Objective-C *Programming*

iOS 12 • Xcode 10

STUDENT GUIDE

ABOUTOBJECTS

Contact

About Objects, Inc.
11911 Freedom Drive
7th Floor
Reston, VA 20190

Main: 571–346–7544
email: info@aboutobjects.com
web: **www.aboutobjects.com**

Course Information

Author: Jonathan Lehr
Revision: 4.4
Last Update: 10/4/2018

Classroom materials for an introductory-level course that provides a rapid introduction to programming in C and Objective-C with the Foundation framework. Geared to developers interested in learning the iOS platform. Includes comprehensive lab exercise instructions and solution code.

Copyright Notice

© Copyright 2014–2018 About Objects, Inc.

Under the copyright laws, this documentation may not by copied, photographed, reproduced, translated, or reduced to any electronic medium or machine–readable form, in whole or in part, without express written consent.

All other copyrights, trademarks, registered trademarks and service marks throughout this document are the property of their respective owners.

All rights reserved worldwide. Printed in the USA.

Objective-C *Programming*

STUDENT GUIDE

ANSI C Basics

Objective-C is a superset of ANSI C. Most Objective-C constructs are translated into C by the compiler—for example, methods are translated into C functions. Perhaps more important though, the iOS SDK consists not only of Objective-C frameworks, but also a significant number of C libraries, which provide the API for such vital technologies as Core Graphics and Grand Central Dispatch.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Topics/Goals

By the end of this module you'll be able to declare and/or define, initialize, and use essential C constructs, including the following:

- Local and global variables
- Functions and parameters
- Data structures
- Pointers and arrays
- Enumerated constants
- Typedef statements

You'll also be able to describe a program's memory segments, and you'll be able to dynamically allocate and free memory in heap storage.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Glossary

Term	Definition
preprocessor	A global text editor that performs substitutions on *preprocessor directives* (a type of markup) in source code prior to compile time.
linker	A program that combines individual object files into a single executable
pointer	A variable that contains the address of another variable.
typedef	A custom label for an existing data type.
enum	A list of symbols that can be used to represent integer constants.
static	Allocated at compile time. Also, as a type qualifier, limits the scope of a function or variable to the current file.
local	The scope of a variable or argument declared within a function.
global	The scope of a variable declared outside of a function.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Data Types

- Hardware supports two fundamental types of arithmetic: **integer** and **floating point**.
- C provides two fundamental data types for declaring variables and functions:

```
int
```

```
float
```

- The data type precedes the symbol being declared.

```
int x; // Defines x as variable of type int.
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Defining Variables

- A variable definition has the following effects:
  - Allocates storage for a value of the declared type.
  - Binds the variable name to the declared type.
  - Makes the name a label for the stored value.

```
// The following line of code defines a variable named 'x'.
//
// Binds 'x' to the int data type.
// Allocates storage for a signed integer (32 bits).
// Associates 'x' with the stored value.
//
// Note: doesn't initialize the stored value.
//
int x;

// You can then assign a value to the variable
x = 42;
```

- A variable definition can specifiy an initial value.

```
// Defines x and initializes it to 42.
int x = 42;
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Constants

Constants have their own data types. Some examples:

```
// signed int.
int i = 42;

// unsigned long.
unsigned long ul = 42ul;

// double.
double d = 42.0;

// float.
float f = 42.0f;

// Character constant; unsigned char.
unsigned char c = 'a';

// String constant; pointer to char (unsigned).
char *s = "abc";
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Enumerated Constants

An `enum` declares a list of labels mapped to constant integer values. The first label will have the value **0**, the next **1**, and so on, unless you specify explicit values.

```
enum PetType {
    Dog,    // 0
    Cat,    // 1
    Bird    // 2
};

// With explicit enumeration values...

enum PetType {
    Dog  = 1,
    Cat  = 9,
    Bird = 86
};

// Using enum labels in place of hard-coded constants.

void PrintPetType(enum PetType petType)
{
    printf("Pet type is ");

    switch (petType)
    {
        case Dog:  printf("dog");    break;
        case Cat:  printf("cat");    break;
        case Bird: printf("bird");   break;
        default:   printf("unknown");
    }

    printf(".\n");
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Type Coercion and Casts

- Type coercion is **implicit type conversion** performed by compiler on expressions with mixed numeric types.
  - Compiler performs **type promotion**, converting simpler/smaller types to more complex/larger types, e.g. int to float, float to double.

```
int a = 2;
float b = 3.0f;
float c = a / b; // a coerced to float. Value: 0.6777...


// Note that the result of the previous line would be
// different if a and b were both of type int.
int a = 2;
int b = 3;
float c = a / b;  // Integer division yields value 0.
```

- Use a **cast** operator to specify **explicit type conversion**.

```
// Cast converts a to float and compiler promotes b to
// float, resulting in floating point division.
//
float c = (float)a / b;  // Value: 0.6777...
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

typedef Statements

A `typedef` statement allows you to create a custom name for an existing data type.

```
// Declare Percentage to be a synonym for float.
typedef float Percentage;

// Here we're substituting Percentage for float.
Percentage IncreaseMyTaxes(Percentage currentTax)
{
    Percentage newTaxRate = currentTax * 2.0;
    return newTaxRate;
}
```

A `typedef` can be conditionally compiled for greater portability.

```
// Conditionally compile for Apple's 64-bit architecture.
// (LP64 means the size of longs and pointers is 64 bits.)
#ifdef __LP64__
typedef double Percentage;
#else
typedef float Percentage;
#endif
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Functions

A C program is essentially a list of functions.

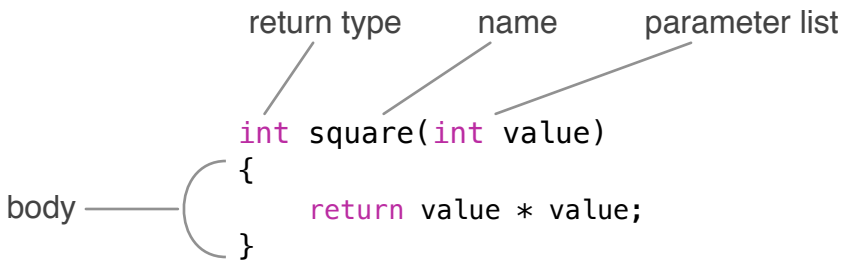
Function Invocation

To call a function, specify its name followed by a pair of parentheses enclosing a comma separated list of arguments. The number of arguments, their data types, and their order must match the declared parameters.

```
double x = fmax(y, z);
```

Function Definitions

Each function definition consists of a name, a return type, and a parameter list, as shown in the following figure:



Note: Use `void` as the return type for functions that don't return a value, and as the data type for empty parameter lists.

```
void printHello(void)
{
    printf("Hello.\n");
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Function Declarations (1)

- A function's name is put in scope by the compiler as soon as it encounters either a definition or a declaration of the function. (Note: functions are global in scope.)
- Compiler assumes default signature for undeclared functions:

```
int functionName(<zero or more args of type int>);
```

- The following code won't compile as is, because compiler will see conflicting types for `printHello`.

```
void doStuff(void)
{
    printHello();    // Compiler warns about implicit declaration
                     // because printHello not in scope.
}


void printHello(void) // Hard error, because definition of
{                     // printHello doesn't match declaration.
    printf("Hello.\n");
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Function Declarations (2)

- To avoid implicitly declaring `printHello`, add a declaration anywhere above `doStuff`.

```
void printHello(void); // Compiler reads explicit declaration of
                       // printHello and puts the name in scope.

// ...


void doStuff(void)
{
    printHello();     // Compiler now sees 'printHello' in scope.
}


void printHello(void) // Definition matches explicit declaration.
{
    printf("Hello.\n");
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Variadic Functions

`printf` is an example of a function that can take a varying number of arguments. It's declared as follows:

```
int printf(const char *format, ...);
```

The above example declares a function that takes as its initial argument a pointer to an immutable C string, followed by zero or more arguments of any type.

Three dots (`...`) is the data type for a variable length argument list.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

The printf Function

- First argument (required) is *format string*.
  - Literal text and optional format specifiers.
  - printf replaces format specifiers with values of corresponding arguments.

Example:

```
int x = 42;
float y = 3.14;

printf("x is %d, y is %.2f \n", x, y);
```

Resulting output:

```
x is 42, y is 3.14
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Format Specifiers

The following table provides a summary of some of the more frequently used format specifiers, along with some of their available options.

Format Specifier	Argument Type	Description
`%d`	`int`	**Decimal (base 10) integer. Use %*nd* to specify a field width; e.g., %12d formats value with a field width of 12, padded on the left. Prefix numeric value with a - (minus sign) to pad on the right, e.g., %-12d.**
`%i`		
`%x`	`int`	**Hexadecimal integer. When given as %#x, prefixes value with 0x.**
`%f`	`float`	**Floating point. By default, decimal precision is six. Use %.*n*f to specify a different precision (e.g., %.2f for two decimal places). Can be combined with field width specifier (see above).**
`%c`	`char`	**An individual character. Use %lc to format multi-byte Unicode sequences.**
`%s`	`char *`	**String. Can be used with field width specifier (see %d, above).**
`%%`	*`none`*	**Literal % sign.**
`%p`	*`pointer`*	**Pointer value, prefixed with 0x.**

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

The Standard C Library

- Ships with C compiler.
- Collection of small libraries, each with its own header file.
    - Subset of the headers in **/usr/include**.
- Header files contain declarations of functions, data structures, and other components of C libary's *application programming interface* (API).

  Example: Standard I/O library declared in **stdio.h**.

```
#import <stdio.h>


int main(int argc, char *argv[])
{
    printf("Hello World!\n");
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

The C Preprocessor

- Global text editor (**cpp**) run prior to compiling source code files.
- Preprocessor markup consists of directives prefixed with **#**.
  - Conditional compilation: **#if, #ifdef, #ifndef, #else, #elif, #endif**
  - Global text substitution (macros and symbolic constants): **#define**
  - Inclusion of files: **#include, #import**
- Directives are removed from source during preprocessing.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Header Files

- Header files contain declarations of functions, data structures, and other components of C libary's application programming interface (API).

  Example: Excerpts from **stdio.h**.

```
// ...

#define BUFSIZ  1024    /* size of buffer used by setbuf */
#define EOF     (-1)

// ...

#ifdef _USE_EXTENDED_LOCALES_
#include <xlocale/_stdio.h>
#endif

// ...

int   getc(FILE *);
int   getchar(void);
char *gets(char *);
void  perror(const char *);
int   printf(const char * __restrict, ...);
int   putc(int, FILE *);
int   putchar(int);
int   puts(const char *);

// ...

#define fropen(cookie, fn) funopen(cookie, fn, 0, 0, 0)
#define fwopen(cookie, fn) funopen(cookie, 0, fn, 0, 0)
```

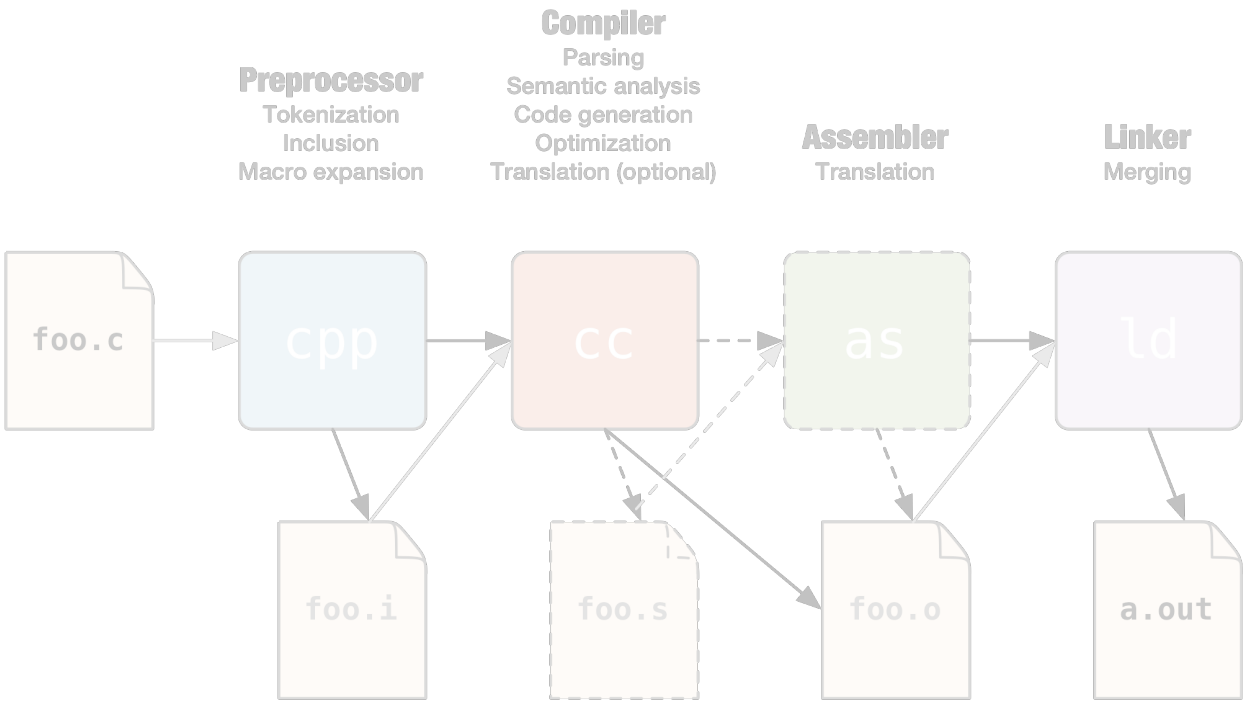
© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Phases of Compilation

- Clang compiler runs through several distinct phases.
  - Compiler and assembler optionally coalesced into single phase, in which case, no **.s** file produced.
- Header files contain declarations of functions, data structures, and other components of C libary's *application programming interface* (API).

Conceptual Phases of Clang Compilation



© Copyright 2014-2018, About Objects, Inc.

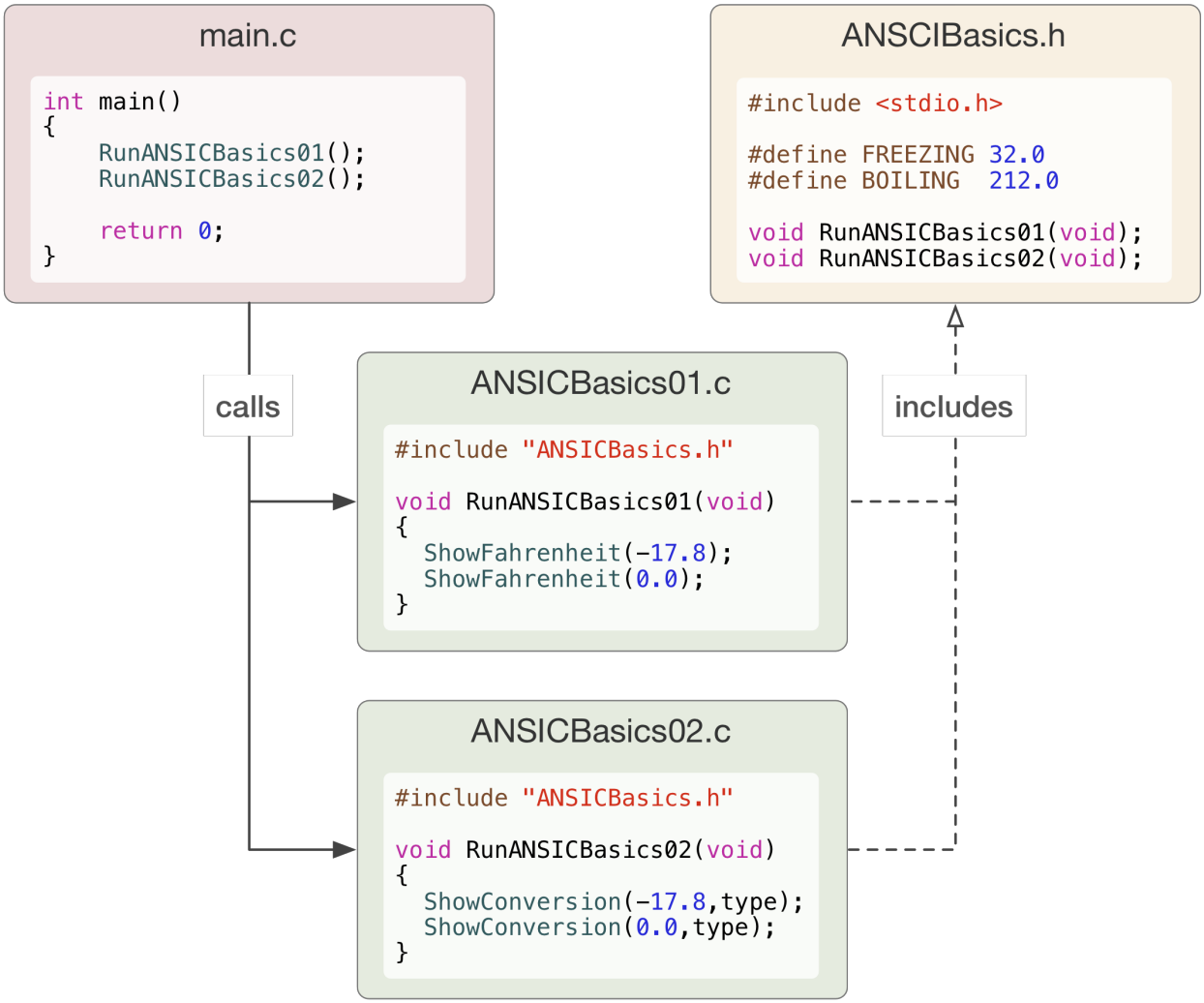
All rights reserved worldwide.

Lab – ANSI C Basics

OVERVIEW

Create a separate **.c** file for each part of the lab, and a single **.h** file to be shared among them, as shown below.

Conceptual Organization of ANSI C Basics Lab



© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 1

1. Create an Xcode project named **ANSI C Labs** as follows:

    1.1. From Xcode's **File** menu select **New -> Project**.

    1.2. From the Template Chooser, select **OS X | Application** on the left, and **Command Line Tool** on the right, and click **Next**.

    1.3. Enter the name of the project and a company identifier in the form that appears. Make sure that **C** is selected in the **Type** dropdown, and then click **Next**.

    1.4. Pick the directory where you want to save the project (the **Desktop** would be fine if you don't have a preference), and click **Save**.

2. Add three new files: **ANSICBasics.h**, **ANSICBasics01.c**, and **ANSICBasics02.c** as follows:

    2.1. From Xcode's **File** menu select **New -> File**.

    2.2. From the Template Chooser, select **OS X | C and C++** on the left, and **Header File** on the right, and click **Next**. Enter the file name **ANSICBasics.h** and select **Create**.

    2.3. Repeat the previous step, but this time select **C File** on the right instead of **Header File**. Do this twice, once to create **ANSICBasics01.c**, and again to create **ANSICBasics02.c**.

3. Add code to the three new files.

    3.1. In **ANSICBasics.h**, declare functions named `RunANSICBasics01`, and `RunANSICBasics02`. The functions should take no arguments and return `void`.

    3.2. Write definitions for the functions declared in the previous step in their corresponding **.c** files, e.g., write a definition of `RunANSICBasics01` in **ANSICBasics01.c**, etc. The initial implementations should simply print out the function's name, followed by a newline.

    3.3. In **main.c**, add calls to the functions `RunANSICBasics01` and `RunANSICBasics02`. Don't forget to include the header in which the functions were declared.

    3.4. Build and run the tool. You should see the names of the functions printed on the console. Make sure this is working correctly before proceeding to the next step.

4. Add two new functions to **ANSICBasics01.c**.

    4.1. Write a function named `ConvertToFahrenheit` that takes an argument named `celsiusTemperature` of type `float`, and returns `float`. Initially make this a stub implementation that simply returns a hard-coded value, such as **99.99**.

    4.2. Write a function named `ShowFahrenheitConversion` that also takes an argument named `celsiusTemperature` of type `float`, but returns `void` instead of `float`. Implement the function to call `ConvertToFahrenheit` to convert the `celsiusTemperature` argument to Fahrenheit, and then print the original and

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

converted values to one decimal place.

4.3. Call the `ShowFahrenheitConversion` function from `RunANSICBasics01`, passing **0.0** as the argument. Build and run to verify that the pair of values is printed correctly.

5. Add the necessary conversion functionality to `ConvertToFahrenheit`.

5.1. In **ANSICBasics.h**, use preprocessor directives to define symbolic constants as follows:
F_FREEZING_POINT. Freezing point of water at sea level, in Fahrenheit. Value, **32.0**.
F_BOILING_POINT. Boiling point of water at sea level, in Fahrenheit. Value, **212.0**.
C_SCALE. Number of degrees between sea-level freezing and boiling points in Celsius scale. Value, **100.0**.
F_SCALE. Number of degrees between sea-level freezing and boiling points in Fahrenheit scale. Value, **(F_BOILING_POINT - F_FREEZING_POINT)**.

5.2. Use the following formula for the conversion: *Fahrenheit temperature = Celsius temperature x (Fahrenheit scale / Celsius scale) + Fahrenheit freezing point*.

5.3. In the `RunANSICBasics01` function, add several more calls to `ShowFahrenheitConversion` with different argument values, and build and run to verify that the formula works as expected.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

PART 2

1. Add a pair of new functions in **ANSICBasics02.c**.

    1.1. Declare a set of enumerated constants with the tag `ConversionType`, and two enumeration values, labelled `CelsiusToFahrenheit`, and `FahrenheitToCelsius`.

    1.2. Add a typedef statement to allow `ConversionType` to be used as a data type.

    1.3. Add a function named `ConvertTemperature` that returns `float`, and takes two arguments: `temperature`, of type `float`, and `type`, of type `ConversionType`. Initially make this a stub implementation that simply returns a hard-coded value, such as **99.99**.

    1.4. Write a function named `ShowConversion` that returns void, and takes the same two arguments as `ConvertTemperature`. Implement the function to call `ConvertTemperature` to convert the provided `temperature`, and then print the original and converted values.

    1.5. In `RunANSICBasics02`, call the `ShowConversion` function once for each conversion type, passing **0.0** as the argument. Build and run to verify that the pairs of values are printed as expected.

2. Add the necessary conversion functionality to `ConvertTemperature`.

    2.1. Use the following formula for the Fahrenheit to Celsius conversion: *Celsius temperature = (Fahrenheit temperature - Fahrenheit freezing point) x (Celsius scale / Fahrenheit scale)*, and the inverse for Celsius to Fahrenheit.

    2.2. In `RunANSICBasics02`, add several more calls to `ShowConversion` with different argument values, and build and run to verify that the formula works as expected.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

Lab Solutions – ANSI C Basics

PART 1

ANSICBasics.h (common declarations used in both parts of lab)

```
#include <stdio.h>

#define F_FREEZING_POINT 32.0
#define F_BOILING_POINT  212.0

// Scale represents number of degrees between freezing and boiling.
#define C_SCALE  100.0
#define F_SCALE (F_BOILING_POINT - F_FREEZING_POINT)

void RunANSICBasics01(void);
void RunANSICBasics02(void);
```

ANSICBasics01.c

```
#include "ANSICBasics.h"

float ConvertToFahrenheit(float celciusTemperature);
void  ShowFahrenheitConversion(float celsiusTemperature);


void RunANSICBasics01(void)
{
    printf("\n%s\n----------\n", __func__);

    ShowFahrenheitConversion(-17.8);
    ShowFahrenheitConversion(0.0);
    ShowFahrenheitConversion(12.8);
    ShowFahrenheitConversion(24.2);
    ShowFahrenheitConversion(37.0);
}


void ShowFahrenheitConversion(float celsiusTemperature)
{
    printf("%.1f degrees Celsius is %.1f degrees Fahrenheit\n",
           celsiusTemperature,
           ConvertToFahrenheit(celsiusTemperature));
}

float ConvertToFahrenheit(float celciusTemperature)
{
    return celciusTemperature * (F_SCALE / C_SCALE) + F_FREEZING_POINT;
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Console Output for Part 1

```
RunANSICBasics01
-----------
-17.8 degrees Celsius is -0.0 degrees Fahrenheit
0.0 degrees Celsius is 32.0 degrees Fahrenheit
12.8 degrees Celsius is 55.0 degrees Fahrenheit
24.2 degrees Celsius is 75.6 degrees Fahrenheit
37.0 degrees Celsius is 98.6 degrees Fahrenheit
```

PART 2

ANSICBasics02.c

```
#include "ANSICBasics.h"

enum ConversionType {
    CelsiusToFahrenheit,
    FahrenheitToCelsius,
};

typedef enum ConversionType ConversionType;

float ConvertTemperature(float, ConversionType);
void ShowConversion(float, ConversionType);


void RunANSICBasics02(void)
{
    printf("\n%s\n-----------\n", __func__);

    ShowConversion(-17.8, CelsiusToFahrenheit);
    ShowConversion(0.0, CelsiusToFahrenheit);

    ShowConversion(0.0, FahrenheitToCelsius);
    ShowConversion(98.6, FahrenheitToCelsius);
    ShowConversion(F_FREEZING_POINT, FahrenheitToCelsius);
    ShowConversion(F_BOILING_POINT, FahrenheitToCelsius);
}


void ShowConversion(float temperature, ConversionType type)
{
    printf("%.1f degrees %s is %.1f degrees %s\n",
           temperature,
           (type == CelsiusToFahrenheit ? "Celsius" : "Fahrenheit"),
           ConvertTemperature(temperature, type),
           (type == FahrenheitToCelsius ? "Celsius" : "Fahrenheit"));
}

float ConvertTemperature(float temperature, ConversionType type)
{
    return (type == CelsiusToFahrenheit ?
            temperature * F_SCALE / C_SCALE + F_FREEZING_POINT :
            (temperature - F_FREEZING_POINT) * C_SCALE / F_SCALE);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Console Output for Part 2

```
RunANSICBasics02
-----------
-17.8 degrees Celsius is -0.0 degrees Fahrenheit
0.0 degrees Celsius is 32.0 degrees Fahrenheit
0.0 degrees Fahrenheit is -17.8 degrees Celsius
98.6 degrees Fahrenheit is 37.0 degrees Celsius
32.0 degrees Fahrenheit is 0.0 degrees Celsius
212.0 degrees Fahrenheit is 100.0 degrees Celsius
```

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

Pointers and Structures

ANSI C data structures are used commonly in every Objective-C development. They are also central to the language's underlying implementation of classes and objects. Pointers are variables that refer directly to the memory locations of data elements, which is a necessity when working directly with heap-based dynamic memory allocation.

In this section, you'll learn the mechanics of working with structures and pointers, as well as closely related concepts such as arrays. You'll also learn about memory segments, scoping rules, storage classes, and the fundamentals of memory management.

As a result, you'll have laid the necessary groundwork for gaining solid insight into how key features of Objective-C, such as classes, categories, objects, methods, and dynamic message dispatching, work behind the scenes.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Topics/Goals

By the end of this section you'll be able to create, initialize and modify variables and arguments with compound data types, work with pointer arithmetic, and manage memory. You'll be able to declare, define, initialize, and modify all of the following:

- Singly- and doubly-indirect pointers.
- Arrays, including arrays of pointers.
- Data structures and structure pointers.
- Strings.
- Global variables.
- Local static variables.

You'll also be able to do the following:

- Pointer arithmetic.
- Work with pointers as function arguments and return values.
- Dynamic memory allocation and deallocation.
- Use type qualifiers such as `static` and `const`.
- Describe C storage classes and scoping rules.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Glossary

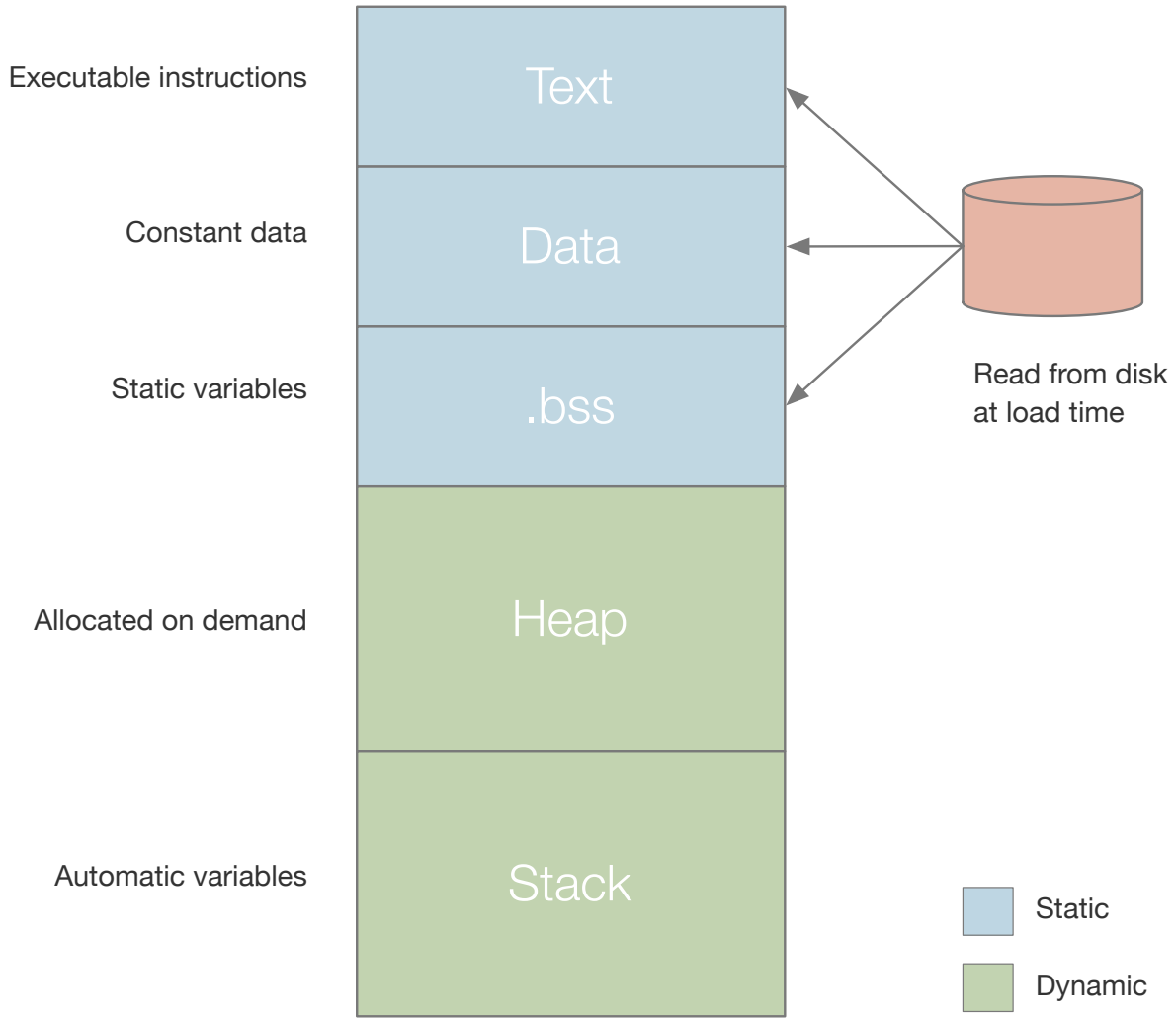
Term	Definition
storage class	The *extent* of a variable or argument, i.e., how long it lives. May be either **static** or **automatic**.
scope	The boundaries of where a given symbol is visible. May be one of the following: block, file, or program.
pointer	A variable or argument whose value is a memory address.
static	Storage class of memory allocated at compile-time. As a type qualifier, limits a symbol's scope to the current file.
automatic	Storage class of memory allocated at runtime.
reference	A memory address. Addresses by nature *refer to* something stored elsewhere in memory.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Memory Segments

The OS provides each app with its own protected virtual memory space, with several distinct segments. The program's executable is mapped into **static** memory segments. *Heap* and *stack* are **dynamic** segments the app uses at runtime.



© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Scopes

- Global symbols.
  - File scope.
  - Program scope.
- Local scope.
  - Local variables.
  - Function arguments.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Pointers

A *pointer* is a variable that contains an address.

- Note: objects in Objective-C object can only be accessed by address.
  - Object addresses are always in heap.
  - Objects can't be allocated on the stack or in static memory.
- Use the * declarator to define pointer variables.

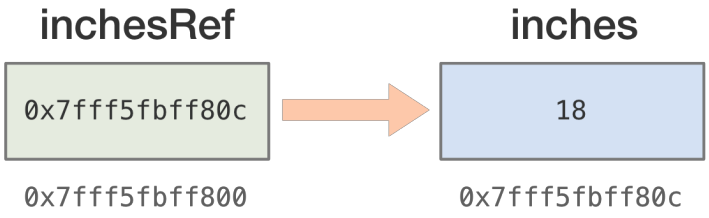
```
// Defines foo to be pointer to value of type int.
int *foo;

// Defines bar to be pointer to value of type pointer to an int.
int **bar;
```

- Use the **&** (*address-of*) operator to obtain a variable's address.

```
int inches = 18;
// Defines inchesRef as a pointer to a value of type int,
// and initializes it to the address of the memory location
// where the value of inches is stored.
int *inchesRef = &inches;

// You can now use inchesRef as a variable. Remember though,
// it contains the address of an int, not an int.
printf("Address of inches: %p\n", inchesRef);
```



© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

Pointer Dereferencing

- Use the ∗ *dereference operator* to access the value at a given address

```
// Use the ∗ operator to dereference inchesRef.
printf("address: %p, value: %d\n", inchesRef, ∗inchesRef);

// Dereference inchesRef to modify its value.
∗inchesRef += 3;

printf("value: %d\n", inches); // Prints ‘value: 21’.
```

- Note that ∗ and **&** are inverse operations.

```
// Silly (though correct) example that uses & to take address of
// inches and then immediately derefences the address with ∗.
//
printf("value: %d\n", ∗&inches);
```

- Potential point of confusion: the effect of the ∗ *declarator* is different from the effect of the ∗ *operator*.

  - As a declarator, ∗ changes the semantics of a declaration or definition at *compile time*.

```
// Here, ∗ modifies the type of foo.
int ∗foo;
```

  - As an operator, ∗ performs a dereference operation at *run time*.

```
// Here, ∗ dereferences foo.
∗foo += 42;
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Arrays

Defining Arrays

An array definition statement has the following effects:

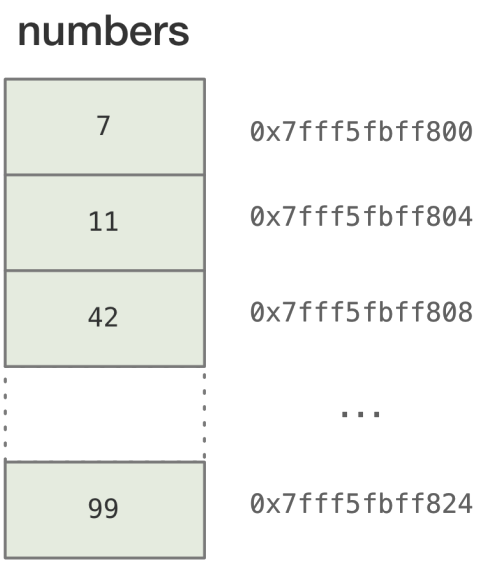
- Allocates a chunk of memory large enough to hold all of the array's elements.
- Makes the name of the array a symbol for the base address of the allocated memory.

```
// Allocates enough contiguous memory for ten ints, and makes
// numbers a label for the address of the first element.

int numbers[10];

// Use subscript notation to reference individual elements
// of an array.

numbers[0] = 7;
numbers[1] = 11;
numbers[2] = 42;
```



© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

Array Initialization

- Array elements are uninitialized unless you provide a *static initializer list*.
- If the initializer list is shorter than the array, the remaining elements are initialized to zero.

```
// Uninitialized. Values of elements undefined.
int a[10];

// Partial list. Elements without explicit initial values
// are initialized to zero.
int b[10] = { 7, 11, 42, 86, 99 };

// Initialized to all zeroes.
int c[10] = { };

// If you omit the array bound, the array is sized automatically.
//
// Note: for multi-dimensional arrays, only the leftmost bound
// can be omitted.
//
int d[] = { 7, 11, 42, 86, 99 };
```

- An array cannot be initialized with another array.

```
int first[] = { 1, 2, 3 };

int second[] = first; // Illegal.

// Remember that the name first is a symbol for the base address
// of the first array. The definition of second allocates storage
// at a different memory location. Your code can't modify either
// array's memory location.
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

The const Qualifier

Variables declared with the `const` type qualifier are immutable.

The compiler requires `const` variables to be explicitly initialized, and doesn't allow them to be modified.

```
// Here we used const to qualify the data type, so the compiler
// will consider pi to be immutable.
const float pi = 3.1416;

pi += 2;  // Illegal. Compilation fails on this line.
```

Pointers to const

The compiler will warn on attempts to initialize a non-const pointer with a const variable.

```
// Here, the declaration of sneakyRef doesn't include the
// const qualifier, so it doesn't correctly match the original
// declaration of pi.
//
float *sneakyRef = &pi;  // Compiler warns on this line.

// Pointer type declarations must match the type of the value
// being assigned, including the const qualifier.
//
const float *goodRef = &pi; // Okay.
```

API Contracts

The `const` qualifier provides an important guarantee to API callers. For example, the `strcpy` function (declared in **string.h**) copies characters from one argument to the other. Can you tell which argument is which?

```
char *strcpy(char *, const char *);
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Strings

- C doesn't have a built-in string data type. Instead, strings are represented as null-terminated arrays of characters.

```
// Initializes letters1 to the string abc (null-terminated).
char letters1[] = { 'a', 'b', 'c', '\0' };
```

- String literals are stored as null-terminated arrays of characters.

```
// The string literal below has the same effect as the
// initializer list shown in the previous example.
char letters2[] = "abc";
```

- Like other arrays, strings are passed by address.

```
// Use the %s format specifier for strings.
printf("%s\n", letters2);
```

String Library Functions

Below are some examples of string functions declared in string.h.

```
// Appends a copy of the characters in the second string to the end
// of the first string, and appends a null character at the end.
char *strcat(char *, const char *);

// Compares two strings. Returns a positive number if the first string
// is lexically greater than the second, zero if they're equal, or
// else a negative number.
int strcmp(const char *, const char *);

// Copies the characters in the second string to the memory pointed
// to by the first argument.
char *strcpy(char *, const char *);

// Returns the length of a null-terminated string. Note: size_t is
// a typedef for unsigned long.
size_t strlen(const char *);
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Pointer Arithmetic

- Array subscripts are translated into pointer notation at compile time.
- The notations are interchangeable, as shown below.

```
char name[] = "Fred Smith";
char *charRef = name;

// Prints: address is 0x7fff5fbff7dd, value is F
printf("address is %p, value is %c\n", charRef, charRef[0]);
printf("address is %p, value is %c\n", name, *name);

// Prints: address is 0x7fff5fbff7de, value is r
printf("address is %p, value is %c\n", &charRef[1], charRef[1]);
printf("address is %p, value is %c\n", name + 1, *(name + 1));

charRef += 5;  // Moves the pointer past the substring “Fred “

// Prints: address is 0x7fff5fbff7e2, value is Smith
printf("address is %p, value is %s\n", charRef, charRef);
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Arrays as Arguments

- Pointers and arrays can be used interchangeably as function arguments.
- Since arguments are passed by value in C, the value passed will be an address in either case.

```
void PrintInReverse(char *s)
{
    char *currChar;

    // Find the end.
    for (currChar = s; *currChar; currChar++); // Empty loop body.

    // Print backwards.
    for (currChar--; currChar != s; currChar--)
        printf("%c", *currChar);

    // Print the last char.
    printf("%c\n", *currChar);
}

int main()
{
    char name[] = "star was noel";
    PrintInReverse(name);       // Prints: leon saw rats
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Structures

- Structures aggregate several variables into a single unit.
- Individual elements are referred to as **structure members**.
- Structures are typically declared in header files.
- In declarations, the keyword `struct` is followed by an optional tag.

```
// A structure declaration

struct Person {
    char *firstName;
    char *lastName;
    int age;
};
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Working with Structures

- Both the `struct` keyword and the structure tag are required when defining instances.
- Use the `.` (dot) operator to access individual members.
- As with arrays, you can provide a static initializer list in curly braces.

```
void DoStuffWithStructs(void)
{
    // Define a variable of type struct Person.
    struct Person fred;

    // Use the '.' operator to access members.
    fred.firstName = "Fred";
    fred.lastName = "Smith";
    fred.age = 32;

    // Initializing a struct.
    struct Person sally = { "Sally", "Jones", 27 };

    // Passing a struct as an argument.
    PrintPerson(sally);
}

void PrintPerson(struct Person aPerson)
{
    printf("First name: %s\n"
           "Last name: %s\n"
           "Age: %d\n",
           aPerson.firstName,
           aPerson.lastName,
           aPerson.age);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Using typedef with Structures

Structure declarations are often combined with `typedef` statements to streamline usage, as shown below.

```
struct Person {
    char *firstName;
    char *lastName;
    int age;
};

// Allows Person to be used as a label for struct Person.
typedef struct Person Person;
```

You can optionally combine the two declarations shown above in a single `typedef` statement (in which case the structure tag can be omitted).

```
typedef struct {
    char *firstName;
    char *lastName;
    int age;
} Person;
```

You can use `Person` as a synonym for `struct Person` anywhere the `typedef` is visible.

```
Person CreatePerson(void)
{
    Person p = { };
    return p;
}
```

NOTE: `typedef` statements are also commonly used with `enum` declarations.

```
enum PetType {
    Dog,
    Cat,
    Bird
};
typedef enum PetType PetType;
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Storage Classes

Automatic

Variables and arguments defined in the scope of a function are *automatic*, meaning they're allocated and deallocated on the stack at run time.

```
void Capitalize(char *word) // word and delta are both automatic.
{
    const char delta = 'a' - 'A';

    word[0] -= delta;
}
```

Static

External variables (variables declared outside the scope of a function) and variables declared with the **static** storage class specifier are allocated at compile time and never deallocated.

```
const char Delta = 'a' - 'A'; // Delta is external, therefore static.

void Capitalize2(char *word)
{
    word[0] -= Delta;
}
```

Local variables are automatic by default, but can be explicitly declared static.

```
void Capitalize3(char *word)
{
    static const char delta = 'a' - 'A'; // Here delta is static.

    word[0] -= delta;
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Storage Classes (cont.)

The `static` keyword limits external declarations to **file scope**. Therefore, symbols explicitly declared `static` needn't be globally unique.

```
// Limits visibility of Bar to the current file.
static const int Bar = 42;

// Allows other files to contain functions named foo.
static void Foo(void)
{
    // ...
}
```

Extern

The `extern` keyword can be used to export global symbols. Makes global variables visible to other files.

```
// Definition in .c file.
const char Delta = 'a' - 'A';

// Declaration in .h file.
extern const char Delta;
```

For libraries, `extern` makes function names visible to code outside the library.

```
extern void Capitalize(char *word);
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Heap Allocation

- So far we've considered two types of memory allocation.
  - Automatic – 'scratchpad' memory; automatically allocated on the stack on function entry and deallocated on function exit.
  - Static – allocated at compile time; never deallocated.
- Allocating memory dynamically in heap splits the difference between automatic and static allocation.
  - Allows your program to reserve memory on demand and deallocate when no longer needed.
  - Gives your program control over object lifetimes, allowing it to manage its own memory footprint.
- C library provides functions (declared in **stdlib.h**) for allocating and freeing memory in heap, including the following:

```
// Allocates size bytes. Returns a pointer to the allocated memory.
void *malloc(size_t size);

// Allocates and zero fills count contiguous objects of size bytes
// each. Returns a pointer to the allocated memory.
void *calloc(size_t count, size_t size);

// Deallocates previously allocated memory pointed to by ptr.
void free(void *ptr);
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Heap Allocation Example

The example below wraps a call to the `Capitalize` function shown earlier in this section to avoid one of its many bugs: passing a string literal to `Capitalize` would cause a crash.

```
char *CreateCapitalizedWord(const char *word)
{
    // Compute size. Add 1 for '\0' delimiter.
    size_t size = strlen(word) + 1;

    // Allocate storage and store address in local variable.
    char *newWord = malloc(size);

    // Copy contents of word to newWord (including '\0' terminator).
    strcpy(newWord, word);

    // Capitalize newWord.
    Capitalize(newWord);

    // Return the address of the capitalized word.
    return newWord;
}

int main(int argc, const char *argv[])
{
    char *myWord = CreateCapitalizedWord("hello");
    printf("%s\n", myWord);

    // Apple uses naming conventions to convey info about memory
    // management. For example, function names that begin with
    // 'create' are assumed to return the address of newly allocated
    // memory the caller is responsible for freeing.
    //
    free(myWord);

    return 0;
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Structure Pointers

C provides syntactic sugar for structure pointers to avoid otherwise clumsy use of pointer dereferencing syntax.

```
Person *fred = calloc(1, sizeof(Person));

// We have to use parens to force the * operator to be
// evaluated before the dot operator.

(*fred).firstName = "Fred";
(*fred).age = 32;

// Rewriting the preceding two lines to use the structure
// pointer dereference operator improves clarity.

fred->firstName = "Fred";
fred->age = 32;
```

Here's a more complete example.

```
Person *CreatePersonWithName(char *firstName, char *lastName)
{
    // Using calloc here so person will be zero filled.
    Person *person = calloc(1, sizeof(Person));

    person->firstName = firstName;
    person->lastName = lastName;

    return person;
}

void ShowStructurePointerExample()
{
    Person *willy = CreatePersonWithName("Willy", "Orca");

    printf("First name: %s\nLast name: %s\nAge: %d\n",
           willy->firstName,
           willy->lastName,
           willy->age);

    free(willy);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Lab – Pointers and Structures

OVERVIEW

Add new **.c** and **.h** files to the **ANSI C Labs** Xcode project files to experiment with pointers and structures.

PART 1

1. Add the following new files to the **ANSI C Labs** Xcode project: **Person.h**, **Person.c**, **Pointers.h**, **Pointers.c**, **Utilities.h**, and **Utilities.c**. Add code to the new files as follows:

    1.1. In **Pointers.h**, declare functions named `RunPointers01`, `RunPointers02`, `RunPointers03`, `RunPointers04`, and `RunPointers05`. The functions should take no arguments and return `void`.

    1.2. In **Utilities.h**, write a `#define` preprocessor directive named `PRINT_FUNCTION_NAME` that uses `printf` to print the name of the current function, followed by a newline. (Requires including **stdio.h**.)

    1.3. Define the functions declared in step **1.1** in **Pointers.c**. The initial implementations should use the macro defined in step **1.2** to print the function's name. (Requires including **Utilities.h**.)

    1.4. In **main.c**, call the new functions. Don't forget to include the header in which you declared them.

    1.5. Build and run the tool. You should see the names of the functions printed on the console. Make sure this is working correctly before proceeding to the next step.

2. Declare two new functions in **Person.h**, and implement them in **Person.c**.

    2.1. Declare a `Person` structure in **Person.h** with the following members: `firstName` and `lastName`, both of type `char *`; and `age` of type `int`. Add a `typedef` statement that makes `Person` a label for `struct Person`.

    2.2. In **Person.h**, declare a function named `AllocPerson` that takes no arguments and returns `Person *`. Write an implementation of `AllocPerson` in **Person.c** that returns a dynamically allocated `Person` structure initialized to all zeroes. (Requires including **stdlib.h**.)

    2.3. In **Pointers.c**, add code to `RunPointers01` that initializes a local variable, `p1`, with the value returned by `AllocPerson`. Write a `printf` statement to print out each member of `p1`. Build and run. The resulting values should be **(null)**, **(null)**, and **0**. (Requires

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

including **stdio.h** and **Person.h**.)

2.4. In **Utilities.h**, declare a function named `CopyString` that takes one argument of type `const char *` and returns `char *`. In **Utilities.c**, implement `CopyString` to dynamically allocate memory for a copy of the string passed as an argument, copy the characters from the original string to the new string, and then return the new string. (Requires including **string.h** and **stdlib.h**.)

2.5. In **Person.h**, declare a function named `InitPerson` that returns `Person *`, and takes the following arguments: `self`, of type `Person *`; `firstName`, of type `const char *`; `lastName`, of type `const char *`; and `age`, of type `int`. Write an implementation of `InitPerson` in **Person.c** that initializes the members of `self` with the provided argument values, and returns `self`. (Note: You should use the `CopyString` function when setting `firstName` and `lastName`. Why might that be important?)

2.6. Add code to `RunPointers01` that calls `InitPerson` with the following arguments: `p1`, **"Sue"**, **"Wilson"**, and **29**. Add another `printf` statement on the next line to print the members of `p1`. Add a line at the end to call `free` to deallocate `p1`. Build and run, and verify that the members of `p1` match the values passed to `InitPerson`. (Requires including **stdlib.h**.)

2.7. Add code at the bottom of `RunPointers01` to define another variable, `p2`, and initialize it with a fully initialized instance of `Person` created by calling `AllocPerson` with the return value from a nested call to `InitPerson` as its first argument, and the following additional arguments: **"Fred"**, "**Smith"**, **32**. Add code to print the members of `p2` and free it when done. Build and run to verify that value of `p2` are as expected.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

PART 2

1. Add functionality to Person.c to generate a string containing a description of a Person. The format of the description string will be as follows:

   <last name>, *<first name>*, **Age:** *<age>*

   1.1. Write functions for getting a person's first and last names with the following signatures:

   ```
   const char *FirstNameFromPerson(Person *self)
   const char *LastNameFromPerson(Person *self)
   ```

   The functions should return the string **N/A** if the corresponding name is currently `NULL`. Define a global constant named `NotAvailable` initialized to the string **N/A** to use for the return value.

   1.2. Write a function named `DescriptionLength` to calculate the length of a description string. `DescriptionLength` should take an argument named **self** of type `Person *` and return `size_t`. Define and initialize global constants as follows:

   ```
   const char *Separator = ", ";
   const char *AgeLabel = "Age: ";
   ```

   Use the following code to obtain a string representation of the person's age:

   ```
   char ageString[4];
   sprintf(ageString, "%d", self->age);
   ```

   Then calculate and return the sum of the lengths of all the components of the description string, using the `strlen` function to obtain the length of each component.

   1.3. Declare a function with the following signature in **Person.h**:

   ```
   char *CreateDescriptionOfPerson(Person *self)
   ```

   This function should return a string composed of a person's first name, last name, and age. Use `sprintf` to compose the string. The first argument you pass to `sprintf` should be a buffer large enough to hold the entire string, including the null character.

   1.4. Add code to `RunPointers02` to allocate and initialize a person with the following values: **"Fred"**, **"Smith"**, and **32**. On the next line, call `CreateDescriptionOfPerson` passing the newly allocated person. Capture the return value in a local variable named `description`. Add a `printf` statement on the next line to print the description string, followed by another line that deallocates the string. Build and run to verify that the content of the description string is as expected.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Lab Solutions – Pointers and…

PART 1

Person.h

```
struct Person {
    char *firstName;
    char *lastName;
    int age;
};
typedef struct Person Person;

Person *AllocPerson(void);
Person *InitPerson(Person *person,
                   char *firstName,
                   char *lastName,
                   int age);
```

Person.c

```
#include <stdlib.h>

#include "Utilities.h"
#include "Person.h"

Person *AllocPerson(void)
{
    return calloc(1, sizeof(Person));
}

Person *InitPerson(Person *self,
                   char *firstName,
                   char *lastName,
                   int age)
{
    self->firstName = CopyString(firstName);
    self->lastName = CopyString(lastName);
    self->age = age;

    return self;
}
```

Pointers.h

```
void RunPointers01(void);
void RunPointers02(void);
void RunPointers03(void);
void RunPointers04(void);
void RunPointers05(void);
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Pointers.c

```
#include <stdio.h>

#include "Utilities.h"
#include "Person.h"

void RunPointers01(void)
{
    PRINT_FUNCTION_NAME;

    Person *p1 = AllocPerson();
    printf("Name: %s %s, Age: %d\n", p1->firstName, p1->lastName, p1->age);

    InitPerson(p1, "Sue", "Wilson", 29);
    printf("Name: %s %s, Age: %d\n", p1->firstName, p1->lastName, p1->age);

    Person *p2 = InitPerson(AllocPerson(), "Fred", "Smith", 32);

    printf("Name: %s %s, Age: %d\n", p2->firstName, p2->lastName, p2->age);

    free(p1);
    free(p2);
}

void RunPointers02(void)
{
    PRINT_FUNCTION_NAME;
}

// ...

void RunPointers05(void)
{
    PRINT_FUNCTION_NAME;
}
```

Utilities.h

```
#define PRINT_FUNCTION_NAME printf("\n%s\n----------\n", __func__)

char *CopyString(const char *string);
```

Utilities.c

```
#include <string.h>
#include <stdlib.h>

#include "Utilities.h"

char *CopyString(const char *string)
{
    if (string == NULL) return NULL;

    char *copy = NULL;
    copy = malloc(strlen(string) + 1);
    strcpy(copy, string);

    return copy;
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

main.m

```
#include <stdio.h>

#include "ANSICBasics.h"
#include "Pointers.h"

int main(int argc, const char *argv[])
{
    RunANSICBasics01();
    RunANSICBasics02();

    RunPointers01();
    RunPointers02();
    RunPointers03();
    RunPointers04();
    RunPointers05();

    return 0;
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 2

Person.h

```
char *CreateDescriptionOfPerson(Person *person);
```

Person.c

```
// ...

#include <string.h>
#include <stdio.h>

const char *Separator = ", ";
const char *AgeLabel = "Age: ";
const char *NotAvailable = "N/A";

// ...

const char *FirstNameFromPerson(Person *self)
{
    if (self->firstName == NULL) {
        return NotAvailable;
    }

    return self->firstName;
}

const char *LastNameFromPerson(Person *self)
{
    if (self->lastName == NULL) {
        return NotAvailable;
    }

    return self->lastName;
}


size_t DescriptionLength(Person *self)
{
    char ageString[4];
    sprintf(ageString, "%d", self->age);

    size_t length = (strlen(FirstNameFromPerson(self)) + strlen(Separator) +
                     strlen(LastNameFromPerson(self)) + strlen(Separator) +
                     strlen(AgeLabel) + strlen(ageString));
    return length;
}

char *CreateDescriptionOfPerson(Person *self)
{
    size_t length = DescriptionLength(self);
    char *description = malloc(length + 1);

    sprintf(description, "%s%s%s%s%s%d",
            LastNameFromPerson(self), Separator, FirstNameFromPerson(self),
            Separator, AgeLabel,
            self->age);

    return description;
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Pointers.c

```
void RunPointers02(void)
{
    PRINT_FUNCTION_NAME;

    Person *p1 = InitPerson(AllocPerson(), "Fred", "Smith", 32);
    const char *description = CreateDescriptionOfPerson(p1);

    printf("%s\n", description);

    free(p1);
    free(description);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Objective-C Fundamentals

Objective-C syntax is derived from SmallTalk, making it a bit different than other commonly-used object-oriented programming languages. This section will give you an overview of syntactic differences, using Java as a point of comparison. It will also introduce you to a few value classes in the Foundation framework, in particular, **NSString** and **NSNumber**.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Topics/Goals

By the end of this module you'll know the basics of working with Objective-C and the Foundation framework, and be able to do the following:

- Create classes and objects
- Declare instance variables and methods
- Use message expressions to invoke methods
- Use introspection to perform dynamic checking
- Write custom initializer methods
- Write accessor methods
- Create and invoke multi-argument methods
- Work with instances of `NSString` and `NSNumber`

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Glossary

Term	Definition
message	An expression that dynamically dispatches a method.
selector	A method name. Used to 'select' a dynamically dispatched method by looking it up in a dispatch table.
dispatch table	A table owned by an object's class containing method names (selectors) paired with pointers to their implementations.
designated initializer	`init...` method that performs the actual initialization for instances of a given class. Other `init...` methods in the same class directly or indirectly call this one.
interface	Declaration of a class or category's methods, properties, and, for classes only, instance variables.
implementation	Method definitions for a given class or category. In modern Objective-C, may also contain instance variable declarations.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Modules (Xcode 5)

- Clang 3.4 adds `@import` directive to Objective-C as alternative to `#import`.
  - Improves compilation time.
  - Reduces fragility of preprocessor markup.

```
@import XCTest;

// Statement above can be used in place of line below.
//
// #import <XCTest/XCTest.h>
```

- Module import functionality is replacement for preprocessor inclusion mechanism.
  - Ensures that headers are only read once.
- When modules feature is enabled, Clang automatically translates `#import` and `#include` statements into module imports.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Compiler Directives

Objective-C is a superset of ANSI C. Many of its features are added to the base C language via *compiler directives*.

Examples of Compiler Directives that Make Use of @

`@interface ... @end`
Used to declare a class, category, or class extension.

`@implementation ... @end`
Surrounds the implementation of a class or category.

`@protocol ... @end`
Used to declare a protocol (similar to a Java interface).

`@optional`
Specifies an optional section of a protocol.

`@property`
Used to declare a property.

Examples of Literals that Make Use of @

Literals are also prefixed with `@` symbols.

`@"This is a string literal."`
An `NSString` literal.

`@12`
An `NSNumber` literal.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Messages

A method is invoked via a *message expression*.

Objective-C message expression:

```
// Send a 'description' message to someObject.
[someObject description];
```

Java equivalent:

```
someObject.toString();
```

Message Expression Semantics

The target of a message expression is called the ***receiver***.

```
[receiver message];
```

The message itself is composed of a ***selector*** (method name) and its arguments (if any).

```
[someObject isEqual:someOtherObject]; // Selector is isEqual:
```

Message Dispatching

Message expressions are interpreted dynamically by the Objective-C *runtime system*.

- The runtime system searches the receiver's *class hierarchy* to find the method implementation.
- If found, the method is then invoked by the runtime system.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

What is an Object?

Conceptually, an object is a dynamically allocated instance of any structure that has as its first member a pointer to a class.

```
// An object is a dynamically allocated instance of a struct
// conceptually similar to the one shown below.

struct ObjC_Person {
    Class isa;     // Class is type for 'pointer to class.'
    char *firstName;
    char *lastName;
    int age;
};
```

The `isa` pointer is inherited from the `NSObject` root class.

NOTE: An object's structure members are ordinarily referred to as ***instance variables***, rather than *members*.

Data Types

Objective-C supports both static and dynamic typing for objects.

```
// Static typing.
Person *fred;

// Dynamic typing.
id bob;
```

Static typing allows the compiler to do more checking, and therefore is more commonly used. *(Later in this section we'll explore where and how dynamic typing can be useful.)*

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

What is a Class?

An Objective-C class provides storage (in heap) for metadata about its instances, such as:

- Instance variables (names, types, and offsets)
- Methods (names, types, and pointers to implementations)
- Protocols
- Instance size

The class structure also includes information about itself, including its:

- Name
- Superclass

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Method Syntax

- Methods are prefixed with plus (**+**) or minus (**-**).
  - Distinguishes *class methods* from *instance methods* (which can otherwise be identical).
- Data types are enclosed in parentheses.

Examples (from NSObject.h)

```
// Instance methods...

- (id)init;

- (NSString *)description;

// Class methods...

+ (id)alloc;

+ (NSString *)description;
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Method Arguments

- Arguments are preceded by colon (**:**) characters.
  - One colon per argument.
  - Colons are considered part of the method name.

```
// From NSObject.h
- (BOOL)isEqual:(id)object;

// Custom method
- (void)setFirstName:(NSString *)name;
```

Example: Calling Methods with a Single Argument

```
// Custom method
[self setFirstName:@"Fred"];


// NSObject method
if ([self isEqual:foo]) {
    // Do something...
};


// NSString method
NSString *s1 = @"Hello World!";
NSString *s2 = [s1 substringToIndex:5]; // Sets s2 to "Hello"
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Multi-Argument Methods

- Method names may be composed of more than one component.
  - Each component ends with a colon, followed by an argument.

Declaring Methods with Multiple Arguments

```
// Declared in NSString.h...

- (NSString *)stringByPaddingToLength:(NSUInteger)newLength
                           withString:(NSString *)padString
                      startingAtIndex:(NSUInteger)padIndex;

// Custom method

- (id)initWithFirstName:(NSString *)aFirstName
               lastName:(NSString *)aLastName;
```

Calling Methods with Multiple Arguments

```
// Custom method

[[Person alloc] initWithFirstName:@"Fred"
                         lastName:@"Smith"];

// NSString method

NSString *s = [s1 stringByPaddingToLength:10
                               withString:@". "
                          startingAtIndex:1];
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Class Methods

- At runtime, an Objective-C class is a dynamically allocated structure that has an **isa** pointer.
  - Allows the class to receive messages at runtime.
- Class methods typically used for:
  - Creating instances
  - Accessing shared, global resources.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Declaring a Class

A class's instance variables, methods, and properties, as well metadata about its superclass and protocols, are declared between an `@interface ... @end` pair.

Example Class Declaration

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

@interface Person : NSObject // Declares Person as NSObject subclass.
{
    // Instance variables.
    // Note: Underscore prefixes are conventional, but not required.

    int _age;
    NSString *_firstName;
}

// Methods.
// Note: Getter methods should not be prefixed with 'get'.

- (int)age;
- (void)setAge:(int)anAge;

- (NSString *)firstName;
- (void)setFirstName:(NSString *)aName;

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Implementing a Class

A class's methods are defined between an `@implementation...@end` pair.

```
// Imports the Person class's interface.
#import "Person.h"

// Implementation of the Person class.
@implementation Person

- (int)age
{
    return _age;
}

- (void)setAge:(int)anAge
{
    _age = anAge;
}

- (NSString *)firstName
{
    return _firstName;
}

- (void)setFirstName:(NSString *)aName
{
    _firstName = aName;
}

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Instance Variable Visibility (1)

Instance Variables Declared in `@interface`

- Default visibility: **protected**.
- Use *visibility modifiers* to override default.

```
@interface Person : NSObject
{
    // protected ivars (Default visibility)
    BOOL _hipster;
    int _streetCred;

@private
    // private ivars
    NSString *_firstName;
    NSNumber *_salary;
    int _age;
}
// ...
@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Instance Variable Visibility (2)

Instance Variables Declared in `@implementation`

- Default visibility: **private**.

```
@implementation Person
{
    // private ivars (Default visibility)
    NSString *_firstName;
    NSNumber *_salary;
    int _age;

@protected
    // protected ivars
    BOOL _hipster;
    int _streetCred;
}
// ...
@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Method Visibility

- Objective-C doesn't provide visibility modifiers for methods or for C functions.
  - Omitting declarations in public headers makes methods and functions *conceptually* private.
- *For C functions only:* `static` keyword restricts scope of function to implementation file (File scope).

```
static size_t DescriptionLength(Person *self)
{
    // ...
}
```

IMPORTANT: *It's possible for functions and methods not in scope at compile time to be invoked successfully at run time.*

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Naming Conventions

Apple's engineering teams place a strong emphasis on code readability. They've produced carefully documented guidelines, and as you'll discover, are remarkably consistent about following them.

Your code will be more readable if you follow suit.

You can read the guidelines online at the following URL, or search for **'Coding Guidelines'** in Xcode's Documentation Browser.

https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CodingGuidelines

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Creating Instances

- `+alloc` method (inherited from **`NSObject`**) is used generically to allocate instances of **`NSObject`** subclasses.
- Performs the following steps:
  - Allocates memory for new instance and initializes to all zeroes.
  - Sets instance's `isa` pointer to the address of the receiver's class.
  - Initializes the instance's reference count (*retain count*).
  - Returns the new instance.
- New instances are not considered fully initialized until they've received an `init...` message.
- Calls to `alloc` must be paired with calls to an initializer method to ensure instances are safe to use prior to sending them other messages.

```
Person *fred = [[Person alloc] init];

[fred setFirstName:@"Fred"];
```

- If an initializer fails, it must free the uninitialized memory and return `nil`.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Messages to nil

- Messages to `nil` are guaranteed by the runtime system to return `nil`. (Or zero if the return value is an integer or float type, though things are a little dicier if the return value is a struct.)
- To illustrate one of many situations in which this can be useful, here's an example of an `init...` method that can sometimes fail.

```
// If file 'Foobar' doesn't exist or is unreadable, foo will be nil.

NSArray *foo = [[NSArray alloc] initWithContentsOfFile:@"Foobar"];

// Since foo is nil, result of any message to foo is also nil.
//
// Note: nil is handled gracefully by NSLog.

NSLog(@"Last objects is %@", [foo lastObject]);
```

Resulting `NSLog` output:

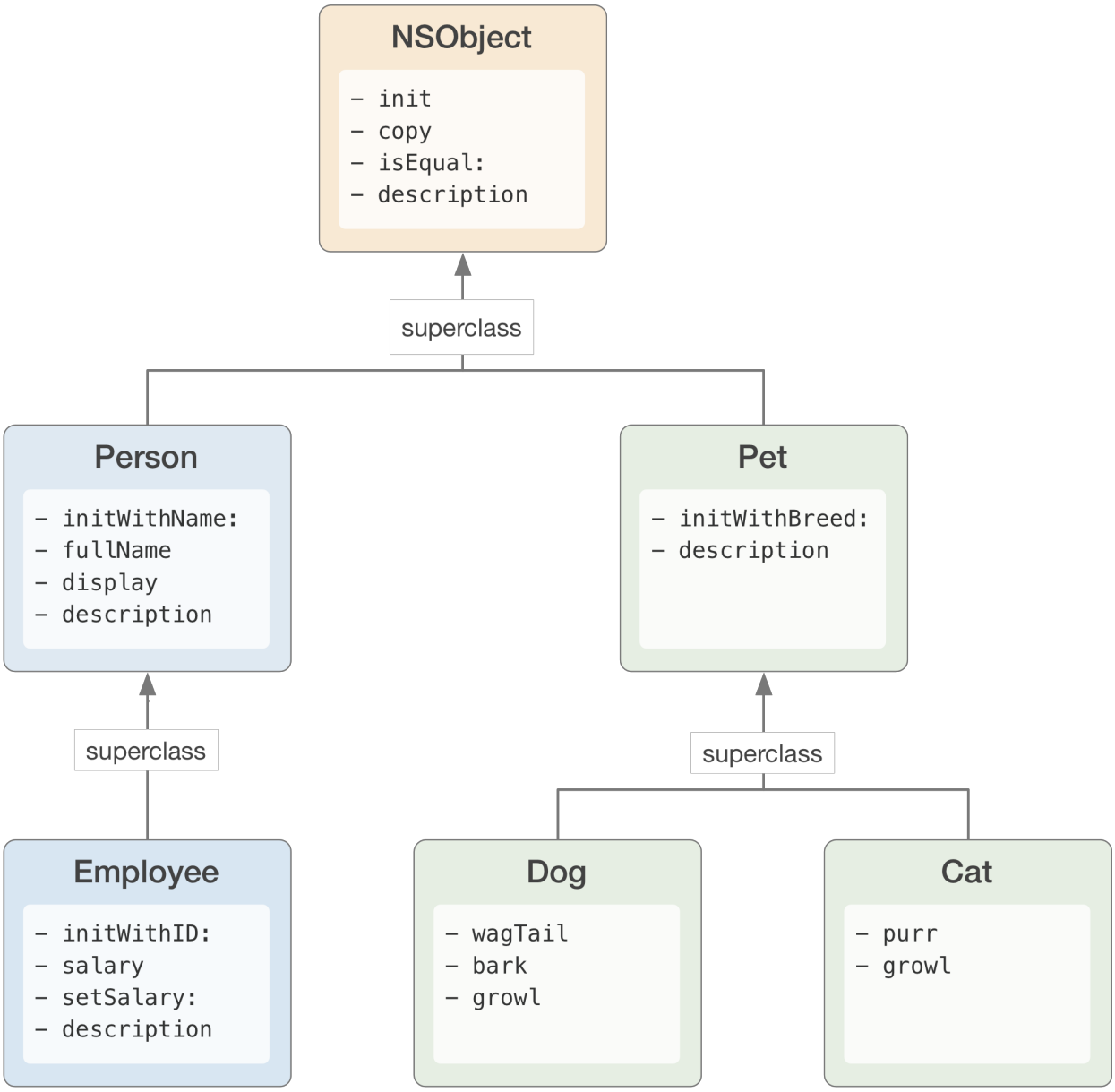
```
2013-10-18 15:56:05.605 xctest[21291:303] Last object is (null)
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Class Hierarchy

Classes form an in-memory hierarchy at runtime, as shown in figure below.



© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Instance Variable Inheritance

Instance data structures are composited at compile time by combining the inherited instance variable declarations into a single **struct**, conceptually like those shown in the following table.

How Compiler Translates ivar Declarations (Conceptual)

Declaration	Compiler-generated `struct`

`@interface NSObject <NSObject>`
`{`
`  Class isa;`
`}`
	`struct NSObject {`
	`  Class isa;`
	`};`

`@interface Pet : NSObject`
`{`
`  NSString *_name;`
`}`
	`struct Pet {`
	`  Class isa;`
	`  NSString *_name;`
	`};`

`@interface Dog : Pet`
`{`
`  NSString *_breed;`
`}`
	`struct Dog {`
	`  Class isa;`
	`  NSString *_name;`
	`  NSString *_breed;`
	`};`

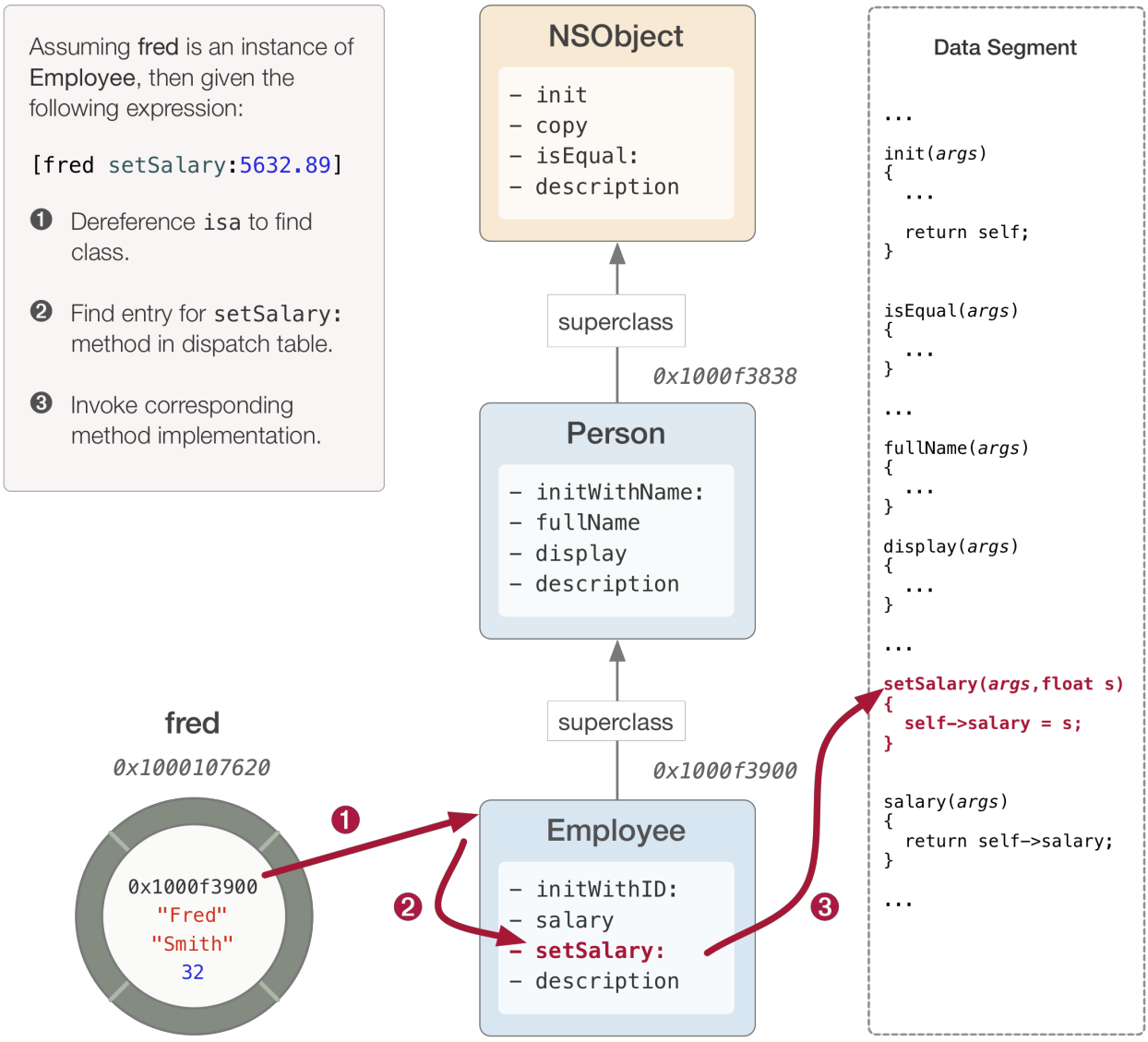
© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Message Dispatching

Messages are dispatched dynamically by the Objective-C runtime system, as shown in the figure below. This is often referred to as *dynamic binding*.

Dispatching a Message



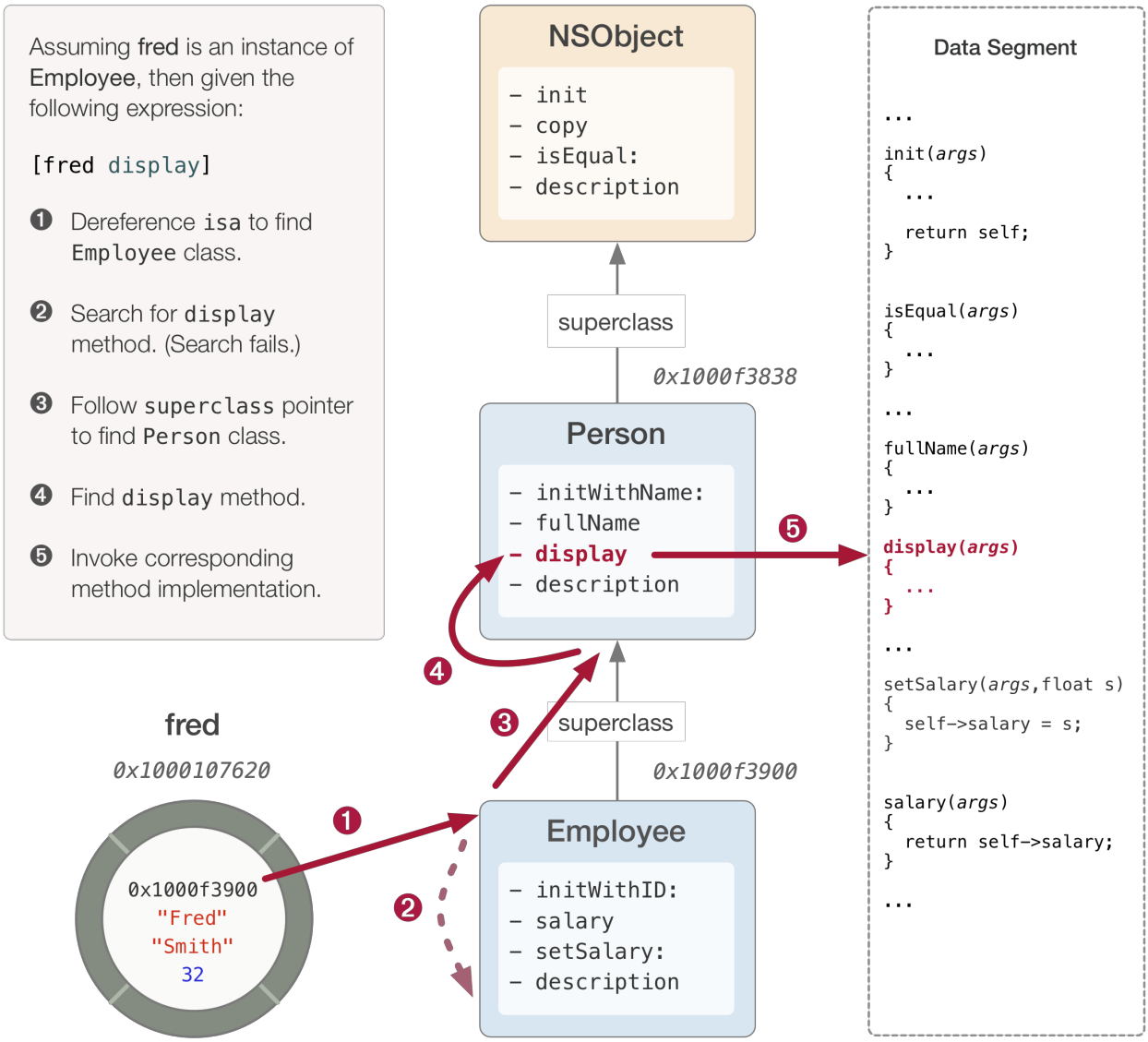
© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Method Inheritance

Method inheritance is implemented by the Objective-C runtime system's message dispatching mechanism, as shown in the figure below.

Dispatching an Inherited Method



© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Initializer Methods

- Must do as follows:
  - Call its superclass's *designated initializer*.
  - Replace the current value of `self` with the value returned by the superclass initializer.
  - If initialization fails, must return `nil`.
  - Otherwise, must return the object that was initialized (ordinarily, `self`).
- Designated initializer
  - Generally the `init...` method that takes the most arguments.
  - Performs actual initialization; other `init...` methods in the same class directly or indirectly call this one.
  - Should be called by subclass designated initializers.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Initializer Example

```
- (id)initWithName:(NSString *)aName
{
    // Capture value returned by super init.
    self = [super init];

    // If return value is nil, object has already been
    // deallocated, so short-circuit initialization.
    if (self == nil)
        return nil;

    // Perform custom initialization.
    _name = [aName copy];

    // Return the initialized object. Note that in theory
    // this can an object other than self.
    return self;
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

The NSString Class

- Instances of **NSString** are immutable.
- Use mutable subclass **NSMutableString** for instances with dynamically modifiable content.

Example: NSString init... Methods

```
//
// Allocates and initializes an empty string.
//
NSMutableString *s1 = [[NSMutableString alloc] init];


char *cString = "Foo";
//
// An NSString is essentially a wrapper for a C string.
//
NSString *s2 = [[NSString alloc] initWithUTF8String:cString];


NSString *name = @"Fred";
//
// Uses printf-style format string with additional support
// for object types (using the %@ format specifier).
//
NSString *s3 = [[NSString alloc] initWithFormat:
                @"Length of %@ is %ld",
                name, [name length]];
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

NSString Factory Methods

- Many Foundation classes provide *convenience creation methods*.
  - Also referred to as *factory methods*.
  - Wrappers for **`alloc`** and corresponding **`init...`** method.
  - Streamline calling code.

Example: NSString Convenience Methods

```
//
// Calls alloc and init. Returns empty mutable string.
//
NSMutableString *s1 = [NSMutableString string];


char *cString = "Foo";
//
// Calls alloc and initWithUTF8String:.
//
NSString *s2 = [NSString stringWithUTF8String:cString];


NSString *name = @"Fred";
//
// Calls alloc and initWithFormat:.
//
NSString *s3 = [NSString stringWithFormat:
                @"Length of %@ is %ld",
                name, [name length]];
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

The NSArray Class

NSArray objects are indexed collections of objects *of any type* (type `id`).

- Can't contain non-object data or **nil** entries.
- Instances are immutable.
- Mutable subclass: **NSMutableArray**.

Example: Creating and Initializing Arrays

```
// C array of type id, with static initializer list
id objs[] = { @"Foo", @42, @"Bar" };
//
// Returns an NSArray that wraps C array
//
NSArray *a1 = [NSArray arrayWithObjects:objs count:3];

//
// Uses nil-terminated list to initialize array
//
NSArray *a2 = [NSArray arrayWithObjects:@"One", @"Two", nil];

//
// Mutable array with initial capacity of 3 elements
//
NSMutableArray *a3 = [NSMutableArray arrayWithCapacity:3];
[a3 addObject:@"Hello"];
[a3 addObject:@"World"];
[a3 addObject:@"!"];

// A mutable array's capacity can grow dynamically, as needed
[a3 insertObject:@"Say" atIndex:0];
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

NSArray Literals

Modern Objective-C provides literal expressions for creating and initializing instances of NSArray.

- Also supports array subscript notation.
- Compiler translates literal syntax into method invocations.

Example: NSArray Literal Syntax

```
//
// Compiler translates into code that calls arrayWithObjects:count:
//
NSArray *objs = @[ @"Foo", @42, @"Bar" ];

for (int i = 0; i < [objs count]; i++)
{
    //
    // Translates subscript into call to objectAtIndexedSubscript:
    //
    NSLog(@"%@", objs[i]);
}
```

Example: Mutable Arrays

```
//
// Using array literal as argument to a factory method
//
NSMutableArray *objs2 = [NSMutableArray arrayWithArray:@[@1, @2, @3]];

// Modifying array elements.
objs2[0] = @42;

//
// Another technique for creating a mutable array from a literal
//
// Note: object copying behavior explored in detail in next section.
//
NSMutableArray *objs3 = [@[@1, @2, @3] mutableCopy];
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Enumerating Arrays

Foundation declares the **NSFastEnumeration** protocol for objects that support enumeration (iteration) behavior.

- All Foundation collection classes conform to NSFastEnumeration.
- Compiler translates `for ... in` syntax into the necessary calls to methods declared in the protocol.
- Custom classes can support fast enumeration by adopting (implementing) the protocol.

Example: Fast Enumeration

```
NSArray *objs = @[ @"Foo", @42, @"Bar" ];
//
// Enumerating an array.
//
for (id currObj in objs)
{
    NSLog(@"%@", currObj);
}

//
// Assuming p1, p2, and p3 are Person objects...
//
NSArray *people = @[ p1, p2, p3 ];
//
// Using a strongly typed loop variable.
//
for (Person *currPerson in people)
{
    NSLog(@"%@", [currPerson firstName]);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Unit Tests with XCUnit

The SDK provides the **XCUnit** framework for writing unit tests.

- Modeled along the same lines as the JUnit framework.
- Nicely integrated with Xcode.
- For now, we'll use it as a convenience for running arbitrary code.

Example: Writing Tests

```
@import XCTest;

// By default, no .h file is created. Instead the @interface
// declaration is placed near the top of the .m file.

@interface UnitTests : XCTestCase

@end


@implementation UnitTests

// The setUp and tearDown methods are run immediately before
// and after each test... method. Their use is optional.

- (void)setUp {
    [super setUp];
}

- (void)tearDown {
    [super tearDown];
}

// Arbitrary test method. Will be run if signature matches the
// one shown below, and the name is prefixed with test.

- (void)test1
{
    // Custom code.
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Lab – Objective-C Basics

OVERVIEW

Create a new, Objective-C command line tool Xcode project as described below.

PART 1

1. Create a new Xcode project using the OS X **Command Line Tool** template. Select **Foundation** from the Type dropdown, and name the project **Objective-C Labs**. Add a Unit Test target as follows:

    1.1. Select the project icon in the Navigator.

    1.2. In the Project Editor's sidebar, select **Add Target...**, and select **Cocoa Unit Testing Bundle** (under **OS X | Other**) in the template chooser.

    1.3. Click **Next**, then enter **UnitTests** for **Product Name** and click **Finish**.

    1.4. You'll now see a UnitTests group in the Navigator containing **UnitTests.m**. That's where you'll write test methods to experiment with the other classes you create during the lab.

2. Create a **Person** class with the following features:

    2.1. Two instance variables of type **NSString *** named **_firstName**, and **_lastName**, and one instance variable of type **int** named **_age**.

    2.2. Pairs of accessor methods (getters and setters) for each instance variable. (Hint: try using Xcode's **Encapsulate** refactoring to save coding effort.)

3. In **UnitTests.m**, write a unit test method named **testPart01**, and do as follows:

    3.1. Define a local variable of type **Person *** named **fred**, and initialize it to point to a new instance of **Person**.

    3.2. Call each of **fred**'s setter methods to populate the object's instance variables.

    3.3. Write an **NSLog** statement that prints out the values of **fred**'s instance variables. Build and run to make sure the instance variables have the expected values.

    Note: Make sure to add **Person.m** to the **UnitTests** target.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 2

1. Implement the following features to the **Person** class, and write a unit test to test them out.

    1.1. Add a custom initializer with the following signature:

    ```
    - (id)initWithFirstName:(NSString *)firstName
                   lastName:(NSString *)lastName
                        age:(int)age;
    ```

    1.2. Add a **fullName** method that returns a string composed of a person's first and last names.

    1.3. Override the inherited **description** method to return a string containing a person's full name and age.

2. In **UnitTests.m**, write a test method named **testPart02** that does as follows:

    2.1. Define a local variable of type **Person *** named **fred**. Initialize **fred** to point to a new instance of **Person** initialized via the new **init...** method added in the previous step.

    2.2. Write an **NSLog** statement that prints **fred**'s full name, obtained by calling the new **fullName** method.

    2.3. Write another **NSLog** statement that prints **fred**'s description, obtained by calling the new **description** method.

    2.4. Write another **NSLog** statement to which you directly pass **fred** as an argument, paired with a `%@` format specifier.

PART 3

1. Add a **display** method to the **Person** class.

    1.1. The **display** method should call **description**, and then use **printf** to print the resulting string.

2. **Bonus Step:** Add a factory method, **+personWithFirstName:lastName:age:** that takes the same arguments as the corresponding **init...** method and returns either **id** or **instancetype**.

3. Write a unit test method named **testPart03** and do as follows:

    2.1. Create an **NSArray** object initialized with three fully populated **Person** objects.

    2.2. Enumerate the array, sending a **display** message to each object in the array.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 4

1. Add support for favoriting people by adding the following features.

    1.1. Add an instance variable of type **NSUInteger** named _**favoritesRanking**. (NB: **NSUInteger** is a conditionally compiled typedef that's an **unsigned long** on 64-bit platforms, and an **unsigned int** on 32-bit platforms.)

    1.2. Add a pair of accessor methods for _**favoritesRanking**. Customize the setter method to ensure that the maximum value never exceeds **5**.

    1.3. Add a method named **favoritesRankingStars** that takes no arguments and returns an **NSString** object that renders the person's current favorites ranking as a string of asterisks. Return a dash (minus sign) character for a ranking of zero.

    1.4. Modify the **description** method to include the favorites ranking stars.

2. Write a unit test method named **testPart04** and do as follows:

    2.1. Create an **NSArray** object initialized with five fully populated **Person** objects.

    2.2. Set each person's favorites ranking. Make sure to test boundary conditions (e.g., a ranking value greater than 5).

    2.3. Enumerate the array, sending a **display** message to each object in the array.

    2.4. **Challenge Question:** Can you figure out a way to call **display** on each object in the array *without* enumerating the array?

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

Bonus Lab – Objective-C Basics

1. In **UnitTests.m**, write a unit test method named **testPart05**, and do as follows:
    1.1. Create an array of five or more fully populated (including favorites ranking) **Person** objects.
    1.2. Read about the following topics in Apple's developer reference material:
        - The **NSArray** method, **sortedArrayUsingDescriptors:**.
        - the **NSSortDescriptor** class.
        - Hint: Consider at least glancing through the associated developer guides.
    1.3. Figure out how to sort the array based on favorites ranking.
2. **Double Bonus**: Figure out how to do an *n*ary sort by ranking, then last name, then first name.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Lab Solutions – Objective-C Basics

PART 1

Person.h

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

@interface Person : NSObject
{
    NSString *_firstName;
    NSString *_lastName;
    int _age;
}

- (NSString *)firstName;
- (void)setFirstName:(NSString *)newValue;

- (NSString *)lastName;
- (void)setLastName:(NSString *)newValue;

- (int)age;
- (void)setAge:(int)newValue;

@end
```

Person.m

```
#import "Person.h"

@implementation Person

- (NSString *)firstName {
    return _firstName;
}
- (void)setFirstName:(NSString *)newValue {
    _firstName = newValue;
}

- (NSString *)lastName {
    return _lastName;
}
- (void)setLastName:(NSString *)newValue {
    _lastName = newValue;
}

- (int)age {
    return _age;
}
- (void)setAge:(int)newValue {
    _age = newValue;
}

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

UnitTests.m

```
#import <XCTest/XCTest.h>
#import "Person.h"

@interface UnitTests : XCTestCase

@end


@implementation UnitTests

- (void)testPart01
{
    Person *fred = [[Person alloc] init];
    [fred setFirstName:@"Fred"];
    [fred setLastName:@"Smith"];
    [fred setAge:32];

    NSLog(@"\nfirst name: %@"
          @"\nlast name: %@"
          @"\nage: %d",
          [fred firstName],
          [fred lastName],
          [fred age]);
}

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 2

Person.h

```
// New declarations...

- (id)initWithFirstName:(NSString *)firstName
               lastName:(NSString *)lastName
                    age:(int)age;

- (NSString *)fullName;
```

Person.m

```
// New method definitions...

- (id)initWithFirstName:(NSString *)firstName
               lastName:(NSString *)lastName
                    age:(int)age
{
    self = [super init];
    if (!self) return nil;

    _firstName = [firstName copy];
    _lastName = [lastName copy];
    _age = age;

    return self;
}

- (NSString *)fullName
{
    return [NSString stringWithFormat:@"%@ %@", [self firstName], [self lastName]];
}

- (NSString *)description
{
    return [NSString stringWithFormat:@"name: %@, age: %d",
            [self fullName], [self age]];
}
```

UnitTests.m

```
// New test method...

- (void)testPart02
{
    Person *fred = [[Person alloc] initWithFirstName:@"Fred"
                                            lastName:@"Smith"
                                                 age:32];

    NSLog(@"%@", [fred fullName]);
    NSLog(@"%@", [fred description]);
    NSLog(@"%@", fred);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 3

Person.h

```
// New declarations...

+ (instancetype)personWithFirstName:(NSString *)firstName
                           lastName:(NSString *)lastName
                                age:(int)age;
- (void)display;
```

Person.m

```
// New method definitions...

+ (instancetype)personWithFirstName:(NSString *)firstName
                           lastName:(NSString *)lastName
                                age:(int)age
{
    return [[self alloc] initWithFirstName:firstName
                                  lastName:lastName
                                       age:age];
}

- (void)display
{
    printf("%s\n", [[self description] UTF8String]);
}
```

UnitTests.m

```
// New test method...

- (void)testPart03
{
    NSArray *people = @[[Person personWithFirstName:@"Fred" lastName:@"Smith" age:32],
                        [Person personWithFirstName:@"Jill" lastName:@"Brown" age:27],
                        [Person personWithFirstName:@"Lee" lastName:@"Jones" age:41]];

    for (Person *currPerson in people)
    {
        [currPerson display];
    }
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 4

Person.h

```
// New ivar declaration...

@interface Person : NSObject
{
    // ...

    NSUInteger _favoritesRanking;
}

// New method declarations...

- (NSUInteger)favoritesRanking;
- (void)setFavoritesRanking:(NSUInteger)newValue;

- (NSString *)favoritesRankingStars;
```

Person.m

```
// New constant definition...

const NSUInteger MaxRanking = 5;

// New method definitions...

- (NSUInteger)favoritesRanking
{
    return _favoritesRanking;
}

- (void)setFavoritesRanking:(NSUInteger)newValue
{
    _favoritesRanking = newValue > MaxRanking ? MaxRanking : newValue;
}

- (NSString *)favoritesRankingStars
{

    if ([self favoritesRanking] == 0)  return @"-";

    return [@"*****" substringToIndex:[self favoritesRanking]];
}

- (NSString *)description
{
    NSString *stars = [self favoritesRankingStars];
    stars = [stars stringByPaddingToLength:MaxRanking
                                withString:@" "
                           startingAtIndex:0];

    return [NSString stringWithFormat:@"%@  %@", stars, [self fullName]];
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

UnitTests.m

```
// New test method...

- (void)testPart04
{
    NSArray *people = @[[Person personWithFirstName:@"Fred" lastName:@"Smith" age:32],
                        [Person personWithFirstName:@"Jill" lastName:@"Brown" age:27],
                        [Person personWithFirstName:@"Lee" lastName:@"Jones" age:41],
                        [Person personWithFirstName:@"Greg" lastName:@"Moore" age:25],
                        [Person personWithFirstName:@"Sue" lastName:@"Davis" age:36]];

    [people[0] setFavoritesRanking:3];
    [people[1] setFavoritesRanking:999];
    [people[2] setFavoritesRanking:0];
    [people[3] setFavoritesRanking:3];
    [people[4] setFavoritesRanking:4];

    printf("\nPeople:\n-------\n");
    for (Person *currPerson in people) {
        [currPerson display];
    }

    NSSortDescriptor *sortDesc = [NSSortDescriptor
                                  sortDescriptorWithKey:@"favoritesRanking"
                                  ascending:NO];
    NSArray *sortedPeeps = [people sortedArrayUsingDescriptors:@[sortDesc]];

    printf("\nSorted People:\n--------------\n");
    for (Person *currPerson in sortedPeeps) {
        [currPerson display];
    }

    [sortedPeeps makeObjectsPerformSelector:@selector(display)];
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Bonus Lab

UnitTests.m

```
// New test method...

- (void)testPart05
{
    NSArray *people = @[[Person personWithFirstName:@"Fred" lastName:@"Smith" age:32],
                        [Person personWithFirstName:@"Jill" lastName:@"Brown" age:27],
                        [Person personWithFirstName:@"Al" lastName:@"Smith" age:41],
                        [Person personWithFirstName:@"Greg" lastName:@"Brown" age:25],
                        [Person personWithFirstName:@"Sue" lastName:@"Davis" age:36]];

    [people[0] setFavoritesRanking:3];
    [people[1] setFavoritesRanking:2];
    [people[2] setFavoritesRanking:3];
    [people[3] setFavoritesRanking:2];
    [people[4] setFavoritesRanking:3];

    printf("\nPeople:\n-------\n");
    for (Person *currPerson in people) {
        [currPerson display];
    }

    NSSortDescriptor *d1 = [NSSortDescriptor
                            sortDescriptorWithKey:@"favoritesRanking"
                            ascending:NO];
    NSSortDescriptor *d2 = [NSSortDescriptor
                            sortDescriptorWithKey:@"lastName"
                            ascending:YES];
    NSSortDescriptor *d3 = [NSSortDescriptor
                            sortDescriptorWithKey:@"firstName"
                            ascending:YES];
    NSArray *sortedPeeps = [people sortedArrayUsingDescriptors:@[d1, d2, d3]];

    printf("\nSorted People:\n--------------\n");
    [sortedPeeps makeObjectsPerformSelector:@selector(display)];
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Categories and Protocols

This section introduces a number of additional features of the Objective-C language, including categories and protocols. It also introduces several important protocols and categories in the Foundation Framework, including **NSCopying**, **NSCoding**, and **NSKeyValueCoding**.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Protocols

Objective-C protocols are similar in nature to Java interfaces.

- Protocols declare (but don't implement) a group of methods that specify an **API contract**.
- By default, classes are required to provide implementations of methods declared in protocols they adopt.
  - However, protocols can have optional sections.
  - Compiler doesn't enforce implementation of the optional protocol methods.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Declaring a Protocol

A protocol can be declared in a separate **.h** file, or incorporated in the **.h** file of a related class.

- Note that a protocol can adopt other protocols.

Declaring a Protocol

```
// Protocol methods are required by default.
@protocol Vocalizing

- (void)growl;
- (void)snarl;

@end
```

Declaring Optional Protocol Methods

```
// All vocalizing objects growl and snarl; some may bark and/or whine.
@protocol Vocalizing

- (void)growl;
- (void)snarl;

@optional
- (void)bark;
- (void)whine;

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Adopting a Protocol

- A class can adopt one or more protocols by listing them in angle brackets in the class declaration.

```
// Declaration can specify a list of protocols.

@interface Widget : NSObject <NSCoding, NSCopying>
```

- For example, the NSCoding protocol declares methods for serializing and deserializing instances.
  - Any class that adopts NSCoding must implement the two methods it declares.

```
// Given the following declaration...
//
@interface Person : NSObject <NSCoding>
// ...

// ...the compiler will warn if the Person class
// doesn't implement both of the following methods.
//
@implementation Person

- (id)initWithCoder:(NSCoder *)aDecoder
{
    // ...
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
    // ...
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

The NSObject Protocol

Declares the minimum set of behaviors needed for anything of type `id` to behave correctly with Foundation objects.

- Adopted by **NSObject** and **NSProxy** classes.
- Methods were all originally declared in **NSObject** class; later moved to protocol so declarations could be shared.
- Includes essential introspection methods.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Dynamic Checking

The **NSObject** protocol includes several methods that are frequently used to perform dynamic checking of such things as an object's:

- Membership in a given class hierarchy.
- Ability to respond to a particular message.
- Protocol conformance.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Dynamic Checking Examples

- Checking class hierarchy membership:

```
NSMutableArray *people = [NSMutableArray array];
NSArray *objs = @[obj1, obj2, obj3];

for (id currObj in objs)
{
    if ([currObj isKindOfClass:[Person class]]) {
        [people addObject:currObj];
    }
}
```

- Checking ability to respond to a given selector:

```
for (id currObj in objs)
{
    if ([currObj respondsToSelector:@selector(fullName)]) {
        NSLog(@"%@", [currObj fullName]);
    }
}
```

- Checking protocol conformance:

```
for (id currObj in objs)
{
    if ([currObj conformsToProtocol:@protocol(Vocalizing)]) {
        [currObj snarl];
        [currObj growl];

        if ([currObj respondsToSelector:@selector(bark)]) {
            [currObj bark];
        }
    }
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Protocols as Type Qualifiers

A *protocol list* can be used to qualify an object type declaration.

- For variable assignments and argument passing, compiler checks **protocol conformance** as well as type.

```
id<Vocalizing> obj1 = [[Dog alloc] init]; // Okay
id<Vocalizing> obj2 = @"Hello";           // Warning

Person<Vocalizing> *p1 = [[Employee alloc] init]; // Okay
Person<Vocalizing> *p2 = [[Person alloc] init];   // Warning
```

- When type-checking receiver of a message:
  - **If receiver is dynamically typed:** compiler assumes receiver has *only* the methods declared in the protocols(s).
  - **If receiver is statically typed:** compiler assumes receiver has additional methods declared in the protocol(s).

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Protocol Type Qualifier Examples

Example: Dynamic Type Qualified with Protocol

```
SEL mySelector = @selector(growl);

id obj1 = [[Dog alloc] init];
if ([obj1 respondsToSelector:mySelector])  // Okay.
{
    // Do stuff...
}

// Here, qualifier narrows declared type.
//
id<Vocalizing> obj2 = [[Dog alloc] init];  // Too narrow.
if ([obj2 respondsToSelector:mySelector])  // Compile error.
{
    // Do stuff...
}

id<Vocalizing, NSObject> obj3 = [[Dog alloc] init];
if ([obj3 respondsToSelector:mySelector])  // Okay.
{
    // Do stuff...
}
```

Example: Static Type Qualified with Protocol

```
// Given Employee class declared as follows...

@interface Employee : Person <Vocalizing>

// ...

Person *emp1 = [[Employee alloc] init];  // Too narrow.
[emp1 growl];                            // Compile error.

// Here, qualifier widens declared type.
//
Person<Vocalizing> *emp2 = [[Employee alloc] init];
[emp1 growl];                            // Okay
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Copying

- NSObject declares the following instance methods:

```
- (id)copy;          // Returns immutable copy
- (id)mutableCopy;   // Returns mutable copy
```

- Call methods from protocols **not adopted by** NSObject

```
@protocol NSCopying

- (id)copyWithZone:(NSZone *)zone;

@end

@protocol NSMutableCopying

- (id)mutableCopyWithZone:(NSZone *)zone;

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Copying Pitfalls

- NSObject implements `copy` and `mutableCopy` as cover methods for `copyWithZone:` and `mutableCopyWithZone:`.
  - However, doesn't implement either protocol.
  - Therefore, sending `copy` or `mutableCopy` triggers unrecognized selector exception.
- While most Foundation classes adopt **NSCopying**, only class clusters adopt **NSMutableCopying**.

WARNING: It's unsafe to send a **copy** message to an object of unknown type without first checking for protocol conformance.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Categories

- A category allows additional methods to be declared and/or implemented on an existing class.
  - Allows splitting lengthy class implementations across multiple files.
  - Allows grouping of method declarations and implementations based on their cohesiveness, regardless of which classes they belong to.
- Also allows adding methods to framework classes.
- Potential pitfall:
  - A category's method implementations are added to the target class **at run time**, when category is loaded.
  - If method name matches existing method in the target class, **original implementation will be replaced**.
- To avoid pitfall, prefix method names with project prefix.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Category Examples

Adding a category to NSArray

```
// Declaring category.
// Typically in NSArray+XYZAdditions.h, but could be anywhere.

@interface NSArray (XYZAdditions)

- (NSArray *)xyz_reversedArray;    // Note use of XYZ_ prefix.

@end


// Implementing category.
// Typically in NSArray+XYZAdditions.m, but could be in any .m file.

@implementation NSArray (XYZAdditions)

- (NSArray *)xyz_reversedArray
{
    NSMutableArray *newArray = [NSMutableArray arrayWithCapacity:
                                [self count]];

    for (id currObj in [self reverseObjectEnumerator]) {
        [newArray addObject:currObj];
    }

    return newArray;
}

@end


// Using category.

#import "NSArray+XYZAdditions.h"

- (void)doStuffBackwardsWithObjects:(NSArray *)someObjs
{
    NSArray *reversedObjs = [someObjs xyz_reversedArray];

    // Do stuff with reversedObjs...
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

NSDictionary

A keyed collection of objects *of any type* (type `id`)

- Each entry in a dictionary is a *key-value pair*.
- Dictionaries cannot contain:
  - non-object types, e.g., **int**, **float**, **struct**
  - **nil** (Use singleton instance of **NSNull** to represent **nil** values.)
- Instances are immutable.
  - Mutable subclass: **NSMutableDictionary**

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

NSDictionary Examples

- Inserting objects in a mutable dictionary.

```
NSMutableDictionary *info = [NSMutableDictionary dictionary];

[info setObject:@"Fred" forKey:@"firstName"];

// Using subscript syntax

info[@"firstName"] = @"Fred";
```

- Retrieving objects from a dictionary.

```
NSString *name = [info objectForKey:@"firstName"];

// Using subscript syntax

NSString *name = info[@"firstName"];
```

- NSDictionary literal expression.

```
NSDictionary *info = @{ @"name" : @"Fred",
                        @"age"  : @32 };
```

- Enumerating a dictionary's keys.

```
for (NSString *key in info)
{
    NSLog(@"key: %@, value: %@", key, info[key]);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Property List Files

Dictionaries, arrays, strings, and several other types of Foundation objects can serialize and deserialize themselves to/from files using one of several **property list** formats.

- Allows nesting of dictionaries and arrays.
    - Supports modeling of data hierarchies.
- Default property list format is XML.
- Alternative formats:
    - Binary (faster, but opaque).
    - Text (more human-readable, but supports fewer types, e.g., no support for NSNumber, NSDate, NSData.)

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Property List Examples

Writing and reading a dictionary to/from a .plist file:

```
// Given the following...

NSDictionary *info = @{ @"name"   : @"Fred W. Smith",
                        @"age"    : @37,
                        @"phones" : @{@"Work" : @"703-123-4567",
                                      @"Home" : @"301-987-6543"},
                        @"kids"   : @[@"Bob", @"Alice", @"Jill"] };


// Write to plist file.

[info writeToFile:@"/tmp/info.plist" atomically:YES];


// Initialize new instance with contents of plist.

NSDictionary *newInfo = [NSDictionary dictionaryWithContentsOfFile:
                         @"/tmp/info.plist"];
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Key-Value Coding (KVC)

General mechanism that adds dictionary-like semantics to **NSObject**.

- **NSKeyValueCoding** category (informal protocol) API built on the following two 'primitive' methods:

```
- (id)valueForKey:(NSString *)key;
```

```
- (void)setValue:(id)value forKey:(NSString *)key;
```

- Allows all objects to be treated with dictionary semantics.
- Used heavily in sophisticated mechanisms throughout all of Apple's frameworks and tools.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

Lab – Categories and Protocols

OVERVIEW

In this lab you'll declare a custom protocol to implement the **Delegate** pattern. You'll also use a category to add a method the **NSArray** class.

PART 1

1. In the **Objective-C Labs** project you created in the previous lab exercise, add a subclass of **NSObject** named **Dog** with the following features:

    1.1. An instance variable of type **NSString** * named **_name**.

    1.2. A custom initializer with the following signature:

    ```
    - initWithName:(NSString *)name
    ```

    1.3. A **name** accessor method.

    1.4. Three methods that take no arguments and return **void**, each of which uses **printf** to print a message on the console. The message should consist of the dog's name followed by a colon and a space, and then the string described below, followed by a newline character.

    - A **growl** method that prints **Grrrrrr!**.
    - A **bark** method that prints **Woof! Woof! Woof!**.
    - A **wagTail** method that prints **[Wags tail.]**.

    1.5. An overridden **description** method that returns the dog's name.

    1.6. A method named **doorbellDidRing** that calls the **growl**, **bark**, and **wagTail** methods.

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

2. Add a subclass of **Person** named **DogOwner** with the following features:

2.1. An instance variables of type **NSMutableArray *** named **_dogs**.

2.2. A private method named **mutableDogs** that lazily initializes and returns the **_dogs** instance variable.

2.3. A public method named **dogs** that returns the value returned by **mutableDogs**.

2.4. A public method named **addDogs:** that takes an argument of type **NSArray ***, and adds its content to the mutable dogs array.

2.5. An overridden **description** method that prints the owner's full name, followed by the a description of each of the owner's dogs.

3. Add a test case named **DogOwnerTests.m** that includes the following:

3.1. An instance variable of type **DogOwner *** named **_owner**.

3.2. A **setUp** method that initializes **_owner** with an instance of **DogOwner**, and then adds three instances of **Dog** named **Bowser**, **Woofsie**, and **Spot** to the owner's dogs array.

3.3. A **testPart01** method that uses **NSLog** to print the owner's description, and then sends **doorbellDidRing** to each of the owner's dogs.

3.4. Build and run to make sure that the descriptions print as expected, and that each dog prints a growl, bark, and wag tail message.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

PART 2

1. Add the following features to the **Dog** class:

    1.1. A protocol named **DogDelegate** that declares three required methods, all of which take an instance of **Dog** as their only argument. The first method, **dogDidHearDoorbell:** should return **void**; the other two, **dogShouldBark:** and **dogShouldWagTail:**, should return **BOOL**.

    1.2. An **_delegate** instance variable of type **id<DogDelegate>**, and a corresponding pair of accessor methods.

    1.3. A **sit** method that prints a message similar to **wagTail**, only with **[Sits.]** as its text instead of **[Wags tail.]**.

    1.4. Modify **doorbellDidRing** as follows:

    - After calling **growl**, send a **dogDidHearDoorbell:** message to the dog's delegate.
    - Add logic to call **bark** only if the dog's delegate is **nil**, *or* if the delegate's **dogShouldBark:** method returns **YES**. Add similar logic before the call to **wagTail**.

2. Add the following features to the **DogOwner** class:

    2.1. Make **DogOwner** conform to the **DogDelegate** protocol.

    2.2. Implement **dogShouldBark:** and **dogShouldWagTail:** to return **NO**.

    2.3. Implement **dogDidHearDoorbell:** to send a **sit** message to **Bowser** and **Woofsie**.

3. In **DogOwnerTests**, write a test method named **testPart02** that does as follows:

    3.1. Make the owner object **Bowser** and **Woofsie**'s delegate.

    3.2. Send **doorbellDidRing** to each dog.

    3.3. Build and run. Make sure that **testPart01** still works as it did previously, and that the output of **testPart02** verfies that **Bowser** and **Woofsie** growl and sit (but don't bark), while **Spot**'s behavior is unchanged.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

PART 3

1. Add a **LABAdditions** category to the **NSArray** class. The category should declare and implement a method named **LAB_fancyDescription** with the following behavior:

    1.1. Print the name of the receiving class, followed by a count of its elements.

    1.2. For each element, print the object's class, followed by its description.

2. Write a unit test method named **testPart03** that sends a **LAB_fancyDescription** to the owner's array of dogs.

3. Build and run, and verify that the output is as expected.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.

Lab Solutions – Categories and…

PART 1

Dog.h

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

@interface Dog : NSObject
{
    NSString *_name;
}

- (id)initWithName:(NSString *)name;
- (NSString *)name;

- (void)doorbellDidRing;

@end
```

Dog.m

```
#import "Dog.h"

@implementation Dog

- (id)initWithName:(NSString *)name
{
    if (!(self = [super init])) return nil;

    _name = [name copy];

    return self;
}

- (NSString *)name {
    return _name;
}

- (void)growl {
    printf("%s: Grrrrrr!\n", [[self name] UTF8String]);
}

- (void)bark {
    printf("%s: Woof! Woof! Woof!\n", [[self name] UTF8String]);
}

- (void)wagTail {
    printf("%s: [Wags tail.]\n", [[self name] UTF8String]);
}

- (NSString *)description {
    return [self name];
}

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

DogOwner.h

```
#import <Foundation/Foundation.h>
#import "Person.h"

@interface DogOwner : Person
{
    NSMutableArray *_dogs;
}

- (NSArray *)dogs;

- (void)addDogs:(NSArray *)newDogs;

@end
```

DogOwner.m

```
#import "DogOwner.h"
#import "Dog.h"

@implementation DogOwner

- (NSArray *)dogs
{
    return [self mutableDogs];
}

- (NSMutableArray *)mutableDogs
{
    if (_dogs == nil) {
        _dogs = [NSMutableArray array];
    }

    return _dogs;
}

- (void)addDogs:(NSArray *)newDogs
{
    [[self mutableDogs] addObjectsFromArray:newDogs];
}

- (NSString *)description
{
    return [NSString stringWithFormat:@"\nOwner: %@\nDogs: %@",
            [self fullName], [self dogs]];
}

@end
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

UnitTests.m

```
#import <XCTest/XCTest.h>
#import "DogOwner.h"
#import "Dog.h"

@interface DogOwnerTests : XCTest

@end


@implementation DogOwnerTests
{
    DogOwner *_owner;
}

- (void)setUp
{
    [super setUp];

    _owner = [[DogOwner alloc] initWithFirstName:@"Fred"
                                        lastName:@"Smith"
                                             age:32];

    Dog *bowser = [[Dog alloc] initWithName:@"Bowser"];
    Dog *woofsie = [[Dog alloc] initWithName:@"Woofsie"];
    Dog *spot = [[Dog alloc] initWithName:@"Spot"];

    [_owner addDogs:@[bowser, woofsie, spot]];
}

- (void)testPart01
{
    NSLog(@"%@", _owner);

    [[_owner dogs] makeObjectsPerformSelector:@selector(doorbellDidRing)];
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 2

Dog.h

```
// Protocol declaration...

@class Dog;

@protocol DogDelegate <NSObject>

- (void)dogDidHearDoorbell:(Dog *)dog;
- (BOOL)dogShouldBark:(Dog *)dog;
- (BOOL)dogShouldWagTail:(Dog *)dog;

@end

// New ivar...

    id<DogDelegate> _delegate;

// New method declarations...

- (id<DogDelegate>)delegate;
- (void)setDelegate:(id<DogDelegate>)delegate;

- (void)sit;
```

Dog.m

```
// New method definitions...

- (id<DogDelegate>)delegate {
    return _delegate;
}

- (void)setDelegate:(id<DogDelegate>)delegate {
    _delegate = delegate;
}

- (void)doorbellDidRing
{
    [self growl];

    [_delegate dogDidHearDoorbell:self];

    if (_delegate == nil || [_delegate dogShouldBark:self]) {
        [self bark];
    }

    if (_delegate == nil || [_delegate dogShouldWagTail:self]) {
        [self wagTail];
    }
}

- (void)sit {
    printf("%s: [Sits.]\n", [[self name] UTF8String]);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

DogOwner.h

```
// Protocol adoption...

#import "Dog.h"

@interface DogOwner : Person <DogDelegate>
```

DogOwner.m

```
// New method definitions...

- (void)dogDidHearDoorbell:(Dog *)dog
{
    if ([[dog name] isEqualToString:@"Bowser"] ||
        [[dog name] isEqualToString:@"Woofsie"])
    {
        [dog sit];
    }
}

- (BOOL)dogShouldBark:(Dog *)dog
{
    return NO;
}

- (BOOL)dogShouldWagTail:(Dog *)dog
{
    return NO;
}
```

UnitTests.m

```
// New test method...

- (void)testPart02
{
    [[_owner dogs][0] setDelegate:_owner];
    [[_owner dogs][1] setDelegate:_owner];
    [[_owner dogs] makeObjectsPerformSelector:@selector(doorbellDidRing)];
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

PART 3

NSArray+LABAdditions.h

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

@interface NSArray (LABAdditions)

- (NSString *)LAB_fancyDescription;

@end
```

NSArray+LABAdditions.m

```
#import "NSArray+LABAdditions.h"

@implementation NSArray (LABAdditions)

- (NSString *)LAB_fancyDescription
{
    NSMutableString *s = [NSMutableString stringWithFormat:
                          @"%@ of %ld elements (",
                          [self class], [self count]];

    for (id currObj in self) {
        [s appendFormat:@"\n    %@, %@", [currObj class], currObj];
    }

    [s appendString:@"\n)\n"];

    return s;
}

@end
```

UnitTests.m

```
// New test method...

- (void)testPart03
{
    NSLog(@"%@", [[_owner dogs] LAB_fancyDescription]);
}
```

© Copyright 2014-2018, About Objects, Inc.

All rights reserved worldwide.

© Copyright 2014-2018, About Objects, Inc. All rights reserved worldwide.