Chapter No. 2
"Instant Gratification – Your First Application"
In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.2 "Instant Gratification – Your First Application"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

**Abhishek Nalwaya** is the author of the book, Rhomobile Beginner's Guide. He is a Ruby enthusiast and loves to participate regularly at Ruby and Ruby on Rails meetup groups. He works for Mckinsey and Company IT. He has spoken at many conferences, meetups, and was the speaker at RubyConf India 2012 and RubyMotion Conference 2013

**Akshat Paul** is a programmer and is working as a lead developer at Mckinsey and Company IT. He has extensive experience of mobile application development and has delivered many enterprise and consumer applications.

In other avatars, Akshat frequently speaks and evangelizes at conferences and meetup groups on various technologies; this way he plays his part in giving back to the community. He has given talks at RubyConfIndia and inspect-RubyMotion Conference. He also has a strong belief in Agile methodologies for creating world-class software, and is a Certified Scrum Master (CSM).

For More Information:  
RubyMotion iOS Development Essentials

With the arrival of the iOS family of devices, the direction of software development has radically changed. Today people are spending considerable amounts of time on smart devices instead of PCs, which is generating an unprecedented amount of revenue that no industry has ever seen. Despite this, it still fits in your pocket.

So far the application development scene for the iOS ecosystem has been dominated by Objective-C. However, with the introduction of the revolutionary RubyMotion tool chain, Ruby developers are no longer outcasts for creating pure native iOS applications. They can make use of every bit of the all-powerful iOS SDK; and the best part is this can be done without using Xcode.

Both Ruby and RubyMotion are the brainchild of folks who wanted to simplify things in a complex world. Yukihiro Matsumoto (also known as Matz) is credited for creating the Ruby programming language, which is often regarded as a developer's best friend. And Laurent Sansonetti is credited for creating the ground-breaking tool chain, RubyMotion.

RubyMotion iOS Development Essentials will appeal to a developer's mind, especially to the technocrats looking for a reliable tool chain for iOS development. This book] is a step-by-step guide to build an iOS application from scratch to deployment.

What This Book Covers

Chapter 1, Getting Ready for RubyMotion, gets you acquainted with RubyMotion. Here, we will start with an introduction to RubyMotion, followed by detailed installation steps.

Chapter 2, Instant Gratification – Your First Application, explains how to create a simple Hello World application and also the structure of RubyMotion applications in general.

Chapter 3, Evolution – From Objective-C to RubyMotion, helps you understand the journey from Objective-C to RubyMotion. This chapter is also a quick guide to understanding the RubyMotion syntax corresponding to its Objective-C syntax.

Chapter 4, Mastering MVC Paradigm, focuses on writing better code with the Model-View-Controller architecture. We will also learn about connecting the application to an external API.

For More Information:
Chapter 5, User Interface – Cosmetics for Your App, describes how the user interface is a key part of an iOS application. Also, this chapter explains how we can use the various user interface elements.

Chapter 6, Device Capability – Power Unleashed, teaches you how to use various device capabilities, such as Camera, Location Manager, Gestures, Core Data, and Address Book. We will create sample applications for each one of them to understand them better.

Chapter 7, Interface Builder and WebView – More Goodies!, explains how to use the interface builder and UIWebView with RubyMotion applications.

Chapter 8, Testing – Let's Fail Gracefully, discusses Unit Testing and Functional Testing in a RubyMotion application by following the philosophy of Test-driven Development.

Chapter 9, Creating a Game Application, helps you create a popular arcade game, Whack-a-Mole, using Cocoa2D and RubyMotion. This is one of the most exciting and unique features of working with RubyMotion where it's possible to create graphical gaming applications.

Chapter 10, Getting Ready for the App Store, explains the process of submitting a RubyMotion application to the Apple App Store.

Chapter 11, Extending RubyMotion, describes how to augment our RubyMotion applications by making use of the already available open source gems, such as TeaCup, BubbleWrap, and Address Book.

For More Information:
Now that we are all charged up about RubyMotion and have our system set up, let's create a simple RubyMotion application. We will try and keep it simple, but sometimes you may feel disconnected by monotonously typing the code. Although, going along is enough for now. Remember that mimicry is a powerful form of learning; that's how we have learned most of our skills, such as talking, reading, writing, and that is how you will learn to program with RubyMotion. We promise you that by the end of this book, you will have sufficient knowledge of RubyMotion to create an iOS application and make it live on the App Store. In this chapter we will cover the following topics:

- Creating your first RubyMotion application
- Understanding the folder structure
- Exploring the command line
- Configuring your application
- REPL – the interactive console
- The debugger

"Dream the impossible, seek the unknown, and achieve greatness."

–Anonymous
Your first application

Let's start with the classic HelloWorld application. As we have discussed in the last chapter, RubyMotion has a terminal-based flow, so let's fire up our terminal and create our very first RubyMotion application.

```
$ motion create HelloWorld
Create HelloWorld
Create HelloWorld/.gitignore
Create HelloWorld/Rakefile
Create HelloWorld/app
Create HelloWorld/app/app_delegate.rb
Create HelloWorld/resources
Create HelloWorld/spec
Create HelloWorld/spec/main_spec.rb
```

If you observe closely the output on the terminal screen, you will see that a lot of files and directories have been generated by a single `motion` command, which automatically creates standard directories, and you will also see the file structure that will quickly bring us onboard with app development, which we can work on later and enhance to make a fully functional application. Moreover, since the structure is common to all the RubyMotion apps, it's easy to understand.

```
Just like the `motion` command, popular frameworks such as Ruby on Rails also have commands such as `rails` to create a predefined layout of the application.
```

The following steps automatically compile the code and start the application on a simulator:

1. Start the application, traverse to the application directory, and type the following command:
   ```
   $ cd HelloWorld
   $ rake
   Build ./build/iPhoneSimulator-6.0-Development
   Compile ./app/app_delegate.rb
   Create ./build/iPhoneSimulator-6.0-Development/HelloWorld.app
   Link ./build/iPhoneSimulator-6.0-Development/HelloWorld.app/
   HelloWorld
   Create ./build/iPhoneSimulator-6.0-Development/HelloWorld.app/
   Info.plist
   Create ./build/iPhoneSimulator-6.0-Development/HelloWorld.app/
   PkgInfo
   ```

For More Information:

Create ./build/iPhoneSimulator-6.0-Development/HelloWorld.dSYM
warning: no debug symbols in executable (-arch i386)
Simulate ./build/iPhoneSimulator-6.0-Development/HelloWorld.app

Wow! The `rake` command automatically compiles the code and starts the application on a simulator. So far, we have not created any views for our application; that's why we can see a blank screen. It looks boring, but remember that we have not written a single line of code. So let's write some code, create some views, and build our application again.

You can open the RubyMotion project in your favorite editor. If you don't have an editor yet, you can use either TextEdit or VIM.

For More Information:
2. Open the file app_delegate.rb in the app folder and add the following code in it:

```ruby
class AppDelegate
  def application(application, didFinishLaunchingWithOptions:launchOptions)
    alert = UIAlertView.new
    alert.message = "Hello World!"
    alert.show
    true
  end
end
```

3. Let's re-run our application by traversing to the application directory and typing the execute command (`rake`):

```bash
$rake
```

The `rake` command will compile our code and fire up the iPhone simulator. We can see a blue pop-up saying **Hello World!** in the following screenshot:
Let's understand the code that we have written in AppDelegate. Here the application method (didFinishLaunchingWithOptions:launchOptions) is called first when our application starts. This will be the starting point of our application and the right place to define our window.

RubyMotion functions are a combination of the usual Ruby name method (didFinishLaunchingWithOptions) with their named parameters; a variable directly follows the function, which it refers to, and therefore, we don't need to know the implementation of the function.

Named parameters were added to RubyMotion to preserve the existing Objective-C APIs, and the extra symbols are required parts of the method name, for example, didFinishLaunchingWithOptions:launchOptions.

As discussed, the code written in AppDelegate will be called automatically as the application is initialized.

In the following code snippet, we created an object alert of the UIAlertView class and then we assigned a Hello World! string to the message attribute of the object. Now we have our alert object ready. To display this alert on the device screen, we call the show method on the alert object as follows:

```ruby
alert = UIAlertView.new
alert.message = "Hello World!"
alert.show
```

UIAlertView is a class that is bundled in the UIKit framework of the iOS. We can use this class to display an alert message on the screen. This class is inherited from UIView that is inherited from UIResponder that, in turn, is inherited from NSObject.

**Why do we see the NS prefix?**

Objective-C is a superset of C and thus doesn't have namespaces like in C++; therefore, the symbols must be prefixed with a unique prefix so that they don't collide. This is particularly important for symbols defined in a framework. The original code for the Cocoa frameworks came from the NextStep libraries, and so the NextStep engineers chose to prefix their symbols with NS.

4. To exit the application, close the simulator by selecting the exit option or press Command + Q.

For More Information:
The iOS simulator is a great tool for testing your applications quickly. It comes bundled with Xcode. But you can't test everything on the simulator. To test the shaking of a device, camera, GPS, Accelerometer, Gyroscope, and other device capabilities, you may require additional products to pass device data to the app in the simulator.

Folder structure

In this section, we will understand the folder structure of our application as we know from the previous section that `motion create <project name>` sets up the directory structure with all the essential files to run a simple RubyMotion application. Let's walk through each one of them to have a precise understanding of their function:

- **The app folder**: This is the core of your application code; you will write most of your code in this folder. RubyMotion iterates in this folder and loads any .rb file that it catches.

  If you want to keep your code somewhere else other than the app directory, add the folder path to the Rakefile.

- **The app_delegate.rb file in the app folder**: This file is at the heart of the RubyMotion application. If you are a little familiar with iOS development, this is the delegate file. A delegate is an object that usually reacts to some event in another object and/or can affect how another object behaves. There are various methods that can be implemented in UIApplicationDelegate. These methods are called during the different phases of an application, such as during the finish of its launch, during termination, when the application is low on memory, and during the occurrence of important changes. While the application is running, tracking its state transitions is one of the main jobs of the application delegate.

  App delegates use the method `application:didFinishLaunchingWithOptions` as the first entry point. This method is called after your application has been launched. When this method is called, your application is in the inactive state. A few other methods available are:

  - `applicationWillEnterForeground`
  - `applicationWillTerminate`
  - `application:shouldSaveApplicationState`
  - `application:shouldRestoreApplicationState`

For More Information:

A full list of available methods can be obtained from the iOS developer library (http://developer.apple.com/library/ios). The good part here is that most of the methods are self-explanatory by their name. For example, `applicationWillEnterForeground` will be called when your application is relaunched.

We see that in some iOS 6 applications, the app is restored to the previous state; we can handle this in an application delegate.

- **The resources folder**: As the name suggests, the resources folder contains static content, such as images, sounds, UI layouts, and icons that we use in our applications.
- **The Spec folder**: This folder contains automated test cases. RubyMotion supports a Ruby testing framework, Bacon; it is a small RSpec clone that is used for writing unit, functional, and UI tests. By default, it creates `main_spec.rb` as an example.
- **Rakefile**: With Rakefile we can configure our application name, resources, gems to be included, and the code location. We will discuss more about Rakefile later in this chapter.

### Some more goodies
We know that it's not so much fun to have only a simple HelloWorld pop-up as our very first application, so let's jazz up our code by adding some more goodies to our alert box; and this time, let's do things in a much better way.

Earlier we had added an alert box in the delegate itself. Actually it is not a good idea to write code in the application delegate. It is better to write code in a Model-View-Controller (MVC) way. Right now we won't cover all three parts of the MVC architecture for now let's begin with the controller for our application and add three buttons in this alert box, add a title, and add a message for the title box.

The class `UIAlertView` that we've used in the last section has numerous properties, such as title, message, delegate, cancelButtonTitle, otherButtonTitles, and many more. Let's use a few of them in our application as follows:

1. Create a file `root_controller.rb` in the app folder and add the following code:

```ruby
class RootController < UIViewController
  def viewDidLoad
    alert = UIAlertView.alloc.initWithTitle "This is foo title",
```
message: "Do you like this example?",
    delegate: nil,
    cancelButtonTitle: "cancel"
    otherButtonTitles: "Yes", "No", nil
    alert.show
end
end

2. To call this controller, we need to update our AppDelegate class. Replace the following code in your app_delegate.rb file:

class AppDelegate
  def application
    (application, didFinishLaunchingWithOptions: launchOptions)
      @window = UIWindow.alloc.initWithFrame
        (UIScreen.mainScreen.bounds)
      @window.rootViewController = RootController.alloc.init
        (UIScreen.mainScreen.bounds)
      @window.rootViewController = RootController.alloc.init
        @window.rootViewController.wantsFullScreenLayout = true
      true
    end
  end
end

3. Start the simulator by running the rake command from the console inside your application directory as follows:
Chapter 2

That's cool; our earlier HelloWorld pop-up has now been replaced with an alert box that has a title, a cancel button, and two other buttons.

The iOS SDK has been built around the MVC pattern that separates responsibilities and ends up with an application that is easy to design and maintain.

Let's understand the code
When an iPhone application starts, it puts a window on the screen, which we have created using the UIWindow class. You can think of a window as a drawing board where you can put anything, such as a button, textbox or label. The instance of the UIWindow class manages and coordinates the views of an application, which are displayed on a device screen.

A UIScreen object contains the bounding rectangle of the device's entire screen. So, UIScreen.mainScreen.bounds returns the rectangle size according to the screen size and orientation of the device.

Every iOS application needs at least one window, which is an instance of the UIWindow class.

You might be wondering, should I remember all the properties and methods of the Apple iOS SDK, such as UIAlertView? It is not necessary to memorize them as one can always refer to the properties and methods from the iOS development library. Nevertheless, having a basic idea about the usage of a class can come in handy at times. The popular IDE, RubyMine, supports RubyMotion. It also has a useful autocompletion feature.

The more you understand, the less you have to memorize.

Exploring the command line
RubyMotion is based on an underlying principle, "to use the tools which developers love". Therefore, to create an application using RubyMotion, we require only two tools; the first is your favorite editor and the second is the terminal. While developing a RubyMotion application, you will be required to familiarize yourself with the command line. Familiarity with the terminal always helps in faster and comfortable development.

For More Information:
Now that we have created our HelloWorld application, let us explore a few commands that we have already used, and remember that RubyMotion uses them considerably. These commands are responsible for inaugurating our RubyMotion projects, motion and rake.

**Motion command – one-stopshop**

As used previously, the motion command creates our RubyMotion project and also supports various other options. The motion command is similar to the popular framework Ruby on Rails' rails command. Before we go any further, let's fire up our terminal and see what can be done using the motion command.

```
$ motion

Usage:
    motion [-h, --help]
    motion [-v, --version]
    motion <command> [<args...]>

Commands:
    account    Access the software license account
    activate   Activate the software license
    create     Create a new project
    ri         Display API reference
    update     Update the software
    support    Create a support ticket
```

- **motion account**: This displays the account/license information on the browser.
- **motion activate**: If you want to activate your RubyMotion framework with a new license or if you have not yet activated the framework, motion activate can be used.
- **motion create <project name>**: This command will generate a RubyMotion project's skeleton that will have all the essential files needed to begin developing an iOS application.
- **motion ri <API-name>**: This command helps us to find the documentation for the API that has been mentioned.
- **motion update**: RubyMotion is a fast-moving framework and often requires updates. motion update updates your framework from the command line itself.

For More Information:

• **motion support**: There may be times when you have questions only an expert can answer. **motion support** helps you connect with RubyMotion directly, and you can ask a question by filling up a form. It can also be used for any feature request or for reporting a bug.

### Rake tasks – get things done fast

Rake is a simple Ruby build program with capabilities similar to **Make**. RubyMotion's **rake** command has many predefined tasks that help you do several trivial jobs, such as compiling your code to test in a simulator or creating a package to test on a device, with ease. Let's fire up our terminal again and check what tasks can be performed using **rake --tasks**.

```bash
$ rake --tasks
```

The following table elaborates the different Rake tasks:

<table>
<thead>
<tr>
<th>Rake task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rake archive</td>
<td>Create a .ipa archive</td>
</tr>
<tr>
<td>Rake archive:distribution</td>
<td>Create a .ipa archive for distribution</td>
</tr>
<tr>
<td>Rake build</td>
<td>Build everything</td>
</tr>
<tr>
<td>Rake build:device</td>
<td>Build the device version</td>
</tr>
<tr>
<td>Rake build:simulator</td>
<td>Build the simulator version</td>
</tr>
<tr>
<td>Rake clean</td>
<td>Clear build objects</td>
</tr>
<tr>
<td>Rake config</td>
<td>Show project config</td>
</tr>
<tr>
<td>Rake ctags</td>
<td>Generate ctags</td>
</tr>
<tr>
<td>Rake default</td>
<td>Build the project, then run the simulator</td>
</tr>
<tr>
<td>Rake device</td>
<td>Deploy on the device</td>
</tr>
<tr>
<td>Rake simulator</td>
<td>Run the simulator</td>
</tr>
<tr>
<td>Rake spec</td>
<td>Same as a spec:simulator</td>
</tr>
<tr>
<td>Rake spec:device</td>
<td>Run the test/spec suite on the device</td>
</tr>
<tr>
<td>Rake specs: simulator</td>
<td>Run the test/spec suite on the simulator</td>
</tr>
<tr>
<td>Rake static</td>
<td>Create a static library</td>
</tr>
</tbody>
</table>

So Rake has plenty of tasks to do, but most importantly, out of all these tasks, if we simply run Rake, it will build and run our application on the iOS simulator.
Rake file – configuring your application

RubyMotion applications are highly configurable using different attributes in a Rakefile. These attributes, by default, come with a sensible value but can be overridden with custom values. Let’s explore each one of them—this section will come in handy, time and again, as we proceed with our application.

To see your current application configuration, run the `rake config` task, and you will be presented with the following list:

```
$ rake config
background_modes       : []
build_dir              : "./build"
codesign_certificate   : "Error"
delegate_class         : "AppDelegate"
deployment_target      : "6.0"
device_family          : :iphone
 entitlements          : {}
files                  : ["./app/app_delegate.rb", ".app/twitter.rb", ".app/twitter_controller.rb"]
fonts                  : []
frameworks             : ["UIKit", "Foundation", "CoreGraphics", "CoreGraphics"]
icons                  : []
identifier             : "com.yourcompany.MacBaconUI"
interface_orientations : [:portrait, :landscape_left, :landscape_right]
libs                   : []
motiondir              : "/Library/RubyMotion"
name                   : "MacBacon UI"
prerendered_icon       : false
provisioning_profile   : "Error"
resources_dir          : "./resources"
sdk_version            : "6.0"
seed_id                : "Error"
short_version          : "1.0"
specs_dir              : ".spec"
status_bar_style       : :default
version                : "1.0"
weak_frameworks        : []
xcode_dir              : "/Applications/Xcode.app/Contents/Developer"
```
You can see the entire configuration settings for your application. These settings can be modified in a Rakefile. You may find it easy to understand what these properties do by their names, but let us explain a few of them:

- **name**: This is where you can specify the name of your project as a string. By default, the name of your application will be the attribute that you passed during `motion create`.
- **version**: This variable saves the current application version as a string; it is 1.0 by default.
- **identifier**: The project identifier is a string that is in reverse DNS—a naming convention that is in the reverse order of the domain name notation—such as `com.yourcompany.yourapp`.
- **delegate_class**: This is where you specify your application delegate class as a string that is loaded once the application starts. The default value is `AppDelegate` and the class is defined in the `app/app_delegate.rb` file. However, we can rename the `AppDelegate` class to a custom name of our choice and this then has to be updated in the Rakefile.
- **Files**: This shows every `.rb` file in the `app` directory in an array format. The default value is the result of executing the following expression: `Dir.glob('./app/*/*.rb')`
- **framework**: This shows the names of the iOS frameworks that are used in our application in an array format. Soon you will be using many iOS frameworks, such as CoreFoundation, CoreMotion, and others, with your application. The build system is capable of dealing with dependencies, therefore they should be mentioned here. The default value is either `UIKit`, `Foundation`, or `CoreGraphics`.
- **libs**: This variable shows the library paths that are to be linked to the application in an array format. It contains the path to public system libraries, for example, `/usr/lib/libz.dylib`. The default value is `[]`, an empty array.
- **build_dir**: This variable is used to specify the directory path where you want the application build to be created in a string format. It must be relative to the project directory. The directory initially gets created automatically. In case it is not created, a temporary directory will be used instead. The default value is `build`.
- **resources_dir**: This variable is used to specify the directory for the resource files where all the images and icons go in a string format. It must be relative to the project directory. The default value is `resources`.

For More Information:

• **spec_dir**: This variable is used to specify the directory of spec files where all our test cases are present in a String format. The default value is `spec`. It should be relative to the project directory.

• **icons**: This variable lists the icons used for the application present in the resources folder in an array format, for example, `icon.png` and/or `icon-72.png`. The files should be in tune with Apple's **HIG (Human Interface Guidelines)**. By default, the value is `[]`, an empty array.

• **fonts**: This variable lists the names of the font files present in the resources directory in an array format. These fonts will be taken into account while either generating the application bundle or testing on a simulator.

• **prerendered_icon**: iOS application icons usually have a reflective shine on them. For that purpose, this property is used. If it is false, we will get the reflective shine on the icon. By default, the value is false.

• **device_family**: With this property, we can specify which family of iOS device our application supports. The values can be `iphone`, `ipad`, or for universal application `[iphone, :ipad]`. By default, it is `:iphone`.

• **interface_orientations**: Apple iOS devices support various orientations for an application. They can be `portrait`, `landscape_left`, `landscape_right`, or `portrait_upside_down`. By default, the value is an array of `:portrait`, `:landscape_left`, or `:landscape_right`.

• **Xcode_dir**: This configuration tells us where the Xcode is installed.

Giving a new value to the `Xcode_dir` property should generally be done first, before changing other `Rakefile` properties.

• **sdk_version**: This configuration lets us decide which SDK version will be used. By default, the value is the most recent version of the supported SDK.

• **deployment_target**: This configuration shows which iOS SDK to target for the RubyMotion project. By default, the value is of the current SDK version that is installed, but this can be changed to any desired version of the iOS SDK, for example, `6.0` that will use iOS SDK Version 6.0.

• **codesign_certificate**: This configuration shows which code-signing certificate is used. By default, the value is the first iPhone developer certificate in the keychain utility; for example, in our case it is **iPhone developer: Paul Akshat (S3KPMT842Z)**.

• **provisioning_profile**: This configuration variable specifies the path of the provisioning profile.

• **seed_id**: The Apple provisioning profile has an identifier. This configuration shows us the same, which is usually the first application identifier picked from the provisioning profile.

For More Information:

REPL – the interactive console

RubyMotion comes with an interactive console that lets us traverse and scan the code that we are using in our application. The good thing is that the console is connected to the application running on the simulator. This means that if we make any changes from the console, it will be reflected on the simulator in real time. Let's try this with our HelloWorld application.

Run the application as follows:

$rake

As expected, it will open a simulator and the terminal screen will show:

(main) >

Now hold the Command key and hover the mouse over the simulator. You will see a red-bordered box. As we move the mouse pointer over an element, we can see its corresponding class object appearing in the terminal window (UIView:0xc5710c0) as seen in the following screenshot. Now click the mouse to select the object that you want to work on dynamically.

For More Information:
Try the following command on the terminal and observe the changes in the simulator:

```ruby
self returns the current object selected by the mouse.

(#<UIView:0x7652680>) > self
=> #<UIView:0x7652680>
```

Create an object `blue` for the `UIColor` class and assign the color blue to the variable as follows:

```ruby
(#<UIView:0x7652680>) > blue = UIColor.blueColor
```

To change the background color of the view, use the `backgroundColor` property of the selected view as follows:

```ruby
=> #<UICachedDeviceRGBColor:0xb05a800>

(#<UIView:0x7652680>) > self.backgroundColor = blue
=> #<UICachedDeviceRGBColor:0xb05a800>
```

Make sure that the background color on the simulator has been changed to blue as shown in the following screenshot:

Let's dismiss the alert box by clicking on any button and put a new alert box with the following code:

```ruby
a = UIAlertView.new
a.title = "My Title"
a.message = "Hello World!"
a.show
```

For More Information:

The simulator shows a new alert box on screen without compiling the code as shown in the following screenshot:

```ruby
(main)> a = UIAlertView.new
=> #<UIAlertView:0xce38c80>
(main)> a.title = 'My Title'
=> "My Title"
(main)> a.message = 'Hello World!'
=> "Hello World!"
(main)> a.show
=> #<UIAlertView:0xce38c80>
(main)> []
```

You can dismiss the alert box as follows:

```ruby
(main) > a.dismiss
```

We can see how REPL is a great tool for developing applications for iOS and how it helps us make changes dynamically. To make these changes permanent we need to add the same code to our source code.

**Debugger – catch your mistakes!**

A typical debugger provides the ability to halt when specific conditions are encountered. It also offers sophisticated functions, such as running a program step by step, breaking or pausing the program for an examination based on breakpoints, and tracking the values of the variables at that state. RubyMotion Version 1.24 and above support debugging using GDB: the GNU project debugger ([http://www.gnu.org/software/gdb/](http://www.gnu.org/software/gdb/)).
The RubyMotion debugger provides the following inbuilt debugging facilities:

- It stops the program at a specific line
- It examines the problem when the program has stopped
- It checks the value for the variables at a specific breakpoint

The RubyMotion compiler implements the **DWARF** debugging format's metadata for the Ruby language. This allows external programs, such as the debugger in our case, to retrieve source-level information about the RubyMotion application. The metadata is saved under a `.dSYM` bundle file at the same level as the `.app` bundle in the build directory of your project.

---

**How to start debugging**

There are three ways in which we can start the debugger.

**While testing on a simulator**

We can start the debugger with a simulator. The debugger will directly attach itself to the app and replace the interactive shell (REPL).

To start, just type:

```
$rake simulator debug=1
```

**While testing on a device**

We can start debugging with the device running simultaneously. The build system will start the iOS debugging server on the device and then remotely attach the debugger on your shell right after the application has been deployed on the device.

```
$rake device debug=1
```

In the release mode, local variables might not be accessible in the debugger as they are optimized to fit into CPU registers.

To test your application on a device, you are required to enroll for the **Apple Developer Program**. We will discuss this in detail in later chapters.

---

**For More Information:**

**Entering commands before starting**

We might need some breakpoint before loading the application; we can do this as follows:

```
rake debug=1 no_continue=1
```

On execution of this command, the GDB will start and we will be able to set the breakpoints. This is discussed in more detail in the next section.

**Breakpoint**

We can put breakpoints at a specific location of our application code using the `break` command and then pass the location where the debugger should stop the execution of the code using the `file_name:line_number` notation.

Let's try putting a breakpoint in our current application. To do so, we need to start our `HelloWorld` application in debugging mode as follows:

```
rake simulator debug=1
/Library/RubyMotion/lib/motion/project/config.rb:89:
  Build ./build/iPhoneSimulator-5.1-Development
  Simulate ./build/iPhoneSimulator-5.1-Development/HelloWorld.app
Attaching to process 86665.
Reading symbols for shared libraries . done
0x8fe6c030 in __dyld__dyld_start ()
Function "rb_exc_raise" not defined.
Breakpoint 1 (rb_exc_raise) pending.
Function "malloc_error_break" not defined.
Breakpoint 2 (malloc_error_break) pending.
Reading symbols for shared libraries ....................................
........................................................................
. done
Breakpoint 1 at 0x37136
Pending breakpoint 1 - "rb_exc_raise" resolved
Breakpoint 2 at 0x97bdec97
Pending breakpoint 2 - "malloc_error_break" resolved
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries ... done
(gdb)
```
Now let's set a breakpoint on the eighth line of the file `app_delegate.rb` as follows:

```bash
(gdb) break app_delegate.rb:8
Breakpoint 3 at 0x800085: file app_delegate.rb, line 8
```

With the preceding command, the execution of your application will halt at line number 8 of the `app_delegate.rb` file.

**Listing breakpoints**

To list the breakpoints that have been set up in the current debugging environment, we use the `info breakpoint` command as follows:

```bash
(gdb) info breakpoint
```

```
Num Type          Disp Enb Address            What
1   breakpoint    keep y   0x000adff6 <rb_exc_raise+6>
2   breakpoint    keep y   0x97bdec97 <malloc_error_break+6>
3   breakpoint    keep y   0x00080085 in rb_scope__application:didFinishLaunchingWithOptions:__ at app_delegate.rb:8
```

We can see that the list of breakpoints created in the last section can also be seen in the list.

**Moving between the different breakpoints**

The `continue` command will continue the execution of the program until it reaches the next breakpoint.

```bash
(gdb) continue
```

We can also use its alias `c` as follows; it is more handy to use:

```bash
(gdb) c
```

```
Breakpoint 3, rb_scope__application:didFinishLaunchingWithOptions:__
  (self=0x9408440, application=0x9401750, launchOptions=0x4) at app_delegate.rb:8
8    alert.show
```

The `next` command will continue the execution of the program until the next source-level location. This is usually the very next line in the Ruby source code. You should have a look at the terminal for the relevant source code line.

```bash
(gdb) next
```
Checking the value of a local variable

This is an important feature of debugging, to check the value of a variable at a specific breakpoint.

(gdb) pro alert
#<UIAlertView:0x944b9b0>

This shows that the alert is an object of the UIAlertView class

**Pro (print-ruby-object)** accepts two parameters as follows:

- The object on which the variable will be retrieved.
- The variable name that you want to get.

To check the variables available for us to execute, run the following command:

```bash
$info locals
```

Checking the value of an instance variable

We can also check the value of an instance variable during some breakpoint using **pri (print-ruby-ivar)** as follows:

```bash
pri self "@tweet"
```

**pri** accepts two commands as follows:

- The object on which the instance variable will be retrieved.
- The instance variable that you want to get. Make sure to include the @ character in the name.

You can use **pri @tweet** instead of **pri self @tweet**.

Disable breakpoint

To disable a breakpoint, use **disable** followed by the breakpoint number; it has to be disabled as follows:

(gdb) disable 3

For More Information:

Exit debugger
Type quit to exit the debugger as follows:

(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from process 6792.

Summary
Let's recap what we have done in this chapter:

• Created a simple RubyMotion application
• Discussed the basic RubyMotion application structure
• Explored the commands available with RubyMotion
• Performed different Rake tasks with RubyMotion
• Learned how to configure your RubyMotion application
• Worked with the interactive console—REPL
• Debugged your application using the RubyMotion debugger

In the next chapter, we turn our attention to RubyMotion data type objects—such as strings and arrays. We will also learn how to interface with C and we will focus on memory management in RubyMotion.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.