Mobile Application Development

Lecture 7
Blocks, Concurrency, Networking

Lecture Summary

- Blocks
- Concurrency and multi-threading
- Grand Central Dispatch (GCD)
- Networking
- UIImage & UIImageView

Closures

- A **closure** is a function (or a reference to function) together with a referencing environment
- A referencing environment is an area where non-local variables are stored and can be accessed
- A function is different from a closure because it cannot access non-local variables (also called *upvalues* of the function)

Closures

- A **closure** is a function (or a reference to function) together with a referencing environment
- A referencing environment is an area where non-local variables are stored and can be accessed
- A function is different from a closure because it cannot access non-local variables (also called *upvalues* of the function)
- Examples of closures:
  - in Java: local classes, anonymous classes

```
final int rounds = 100;
Runnable runnable = new Runnable() {
    public void run() {
        for (int i = 0; i < rounds; i++) {
            // ...
        }
    }
};
new Thread(runnable).start();
```

Blocks

- **Blocks** are a language-level extension of C, Objective-C, and C++ introduced to support closures:
  - blocks can access non-local variables from the enclosing scope
- Blocks are segments of code that can be passed around to methods or functions as if they were values
- Blocks are Objective-C objects, thus they can be added to `NSArray` or `NSDictionary` instances

https://developer.apple.com/library/ios/documentation/cocoa/conceptual/Blocks/Articles/00_Introduction.html#//apple_ref/doc/uid/TP40007502

Defining blocks

- A block is defined with a caret (^) and its start and end are marked by curly braces:

```
^{
    // block body ...
    NSLog(@"Within block...");
}
```

- Blocks can be assigned to variables:

```
block = ^{
    NSLog(@"Within block...");
};
```

- The above block does not return any value and takes no argument
- A block can be executed, as if it were a C function:

```
block();
```

Defining blocks

- Blocks can return a value and take arguments, just like functions
- General syntax of a **block signature**:

```
return_type(^block_name)(arguments);
```

- Some examples of block definition:

Block type definition	Input arguments	Return type
`void(^BlockA)(void);`	-	-
`int(^BlockB)(int);`	one `int`	one `int`
`int(^BlockC)(int,int);`	two `int`s	one `int`
`void(^BlockD)(double);`	one `double`	-

Defining blocks

- `typedef` can be used to define blocks with the same signature to simplify code

```
typedef int(^ReturnIntBlock)(int);

ReturnIntBlock square = ^(int val){

    return val * val;

};
int a = square(10);

// ...

ReturnIntBlock cube = ^(int val){

    return val * val * val;

};
int b = cube(10);
```

Working with blocks

```
int(^max)(int, int) = ^(int a, int b){

    return (a > b ? a : b);

};

int maximum = max(first,second);
```

Accessing variables within blocks

- Blocks can access values from the enclosing scope (blocks are closures)
- A block literal declared from within a method can access the local variables accessible in the local scope of the method

→

```
- (void)aMethod{
    // ...
    int var = 10;
    void(^example)(void) = ^{
        NSLog(@"Accessing variable %d",var);
    };
    // ...
}
```

- Only the value of non-local variables is captured, so inside the block it is unaffected
- The value of non-local variables cannot be changed (read-only access)

→

```
- (void)aMethod{
    // ...
    int var = 10;
    void(^example)(void) = ^{
        NSLog(@"Accessing variable %d",var);
    };
    // ...
    var = 100;
    example(); // var is still 10
}
```

Accessing variables within blocks

- If a non-local variable should be changed inside a block, it must be properly marked when it is declared with the `__block` directive
- The `__block` directive also marks the variable for shared storage between the enclosing scope and the block, so its value might change as it is not passed in by value only

```
- (void)aMethod{
    // ...
    __block int var = 10;
    void(^example)(void) = ^{
        NSLog(@"Accessing variable %d",++var);
    };
    // ...
    var = 100;
    example(); // prints 101
}
```

Blocks as arguments

- Blocks are objects: they can be passed in as arguments to methods and functions
- Similar to passing `void*`, except that blocks are much more than function pointers
- Typically, blocks are used to create callback methods or execute them in other parts of the code that can be parametrized

Blocks as arguments

```
- (void)downloadFromUrl:(NSURL *)url completion:(void(^)(NSData*))callback;
```

- The last argument of this method is a block, which takes a **NSData*** argument and returns nothing, that will be executed as callback
- It is equivalent to the following declaration:

```
typedef void(^DownloadBlock)(NSData*);

- (void)downloadFromUrl:(NSURL *)url completion:(DownloadBlock)callback;
```

Blocks as arguments

- The method's implementation might look like

```
- (void)downloadFromUrl:(NSURL *)url completion:(void(^)(NSData*))callback{
    // download from the URL
    NSData *data = ...;
    // ...
    callback(data);
}
```

Blocks as arguments

- To invoke the method, we must pass in a **`void(^)(NSData*)`** block

```
void(^DownloadCompleteHandler)(NSData *) = ^(NSData *data){

    NSLog(@"Download complete [%d bytes]!", [data length]);

};

// ...

[self downloadFromUrl:url completion:DownloadCompleteHandler];
```

Concurrency

- It is very important to be able to execute many tasks in parallel
- The main thread is the application's thread responsible for handling user-generated events and managing the UI
- A long-running task (or a task that might take an undefined amount of time) should never be performed in the main thread
- Background threads are needed in order to keep the main thread ready for handling event, and therefore to keep the application responsive
- Multiple threads perform different tasks simultaneously
- Threads must be coordinated to avoid that the program reaches an inconsistent state

https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html

Concurrency

- How does iOS handle the execution of many tasks at the same time?
- iOS provides mechanisms to work with threads (UNIX-based)
- Threads are a lightweight low-level feature that can be used to provide concurrency in a program
- However, using threads introduces complexity to programs because of several issues, such as:
  - synchronization
  - scalability
  - shared data structures
- Threads introduce uncertainty to code and overhead at runtime for context switch
- Alternative methods have been introduced in order to progressively eliminate the need to rely on threads

Threads in iOS

- iOS provides several methods for creating and managing threads
- POSIX threads can be used (C interface)
- The Cocoa framework provides an Objective-C class for threads, `NSThread`

```
[NSThread detachNewThreadSelector:@selector(threadMainMethod:) toTarget:self withObject:nil];
```

- `NSObject` also allows the spawning of threads

```
[obj performSelectorInBackground:@selector(aMethod) withObject:nil];
```

- Using threads is discouraged, iOS provides other technologies that can be used to effectively tackle concurrency in programs

Alternative to threads

- Instead of relying on threads, iOS adopts an **asynchronous design approach**:

1. acquire a background thread
2. start the task on that thread
3. send a notification to the caller when the task is done (callback function)

- An asynchronous function does some work behind the scenes to start a task running but returns before that task might actually be complete
- iOS provides technologies to perform asynchronous tasks without having to manage the threads directly

Grand Central Dispatch

- **Grand Dispatch Central** (GCD) is one of the technologies that iOS provides to start tasks asynchronously
- GCD is a C library which moves the thread management code down to the system level
- GCD takes care of creating the needed threads and of scheduling tasks to run on those threads
- GCD is based on **dispatch queues**, which are a C-based mechanism for executing custom tasks

Dispatch queues

- A dispatch queue executes tasks, either synchronously or asynchronously, always in a FIFO order (a dispatch queue always dequeues and starts tasks in the same order in which they were added to the queue)
- Serial dispatch queues run only one task at a time, each time waiting that the current task completes before executing the next one
- Concurrent dispatch queues can execute many tasks without waiting for the completion of other tasks
- With GCD, tasks to be executed are first defined and then added to an appropriate dispatch queue
- Dispatch queues provide a straightforward and simple programming interface
- The tasks submitted to a dispatch queue must be encapsulated inside either a function or a block object

Operation queues

- **Operation queues** are an Objective-C equivalent to concurrent dispatch queues provided by Cocoa (implemented by the class `NSOperationQueue`)

- Operation queues do not necessarily take a FIFO ordering, as they take into account also the possible dependencies among operations

- Operations added to an operation queue are instances of the `NSOperation` class

- Operations can be configured to define dependencies, in order to affect the order of their execution

Working with queues

- Getting the main queue

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();
```

- Creating a queue

```
dispatch_queue_t queue = dispatch_queue_create("queue_name", NULL);
```

- Executing a task defined in a block asynchronously on a queue

```
dispatch_async(queue, block);
```

- Executing a task on the main queue

```
dispatch_async(dispatch_get_main_queue(), block);
```

- If not using ARC, the queue must be released

```
dispatch_release(queue);
```

Working with queues

```
dispatch_queue_t queue = dispatch_queue_create("queue_name", NULL);

dispatch_async(queue, ^{

    // long running task...

    dispatch_async(dispatch_get_main_queue(), ^{

        // update UI in main thread...

    });

});
```

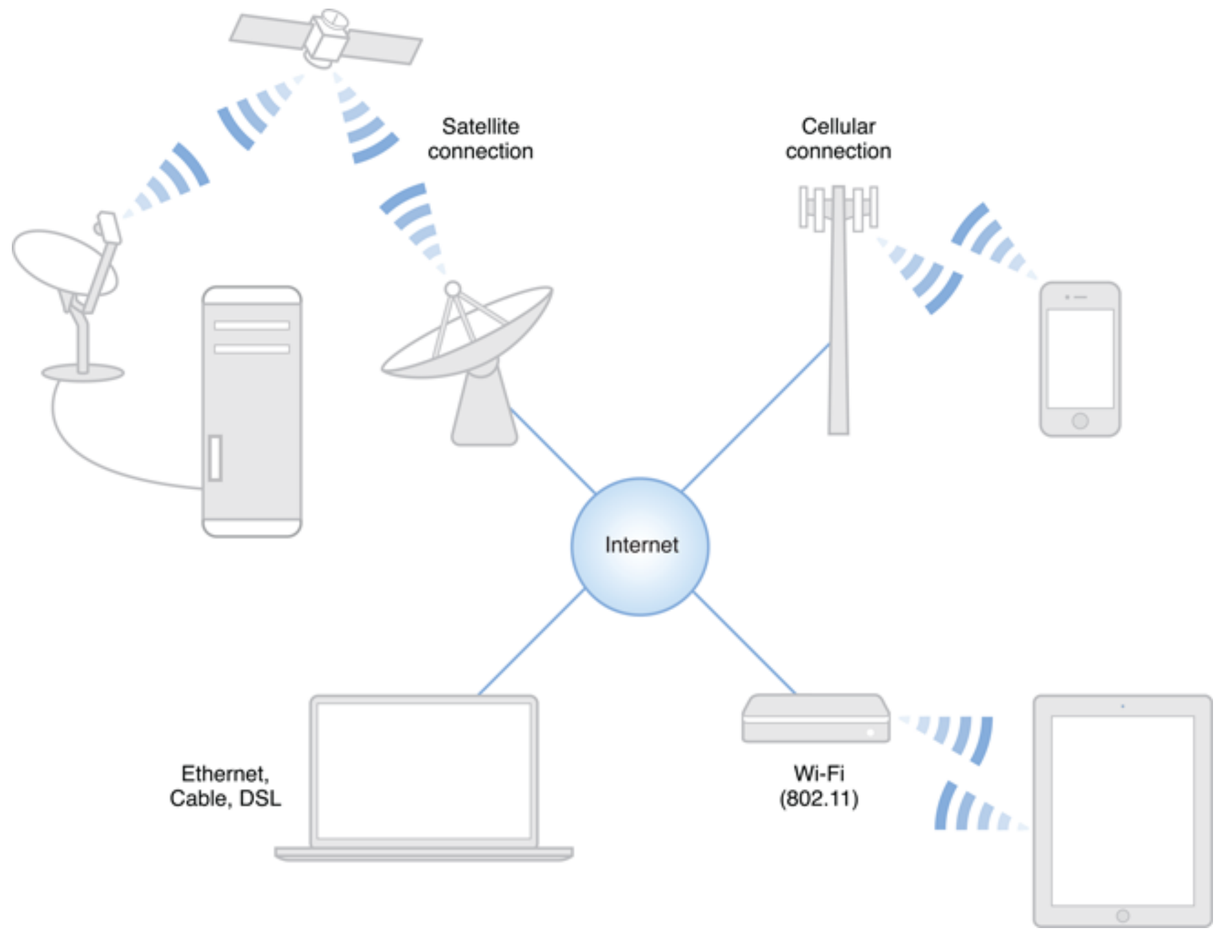
Networking

- iOS provides APIs for networking at many levels, which allow developers to:
  - perform HTTP/HTTPS requests, such as GET and POST requests
  - establish a connection to a remote host, with or without encryption or authentication
  - listen for incoming connections
  - send and receive data with connectionless protocols
  - publish, browse, and resolve network services with Bonjour

https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010220-CH12-BBCFIHFH

Networking

- Many applications rely on data and services provided by remote hosts, thus networking plays a central role in today's app development
- As a general rule, networking should be dynamic and asynchronous... and NOT on the main thread!
- Connectivity status might change abruptly, so apps should be designed to be robust to maintain usability and user experience
- It is a bad design to make assumptions on the speed of the connection (even though the type of connection may be desumed)



NSURL

- **NSURL** objects are used to manipulate URLs and the resources they reference
- **NSURL** objects are also used to refer to a file
- Typically, an instance of **NSURL** can be created with either of the following methods:

```
NSURL *aUrl = [NSURL URLWithString:@"http://mobdev.ce.unipr.it"];

NSURL *bUrl = [[NSURL alloc] initWithString:@"http://mobdev.ce.unipr.it"];

NSURL *cUrl = [NSURL URLWithString:@"/path" relativeToURL:aURL];
```

Loading URLs to NSString and NSData

- **`NSString`** and **`NSData`** instances can be created with information coming from a remote URL:

```
NSURL *url = ...;
NSError *error;

NSString *str = [NSString stringWithContentsOfURL:url encoding:NSUTF8StringEncoding error:&error];

NSData *data = [NSData dataWithContentsOfURL:url options:NSDataReadingUncached error:&error];
```

- **`stringWithContentsOfURL:encoding:error:`** returns a string created by reading data from a given URL interpreted using a given encoding
- **`dataWithContentsOfURL:options:error:`** creates and returns a data object containing the data from the location specified by the given URL
- These methods are synchronous, so it is better to execute them inside a background dispatch queue to keep an asynchronous design

NSURLRequest

- **`NSURLRequest`** objects represent a URL load request independent of protocol and URL scheme
- The mutable subclass of **`NSURLRequest`** is **`NSMutableURLRequest`**
- An **`NSURLRequest`** is constructed with a given **`NSURL`** argument:

```
NSURLRequest *req = [NSURLRequest requestWithURL:url];
NSURLRequest *req2 = [[NSURLRequest alloc] initWithURL:url];
```

- **`NSMutableURLRequest`** instances can be configured to set protocol independent properties and HTTP-specific properties:
  - URL, timeout
  - HTTP method, HTTP body, HTTP headers

NSURLConnection

- **`NSURLConnection`** objects provide support to perform the loading of a URL request
- An **`NSURLConnection`** can load a URL request either synchronously or asynchronously
- Synchronous URL loading:

```
NSError *error;
NSURLResponse *response = nil;
NSData *data = [NSURLConnection sendSynchronousRequest:req
                                     returningResponse:&response
                                                 error:&error];
```

- Asynchronous URL loading:

```
[NSURLConnection
    sendAsynchronousRequest:req
                      queue:[NSOperationQueue currentQueue]
          completionHandler:^(NSURLResponse *response, NSData *data, NSError *connectionError) {
                              // ...
                          }
];
```

NSURLConnectionDelegate

- The **`NSURLConnectionDelegate`** protocol provides methods for receive informational callbacks about the asynchronous load of a URL request
- Delegate methods are called on the thread that started the asynchronous load operation for the associated **`NSURLConnection`** object
- **`NSURLConnection`** with specified delegate:

```
NSURLConnection *conn = [[NSURLConnection alloc] initWithRequest:req delegate:self];
```

- Starting and stopping the loading of a URL with **`NSURLConnection`**:

```
[conn start];

[conn cancel];
```

NSURLConnectionDelegate

- The **`NSURLConnectionDelegate`** and **`NSURLConnectionDataDelegate`** protocols define methods that should be implemented by the delegate for an instance of **`NSURLConnection`**
- In order to keep track of the progress of the URL loading the protocol methods should be implemented

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data{
    NSLog(@"received %d bytes", [data length]);
}

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response{
    NSLog(@"received response");
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection{
    NSLog(@"loading finished");
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error{
    NSLog(@"loading failed");
}
```

NSURLSession

- The `NSURLSession` class and related classes provide an API for downloading content via HTTP
- This API provides delegate methods for:
  - supporting authentication
  - giving the app the ability to perform background downloads when the app is not running or while your app is suspended
- The `NSURLSession` API is highly asynchronous: a completion handler block (that returns data to the app when a transfer finishes successfully or with an error) must be provided
- The `NSURLSession` API provides status and progress properties, in addition to delivering this information to delegates
- It supports canceling, restarting (resuming), and suspending tasks, and it provides the ability to resume suspended, canceled, or failed downloads where they left off
- The behavior of a `NSURLSession` depends on its session type and its session task

Types of sessions

- There are three types of sessions, as determined by the type of configuration object used to create the session:

1. **Default sessions** use a persistent disk-based cache and store credentials in the user's keychain
2. **Ephemeral sessions** do not store any data to disk; all caches, credential stores, and so on are kept in RAM and tied to the session; when your app invalidates the session, they are purged automatically
3. **Background sessions** are similar to default sessions, except that a separate process handles all data transfers

Types of tasks

- There are three types of tasks supported by a session:

1. **Data tasks** send and receive data using NSData objects
   - data tasks are intended for short, often interactive requests from your app to a server
   - data tasks can return data to your app one piece at a time after each piece of data is received, or all at once through a completion handler
   - data tasks do not store the data to a file, so they are not supported in background sessions
2. **Download tasks** retrieve data in the form of a file, and support background downloads while the app is not running
3. **Upload tasks** send data (usually in the form of a file), and support background uploads while the app is not running

Working with NSURLSession

- The `NSURLSession` instance can be created to execute its completion handler block in the main queue or in another queue

Working with NSURLSession

- The **`NSURLSession`** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **in the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf
                                                      delegate:nil
                                                 delegateQueue:[NSOperationQueue mainQueue]];

NSURLSessionDownloadTask *task;

task = [session
        downloadTaskWithRequest:request
              completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

                  /* this block is executed on the main queue */
                  /* UI-related code ok, but avoid long-running operations */
              }
];

[task resume];
```

Working with NSURLSession

- The **`NSURLSession`** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **in the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf
                                                      delegate:nil
                                                 delegateQueue:[NSOperationQueue mainQueue]];

NSURLSessionDownloadTask *task;

task = [session
        downloadTaskWithRequest:request
              completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

                  /* this block is executed on the main queue */
                  /* UI-related code ok, but avoid long-running operations */
              }
];

[task resume];
```

setting the main queue as the delegate queue to execute the completion handler

Working with NSURLSession

- The **`NSURLSession`** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **in the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf
                                                      delegate:nil
                                                 delegateQueue:[NSOperationQueue mainQueue]];

NSURLSessionDownloadTask *task;

task = [session
        downloadTaskWithRequest:request
              completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

                  /* this block is executed on the main queue */
                  /* UI-related code ok, but avoid long-running operations */
              }
];

[task resume];
```

Working with NSURLSession

- The **`NSURLSession`** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **off the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf];

NSURLSessionDownloadTask *task;

task = [session
        downloadTaskWithRequest:request
              completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

                  /* non-UI-related code here (not in the main thread) */

                  dispatch_async(dispatch_get_main_queue(), ^{
                     /* UI-related code (in the main thread) */
                  });

              }
];
```

Working with NSURLSession

- The **`NSURLSession`** instance can be created to execute its completion handler block in the main queue or in another queue

- Executing the completion handler **off the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf];

NSURLSessionDownloadTask *task;
```

no delegate queue

```
task = [session
        downloadTaskWithRequest:request
              completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

                  /* non-UI-related code here (not in the main thread) */

                  dispatch_async(dispatch_get_main_queue(), ^{
                     /* UI-related code (in the main thread) */
                  });

              }
];
```

Working with NSURLSession

- The **`NSURLSession`** instance can be created to execute its completion handler block in the main queue or in another queue
- Executing the completion handler **off the main queue**:

```
NSURLSessionConfiguration *conf = [NSURLSessionConfiguration defaultSessionConfiguration];

NSURLSession *session = [NSURLSession sessionWithConfiguration:conf];

NSURLSessionDownloadTask *task;

task = [session
        downloadTaskWithRequest:request
              completionHandler:^(NSURL *localfile, NSURLResponse *response, NSError *error) {

                  /* non-UI-related code here (not in the main thread) */

                  dispatch_async(dispatch_get_main_queue(), ^{
                     /* UI-related code (in the main thread) */
                  });

              }
];
```

Sockets and streams

- `NSStream` is an abstract class for objects representing streams
- `NSStream` objects provide a common way to read and write data to and from:
  - memory
  - files
  - network (using sockets)
- `NSInputStream` are used to read bytes from a stream, while `NSOutputStream` are used to write bytes to a stream
- With streams it is possible to write networking code that works at the TCP level, instead of application level (such as HTTP/HTTPS URL loading)

Network Activity Indicator

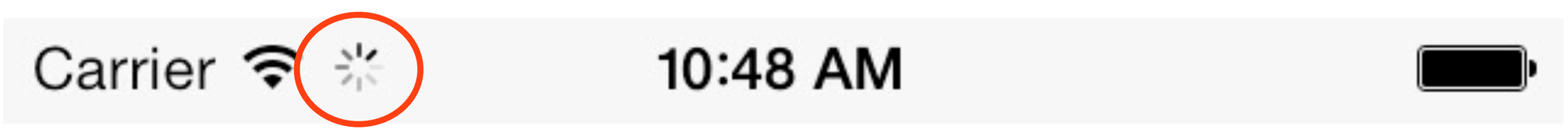
- When performing network-related tasks (typically, for more than a couple of seconds), such as uploading or downloading contents, feedback should be provided to the user
- A **network activity indicator** appears in the status bar and shows that network activity is occurring

- Use the `UIApplication` method `networkActivityIndicatorVisible` to control the indicator's visibility:

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
```

Network Activity Indicator

- When performing network-related tasks (typically, for more than a couple of seconds), such as uploading or downloading contents, feedback should be provided to the user
- A **network activity indicator** appears in the status bar and shows that network activity is occurring



- Use the `UIApplication` method `networkActivityIndicatorVisible` to control the indicator's visibility:

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
```

UIActivityIndicatorView

- An activity indicator is a spinning wheel that indicates a task is being executed
- An activity indicator should be displayed to provide feedback to the user that your app is not stalled or frozen, typically when performing long-running tasks (CPU-intensive or network tasks)
- The activity indicator view is provided by the `UIActivityIndicatorView` class
- A `UIActivityIndicatorView` can be started and stopped using the `startAnimating` and `stopAnimating` methods
- The `hidesWhenStopped` property can be used to configure whether the receiver is hidden when the animation is stopped
- A `UIActivityIndicatorView`'s appearance can be customized with the `activityIndicatorViewStyle` or `color` properties

UIImage

- The **`UIImage`** class is used to display image data
- Images can be created from files (typically PNG and JPEG files) and (network) data

```
UIImage *imgA = [UIImage imageNamed:@"image.png"];

NSData *data = ...;
UIImage *imgB = [UIImage imageWithData:data];

UIImage *imgC = [UIImage imageWithContentsOfFile:@"/path/to/file"];
```

- Image objects are immutable
- Supported image file formats: TIFF, JPEG, PNG, GIF
- Image properties: `imageOrientation`, `scale`, `size`

UIImageView

- A `UIImageView` displays an image or an animated sequence of images
- To display a single image, the image view must be configured with the appropriate `UIImage`
- For multiple images, also the animations among images must be configured
- A UIImageView object can be dragged into a view using the object palette
- The image to be displayed can be configured in storyboard

- The image property can be used to set programmatically the `image` property

UIImageView

- Image views can perform expensive operations that can affect the performance of the application, such as scaling the image and setting the transparency to mix the image with lower layers

- Possible countermeasures that can have a positive impact on performance are:
  - provide pre-scaled images where possible (e.g. thumbnails for previews)
  - limit image size (e.g. pre-scaling or tiling large images)
  - disable alpha blending except where needed

Mobile Application Development

Lecture 7
Blocks, Concurrency, Networking