Chapter 3
Managing Memory with ARC

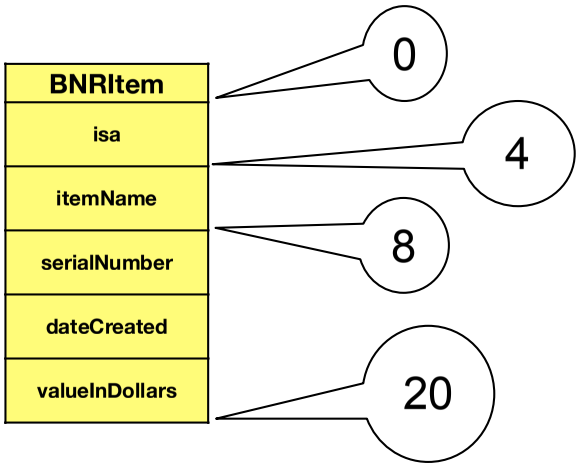
- The Stack
- The Heap
- Pointer Variables and Object Ownership
- Memory Management
- Strong and Weak References
- Properties
- Copying
- Dot Syntax
- Autorelease Pool and ARC

The Stack and the Heap

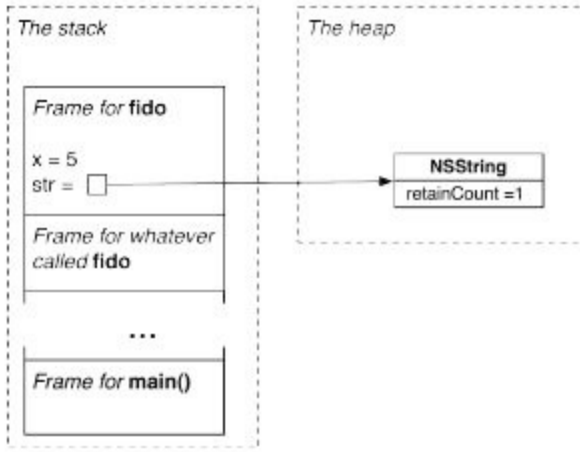
- When a function is loaded in memory to run it gets a fixed amount of memory allocated to it called frame.
- The frames for all functions are allocated in a fashion similar to a stack. This is referred to as the stack of the application
- Memory for local variables is allocated from the frame. (see x = 5 in the Frame of fido to the right).
- Instances of objects usually require more memory than primitive variables so when an function needs an instance of an object a two step process is employed:

  The memory needed for the instance, see BNRItem to the right, is allocated from a big chunk of ram allocated to the application and called the heap (A heaping mess of objects).

  The address of this memory is stored in the local variable in the frame of the function that points to this instance

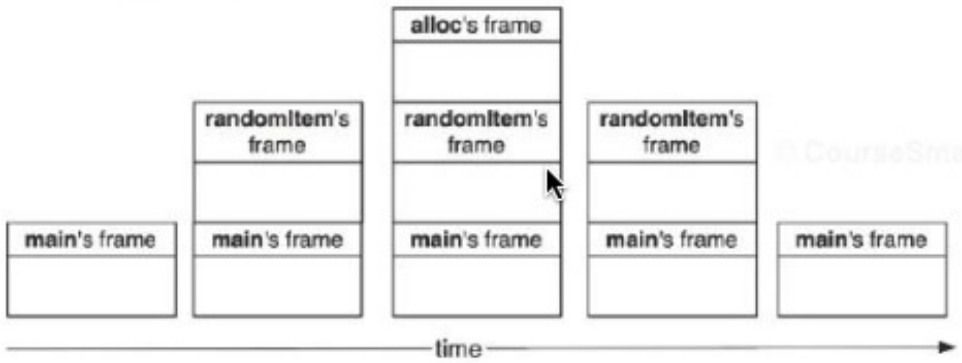


Pointer on the stack, object on the heap



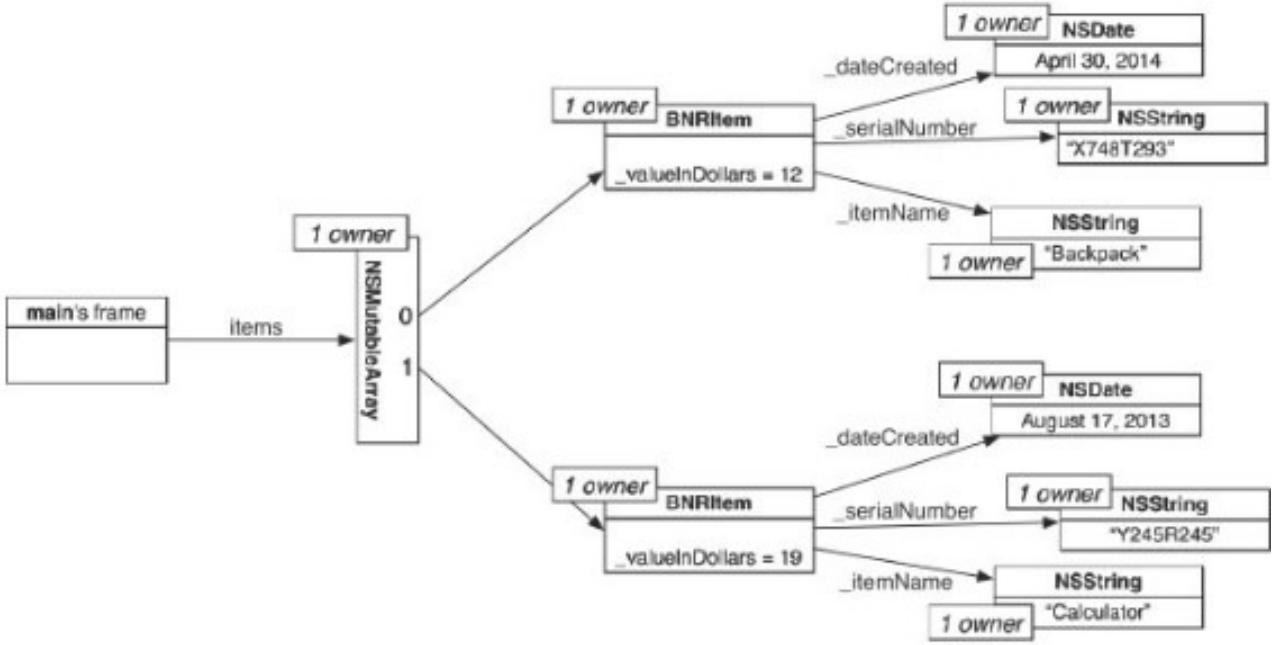
The Stack of RandomPossessions

- The stack grows and shrinks from the top.
- When a new function is loaded it's frame is allocated on top of the stack, and when it is done, its frame is removed from the top of the list.



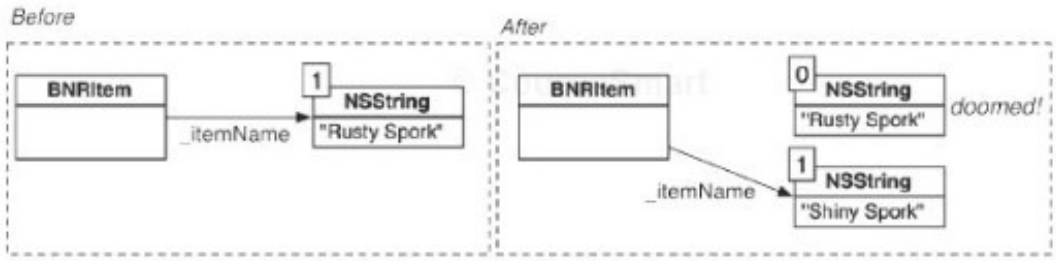
Pointer Variables and Object Ownership

- When a method (or function) has a local variable pointing to an object we say the method owns the object.
- When an object has an instance variable that points to another object the object with the pointer owns the object being pointed to.
- RandomPossessions main owns items which is an instance of NSMutableArray
- the Backpack instance of BNRItem owns the dateCreated instance it points to.



Losing Ownership

- There are four ways in which an object can loose one of it's owners:
  - The variable that points to the object is reassigned to point to another object (See example below).
  - The variable that points to the object is assigned to point to nil.
  - The owner of the object (variable that points to the object) is itself got destroyed.
  - An object in a collection is removed from that collection
- A destruction of a single object may set off a chain reaction of loss of ownership



Destroying items

```
11
12
13  @implementation BNRItem
14
15  - (void)dealloc
16  {
17      NSLog(@"Destroyed: %@", self);
18  }
19
20  // Class method to create a random item
21  +(instancetype) randomItem
22  {
```

```
43              NSLog(@"%@", [items objectAtIndex: i]);
44          }
45
46          // Using fast enumeration
47          for (BNRItem *item in items)
48          {
49              NSLog(@"%@", item);
50          }
51
52          NSLog(@"Setting items to nil ...");
53          items = nil;
54      }
```

```
2014-05-17 12:58:51.843 RandomPossessions[6500:303] Rusty Bear (9Z7M3): Worth $99, recorded on 2014-05-17 16:58:51 +0000
2014-05-17 12:58:51.844 RandomPossessions[6500:303] Shiny Spork (7E2L0): Worth $33, recorded on 2014-05-17 16:58:51 +0000
2014-05-17 12:58:51.844 RandomPossessions[6500:303] Setting items to nil ...
2014-05-17 12:58:51.844 RandomPossessions[6500:303] Destroyed: Rusty Mac (8Q2U8): Worth $23, recorded on 2014-05-17 16:58:51 +0000
2014-05-17 12:58:51.844 RandomPossessions[6500:303] Destroyed: Fluffy Mac (2Q7N7): Worth $29, recorded on 2014-05-17 16:58:51 +0000
2014-05-17 12:58:51.845 RandomPossessions[6500:303] Destroyed: Rusty Bear (9Z7M3): Worth $99, recorded on 2014-05-17 16:58:51 +0000
2014-05-17 12:58:51.845 RandomPossessions[6500:303] Destroyed: Shiny Spork (7E2L0): Worth $33, recorded on 2014-05-17 16:58:51 +0000
Program ended with exit code: 0
```

ARC and Memory Management

- To manage our limited RAM effectively we need to make sure that objects that are no longer needed by our application relinquish their memory back so that it can be reused and to insure that still needed objects remain with their memory intact.
- Apple (Mac OS) ensure this by maintaining a count of the object's owners.
- As long as an object has an owner, then the OS assumes it is still needed and keeps (*retain*) the objects memory allocated.
- When an object is released by all its owners, the owners count for the object becomes zero. The OS assumes that the object is no longer needed and it deallocate the memory used by the object i.e. (*release*) it .
- If an object that is no longer needed is kept around we encounter a "*memory leak*"
- When a still needed object is deallocated early we have "*premature deallocation*"
- With the introduction of iOS5 we no longer need to keep track of the owners of an object manually, it is done automatically for us by a memory management scheme known as "Automatic Reference Counting" or ARC

The retain cycle and strong references

- Anytime a pointer in object A points to another object say B we say that A owns B and in this case the reference of A to B is called a strong reference.
- A retain cycle is the situation when we have two (or more) objects hold strong references to each other, e.g. if object A has a strong reference to object B and object B has a strong reference to A.
- A retain cycle leads to memory leak because none of the objects in the cycle can be destroyed, i.g. A can never be destroyed since it has strong reference to B and B can never be destroyed since it has a strong reference to A.

Setting up a circular containment in BNRItem

```
    int _valueInDollars;
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    BNRItem *_container;
}

+(instancetype) randomItem;
-(instancetype)initWithName: (NSString *) name valueInDoll
-(instancetype)initWithItemName:(NSString *)name;

- (void)setContainedItem:(BNRItem *)item;
- (BNRItem *)containedItem;

- (void)setContainer:(BNRItem *)item;
- (BNRItem *)container;
```

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

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

        BNRItem *backpack = [[BNRItem alloc] initWithItemName:@"Backpack"];
        [items addObject: backpack];

        BNRItem *calculator = [[BNRItem alloc] initWithItemName:@"Calculator"];
        [items addObject: calculator];

        backpack.containedItem = calculator;

        backpack = nil;
        calculator = nil;

        // Using fast enumeration
        for (BNRItem *item in items)
        {
            NSLog(@"%@", item);
        }

        NSLog(@"Setting items to nil ...");
        items = nil;
    }
    return 0;

}
```

circular reference

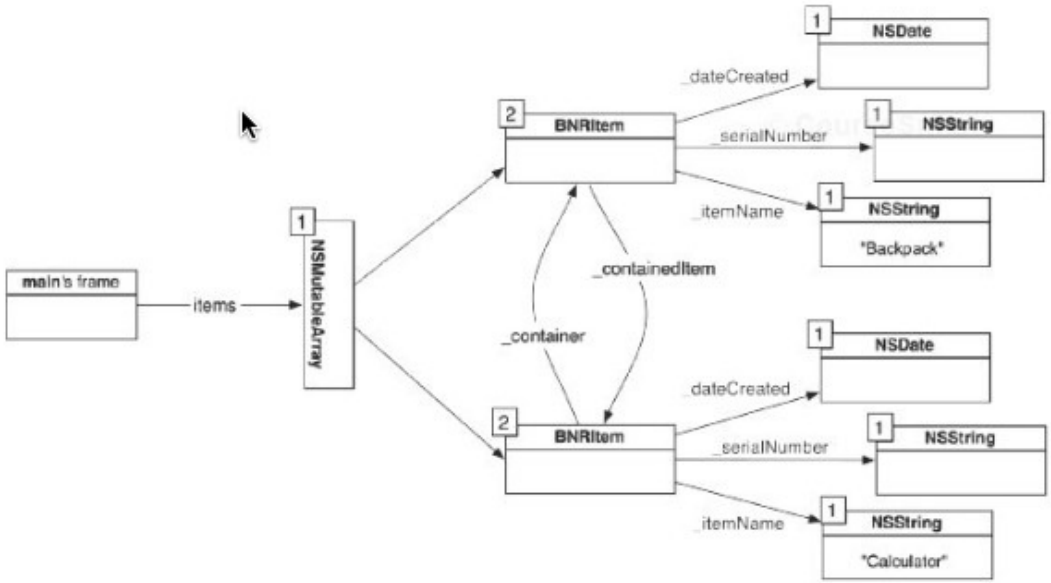
Destroying objects

No de-allocation of memory

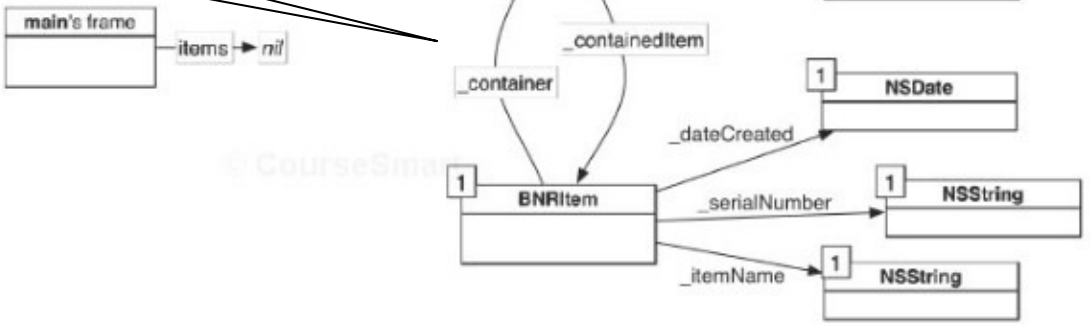
No Selection

```
2014-05-17 13:46:26.209 RandomPossessions[6703:303] Backpack ((null)): Worth $0, recorded on 2014-05-17 17:46:26 +0000
2014-05-17 13:46:26.209 RandomPossessions[6703:303] Calculator ((null)): Worth $0, recorded on 2014-05-17 17:46:26 +0000
2014-05-17 13:46:26.210 RandomPossessions[6703:303] Setting items to nil ...
Program ended with exit code: 0
```

RandomItems with strong reference cycle



Memory Leak

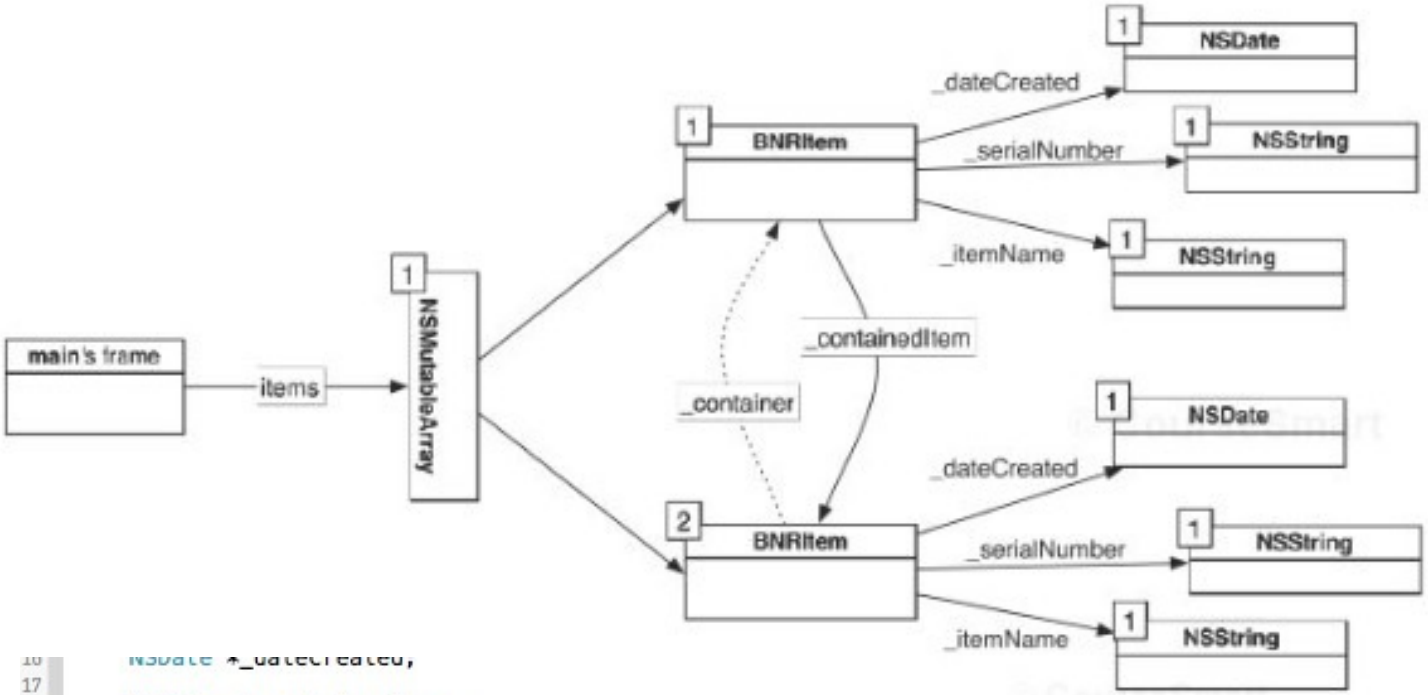


Weak References

- To eliminate the existence of a retain cycle we define a new type of ownership we call weak ownership in which an object can point to (contain a pointer to) another object but not own the object.
- The declaration of weak references is done by using the qualifier __weak in the pointer declaration statement as in

  __weak BNRItem *container;
- The the object containing this statement has a weak reference to container (calculator points to backpack but does not own it).
- A weak reference knows when the object that it points to is destroyed and responds by setting itself to nil.
- Thus, if the backpack is destroyed, the calculator's _ container instance variable will be automatically set to nil. This is convenient. If _ container was not set to nil, then destroying the object would leave you with a dangling pointer, which could crash your application.

RandomItems with strong reference cycle avoided




```
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    // BNRItem *_container;
    __weak BNRItem *_container;
}
```

```
No Selection
2014-05-17 14:24:29.952 RandomPossessions[6868:303] Backpack ((null)): Worth $0, recorded on 2014-05-17 18:24:29 +0000
2014-05-17 14:24:29.952 RandomPossessions[6868:303] Calculator ((null)): Worth $0, recorded on 2014-05-17 18:24:29 +0000
2014-05-17 14:24:29.953 RandomPossessions[6868:303] Setting items to nil ...
2014-05-17 14:24:29.953 RandomPossessions[6868:303] Destroyed: Backpack ((null)): Worth $0, recorded on 2014-05-17 18:24:29 +0000
2014-05-17 14:24:29.953 RandomPossessions[6868:303] Destroyed: Calculator ((null)): Worth $0, recorded on 2014-05-17 18:24:29 +0000
Program ended with exit code: 0
```

Deciding on weak references

- We should break the relationship between the objects in a retain cycle as a parent/child relationship.

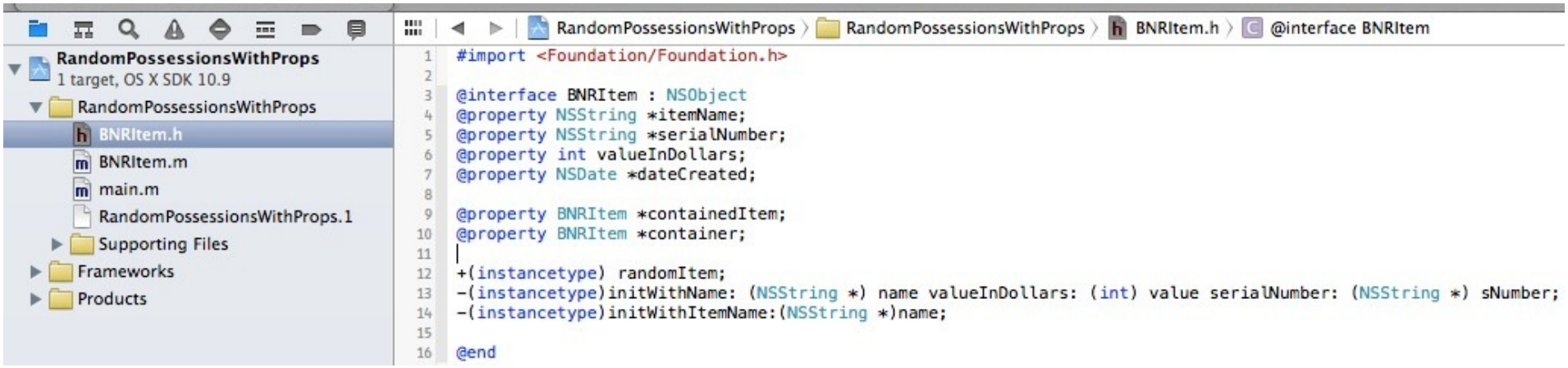
  Parents have a strong reference to their children

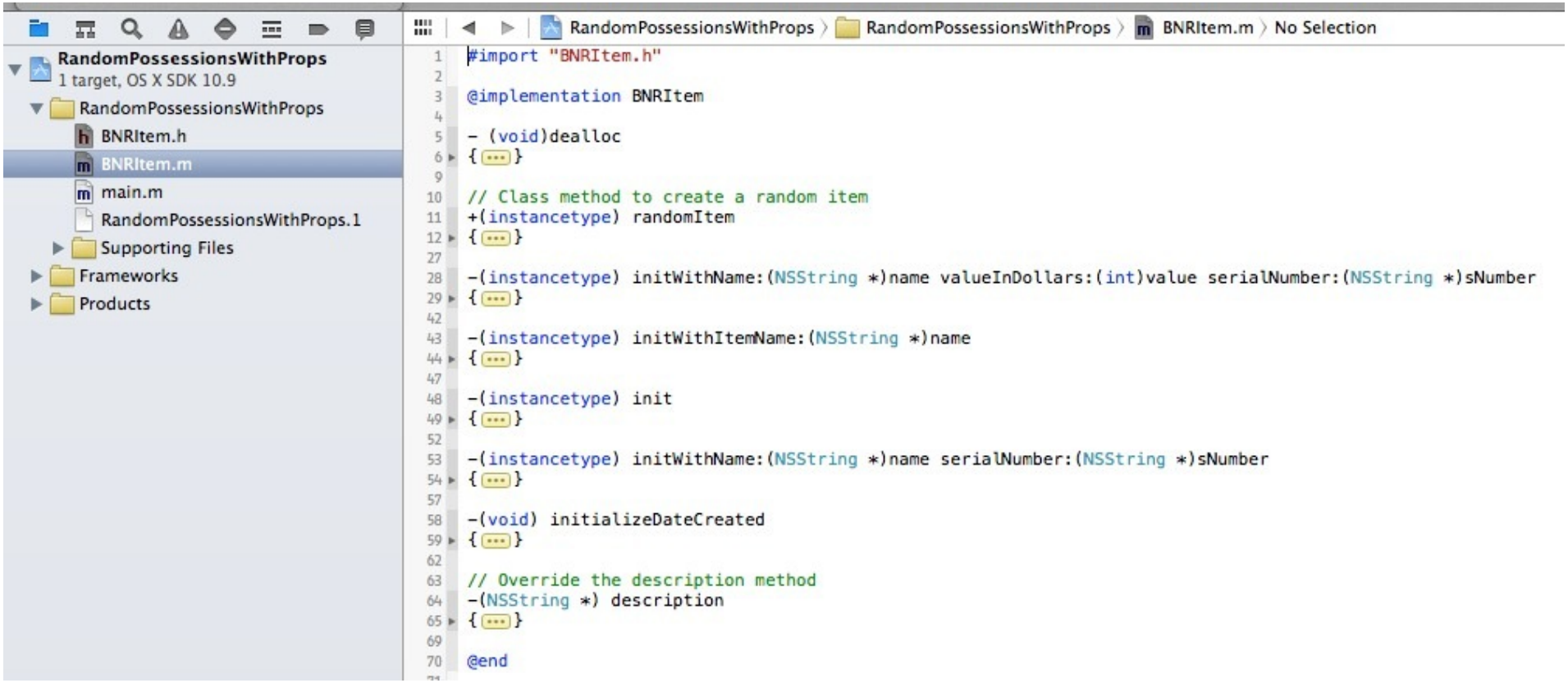
  Children have a weak reference to their parents
- If the parent object is destroyed, the child (which has a weak reference to the parent and does not own it) becomes aware of it and the pointer to the parent becomes automatically nil.
- __unsafe_unretained is used for backward compatibility since iOS prior to iOS5 cannot use weak references. These references do not take ownership of the object they point to but do not set it t nil either (that is why it is unsafe).

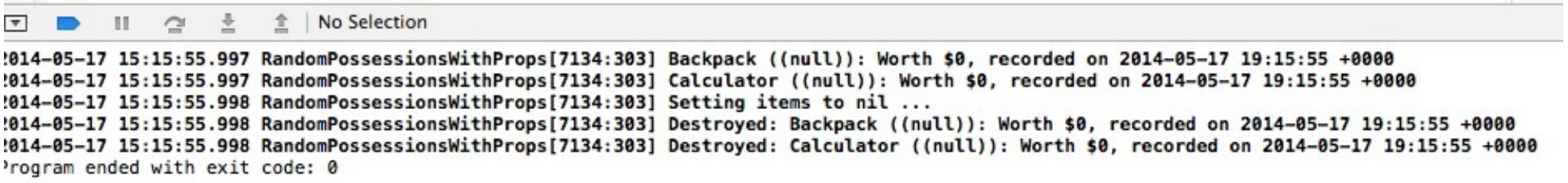
Properties

- Properties are a convenience offered to spare us having to write getters and setters for instance variables.
- Properties are declares in the interface of the class using

  @property (Attribute list) dataType propertyName
- When you declare a property, you are declaring an instance variable, it's setter and it's getter
- Current convention in Objective-C is to use an _ as the first character for instance variables.




RandomPossessionsWithProps
1 target, OS X SDK 10.9
RandomPossessionsWithProps
BNRItem.h
BNRItem.m
main.m
RandomPossessionsWithProps.1
Supporting Files
Frameworks
Products
RandomPossessionsWithProps 〉 RandomPossessionsWithProps 〉 BNRItem.m 〉 No Selection
#import "BNRItem.h"
@implementation BNRItem
- (void)dealloc
// Class method to create a random item
+(instancetype) randomItem
-(instancetype) initWithName:(NSString *)name valueInDollars:(int)value serialNumber:(NSString *)sNumber
-(instancetype) initWithItemName:(NSString *)name
-(instancetype) init
-(instancetype) initWithName:(NSString *)name serialNumber:(NSString *)sNumber
-(void) initializeDateCreated
// Override the description method
-(NSString *) description
@end


No Selection
2014-05-17 15:15:55.997 RandomPossessionsWithProps[7134:303] Backpack ((null)): Worth $0, recorded on 2014-05-17 19:15:55 +0000
2014-05-17 15:15:55.997 RandomPossessionsWithProps[7134:303] Calculator ((null)): Worth $0, recorded on 2014-05-17 19:15:55 +0000
2014-05-17 15:15:55.998 RandomPossessionsWithProps[7134:303] Setting items to nil ...
2014-05-17 15:15:55.998 RandomPossessionsWithProps[7134:303] Destroyed: Backpack ((null)): Worth $0, recorded on 2014-05-17 19:15:55 +0000
2014-05-17 15:15:55.998 RandomPossessionsWithProps[7134:303] Destroyed: Calculator ((null)): Worth $0, recorded on 2014-05-17 19:15:55 +0000
Program ended with exit code: 0

Property Attributes

- Properties are declares in the interface of the class using

  @property (Attribute list) dataType propertyName

- The attributes specify:

  Multi-Threading behavior (Atomicity): Use it to specify that accessor methods are not atomic (nonatomic). By default, accessors are atomic meaning that they are thread safe and all needed locking is generated. If you specify nonatomic, a synthesized accessor for an object property simply returns the value directly.

  Read/Write behavior (Writability): Specify whether or not a property has an associated set accessor.

  - readwrite = should be treated as read/write (default).
  - readonly = Does not need a setter just a getter.

  Memory management attribute: These are

  strong which is the default (owning the destination object),

  weak (not owning the destination object),

  copy (a copy of the object should be used, assign (the setter uses simple assignment. This attribute is the default),

  unsafe_unretained (will discuss later)

Property Synthesis

- Declaring a property in a class interface only declares the accessor methods in a class interface.
- In order for a property to automatically generate an instance variable and the implementations for its methods, it must be synthesized, either implicitly or explicitly.
- Properties are implicitly synthesized by default.
- A property is explicitly synthesized by using the the @synthesize directive in an implementation file as in

  @synthesize propertyName = backing variable
- The typical backing variable name is _ propertyName
- If the backing variable name is omitted a default one will be created which is the same as propertyName
- If you do not want a backing variable you must override the setter and getter for a property which prevents the compiler for synthesizing the property

Custom accessors with properties

By default, the accessors that a property implements are very simple and look like this:

```
3  @implementation BNRItem
4
5  - ( void) setContainedItem:( BNRItem *) containedItem
6  {
7      _containedItem = containedItem;
8  }
9
```

When the compiler sees that you have implemented setContainedItem:, it will not create a default setter for containedItem. It will still create the getter method, containedItem.

However, for the containedItem property, the default setter method is not sufficient.
The implementation of setContainedItem: needs an extra step
It should also set the container property of the item being contained.
You can replace the default setter by implementing the setter yourself in the implementation file to make it look like:

```
3  @implementation BNRItem
4
5  - ( void) setContainedItem:( BNRItem *) containedItem
6  {
7      _containedItem = containedItem;
8      self.containedItem.container = self;
9  }
```

Using Properties in Random Possessions

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

@interface BNRItem : NSObject

@property (nonatomic, copy) NSString *itemName;
@property (nonatomic, copy) NSString *serialNumber;
@property (nonatomic)       int valueInDollars;
@property (nonatomic, readonly, strong) NSDate *dateCreated;

@property (nonatomic, strong) BNRItem *containedItem; // A pointer to the child (the contained item)
@property (nonatomic, weak)   BNRItem *container;     // A pointer to the parent (the container)

+(id) randomItem;

-(id) initWithName: (NSString *) name valueInDollars: (int) value serialNumber: (NSString *) sNumber;
-(id) initWithName: (NSString *)name serialNumber: (NSString *)sNumber;

-(void) initializeDateCreated;

@end
```

The Interface

The Implementation

- Since itemName and serialNumber are mutable strings we must prevent them from being change inadvertently by other processes so we use a copy setters to give us our own copy of the string when we initialize the property.
- We do not need a setter for dateCreated we define it as a readonly property.
- valueInDollars is just the value of a primitive so the setter needs to assign the value to the instance variable. This means that the setter semantics is assign which is the default.
- the property container points to the container containing the pointing item. This is to say that the container is the parent and the contained item is the child whics is reflected in the weak setter attribute for container

```objc
#import "BNRItem.h"

@implementation BNRItem

@synthesize itemName;
@synthesize serialNumber;
@synthesize valueInDollars;
@synthesize dateCreated;

@synthesize containedItem;
@synthesize container;
```