (void)addEvent:(Event *)event;
- (void)removeEvent:(Event *)event;

@end
```

# Being thread safe

## Thread safety

```
@interface FavoriteEventsController : NSObject

@property (nonatomic, strong, readonly) NSArray<Event *> *events;

- (void)addEvent:(Event *)event;
- (void)removeEvent:(Event *)event;

@end
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (instancetype)init {
  self = [super init];
  if (self) {
    dispatch_queue_attr_t attributes = DISPATCH_QUEUE_CONCURRENT;
    if (&dispatch_queue_attr_make_with_qos_class != NULL) {
     attr = dispatch_queue_attr_make_with_qos_class(attributes,
                                                    QOS_CLASS_BACKGROUND, -1);
    }
    dispatch_queue_t workerQueue = dispatch_queue_create("fav", attr);
    _workerQueue = workerQueue;

    _events = @[];
  }
  return self;
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (instancetype)init {
  self = [super init];
  if (self) {
    dispatch_queue_attr_t attributes = DISPATCH_QUEUE_CONCURRENT;
    if (&dispatch_queue_attr_make_with_qos_class != NULL) {
     attr = dispatch_queue_attr_make_with_qos_class(attributes,
                                                    QOS_CLASS_BACKGROUND, -1);
    }
    dispatch_queue_t workerQueue = dispatch_queue_create("fav", attr);
    _workerQueue = workerQueue;

    _events = @[];
  }
  return self;
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (instancetype)init {
  self = [super init];
  if (self) {
    dispatch_queue_attr_t attributes = DISPATCH_QUEUE_CONCURRENT;
    if (&dispatch_queue_attr_make_with_qos_class != NULL) {
     attr = dispatch_queue_attr_make_with_qos_class(attributes,
                                                    QOS_CLASS_BACKGROUND, -1);
    }
    dispatch_queue_t workerQueue = dispatch_queue_create("fav", attr);
    _workerQueue = workerQueue;

    _events = @[];
  }
  return self;
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (instancetype)init {
  self = [super init];
  if (self) {
    dispatch_queue_attr_t attributes = DISPATCH_QUEUE_CONCURRENT;
    if (&dispatch_queue_attr_make_with_qos_class != NULL) {
     attr = dispatch_queue_attr_make_with_qos_class(attributes,
                                                    QOS_CLASS_BACKGROUND, -1);
    }
    dispatch_queue_t workerQueue = dispatch_queue_create("fav", attr);
    _workerQueue = workerQueue;

    _events = @[];
  }
  return self;
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (instancetype)init {
  self = [super init];
  if (self) {
    dispatch_queue_attr_t attributes = DISPATCH_QUEUE_CONCURRENT;
    if (&dispatch_queue_attr_make_with_qos_class != NULL) {
     attr = dispatch_queue_attr_make_with_qos_class(attributes,
                                                    QOS_CLASS_BACKGROUND, -1);
    }
    dispatch_queue_t workerQueue = dispatch_queue_create("fav", attr);
    _workerQueue = workerQueue;

    _events = @[];
  }
  return self;
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (NSArray<Event *> *)events {
    __block NSArray *events;
    dispatch_sync(self.workerQueue, ^{
        events = _events;
    });
    return events;
}

- (void)setEvents:(NSArray<Event *> *)events {
    dispatch_barrier_async(self.workerQueue, ^{
        _events = events;
    });
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (NSArray<Event *> *)events {
    __block NSArray *events;
    dispatch_sync(self.workerQueue, ^{
        events = _events;
    });
    return events;
}

- (void)setEvents:(NSArray<Event *> *)events {
    dispatch_barrier_async(self.workerQueue, ^{
        _events = events;
    });
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (NSArray<Event *> *)events {
    __block NSArray *events;
    dispatch_sync(self.workerQueue, ^{
        events = _events;
    });
    return events;
}

- (void)setEvents:(NSArray<Event *> *)events {
    dispatch_barrier_async(self.workerQueue, ^{
        _events = events;
    });
}
```

# Being thread safe

## Thread safety - FavoriteEventsController

```
- (void)addEvent:(Event *)event {
  dispatch_barrier_async(self.workerQueue, ^{
    _events = [_events arrayByAddingObject:event];
  });
}

- (void)removeEvent:(Event *)event {
  dispatch_barrier_async(self.workerQueue, ^{
    NSPredicate *p = [NSPredicate
                      predicateWithFormat:@"NOT (SELF == %@)", event];
    _events = [_events filteredArrayUsingPredicate:p];
  });
}
```

# Convenience

# Convenience

## Dispatch groups - Synchronously

```
dispatch_group_t fetchGroup = dispatch_group_create();
for (int i = 0; i < 10; i++) {
  dispatch_group_enter(fetchGroup);
  [Fetcher doWithCompletionHandler:^(BOOL success) {
    // Handle completion
    dispatch_group_leave(fetchGroup);
  }];
}
dispatch_group_wait(fetchGroup, DISPATCH_TIME_FOREVER);
// Do stuff after all completions finished
```

# Convenience

## Dispatch groups - Asynchronously

```
dispatch_group_t fetchGroup = dispatch_group_create();
for (int i = 0; i < 10; i++) {
    dispatch_group_enter(fetchGroup);
    [Fetcher doWithCompletionHandler:^(BOOL success) {
        // Handle completion
        dispatch_group_leave(fetchGroup);
    }];
}
dispatch_group_notify(fetchGroup, dispatch_get_main_queue(), ^{
    // Do stuff after all completions finished
});
```

# Thank you

Michael Ochs
@_mochs
http://ios-coding.com