

Introduction To Objective-C

CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 13 — 10/09/2012

Goals of the Lecture

- Present an introduction to Objective-C
 - As implemented by the Apple LLVM Compiler 4.0 (a.k.a. Clang)
- Coverage of the language will be INCOMPLETE
 - We'll see the basics... there is a lot more to learn
- There is a nice Objective-C tutorial located here:
 - http://cocoadevcentral.com/d/learn_objectivec/
- although everything it says about memory management is now obsolete (!)

History (I)

- Brad Cox created Objective-C in the early 1980s
 - It was his attempt to add object-oriented programming concepts to the C programming language
 - NeXT Computer licensed the language in 1988;
 - it was used to develop the NeXTSTEP operating system, programming libraries and applications for NeXT
- In 1993, NeXT worked with Sun to create OpenStep, an open specification of NeXTSTEP on Sun hardware

History (II)

- In 1997, Apple purchased NeXT and transformed NeXTSTEP into MacOS X which was first released in the summer of 2000
 - Objective-C has been one of the primary ways to develop applications for OS X for the past 12 years
- In 2008, it became the primary way to develop applications for iOS targeting (currently) the iPhone and the iPad and (perhaps in the future) the Apple TV

Objective-C is “C plus Objects” (I)

- Objective-C makes a small set of extensions to C which turn it into an object-oriented language
- It is used with two object-oriented frameworks
 - The Foundation framework contains classes for basic concepts such as strings, arrays and other data structures and provides classes to interact with the underlying operating system
 - The AppKit contains classes for developing applications and for creating windows, buttons and other widgets

Objective-C is “C plus Objects” (II)

- Together, Foundation and AppKit are called Cocoa
- On iOS, AppKit is replaced by UIKit
 - Foundation and UIKit are called Cocoa touch
- In this lecture, we focus on the Objective-C language,
 - we’ll see a few examples of the Foundation framework
 - we’ll see examples of UIKit in Lecture 16 when I introduce the iOS framework

C Skills? Highly relevant

- Since Objective-C is “C plus objects” any skills you have in the C language directly apply
 - statements, data types, structs, functions, etc.
- What the OO additions do, is reduce your need on
 - structs, malloc, and dealloc
 - indeed with automatic reference counting, memory management is no longer a primary concern
 - and enable all of the object-oriented concepts we’ve been discussing
- Objective-C and C code otherwise freely intermix

Development Tools (I)

- Apple's XCode is used to develop in Objective-C
 - Behind the scenes, XCode makes use of Apple's Clang compiler to compile Objective-C programs
 - LLVM is a virtual machine targeted by Clang)
- It represents a single unified environment to create both the code and the UI for OS X and iOS apps

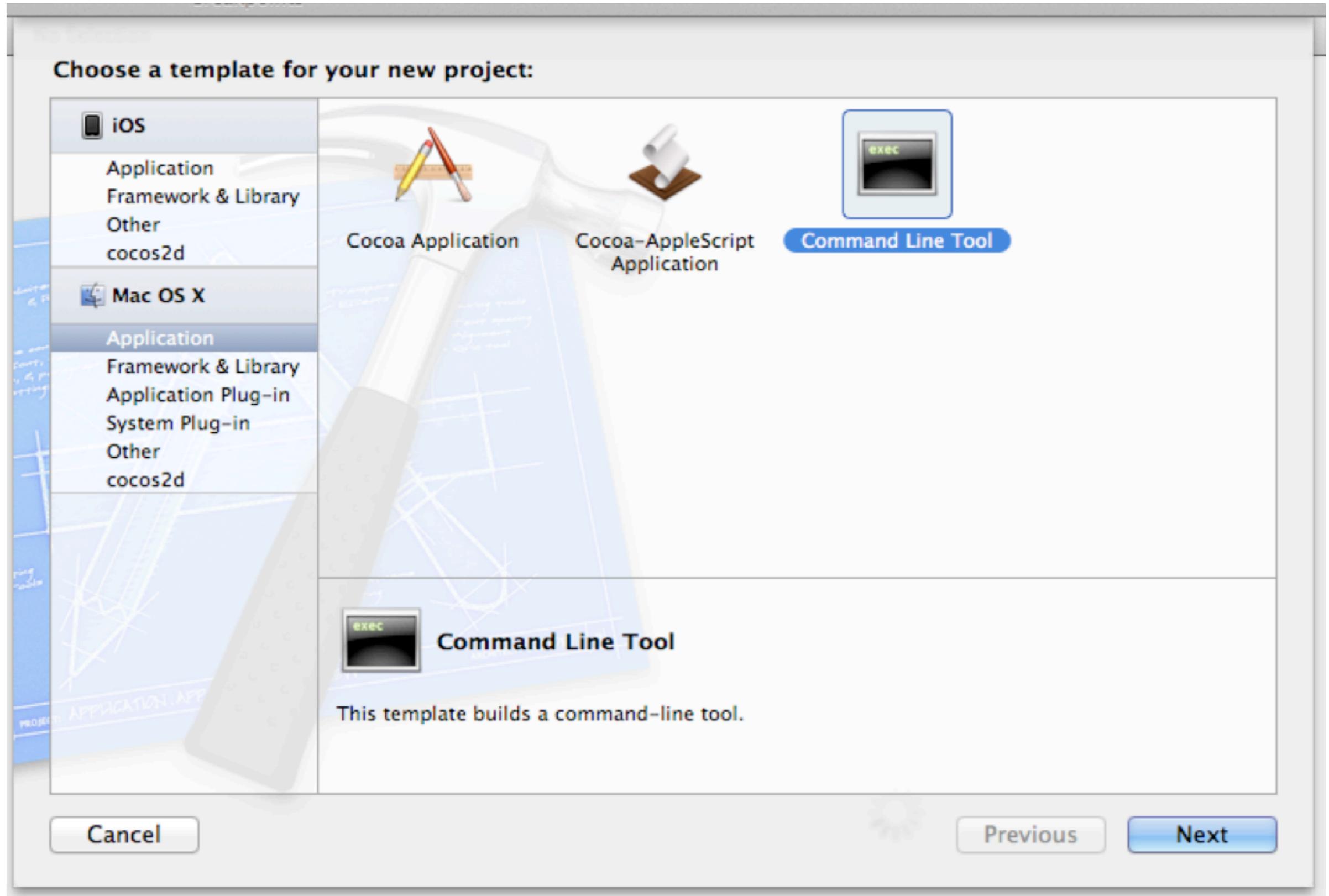
Development Tools (II)

- XCode is available on the Mac App Store
 - It is “free” for users of OS X Lion and OS X Mountain Lion
 - Mountain Lion costs \$20
- Clicking Install in the App Store downloads XCode to Applications
 - On first launch, XCode will then download additional libraries that it needs
 - You can also go into the preferences and install its command-line tools to gain access to the clang and gcc compilers in the terminal

Hello World

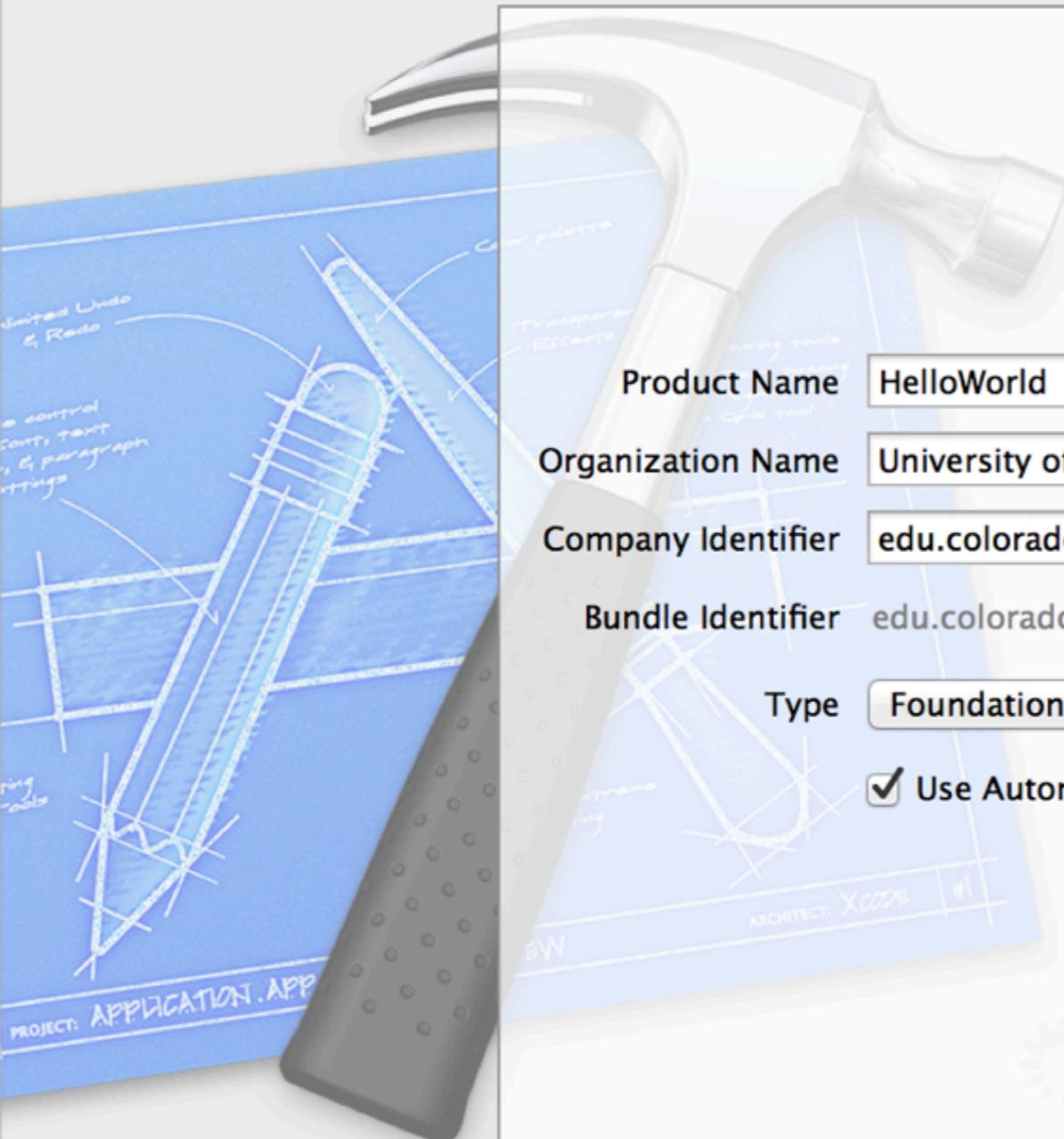
- As is traditional, let's look at our first objective-c program via the traditional Hello World example
- To create it, we launch XCode and create a New Project
 - select Application under the MacOS X
 - select Command Line Tool on the right (click Next)
 - select Foundation and type "Hello World" (click Next)
 - select a directory, select checkbox for git (click Finish)

Step One



Step Two

Choose options for your new project:



Product Name

Organization Name

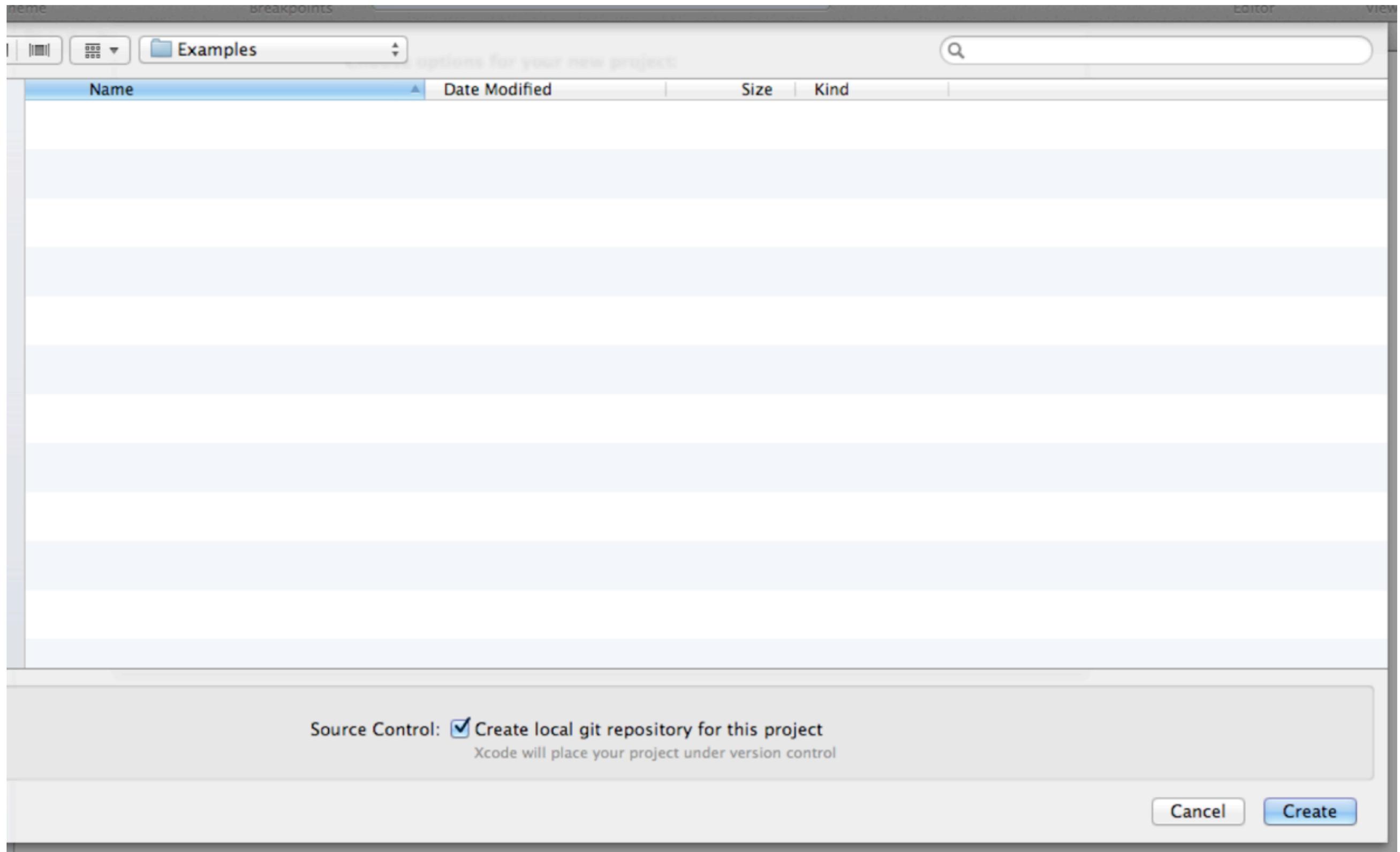
Company Identifier

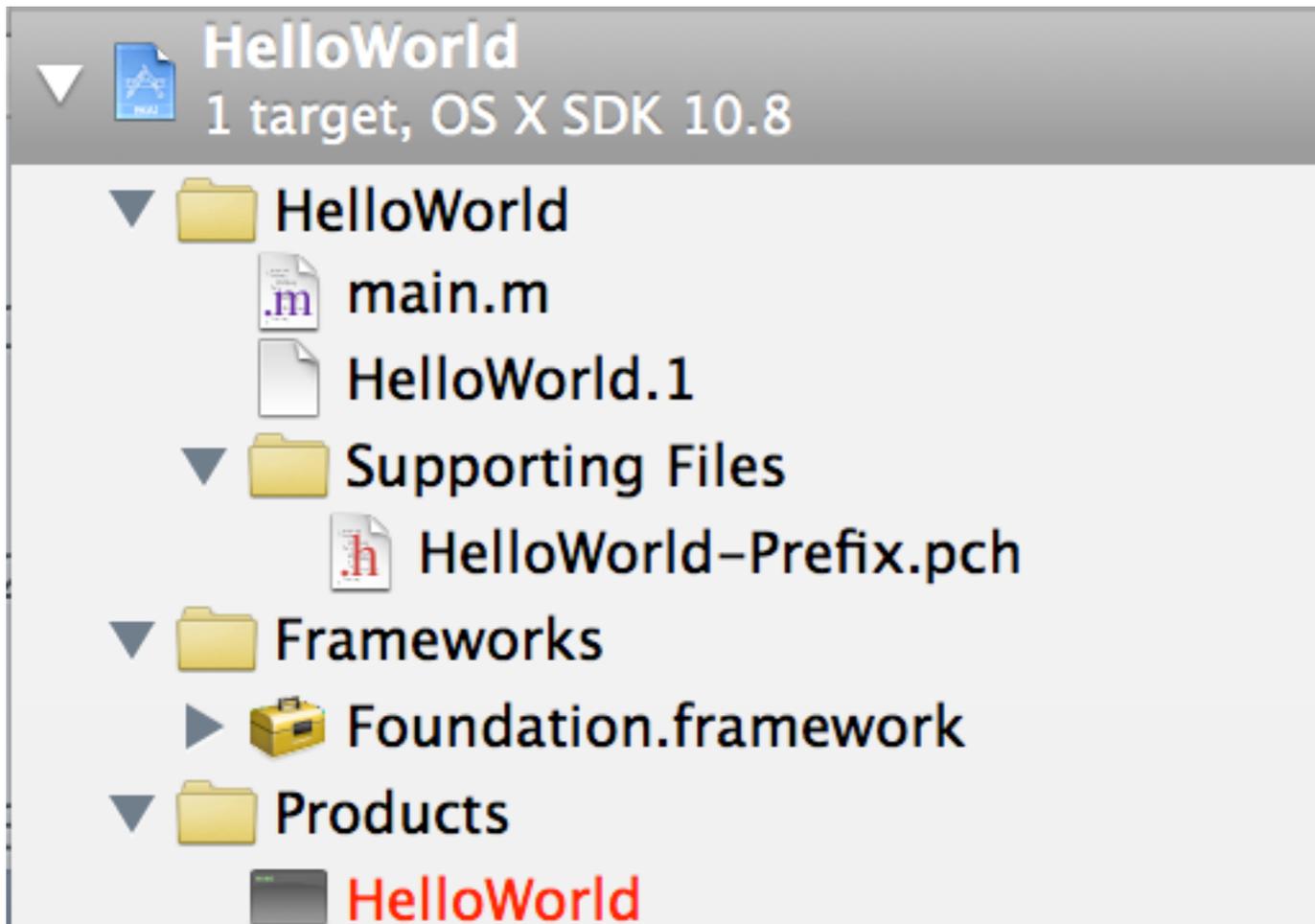
Bundle Identifier

Type

Use Automatic Reference Counting

Step Three

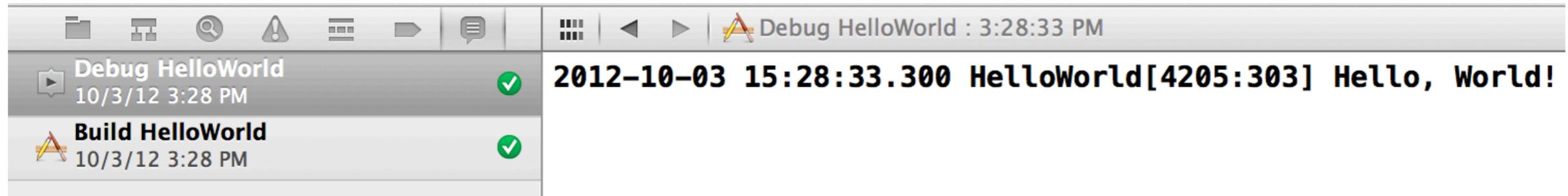




Similar to what we saw with Eclipse, XCode creates a default project for us;

There are folders for this program's source code (main.m), frameworks, and products (the application itself)

Note: the Foundation framework is front and center and HelloWorld is shown in red because it hasn't been created yet



The template is ready to run; clicking “Run” brings up a console that shows “Hello, World!” being displayed;

Exciting, isn't it?

One thing to note is that this console is also where the debugger will display its controls if a breakpoint is encountered at run-time.

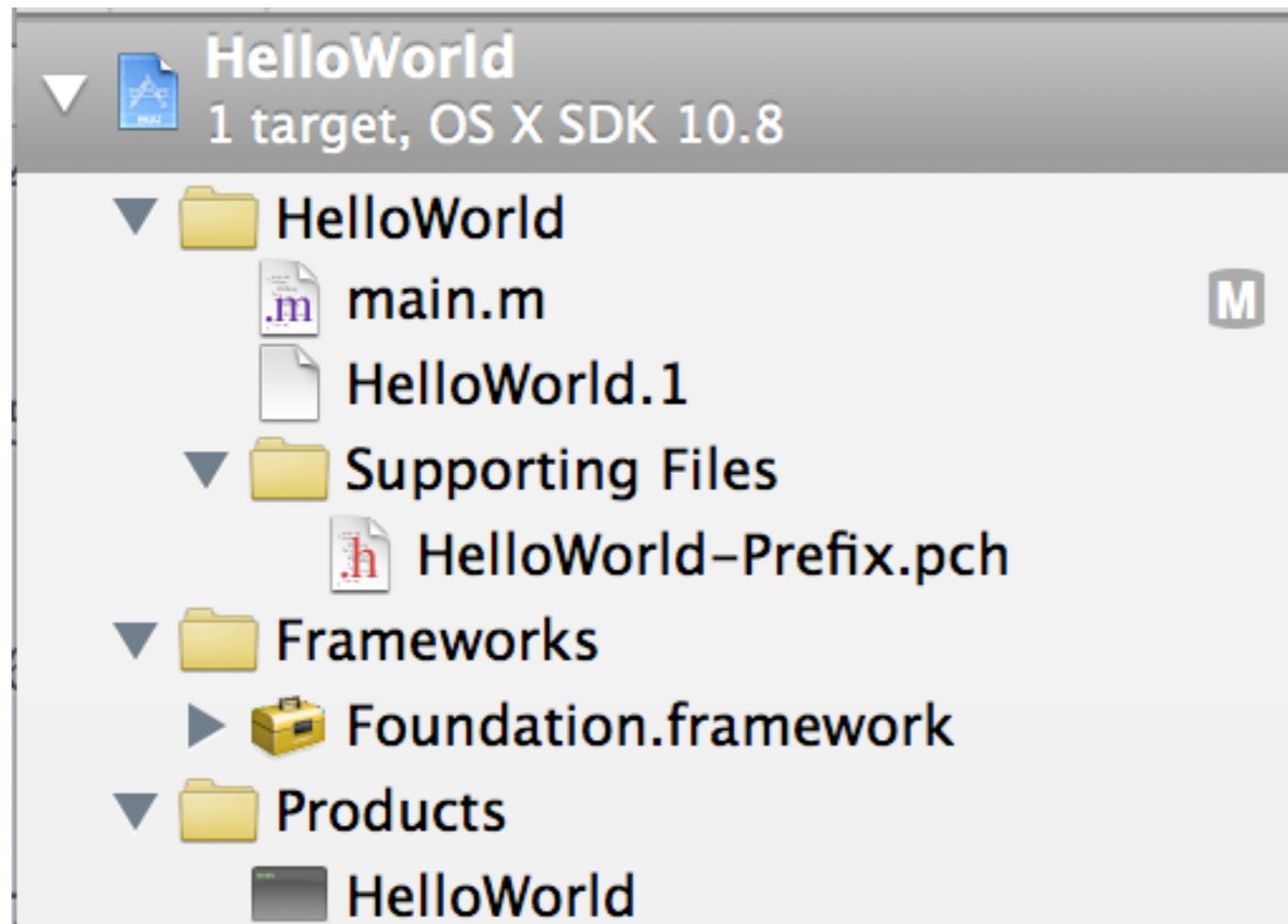
XCode previously made use of gcc and gdb by default for compiling and debugging programs (respectively). Apple has now switched to using clang and the clang debugger by default.

```
9 #import <Foundation/Foundation.h>
10
11 int main(int argc, const char * argv[])
12 {
13     @autoreleasepool {
14         NSLog(@"Hello, World!");
15     }
16     return 0;
17 }
18
```

The code for the program is not too surprising. It's a C function with two Objective-C extensions. The “@” sign is usually a good indication that you're looking at an Objective-C extension to the core C programming language.

@autoreleasepool is a no-op since we turned automatic reference counting on when we created the project. In the past, it played an important role in memory management. For now, you can ignore it.

NSLog() is a C function. It acts like printf, but adds information (timestamp and process info) to its output.

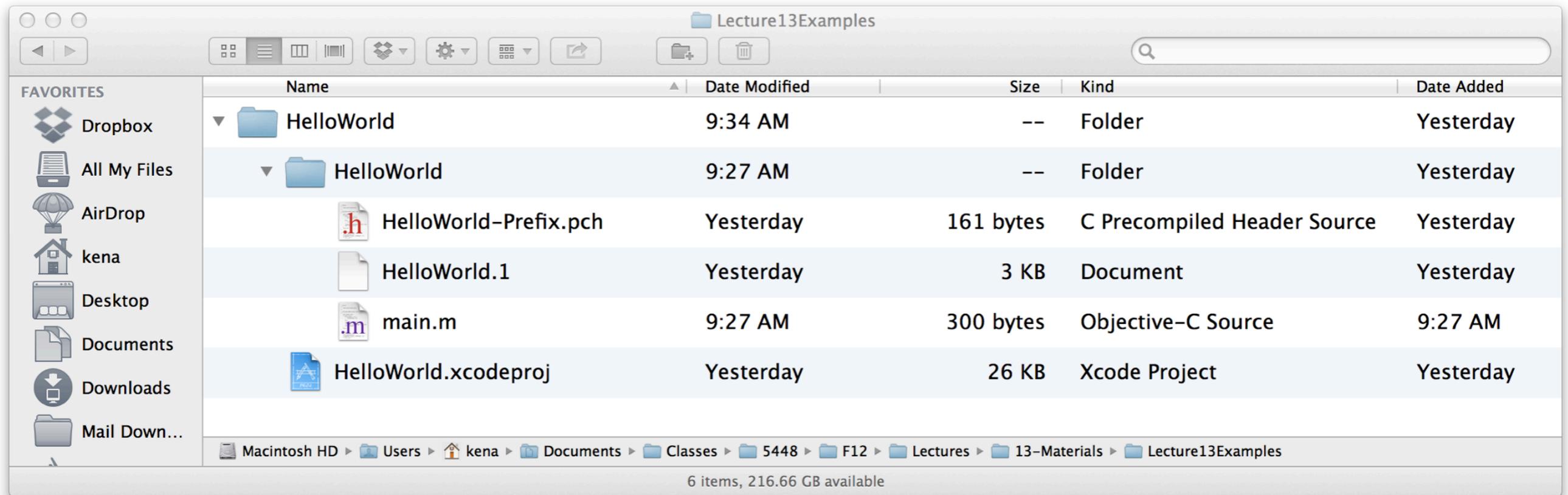


Note: after we built and ran the program, the HelloWorld product is no longer listed in red. It has actually been built and lives (somewhere) in the file system.

(We'll find out where in a moment.)

Also notice that an “M” has appeared next to main.m after I edited it to get rid of an auto-generated comment and to reduce the amount of whitespace used by the function. (I'll show these edits during my demo.)

The M is an aspect of XCode's git integration. It's telling me that the file was modified and that (eventually) I can check these changes into the git repository that XCode automatically created for me. I won't be discussing the git integration further; I just wanted you to know it was there.



The resulting project structure on disk does not map completely to what is shown in Xcode; The source file, man page, and pre-compiled header file are all stored in a sub-directory of the main directory.

The project file HelloWorld.xcodeproj is stored in the main directory. It is the file that keeps track of all project settings and the location of project files.

XCode project directories are a lot simpler now that files generated during a build are stored elsewhere.

Where is the actual application?

- After you ran the application, HelloWorld switched from being displayed in red to being displayed in black
 - You can right click on HelloWorld and select “Show in Finder” to see where XCode placed the actual executable
- By default, XCode creates a directory for your project in
 - ~/Library/Developer/XCode/DerivedData
- For HelloWorld, XCode generated 22 directories containing 38 files!
 - This is where XCode stores all the information it needs to recompile your project as fast as possible (pre-compiled headers, etc.) as well as where it stores it’s log files, full-text index (for fast search), and the like

```
Resolution:Debug $ pwd
/Users/kena/Library/Developer/Xcode/DerivedData/HelloWorld-flzyhuxjcsbwegesmxduvpjjldam/Build/Products/Debug
Resolution:Debug $ ls
HelloWorld*
Resolution:Debug $ ./HelloWorld
2012-10-04 09:42:47.146 HelloWorld[6151:707] Hello, World!
Resolution:Debug $
```

The resulting executable can be executed from the command line, fulfilling the promise that we were creating a command-line tool

Note the “2012-10-04 09:42:47.146 HelloWorld[6151:707]” is generated by NSLog()

```
9 #import <Foundation/Foundation.h>
10
11 int main(int argc, const char * argv[])
12 {
13     @autoreleasepool {
14         NSLog(@"Hello, World!");
15     }
16     return 0;
17 }
18
```

Returning to the code, we can see that:

Objective-C programs start with a function called main, just like C programs.

#import is similar to C's #include except it ensures that header files are included once and only once

Thus our program calls a function, NSLog, and returns 0

Let's add a breakpoint at the call to NSLog().

```
9  #import <Foundation/Foundation.h>
10
11  int main(int argc, const char * argv[])
12  {
13      @autoreleasepool {
14          NSLog(@"Hello, World!");
15      }
16      return 0;
17  }
```

We add a breakpoint, by clicking on the line numbers listed on the left hand side of the source code file.

Here, we have added a breakpoint to line 14.

To disable the breakpoint, click on it.

To delete the breakpoint, drag it off the numbers and let go.

```
9 #import <Foundation/Foundation.h>
10
11 int main(int argc, const char * argv[])
12 {
13     @autoreleasepool {
14         NSLog(@"Hello, World!");
15     }
16     return 0;
17 }
18
19
```

Thread 1: breakpoint 1.1

Auto ▾ | HelloWorld > Thread 1 > 0 main

Auto ▾

- argc = (int) 1
- argv = (const char **) 0x00007fff5fbff860
 - *argv = (const char *) 0x00007fff5fbff9b8 "/Users/kena/Library/Developer/Xcode/DerivedData/..."
 - **argv = (const char) '/'

All Output ▾ | Clear

```
(lldb) p argv
(const char **) $0 = 0x00007fff5fbff860
(lldb) p argc
(int) $1 = 1
(lldb) p argv[0]
(const char *) $2 = 0x00007fff5fbff9b8 "/Users/kena/Library/
Developer/Xcode/DerivedData/HelloWorld-
flzyhuxjcsbwegesmxduvpjjldam/Build/Products/Debug/HelloWorld"
(lldb)
```

Now, when we run the program, we stop at the breakpoint in the debugger (lldb). This debugger is similar to gdb. Here you can see that I examined the contents of local variables on the left and then printed those same values out in the debugger on the right. The (small) buttons on the left give you standard “step-by-step” debugging controls.

Let's add objects...

- Note: This example comes from “Learning Objective-C 2.0: A Hands-On Guide to Objective-C for Mac and iOS Developers” written by Robert Clair
 - It is an excellent book that I highly recommend
 - His review of the C language is an excellent bonus to the content on Objective-C itself
 - I also recommend “Objective-C Programming: The Big Nerd Ranch Guide” by Aaron Hillegass
- We're going to create an Objective-C class called Greeter to make this HelloWorld program a bit more object-oriented

First, we are going to add a class

- Select File ⇒ New File
- In the resulting Dialog (see next three slides)
 - Select Cocoa Class under Mac OS X
 - Select Objective-C class (click Next)
 - Name your new class “Greeter”
 - Select NSObject as your superclass (click Next)
 - Make sure file is called “Greeter.m” add to HelloWorld Group and HelloWorld Target. (Click Create.)

Step One

Choose a template for your new file:

iOS	 Objective-C class	 Objective-C category	 Objective-C class extension	 Objective-C protocol
Cocoa Touch C and C++ User Interface Core Data Resource Other cocos2d				
OS X	 Objective-C test case class			
Cocoa C and C++ User Interface Core Data Resource Other cocos2d				
	 Objective-C class			
	An Objective-C class, with implementation and header files.			

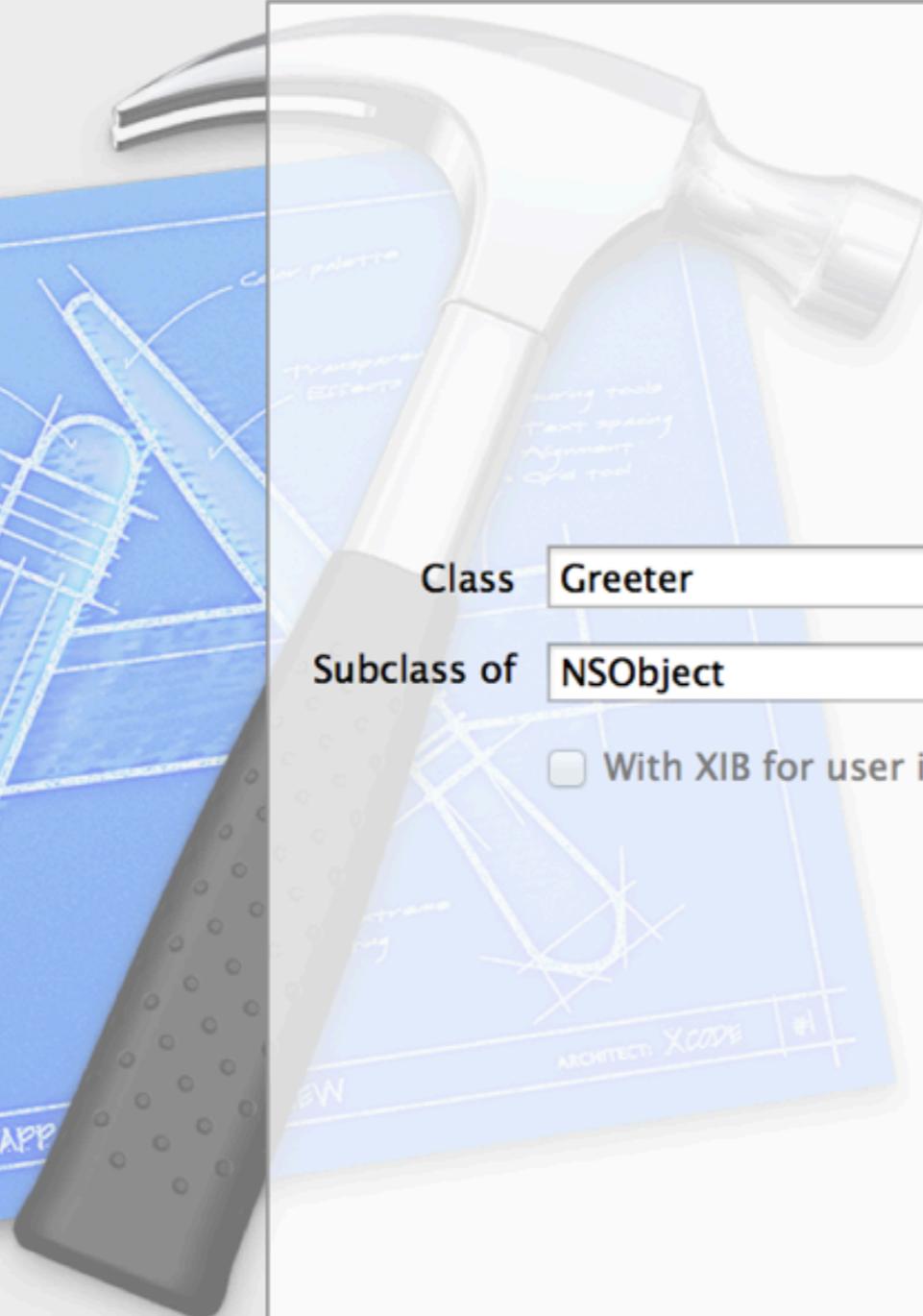
Cancel

Previous

Next

Step Two

Choose options for your new file:



Class

Subclass of

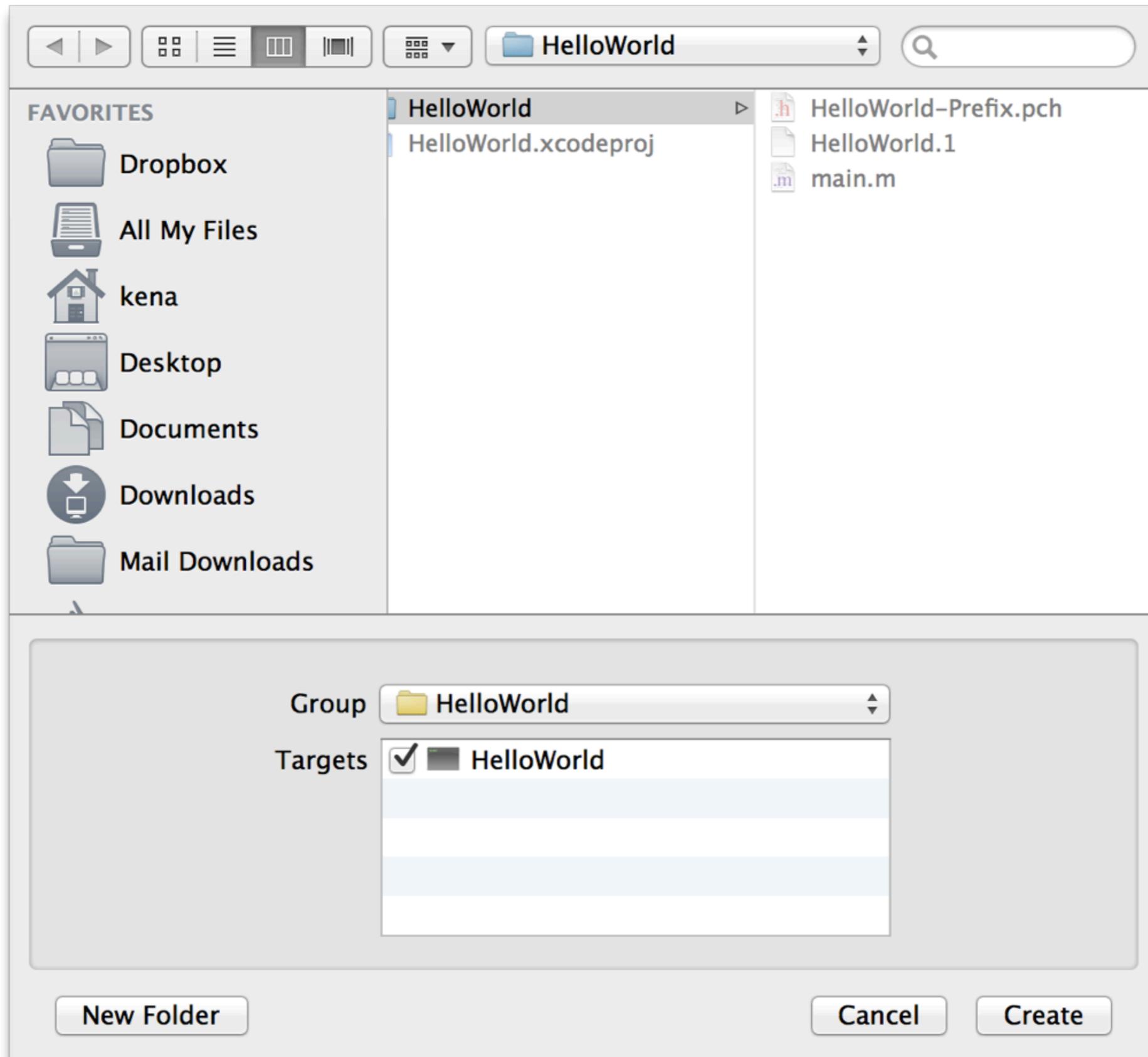
With XIB for user interface

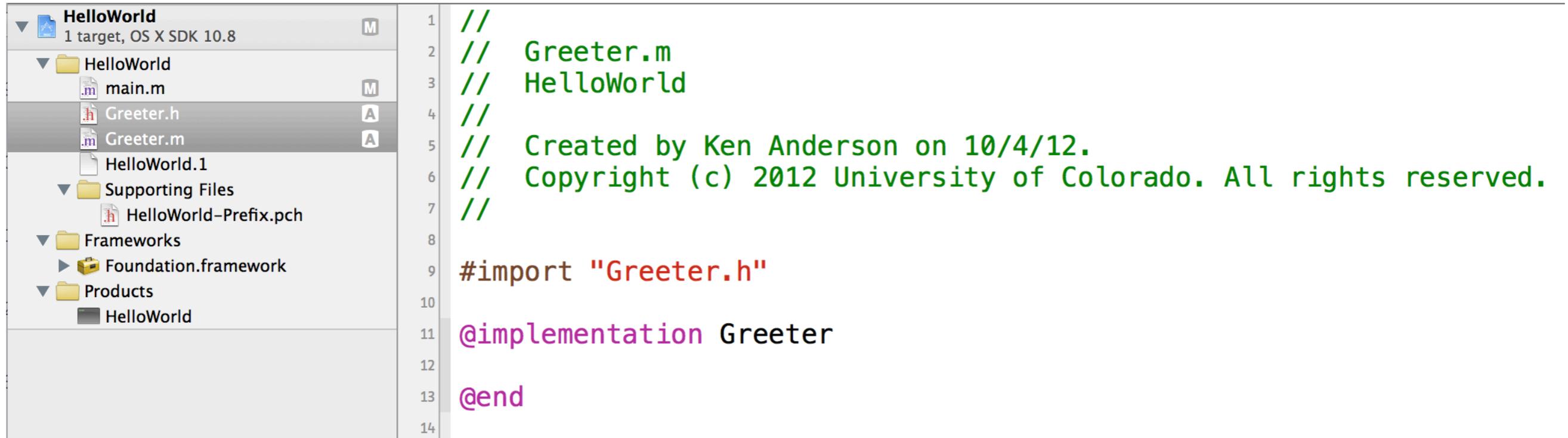
Cancel

Previous

Next

Step Three





Greeter.h and Greeter.m are added to our project. (The “A” next to their names is a “git annotation” meaning that git has detected the two new files.)

Objective-C classes are split between a header file and an implementation file. The “.m” suffix refers to the letter ‘m’ in ‘i**M**plementation’.

All method definitions will be inserted between the @implementation keyword and the @end keyword. The “@” symbol once again alerts us to the fact that we’re looking at Objective-C language constructs.

Objective-C classes

- Classes in Objective-C are defined in two files
 - A header file which defines the properties (instance variables on steroids) and method signatures of the class
 - We'll discuss properties in more details soon
 - An implementation file (.m) that provides the method bodies

Header Files

- The header file of an Objective-C class traditionally has the following structure

```
<import statements>
```

```
@interface <classname> : <superclass name> {
```

```
    <attribute definitions>
```

```
}
```

```
<method signature definitions>
```

```
@end
```

Header Files

- With more recent versions of Objective-C (sometimes called Objective-C 2.0), the structure has changed to the following

```
<import statements>
```

```
@interface <classname> : <superclass name>
```

```
    <property definitions>
```

```
    <method signature definitions>
```

```
@end
```

- Note: the previous structure is still supported

What's the difference?

- In Objective-C 2.0, the need for defining the attributes of a class has been greatly reduced due to the addition of properties
 - When you declare a property, you automatically get
 - an attribute (i.e. instance variable)
 - a getter method
 - and a setter method
 - synthesized (automatically added) for you

New Style

- In this class, I'll be using the new style promoted by Objective-C 2.0
 - Occasionally we may run into code that uses the old style, I'll explain the old style when we encounter it

Objective-C additions to C (I)

- Besides the very useful `#import`, the best way to spot an addition to C by Objective-C is the presence of this symbol
 - `@`

Objective-C additions to C (II)

- In header files, the two key additions from Objective-C are
 - @interface
- and
 - @end
- @interface is used to define a new objective-c class
 - As we saw, you provide the class name and its superclass; Objective-C is a single inheritance language
- @end does what it says, ending the @interface compiler directive

Greeter's interface (I)

```
9  #import <Foundation/Foundation.h>
10
11  @interface Greeter : NSObject
12
13  @property (nonatomic, copy) NSString* greeting;
14
15  - (void) greet;
16
17  @end
```

We've added one property called `greeting`. Its type is “NSString *” which can be read as “an instance of NSString” or “a pointer to an NSString”

We've added one method called “`greet`”. It has no parameters and its return type is “void”.

“NS” refers to NextStep; NeXT lives on!

Objective-C Properties (I)

- An Objective-C property helps to define the public interface of an Objective-C class
 - It defines an instance variable, a getter and a setter all in one go

```
@property (nonatomic, copy) NSString* greeting;
```

- “nonatomic” tells the runtime that this property will never be accessed by more than one thread (use “atomic” otherwise)
- “copy” tells the automatic reference counting system that when we get a new value for this property that we should store a copy of it and that when we return this value, we should return a copy
 - this protects our instance of the string from manipulation by other objects

Objective-C Properties (II)

- `@property (nonatomic, copy) NSString* greeting;`
- After the property attributes (in this case `nonatomic` and `copy`), the type of the property is specified and finally the property's name
 - A property can be of any C or Objective-C type
 - although they are primarily used with Objective-C classes and (sometimes) primitive types such as `int`, `long`, and the like

Objective-C Properties (III)

- `@property (nonatomic, copy) NSString* greeting;`
- If you have an instance of Greeter
 - `Greeter* ken = [[Greeter alloc] init];`
- You can assign the property using dot notation
 - `ken.greeting = @"Say Hello, Ken";`
- You can retrieve the property also using dot notation
 - `NSString* theGreeting = ken.greeting;`

Objective-C Properties (IV)

- Dot notation is simply “syntactic sugar” for calling the automatically generated getter and setter methods
 - `NSString* theGreeting = ken.greeting;`
- is equivalent to
 - `NSString* theGreeting = [ken greeting];`
- The above is a call to a method that is defined as
 - `- (NSString*) greeting;`

Objective-C Properties (V)

- Dot notation is simply “syntactic sugar” for calling the automatically generated getter and setter methods
 - `ken.greeting = @"Say Hello, Ken";`
- is equivalent to
 - `[ken setGreeting:@"Say Hello, Ken"];`
- The above is a call to a method that is defined as
 - `- (void) setGreeting:(NSString*) newGreeting;`

Objective-C Methods (I)

- It takes a while to get use to Object-C method signatures
 - - (void) setGreeting: (NSString*) newGreeting;
- defines an instance method (-) called setGreeting:
- The colon signifies that the method has one parameter and is PART OF THE METHOD NAME
- In this case, the parameter's name is
 - newGreeting
- and it's type is NSString*
- Thus, the method names setGreeting: and setGreeting refer to TWO different methods
 - the former has one parameter; the latter has no parameters

Objective-C Methods (II)

- A method with multiple parameters will have multiple colon characters and the parameter definitions are interspersed with the method name
 - - (void) setStrokeColor: (NSColor*) strokeColor
 - andFillColor: (NSColor*) fillColor;
- The above signature defines a method whose name is
 - setStrokeColor:andFillColor:
- The two parameters are called
 - strokeColor and fillColor
- and they are both of type NSColor*

NSString * and NSColor *

- We've now seen examples of types
 - NSString * and NSColor *
- What does this mean?
 - The * in C means “pointer”
 - Thus, this can be read as
 - “pointer to <class>”
 - it simply means an instance has been allocated and we have a pointer to the instance

Let's implement the method bodies

- The implementation file of a class looks like this

```
<import statements>
```

```
<optional class extension>
```

```
@implementation <classname>
```

```
    <method body definitions>
```

```
@end
```

- Let's ignore the “optional class extension” part for now

Greeter's implementation (I)

```
9  #import "Greeter.h"
10
11  @implementation Greeter
12
13  @synthesize greeting=_greeting;
14
15  - (id) init {
16      self = [super init];
17      if (self) {
18          self.greeting = @"Howdy!!";
19      }
20      return self;
21  }
22
23  - (void) greet {
24      NSLog(@"%@", self.greeting);
25  }
26
27  @end
```

High-Level Overview

1. #import of header file
2. Creation of property via the @synthesize statement
3. Definition of constructor (init)
4. Implementation of the greet method

Greeter's implementation (I)

```
9  #import "Greeter.h"
10
11  @implementation Greeter
12
13  @synthesize greeting=_greeting;
14
15  - (id) init {
16      self = [super init];
17      if (self) {
18          self.greeting = @"Howdy!!";
19      }
20      return self;
21  }
22
23  - (void) greet {
24      NSLog(@"%@", self.greeting);
25  }
26
27  @end
```

High-Level Overview

1. #import of header file
2. Creation of property via the @synthesize statement
3. Definition of constructor (init)
4. Implementation of the greet method

Greeter's implementation (II)

```
9  #import "Greeter.h"
10
11  @implementation Greeter
12
13  @synthesize greeting=_greeting;
14
15  - (id) init {
16      self = [super init];
17      if (self) {
18          self.greeting = @"Howdy!!";
19      }
20      return self;
21  }
22
23  - (void) greet {
24      NSLog(@"%@", self.greeting);
25  }
26
27  @end
```

The @synthesize statement causes the compiler to:

1. Create an instance variable called `_greeting` of the appropriate type
2. Generate the setter method called **setGreeting:**
3. Generate the getter method called **greeting**

We prefix the instance variable name with an underscore to help distinguish it from the property name

Greeter's implementation (III)

```
9  #import "Greeter.h"
10
11  @implementation Greeter
12
13  @synthesize greeting=_greeting;
14
15  - (id) init {
16      self = [super init];
17      if (self) {
18          self.greeting = @"Howdy!!";
19      }
20      return self;
21  }
22
23  - (void) greet {
24      NSLog(@"%@", self.greeting);
25  }
26
27  @end
```

Objective-C constructors follow a standard pattern.

1. We call the constructor of the superclass.
2. We make sure we get back a valid (non-zero) object.
3. If so, we initialize our properties.
4. We then return the allocated object.

If the call to [super init] fails, it returns “nil” which is what we then return

Greeter's implementation (IV)

```
9  #import "Greeter.h"
10
11  @implementation Greeter
12
13  @synthesize greeting=_greeting;
14
15  - (id) init {
16      self = [super init];
17      if (self) {
18          self.greeting = @"Howdy!!";
19      }
20      return self;
21  }
22
23  - (void) greet {
24      NSLog(@"%@", self.greeting);
25  }
26
27  @end
```

The implementation of the greet method is straightforward.

We simply print our greeting to standard out using the NSLog() function.

NSLog() is similar to C's printf() function. It can take any number of arguments, one for each placeholder in its format string.

%@ means "Objective-C object"; The specific object has its description method called. This method is similar to Java's toString() method.

Note: %s, %d, %f, etc. are all supported by NSLog().

Calling methods (I)

- The method invocation syntax of Objective-C is
 - `[object method:arg1 method:arg2 ...];`
- Method calls are enclosed by square brackets
 - Inside the brackets, you list the object being called (the receiver)
 - Then the method name plugging in any arguments required by the methods parameters
 - similar to the method definition, you split on the “:” in the method name in order to deliver the parameter in-line

Calling Methods (II)

- Here's a call using the setter method for the greeting property; @"Howdy!" is a shorthand syntax for creating an NSString instance
 - `[greeter setGreeting: @"Howdy!"];`
- Here's a call to the same method where we get the greeting from some other Greeter object
 - `[greeterOne setGreeting: [greeterTwo greeting]];`
- Above we nested one call inside another; now a call with multiple args
 - `[rectangle setStrokeColor: [NSColor red] andFillColor: [NSColor green]];`

Memory Management (I)

- Memory management of Objective-C objects used to involve the use of six methods
 - alloc, init, dealloc, retain, release, autorelease
- In Objective-C 2.0, a garbage collector was added which turned retain, release, and autorelease into no-ops
 - The garbage collector could be used on the Mac but not on iOS
- In the most recent versions of Objective-C and XCode, the garbage collector has gone the way of the Dodo
 - (alas, Garbage Collector, we hardly knew ye!)
- and has been replaced with automatic reference counting (it's more efficient)
 - which does everything for you behind the scenes!

Memory Management (II)

- That said, you still need to
 - define your constructors
 - you have to show how to initialize your properties
 - and init is a good place to do other types of initialization
 - such as establishing database connections or registering interest in a notification, etc.
 - occasionally override dealloc
 - not to handle memory-management concerns (which ARC does for you)
 - but to handle things like closing database connections and deregistering interest in a notification, etc.

A new main method

- We now need a new version of main to make use of our new Greeter class
 - We'll import its header file
 - We'll instantiate an instance of the class
 - We'll call its greet method (to confirm that the constructor worked)
 - We'll set its greeting property to something different
 - We'll call its greet method again
- We don't have to worry about deallocating the greeter object.
 - ARC will take care of that for us

```
9 #import <Foundation/Foundation.h>
10 #import "Greeter.h"
11
12 int main(int argc, const char * argv[])
13 {
14     @autoreleasepool {
15         Greeter* greeter = [[Greeter alloc] init];
16         [greeter greet];
17         greeter.greeting = @"Hello from Objective-C!!";
18         [greeter greet];
19     }
20     return 0;
21 }
```

```
2012-10-04 11:23:17.649 HelloWorld[6719:303] Howdy!!
2012-10-04 11:23:17.651 HelloWorld[6719:303] Hello from Objective-C!!
```

Some things not (yet) discussed

- Objective-C has a few additions to C not yet discussed
 - The type `id`: `id` is defined as a pointer to an object
 - `id iCanPointAtAString = @"Hello";`
 - Note: no need for an asterisk in this case
 - The keyword `nil`: `nil` is a pointer to no object
 - It is similar to Java's `null`
 - The type `BOOL`: `BOOL` is a boolean type with values `YES` and `NO`; used throughout the Cocoa frameworks

Wrapping Up (I)

- Basic introduction to Objective-C
 - main methods
 - class and method definition and implementation
 - method calling syntax
 - creation and use of objects
- More to come as we use this knowledge to explore the iOS platform in future lectures

Coming Up Next

- Lecture 14: Review for Midterm
- Lecture 15: Midterm
- Lecture 16: Introduction to iOS