Chapter 2
Objective-C

- Objects
- Using Instances
- NSString, NSArray, NSMutable ...
- Subclassing
- Exceptions
- Fast Enumeration

Objective-C

- iOS applications are written in Objective-C using Cocoa Touch library
- Objective-C is an extension of the C language initiated by the NeXTSTEP company and used as the main language of their OS.
- Objective-C uses Smalltalk style messaging,

  myCheckBook calculateBalance;

  means that we are sending the message calculateBalance to the instance myCheckBook
- Cocoa Touch is a user interface framework work developed by Apple using Objective-C to contain needed API to process user interactions when using iPod Touch, the iPhone, and the iPad

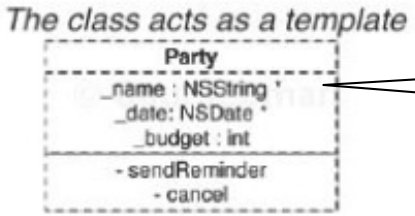
Objects

- Objects in Object-Oriented languages like Objective-C are an extension of structs in C.
- struct in C is designed to house the data pertaining to the attributes of the object being represented by the struct.
- Classes (and Objects instantiated from them) are designed to include data attributes representing the characteristics of the entity being modeled and methods representing the capabilities and actions that can be performed by the modeled identity.
- The class party to be used in creating myParty should:

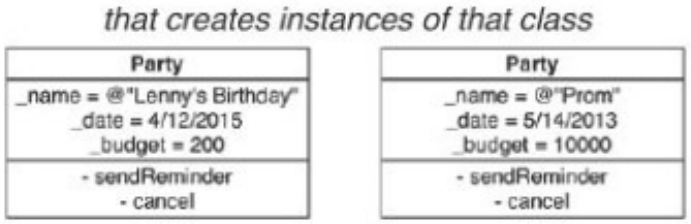
  house attributes such as the date of my party, the time of my party and the location of my party.

  be able to access my address book and send out emails inviting or reminding all my family members to come to my party.

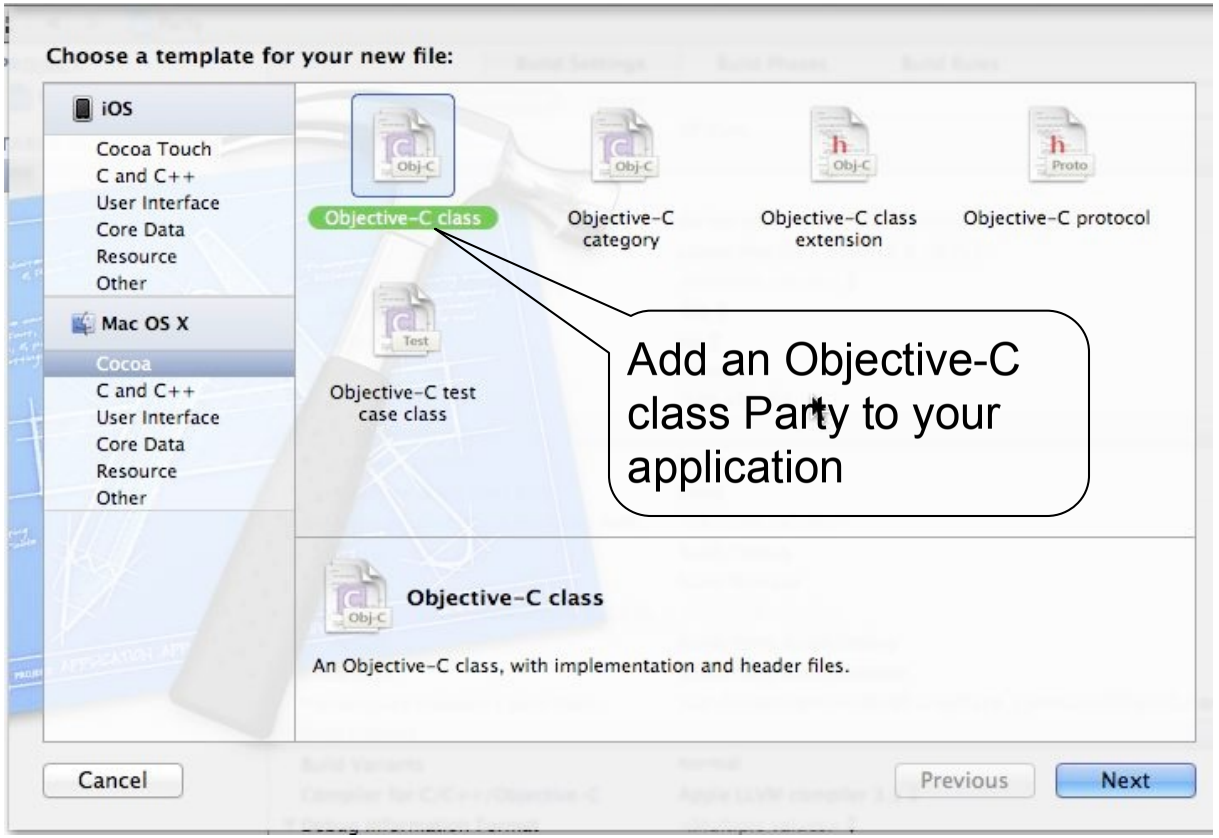
The Party Class



The Party class
Attributes: name, date, and budget
Methods: sendReminder, cancel



Lenny's Birthday Party is an instance of Party



Add an Objective-C class Party to your application

Using Instances

- To use an instance of a class, you must have a variable that points to that object.
- A pointer variable stores the location of an object in memory, not the object itself. (It “ points to” the object.)
- A variable that points to a Party object is declared like this:

```
Party * partyInstance;
```

- Creating this pointer does not create a Party object – only a variable that can point to a Party object.
- Notice that this variable’s name does not start with an underscore; it is not an instance variable. It is meant to be a pointer to an instance of Party.

Creating Objects

- An object has a life span: it is created, sent messages, and then destroyed when it is no longer needed.
- To create an object, you send an alloc message to a class. In response, the class creates an object in memory (on the heap, just like malloc() would) and gives you the address of the object, which you store in a variable:

```
Party * partyInstance = [ Party alloc];
```

- The first message you always send to a newly allocated instance is an initialization message. Although sending an alloc message to a class creates an instance, the instance is not ready for work until it has been initialized.

```
Party * partyInstance = [ Party alloc]; [ partyInstance init];
```

- Because an object must be allocated and initialized before it can be used, you always combine these two messages in one line.

```
Party * partyInstance = [[ Party alloc] init];
```

- Combining two messages in a single line of code is called a nested message send.

Recap

- Create an object (instance) of your class:

```
ObjectClassName *myInstanceName = [[ObjectClassName alloc] init]
```

- Instances are always pointers to a class type:

```
NSMutableArray *myArray = [[NSMutableArray alloc] init];
or
NSMutableArray *myArray = [NSMutableArray alloc];
[myArray init];
```

- alloc is the allocation method implemented by NSObject and allocates necessary memory for our object.
- alloc should never be user by itself ans should also be followed by an initialization method such as init.
- init may be overridden

Sending Messages

- An Objective-C message has three parts:

  The receiver which is the instance whose method is to be executed.

  The selector which is the method selected for execution. Any part of the selector name (method name) preceding an argument must end with :. The arguments are disbursed through out the name.

  The arguments which are to be disbursed through out the name of the selector.

- Objective-C's message style is:

  [receiver selector arguments];

- Some messages

```
[myParty sendReminder];
[myParty setBudget: 200];
Party lennyParty = [[Party alloc] initWith: @"Lenny's BDay Party" date: @"2012-12-23" budget: 200];
```

- Line 3 creates an instance of Party called lennyParty and it initializes the name to Lenny's BD Party, with a budget of 200 and to take place on December 23, 2012 (assuming the I implemented such method)
- It is OK to send messages to nil (equivalent to NULL in other languages) objects.
- If a message that returns a value is sent to a nil object it will return zero.

Destroying Objects

- To destroy an object, you set the variable that points to it to nil.
- partyInstance = nil;
- This line of code destroys the object pointed to by the partyInstance variable and sets the value of the partyInstance variable to nil.
- The value nil is the zero pointer. (C programmers know it as NULL. Java programmers know it as null.)
- A pointer that has a value of nil is typically used to represent the absence of an object. For example, a party could have a venue. While the organizer of the party is still determining where to host the party, venue would point to nil.
- If you send a message to a variable that is nil, nothing happens.
- In earlier versions of iOS we had to track all uses (references) of our objects by incrementing or decrementing references to these objects. We also had to release memory allocated to our objects using the release message.
- For iOS 5 Mac OS we longer need to do that, to destroy an object we set it to nil (equivalent to NULL in other languages). More on this in next chapter

The Interface and Implementation of Party

The header file defining the interface of the Party class.

The implementation file where all methods have to be implemented including setters and getters

Party > Party.h > @interface Party

```
1  //
2  //  Party.h
3  //  Party
4  //
5  //  Created by Imam Ibrahim on 5/9/12.
6  //  Copyright (c) 2012 __MyCompanyName__. All
       rights reserved.
7  //
8
9  #import <Foundation/Foundation.h>
10
11 @interface Party : NSObject
12
13 @property (strong, nonatomic) NSString *name;
14 @property (strong, nonatomic) NSDate *date;
15 @property int budget;
16
17 -(void) sendReminder;
18 -(void) cancel;
19
20 @end
21
```

Everything defined here is considered public

Co... > Party.m > @implementation Party

```
1  //
2  //  Party.m
3  //  Party
4  //
5  //  Created by Imam Ibrahim on 5/9/12.
6  //  Copyright (c) 2012 __MyCompanyName__. All
       rights reserved.
7  //
8
9  #import "Party.h"
10
11 @implementation Party
12
13 @synthesize name = _name;
14 @synthesize date = _date;
15 @synthesize budget = _budget;
16
17 -(void) sendReminder
18 {
19     NSLog(@"I am sending you a reminder");
20 }
21
22 -(void) cancel
23 {
24     NSLog(@"Sorry to inform you that my party got
           canceled");
25 }
26
27 @end
28
```

Implement getters and setters for the properties and associate variables with them

Using Instances: Instantiating myParty

The main.m of my Party application.

mport the Party nterface “Party.h” le

Create myParty as an object pointer to Party

Initialize the date of my party.

Have myParty send the reminder then send a cancel message

```
//
//  main.m
//  Party
//
//  Created by Imam Ibrahim on 5/9/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "Party.h"

int main(int argc, const char * argv[])
{
    @autoreleasepool {

        NSLog(@"Hello, World!");
        Party *myParty = [[Party alloc] init];

        myParty.date = [[NSDate alloc] initWithString: @"2012-12-23 16:45:32 -0400"];
        NSLog(@"The date of my party is %@", myParty.date.description);

        [myParty sendReminder];
        [myParty cancel];
    }
    return 0;
}
```

```
All Output
2012-05-09 14:43:21.172 Party[6573:403] Hello, World!
2012-05-09 14:43:21.180 Party[6573:403] The date of my party is 2012-12-23 20:45:32 +0000
2012-05-09 14:43:21.181 Party[6573:403] I am sending you a reminder
2012-05-09 14:43:21.182 Party[6573:403] Sorry to inform you that my party got canceled
```

The RandomPossessions Tool

- The “RandomPossessions” Tool is the application through which we will introduce some of Objective-C classes and concepts.
- I will deviate sightly from the textbook example in order to illustrate additional points.

NSString

- The NSString class declares the programmatic interface for an object that manages immutable strings.
- An immutable string is a text string that is defined when it is created and subsequently cannot be changed.
- NSString is implemented to represent an array of Unicode characters, in other words, a text string.
- The mutable subclass of NSString is NSMutableString.
- To hard-code a string you use @”desired string”. This is a convenience method t specify a constant string object.
- NSString examples:

```
//Create an NSString object and initialize it to my name.
NSString *myName = @”Ibrahim Imam”;
//obtain the length of  myName.
[myName length];
```

NSLog and Format Strings

- NSLog is documented as:

  Logs an error message to the Apple System Log facility.

  void NSLog (NSString *format, ...);

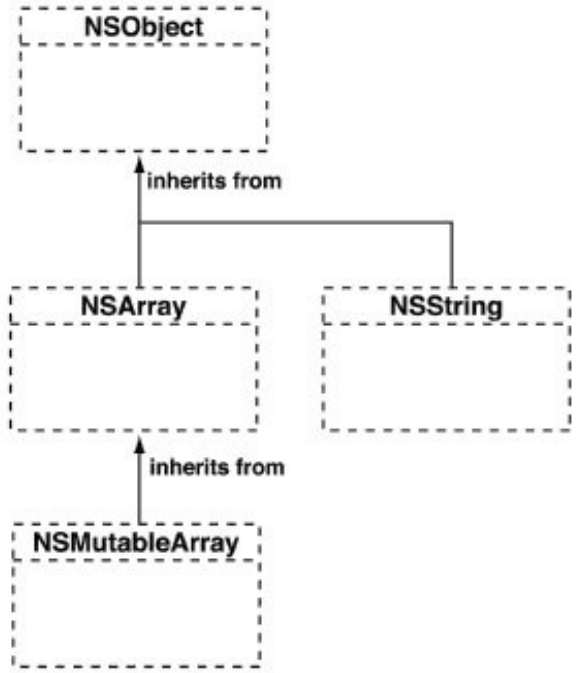
- The string format uses format specifier similar to the ones we use in printf such as %i, %d for integer values, %f of %g for float values.
- A special Objective-C specifier is %@. This format call the description message for the object and uses this string in the format string.
- NSLog example using %@ to call the description of myName

```
NSLog(@"a = %d, b = %5.2f, c = %c, my name = %@", a, b, c, myName);

Output line:

2012-05-09 16:38:05.547 RandomPossessions[7473:403] a = 2, b =  2.50, c = A, my
name = Ibrahim Imam
```

NSArray and NSMutableArray

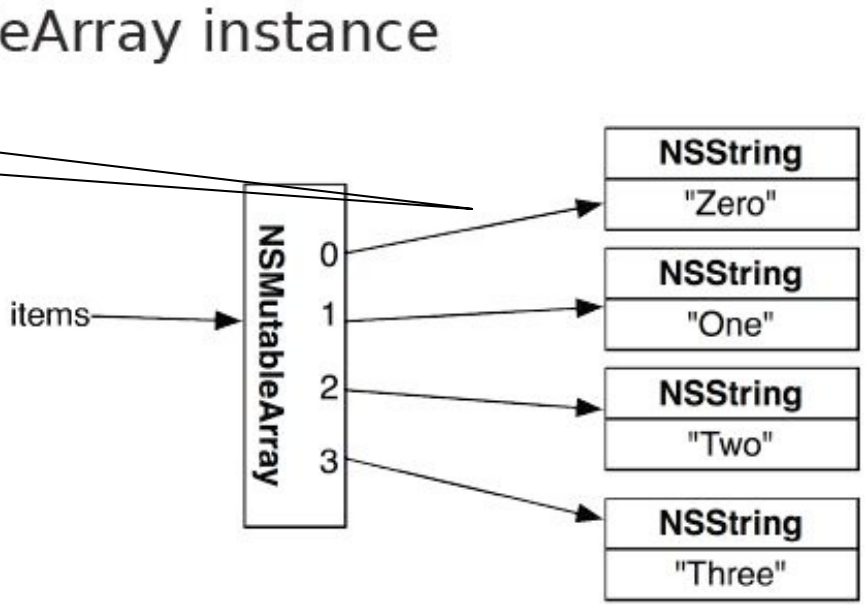
- Objective-C provides us with a container that will allow us to access objects by an index. This container is called NSArray.
- Once NSArray objects are initialized the array cannot be changed. Thus NSArray are static ordered collections.
- A subclass of NSArray is NSMutableArray which is dynamic and you can add or remove objects from NSMutableArray objects.
- Arrays in Objective-C do not hold the objects they are supposed to contain. They merely keep pointers to their objects.
- Objects in the array do not have to be the same instances of the same class, you may have say an NSString object at index 0 and an NSInteger at index 1.
- You cannot have primitive types in an an array, so if you are to store the value 7, you have to create and NSNumber object with value 7 and add it as an object:

```
[NSNumber numberWithInt: 7]
```



NSArray and NSMutableArray

Arrays contain pointers to their objects.



Objects in an array do not need to belong to same class

```
64
65  int arrayExample()
66  {
67      @autoreleasepool {
68          NSMutableArray *items = [[NSMutableArray alloc] init];
69
70          [items addObject: @"One"];
71          [items addObject: @"Two"];
72          [items addObject: @"Three"];
73          [items addObject: [NSNumber numberWithInt: 7]];
74
75          [items insertObject: @"Zero" atIndex: 0];
76          NSLog(@"%@", items);
77
78          for (int i =0; i < [items count]; i++)
79          {
80              NSLog(@"%@", [items objectAtIndex: i]);
81          }
82      }
83      return 0;
84  }
85
```

```
All Output
2012-05-10 14:29:42.886 RandomPossessions[536:403] (
    Zero,
    One,
    Two,
    Three,
    7
)
```

Creating a number object from 7 to store in an array

Instance Variables and Accessors

- Instance variables are to be specified in a code block {} in the interface section of the class definition.
- Instance variable defined in the interface section and not qualified by a scope have a protected scope by default, i.e. they can be accessed by any method defined by the class and all its subclasses.
- Scope directives that may be used to control access to instance variables are:

  @protected, this is the default as specified above

  @private, These variables can be accessed by the methods of the class but not it's subclasses. This is the default for any instance variable defined in the implementation section.

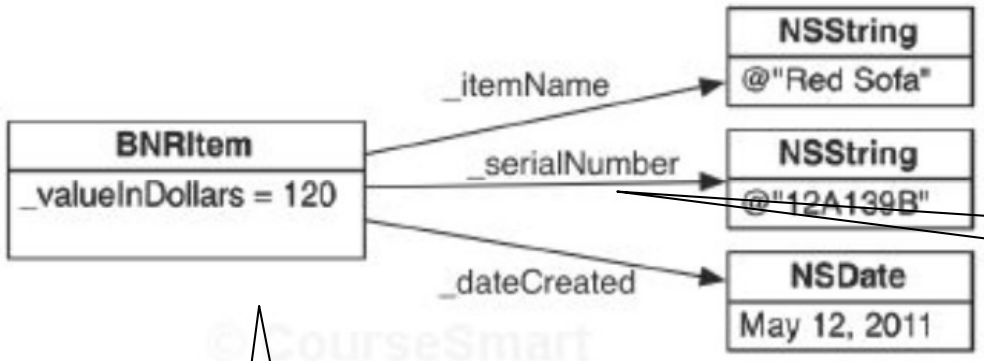
  @public, any method defined in any class or module can directly access these variables

  @package, For the 64-bit images. Any method in any of these images that implement this class can access these variables

Instance variables of BNRItem

Instance variables defined in the interface section. Their scope is “protected

```
8
9   #import <Foundation/Foundation.h>
10
11  @interface BNRItem : NSObject
12  {
13      // All Instance variables need to be declared here
14      // These variables are available to the class and all its
15      // subclasses
16      // This is the default and it is equivalent to "protected scope"
17      NSString *_itemName;
18      NSString *_serialNumber;
19      int _valueInDollars;
20      NSDate *_dateCreated;
21  }
22
```



The object stores pointers to instance variables that are instances of objects themselves

The object stores primitive types locally

Accessors

- Methods that provide a get and set access to instance variables, known as getters and setters, are collectively called accessors.
- In Objective-C, all getters have the same name as the instance variable (we do not use get) and all setters take the form setInstanceVariableName.

```
35  // Accessors for itemName, the setter and getter
36  -(void) setItemName: (NSString *)str;
37  -(NSString *)itemName;
38
39  -(void) setSerialNumber: (NSString *)str;
40  -(NSString *)serialNumber;
41
42  -(void) setValueInDollars: (int)i;
43  -(int)valueInDollars;
44
45  -(void) initializeDateCreated;
46  -(NSDate *)dateCreated;
```

dateCreated has a getter but does not have a setter. initializeDateCreated initializes the date when an object is created but is not used as a setter to set the date.

Instance methods

- Instance methods are the part of the interface that is designed to communicate with instances of objects. This is to say that these messages are to be sent to instances of the class and not the class itself
- Instance methods are prefixed with the character - when declared and implemented.
- Accessors are instance methods and so is the methods init and description and any other initializers we decide to implement.
- Instance methods declared in the interface section have a public scope, they can be sent to instances of the class from outside the class's implementation (as in main for instance)
- Instance methods may be defined in an "interface" extension in the implementation file. These methods are private and can only be called from within the class

Private Instance Methods

```objc
#import "BNRItem.h"

@interface BNRItem ()

// All methods declared here are private
// they can be called by other methods implemented in this section
// the cannot be sent as messages to instances outside the methods implementated here
// notice the () in @interface BNRItem ()

-(void) doNothingButPrint;

@end

@implementation BNRItem
```

Interface extension

Private instance method

Private instance method sent from within the class. It cannot be sent from outside the class's implementation

```objc
-(int) valueInDollars
{
    [self doNothingButPrint]; // Private methods can be called in class only
    return valueInDollars;
}
```

Initializers

- A class may implement several initializers used to initialize it's instance variables.
- Initializers are instance methods
- They all must begin with the phrase "init".
- One of these need to be documented (or at least thought of) as a designated initializer.
- It is customary to override init to provide a customized initialization of class's instance variables.
- It is typical for initializers to rely on other initializers such as init relying on the designated initializer.

Initializers of BNRItem

```objc
// The "Designated" initializer.
-(instancetype)initWithName: (NSString *) name valueInDollars: (int) value serialNumber: (NSString *) sNumber;
-(instancetype)initWithItemName:(NSString *)name;
// The Silver Challenge initializer, It is not the designated initializer and
// it does use the designated initializer.
-(instancetype) initWithName: (NSString *)name serialNumber: (NSString *)sNumber;
```

```objc
-(instancetype) initWithName:(NSString *)name valueInDollars:(int)value serialNumber:(NSString *)sNumber
{
    // Call The "Designated" initializer for the super class
    self = [super init];
    if (self)
    {
        // initialize the instant variables
        [self setItemName: name];
        [self setValueInDollars: value];
        [self setSerialNumber: sNumber];
        _dateCreated = [[NSDate alloc] init];
    }
    return self;
}

-(instancetype) initWithItemName:(NSString *)name
{
    return [self initWithName:name valueInDollars: 0 serialNumber:nil];
}

-(instancetype) init
{
    return [self initWithName:@"Item" valueInDollars: 0 serialNumber: @""];
}
```

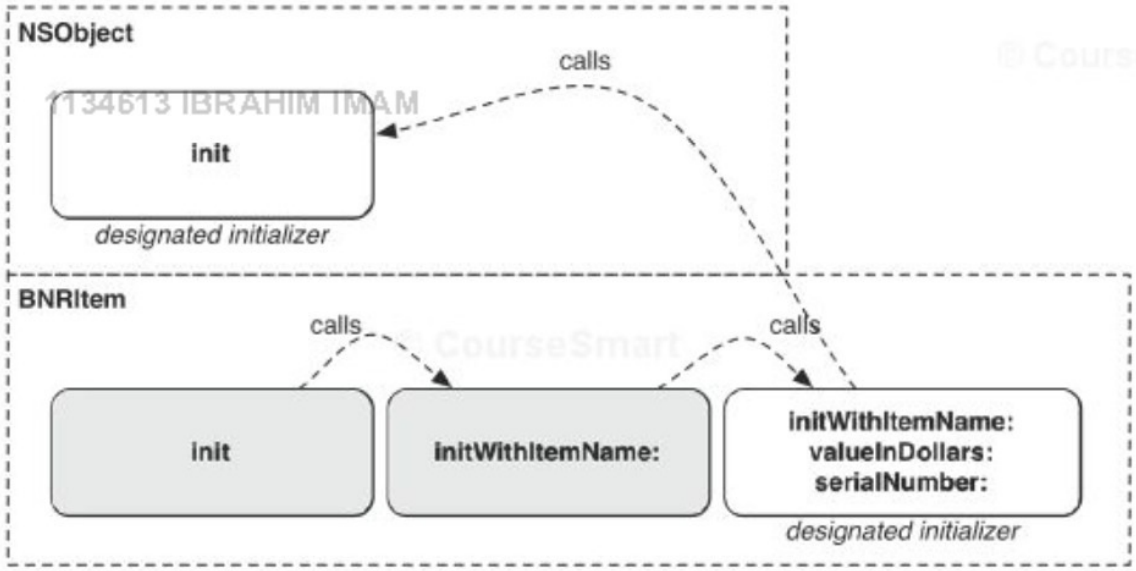
Designated Initialize

Another initializer

Overriding init

Using Initializers

- Using initializers in a chain reduces the possibility of error and makes maintaining code easier.
- The programmer who created the class makes it clear which initializer is the designated initializer.
- You only write the core of the initializer once in the designated initializer, and other initialization methods simply call the designated initializer (directly or indirectly) with default values.



- A class inherits all initializers from its superclass and can add as many as it wants for its own purposes.
- Each class picks one initializer as its designated initializer.
- The designated initializer calls the superclass's designated initializer (directly or indirectly) before doing anything else.
- Any other initializers call the class's designated initializer (directly or indirectly).
- If a class declares a designated initializer that is different from its superclass, the superclass's designated initializer must be overridden to call the new designated initializer (directly or indirectly).

instancetype, id, and isa

- instancetype keyword can only be used for return types, and it matches the return type to the receiver. init methods are always declared to return instancetype.
- Making an initializer return a pointer to the same type as the object itself will cause problem in subclassing.
- id is a type defined to stand for “a pointer to any object”.

  Use id when you want to point to an object but not sure what type it is going to be.

  Suppose you have a class called A that has a subclass called B, since B inherits all methods of A and if you have a method that returns a pointer to A when sent to an instance of A but a pointer to be when sent an instance of B, then you cannot define the method to return a pointer to A. Thus you use id as a return type. The OS will figure out what id is during run time.
- Objects are designed to know their type. This is done by having every object contain an instance variable called isa (“is a”) pointing to the class of the object.

  Run time environment will always be able to tell what type of object something is by looking at isa.

  All instances of BNRItem have their isa pointing to the class BNRItem

self, and super

- Inside a method, self is an implicit local variable pointing to the receiver.

  [myObject doSomething]; inside doSomething self points to myObject.

  Typically self is used so that an object can send messages to itself or return a pointer to itself

  In Objective-C self is a pointer so you do not need *self
- There are instances when an object needs to all a method the belongs to its superclass, such as the case when an initializer needs to make sure that the initializer of the superclass is called first. For this purpose the message is sent to super.

  super is a pointer to the superclass from which our object is subclassed.

Class Methods

- Class methods are sent to the Class itself such as alloc
- They are usually designed to :
  - Create instances such as the randomItem we will see in the RandomPossessions app, or
  - Retrieve or set some global property of the class
- Class methods are prefixed with + when they are declared or implemented

randomItem class method

Class Method randomItem

```
// Class method to create a random item
+(instancetype) randomItem
{
    int randomVersion = rand()%10;
    [BNRItem setVersion: randomVersion];
    NSLog(@"The version is %ld", [BNRItem version]);
    NSArray *randomAdjectiveList = [NSArray arrayWithObjects: @"Fluffy", @"Rusty", @"Shiny", nil];
    NSArray *randomNounList = [NSArray arrayWithObjects: @"Bear", @"Spork", @"Mac", nil];

    NSInteger adjectiveIndex = rand()%[randomAdjectiveList count];
    NSInteger nounIndex = rand()%[randomNounList count];
    NSString *randomName = [NSString stringWithFormat: @"%@ %@",[randomAdjectiveList objectAtIndex: adjectiveIndex], [randomNounList objectAtIndex: nounIndex]];

    NSString *randomSerialNumber = [NSString stringWithFormat: @"%c%c%c%c%c", '0'+rand()%10, 'A'+rand()%26 ,'0'+rand()%10, 'A'+rand()%26, '0'+rand()%10];

    int randomValue = rand()%100;

    BNRItem *newItem = [[self alloc] initWithName:randomName valueInDollars: randomValue serialNumber:randomSerialNumber];
    return  newItem;
}
```

Using class method to create instances

```
q = [q initWithName: @"Green Sofa" valueInDollars: 200
NSLog(@"%@",q);

NSMutableArray *items = [[NSMutableArray alloc] init];

for (int i = 0; i < 4; i++)
{
    BNRItem *v = [BNRItem randomItem];
    [items addObject: v];
}

for ( int i = 0;  i < [items count]; i++)
{
    NSLog(@"%@", [items objectAtIndex: i]);
```

```
+(id) randomItem
{
    [BNRItem setVersion: 2];
    NSLog(@"The version is %ld", [BNRItem version]);
    NSArray *randomAdjectiveList = [NSArray arrayWithObj
```

Using class methods to set and get some global properties of a class

Fast Enumeration and Exception Handling

```
// Using fast enumeration

for (BNRItem *item in items)
{
    NSLog(@"%@", item);
}
```

Iterating through an array using fast enumeration

If you do not catch it here it might be too late

```
29          [p initializeDateCreated];
30
31          [p doSomeWeiredStuff];      No visible @interface for 'BNRItem' declares the selector 'doSomeWeiredStuff'
32
33          NSLog(@"%@ %@ %i %@", [p itemName], [p serialNumber], [p valueInDollars], [p
```

```
2012-05-03 14:26:43.602 RandomPossessions[463:403] Rusty Spork (5S3Y5): Worth $69, recorded on 2012-05-03 18:26:43 +0000
2012-05-03 14:26:43.602 RandomPossessions[463:403] -[BNRItem doSomethignWeired]: unrecognized selector sent to instance 0x7fdfba600680
2012-05-03 14:26:43.607 RandomPossessions[463:403] *** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[BNRItem
doSomethignWeired]: unrecognized selector sent to instance 0x7fdfba600680'
*** First throw call stack:
(
    0   CoreFoundation                      0x00007fff896ebfc6 __exceptionPreprocess + 198
    1   libobjc.A.dylib                     0x00007fff8d1fbd5e objc_exception_throw + 43
    2   CoreFoundation                      0x00007fff897782ae -[NSObject doesNotRecognizeSelector:] + 190
    3   CoreFoundation                      0x00007fff896d8e73 ___forwarding___ + 371
    4   CoreFoundation                      0x00007fff896d8c88 _CF_forwarding_prep_0 + 232
    5   RandomPossessions                   0x000000010e505ac9 main + 1465
    6   RandomPossessions                   0x000000010e505504 start + 52
    7   ???                                 0x0000000000000001 0x0 + 1
)
terminate called throwing an exception(lldb)
```

Typical crash because a message could not be resolved