Introduction To Objective-C

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 12 — 09/29/2011

Goals of the Lecture

- Present an introduction to Objective-C 2.0
- 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/

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 MacOS for the past 11 years
- In 2008, it became the primary way to develop applications for iOS targeting (currently) the iPhone and the iPad and (soon, I'm guessing) 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 13

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, dealloc and the like
  - 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 either gcc or Apple's own LLVM to compile Objective-C programs
- The latest version of Xcode, Xcode 4, has integrated functionality that previously existed in a separate application, known as Interface Builder
  - We'll see examples of that integration next week

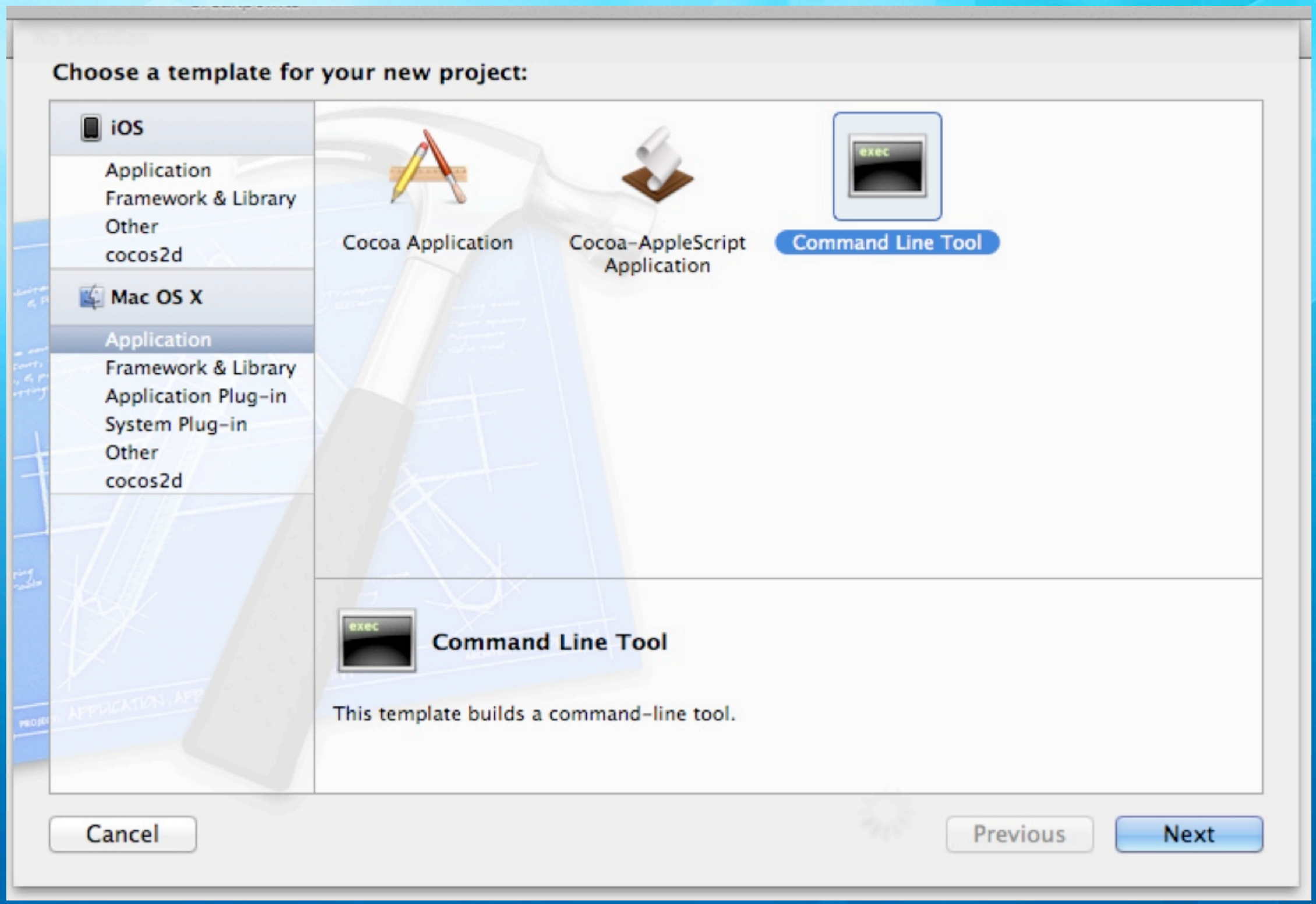
Development Tools (II)

- XCode is available on the Mac App Store
  - It is free for users of OS X Lion
  - Otherwise, I believe it costs $5 for previous versions of OS X
- Clicking **Install** in the App Store downloads a program called "Install XCode". You then run that program to get XCode installed

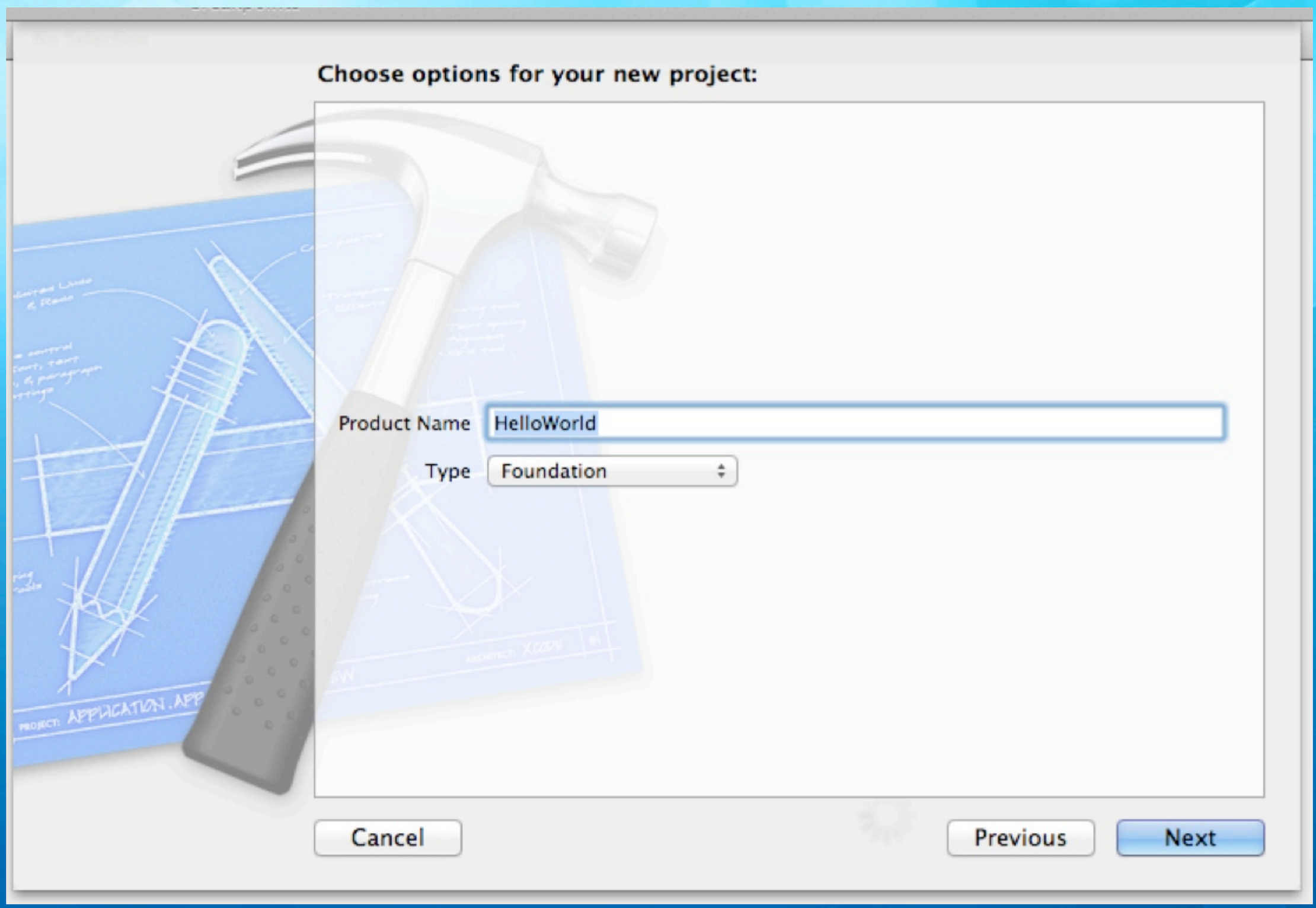
Hello World

- 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)

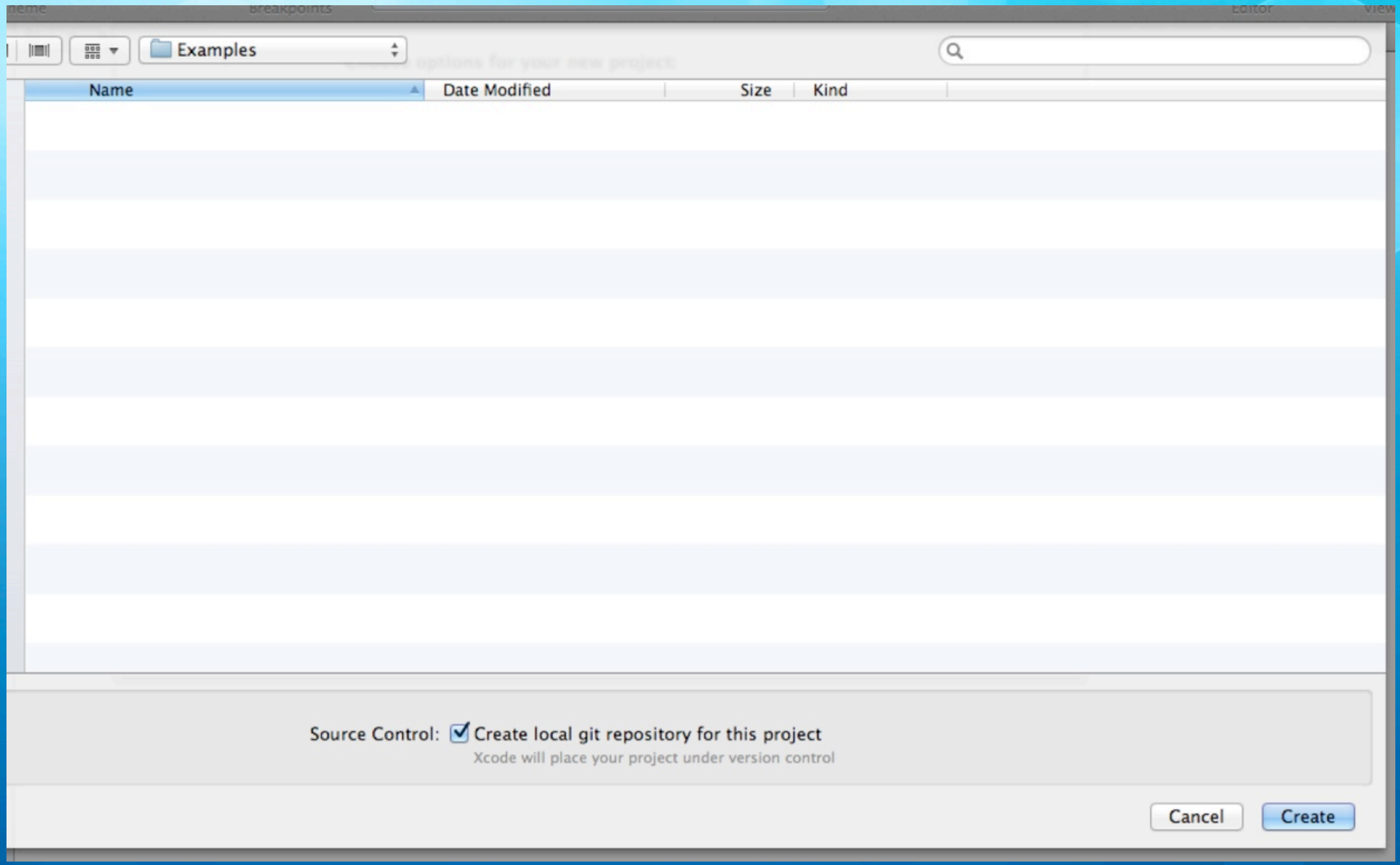
Step One



Step Two


Choose options for your new project:
Product Name
HelloWorld
Type
Foundation
Cancel
Previous
Next

Step Two



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

There are folders for this program's source code (.m and .h files), 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

All Output ▲▼ Clear

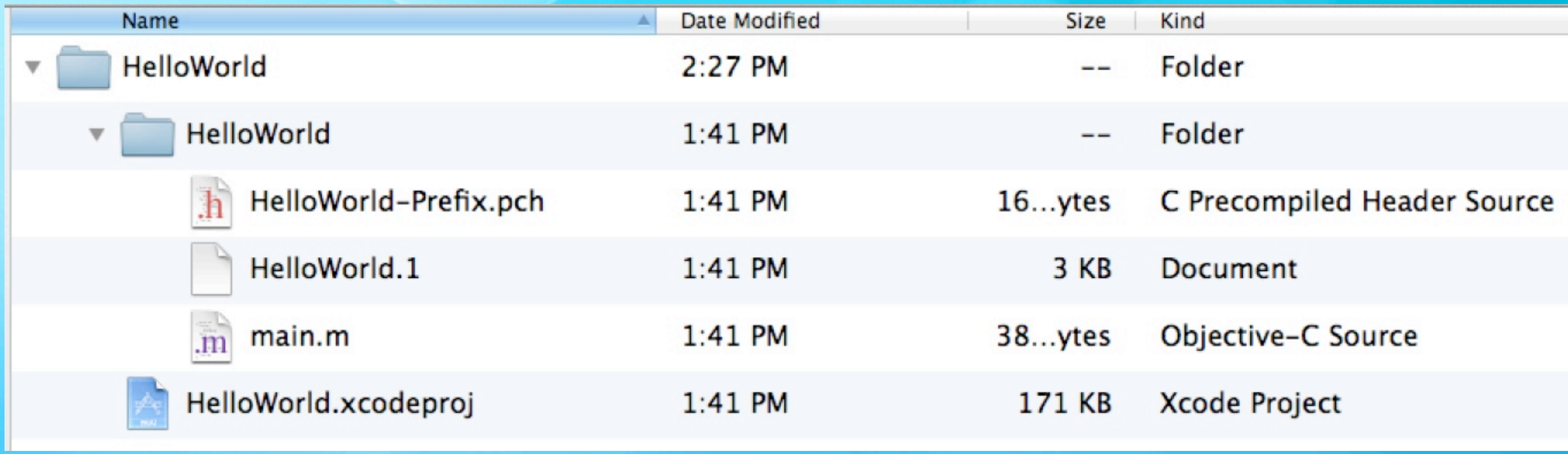
```
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Fri Jul  1 10:50:06 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys000
[Switching to process 4616 thread 0x0]
2011-09-24 14:18:01.040 HelloWorld[4616:707] Hello, World!
Program ended with exit code: 0
```

Exciting, isn't it?

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

One interesting thing to note is that the program is being run by gdb

You can hide gdb's output by switching the pop-up menu in the upper left to "Target Output"



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.

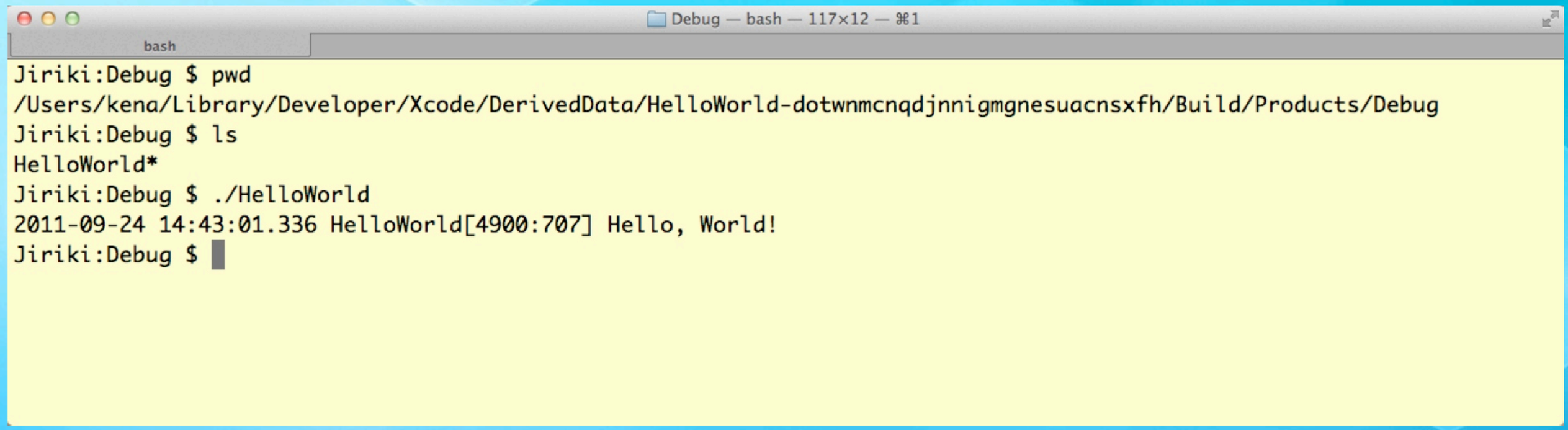
XCode project directories are a lot simpler now that files generated during a build are stored elsewhere.

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 20 directories containing 31 files!

Why so many files and directories?

- In addition to the actual results of compiling the source code, XCode stores in DerviedData
  - Logs and Indexes (for code autocomplete feature)
  - Build information, including
    - .o files, precompiled headers, debug information, etc.
- The actual executable was located at
  - ~/Library/Developer/Xcode/DerivedData/HelloWorld-dotwnmcnqdjnnigmgnesuacnsxfh/Build/Products/Debug



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

As you can see, most of the text on Slide 15 was generated by gdb... our command line tool doesn't do much but say hi to the world.

Note the "2011-09-24 14:43:01.336 HelloWorld[4900:707]" is generated by a function called NSLog() as we'll see next

```objc
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // insert code here...
    NSLog(@"Hello, World!");
    [pool drain];
    return 0;
}
```

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 only included once and only once

Ignore the "NSAutoreleasePool" stuff for now

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

The blue arrow indicates that a breakpoint has been set; gdb will stop execution on line 7 the next time we run the program

HelloWorld.m: HelloWorld – Debugger

Debug | x86_64

Overview | Breakpoints | Build and Debug | Tasks | Restart | Continue | Step Over | Step Into | Step Out

#	Thread-1-<com.apple.main-thread>
0	main

Variable	Value	Summary
▼ Arguments		
argc	1	
▶ argv	0x7fff5fbff498	
▼ Locals		
▶ pool	0x100109020	
▶ Globals		
▶ Registers		
▶ Vector Registers		
▶ x87 Registers		

HelloWorld.m:7:1 | main()

```
1  #import <Foundation/Foundation.h>
2
3  int main (int argc, const char * argv[]) {
4      NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
5
6      // insert code here...
7      NSLog(@"Hello, World!");
8      [pool drain];
9      return 0;
10 }
11
```

GDB: Stopped at breakpoint 1 (hit count : 1) – 'main() – Line 7'

Succeeded

gdb is integrated into XCode; here gdb is stopped at our breakpoint; this is XCode 3, XCode 4 looks similar

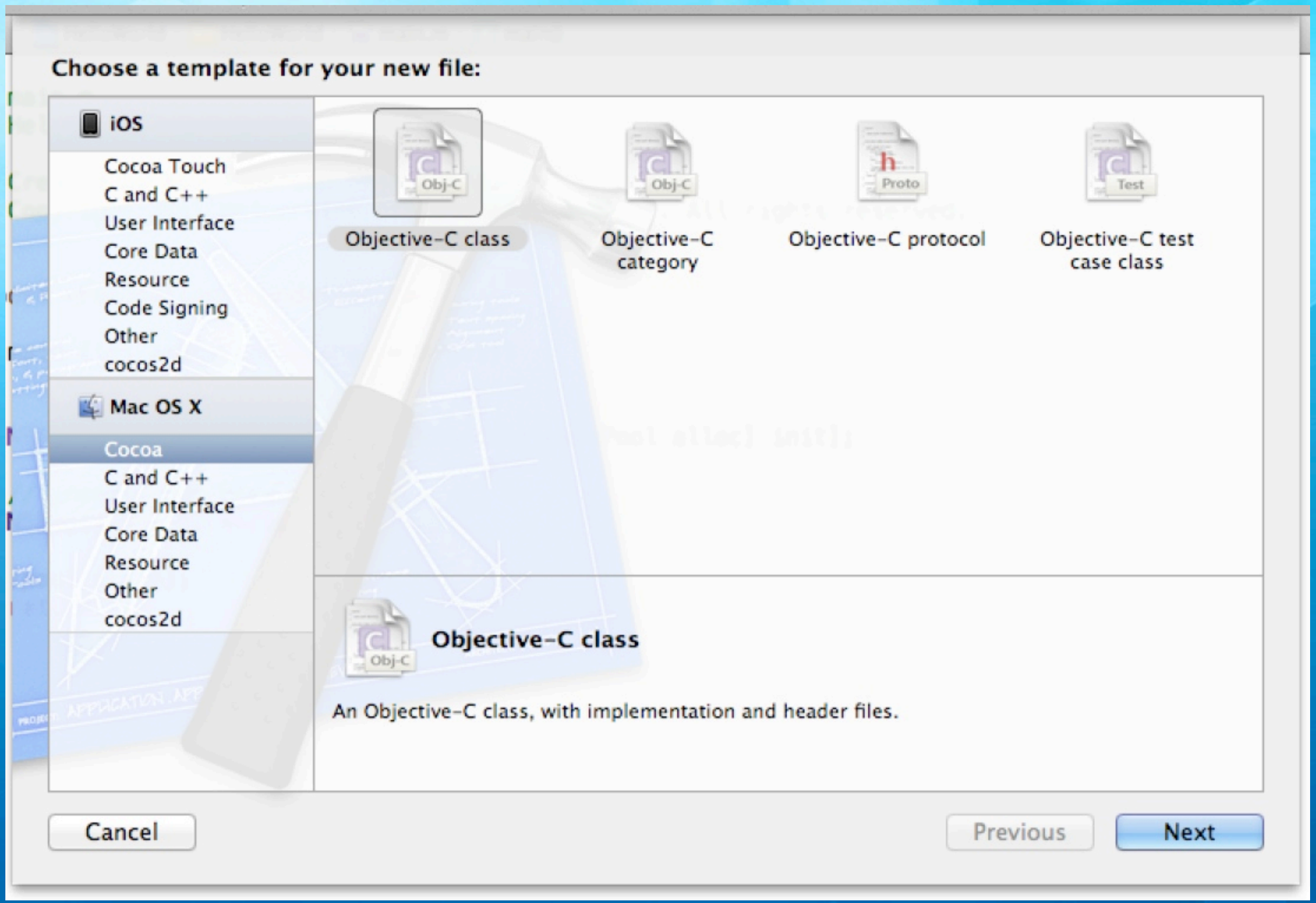
Let's add objects…

- 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
- We're going to create an Objective-C class called Greeter to make this HelloWorld program a bit more object-oriented

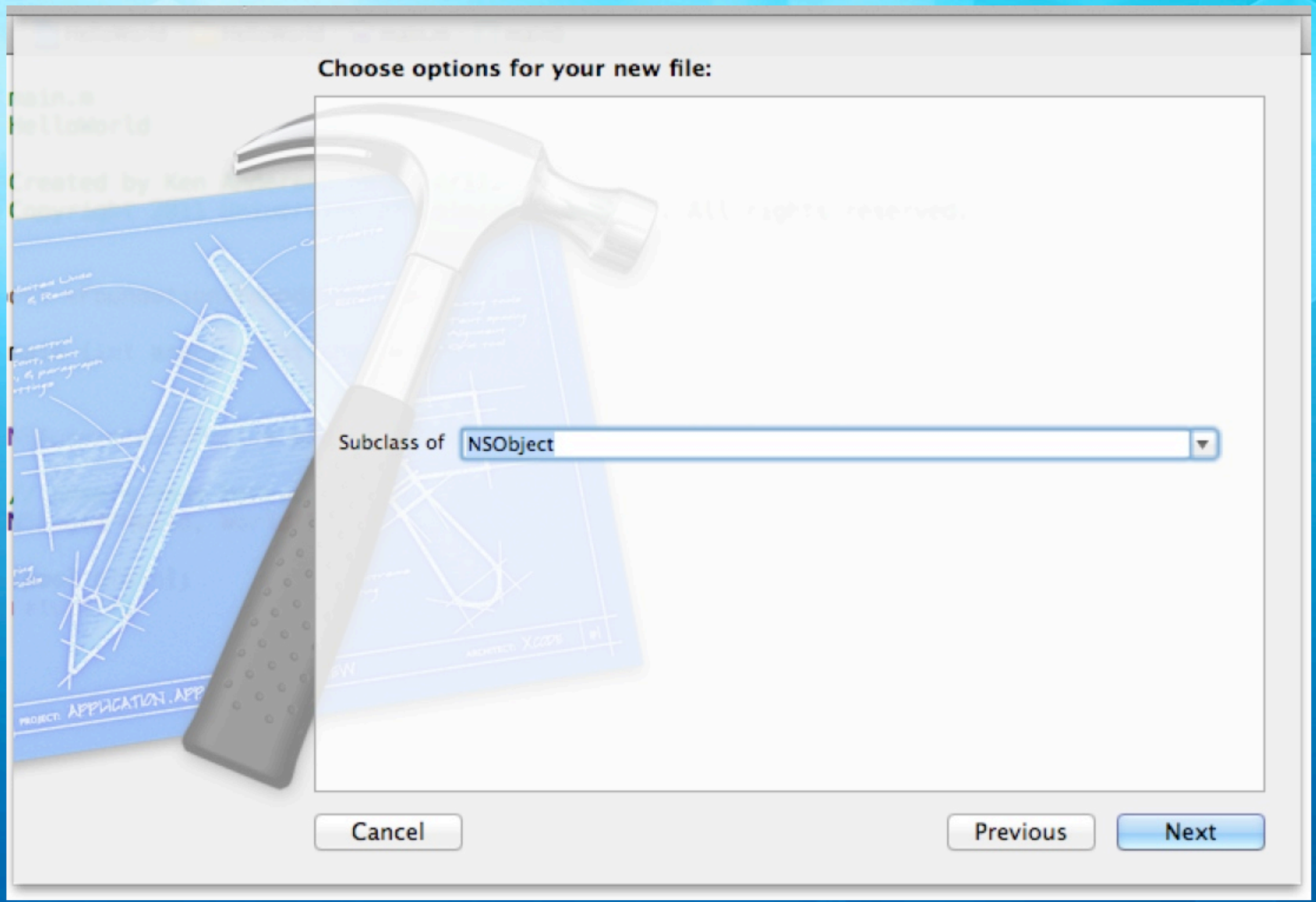
First, we are going to add a class

- 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)
  - Select NSObject as your superclass (click Next)
  - Name file "Greeter.m" add to HelloWorld Group and HelloWorld Target. (Click Save.)

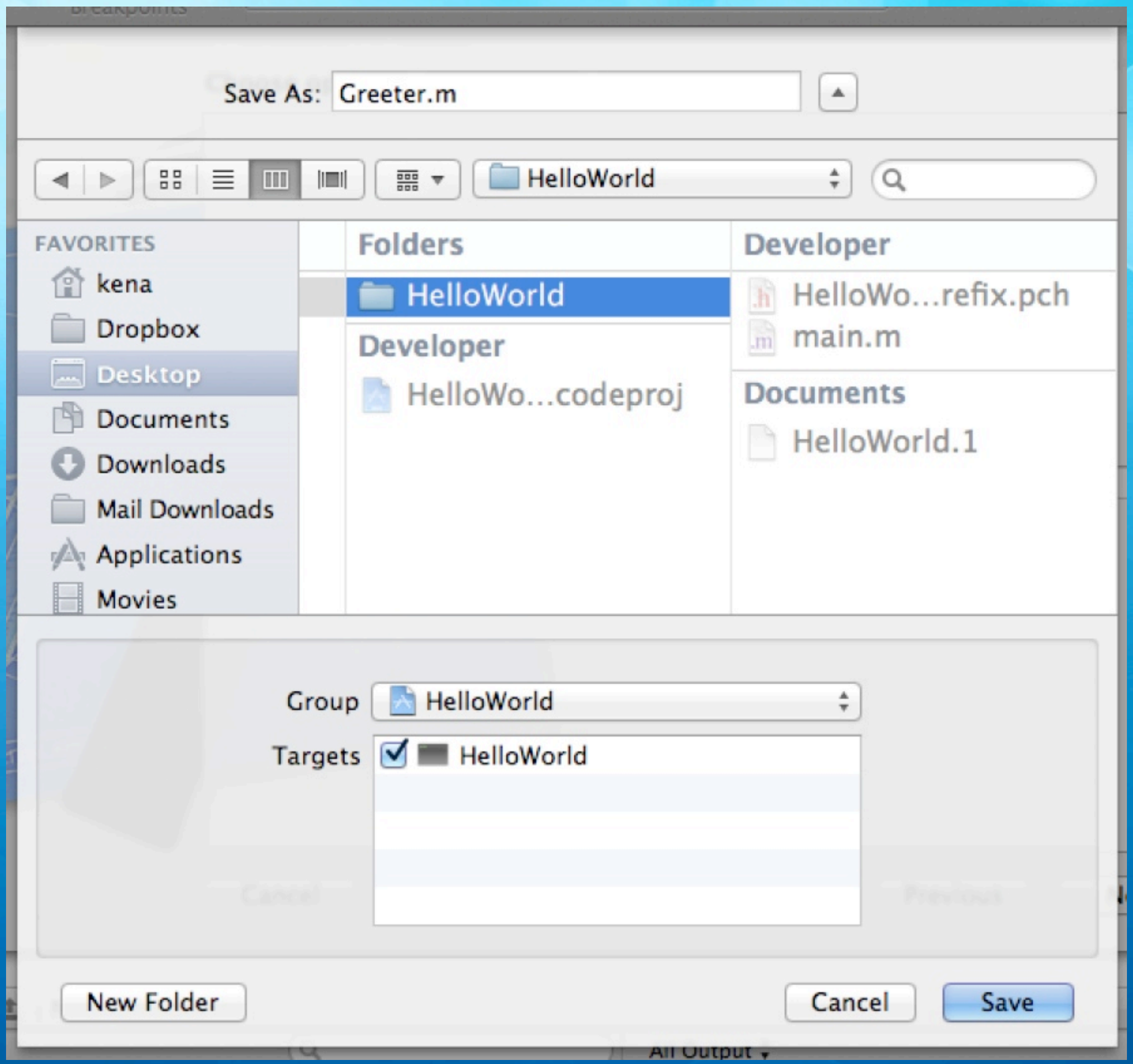
Step One

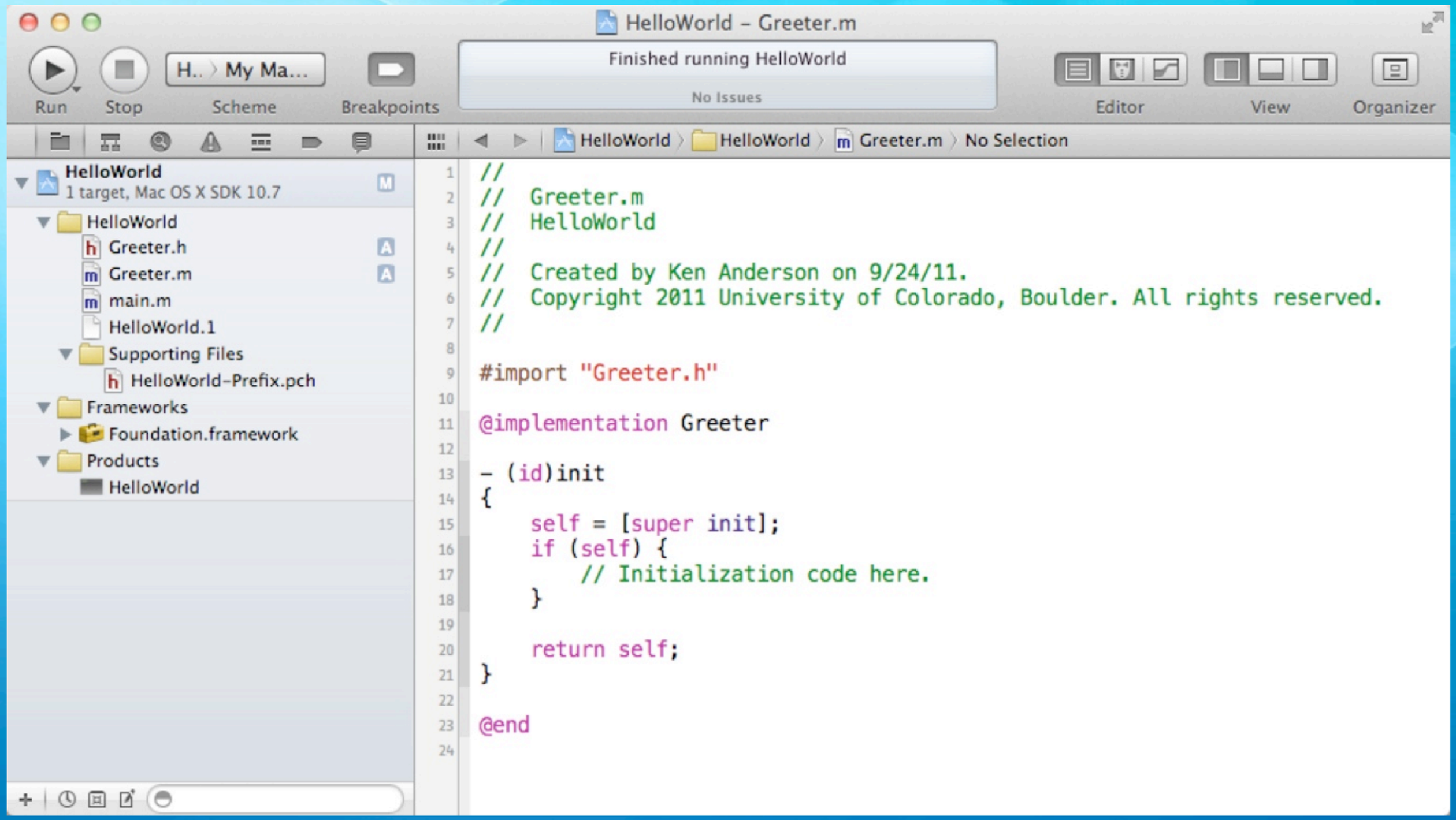


Step Two


Choose options for your new file:
Subclass of
NSObject
Cancel
Previous
Next

Step Three





Greeter.h and Greeter.m are added to our project. (Note: the "A" next to their names is a "git annotation" meaning that git has detected that two new files are ready to be added to the repository.) Greeter.m is shown with a default constructor.

Objective-C classes

- Classes in Objective-C are defined in two files
  - A header file which defines the attributes and method signatures of the class
  - 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 Objective-C 2.0, the structure has changed to the following (the previous structure is still supported)

```
<import statements>

@interface <classname> : <superclass name>

<property definitions>

<method signature definitions>

@end
```

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 (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)

```
1  //
2  //  Greeter.h
3  //  HelloWorld
4  //
5  //  Created by Ken Anderson on 9/24/11.
6  //  Copyright 2011 University of Colorado, Boulder. All rights reserved.
7  //
8
9  #import <Foundation/Foundation.h>
10
11 @interface Greeter : NSObject
12
13 @property (nonatomic, copy) NSString* greetingText;
14
15 - (void) greet;
16
17 @end
18
```

```objc
//
//  Greeter.h
//  HelloWorld
//
//  Created by Ken Anderson on 9/24/11.
//  Copyright 2011 University of Colorado, Boulder. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface Greeter : NSObject

@property (nonatomic, copy) NSString* greetingText;

- (void) greet;

@end

```

We've added one property: It's called greetingText. Its type is NSString* which means "pointer to an instance of NSString"

We've also added one method called greet. It takes no parameters and its return type is "void".

(By the way, NS stands for "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* greetingText
- "nonatomic" tells the runtime that this property will never be accessed by more than one thread (use "atomic" otherwise)
- "copy" is related to memory management and will be discussed later

Objective-C Properties (II)

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

Objective-C Properties (IV)

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

Objective-C Properties (V)

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

Objective-C Methods (I)

- It takes a while to get use to Object-C method signatures

```
- (void) setGreetingText: (NSString*) newText;
```

- defines an instance method (-) called setGreetingText:
- The colon signifies that the method has one parameter and is PART OF THE METHOD NAME
  - newText of type (NSString*)
- The names setGreetingText: and setGreetingText refer to TWO different methods; the former has one parameter

Objective-C Methods (II)

- A method with multiple parameters will have multiple colon characters and the parameter defs are interspersed with the method name

```
- (void) setStrokeColor: (NSColor*) strokeColor
             andFillColor: (NSColor*) fillColor;
```

- The above signature defines a method with two parameters called setStrokeColor:andFillColor:

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

```
#import "Greeter.h"

@implementation Greeter

@synthesize greetingText=_greetingText;

- (id)init {
    self = [super init];
    if (self) {
        self.greetingText = @"Default Greetings!";
    }

    return self;
}

- (void) greet {
    NSLog(@"%@", self.greetingText);
}

- (void) dealloc {
    [_greetingText release];
    [super dealloc];
}

@end
```

```objc
#import "Greeter.h"

@implementation Greeter

@synthesize greetingText=_greetingText;

- (id)init {
    self = [super init];
    if (self) {
        self.greetingText = @"Default Greetings!";
    }

    return self;
}

- (void) greet {
    NSLog(@"%@", self.greetingText);
}

- (void) dealloc {
    [_greetingText release];
    [super dealloc];
}

@end
```

@synthesize is used to actually create the instance variable, setter and getter of a property. In this case, we are asking that the instance variable for the property greetingText be called _greetingText.

This allows us to be certain when we are accessing the property in the .m file and when we are accessing

```objc
#import "Greeter.h"

@implementation Greeter

@synthesize greetingText=_greetingText;

- (id)init {
    self = [super init];
    if (self) {
        self.greetingText = @"Default Greetings!";
    }

    return self;
}

- (void) greet {
    NSLog(@"%@", self.greetingText);
}

- (void) dealloc {
    [_greetingText release];
    [super dealloc];
}

@end
```

init is the constructor. It calls init on the superclass and makes sure we got an allocated object. If so, we initialize our property to a proper value.

If the call to init on the superclass fails it returns "nil" which is what we then return

```objc
#import "Greeter.h"

@implementation Greeter

@synthesize greetingText=_greetingText;

- (id)init {
    self = [super init];
    if (self) {
        self.greetingText = @"Default Greetings!";
    }

    return self;
}

- (void) greet {
    NSLog(@"%@", self.greetingText);
}

- (void) dealloc {
    [_greetingText release];
    [super dealloc];
}

@end
```

Here is the implementation of our greet method. It simply prints our greetingText to standard out using the NSLog() function.

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

%@ means "object"; %s, %d, etc. also supported

```objc
#import "Greeter.h"

@implementation Greeter

@synthesize greetingText=_greetingText;

- (id)init {
    self = [super init];
    if (self) {
        self.greetingText = @"Default Greetings!";
    }

    return self;
}

- (void) greet {
    NSLog(@"%@", self.greetingText);
}

- (void) dealloc {
    [_greetingText release];
    [super dealloc];
}

@end
```

Finally, dealloc is the "destructor" of the class. Unlike finalize() in Java, dealloc is guaranteed to be called when an instance of Greeter is deallocated.

Here we access our instance variable directly and tell it to go away. We then call dealloc on the super class

Note: we do not access our property in dealloc; there are situations where the property mechanism may not work in the dealloc method

```objc
#import "Greeter.h"

@implementation Greeter

@synthesize greetingText=_greetingText;

- (id)init {
    self = [super init];
    if (self) {
        self.greetingText = @"Default Greetings!";
    }

    return self;
}

- (void) greet {
    NSLog(@"%@", self.greetingText);
}

- (void) dealloc {
    [_greetingText release];
    [super dealloc];
}

@end
```

Note: we do NOT define method bodies for greetingText and setGreetingText:

The getter and setter methods are automatically generated.

We don't need to implement them.

Note: We could override them if we needed to.

But first, 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
  - Then the method with any arguments for the methods parameters

But first, calling methods (II)

- Here's a call using Greeter's setter method; @"Howdy!" is a shorthand syntax for creating an NSString instance
  - `[greeter setGreetingText: @"Howdy!"];`
- Here's a call to the same method where we get the greeting from some other Greeter object
  - `[greeterOne setGreetingText:[greeterTwo greetingText]];`
- 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 involves the use of six methods
  - alloc, init, dealloc, retain, release, autorelease
- Objects are created using alloc and init
- We then keep track of who is using an object with retain and release
- We get rid of an object with dealloc (although, we never call dealloc ourselves)

Memory Management (II)

- When an object is created, its retain count is set to 1
  - It is assumed that the creator is referencing the object that was just created
- If another object wants to reference it, it calls retain to increase the reference count by 1
  - When it is done, it calls release to decrease the reference count by 1
- If an object's reference count goes to zero, the runtime system automatically calls dealloc

Memory Management (III)

- I won't talk about autorelease today, we'll see it in action soon
- Objective-C 2.0 added a garbage collector to the language
  - When garbage collection is turned on, retain, release, and autorelease become no-ops, doing nothing
  - However, the garbage collector is not available when running on iOS, so the use of retain and release are still with us
- Apple recently released "automatic reference counting" which may make all of this go away (including the garbage collector)

Memory Management (IV)

```
- (void) setGreetingText: (NSString *) newText {
    [newText retain];
    [_greetingText release];
    _greetingText = newText;
}
```

In a typical auto-generated setter method, the following memory management dance occurs:

retain the new value, release the old, set our instance variable to point at the new value

Memory Management (IV)

```
- (void) setGreetingText: (NSString *) newText {
    [newText retain];
    [_greetingText release];
    _greetingText = newText;
}
```

We perform this “dance” in this order because there is a slight chance that _greetingText == newText

If so, if we called release first, we would deallocate our instance variable with no way of getting its value back!

The dealloc method

```
- (void) dealloc {
    [_greetingText release];
    [super dealloc];
}
```

- The dealloc method releases the NSString that we are pointing at with our automatically generated instance variable and then invokes the dealloc method of our superclass
  - We've now seen examples of the self and super keywords

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 set its greeting text
  - We'll call its greet method
  - We'll release it

```objc
#import <Foundation/Foundation.h>
#import "Greeter.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Greeter* myGreeter = [[Greeter alloc] init];
    myGreeter.greetingText = @"Hello from Objective-C!!";
    [myGreeter greet];
    [myGreeter release];
    [pool drain];
    return 0;
}
```

As you can see, we create an instance of Greeter. We then set its greetingText property, invoke the greet method, and release our instance of myGreeter.

Since we were the only ones pointing at our instance of Greeter, its dealloc method is automatically called. This in turn releases the string pointed at by our greetingText property.

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 of objects and memory management
- More to come as we use this knowledge to explore the iOS platform in future lectures

Coming Up Next

- Homework 4 Due on Monday
- Lecture 13: Introduction to iOS
- Lecture 14: Review for Midterm
- Lecture 15: Midterm
- Lecture 16: Review of Midterm