Roadmap
Jump to Submission · Next Checkpoir

Welcome to Bloc

Congratulations on starting your journey to learn iOS app development. You will be amazed with how much you can learn in a remote apprenticeship. To get the most out of Bloc, it's important to dedicate enough time and immerse yourself in the apprenticeship.

How Much Time?

Bloc Program	Time Expectation
12-Week	40 hours / week
18-Week	25 hours / week
36-Week	15 hours / week

Ideally, spread your time out so that you are either writing or reading code every day of the week. Commit to this schedule, and you'll be able to build and contribute to real applications upon graduation.

Apprenticeship Structure

Bloc's remote iOS apprenticeship is made up of the Roadmap, appointments, code reviews and messages. Each of these mediums are described below.

Roadmap

The Roadmap represents Bloc's recommended curriculum. By working with your mentor and dedicating the requisite time, you will progress through the Roadmap and learn the material detailed in the checkpoints.

The world of iOS development moves fast, so it's important to note that the Roadmap is always being improved. If we deploy a major update during your apprenticeship, you will be notified.

Everyone learns differently, so the Roadmap is not meant to be 100% rigid. Mentors may take you off-roading occasionally, as they see fit. Moreso than the Roadmap, your mentor is your ultimate guide in Bloc.

Appointments

Appointments are your one-on-one meetings with mentors. Each appointment is usually 30 minutes in length, though the actual duration will depend on the topic and your mentor's advice.

Bloc Program	Required Weekly Appointments

18-Week	2
36-Week	1

The primary goal for each appointment is to **pair program** with your mentor. Pair programming is a cornerstone of the remote apprenticeship, and is the best process for learning from a mentor.

The software requirements for appointments are:

- **Screenhero** - This is a screen sharing application that allows you to pair program with your mentor.
- **Google Hangouts** or **Skype** for video chat.
- **Xcode** - the software used to make iOS apps (we'll cover installation next).

Code Reviews

Like pair programming, **code reviews** are a cornerstone of the remote apprenticeship. Code reviews are asynchronous conversations regarding code-specific issues, and are best asked in the form of **pull requests**. Pull requests may not be necessary in every situation, but they are great way to pose questions to your mentor. In a subsequent checkpoint, you'll set up a GitHub account where you can create share repositories to collaborate with your mentor. Your mentor will also review the pull request process with you.

Bloc Overflow

Between appointments with your mentor, you will encounter problems. Good developers know how to ask good questions, and good questions make it much easier to get help. Aspiring and professional developers alike use **Stack Overflow** extensively to get and give help. More importantly, the Stack Overflow community values pragmatic questions and answers, so merely participating in Stack Overflow is time well-spent.

We believe that Stack Overflow is an important tool to start using as you learn to code, but we also believe that your experience using Stack Overflow can be significantly enhanced with mentorship. We built **Bloc Overflow** to combine the utility of Stack Overflow with the guidance of a mentor. As you encounter problems between appointments, work with your mentor to submit thoughtful questions to Stack Overflow. Once you submit a question and link it to Bloc (using our "Blocmarklet"), your question will be viewed by Bloc mentors and your course director, in addition to the rest of the Stack Overflow community. **Simply put, Bloc Overflow is the best way to get help between mentor appointments.**

Before you submit your first question, read our **guide for getting help on Stack Overflow** and ask your mentor if you have concerns.

Messages

Messages allow you to communicate with your mentor within the Bloc app. Use these whenever you get stuck in between meetings. Your mentor will generally respond promptly, and they can provide useful resources and explanations outside of the meeting format. Use **Markdown syntax** when you send messages so you can properly format code blocks.

Community

We created a Facebook group for students to collaborate on projects, discuss opportunities and challenge each other to develop and maintai great habits. **The Bloc Hacker Club** is a place you'll want to be during Bloc and after you graduate. Meet your fellow students, join a team, work on projects, and become a better developer with your peers.

Hacker Club teams are a great way to progress through your course. Each Hacker Club team consists of four students. Teammates work together as they progress through their course. Teammates should build great habits together and hold each other accountable for their wor and progress. Many teams continue to work together *after* they graduate as well. Teams with graduates can build side projects, review open

Basecamp to organize teams, and your Program Coordinator will place you on one. Once you are placed on a team, introduce yourself and find some common times to meet with your teammates. Teams are separate and distinct from your mentor: joining - or not joining - a team will not affect your relationship with your mentor.

To have a successful apprenticeship, your expectations must be aligned with your mentor. It's important to understand what's expected of you, and what you should expect of your mentor.

Mentor expectations

- Your mentor will lead your apprenticeship and may adjust the course to accommodate your learning curve
- Your mentor will be on time for meetings and meetings will last approximately 30 minutes
- Your mentor will respond promptly to issues that are blocking your progress
- Your mentor will review and complete submitted checkpoints and assignments in a timely fashion
- Your mentor will conduct regular code reviews

Apprentice expectations

- You will trust the knowledge and experience of your mentor and accept the assignments and projects provided by your mentor
- You will be self-driven, motivated to learn, and able to commit at least 15/25/40 hours a week to working on the course based on your track
- You will attend meetings on time and be available for at least 30 minutes to work with your mentor
- You will focus on how well you understand the information and not just whether or not your answer is "correct"
- You will keep your mentor informed when you get stuck, letting them know what the problem is and how you've sought to resolve it

You are encouraged to establish additional expectations with your mentor. If you are unsatisfied with your mentor, provide them with feedback. You should additionally provide your Program Coordinator with feedback on your Mentor, the curriculum, and the course experience in general. Discuss your goals and expectations during your first meeting.

assignment completed

Bloc Veterans Program

Employer Sponsored

FAQ

Blog

Engineering Blog

Refer-a-Friend

Privacy Policy

Terms of Service

MADE BY BLOC

Tech Talks & Resources

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Install and Get to Know Xcode

Xcode is where you'll spend the majority of your time developing apps.

"Xcode" includes:

- The Xcode **IDE**,
- LLVM compiler, which turns the code you write into an app,
- Instruments, for measuring and tuning your app's performance,
- iOS Simulator, for running your app on your computer, and
- the latest OS X and iOS **SDKs**, which make it simpler to write native apps for these platforms

Download it if you haven't already.

iOS Developer Account

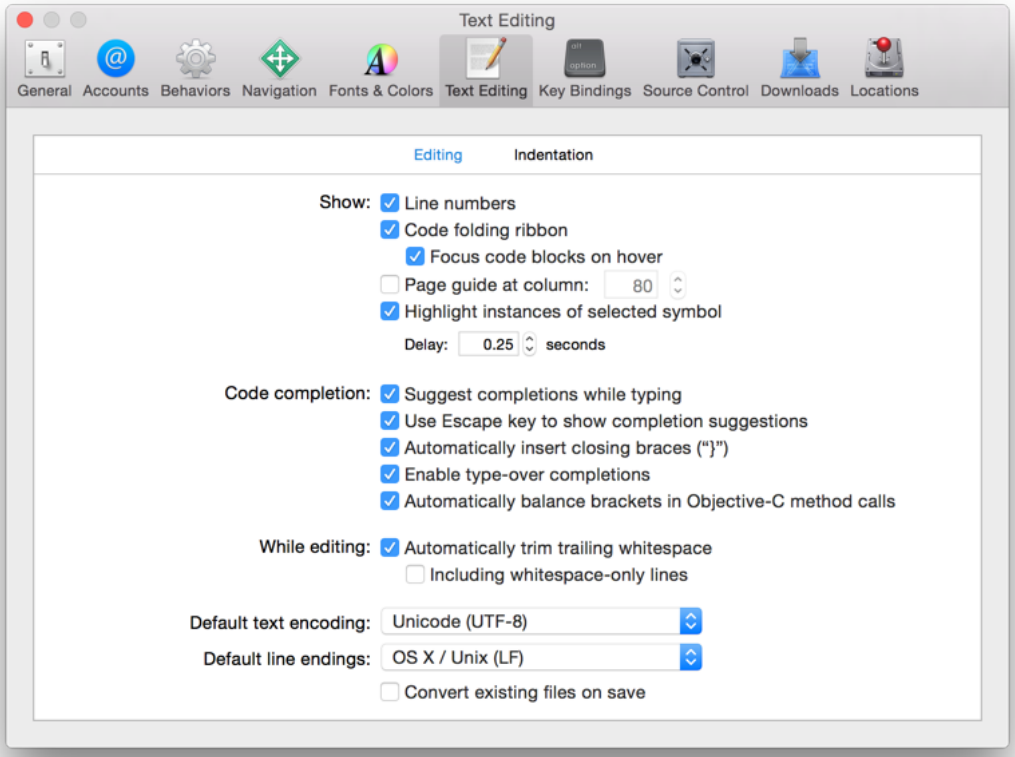
While Xcode is downloading, consider signing up for Apple's **iOS Developer Program**, which:

- allows you to test and debug your code directly on iOS devices
- gives you access to unreleased betas of iOS and Xcode
- grants you access to the Apple Developer Forums, where you can ask questions of Apple engineers and other knowledgable people
- lets you distribute your apps on the App Store

It costs $99/year. Bloc strongly recommends joining the iOS Developer Program, but we don't require it. If you don't want to sign up now, you can still run your apps in Xcode's iOS Simulator.

Configuring Xcode

Once Xcode is downloaded, launch it. Before we get started, let's change a few settings. Go to **Xcode** > **Preferences...**:

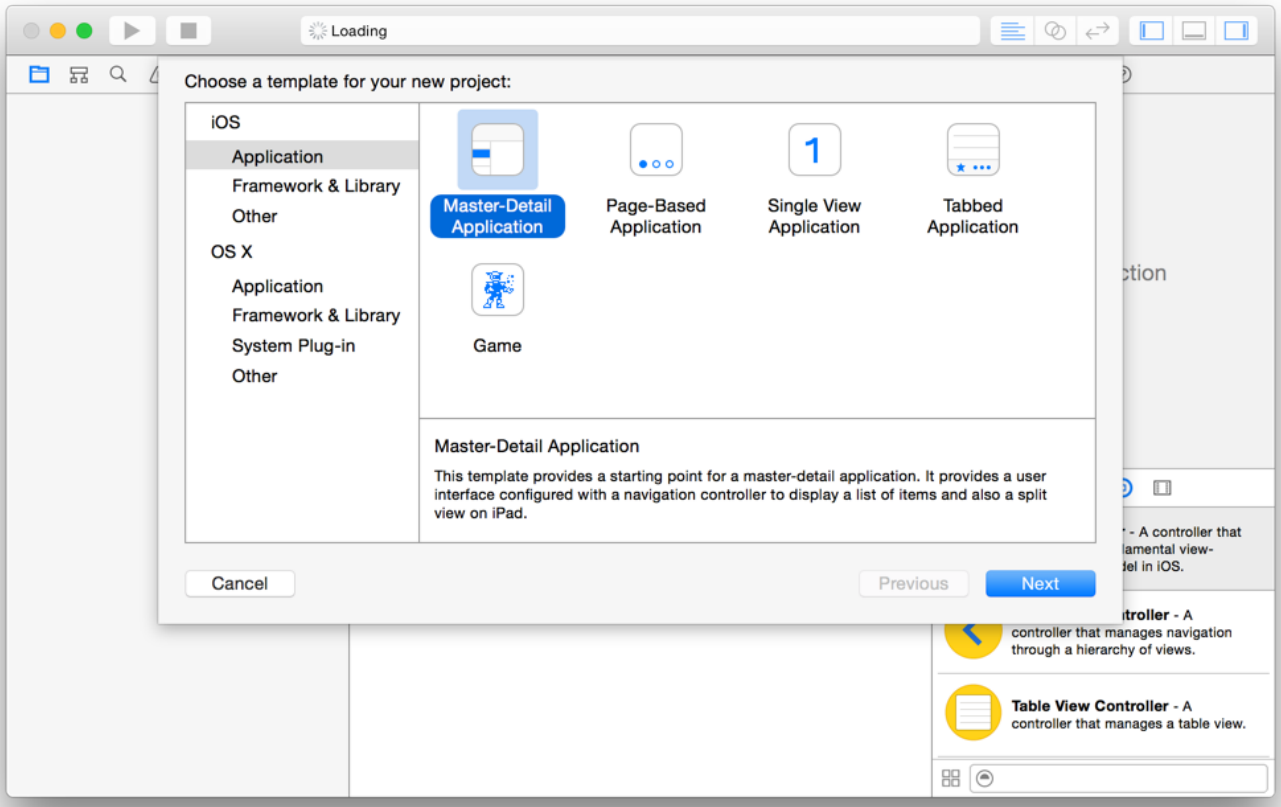


- If you signed up for a developer account, go to the **Accounts** pane and press **+** to add your account.
- In the **Text Editing** Pane, check "Show Line Numbers." This will make it easier for you and your mentor to discuss specific lines of code.
- Under **Downloads**, you may wish to download the newest iOS doc set. This will allow you to read and browse documentation without ar Internet connection. Here, you can also download old simulators; this is helpful if you want to test your app on old versions of iOS.

Now, close Preferences.

Taking Xcode for a Spin

Select **Create a New Xcode Project** from the **Welcome to Xcode** menu. (If you don't see it, press **File** > **New** > **Project...**)

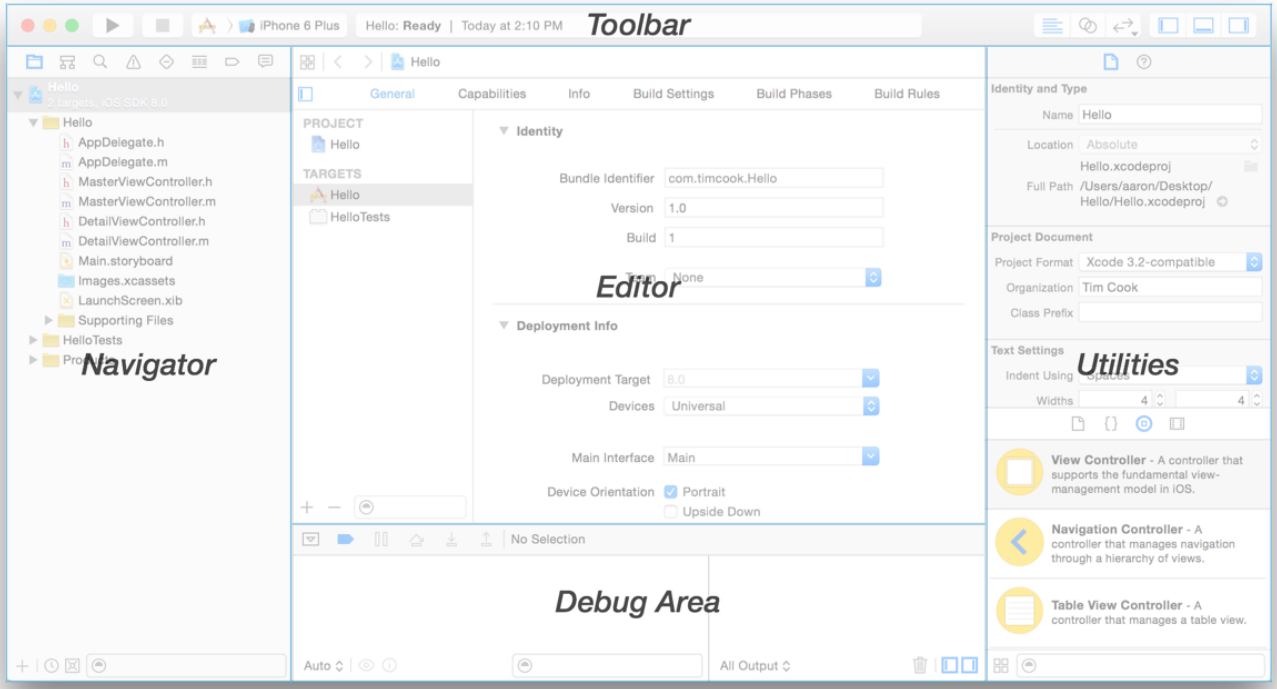


We'll go through the templates in more detail later. For now:

1. Select **Application** under **iOS** in the menu on the left
2. Select **Master-Detail Application**
3. Press **Next**
4. Enter "hello" in the **Product Name** text field
5. Enter your name in the **Organization Name** text field. For example: `John Appleseed`
6. Enter `com.<your user handle>` in the **Company Identifier** text field. For example: `com.john-appleseed`
7. Select **Objective-C** as your language
8. Leave the remaining options as they are and press **Next**
9. Select a folder to save your Xcode project in, and press **Create** (Tip: you may want to create a new folder called "Xcode Projects" to keep everything organized on your computer).

There are five main areas of the Xcode window:

Section	Location	Purpose
Toolbar	At the top, with the play / pause buttons	Easy access to common functions
Navigator	Left column	Find stuff in your project, like files, compiler warnings, and unit tests
Editor	Center column	Edit code, user interface files, and build settings
Utilities	Right column	Miscellaneous tools to make your life easier
Debug Area	Below the editor (hidden by default)	See logs and observe what happens while your app runs



Try out these three keyboard shortcuts, you'll probably use them often:

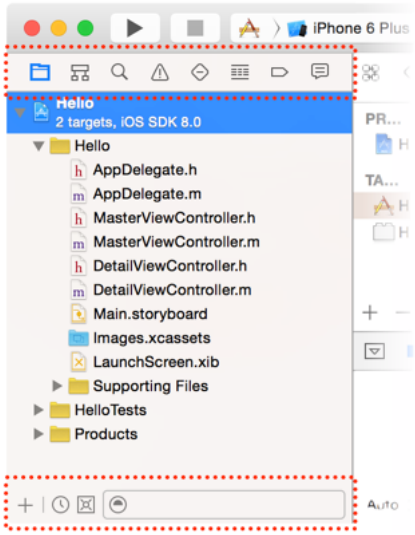
- ⌘**0** (command zero)
- ⌥⌘**0** (option, command, zero)
- ⇧⌘**Y** (shift, command, "Y")

All of these commands, and more, can be controlled in Xcode's **View** menu.

Xcode's Eight Great Navigators

We're going to explore the eight different navigators built-in to Xcode. They can be accessed quickly using ⌘**1** through ⌘**8** on your keyboard.

Starting with a folder icon, and ending with a chat bubble icon, each navigator has its own icon on the **navigator selector bar** in the top left corner. The **filter bar** in the bottom left corner helps you quickly find specific items. Both bars are marked below.



⌘1 - Project Navigator

linked to folder structure in your hard drive. Move files around in here, and they'll stay put on your hard drive.

⌘2 - Symbol Navigator

Navigate your project's class hierarchy. This is useful for finding a specific method or class if you can't remember which file it's in.

It is uncommon to use this navigator, since the ⇧⌘**O** (shift, command, "O") keyboard shortcut will usually find things quicker.

> *If you're not familiar with the phrases "class", "method", or "class hierarchy" in this context, don't worry: it's covered in the next checkpoints.*

⌘3 - Find Navigator

Find (and optionally replace) any string within your project.

⌘4 - Issue Navigator

Here, Xcode displays warnings, errors, and other issues found when opening, analyzing, and building your project.

- Warnings: warnings are displayed in yellow. They highlight a probable bug, which you can opt to ignore (though you usually shouldn't)
- Errors: errors are displayed in red. They must be resolved before you can run your app again.

⌘5 - Test Navigator

Navigate through, and run your app's Unit Tests.

> ***Unit tests*** *comprise code which you write to verify that other code works as expected.*

⌘6 - Debug Navigator

While your app is running, see the app's threads and associated stack information.

> *If you're not familiar with threads and stack information, think of it this way: the debug navigator shows you what your app is doing while it runs, so you can compare this to what you're expecting. You typically do this when something isn't working correctly, often as reported by a user. This is known as* ***debugging****.*

⌘7 - Breakpoint Navigator

"Breakpoints" are used in debugging. They pause your app when it reaches a certain line of code or condition so you can inspect what's happening.

The breakpoint navigator lets you view and delete breakpoints, as well as specify additional triggering conditions.

⌘8 - Log Navigator

Various tasks will generate log files, such as building your app, and continuous integration. These logs are displayed here.

1. Use the ⇧⌘**O** keyboard shortcut to find the **`viewDidLoad`** method in the DetailViewController class. (This is referred to as **`[DetailViewController -viewDidLoad]`**.)
2. Set a breakpoint on the line that calls **`[self configureView];`** by clicking on the line number.
3. Press ⌘**7** to activate the Breakpoint Navigator. Observe your breakpoint in the navigator.
4. Drag your breakpoint out of the navigator to remove it.
5. Press ⌘**1** to activate the Project Navigator.
6. Click on **`MasterViewController.m`** to open it in the Editor.
7. Delete the **`}`** at the end of the **`viewDidLoad`** method. Some errors and warnings will appear.
8. Press ⌘**4** to activate the Issue Navigator. Observe the issues that appear.
9. Press the play button in the toolbar. After you see the "Build Failed" alert, press ⌘**8** to activate the Log Navigator.
10. Browse the logs.

You should now have a basic understanding of the Xcode interface, especially how the Navigator and Editor areas interact. Send a message to your mentor with the questions you have.

assignment completed

COURSES

Full Stack Web Development

Frontend Web Development

UX Design

Android Development

iOS Development

ABOUT BLOC

Our Team | Jobs

Bloc Veterans Program

Employer Sponsored

FAQ

Blog

Engineering Blog

Refer-a-Friend

Privacy Policy

Terms of Service

MADE BY BLOC

Tech Talks & Resources

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Command Line

The Command Line

> *"If you have any trouble sounding condescending, find a UNIX user to show you how it's done." — Scott Adams, Dilbert Cartoonist*

You know that super cool green-on-black hacker-looking screen you always see in the movies? It's not really that cool in real life... it's way cooler. Nearly every developer needs this tool. This checkpoint will familiarize you with the basic commands and principles of the command line.

What is the Command Line?

The command line allows developers to navigate their filesystem, modify files, execute programs, install software, create new applications, and much more. The terms command line, command line prompt, terminal, and **shell** are used interchangeably in the developer community.

There are many different "flavors" of **Unix** shells that function on any Unix based operating system such as Linux and OS X. However, the content in this checkpoint is shell and operating system agnostic.

Why Do I Need to Learn the Command Line?

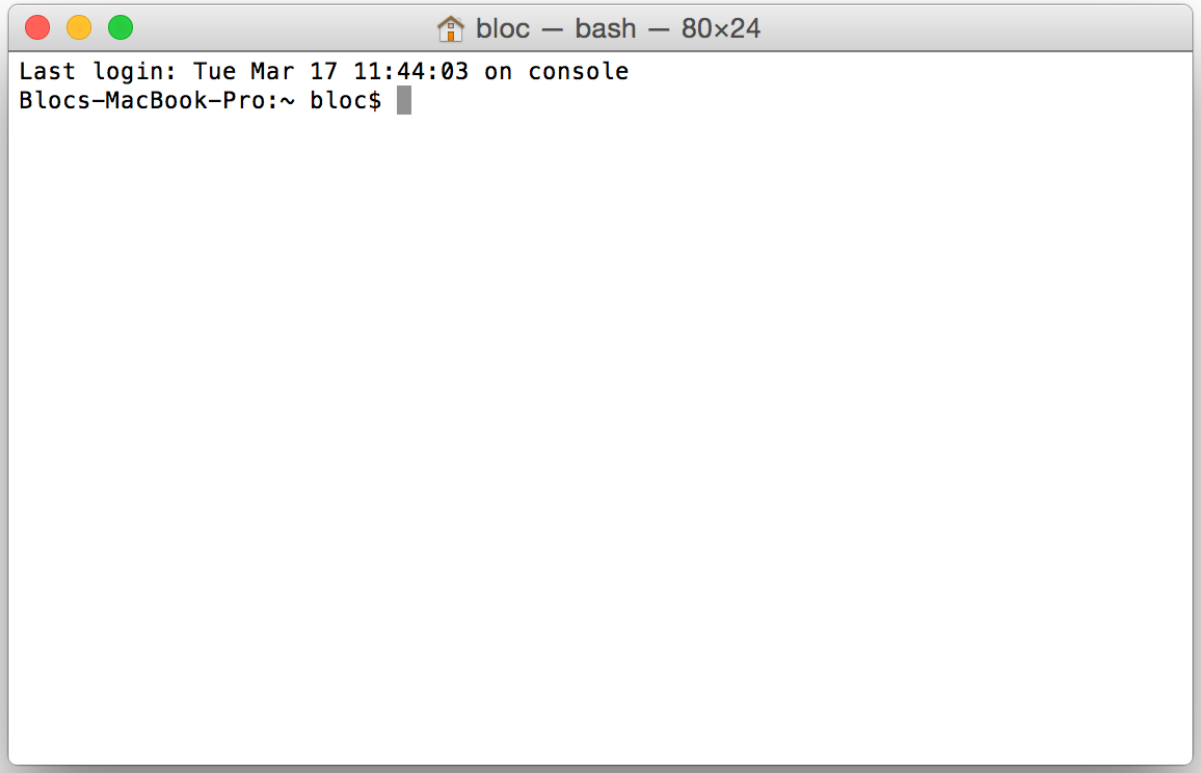
The command line is often the only, or most efficient wa ? :cess systems, install software, and execute programs and tests.

How Do I Open the Command Line?

Use the following instructions to open the command line:

Operating System	Guide
OS X	The command line utility packaged with the operating system is **Terminal**. To access Terminal, use Finder and go to the **`Applications > Utilities`** folder or use Spotlight and type *Terminal*.
Windows	On Windows, we recommend using **Git BASH** for all command line activities. Download it from the Git BASH website, install it, and open it when the installation finishes.
Linux	Read Ubuntu's **"Starting a Terminal"** section.

Where am I?

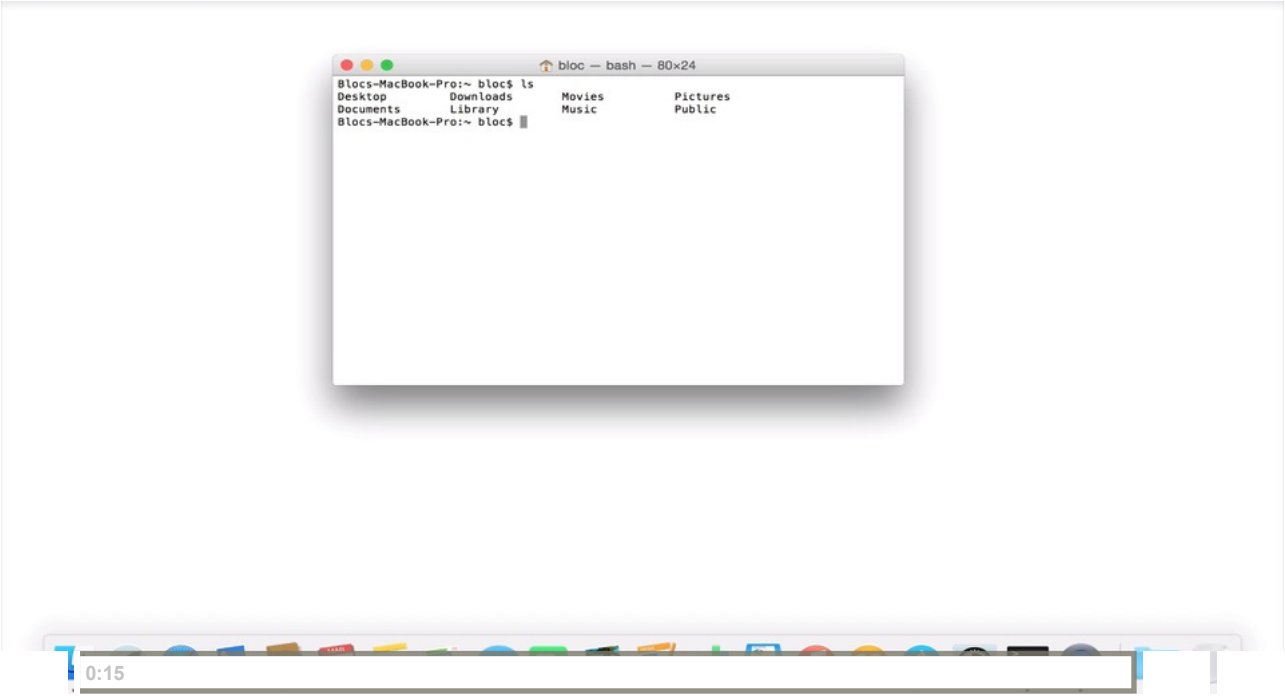


> *Terminal in OS X*

When the shell is opened, we are presented with a prompt. At this point, the shell is complacent and waiting for instruction. We can dictate any command to the shell. Every command is executed by the operating system after return is pressed. Type **`ls`** (lower case 'L', lower case 'S') and then press return ⏎. After executing **`ls`**, type **`open .`** (don't forget the space) and press return ⏎. Feel free to copy the commands:

```
~
$ ls
Applications    Desktop        Downloads        Movies        Pictures
Documents           Library        Music            Public
$ open .
```

Here is a video demonstration for OS X:



The operating system executed the **`ls`** command after return was pressed. The **`ls`** command listed the contents of our current directory. A directory in Unix is the same thing as a folder in Finder (OS X) or a folder in File Explorer (Windows). Notice the names in the output of **`ls`** match the folder titles in Finder.

The **`open .`** command directed the operating system to open 'this', the period, which refers to the current directory. The operating system chose Finder to open the directory and display its contents.

Similar to the Finder window in OS X, the shell has a default location when opened. The Finder will default to the last folder that we were operating in when we closed it. However, the shell always defaults to a directory specified in its configuration file. By default, the shell places us in our **`HOME`** directory.

What Can I Do Now?

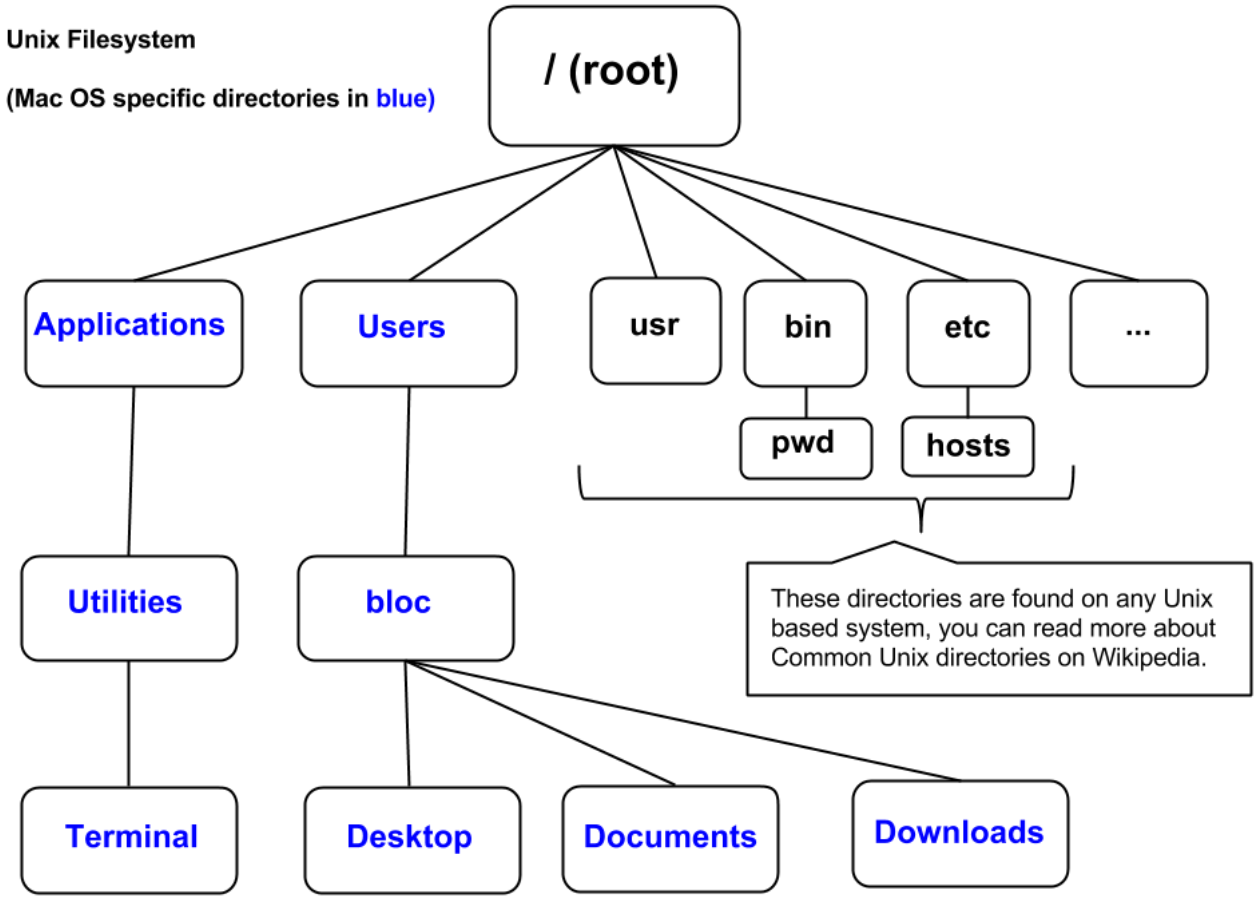
Now that we have a frame of reference, let's go over some basic commands to traverse the filesystem. Verify our current working directory by executing the **`pwd`** command. The **`pwd`** command stands for 'print working directory'.

~

```
$ pwd
/Users/bloc
```

The **`pwd`** command prints the name of our current directory to the command prompt. For OS X users it will be **`/Users/your_username`**, Windows users will see **`/c/home/your_username`**, and Linux users will see **`/home/your_username`**. This is our **`HOME`** directory. The **`~`** character is also a reference to the **`HOME`** directory in any Unix based system.

The filesystem is a tree, at the top of the tree is the root denoted by **`/`**. Below the root there are branches and other **vertices**). The following graphic illustrates parts of the OS X filesystem:



The **cd** (change directory) command offers the ability to traverse the branches to get to other vertices. Let's move through the filesystem by using the **cd** command:

```
~
$ cd ..
```

This command navigates the shell to its immediate *parent* directory. The parent of any vertex in the graph is the vertex that is connected directly above. Given the graph above, the parent of **bloc** is **Users**. The **..** is another way to reference the parent directory from the current working directory in the shell.

Perform a **pwd** again. The output should display the parent vertex of the **bloc** directory: **/Users** on OS X, **/c/home** in Git Bash, and **/home** on Linux machines.

```
$ pwd
/Users
```

Pay special attention to the naming. In Windows or Linux the **/c/home** and **/home** directories are *not* your **HOME** directory, but rather the parent vertices of your **HOME** directory, respectively.

Performing an **ls** will display the **HOME** directory:

```
~
$ ls
Guest     Shared    your_username
```

To get back to the **HOME** directory, use **cd** again:

Executing the **`ls`** command again will output the contents of the HOME directory:

~

```
$ ls
Applications    Desktop        Downloads        Movies        Pictures
Documents           Library        Music            Public
```

In our previous example, we used the **`..`** operator to move to the directory above our **`HOME`** directory. The **`..`** operator is an example of a *relative path*. Relative paths refer to other files or directories in the filesystem with respect to the current working directory.

In the graphic above, the relative path from the **`Desktop`** directory to the **`Users`** directory is: **`../../`** since the **`Users`** directory is two directories immediately above the **`Desktop`** directory in the filesystem hierarchy. Imagine we want to get a reference to the **`hosts`** file from the **`Desktop`** directory. Using our graphic above, the relative path from **`Desktop`** to **`/etc/hosts`** is **`../../../etc/hosts`**.

Unix also gives us the ability to reference the **`hosts`** file using *absolute paths*. An absolute path is the fully qualified path of a file or directory starting from the root. The absolute path of the **`hosts`** file is **`/etc/hosts`**. The absolute path of the **`Desktop`** directory is **`/Users/bloc/Desktop`**. Notice how both examples start at the root **`/`**.

Addresses are like paths in Unix. We can say Charlie's house is two doors down from mine. Or, we can address his house by the full address: 1234 Same Street. The former is relative as it's a correlation between a point of reference and Charlie's house whereas the latter is the exact, or absolute, address. They both reference the same house, but in a different manner.

Read more about **Unix paths**.

How Can I Manipulate Files and Directories?

We can make our way through the filesystem, but that doesn't offer us much utility. This is a list of useful commands:

Command	Guide
mkdir	Make a directory.
file	Determine the type of a file.
touch	Create file or change file modification access time.
cp	Copy a file.
mv	Move or rename a file.
rm	Remove a file.
rmdir	Remove a directory.
history	Display a chronological list of the previously issued commands.

Let's go through an example that uses each of the above commands. First issue the **`mkdir`** command:

~

```
$ mkdir bloc
$ ls
bloc
$ file bloc
bloc: directory
```

~

```
$ cd bloc
$ ls
```

The output is blank. Use **`touch`** to create a new file:

~/bloc

```
$ touch bloc_file
$ ls
bloc_file
$ file bloc_file
bloc_file: empty
```

`touch` created an empty file named **`bloc_file`** with an unspecified type. Copy **`bloc_file`**, rename it and remove the original:

~/bloc

```
$ cp bloc_file new_bloc_file
$ ls
bloc_file    new_bloc_file
$ mv new_bloc_file renamed_bloc_file
$ ls
bloc_file    renamed_bloc_file
$ rm bloc_file
$ ls
renamed_bloc_file
```

Remove **`renamed_bloc_file`**, traverse up and out of the **`bloc`** directory and remove the **`bloc`** directory:

~

```
$ rm renamed_bloc_file
$ ls

$ cd ..
$ ls
Applications    Desktop        Downloads        Movies        Pictures
Documents           Library        Music            Public        bloc
$ rmdir bloc
$ ls
Applications    Desktop        Downloads        Movies        Pictures
Documents           Library        Music            Public
```

The output shows **`bloc`** no longer exists. Finally, issue the history command to display the list of commands we executed in chronological order with the oldest at the top.

~

```
285  mkdir bloc
286  ls
287  file bloc
288  ls
289  cd bloc
290  ls
291  touch bloc_file
292  file bloc_file
293  ls
294  cp bloc_file new_bloc_file
295  ls
296  mv new_bloc_file renamed_bloc_file
297  ls
298  rm bloc_file
299  ls
300  rm renamed_bloc_file
301  cd ..
302  ls
303  rmdir bloc
304  history
```

What Are Man Pages?

Man pages, **short for manual pages**, are embedded in every distribution of Unix. Man pages can be terse but are a valuable resource. Let's see an example of a man page:

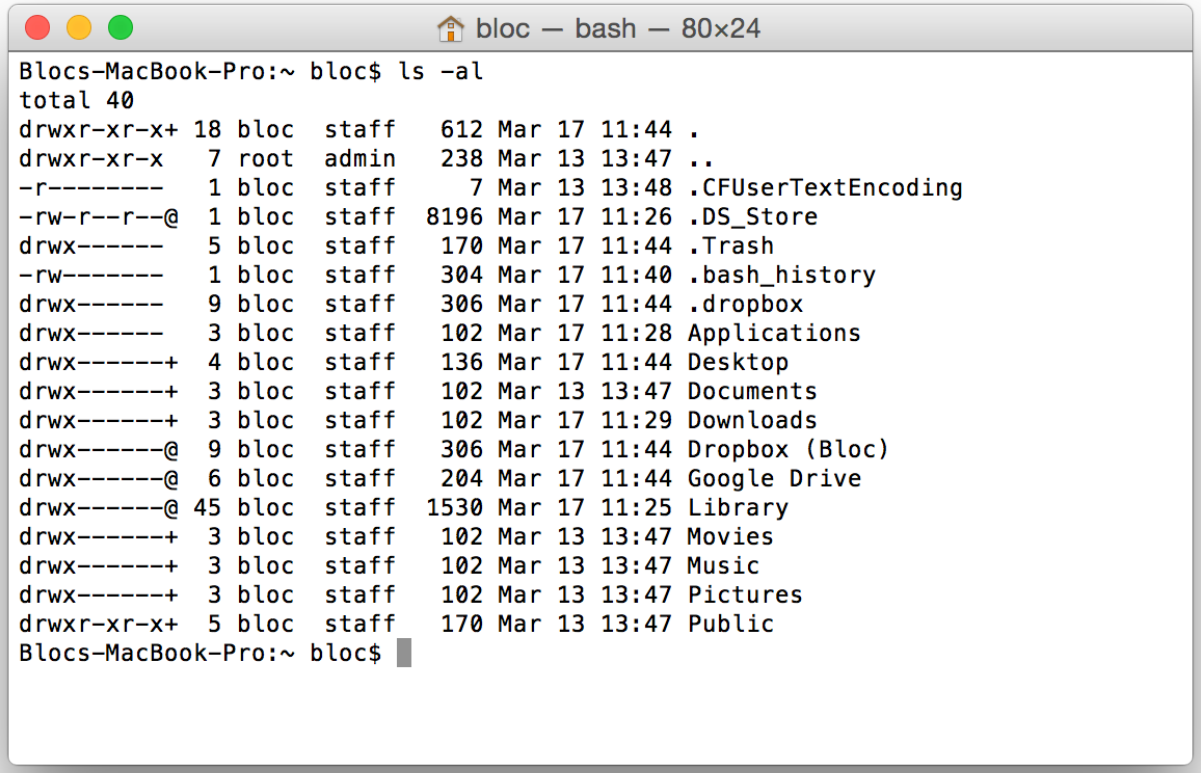
```
$ man ls
```

> *Use the 'j' and 'k', the 'spacebar' and 'u' keys, or ↓ and ↑ to scroll up and down. Use the 'q' key to escape out of the man page.*

What are Command Options?

Notice in the description of the **`ls`** man page there are many options. This is typical of most Unix commands. Almost all commands have the ability to accept input. Options are also commonly referred to as *flags*. The **`ls`** command takes many flags and exemplifies a typical Unix command. Issue a **`cd`** with no options; this will navigate the shell to our **`HOME`** directory and then perform **`ls -a -l`**:

```
$ cd
$ ls -a -l
total 15
drwxr-xr-x+ 39 your_username  staff   1326 Mar 18 17:31 .
drwxr-xr-x   7 root       admin    238 Mar 13 13:47 ..
-r--------   1 your_username  staff      7 Mar  8 21:26 .CFUserTextEncoding
-rw-r--r--@  1 your_username  staff  12292 Mar 17 14:37 .DS_Store
drwx------   2 your_username  staff     68 Mar 18 13:07 .Trash
-rw-------   1 your_username  staff    638 Mar 17 11:49 .bash_history
drwx------   7 your_username  staff    238 Mar 13 16:26 Applications
drwx------+  4 your_username  staff    136 Mar 18 10:27 Desktop
drwx------+  7 your_username  staff    238 Mar 11 13:40 Documents
drwx------+  8 your_username  staff    272 Mar 18 16:01 Downloads
drwx------@ 47 your_username  staff   1598 Mar 16 18:13 Library
drwx------+  3 your_username  staff    102 Mar  8 21:25 Movies
drwx------+  3 your_username  staff    102 Mar  8 21:25 Music
drwx------+  5 your_username  staff    170 Mar 17 13:00 Pictures
drwxr-xr-x+  5 your_username  staff    170 Mar  8 21:25 Public
```



> *Our output of* **`ls -a -l`** *in Terminal*

In the example, the '-a' instructs **`ls`** to "include directory entries whose names begin with a dot **`.`**". Any file that begins with a dot in Unix is a hidden file, so we instructed **`ls`** to display all hidden files. The '-l' flag tells **`ls`** to display the contents of the directory in long format. Use man pages to read more about the options for **`ls`** or any other command.

What is Tab Completion?

Does typing every single folder and file name sound like fun? If so, skip this section. For the remaining 99.999% of us who answered, "no" to the previous question, there's tab completion. Try the following:

```
$ ls -l /bin/p<TAB>
pax  ps   pwd
```

After hitting the tab, the shell shows all files and directories in **`/bin`** that start with the letter 'p'. The shell also shows a few commands like **`pax`**, **`ps`** and **`pwd`**. If the next character we type is 's' then the shell will eliminate **`pax`** and **`pwd`** from the list.

```
$ ls -l /bin/ps<TAB>
-rwsr-xr-x  1 root  wheel  46688 Sep  9  2014 /bin/ps
```

Tab completion works with almost every Unix command and is a handy feature.

How Can I get Additional Help?

- **The Command Line Crash Course**
- **Bash Guide**

Using the command line:

- Create a directory named `checkpoint_exercise` in your `HOME` directory.
- Navigate into `checkpoint_exercise`.
- Create a file named `bloc_cl_checkpoint.txt` in the `checkpoint_exercise` directory.
- Open and edit `bloc_cl_checkpoint.txt` so that it reads, "Hello World!"
- Print the contents of the file in your shell.
- Rename `bloc_cl_checkpoint.txt` to `cl_checkpoint.txt`
- Print the contents of the renamed file in your shell again.
- Copy and paste the output from your history command to show your mentor how you did it.

> - *Hint (OS X): use* **`open bloc_cl_checkpoint.txt`** *to open the file in TextEdit.*
> - *Hint (Windows): use* **`notepad bloc_cl_checkpoint.txt`** *to open the file in NotePad.*
> - *Hint: you can use the* **`cat`** *command to display the contents of a file.*

Use **Markdown formatting** to format any code snippets you include in your description. Here is an example of how to format the `ls` command in markdown.

```bash
$ ls
```

assignment completed

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Git and GitHub

Git and GitHub

Throughout your life as a developer, you will find that nearly every team of software engineers relies on some sort of source tracking system. They employ a mechanism which allows them to share a consistent code base while remaining autonomous from one another on separate computers. This system is known as **version control**.

> ***Deep Dive: Introduction to Git***

GitHub

GitHub is a service which maintains a copy of your Git **repository** – another term for code base – in the cloud. Github is where Bloc keeps all of its repositories, including the exercises required for the subsequent checkpoints. If you haven't already, take a few moments to **create a Github account**.

Command Line Tool

Download and install **Git for OS X**. There are several gr ? ıser interfaces for Git and they will be discussed in this checkpoint. However, fc the purposes of setup and fundamental developer skills, it's best to know how to use Git in the command line.

Git Identity

Your git identity is associated with every change you make to the repositories you contribute to. Therefore, establishing it helps others recognize you when they investigate the changes you've committed to the repository. Perform the following command and use your real name:

Terminal

```
$ git config --global user.name "FirstName LastName"
```

When contributing to large projects, a co-contributor may wish to communicate with you. Give them the opportunity by configuring your email:

Terminal

```
$ git config --global user.email you@website.com
```

Push Configuration

Terminal

```
$ git config --global push.default simple
```

(The alternative is "matching"; the difference is discussed on **this Stack Overflow** question.)

SSH Keys

Your SSH key is used by GitHub to identify your computer securely. These are private pieces of information which you should never share publicly.

Here's our SSH key:

Terminal

```
Just kidding…
```

Generate an SSH key for your machine by following **these instructions.**. Once your key is generated, follow the remaining steps in GitHub's guide to add them to your account.

Fork

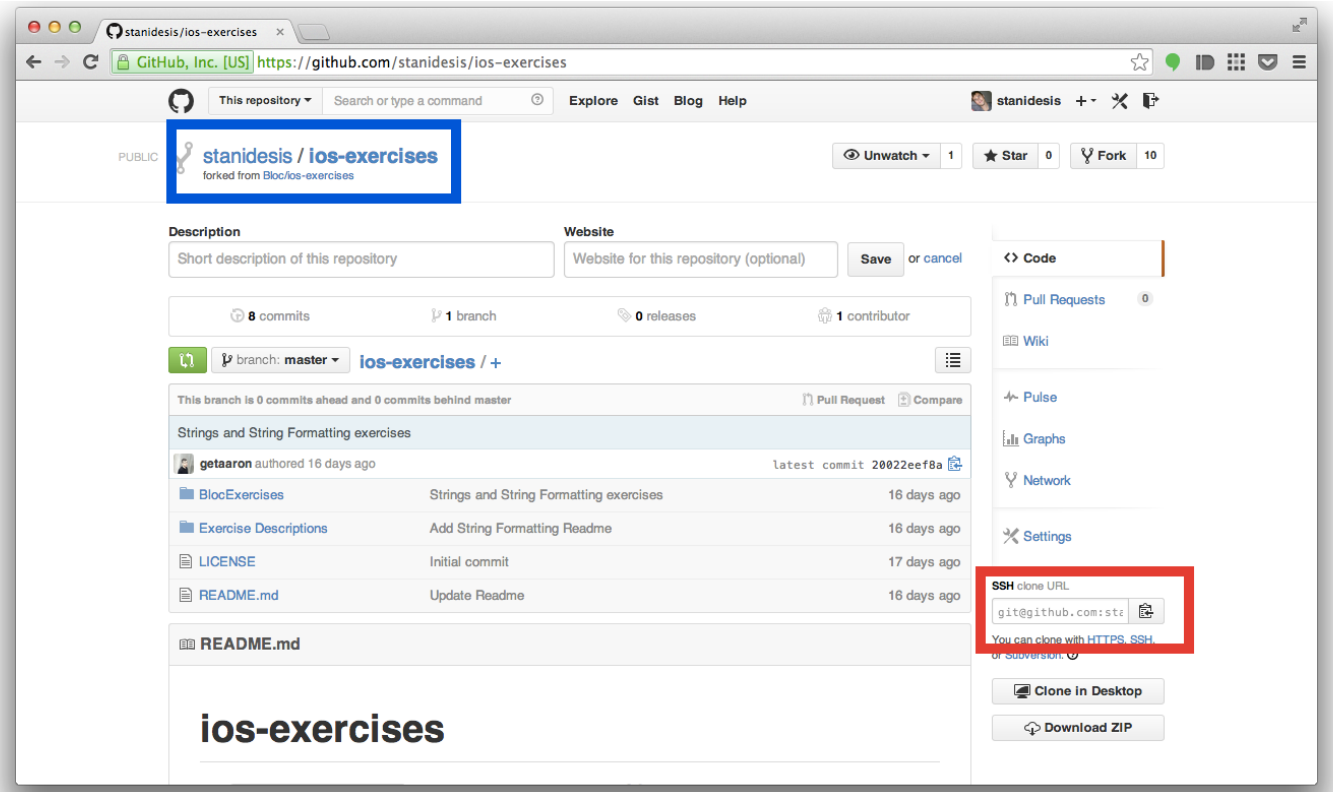
Navigate to **Bloc's iOS exercise repository** and click **Fork**. Forking is the process of copying an existing repository on GitHub and pasting it as a brand new one under your account.

> *Your forked repository will not receive updates made to the original and vice-versa.*

As you work through the exercises, you'll be working on your fork of the **`ios-exercises`** repo, not on Bloc's main fork.

Clone

Copy the git clone address of your forked **`ios-exercise`** repository:



> *The **red** box indicates the git URL you need to copy. Make sure you see your GitHub user name where the **blue** box indicates. This ensures that you've navigated to your fork and not the original.*

Open your Terminal and **cd** into the directory where you keep your projects. We recommend a **work** folder inside your home directory or on your desktop but choose whatever works best for you. Once inside this directory, enter the following command using your copied clone address:

/Users/user-name/work/

```
$ git clone [paste clipboard contents]
```

> *Your final command should look like* **`git clone git@github.com:user-name/ios-exercises.git`**.

After the operation completes, you should see a brand new **ios-exercises** directory in your work folder. **cd** into that folder and enter the following command:

/Users/user-name/work/ios-exercises

```
$ git status
```

Which should print out something resembling:

/Users/user-name/work/ios-exercises

```
# On branch master
nothing to commit, working directory clean
```

Sync

After you've forked and cloned a repository, changes made to the original will not be reflected in your local copy unless you perform a **sync**. Begin by executing **`remote`**:

/Users/user-name/work/ios-exercises

```
$ git remote -v
origin  https://github.com/[user-name]/ios-exercises.git (fetch)
origin  https://github.com/[user-name]/ios-exercises.git (push)
```

This command displays the current remote locations of your repository.

Now, add a reference to Bloc's original repository with the following command:

/Users/user-name/work/ios-exercises

```
$ git remote add upstream https://github.com/Bloc/ios-exercises.git
```

Check your remote repositories again:

/Users/user-name/work/ios-exercises

```
$ git remote -v
origin  https://github.com/{your-user-name}/ios-exercises.git (fetch)
origin  https://github.com/{your-user-name}/ios-exercises.git (push)
upstream  https://github.com/Bloc/ios-exercises.git (fetch) # New entries
upstream  https://github.com/Bloc/ios-exercises.git (push)
```

Now you're ready to perform a full sync by first **fetching** and then **merging** the upstream changes:

/Users/user-name/work/ios-exercises

```
$ git fetch upstream
# Grab the upstream remote's branches
remote: Counting objects: 75, done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 62 (delta 27), reused 44 (delta 9)
Unpacking objects: 100% (62/62), done.
From https://github.com/Bloc/ios-exercises
 * [new branch]      master     -> upstream/master
```

Fetch retrieves the latest changes for a given repository. Note that the sample output below represents a merge after changes have been made to the upstream (Bloc's) repo. Since you just forked and cloned it, there probably won't be any changes. When there aren't new changes to merge, you'll see a message stating: **`Already up to date`**. The output below displays merge results if you were in fact out of sync with the upstream repo

/Users/user-name/work/ios-exercises

```
$ git merge upstream/master
Updating 34e91da..16c56ad
Fast-forward
  README.md                 |    5 +++--
  1 file changed, 3 insertions(+), 2 deletions(-)
```

> *Merge will re-commit changes found in one branch onto another*

You've done it! Occasionally, Bloc will iterate and improve the code found in the **`ios-exercises`** repository. In order for your fork to receive the benefits of our hard work, you will need to perform a sync again, e.g. **`fetch`** followed by a **`merge`**.

Ignoring Certain Files with `.gitignore`

Sometimes there are certain files that we never want to commit. For example, Xcode files that remember which windows you had open shouldn't be committed because no other user will ever care about that.

A list of files to ignore is maintained in a file called `.gitignore`. GitHub maintains **a list of common ignore files for different languages**.

The `ios-exercises` project already includes one, so you don't need to make one. If you want, **read it here**. It's similar to the one maintained by GitHub, but with comments explaining what the different file types are.

Commit

A git **Commit** is a collection of changes which you submit to your repository. In best practice they are grouped in such a way as to represent a single understandable adjustment. Common commits will:

- add a new feature,
- fix a bug, or
- re-arranging files.

A commit is also associated with an optional message which helps other contributors understand which task was completed by that commit.

Let's make your first change to `ios-exercises` by performing the following commands:

/Users/user-name/work/ios-exercises

```
$ echo "\nMy first git change" >> README.md
$ git status
```

`git status` outputs the state of your repository. It should print something which resembles the following:

/Users/user-name/work/ios-exercises

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git is telling us that `README.md` has been modified. To see which changes have occurred in that file, perform the following git command:

/Users/user-name/work/ios-exercises

```
$ git diff README.md
```

`git diff` outputs the deletions, insertions and modifications made to a file which have yet to be committed

/Users/user-name/work/ios-exercises

```
index 6235ef4..612dc90 100644
--- a/README.md
+++ b/README.md
@@ -2,4 +2,5 @@ ios-exercises
=============

 - `BlocExercises.xcodeproj` contains the Xcode project with iOS exercises.
 -- Specific assignments are in [Exercise Descriptions](Exercise%20Descriptions/).
 \ No newline at end of file
 +- Specific assignments are in [Exercise Descriptions](Exercise%20Descriptions/).
 +My first git change
```

After reminding yourself of all the changes you've made, you're ready to make your first commit. Press **q** to exit the diff view you were just reviewing.

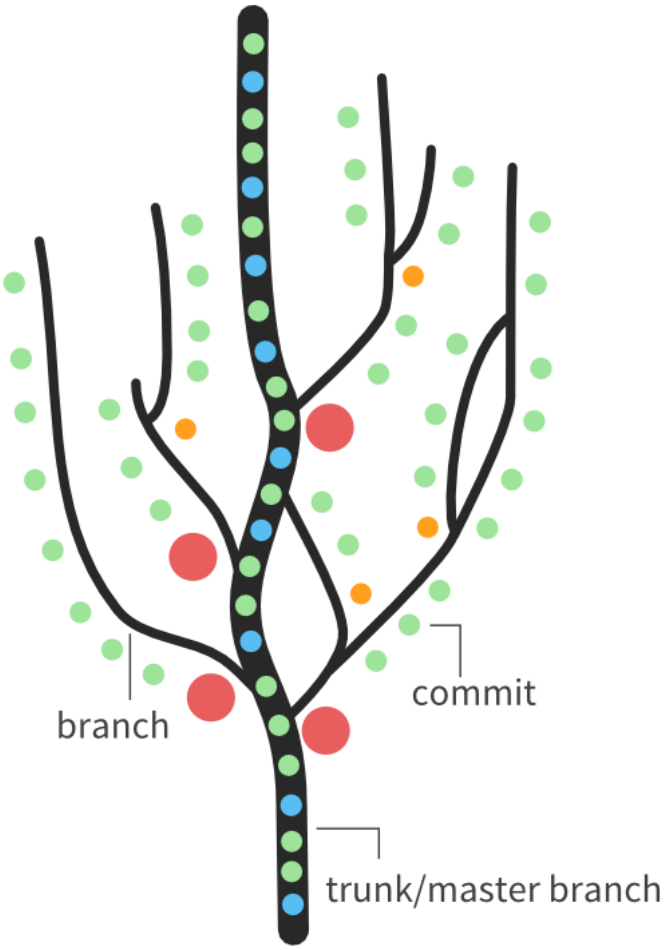
/Users/user-name/work/ios-exercises

```
# First, add all of the un-staged changes
$ git add .
# Second, perform the commit
$ git commit -m "My first commit to ios-exercises"
# Push your changes to GitHub
$ git push
```

`git push` and its complimentary command, **`git pull`** deal with syncing changes between your local repository and its remote origin on GitHub. Pushing sends your changes to Github, pulling recovers any changes not found in your local copy.

Branches

A simple way to visualize your git repository is to imagine a tree. Your main branch, usually **`master`**, is the trunk or primary root of the tree.



`master` refers to the version of your project which is live, published or in some form accessible by the public. Another common branch developers include in their repositories is aptly named **`dev`**, short for "development." If the **`master`** branch is v1.0.0, **`dev`** is where v1.0.1 is currently being worked on.

View your available local branches with the following command:

/Users/user-name/work/ios-exercises

```
$ git branch
```

Which should output:

/Users/user-name/work/ios-exercises

```
* master
```

Pull Requests

Later, when working on projects with the aide of your mentor you will submit **pull requests**. A pull request submits a branch or fork of your repository for consideration. If you want your branch to become part of **`master`**, it's best practice to ask others to review your changes before merging them in. Create a new branch:

/Users/user-name/work/ios-exercises

```
$ git branch my-first-branch
```

```
$ git branch
* master
  my-first-branch
```

You see two branches and **`master`** is the active branch, indicated by the **`*`**. When you commit changes they are always committed to the active branch. Switch to your new branch by performing a **`checkout`** command:

/Users/user-name/work/ios-exercises

```
$ git checkout my-first-branch
Switched to branch 'my-first-branch'
```

`my-first-branch` is identical to **`master`** initially because we performed our **`git commit`** command while **`master`** was the active branch. However, any changes we commit henceforth will be strictly associated with **`my-first-branch`** and that branch *only* until we check out a different branch.

Summary

Git is like the undo command on steroids. Using this tool, you can maintain an extensive history of your app's code. Here are the basic concepts you should understand:

- **Repository.** A collection of code and other files, contained in at least one branch.
- **Branch.** Within a repository, each branch contains a different version of your code.
- **Fork.** A copy of a repository which may have some changes.
- **Commit.** A specific change. Each branch is essentially a list of commits.
- **Merge.** Merge some commits, or an entire branch, into another branch.
- **Push.** Sends changes on your local computer to the remote repository (usually GitHub).
- **Pull**. Copy changes that others have made from the remote repository to your local computer.
- **Pull Request.** A request for someone else to review your changes and merge them in to a specific repository and branch.

Make a unique change inside of **`my-first-branch`**, commit it, and push it to GitHub using the following command:

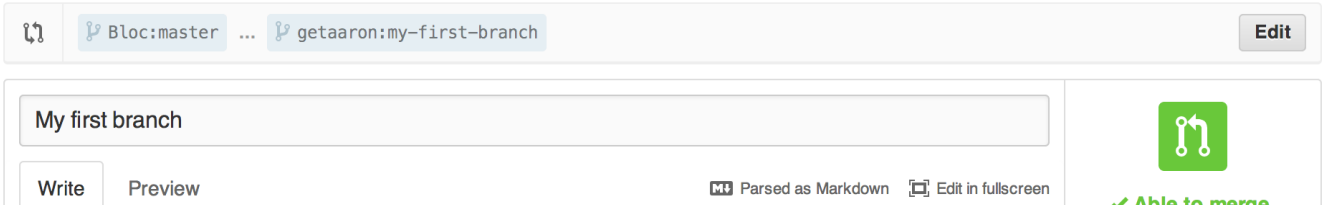
/Users/user-name/work/ios-exercises

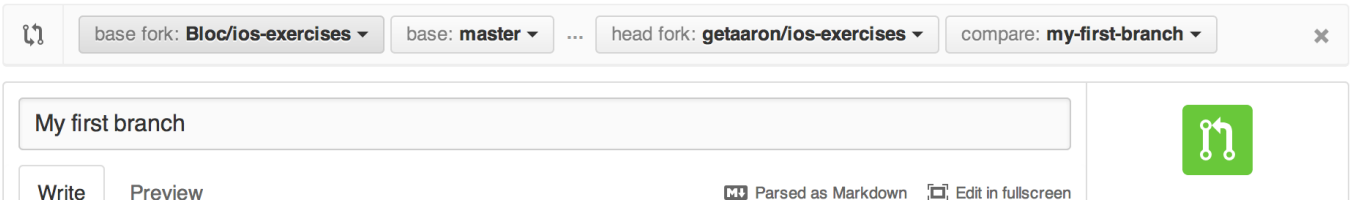
```
$ git push origin my-first-branch
```

Open your **`ios-exercises`** repo page in GitHub and you'll see a green button that reads "Compare & pull request".

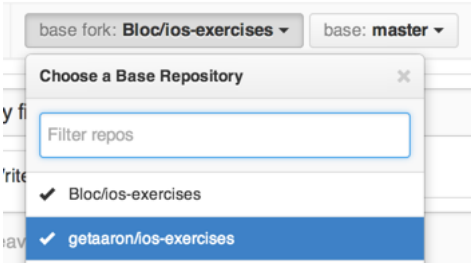
Make sure your pull request is requesting to merge your changes into your fork, **not Bloc's main fork**.

If **`Bloc:master`** appears, like so:





Click the "base fork" drop down, and select your fork:



Review the diff and click the green "Create pull request" button. **Post a link to the pull request in the assignment box** and speak with your mentor about the pull request work flow that they use as a professional developer.

assignment completed

COURSES

Full Stack Web Development
Frontend Web Development
UX Design
Android Development
iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy
Terms of Service

MADE BY BLOC

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Objective-C Syntax

Objective-C is the primary language of Mac OS X and iOS applications. For years it was the only option. This is no longer the case with the new arrival of the Swift language. Despite being older, Objective-C is still used by the overwhelming majority of the iOS development community. Either way, it is more important to learn the fundamentals of programming in general, than any one language specifically.

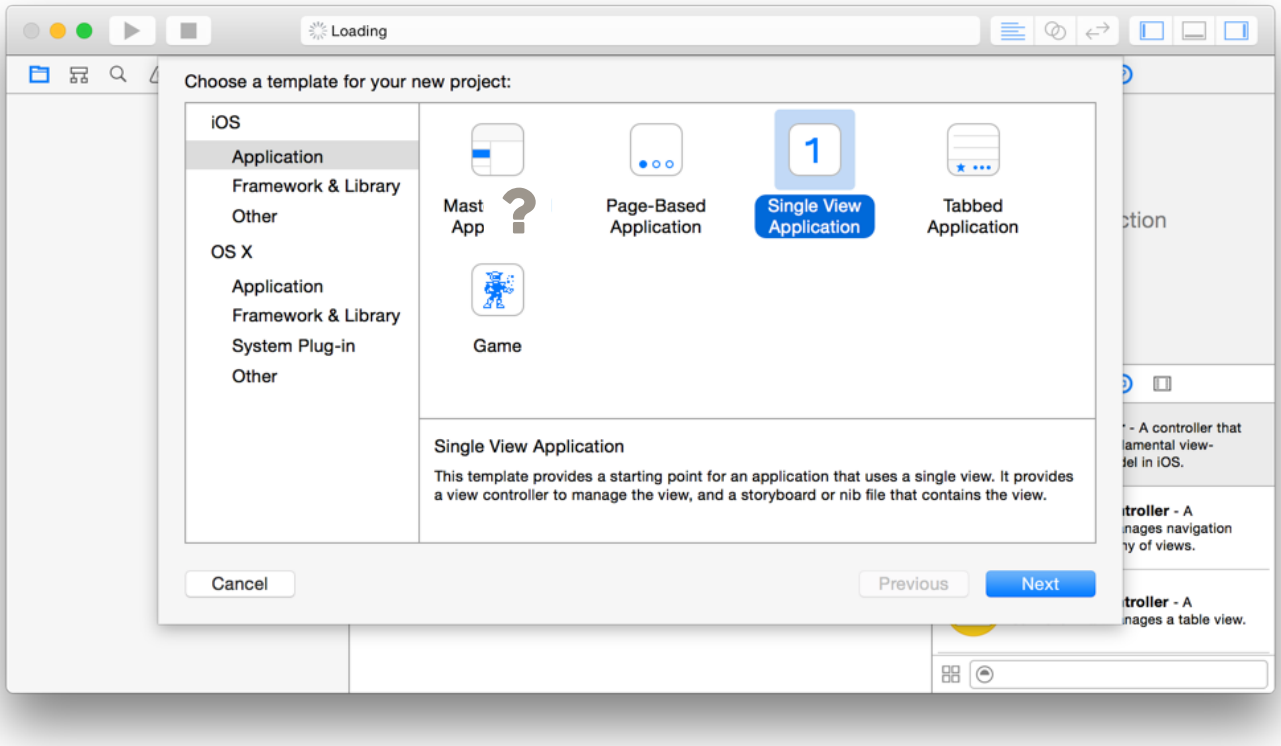
In this checkpoint you will be introduced to programming and the Objective-C language.

The exercises in this checkpoint cover:

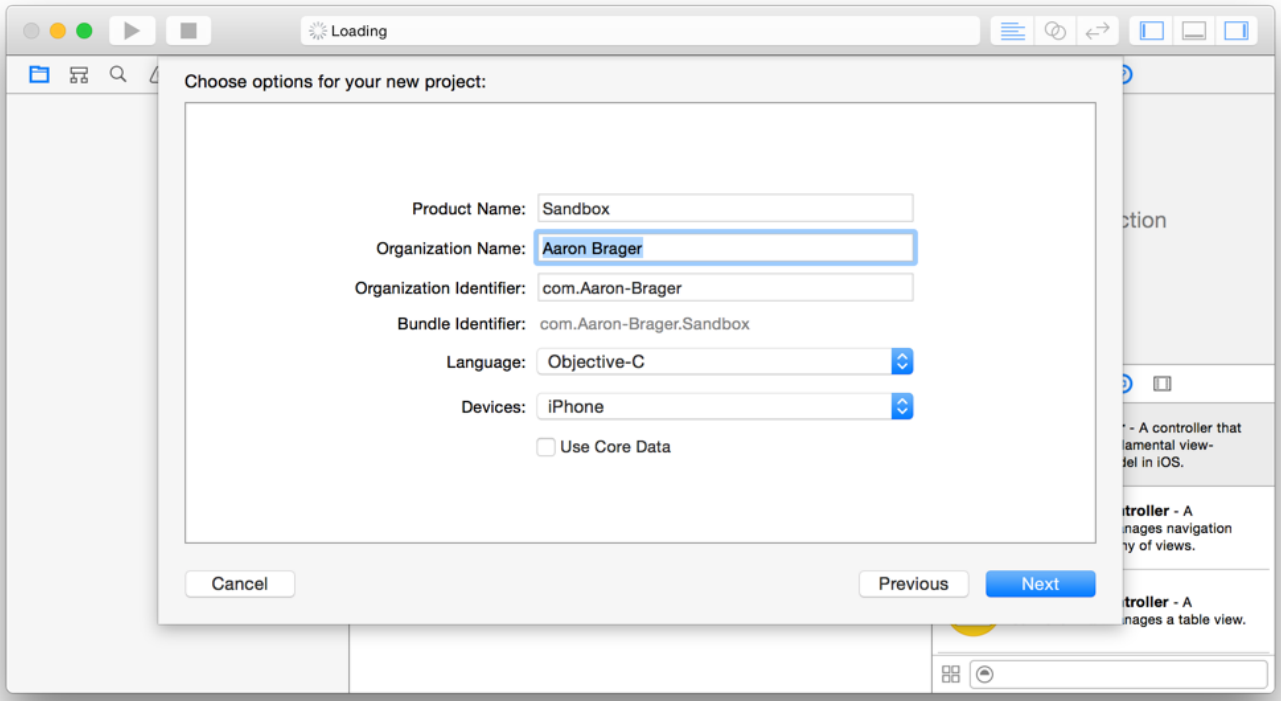
- Objective-C syntax
- Printing things to the console
- How to use strings

Every craftsman needs a worktable and every painter needs a canvas. A programmer is no exception. We'll need a place to put all the wonderful code we're going to write. We'll use Xcode to make the simplest type of iOS application.

Open Xcode and create a new project. For the project template we'll select **Single View Application** under the iOS Application section.



On the next screen, enter "Sandbox" for the **Product Name** -- it doesn't matter what you enter for **Organization Name** and **Company Identifier**. Select **Objective-C** under **Language**, **iPhone** under **Devices**, and then press **Next**.



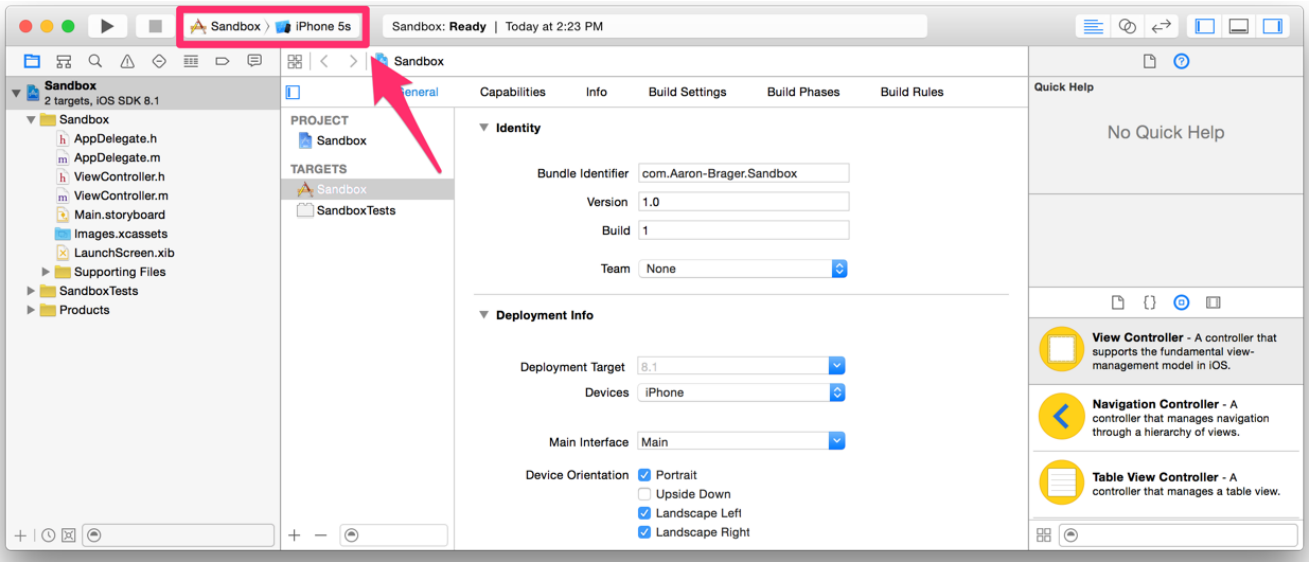
On the next screen pick where you want to save the project and click **Create**.

> *Note that you'll use this project to experiment with code throughout the Objective-C checkpoints. You'll use a **different** project to complete assignments though. We'll set up the other project (named `ios-exercises`) in the next checkpoint.*

Running the Application

Now we have our first application. Let's try running it.

Before we run it you may need to change run target:



This option tells Xcode where to run the project. For an iOS project it needs to either run on an iOS device or a simulator.

We'll change it to use the iPhone 5s simulator so we don't have to connect a real device.

Click the play button in the upper left hand corner or press ⌘**R**.

We should see an empty iPhone application open.

Open **`AppDelegate.m`** in Xcode's Project Navigator.

When you open it you'll see a bunch of code has already been written for us. This code is called *boilerplate code*. It is the minimum amount of

transitions of the application to and from the background.

For these early checkpoints we will be primary working inside this file to learn the basics of programming in Objective-C.

Focus on a chunk of code near the beginning of the file that looks like this

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    return YES;
}
```

This chunk of code is called a *method*. We'll cover methods in more depth later. For now think of it as a block of code that can be run at a later time by 'calling' it.

Now notice the line that reads **`// Override point for customization after application launch.`**

This line is called a **comment**. A comment is simply code that does not run. Any code that comes after two slashes (**`//`**) will not run. Comments are useful for adding instructions and context in your program. This particular comment is explaining what this method does. It's telling us that this method will run once after the application finishes launching. Let's test that by adding some code to this method.

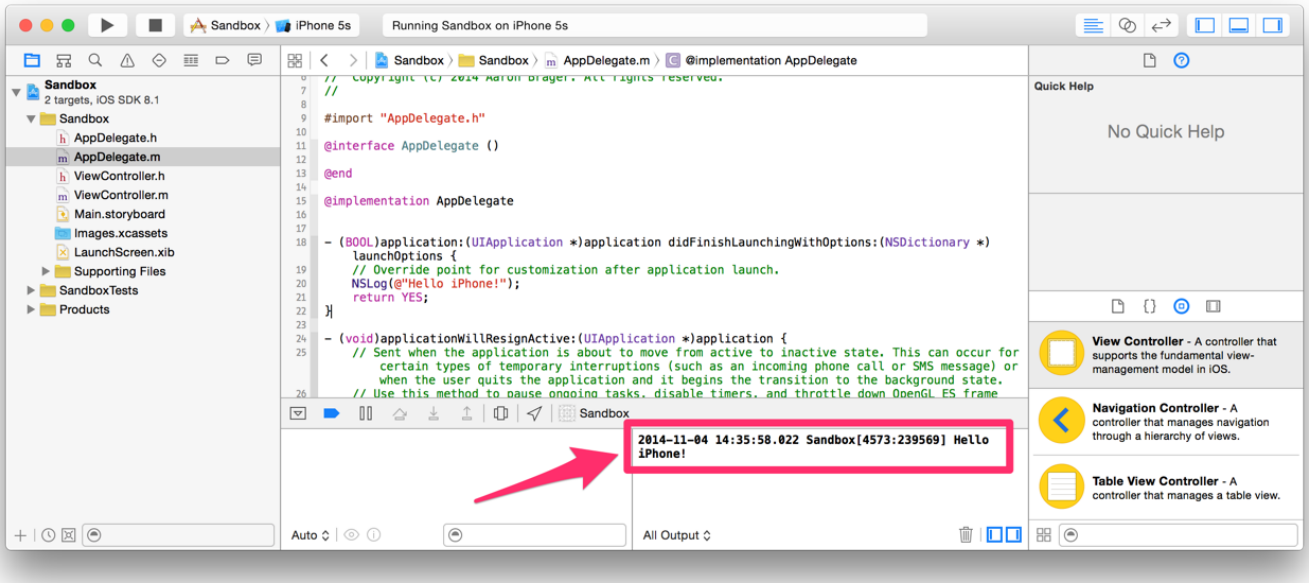
AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
+     NSLog(@"Hello iPhone!");
    return YES;
}
```

> ***Note:*** *A line of this code is highlighted in green. This convention is used to show new code that has been added.*

The code that we added is called an **`NSLog`**. When this line executes, text will print to the console.

Run the application again and you should see it log the text **`"Hello iPhone!"`**.



that reads **`NSLog(@"Hello, iPhone!");`**.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
-     NSLog(@"Hello iPhone!");
+     //NSLog(@"Hello iPhone!");
    return YES;
}
```

> ***Note:*** *This code snippet has a red highlighted line. This is convention for code that has been removed.*

To comment or uncomment code quickly you can press ⌘**/**

If you run the application again you'll notice that console doesn't say **`"Hello iPhone!"`**. This is because our program didn't *do* anything. The code that we just commented out is called a **log statement**. Log statements are also useful for debugging applications.

Let's change the text that we're printing. Un-comment the code by removing the slashes and change the text to read **`NSLog(@"Hello France!");`**

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
-     //NSLog(@"Hello iPhone!");
+     NSLog(@"Hello France!");
    return YES;
}
```

Run the app and the output should read **`Hello France!`**.

Strings

Let's look at the part of the log statement inside the parentheses -- the part that reads **`@"Hello France!"`**. In programing this is called a **string**. A string is a data type used for storing text. In Objective-C strings are preceded with an **`@`** symbol followed by the text information surrounded in double quotes.

Let's see if we can type another log statement on a new line to print your name.

AppDelegate.m

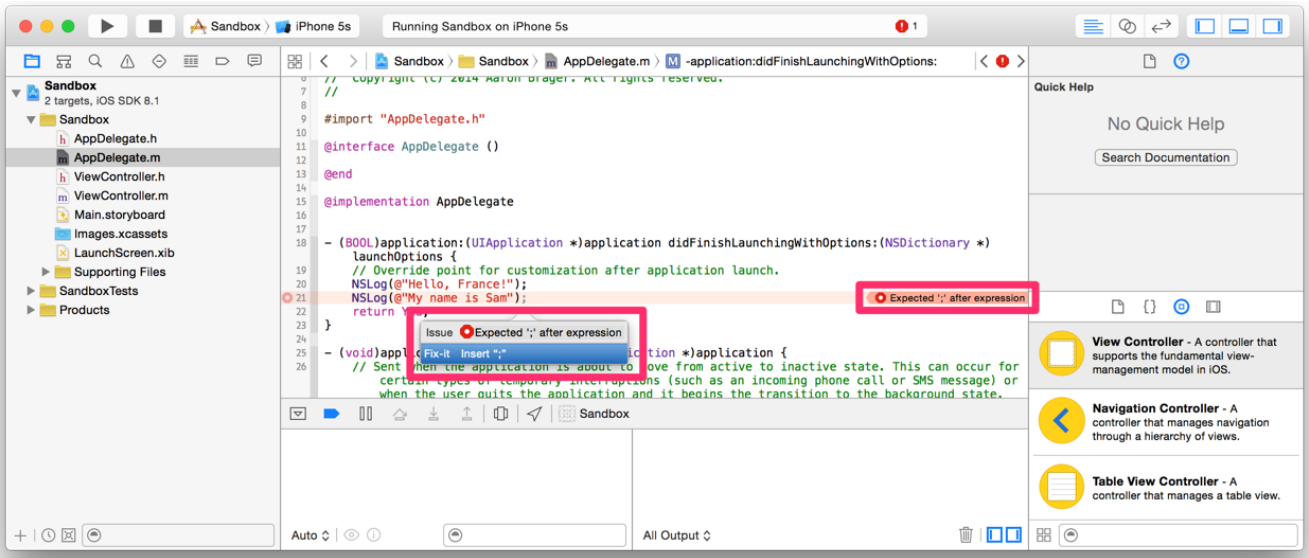
```
// Override point for customization after application launch.
NSLog(@"Hello, France!");
+ NSLog(@"My name is Sam");
```

Objective-C statements must end in semicolons **`;`**. Semicolons tell the compiler that we are at the end of one statement and ready to start another.

If we remove the semicolon Xcode will give us a helpful warning, telling us that our code is invalid:

AppDelegate.m

```
// Override point for customization after application launch.
NSLog(@"Hello, France!");
- NSLog(@"My name is Sam");
+ NSLog(@"My name is Sam")
```



Methods

Let's print something longer:

AppDelegate.m

```
// Override point for customization after application launch.
NSLog(@"Hello, France!");
NSLog(@"My name is Sam");
+ //Chant
+ NSLog(@"Lions and Tigers and Bears, Oh My!");
+ NSLog(@"Lions and Tigers and Bears, Oh My!");
+ NSLog(@"Lions and Tigers and Bears, Oh My!");
+ NSLog(@"Lions and Tigers and Bears, Oh My!");
```

That's a lot to type. It would be nice if we could just define our message once and print it any number of times we wanted. For that we can us a **Method**

Up to this point, you've written code as individual and standalone lines of syntax. If you wanted to reuse a line of code in this respect, you'd have to type it in again.

Methods allow you to write code, and reuse it later by calling the same method again. Take note of two common terms:

- **Call** - run or execute a method.
- **Define** - compose or write a method.

Methods have a structure that must be used to define them. Write your first by adding this code to the end of the file but above the line that reads **@end**:

AppDelegate.m

```
{
    // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.
}

+ - (void) chant
+ {
+    //Method code here
+    NSLog(@"Lions and Tigers and Bears, Oh My!");
+ }
@end
```

The method definition begins with a **-** symbol followed by a return type **(void)**. Some methods return data when you call them. We do not want this method to return data, so we use **void** -- the method will return *nothing*. We'll cover other return types later.

The next word - **chant** - is the name of the method. This is what we will use to call the method later. Next we have an open curly brace **{** some code, and a closing curly brace **}**. These curly braces mark the **body** of the method. The code between the curly braces is what gets executed when the method is called.

Once a method has been defined we use it by **calling** it:

```
[self chant];
```

Now let's update our code to use the new method:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    NSLog(@"Hello France!");
    NSLog(@"My name is Sam");

-     NSLog(@"Lions and Tigers and Bears, Oh My!");
-     NSLog(@"Lions and Tigers and Bears, Oh My!");
-     NSLog(@"Lions and Tigers and Bears, Oh My!");
-     NSLog(@"Lions and Tigers and Bears, Oh My!");
+     [self chant];
+     [self chant];
+     [self chant];
+     [self chant];

    return YES;
}
```

Here we are **calling** the method 4 times. When we run the app again it should print **"Lions and Tigers and Bears, Oh My!"** 4 times.

Roadmap · **Previous Checkpoint** **Jump to Submission** · **Next Checkpoir**

Your assignment | **Ask a question** | **Submit your work**

Log some more strings with **NSLog**. Make sure you are comfortable with Xcode, in the context of the steps you've completed thus far.

As you go through the exercises, be sure to take notes. You can use **Gists** to take notes and share them with your mentor. Gists are a great place to take notes when you are programming because they allow you to easily format, search and share information with others. You can also keep them private if you prefer.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Unit Tests

Unit tests ensure that the code we write is working as we expect. Just like Internal Affairs is filled with police officers that investigate other police officers, unit tests are made of code that tests other code.

Why would you need to test code? When an application grows in size and complexity it becomes increasingly likely that code you add to one part of the app will break something in another part. We write automated tests to alert us if a change has broken something.

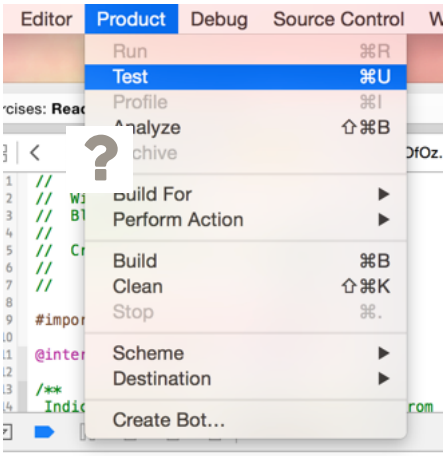
We use automated tests for our exercises in this course. We'll provide you with unit tests and you will write the code to make those tests pass

In this checkpoint we'll cover the following:

- navigating the `ios-exercises` **repo**
- running tests
- reading test results
- learning **`setUp`**, **`test____`**, and **`tearDown`** methods

Step one is to open the exercises project that we cloned in the Git and GitHub checkpoint. You should find it in the **`ios-exercises/BlocExercises`** directory.

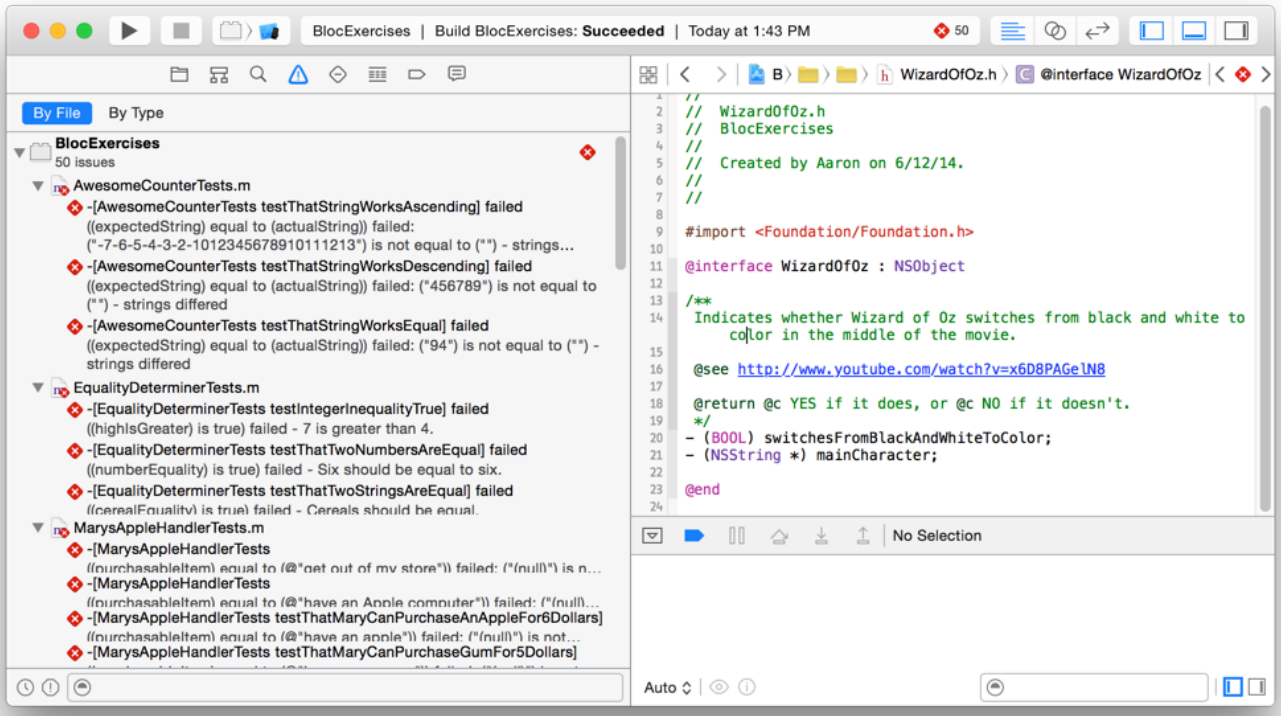
To run the tests click: **Product** > **Test**. You'll run this command a lot so it's best to learn its keyboard shortcut: ⌘**U**. (command "U")



This causes three things to happen.

1. Xcode builds the project
2. It opens the iOS simulator
3. It runs the tests in the simulator and displays the output

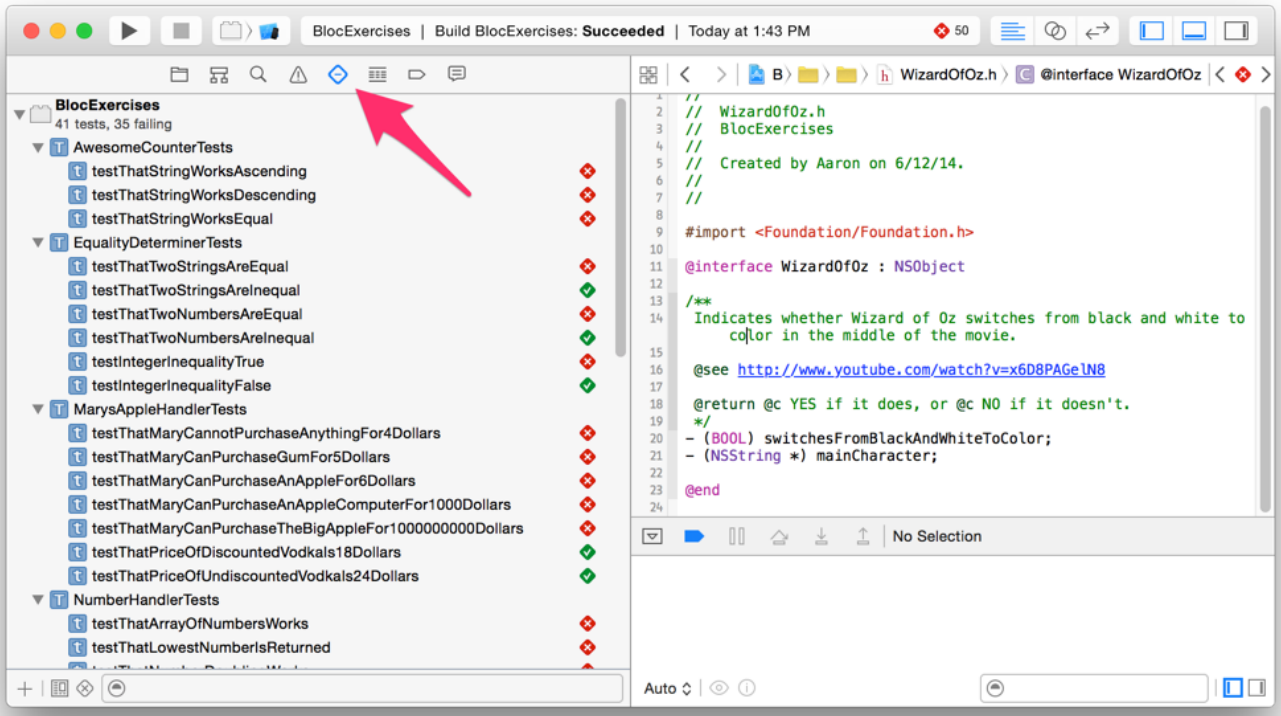
Here's how the failing tests look in the Issue Navigator:



There are two main places to see a list of test results:

- All tests, and their pass/fail status, are shown in the Xcode **Test Navigator**.
- Only the failing tests, along with errors and warnings, in the **Issue Navigator**.

We'll get to errors and warnings later. For now we'll focus on tests, so let's select the panel to the right of this one named **Test Navigator**.



list of green checkmarks can be very therapeutic to seasoned programmers!

Let's look at one of these tests. Click on the first item under **`WizardOfOzTests`**, which should be at the bottom of the navigator window.

There's a lot of code in this file but focus on the section that looks like this:

WizardOfOzTests.m

```
- (void)setUp
{
    [super setUp];
    // Put setup code here. This method is called before the invocation of each test method in the class.

    self.wonderfulWizard = [[WizardOfOz alloc] init];
}
```

This method is named **`setUp`**. The part inside the curly braces **`{...}`** is called the method's **body**. The body is the part that does work when th method is called.

Every time we run the tests. Xcode reads every test file in the project and runs every method inside them. There are a total of four methods ir this file.

Every test file in this project has a **`setUp`** and **`tearDown`** method. These methods are special in that **`setUp`** will always run first and **`tearDown`** wi always run last.

`testThatColorfulnessOfFilmIsAccuratelyPortrayed` is the actual test. Methods like this one correspond to the items in the **Test Navigator**. They contain the logic that will test our code in future exercises. Notice that there are red X's to the left of the code.

```
- (void)testThatColorfulnessOfFilmIsAccuratelyPortrayed
{
    BOOL colorSwitches = [self.wonderfulWizard switchesFromBlackAndWhiteToColor];
    XCTAssertTrue(colorSwitches == YES, @"The color switches from black and white to color, but the
        `switchesFromBlackAndWhiteToColor` method indicated that it doesn't.");
}

@end
```

This is another indicator that this test method is failing.

If we hover our mouse cursor over this red X it will turn into a play symbol. Clicking it will cause Xcode to only run this single test.

This will be useful when we want to test a single method.

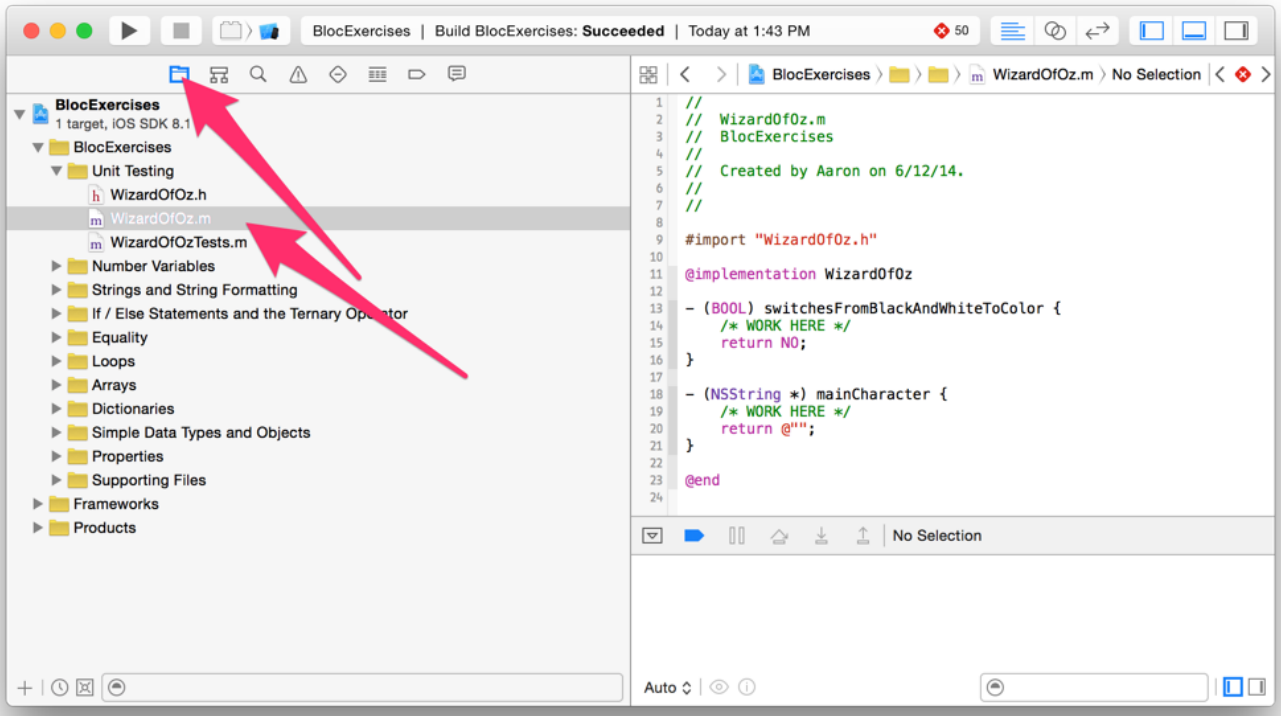
In summary, the test file works like this:

1. **`setUp`** always runs first. It prepares the code to be tested
2. **`testThatColorfulnessOfFilmIsAccuratelyPortrayed`** runs and reports the results
3. **`testTheMainCharacterIsCorrect`** runs and reports the results
4. **`tearDown`** always runs last. This is where we clean up after ourselves
5. Move onto the next test file, and repeat

Making the Test Pass

Look at the end of **`testThatColorfulnessOfFilmIsAccuratelyPortrayed`** and notice the code reading **`XCTAssertTrue`**. This is called an assertion. This one is failing because it is expecting the method **`switchesFromBlackAndWhiteToColor`** to return a **`YES`** value. When this assertion fails it prints the string: **`@"The color switches from black and white to color, but the switchesFromBlackAndWhiteToColo method indicated that it doesn't."`** as an error.

Switch back to the Project Navigator, open up the Unit Testing folder, and select **`WizardOfOz.m`**.

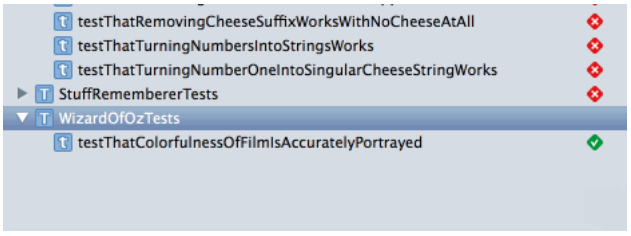


We can make this test pass by changing the method in **`WizardOfOz.m`**:

WizardOfOz.m

```
#import "WizardOfOz.h"

@implementation WizardOfOz

 - (BOOL) switchesFromBlackAndWhiteToColor {
    /* WORK HERE */
-     return NO;
+     return YES;
 }
```

After making that change click the arrow next to the **`testThatColorfulnessOfFilmIsAccuratelyPortrayed`** inside the test navigator.



The test should now be passing.

How Unit Test Assignments Work

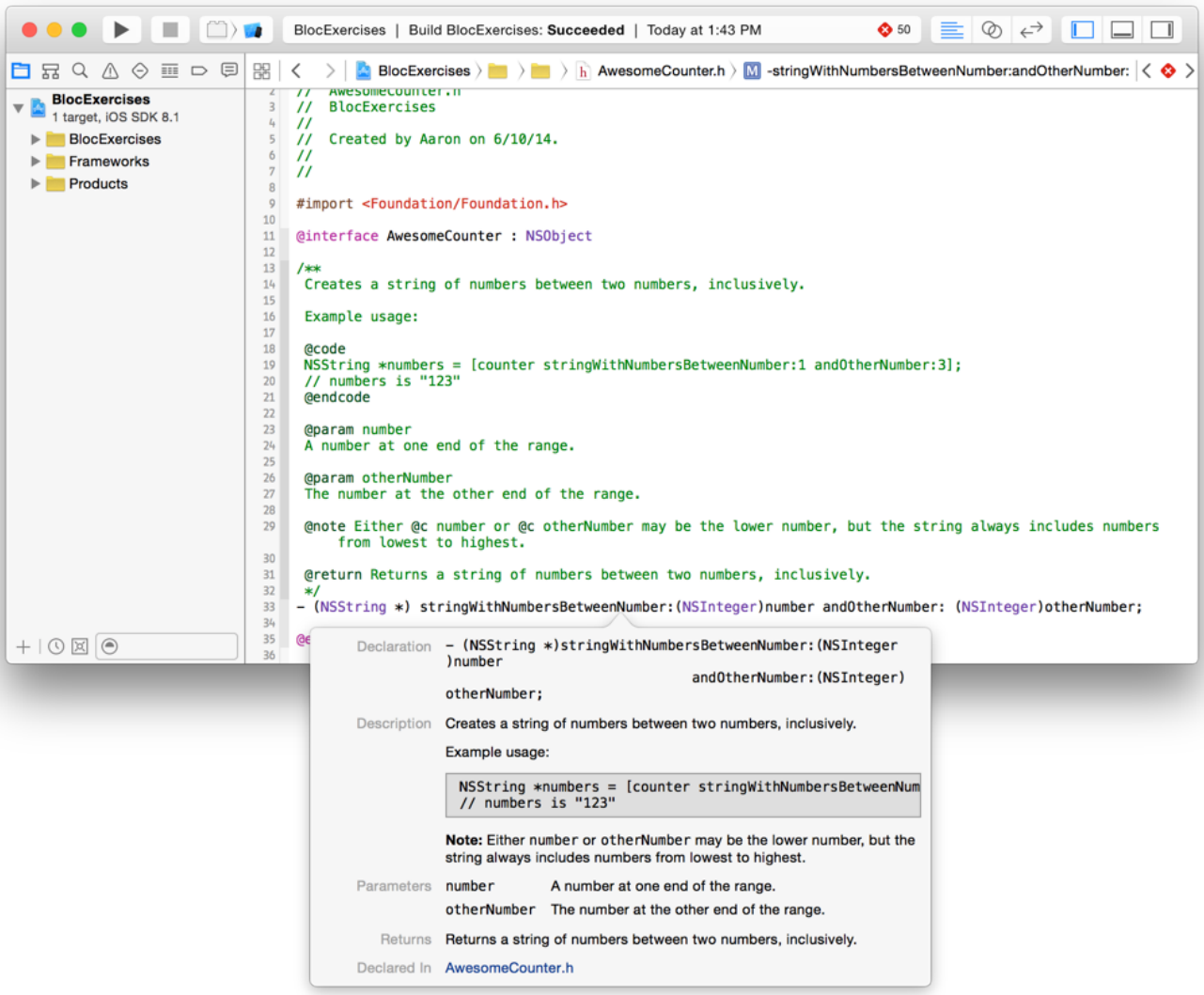
As you proceed through the next group of checkpoints, each checkpoint's assignment will use this unit test library. (Each checkpoint has a separate folder of tests.)

When you open Xcode to complete your assignments, you should read both the documentation in the **`.h`** file, as well as the tests that are running each method.

to **`return`** the expected result at the end of the method (just like we wrote **`return YES;`** above.)

The header files contain notation like **`@code`**, **`@note`**, **`@c`**, etc. within the comments. These are all style markers. This special format allows you t
option-click on the method name and see the documentation with the formatting rendered.

Here's an example of an assignment in a future checkpoint:



In this example, your method will be passed two numbers (for example, **1** and **3**), and you're expected to write code that returns **`"123"`**.

You should generally try to complete assignments on your own, but when you get stuck you can send a Message to your mentor. Feel free to also use other online resources like the Apple Developer Forum and Stack Overflow.

Roadmap · **Previous Checkpoint** **Jump to Submission** · **Next Checkpoi**

Your assignment	Ask a question	Submit your work

Before you commit and push your changes, check to make sure you're on the master branch. In the third checkpoint, *Git and GitHub*, you created and checked out the branch, 'my-first-branch'. If you never checked out master again, you should do that now.

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git branch
  master
* my-first-branch
```

If you're still on 'my-first-branch', merge it into `master`.

/Users/user-name/work/ios-exercises

```
$ git checkout master
$ git merge my-first-branch
```

You should see something like this generated as a result:

/Users/user-name/work/ios-exercises

```
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

If you get a merge conflict (you shouldn't have):

/Users/user-name/work/ios-exercises

```
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

You'll need to resolve it. To learn about resolving conflicts, see **Git Basics: Branching and Merging** under the "Basic Merging Conflicts" section, or send a note to your mentor.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed unit tests"
$ git push origin master #=> send your changes to your remote GitHub repo
```

Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and do the following:

- Make the second test `WizardOfOzTests.m` pass by modifying the `mainCharacter` method in `WizardOfOz.m`

Exercise descriptions are found in the header (`.h`) file, but you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

When you've completed these exercises, commit, push and submit this assignment with the relevant repo and commit links.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Numbers and Variables

The ability to store temporary data is expected of most programming languages and Objective-C is no exception. When writing software you'l find that maintaining data types like numbers, strings, and booleans will become vital to your application's functionality. Applications persist their data in the form of **variables**. Variables are **mutable** placeholders for digital information.

This checkpoint covers the following topics:

- Primitive C data types
- Objective-C data types
- Mathematical operations

C Data Types

Since Objective-C is a superset of C, it has access to all of the data types present in the C language. Let's start with the easiest, **`int`**.

Integer

`int` represents an integer. Its range is limited to [-2147483648, 2147483647] due to the fact that it's represented by 32 bits. In computing, all data are represented by binary digits. If you're interested in learning more about binary, check out this **handy guide**.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application di( ? LaunchingWithOptions:(NSDictionary *)launchOptions
{
  int theMeaningOfLife = 42;
  return YES;
}
```

`theMeaningOfLife` is the name of your very first variable. All variables must have a name. Furthermore, all variables must also have a type. **`theMeaningOfLife`**'s type is **`int`**. Variables may optionally be assigned a value, in this case, **`42`**. We've established a new variable and assigned it a value with the **`=`** sign.

Integers must be assigned whole numbers. They cannot be fractions or anything other than whole numbers.

Long

A **`long`** can be treated exactly the same as an **`int`**. However, a **`long`** has 64-bit representation. Therefore, a **`long`**'s range is [-9223372036854775808, 9223372036854775807]. Let's see an example of how a **`long`** and an **`int`** differ:

AppDelegate.m

```
{
  int intMaxIntSizePlusOne = 2147483648; // 2147483647 + 1
  long longMaxIntSizePlusOne = 2147483648; // 2147483647 + 1
  NSLog(@"Overflowing int: %d", intMaxIntSizePlusOne);
  NSLog(@"Accurate long: %ld", longMaxIntSizePlusOne);
  return YES;
}
```

Overflow occurs when a variable breaches its maximum or minimum representation. At the point of overflow, the variable goes from one extreme to the other. 2147483647 + 1 becomes -2147483648 inside of an **int**. Here's what was printed to the console:

```
2014-06-11 11:57:24.300 Test[7766:303] Overflowing int: -2147483648
2014-06-11 11:57:24.303 Test[7766:303] Accurate long: 2147483648
```

As you can see, the **long** remained true to the original value we assigned it since the whole number value was comfortably within its 64-bit range. Whereas **int**, having breached its maximum capacity overflowed and became **-2147483648**.

Float

A **float** is a fractional value represented by 32 bits. Due to memory limitations, floats are rarely 100% accurate. While they may be capable of representing 1.5 and many other finitely limited decimal places, a fraction such as 1/3 is impossible to represent accurately. This is due to the fact that the decimal place is infinite, e.g. 0.3333333–repeating or pi, 3.14159265359... Floats are accurate up to six digits:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  float accurateFloat = 1.5;
  float inaccurateFloat = 567.8976523;
  NSLog(@"Accurate Float: %f", accurateFloat);
  NSLog(@"Inaccurate Float: %f", inaccurateFloat);
  return YES;
}
```

What's printed to the command line is this:

```
2014-06-11 11:14:34.637 Test[7586:303] Accurate Float: 1.500000
2014-06-11 11:14:34.639 Test[7586:303] Inaccurate Float: 567.897644
```

As you can see, the float we assigned in the source code was not the one stored in our float variable. The first 6 digits were accurate, the remaining digits were misrepresented. This results due to a limitation in the **IEEE 754 standard**.

Because of these inaccuracies, you should never use a **float** to store any data which needs to be stored precisely, such as an amount of money.

Double

A **double** is identical to a float in nearly every way except it is allotted 64 bits for its representation; *double* the size of float's capacity. Doubles are more accurate than floats with a total of 15 digits guaranteed:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  double accurateDouble = 1.5;
  double inaccurateDouble = 567.8976523;
  NSLog(@"Accurate Double: %.21g", accurateDouble);
  NSLog(@"Inaccurate Double: %.21g", inaccurateDouble);
  return YES;
```

What's printed to the command line is this:

```
2014-06-11 11:27:17.347 Test[7652:303] Accurate Double: 1.5
2014-06-11 11:27:17.348 Test[7652:303] Inaccurate Double: 567.897652300000004288
```

As you can see, the double is substantially closer to the original value we assigned it– **567.8976523**–than the float was.

Objective-C Data Types

The issue with **int**, **float** and **double** is that they are defined with respect to the size of their representation. An **int** or **float** will always require 32 bits while a **double** will require 64 bits. However, 32-bit architectures like those running on older versions of the iPhone, iPod, iPad MacBooks, etc. are not capable of employing 64 bit representations of anything.

When creating new software today, best practice dictates using architecture-agnostic data types which will allow your code to execute properly on older hardware. Objective-C provides several data types which help alleviate this problem.

NSInteger

NSInteger can be treated *just* like an integer. However, unlike a regular **int** or **long**, it is architecture-agnostic. When compiling for 32 bit systems, **NSInteger** is just another name for **int** and when compiling for 64-bit hardware it represents **long**. Bloc recommends the use of **NSInteger** over **long** or **int** unless you're absolutely certain of the range that your application requires.

CGFloat

CGFloat's behavior is analogous to **NSInteger**'s. It operates under the exact same conditions except it swaps between **float** for 32-bit systems and **double** when compiling for 64-bit systems. Again, Bloc recommends the use of **CGFloat** over a strict **float** or **double** because it ensures the highest accuracy available to you on the target architecture.

Math

> *Tum, tum tuuuummm!*

You came here to learn about variables and math and we're all out of variables. Math is crucial to programming in every sense. Thankfully, simple math operations are the most commonly required. Operations like addition, subtraction, multiplication and division are incredibly commonplace in programming.

Add and Subtract

Here are some examples of addition and subtraction:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  NSInteger one = 1;
  NSInteger two = one + 1;
  NSInteger negativeOne = two - 3;
  NSInteger zero = one + negativeOne;
  NSLog(@"one is: %ld", (long)one);
  NSLog(@"two is: %ld", (long)two);
  NSLog(@"negativeOne is: %ld", (long)negativeOne);
  NSLog(@"zero is: %ld", (long)zero);
  return YES;
}
```

Addition and subtraction work just as you imagine them to on a calculator. Here's the output from our code above:

```
2014-06-11 13:46:22.023 Test[8050:303] one is: 1
2014-06-11 13:46:22.025 Test[8050:303] two is: 2
2014-06-11 13:46:22.026 Test[8050:303] negativeOne is: -1
2014-06-11 13:46:22.027 Test[8050:303] zero is: 0
```

Increment and Decrement

The increment and decrement operations are quick shortcuts for adding or subtracting **1** from any number:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  NSInteger one = 0;
  one++; // Post-increment
  NSLog(@"One is now: %ld", (long)one);
  NSLog(@"A pre-increment makes one: %ld", (long)++one);
  NSLog(@"A post-increment still prints: %ld", (long)one++);
  NSLog(@"But changes one to: %ld", (long)one);
  return YES;
}
```

The sample above produces the following output:

```
2014-06-11 14:44:52.176 Test[8329:303] One is now: 1
2014-06-11 14:44:52.178 Test[8329:303] A pre-increment makes one: 2
2014-06-11 14:44:52.179 Test[8329:303] A post-increment still prints: 2
2014-06-11 14:44:52.179 Test[8329:303] But changes one to: 3
```

A pre-increment (e.g. **++variableName**) will increase the value of **variableName** by **1** before evaluating the rest of the statement. A post-increment (e.g. **variableName++**) increments **variableName** by **1** after it's value has been used. In the example above we had the following line:

```
NSLog(@"A post-increment still prints: %ld", one++);
```

When printing **one** it used **one**'s current value of **2** but after the printing was completed, **one** was then incremented.

When we invoked:

```
NSLog(@"A pre-increment makes one: %ld", ++one);
```

one was incremented *before* it was printed. Its value became **2** and was printed as such.

Multiply and Divide

You can multiply and divide just as easily as you add and subtract. In terms of performance, division operations are the costliest but don't let that deter you from calculating pie-chart slices.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  NSInteger twelve = 12;
  NSInteger three = 3;
  NSLog(@"12 / 3 = %d", twelve / three);

  CGFloat oneThird = 1.0 / 3.0;
  NSLog(@"1 / 3 = %.20f", oneThird);

  NSInteger five = 5;
  NSInteger six = 6;
  NSInteger thirty = five * six;
  NSLog(@"5 x 6 = %d", thirty);
  return YES;
}
```

Which prints the following to the log:

```
2014-06-11 14:52:56.520 Test[8394:303] 12 / 3 = 4
2014-06-11 14:52:56.522 Test[8394:303] 1 / 3 = 0.33333333333333331483
2014-06-11 14:52:56.523 Test[8394:303] 5 x 6 = 30
```

You can mix and match types when multiplying or dividing, but a variety of factors will determine the final outcome. If you don't care for the remainder, divide into an **`NSInteger`** type and the remainder will be lopped off and rounded down to a whole number. However, when decimals and accuracy are important to your code, stick to dividing with **`CGFloat`** values.

Mod

The modulo operator – **`%`** – returns the integer remainder of a quotient. For instance, when dividing **`5`** into **`30`**, the remainder is **`0`** because **`5`** divides evenly into **`30`** six times. Division does not always work out evenly though. When dividing **`8`** into **`30`** the outcome is **`3.75`**. The remainde of this division is calculated by multiplying **`.75`** by **`8`** which results in an integer remainder of **`6`**.

AppDelegate.m

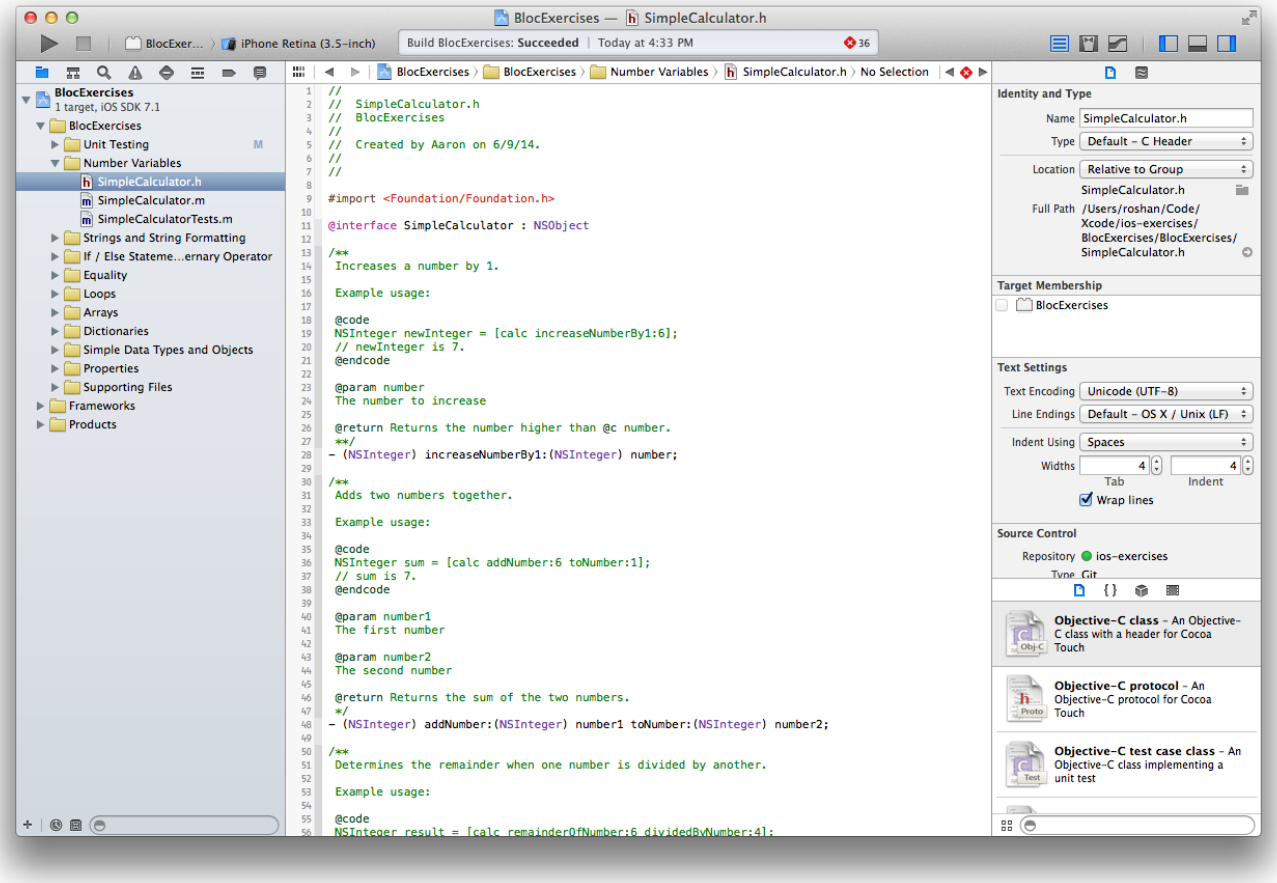
```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  NSInteger thirty = 30;
  NSInteger five = 5;
  // the % symbol is used for modulus
  NSLog(@"Remainder of 30 / 5 = %d", thirty % five);

  NSInteger eight = 8;
  NSLog(@"Remainder of 30 / 8 = %d", thirty % eight);
  return YES;
}
```

The output of this sample mod code is:

```
2014-06-11 15:01:10.942 Test[8437:303] Remainder of 30 / 5 = 0
2014-06-11 15:01:10.944 Test[8437:303] Remainder of 30 / 8 = 6
```

Learn more about **modulo here**.

Open the BlocExercises project in Xcode. Open the **Number Variables** folder and implement solutions to make the tests pass.



Exercise descriptions are found the the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed numbers and variables"
$ git push origin master #=> send your changes to your remote GitHub repo
```

When you've completed the corresponding exercises, submit this assignment with the relevant repo and commit links.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Strings

Strings

The computing world relies heavily on human-readable information. Everything from the text you see on screen to the error messages handled internally by an application is typically represented by a natural language. Strings are used to store and manipulate natural language characters.

NSString

When working with strings in Objective-C, use the **`NSString`** class. You can think of **`NSString`** as just another variable type. Let's see it in actior

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *wGP = @"\"with great power, comes great responsibility.\"";
    NSLog(@"Uncle Ben always said, %@", wGP);
    return YES;
}
```

> *For a variable type such as* **`NSString`** *and other c ? ojects, an* ***** *is required in front of their name. You will learn why in a later checkpoint.*

This code sample prints the following to the console:

```
2014-06-12 09:37:20.201 Test[796:303] Uncle Ben always said, "with great power, comes great responsibility."
```

String Literals

`wGP` is your very first **`NSString`**. A string can be assigned using a **string literal.** A string literal is a natural language string found inside of source code. When presenting text to users, it is *not* recommended to do so with string literals as these are not localized to their region.

For example, the phrase "Hello, world!" is perfectly acceptable to English-speaking locales yet likely unacceptable in the Latin American, Asian and other regions where English may not be the most commonly spoken language.

> *You will learn more about string localization later, so we'll stick with string literals for now.*

A string literal requires special syntax in Objective-C. Each literal begins with the **`@`** symbol and is enclosed by double quotes, **`" "`**. Let's look at our example **`wGP`** again, **`@"\"with great power, comes great responsibility.\""`**. The actual *content* of the string is **`\"with great power comes great responsibility.\"`**

our string's content, we prefix these special characters with a backslash (****). This is required so that the final output of our string to the command prompt has properly placed quotes in it:

```
2014-06-12 09:37:20.201 Test[796:303] Uncle Ben always said, "with great power, comes great responsibility."
```

Formatting

NSString has methods which help developers manipulate and present literal information. You've encountered one before, even in this very checkpoint. Let's look at **NSLog** again from the code sample we displayed above:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *wGP = @"\"with great power, comes great responsibility.\"";
    NSLog(@"Uncle Ben always said, %@", wGP);
    return YES;
}
```

The line, **NSLog(@"Uncle Ben always said, %@", wGP);** is generating a formatted string and outputting it to the console. **NSLog** was passed two arguments. The first is always an **NSString**. In this invocation we've used the string literal **@"Uncle Ben always said, %@"**. The **%@** is a special syntax called a **format specifier**.

Format Specifiers

As you may have noticed in the **Numbers and Variables** checkpoint, we can insert data into strings using the **%** sign. The **%** is called a **string format specifier.** It marks the subsequent characters as a placeholder for data to be inserted. When a format specifier is included, additiona arguments are required in order to specify the information to include for each specifier.

In the sample code above, **wGP** is the additional argument. There are a variety of format specifiers. The one used above, **%@**, means that an object like **NSString** must be placed in that location. Here's a recap of some other commonly used specifiers:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger five = 5;
    NSInteger three = 3;
    NSInteger two = 2;
    NSString *apples = @"apples";
    NSLog(@"If I start with %ld %@ and eat %ld of them, I have %ld left!", (long)five, apples, (long)three, (long)two);
    return YES;
}
```

The code sample above prints the following to the command prompt:

```
2014-06-12 10:25:47.586 Test[930:303] If I start with 5 apples and eat 3 of them, I have 2 left!
```

%ld is used to specify that a **long** value will be formatted into the string at that location. Once again we've employed the **%@** specifier in order to insert **apples** into our final string. Notice that the arguments must match the order of the specifiers. From left-to-right we first see a **%ld** then a **%@** followed by two more **%ld**s. Therefore, our formatting arguments are presented in the same order: a number, a string, a number and another number.

There are many more specifiers to choose from. When displaying floats or doubles you may use **%f**. There's a bevy of these and you can see the **full list here**.

stringWithFormat:

Printing to the log is great for debugging, but what if you want to present that little nugget of apple wisdom to the user? We can save formatted strings to a new **NSString** variable by employing **[NSString +stringWithFormat:]**. Here's the same sample code re-written to support storing the formatted string:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger five = 5;
    NSInteger three = 3;
    NSInteger two = 2;
    NSString *apples = @"apples";
    NSString *appleString = [NSString stringWithFormat:@"If I start with %ld %@ and eat %ld of them, I have %ld left!", (long)five, ap
    NSLog(appleString);
    return YES;
}
```

This code sample accomplishes the exact same thing as our sample above, yet now we've stored the result of our formatted string into a new **NSString**. This string, **appleString**, can be displayed to the user on a label, in an alert dialog, or in some other meaningful way.

You can imagine how this would be useful in a game where a piece of text must communicate a user's score. "You scored 185 points!" would come from a formatted string like **@"You scored %ld points!"**.

NSRange

Developers often need to adjust, replace, or remove portions of an **NSString**. **NSRange** describes a range of characters within an **NSString** by providing a starting location and a length. In the string **@"Pluto is a planet in our solar system."** we have a potentially controversial statement. To clear things up, let's change the word **planet** to something different:

When you know the string literal ahead of time, it's easy to determine the range of characters you'd like to alter. In the case of the word **planet**, we can declare its range:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *plutoStatement = @"Pluto is a planet in our solar system.";
    NSRange planetRange = NSMakeRange(11, 6);
    return YES;
}
```

NSMakeRange(location, length) helps us easily create an **NSRange** variable. The location of **planet** is **11** because the **p** character begins at the 11th index of our phrase. Indexes are 0-based and therefore the letter **P** is at index **0**, the letter **l** is at index **1**, the letter **u** is at index **2**,

> *If the string isn't found, you'll get a range where* **`location`** *is equal to* **`NSNotFound`***. This is explained in more detail later in this checkpoint.*

rangeOfString:

Instead of counting indexes ourselves we can use **`[NSString -rangeOfString:]`** to get an **`NSRange`** dynamically. Often times a developer may not know what string they're working with since it comes from a place not found in the source code. (Strings often come from an API service in the cloud or a localization file, and these strings may not be known beforehand by the developer.)

To recover an **`NSRange`** from any string, literal or otherwise, we can use **`rangeOfString`**:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *plutoStatement = @"Pluto is a planet in our solar system.";
    NSRange planetRange = [plutoStatement rangeOfString:@"planet"];
    NSLog(@"planetRange starting point: %lu", (unsigned long) planetRange.location);
    NSLog(@"planetRange length: %lu", (unsigned long) planetRange.length);
    return YES;
}
```

Which prints the following predictable information to the command prompt:

```
2014-06-12 11:04:37.456 Test[1075:303] planetRange starting point: 11
2014-06-12 11:04:37.458 Test[1075:303] planetRange length: 6
```

NSStringCompareOptions

Optionally you may pass comparison options to a similar method, **`[NSString -rangeOfString:options:]`**. These help specify the types of matches you're willing to accept. For instance we may use the **`NSCaseInsensitiveSearch`** option which allows us to search for **`PLANET`**, **`plAnet`**, **`planET`**, etc. and still recover the same range of characters:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *plutoStatement = @"Pluto is a planet in our solar system.";
    NSRange planetRange = [plutoStatement rangeOfString:@"planET" options:NSCaseInsensitiveSearch];
    NSLog(@"Planet Range is: %@", NSStringFromRange(planetRange));
    return YES;
}
```

Despite the fact that **`planET`** is not identical to the form of **`planet`** found within our original string literal, it will still match the range due to the fact that we specified the **`NSCaseInsensitiveSearch`** option:

```
2014-06-12 11:07:05.785 Test[1108:303] Planet Range is: {11, 6}
```

There are other options, with different effects, which you can **see in the `NSString` documentation**.

stringByReplacingCharactersInRange:withString:

Now that we understand **`NSRange`** and how to find one from within an existing string, let's use a new method to *replace* characters found within a range. Remember that our original statement, **`Pluto is a planet in our solar system.`** was a little controversial and we want to improve it. Let's use **`[NSString -stringByReplacingCharactersInRange:withString:]`** to accomplish that:

```
{
    NSString *plutoStatement = @"Pluto is a planet in our solar system.";
    NSRange planetRange = [plutoStatement rangeOfString:@"planet"];
    NSString *technicallyCorrectStatement = [plutoStatement stringByReplacingCharactersInRange:planetRange withString:@"large object i
    NSLog(@"%@", technicallyCorrectStatement);
    return YES;
}
```

When we print **`technicallyCorrectStatement`** to the console, we see the following output:

```
2014-06-12 11:44:33.037 Test[1210:303] Pluto is a large object in the Kuiper belt in our solar system.
```

And we've appeased the scientific community.

NSNotFound

When recovering an **`NSRange`** object from an **`NSString`**, it's possible to recover an *invalid* range. Consider the case where you might look for a string literal within an **`NSString`** that **does not** contain it. In one of my favorite Beatles classics, John Lennon reminds us that all I need is, in fact, love:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *beatlesLyric = @"All you need is…";
    NSRange loveRange = [beatlesLyric rangeOfString:@"love"];
    NSLog(@"Location of 'love': %ld", loveRange.location);
    NSLog(@"NSNotFound: %ld", NSNotFound);
    return YES;
}
```

Unfortunately, the script above prints the following horrific number:

```
2014-06-12 11:05:23.241 Test[1178:33137] Location of 'love': 9223372036854775807
2014-06-12 11:05:23.242 Test[1178:33137] NSNotFound: 9223372036854775807
```

"love"'s location was not found in the string, therefore the range's location was set to **`NSNotFound`**. In later checkpoints, you may use this value to determine whether or not an **`NSRange`** is valid before using it.

Mutable Strings

`NSString` is immutable. That means that once an **`NSString`** has been declared and assigned, *its contents can no longer be altered.* This is why **`[NSString -stringByReplacingCharactersInRange:withString:]`** had its result placed into a *new* **`NSString`**. All of the **`NSString`**s we've used have remained exactly as they were declared. In the example above, after we've called **`stringByReplacingCharactersInRange:withString:`**, **`plutoStatement`** is *still* equal to **`"Pluto is a planet in our solar system."`**

However, there's an alternative type which allows us to modify our string instead of having to create a new one each time: **`NSMutableString`**. It's slightly more complicated to declare an **`NSMutableString`** than an **`NSString`** but as a reward, you have access to direct modifications:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableString *modifiableStatement = [@"Pluto is a planet in our solar system." mutableCopy];
    NSRange planetRange = [modifiableStatement rangeOfString:@"planet"];
    [modifiableStatement replaceCharactersInRange:planetRange withString:@"large object in the Kuiper belt"];
    NSLog(@"%@", modifiableStatement);
    return YES;
}
```

Mutable strings have several other methods available to them that **`NSString`** is not permitted, such as **append**:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableString *modifiableStatement = [NSMutableString stringWithString:@"Pluto is a planet in our solar system."];
    [modifiableStatement appendString:@" It *used* to be considered a major planet."];
    NSLog(@"%@", modifiableStatement);
    return YES;
}
```

The sample above prints the following to the command prompt:

```
2014-06-12 11:57:24.351 Test[1324:303] Pluto is a planet in our solar system. It *used* to be considered a major planet.
```


Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and implement solutions to make the tests pass. Exercise descriptions are found the the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed strings"
$ git push origin master #=> send your changes to your remote GitHub repo
```

When you've completed the corresponding exercises, submit this assignment with the relevant repo and commit links.

assignment completed

COPYRIGHT © 2015 BLOC

Equality

Equality

Comparing variables is an important part of programming. Often we'd like to know whether one thing is identical to another before proceeding. Imagine a user accidentally posts two updates to the number of calories they've consumed during the day. First we check if the calorie count is identical to the previous one before performing a costly network request to update the cloud service where it's stored. Why bother if they're already equal?

Equals

When comparing two variables in Objective-C, it's best to stick with one of three choices. An equals comparison operator returns a **BOOL** which either indicates that **YES** these two are equal or **NO** they are not.

==

== is the simplest way to determine equality. For primitive data types like **int**, **long**, **float**, **NSInteger**, etc., **==** is sufficient in determining whether or not two values are identical:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application di  LaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger twentyFive = 25;

    // Compare twentyFive and 25
    BOOL areTheyEqual = twentyFive == 25;

    NSLog(@"Are they equal? %@", areTheyEqual ? @"Yes" : @"No");
    return YES;
}
```

Don't worry about the code found in the **NSLog(…);** statement for now. You'll learn about the ternary operator and conditionals in an upcoming checkpoint. The sample code above prints **Yes** if **areTheyEqual** is **YES** and **No** if it's **NO**. The output to the command prompt is as follows:

```
2014-06-12 13:58:08.647 Test[2167:303] Are they equal? Yes
```

This same example can work with **CGFloat**, **int**, **long**, etc:

AppDelegate.m

```
{
    CGFloat oneThird = 1.0/3.0;
    CGFloat alsoOneThird = 1.0/3.0;
    BOOL areTheyEqual = oneThird == alsoOneThird;
    NSLog(@"Are they equal? %@", areTheyEqual ? @"Yes" : @"No");
    return YES;
}
```

Which in turn, also prints **`2014-06-12 14:02:49.540 Test[2200:303] Are they equal? Yes`** to the console log. **`==`** is also good for comparing whether or not two objects are *identical*. Two objects are identical if and only if they point to the exact same data in memory. This check can be performed by the **`==`** operator:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString* stringOne = @"Hello, there!";
    NSString* stringTwo = stringOne;
    BOOL areTheyEqual = stringOne == stringTwo;
    NSLog(@"Are they identical? %@", areTheyEqual ? @"Yes" : @"No");
    return YES;
}
```

Yes, those two strings are *identical* because they literally point to the same piece of data in memory. When we made the assignment **`stringTwo = stringOne;`** nothing new was created other than the variable name **`stringTwo`** which is a reference to the same data found in **`stringOne`**. However, let's look at a different example using the **`NSNumber`** class:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSNumber* fiveA = [NSNumber numberWithFloat:5.0];
    NSNumber* fiveB = [NSNumber numberWithFloat:5.0];
    BOOL areTheyEqual = fiveA == fiveB;
    NSLog(@"Are they identical? %@", areTheyEqual ? @"Yes" : @"No");
    return YES;
}
```

This sample code is the first that outputs **`No`** to the console log. Why? **`fiveA`** and **`fiveB`** may represent the exact same number–**`5.0`**–but they d not live in the same memory location; they are separate. Read on to learn how we can compare separate but equal objects.

> *When two different objects represent identical data, they are sometimes called semantically equivalent.*

isEqual:

Often times we care less about whether two objects are identical and more about whether or not they are *equal.* The difference is subtle yet critically important. In the example above, the two **`NSNumber`** instances were *equal* but not *identical.* Let's change the code such that a more valuable comparison is performed:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSNumber* fiveA = [NSNumber numberWithFloat:5.0];
    NSNumber* fiveB = [NSNumber numberWithFloat:5.0];
    BOOL areTheyEqual = [fiveA isEqual:fiveB];
    NSLog(@"Are they equal? %@", areTheyEqual ? @"Yes" : @"No");
    return YES;
}
```

any class can write its own custom version. In **NSNumber**, **isEqual** compares its own number with the number passed into it. It doesn't check whether or not the two pieces of data occupy the same place in memory, it checks if they represent the same value.

isEqualTo:

Unfortunately, **isEqual:** is a little slow. If you find yourself in the fortunate position of knowing exactly which types of data you're comparing, you can use the more specific **isEqualTo<Type>:** method. Here's the same example from above re-written to use a specific form of **isEqual:**

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSNumber* fiveA = [NSNumber numberWithFloat:5.0];
    NSNumber* fiveB = [NSNumber numberWithFloat:5.0];
    BOOL areTheyEqual = [fiveA isEqualToNumber:fiveB];
    NSLog(@"Are they equal? %@", areTheyEqual ? @"Yes" : @"No");
    return YES;
}
```

isEqualToNumber: is faster than the ubiquitous **isEqual:** however, it requires that both parameters be **NSNumber** types. You'll find that many classes offer shortcut **isEqualTo…** methods. Type them into Xcode and the possible options will appear as you type.

Comparators

Comparing equality isn't all that's worth comparing. When working with numbers we may want to compare magnitude as well. For that, we have a variety of operators that also evaluate to **BOOL** values.

Comparator	Symbol	Purpose	Example 1	Example 2
Less than	**<**	Return **YES** if A is less than B	**5 < 6** results in **YES**	**6 < 6** results in **NO**
Less than or equal to	**<=**	Return **YES** if A is less than or equal to B	**5 <= 6** results in **YES**	**6 <= 6** results in **YES**
Greater than	**>**	Return **YES** if A is greater than B	**5 > 6** results in **NO**	**7 > 6** results in **YES**
Greater than or equal to	**>=**	Return **YES** if A is greater than or equal to B	**5 >= 6** results in **NO**	**6 >= 6** results in **YES**
Not equal	**!=**	Return **YES** if A is not equal to B	**5 != 6** results in **YES**	**6 != 6** results in **NO**

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger twelve = 12;
    NSInteger thirteen = 13;
    NSLog(@"Is 12 greater than 13? %@", twelve > thirteen ? @"Yes" : @"No");
    NSLog(@"Is 12 less than 13? %@", twelve < thirteen ? @"Yes" : @"No");
    NSLog(@"Is 13 greater than or equal to 12? %@", thirteen >= twelve ? @"Yes" : @"No");
    NSLog(@"Is 13 less than or equal to 12? %@", thirteen <= twelve ? @"Yes" : @"No");
    NSLog(@"Is 12 different than 13? %@", twelve != thirteen ? @"Yes" : @"No");
    return YES;
}
```

Which of course will output the following:

```
2014-06-13 09:35:20.661 Test[775:303] Is 12 less than 13? Yes
2014-06-13 09:35:20.662 Test[775:303] Is 13 greater than or equal to 12? Yes
2014-06-13 09:35:20.662 Test[775:303] Is 13 less than or equal to 12? No
2014-06-13 09:35:20.662 Test[775:303] Is 12 different than 13? Yes
```

Not Equal

The not symbol (**!** - an exclamation mark) returns the logical opposite of the **BOOL** value it precedes. We can turn a **NO** into a **YES** by surrounding that statement with parentheses and placing a **!** at the front. For example:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    BOOL not14LessThan5 = !(14 < 5);
    NSLog(@"Is the opposite of 14 less than 5 true? %@", not14LessThan5 ? @"Yes" : @"No");
    return YES;
}
```

As you can see above, **14 < 5** would usually evaluate to **NO**, a false statement. However, if surrounded by parentheses and not'd–**!**–we flip it t a **YES**. This code snippet outputs:

```
2014-06-13 09:40:03.719 Test[808:303] Is the opposite of 14 less than 5 true? Yes
```

Checking For Nil

If a variable type requires an ***** preceding its name (**NSNumber**, **NSString**, etc.) this means its data may be absent. When you assign a primitive type such as **int**, **long**, **BOOL**, **NSInteger**, **CGFloat**, etc. you are guaranteed the existence of that variable.

> *If you don't specify the default value for a simple type, the result is undefined, so you should always specify a default value, usually* **0**.

Here's an example showing the difference:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger someInt = 0;
    NSString *someString;
    NSLog(@"What's our integer? %ld", (long)someInt);
    NSLog(@"And our string? %@", someString);
    return YES;
}
```

This prints the following to the command prompt:

```
2014-06-13 09:52:32.939 Test[847:303] What's our integer? 0
2014-06-13 09:52:32.941 Test[847:303] And our string? (null)
```

The default, uninitialized **NSString** object is **nil**, which appears as **(null)** when converted to a string. **nil** is a technical term for *nothing*, *nada zip*. Let's see what happens if we try to use **someString** before it's been initialized:

AppDelegate.m

```
{
    NSString *someString;
    NSMutableString *someMutableString = [someString mutableCopy];
    [someMutableString appendString:@"A Rainbow Unicorn"];
    NSLog(@"And our new mutable string? %@", someMutableString);
    return YES;
}
```

If you guessed that **someMutableString** would become **"A Rainbow Unicorn"**, sorry to crush your fanciful magic dreams. This, once again, is printed to the log:

```
2014-06-13 10:16:01.354 Test[980:303] And our new mutable string? (null)
```

This is because you cannot invoke methods on objects that *aren't there*.

We can't call **mutableCopy** on an absent object because there's nothing to make a copy of; it assigns emptiness to **someMutableString**. And since **someMutableString** is also nothing, we can't **append** anything to it because it never existed to begin with. Due to the nature of the Objective-C runtime, these operations are technically *legal* although they certainly could result in crashes on many other platforms.

There are several ways to check whether or not an object exists, the first of which is comparing it to **nil**:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *someString;
    BOOL someStringExists = someString != nil;
    NSLog(@"Does some string exist? %@", someStringExists ? @"Yes" : @"No");
    return YES;
}
```

This prints the following to the command prompt:

```
2014-06-13 10:24:45.293 Test[1010:303] Does some string exist? No
```

You can also use **!** before an object to test if it exists.

!someObject is shorthand for **someObject == nil**.

Here's an example:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSString *someString;
    BOOL someStringIsNil = !someString;
    NSLog(@"Is someString nil? %@", someStringIsNil ? @"Yes" : @"No");
    return YES;
}
```

Remember that the **!** operator returns the logical opposite. When placing it in front of an object's name, the object is treated as a **BOOL**.

nil, **null**, etc. are aliases for **0**.

0 is also **NO** or **false** while **YES** and **true** are represented by the number **1**.

The opposite of **NO** is **YES**. Therefore, if an object is missing it's represented by **nil**, and the opposite of that resolves to **YES**.

Your assignment	Ask a question	Submit your work

Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and implement solutions to make the tests pass. Exercise descriptions are found the the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed equality"
$ git push origin master #=> send your changes to your remote GitHub repo
```

When you've completed the corresponding exercises, submit this assignment with the relevant repo and commit links.

assignment completed

COURSES

Full Stack Web Development
Frontend Web Development
UX Design
Android Development
iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy
Terms of Service

MADE BY BLOC

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

If / Else and Ternary Operators

Life is all about making decisions. When we write code, we state our intentions so the computer can act on our behalf. Along these lines, we must instruct the computer to act differently in different situations.

Imagine we are in charge of an amusement park and we need to train new ticket taker employees. We might tell them to only let children ride the SuperMegaDeath™ roller coaster if they are above a certain height. If the child is above 120 centimeters tall, then they can ride, otherwise they can't.

This is conditional logic. In programming, conditional logic is how we instruct the computer to make decisions.

It turns out that the employees at our amusement park are slacking off too much, so we decide that we can write a program to handle the process for us. This program will have to check a child's height and print out a message to say if the child is allowed on the ride.

Let's start with a simple case in which we already know the child is tall enough.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    BOOL childIsTallEnough = YES;
    return YES;
}
```

We've defined a variable to hold this value. The type of the variable will be a **Boolean**, as indicated by the word **BOOL** in the code. A boolean is simply a yes or no value. In the example above, the value is **YES**.

?

> *Note that in many programming languages* **TRUE** *and* **FALSE** *are equivalent to* **YES** *and* **NO**. *Objective-C supports both options. Keep that in mind as you might encounter code that uses* **TRUE** *and* **FALSE**.

Let's print a message if the child is tall enough.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    BOOL childIsTallEnough = YES;
+    if (childIsTallEnough) {
+        NSLog(@"Enjoy the Ride!");
+    }
    return YES;
}
```

This is a conditional. It works by checking the value inside the parentheses to see if it is true. If it is true, or **YES**, then it runs the code inside th curly braces.

Run the project and the program should print the message.

Try changing the **YES** to a **NO** and see what happens when you run the project again.

```
{
-     BOOL childIsTallEnough = YES;
+     BOOL childIsTallEnough = NO;
    if (childIsTallEnough) {
        NSLog(@"Enjoy the Ride!");
    }
    return YES;
}
```

The program should print nothing.

This isn't exactly what we want. It would be better to politely ask the child to step out of line if they aren't tall enough. We can accomplish this by adding an else statement.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    BOOL childIsTallEnough = NO;
    if (childIsTallEnough) {
        NSLog(@"Enjoy the Ride!");
+    } else {
+        NSLog(@"Beat it Kid!");
+    }
    return YES;
}
```

The **`else`** tells the program what to do when the value of the condition is **`NO`**.

Run the code again it should print "Beat it Kid!"".

Adding Math

Now that we have our functionality working let's add real numbers to determine what it means to be tall enough.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
+    NSInteger minimumHeight = 120;
+    NSInteger childHeight = 130;

-    BOOL childIsTallEnough = NO;
+    BOOL childIsTallEnough = (childHeight >= minimumHeight);
    if (childIsTallEnough) {
        NSLog(@"Enjoy the Ride!");
    } else {
        NSLog(@"Beat it Kid!");
    }
    return YES;
}
```

We've created number variables to store the minimum height and the child's height. We've also validated that the child is tall enough with the greater than or equal operator **`>=`**. You probably remember from math class that **`>=`** means the value on the left is greater than or equal to the value on the right.

`(childHeight >= minimumHeight)` should be true in this case. Run the program again and it should print "Enjoy the ride!".

We don't need the **`childIsTallEnough`** variable anymore, so we can replace it in our conditional with the comparison that we wrote in the last step.

AppDelegate.m

```
{
    NSInteger minimumHeight = 120;
    NSInteger childHeight = 130;

-    BOOL childIsTallEnough = (childHeight >= minimumHeight);

-    if (childIsTallEnough) {
+    if (childHeight >= minimumHeight) {
       NSLog(@"Enjoy the Ride!");
    } else {
       NSLog(@"Beat it Kid!");
    }
    return YES;
}
```

This program is improving, but so far it's only handling a single child's height. This is going to cause major problems when we have long lines of people trying to ride the roller coaster.

To make our program handle multiple children we will create a method.

Add an empty method to the end of our file and call it **`checkChildHeight`**.

AppDelegate.m

```
    return YES;
}

+ - (void)checkChildHeight:(NSInteger)childHeight {
+
+ }
```

This method does nothing right now, so let's move our condition into it:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
-    NSInteger minimumHeight = 120;
    NSInteger childHeight = 130;

-    if (childHeight >= minimumHeight) {
-        NSLog(@"Enjoy the Ride!");
-    } else {
-        NSLog(@"Beat it Kid!");
-    }
    return YES;
}

- (void) checkChildHeight:(NSInteger)childHeight {
+    NSInteger minimumHeight = 120;

+    if (childHeight >= minimumHeight) {
+        NSLog(@"Enjoy the Ride!");
+    } else {
+        NSLog(@"Beat it Kid!");
+    }
}
```

Now that the behavior is in a method we have to call that method with the height we want to check.

AppDelegate.m

```
{
    NSInteger childHeight = 130;
+     [self checkChildHeight:childHeight]; //prints "Enjoy the ride!

+     NSInteger secondChildHeight = 110;
+     [self checkChildHeight:secondChildHeight]; //prints "Beat it Kid!"
    return YES;
}
```

Removing duplication

This program can now handle any situation we throw at it, but there's one thing left to do.

Notice how in each part of the conditional we have to write **NSLog**. It would be nice if we only had to write that once. In order to accomplish that we are going to separate the process of determining the message from the act of printing it.

Create an empty string variable to hold our message, and use the conditional to assign a string based on its result:

AppDelegate.m

```
- (void) checkChildHeight:(NSInteger)childHeight {
    NSInteger minimumHeight = 120;

+     NSString *message;

    if (childHeight >= minimumHeight) {
-         NSLog(@"Enjoy the Ride!");
+         message = @"Enjoy the Ride!";
    } else {
-         NSLog(@"Beat it Kid!");
+         message = @"Beat it Kid!";
   }
}
```

Finally, we log the message variable.

AppDelegate.m

```
- (void) checkChildHeight:(NSInteger)childHeight {
    NSInteger minimumHeight = 120;

    NSString *message;

    if (childHeight >= minimumHeight) {
        message = @"Enjoy the Ride!";
    } else {
        message = @"Beat it Kid!";
    }

+     NSLog(@"%@", message);
}
```

When we run the program again it should be working just as it was before.

Ternary

We've removed the duplication with **NSLog** but we added a different kind of duplication in its place. Our program is also longer than it was before.

To solve these problems we'll use a ternary.

A ternary is a one line conditional statement. A ternary looks like this:

If **`conditional`** is true; **`variable`** is assigned **`yes_value`**. Otherwise, it's assigned **`no_value`**.

We can change our code to use this like so:

AppDelegate.m

```
- (void) checkChildHeight:(NSNumber *)childHeight {
    NSInteger minimumHeight = 120;

-     NSString *message;

-     if (childHeight >= minimumHeight) {
-         message = @"Enjoy the Ride!";
-     } else {
-         message = @"Beat it Kid!";
-     }

+      NSString *message = (childHeight >= minimumHeight) ? @"Enjoy the ride" : @"Take a Hike";

    NSLog(@"%@", message);
}
```

Our final method looks like this:

AppDelegate.m

```
- (void) checkChildHeight:(NSNumber *)childHeight {
    NSInteger minimumHeight = 120;

    NSString *message = (childHeight >= minimumHeight) ? @"Enjoy the ride" : @"Take a Hike";

    NSLog(@"%@", message);
}
```


Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint. Implement a solution using **`if else`** statements for the first exercise. For the second exercise, which is a method named **`dollarCostForAppleFlavoredVodka`**, refactor the conditional with a ternary and make sure the tests still pass.

Exercise descriptions are found the the header (**`.h`**) file, while you should implement your solutions in the implementation (**`.m`**) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed if, else, ternary"
$ git push origin master #=> send your changes to your remote GitHub repo
```

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy
Terms of Service

MADE BY BLOC

Tech Talks & Resources
Programming Bootcamp Comparison
Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
Swiftris: Build Your First iOS Game with Swift
Webflow Tutorial: Design Responsive Sites with Webflow
Ruby Warrior
Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io
Considering enrolling? (404) 480-2562
Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Loops

In programming you'll often find yourself working with large sets of data. Typically, your application will be required to iterate through each piece and perform an operation on it. When washing dishes for example, you remove a single dish from the dirty tray, wash it, then dry it. Once that dish is clean, you replace it and move onto the next one in the dirty tray. Your dirty dishes comprise your data set and the cleaning of each represents the operation to perform.

While Loop

The **`while`** loop is the most basic of looping structures available. Let's see an example of some work that *should* be done in a **`while`** loop:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger five = 5;
    NSInteger fiveFactorial;
    // Find the factorial
    fiveFactorial = five;
    fiveFactorial = fiveFactorial * --five;
    fiveFactorial = fiveFactorial * --five;
    fiveFactorial = fiveFactorial * --five;
    fiveFactorial = fiveFactorial * --five;
    NSLog(@"The factorial of 5 is %ld", (long) fiveFactorial);
    return YES;
}
```

> *The factorial of a number is a mathematical operation requiring the multiplication of the original number with all positive whole numbers smaller than itself. In this case, we're calculating* **`5 * 4 * 3 * 2 * 1`**. ***Read more here.***

As you can see, we repeated a particular operation four times - **`fiveFactorial = fiveFactorial * --five;`** - the code was *literally* copied and pasted. Loops help us avoid repetitive code and reuse logic we've already discovered in a compact and intuitive way.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger five = 5;
    NSInteger fiveFactorial = five;
    while (five > 1) {
        fiveFactorial = fiveFactorial * --five;
    }
    NSLog(@"The factorial of 5 is %ld", (long)fiveFactorial);
    return YES;
}
```

A **`while`** loop requires two things:

- **A conditional statement**

This came in the form of **`--five`**. We decremented the **`five`** variable on each cycle of the loop. If we failed to update the conditional variable, it would remain the same–**`5`**. As a result, the conditional statement would forever evaluate to **`YES`** and the loop would infinitely repeat. In programming terms, this kind of bug is known as an *infinite loop.*

Let's print a statement within our loop to give you a better idea of what's happening:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger five = 5;
    NSInteger fiveFactorial = five;
    while (five > 1) {
        NSLog(@"five is now: %ld", (long) five);
        fiveFactorial = fiveFactorial * --five;
    }
    NSLog(@"The factorial of 5 is %ld", (long)fiveFactorial);
    return YES;
}
```

Executing this code results in the following printed statements:

```
2014-06-13 13:16:45.332 Test[1764:303] five is now: 5
2014-06-13 13:16:45.342 Test[1764:303] five is now: 4
2014-06-13 13:16:45.342 Test[1764:303] five is now: 3
2014-06-13 13:16:45.343 Test[1764:303] five is now: 2
2014-06-13 13:16:45.343 Test[1764:303] The factorial of 5 is 120
```

For Loop

The **`for`** loop is just another way to create a loop. The syntax is slightly different. Rest assured that anything you can do with a **`while`** loop, you can do with a **`for`** and vice-versa. Choose whichever you feel most comfortable with. Here's the example from above refactored using a **`for`** loop:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSInteger fiveFactorial = 1;
    for (NSInteger five = 5; five > 1; five--) {
        fiveFactorial = fiveFactorial * five;
        NSLog(@"five is now: %ld", (long)five);
    }
    NSLog(@"The factorial of 5 is %ld", (long)fiveFactorial);
    return YES;
}
```

```
2014-06-13 13:30:39.537 Test[1802:303] five is now: 5
2014-06-13 13:30:39.540 Test[1802:303] five is now: 4
2014-06-13 13:30:39.540 Test[1802:303] five is now: 3
2014-06-13 13:30:39.540 Test[1802:303] five is now: 2
2014-06-13 13:30:39.541 Test[1802:303] The factorial of 5 is 120
```

As you can see from the output, the **`for`** loop produces the exact same results as the **`while`**. A **`for`** loop consists of three things: an *initialization*, a *conditional statement* and an *update*. Here's the example again:

AppDelegate.m

```
// for (initialization; conditional statement; update) {
for (NSInteger five = 5; five > 1; five--) {
```

The first statement following the open parenthesis **(** is the initialization. This is most commonly used to initialize the variable which will be compared in the conditions and updated after each loop.

- **Conditional statement**
  Following the initialization and a semi-colon **;**, the conditional statement behaves identically to the one found in the **`while`** loop. It is checked at the beginning of each cycle and as long as it evaluates to **`YES`**, the looping will continue.
- **Update**
  This last section is reserved for updating the conditional variable; the one that was declared during the initialization and validated in the conditional statement. The update statement is executed at the end of each loop cycle.

Open the BlocExercises project in Xcode. Open the folder that corresponds to this checkpoint and implement solutions to make the tests pass. Exercise descriptions are found the the header (`.h`) file, while you should implement your solutions in the implementation (`.m`) file. The test file will also give you some clues for implementing a proper solution, so be sure to read the tests as well.

Run the tests that correspond with the implementation file. Press ⌘**5** to view the **Test Navigator** and click the "play" button to the right of the corresponding test group. You can also run individual tests by clicking the "play" button corresponding to a specific test.

Commit and push your changes to GitHub:

Terminal

```
$ pwd #=> you should be in your forked ios-exercises directory. If not, cd into it.
$ git status #=> you should see the exercise file you just updated
$ git add . #=> stage the changes
$ git commit -m "Completed loops"
$ git push origin master #=> send your changes to your remote GitHub repo
```

When you've completed the corresponding exercises, submit this assignment with the relevant repo and commit links.

assignment completed

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

COPYRIGHT © 2015 BLOC

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Arrays

> *"You are never too old to set another goal or to dream a new dream."*
> *- C.S. Lewis*

An array of objects in programming is similar to its real life counterpart, it maintains an ordered set of items. Arrays are ubiquitous among software applications; they are most often used to gather similar elements into a list. For example, if you were to write a todo-list application, you may keep each of the user's tasks in an array.

> *This is a challenging checkpoint. Take your time with it. Read a section or two then play around with the code snippets in Xcode until you are comfortable.*
>
> *You do not have to understand all of this material on your first pass; there's a lot to absorb. We've all struggled with programming concepts before and arrays are no exception.*

NSArray

NSArray is the de-facto array class in Objective-C. Here's an example of storing a to-do list in an array:

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application di(  )LaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSArray *myTodoList = [NSArray arrayWithObjects:@"Clean the house",
                           @"Feed the dog",
                           @"Take out the trash",
                           @"Fight crime",
                           nil];
    NSLog(@"I have %ld things to do today!", myTodoList.count);
    NSLog(@"First thing I need to do: %@", myTodoList[0]);
    return YES;
}
```

output

```
2014-06-16 09:35:26.176 Test[776:303] I have 4 things to do today!
2014-06-16 09:35:26.186 Test[776:303] First thing I need to do: Clean the house
```

Your first array, **myTodoList**, has a total of four objects in it. As you may have noticed from the first **NSLog** statement, you can recover the size of an array by accessing the **count** property. This is done by adding a period (**.**) to the end of its name followed by **count**.

To access an individual element within your array, place square brackets **[…]** after its name and a digit within them to indicate which element you'd like to recover. Arrays are *0-indexed*. The first element in the array lies at index **0**, the second at index **1**, so on and so forth. Array indices range from **0** to **count - 1**.

Here's an alternative way to instantiate and access elements within your arrays:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSArray *myTodoList = @[@"Clean the house",
                            @"Feed the dog",
                            @"Take out the trash",
                            @"Fight crime"];
    NSString *firstThing = [myTodoList objectAtIndex:0];
    NSLog(@"First thing I need to do: %@", firstThing);
    return YES;
}
```

output

```
2014-06-16 09:46:25.594 Test[821:303] First thing I need to do: Clean the house
```

Similar to how **NSString** literals are established by placing an **@** symbol followed by a pair of quotes, a hard-coded **NSArray** can be instantiate by proceeding the **@** symbol with a couple of square brackets: **@[object0, object1, …, lastObject]**.

NSArrays are immutable; once you've established an **NSArray** it can no longer be altered. You cannot remove, add, or re-assign the elements of an **NSArray**. However...

NSMutableArray

NSMutableArray is perfect for when you expect to alter an array after it's been instantiated. This array type supports all the features of **NSArray** while also being modifiable, adjustable and all around malleable. It's far more likely that an application supporting todo-list functionality would employ an **NSMutableArray** rather than an **NSArray**. Use **NSArray**s when you're certain that the contents will never need t change.

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSMutableArray *myMutableTodoList = [@[@"Clean the house",
                                           @"Feed the dog",
                                           @"Take out the trash",
                                           @"Fight crime"] mutableCopy];
    // Whoops, forgot one!
    [myMutableTodoList addObject:@"Solve world hunger"];
    // I cleaned the house last week, no need to do it again ;)
    [myMutableTodoList removeObjectAtIndex:0];
    NSLog(@"First thing I need to do: %@", [myMutableTodoList objectAtIndex:0]);
    return YES;
}
```

output

```
2014-06-16 10:02:13.677 Test[887:303] First thing I need to do: Feed the dog
```

After removing the item at index **0**, each remaining item is shifted down by 1. What was previously at index **1** is now at index **0**, index **2** is now at index **1**, etc. Therefore when we print the contents at index **0** we see that to **Feed the dog** is now our first task.

Sorting

Another useful feature belonging to mutable arrays is the ability to sort their contents. You may sort an array by several means. The first of the two methods you will learn is to employ an **NSSortDescriptor**. Let's sort the to-do list alphabetically:

AppDelegate.m