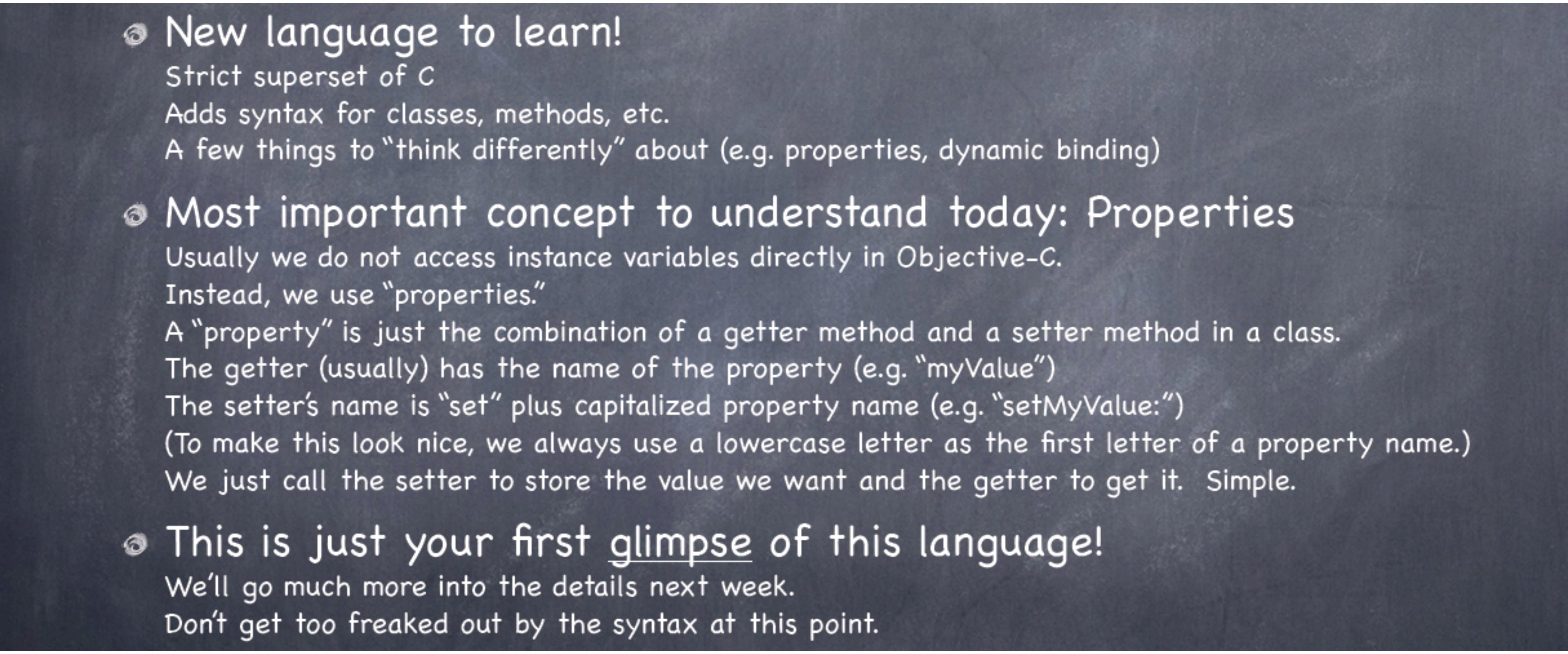
iOS 7 & Objective C

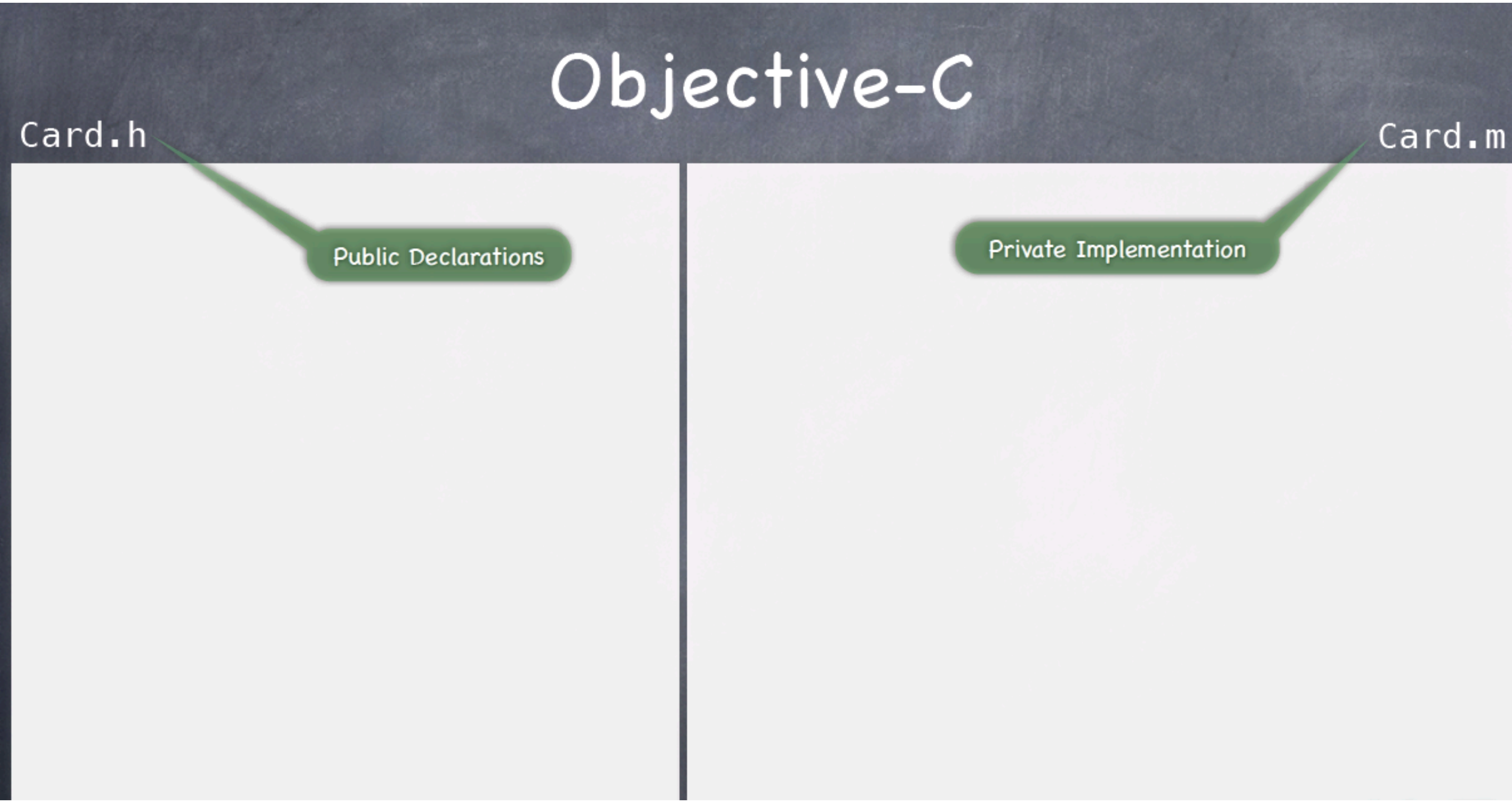
Lecture 2

Objective-C

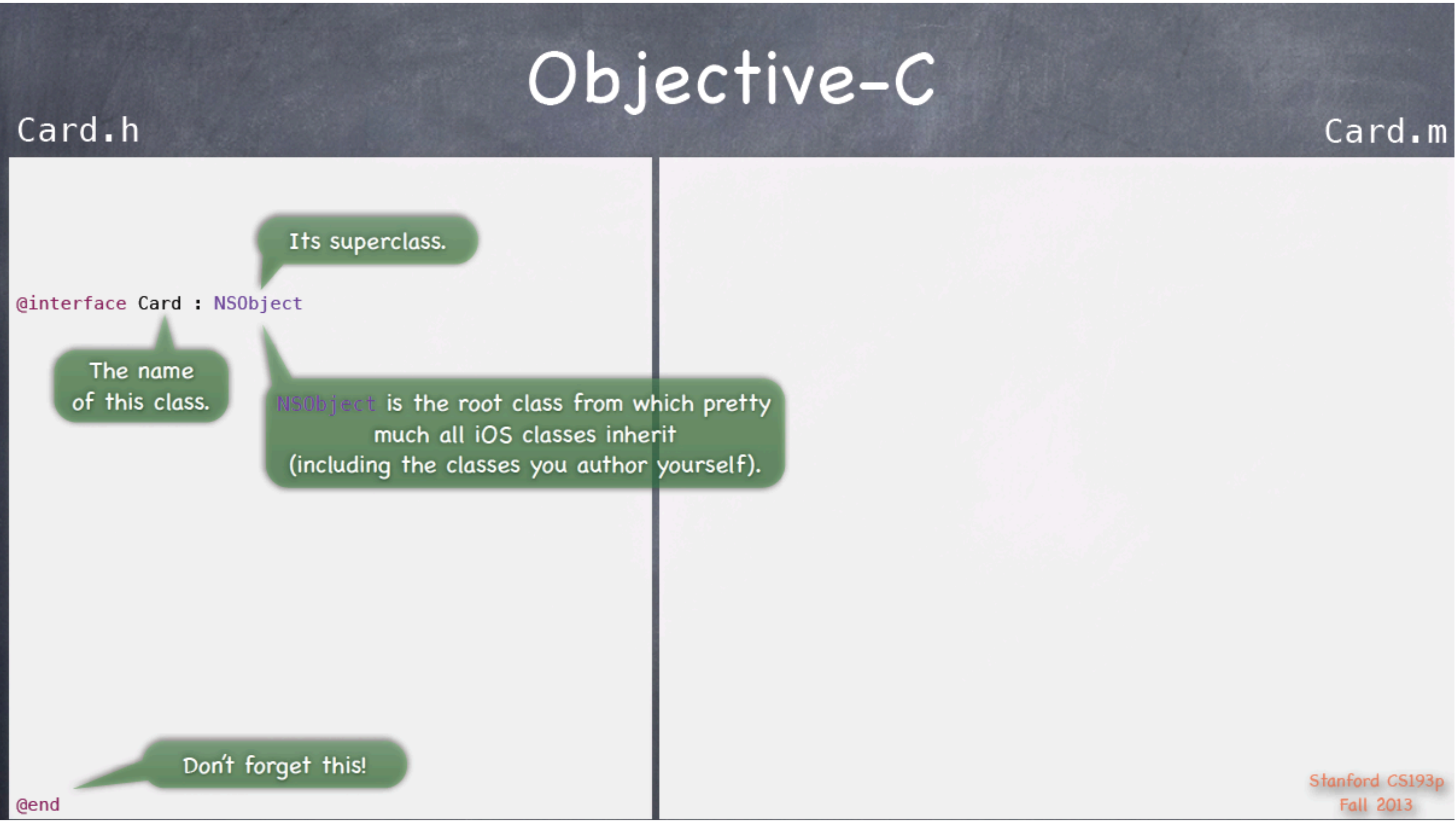


- New language to learn!
  Strict superset of C
  Adds syntax for classes, methods, etc.
  A few things to "think differently" about (e.g. properties, dynamic binding)
- Most important concept to understand today: Properties
  Usually we do not access instance variables directly in Objective-C.
  Instead, we use "properties."
  A "property" is just the combination of a getter method and a setter method in a class.
  The getter (usually) has the name of the property (e.g. "myValue")
  The setter's name is "set" plus capitalized property name (e.g. "setMyValue:")
  (To make this look nice, we always use a lowercase letter as the first letter of a property name.)
  We just call the setter to store the value we want and the getter to get it. Simple.
- This is just your first glimpse of this language!
  We'll go much more into the details next week.
  Don't get too freaked out by the syntax at this point.

2 Files For Every Class



Header File ".h"



Objective-C

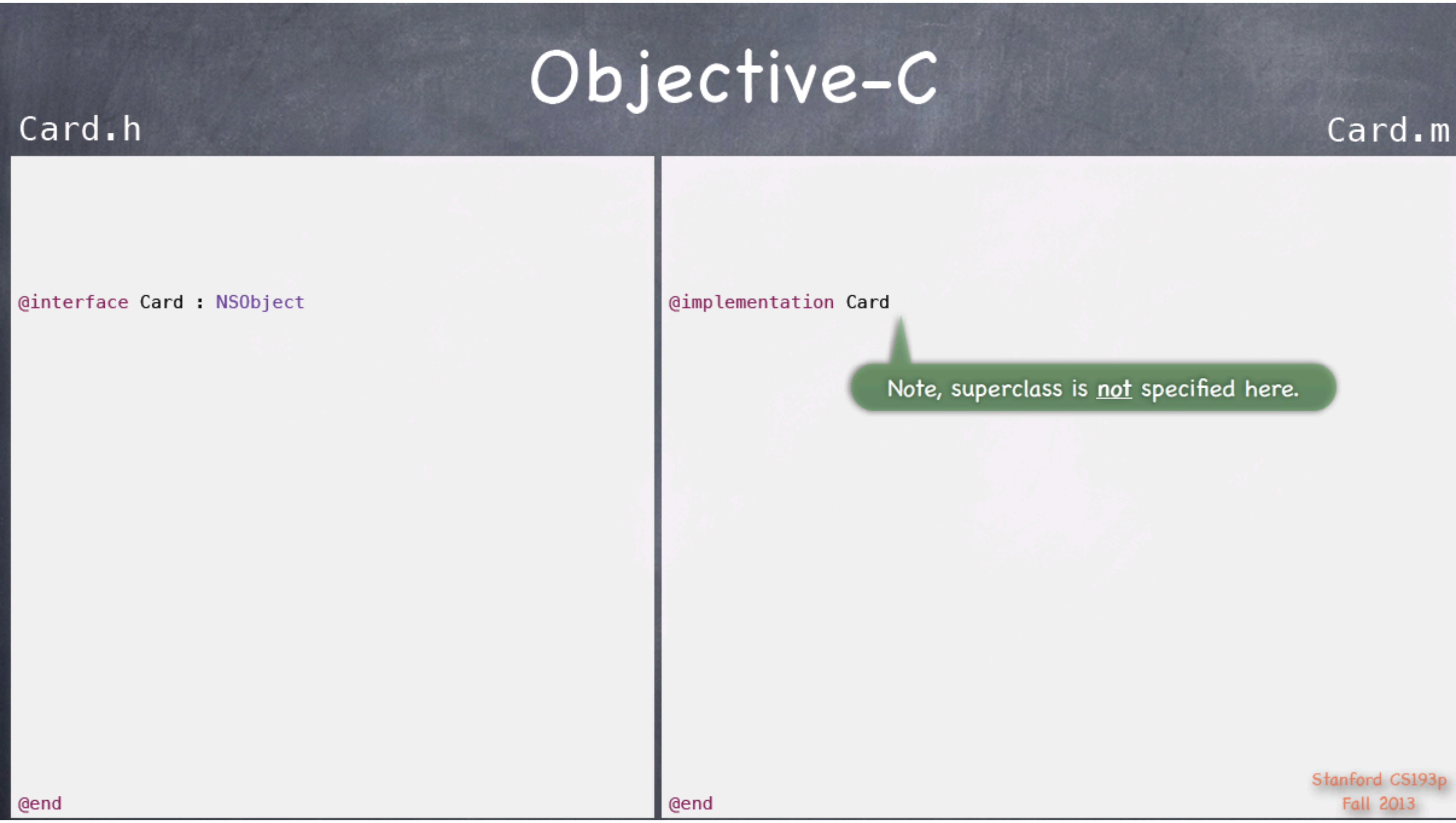
Card.h

Card.m

```
@interface Card : NSObject

@end
```

Its superclass.
The name of this class.
NSObject is the root class from which pretty much all iOS classes inherit (including the classes you author yourself).
Don't forget this!
Stanford CS193p Fall 2013

Implementation File ".m"



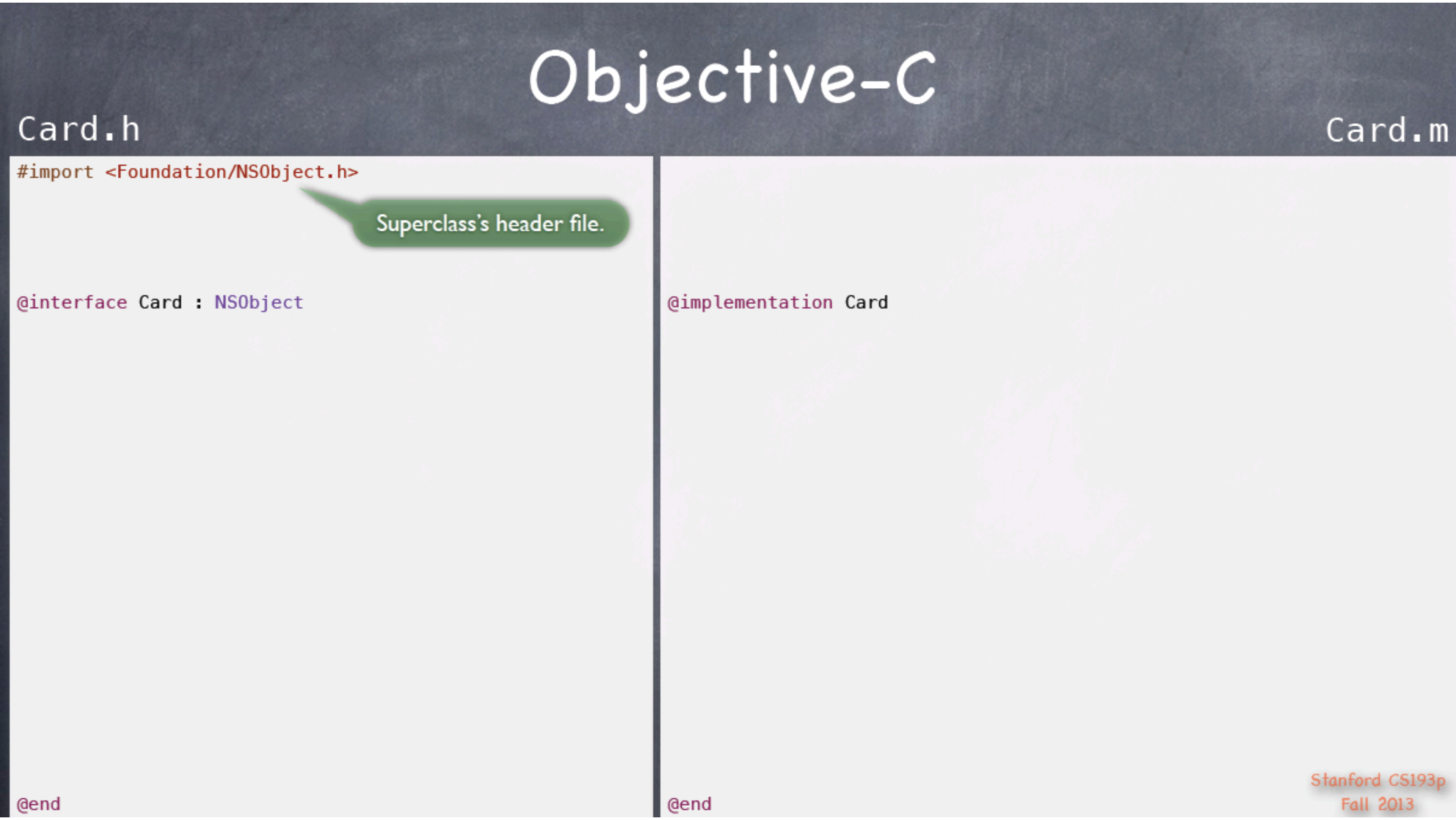
Card.h

```
@interface Card : NSObject

@end
```

Card.m

```
@implementation Card

@end
```

#import



Objective-C

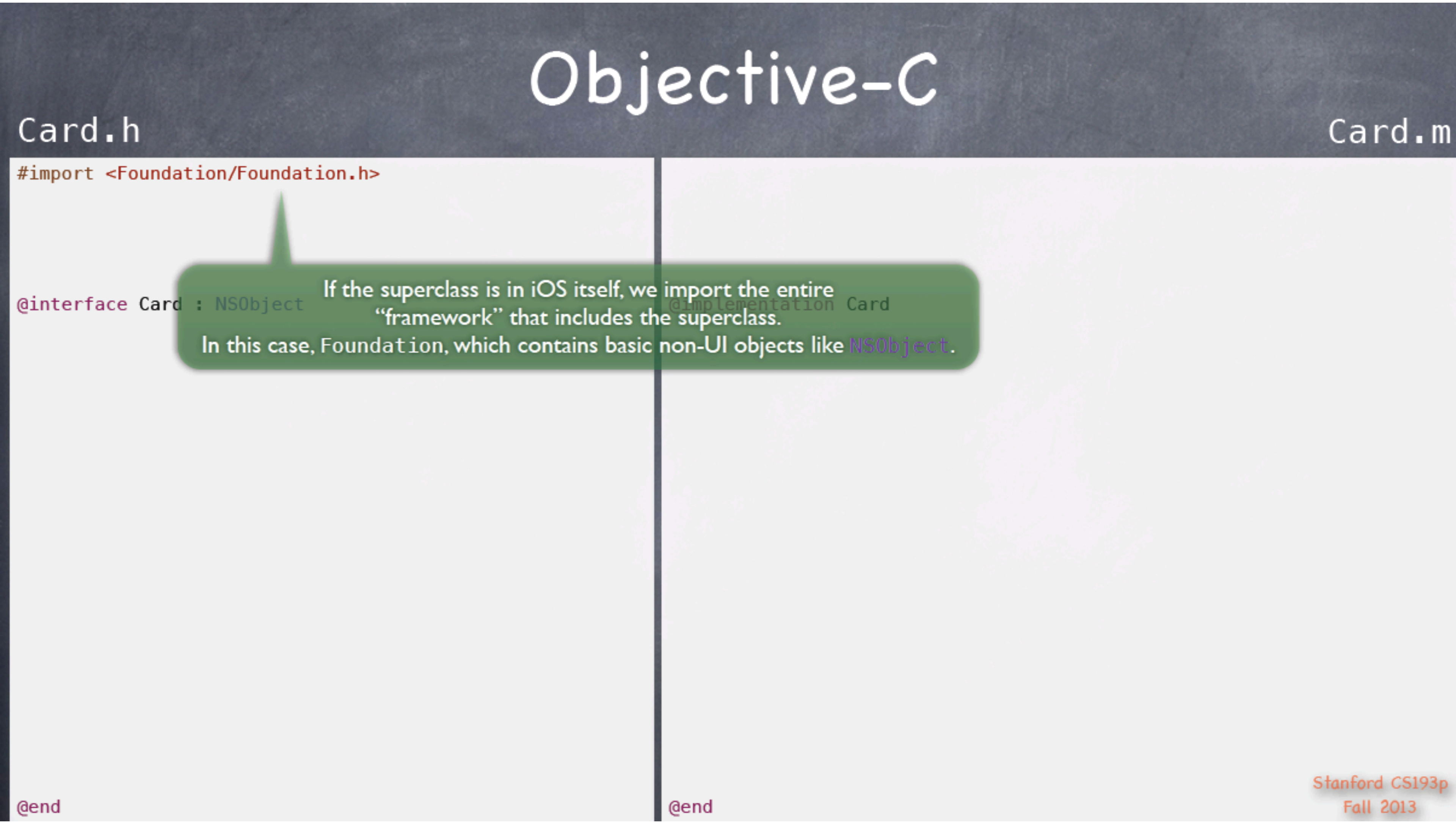
Card.h

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



@interface Card : NSObject









@end
```

Card.m

```
@implementation Card









@end
```

Superclass in iOS



Objective-C

Card.h

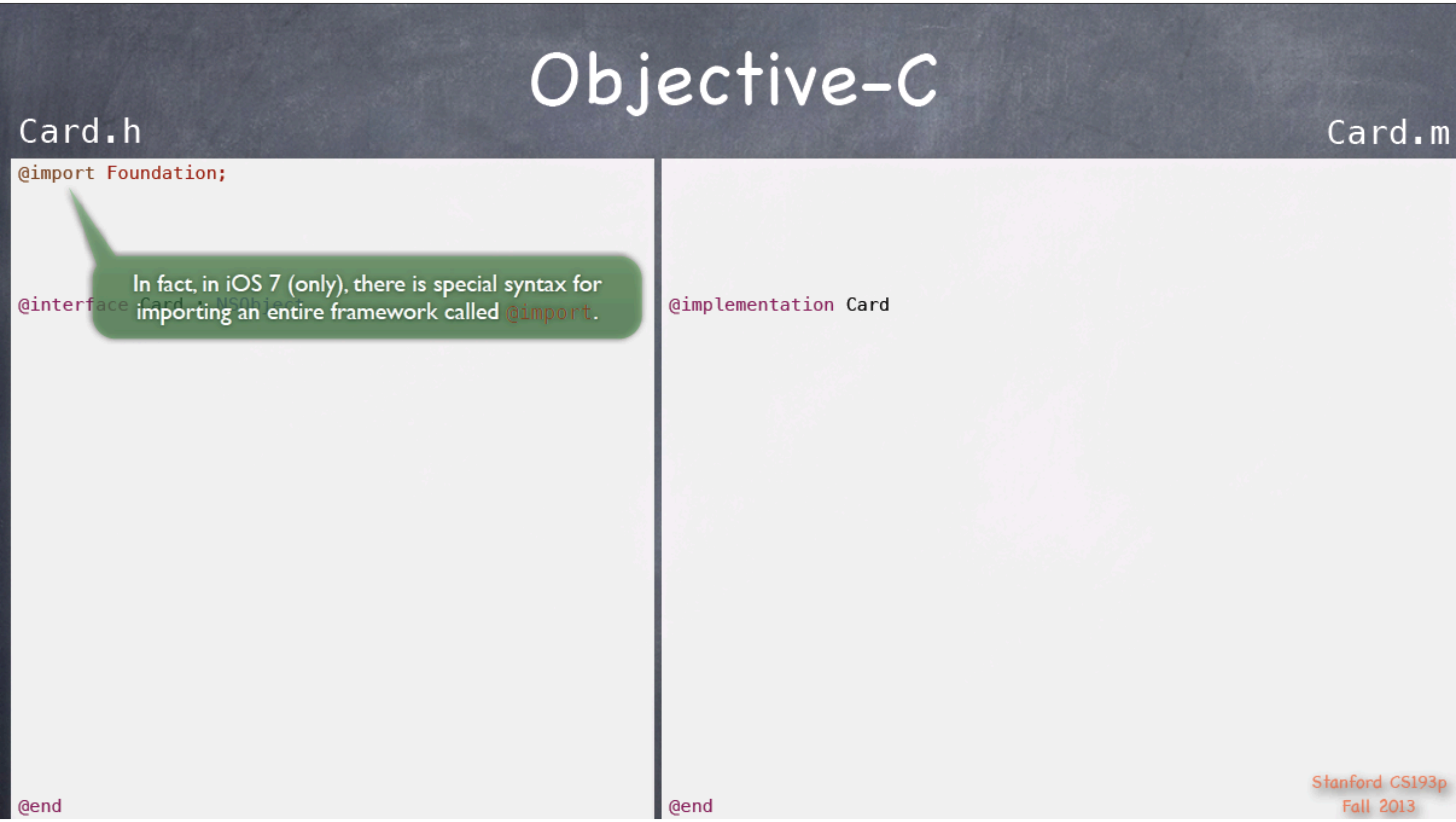
```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@end
```

Card.m

```
@implementation Card

@end
```

If the superclass is in iOS itself, we import the entire "framework" that includes the superclass.
In this case, Foundation, which contains basic non-UI objects like NSObject.

@import



Objective-C

Card.h

```
@import Foundation;



@interface Card : NSObject







@end
```

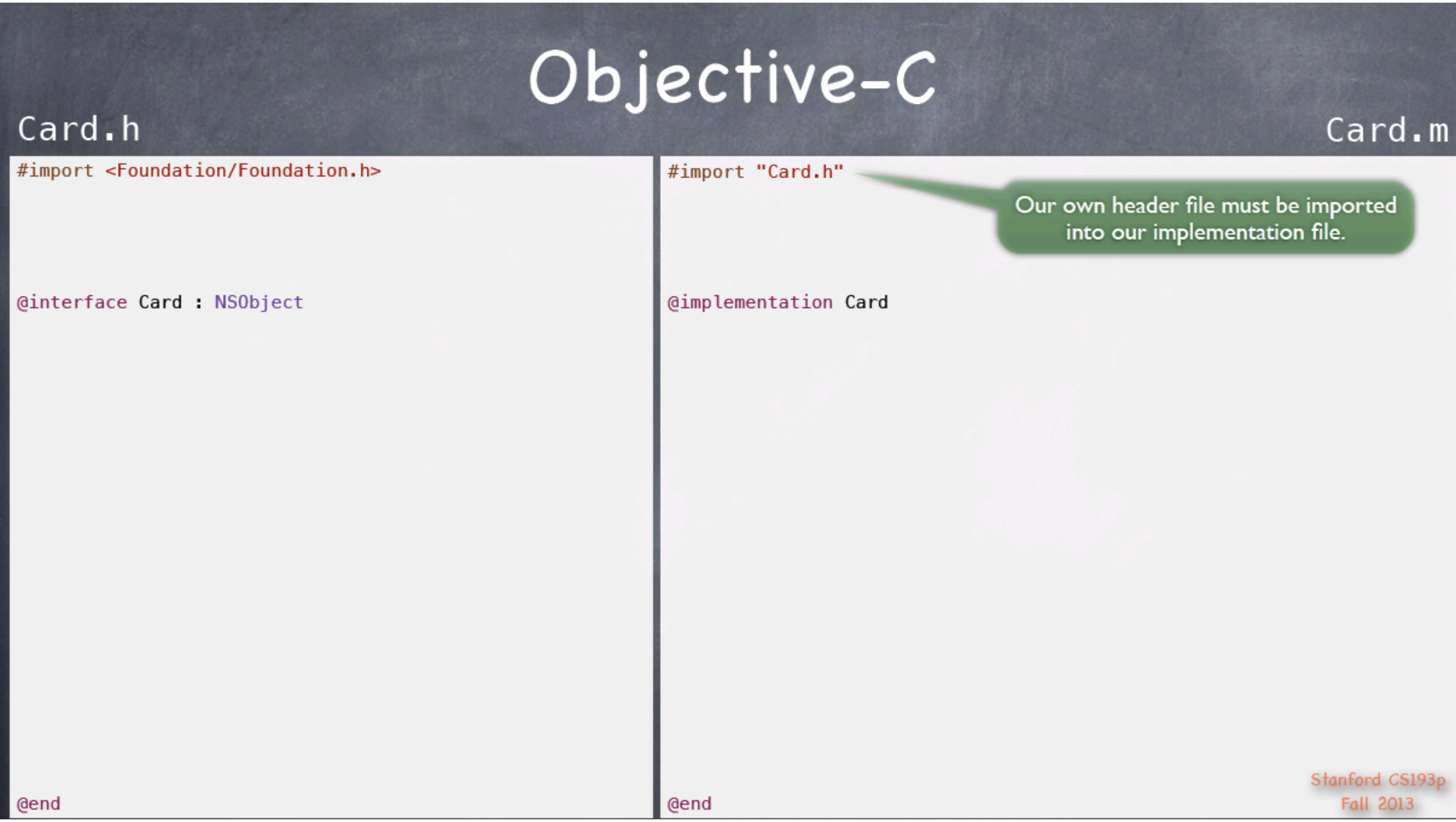
In fact, in iOS 7 (only), there is special syntax for importing an entire framework called `@import`.

Card.m

```
@implementation Card







@end
```

Stanford CS193p
Fall 2013

Must Import Our Own Header File



Objective-C

Card.h

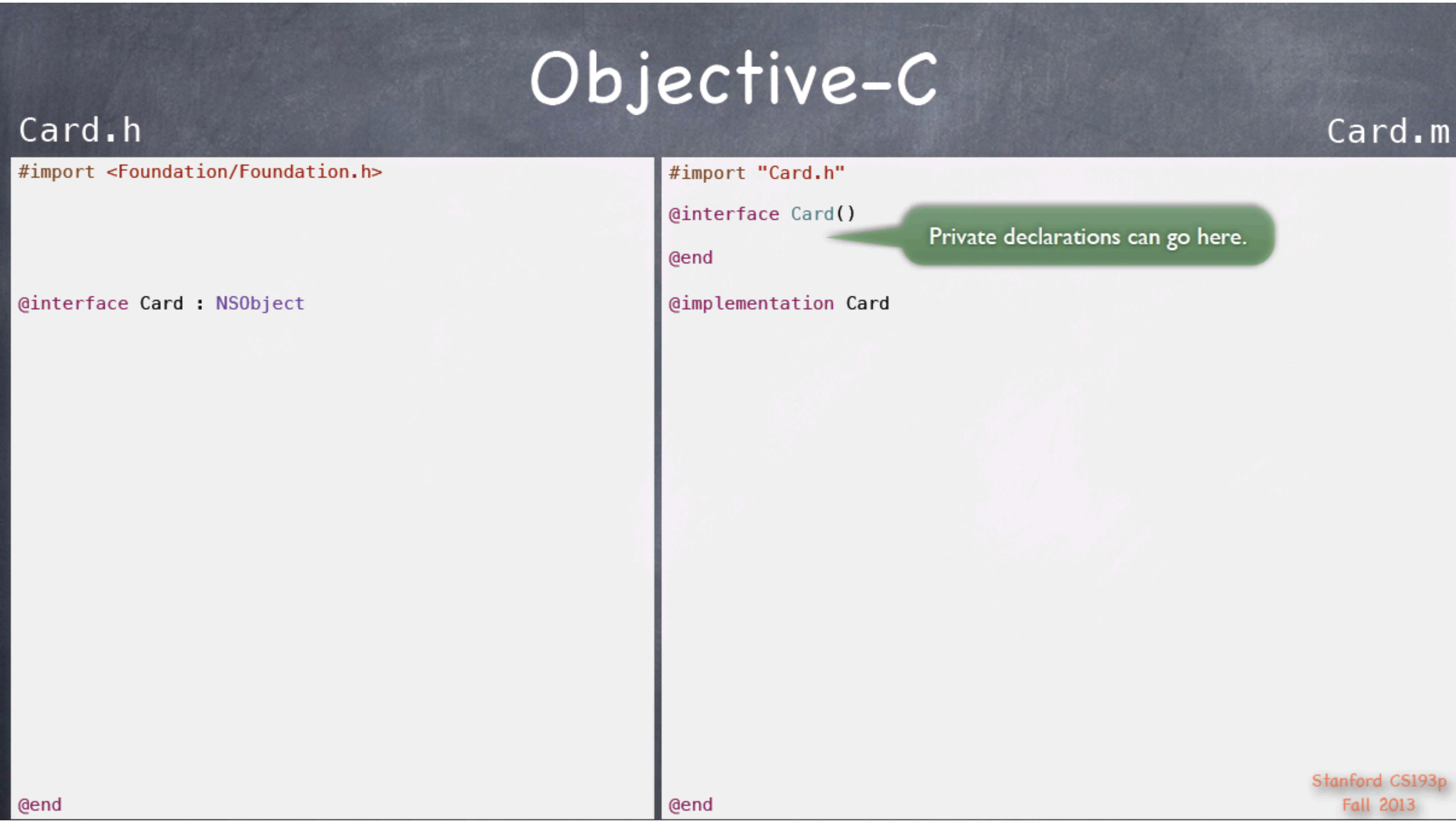
```
#import <Foundation/Foundation.h>



@interface Card : NSObject




















@end
```

Card.m

```
#import "Card.h"



@implementation Card




















@end
```

Our own header file must be imported into our implementation file.

Private Declarations



Objective-C

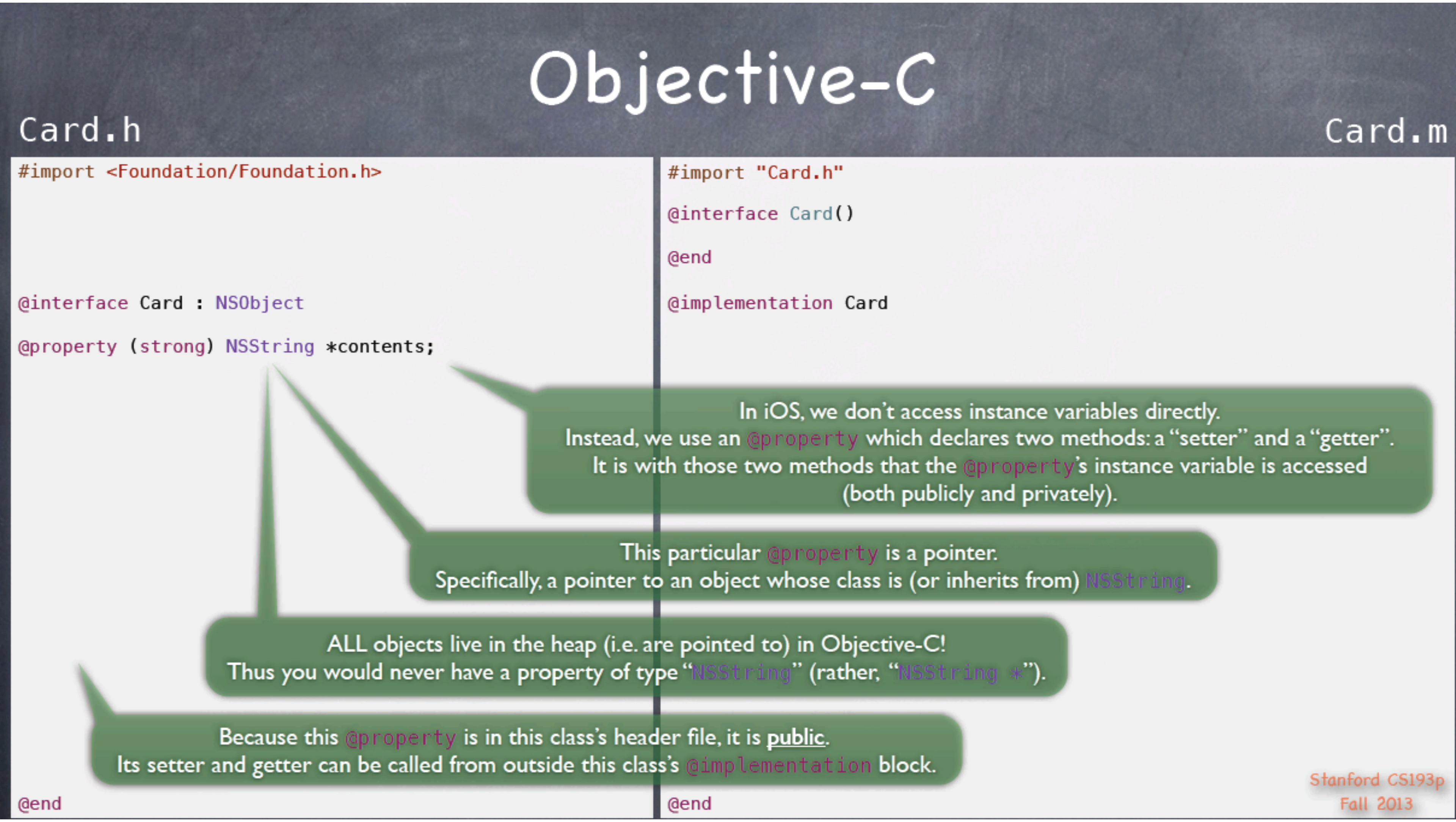
Card.h

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



@interface Card : NSObject

@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@end
```

Our First Property



Objective-C

Card.h

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

@interface Card : NSObject

@property (strong) NSString *contents;

@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@end
```

In iOS, we don't access instance variables directly.
Instead, we use an @property which declares two methods: a "setter" and a "getter".
It is with those two methods that the @property's instance variable is accessed
(both publicly and privately).

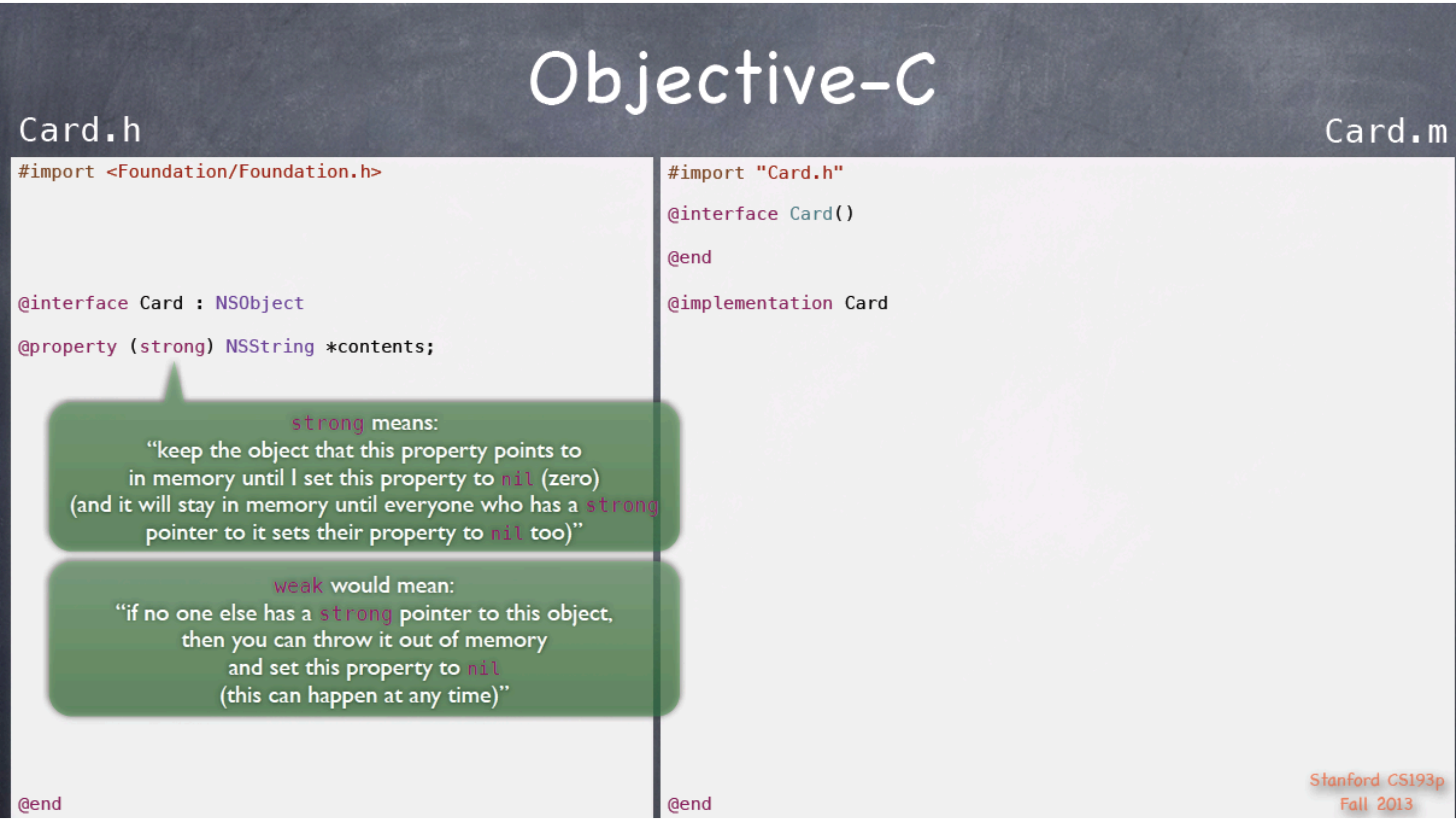
This particular @property is a pointer.
Specifically, a pointer to an object whose class is (or inherits from) NSString.

ALL objects live in the heap (i.e. are pointed to) in Objective-C!
Thus you would never have a property of type "NSString" (rather, "NSString *").

Because this @property is in this class's header file, it is <u>public</u>.
Its setter and getter can be called from outside this class's @implementation block.

Stanford CS193p
Fall 2013

strong or weak



Objective-C

Card.h

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



@interface Card : NSObject

@property (strong) NSString *contents;






@end
```

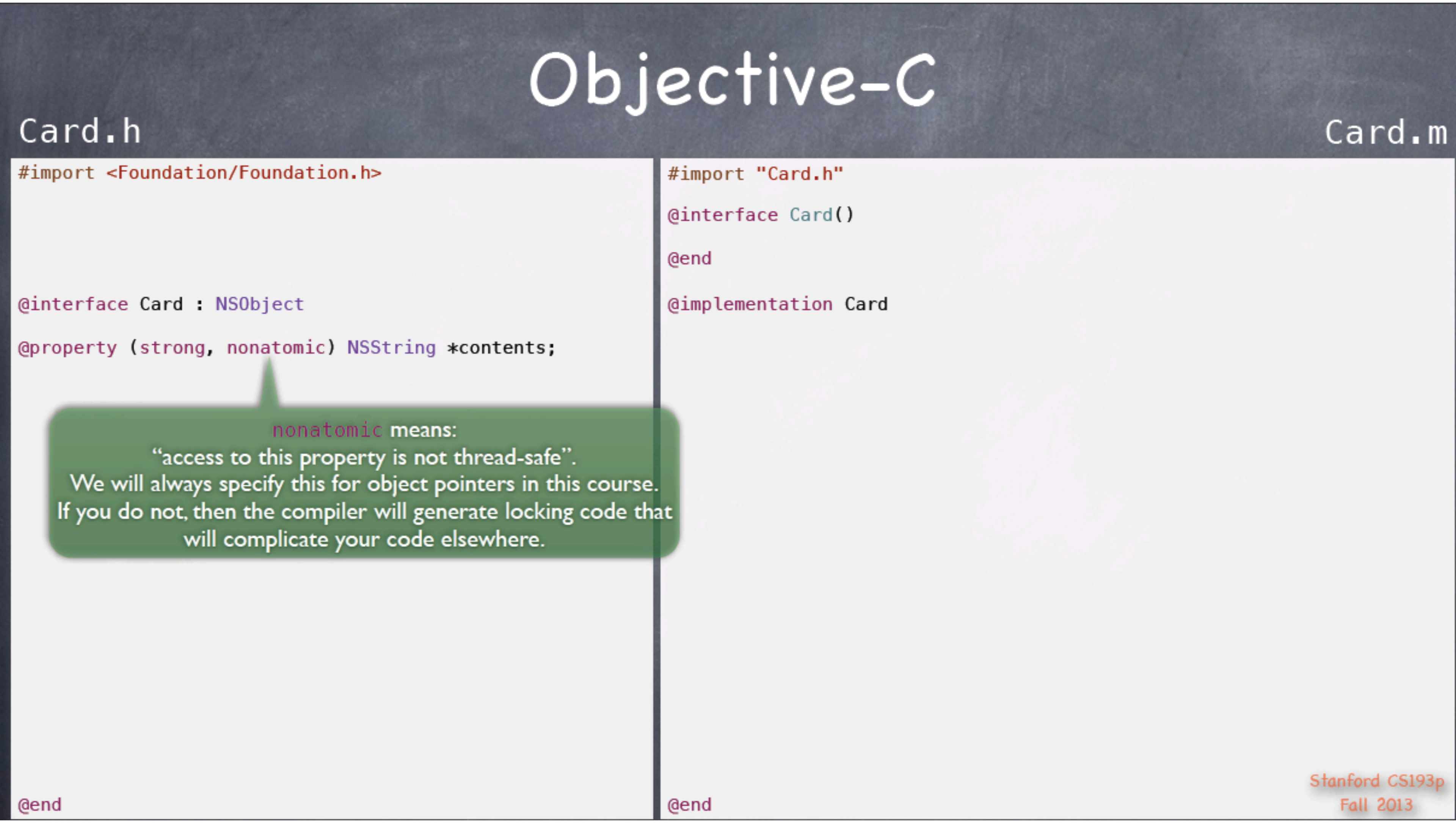
strong means:
"keep the object that this property points to
in memory until I set this property to nil (zero)
(and it will stay in memory until everyone who has a strong
pointer to it sets their property to nil too)"

weak would mean:
"if no one else has a strong pointer to this object,
then you can throw it out of memory
and set this property to nil
(this can happen at any time)"

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card






@end
```

atomic or nonatomic



Objective-C

Card.h

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



@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;









@end
```

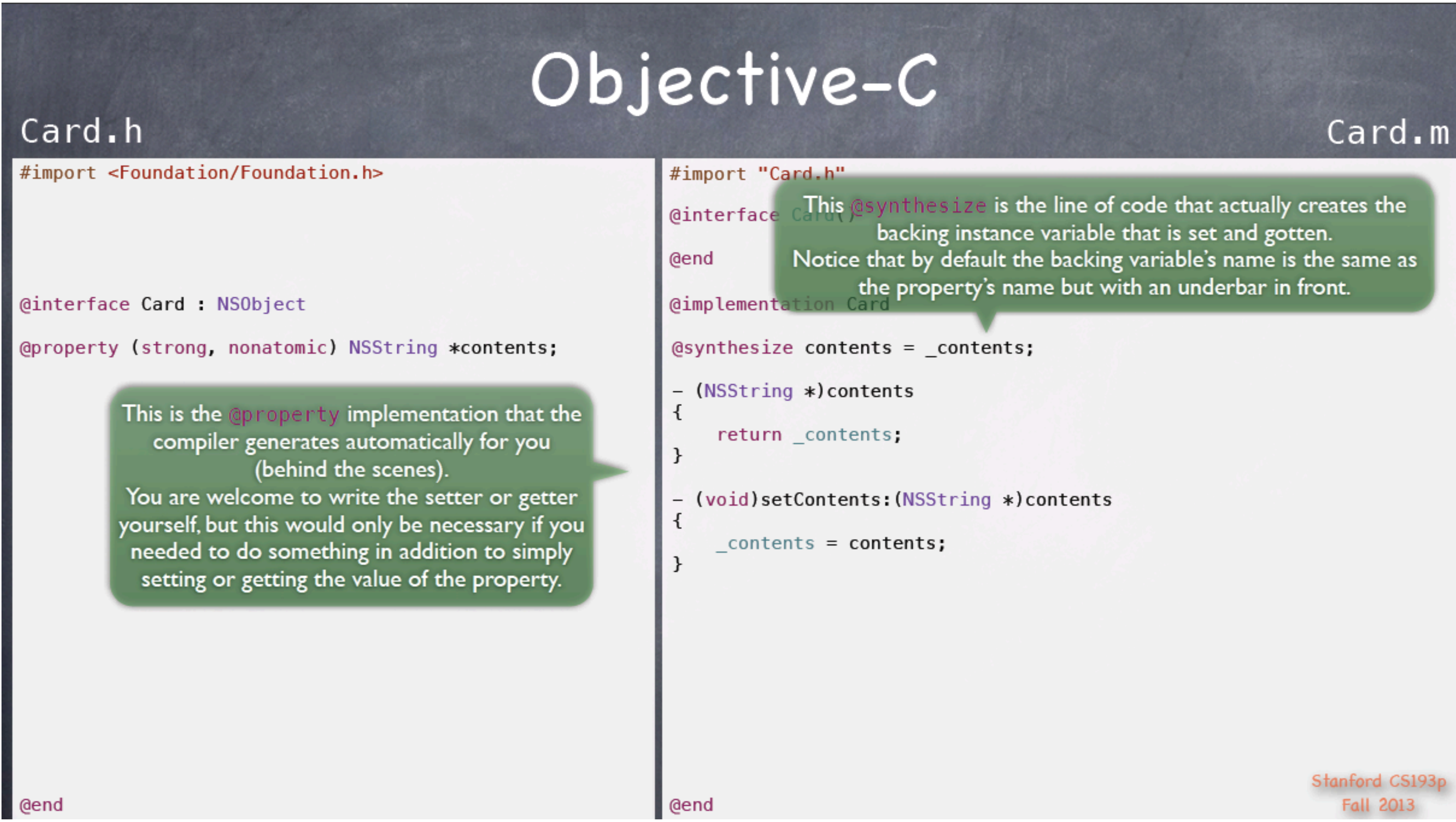
nonatomic means:
"access to this property is not thread-safe".
We will always specify this for object pointers in this course.
If you do not, then the compiler will generate locking code that will complicate your code elsewhere.

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card














@end
```

Stanford CS193p
Fall 2013

synthesize



Objective-C

Card.h

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



@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;




@end
```

This is the @property implementation that the compiler generates automatically for you (behind the scenes).
You are welcome to write the setter or getter yourself, but this would only be necessary if you needed to do something in addition to simply setting or getting the value of the property.

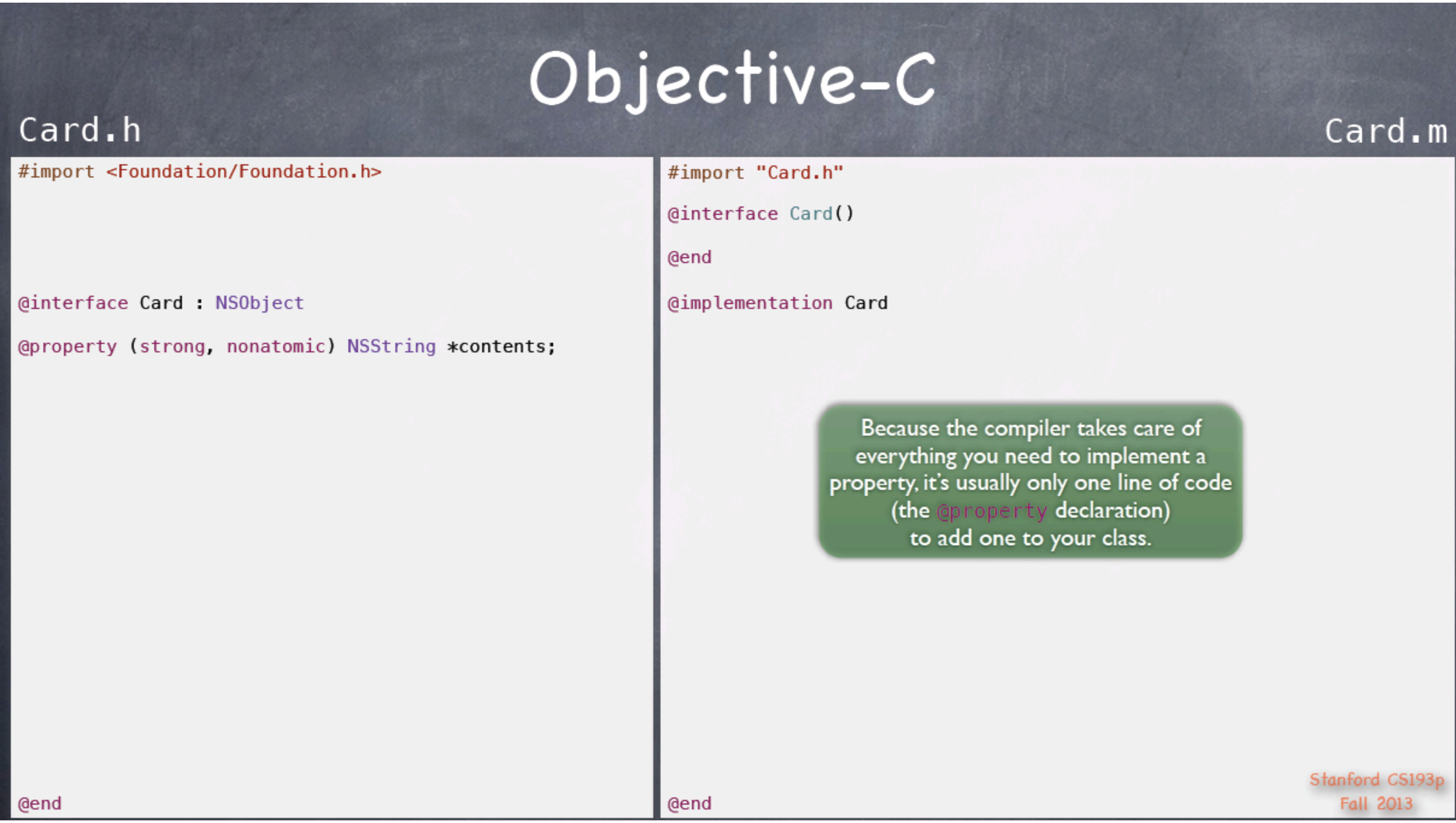
Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@synthesize contents = _contents;

- (NSString *)contents
{
    return _contents;
}

- (void)setContents:(NSString *)contents
{
    _contents = contents;
}

@end
```

This @synthesize is the line of code that actually creates the backing instance variable that is set and gotten.
Notice that by default the backing variable's name is the same as the property's name but with an underbar in front.

Stanford CS193p
Fall 2013

Hidden Getter & Setter



Objective-C

Card.h

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



@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;











@end
```

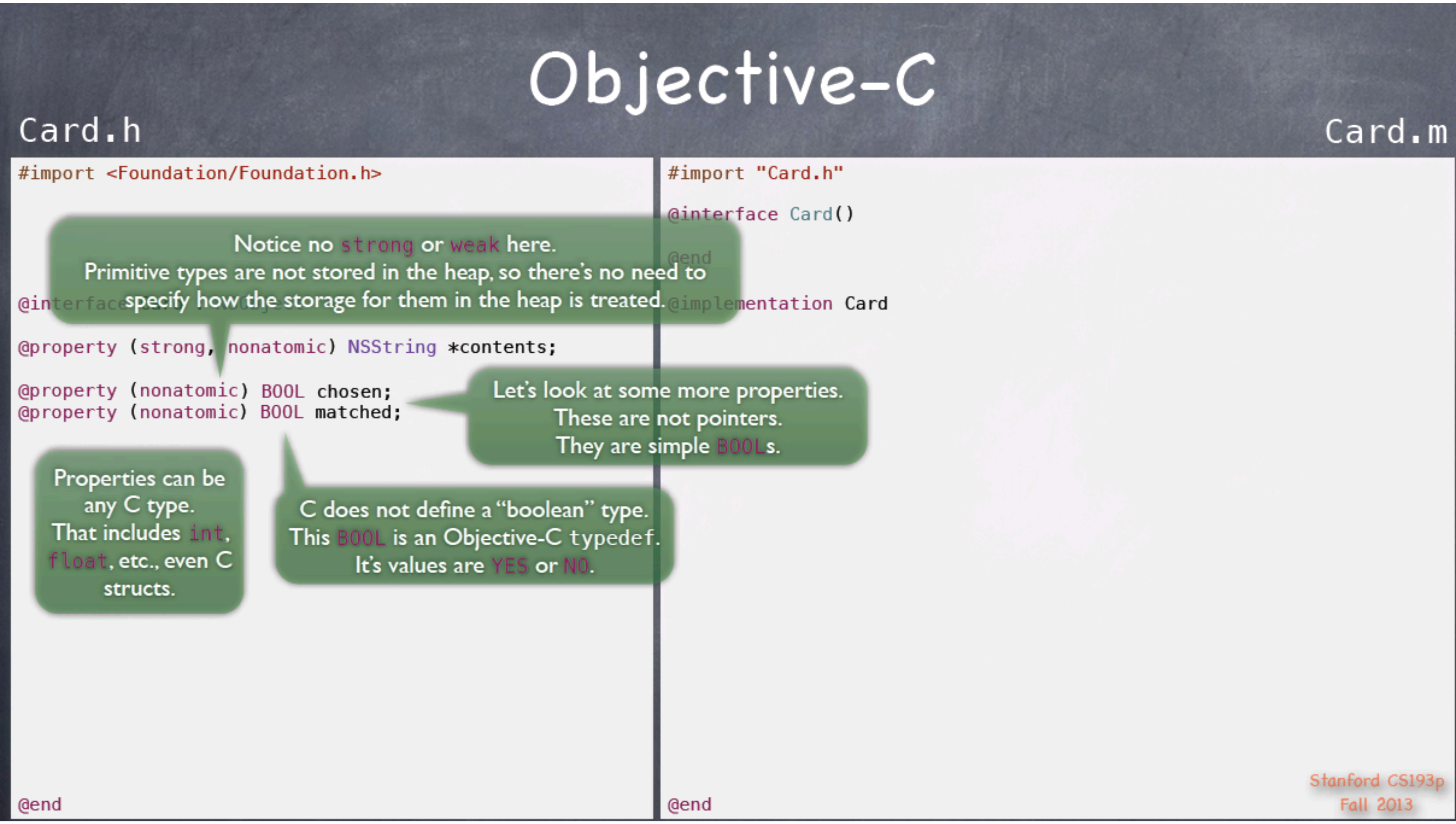
Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card











@end
```

Because the compiler takes care of everything you need to implement a property, it's usually only one line of code (the @property declaration) to add one to your class.

Stanford CS193p
Fall 2013

Primitive Properties



Objective-C

Card.h

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

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic) BOOL chosen;
@property (nonatomic) BOOL matched;

@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@end
```

Notice no strong or weak here.
Primitive types are not stored in the heap, so there's no need to specify how the storage for them in the heap is treated.

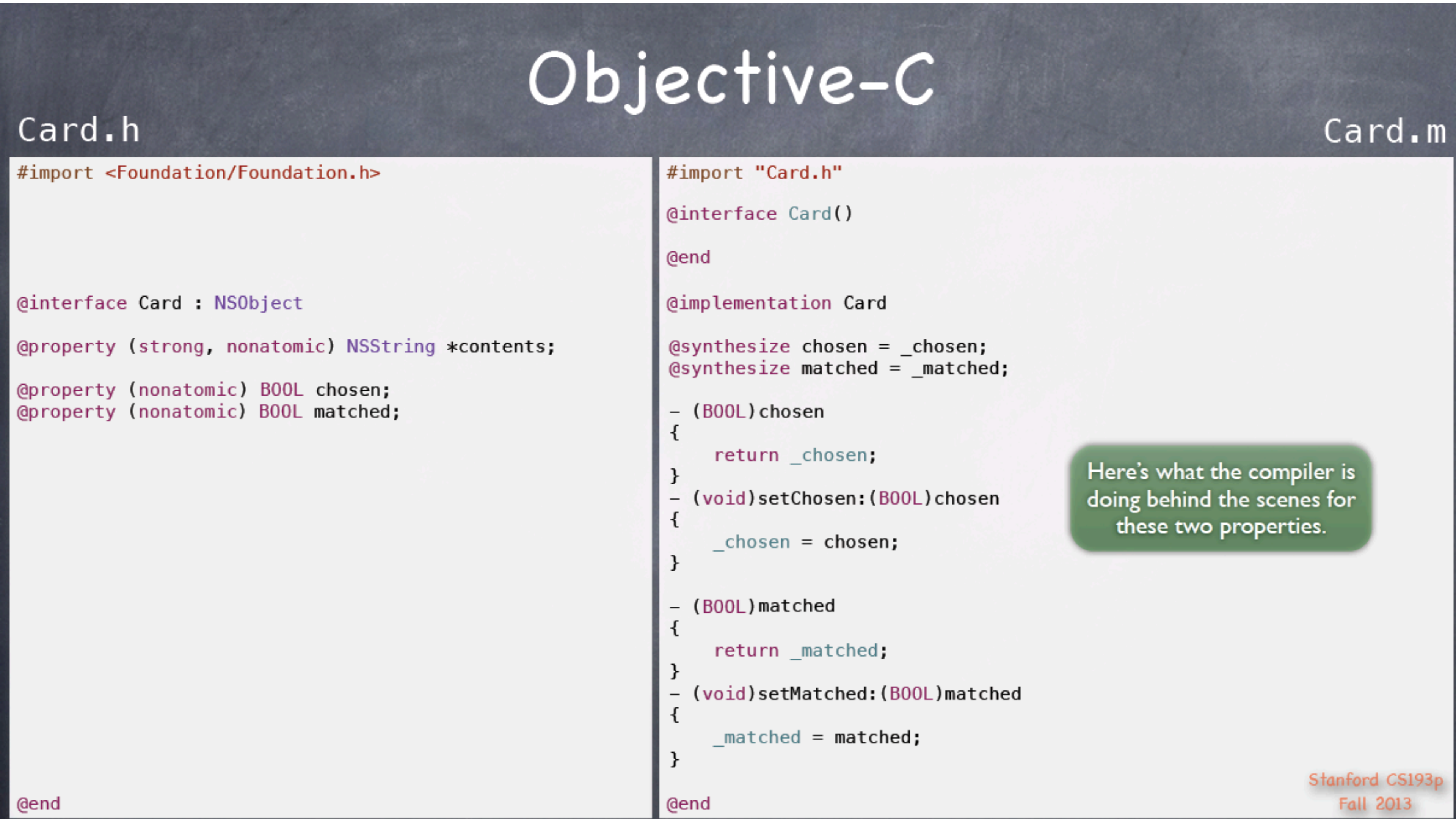
Let's look at some more properties.
These are not pointers.
They are simple BOOLs.

Properties can be any C type.
That includes int, float, etc., even C structs.

C does not define a "boolean" type.
This BOOL is an Objective-C typedef.
It's values are YES or NO.

Stanford CS193p
Fall 2013

Behind The Scenes



Objective-C

Card.h

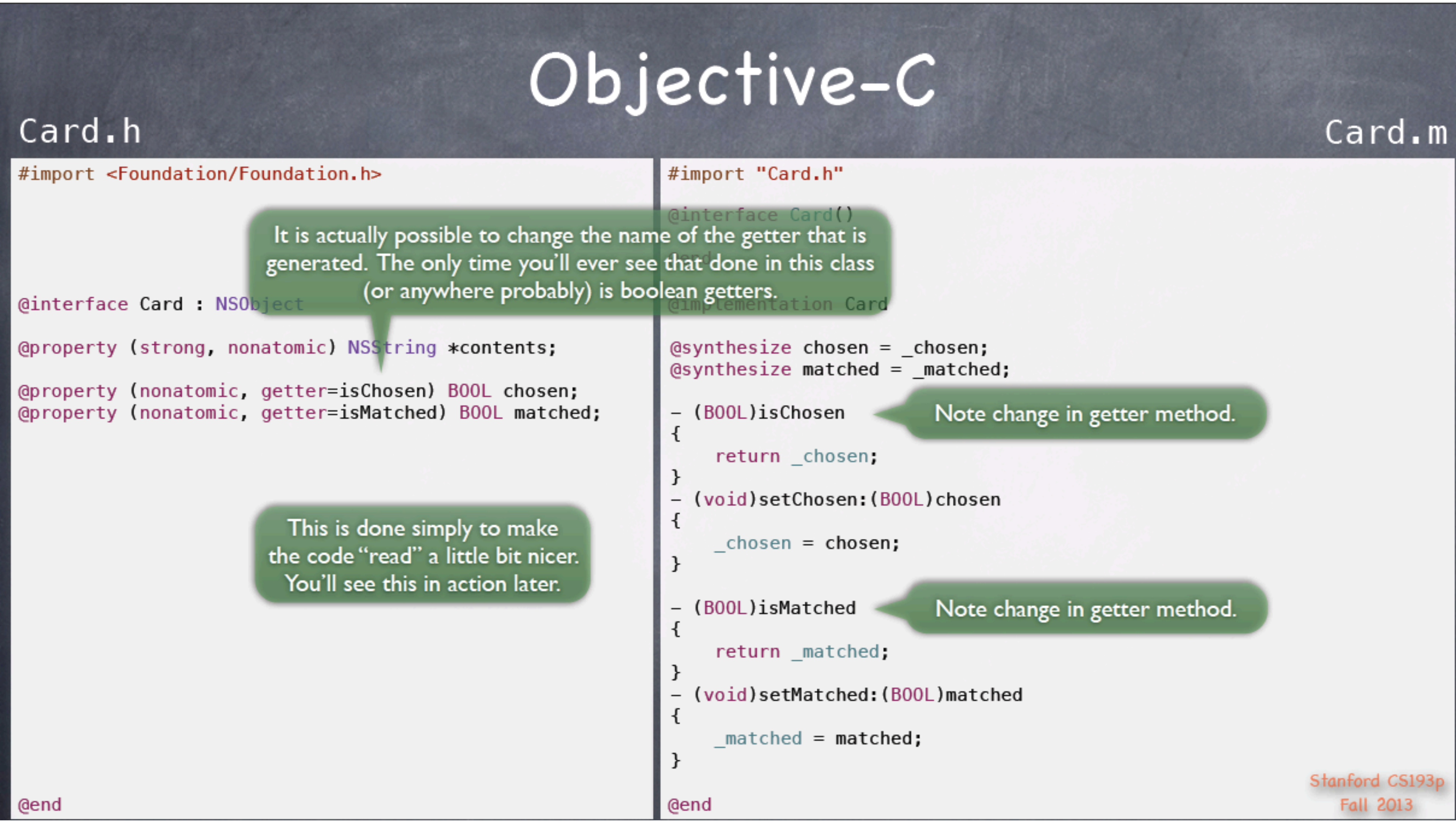
```
#import <Foundation/Foundation.h>




@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic) BOOL chosen;
@property (nonatomic) BOOL matched;











@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@synthesize chosen = _chosen;
@synthesize matched = _matched;

- (BOOL)chosen
{
    return _chosen;
}
- (void)setChosen:(BOOL)chosen
{
    _chosen = chosen;
}

- (BOOL)matched
{
    return _matched;
}
- (void)setMatched:(BOOL)matched
{
    _matched = matched;
}

@end
```

Here's what the compiler is doing behind the scenes for these two properties.

Change Getter Name



Objective-C

Card.h

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

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

@end
```

Card.m

```
#import "Card.h"

@interface Card()
@end

@implementation Card

@synthesize chosen = _chosen;
@synthesize matched = _matched;

- (BOOL)isChosen
{
    return _chosen;
}
- (void)setChosen:(BOOL)chosen
{
    _chosen = chosen;
}

- (BOOL)isMatched
{
    return _matched;
}
- (void)setMatched:(BOOL)matched
{
    _matched = matched;
}

@end
```

It is actually possible to change the name of the getter that is generated. The only time you'll ever see that done in this class (or anywhere probably) is boolean getters.

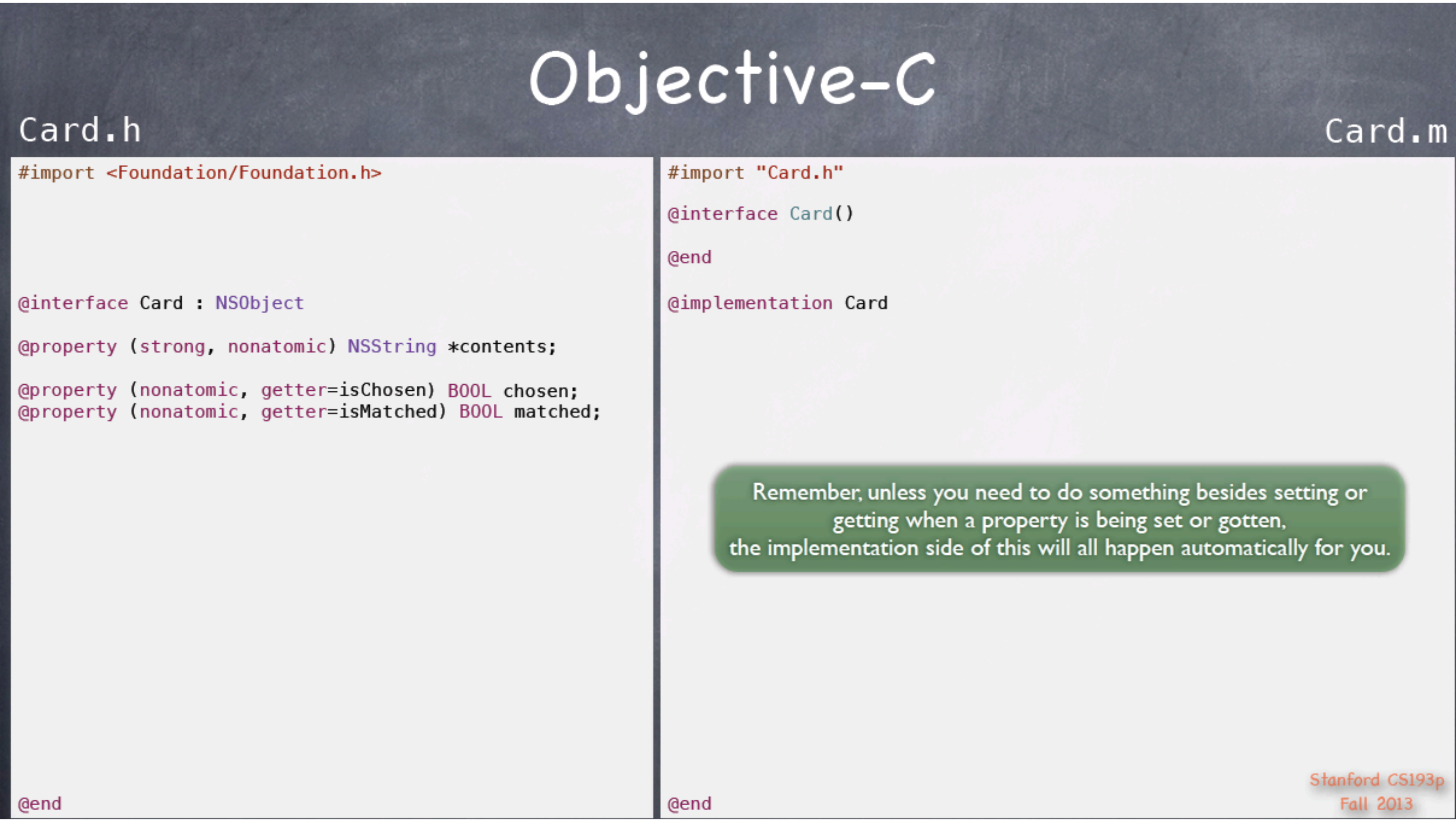
This is done simply to make the code "read" a little bit nicer. You'll see this in action later.

Note change in getter method.

Note change in getter method.

Stanford CS193p
Fall 2013

Getter & Setter
Still Hidden



Objective-C

Card.h

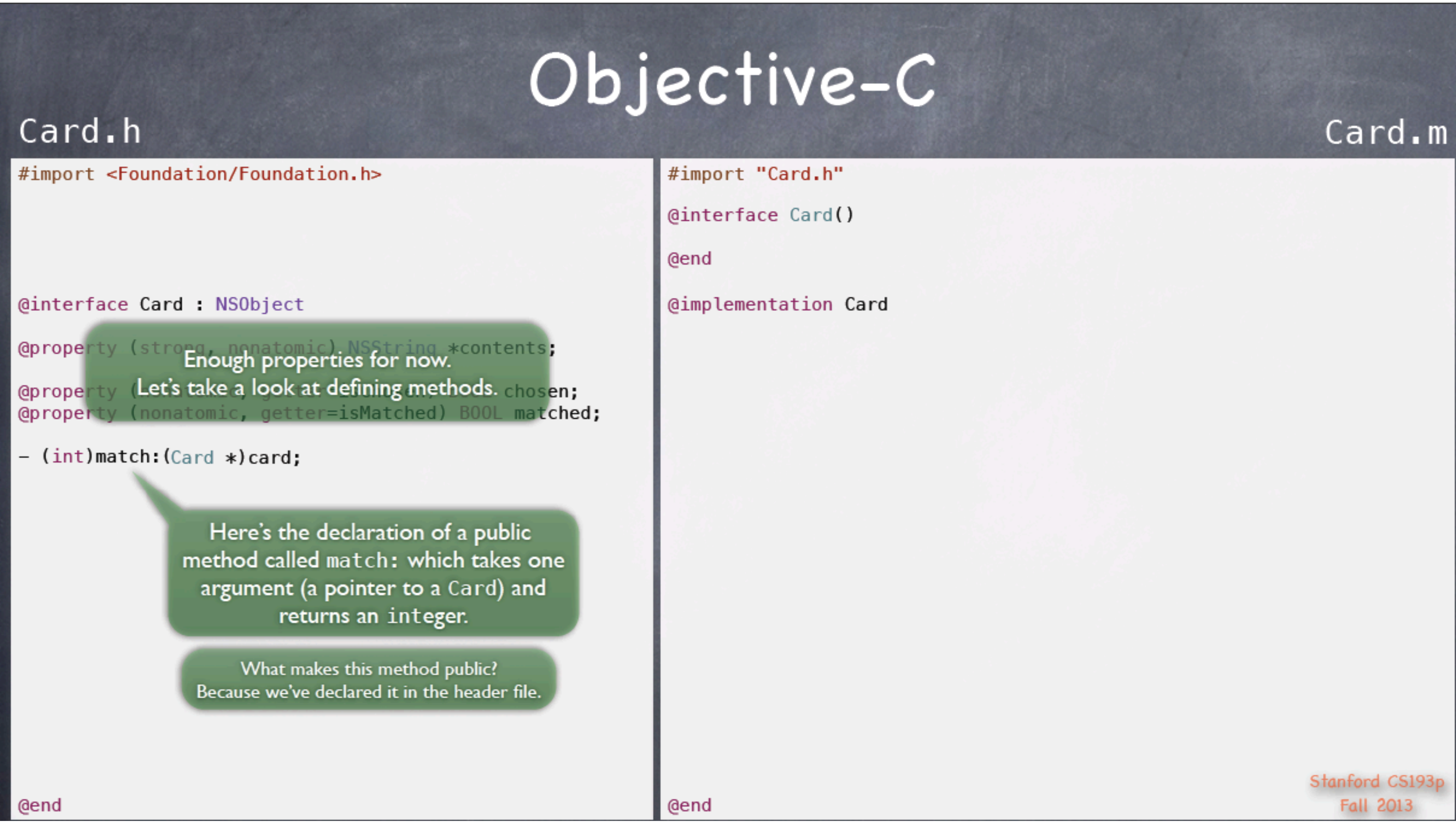
```
#import <Foundation/Foundation.h>



@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;






@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card






@end
```

Remember, unless you need to do something besides setting or getting when a property is being set or gotten, the implementation side of this will all happen automatically for you.

Public Method Declaration



Objective-C

Card.h

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

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;

@end
```

Enough properties for now.
Let's take a look at defining methods.

Here's the declaration of a public method called match: which takes one argument (a pointer to a Card) and returns an integer.

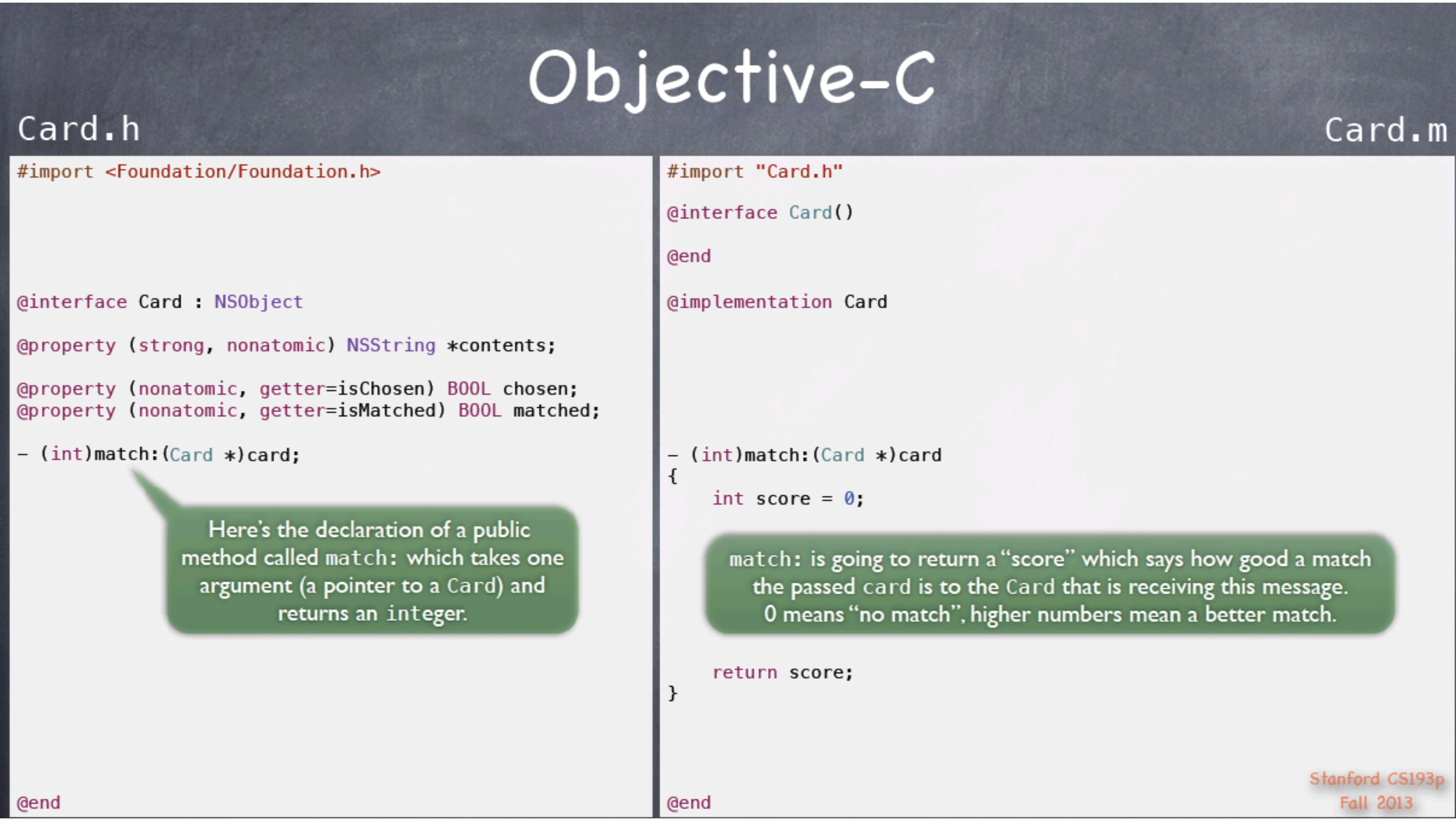
What makes this method public?
Because we've declared it in the header file.

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@end
```

Stanford CS193p
Fall 2013

Public Method Implementation



Objective-C

Card.h

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



@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;









@end
```

Here's the declaration of a public method called match: which takes one argument (a pointer to a Card) and returns an integer.

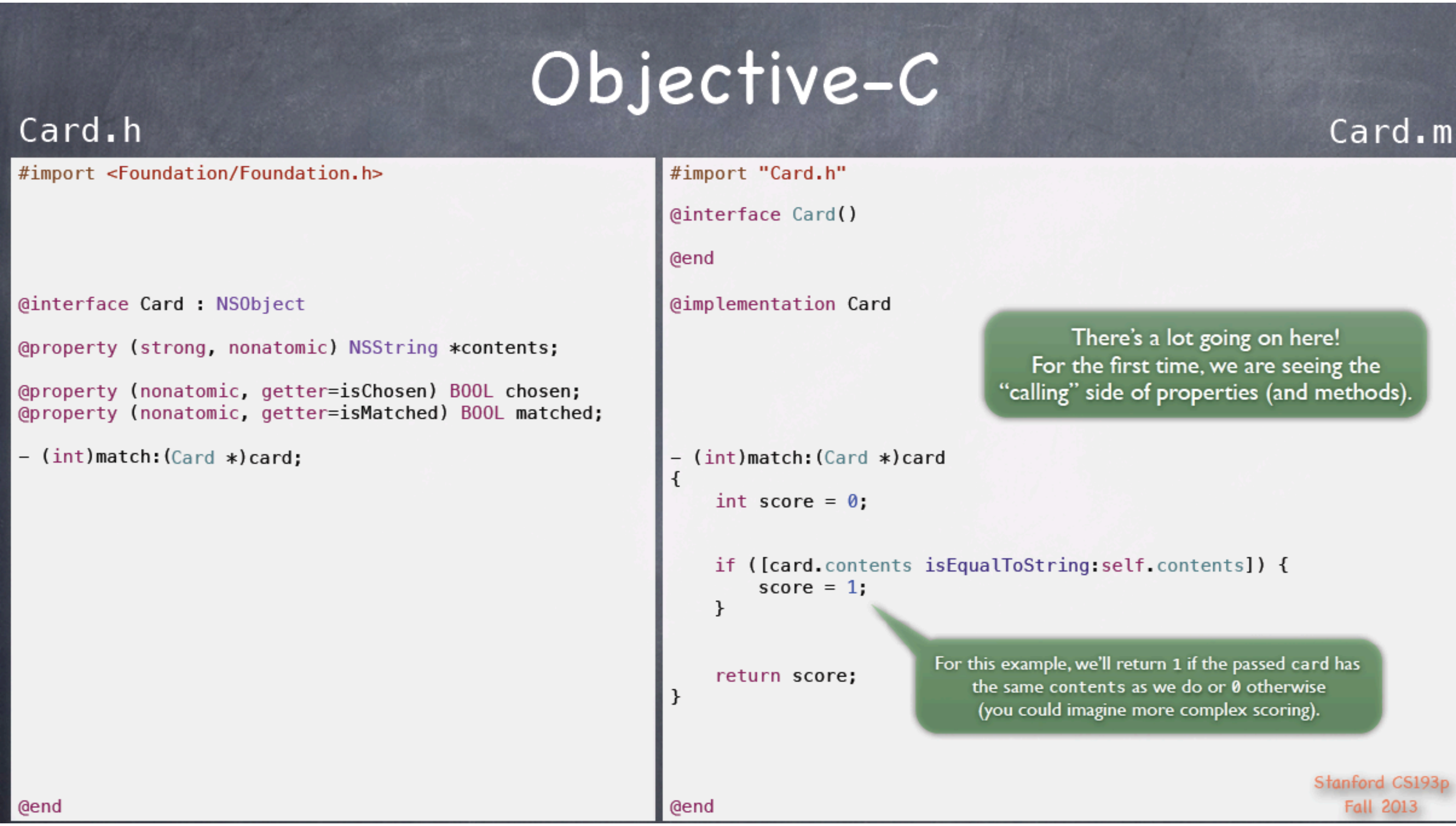
Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card





- (int)match:(Card *)card
{
    int score = 0;



    return score;
}



@end
```

match: is going to return a "score" which says how good a match the passed card is to the Card that is receiving this message. 0 means "no match", higher numbers mean a better match.

Stanford CS193p
Fall 2013

Call A Method With [] or .



Objective-C

Card.h

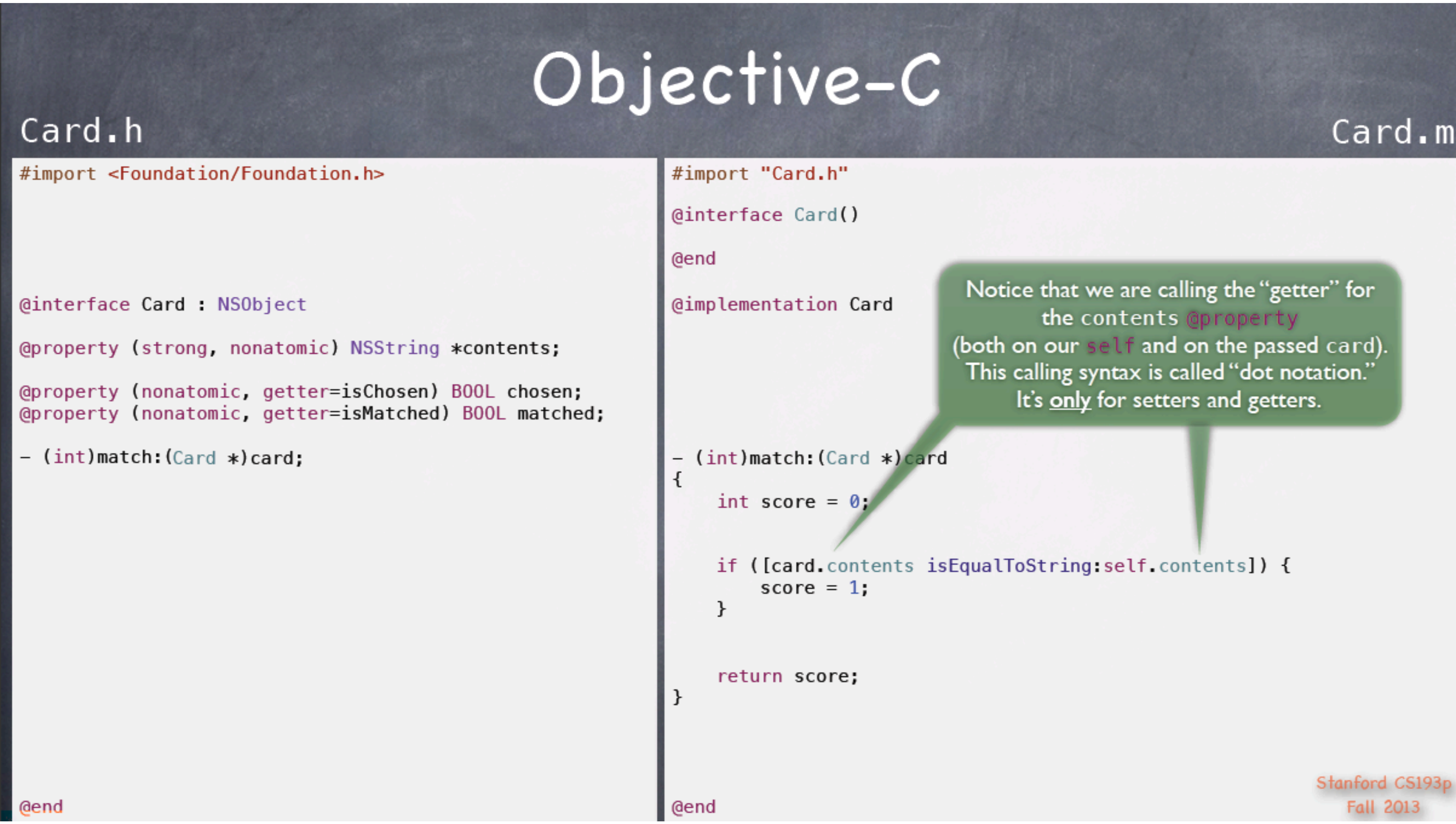
```
#import <Foundation/Foundation.h>




@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;















@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card






- (int)match:(Card *)card
{
    int score = 0;


    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }


    return score;
}





@end
```

There's a lot going on here!
For the first time, we are seeing the "calling" side of properties (and methods).
For this example, we'll return 1 if the passed card has the same contents as we do or 0 otherwise (you could imagine more complex scoring).
Stanford CS193p Fall 2013

"." Notation For Getters and Setters Only



Objective-C

Card.h

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



@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;






@end
```

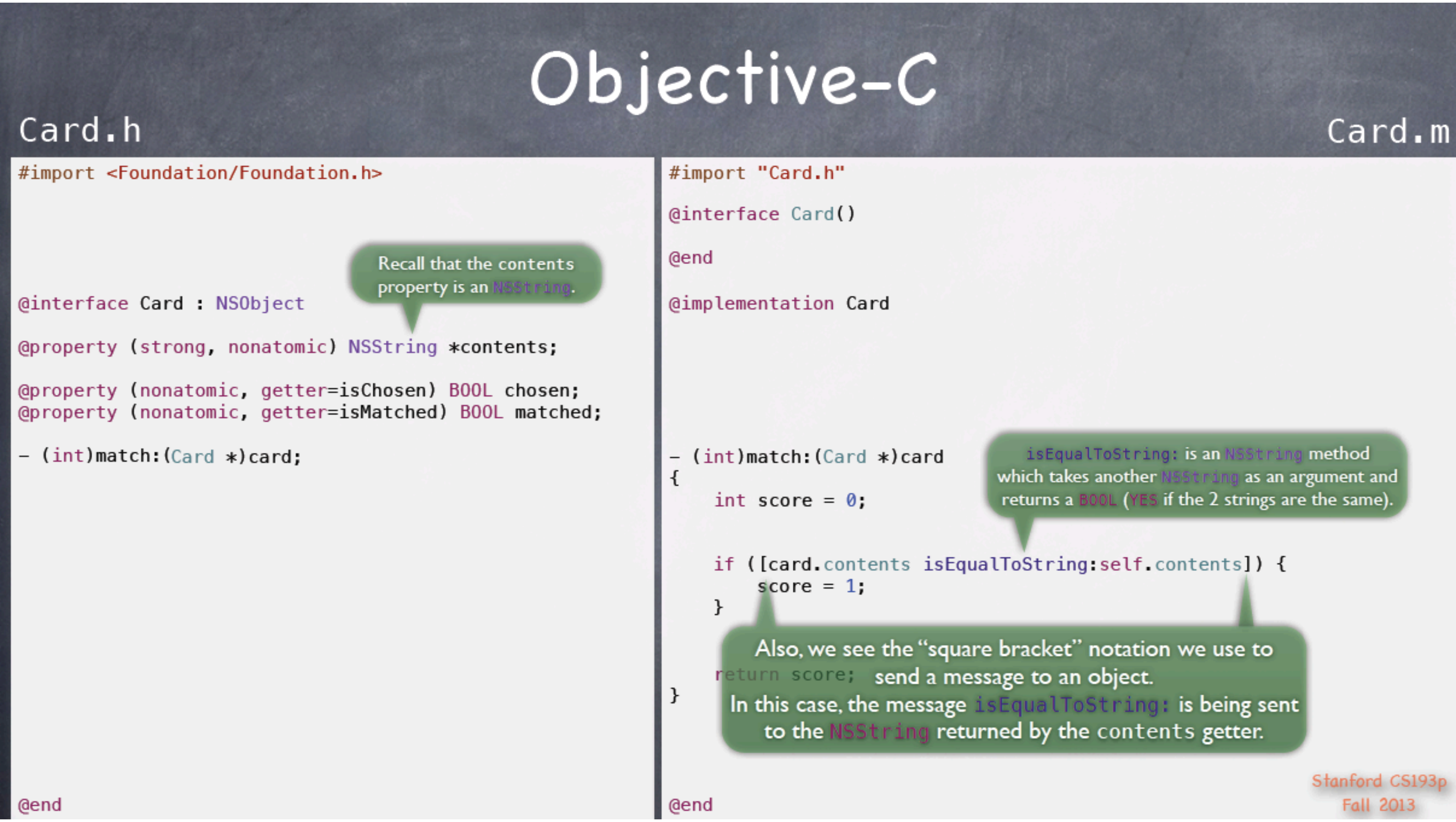
Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card




- (int)match:(Card *)card
{
    int score = 0;

    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }

    return score;
}



@end
```

Notice that we are calling the "getter" for the contents @property (both on our self and on the passed card). This calling syntax is called "dot notation." It's only for setters and getters.

Stanford CS193p
Fall 2013

"[]" Notation For Everything Else



Objective-C

Card.h

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



@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;









@end
```

Recall that the contents property is an NSString.

Card.m

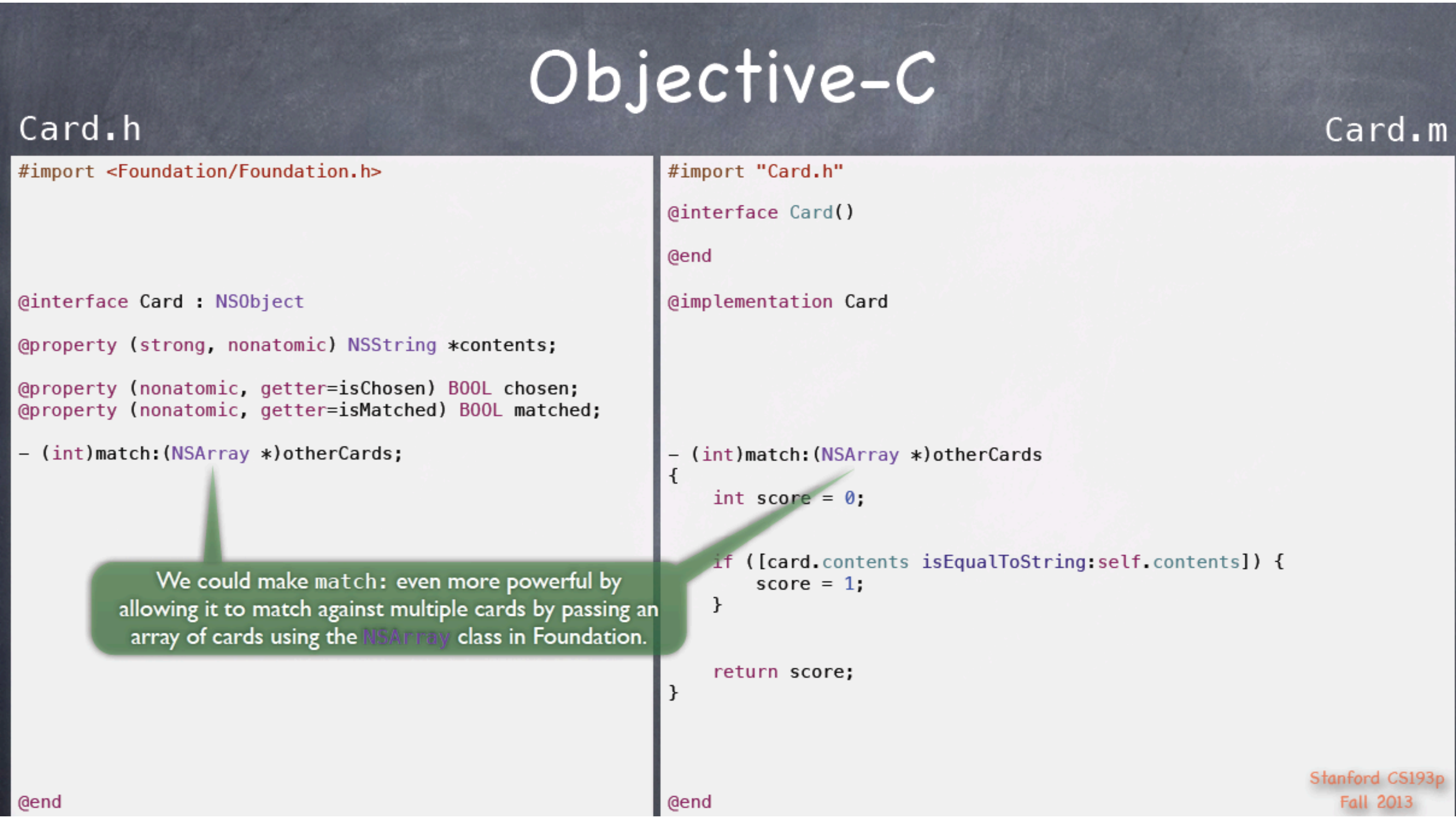
```
#import "Card.h"

@interface Card()

@end

@implementation Card






- (int)match:(Card *)card
{
    int score = 0;


    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }

    return score;
}



@end
```

isEqualToString: is an NSString method which takes another NSString as an argument and returns a BOOL (YES if the 2 strings are the same).

Also, we see the "square bracket" notation we use to send a message to an object.
In this case, the message isEqualToString: is being sent to the NSString returned by the contents getter.

Stanford CS193p
Fall 2013

Match Multiple Cards Declaration



Objective-C

Card.h

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




@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(NSArray *)otherCards;










@end
```

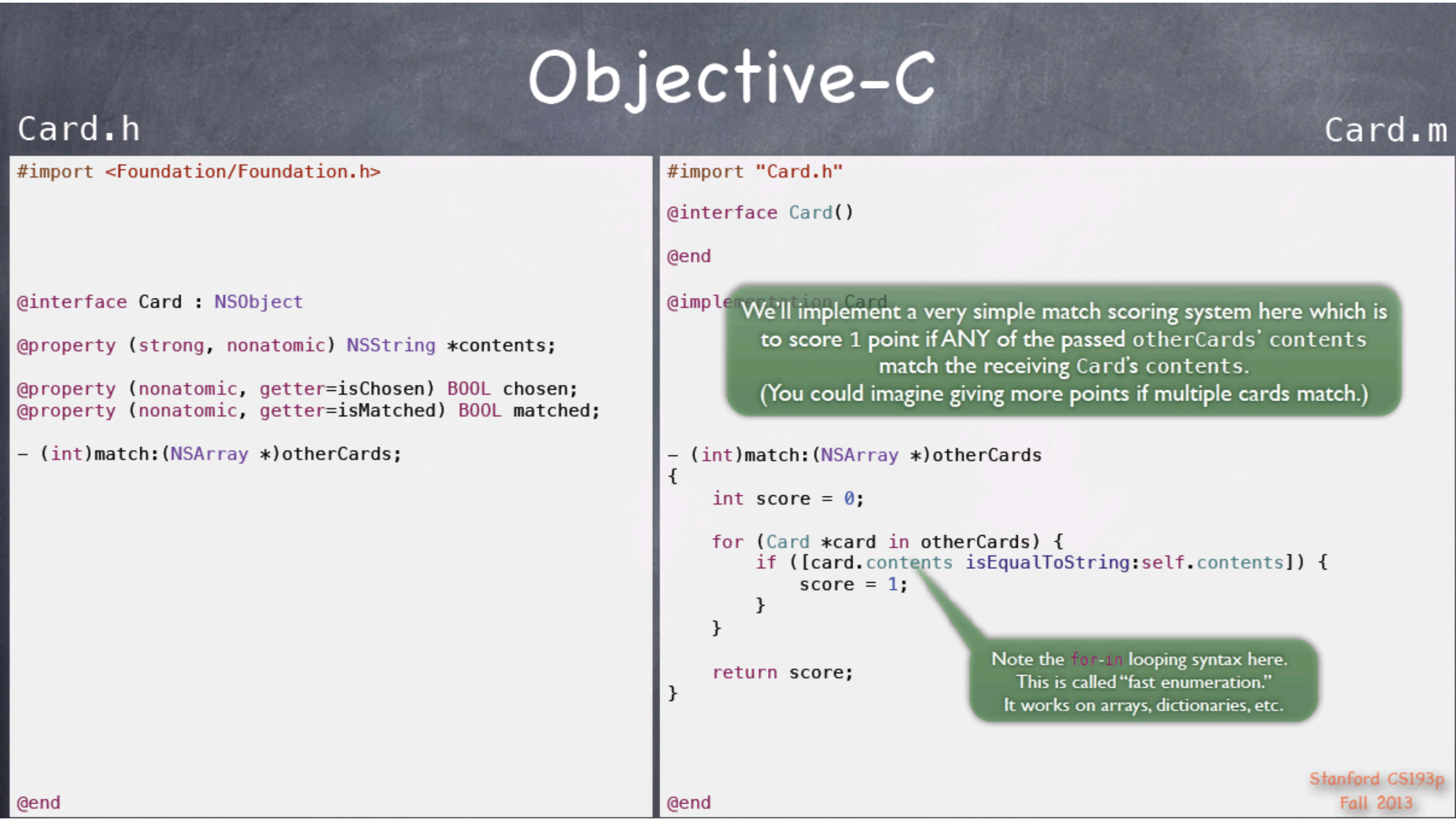
Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card






- (int)match:(NSArray *)otherCards
{
    int score = 0;


    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }


    return score;
}




@end
```

We could make match: even more powerful by allowing it to match against multiple cards by passing an array of cards using the NSArray class in Foundation.

Stanford CS193p
Fall 2013

Match Multiple Cards Implementation



Objective-C

Card.h

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




@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(NSArray *)otherCards;









@end
```

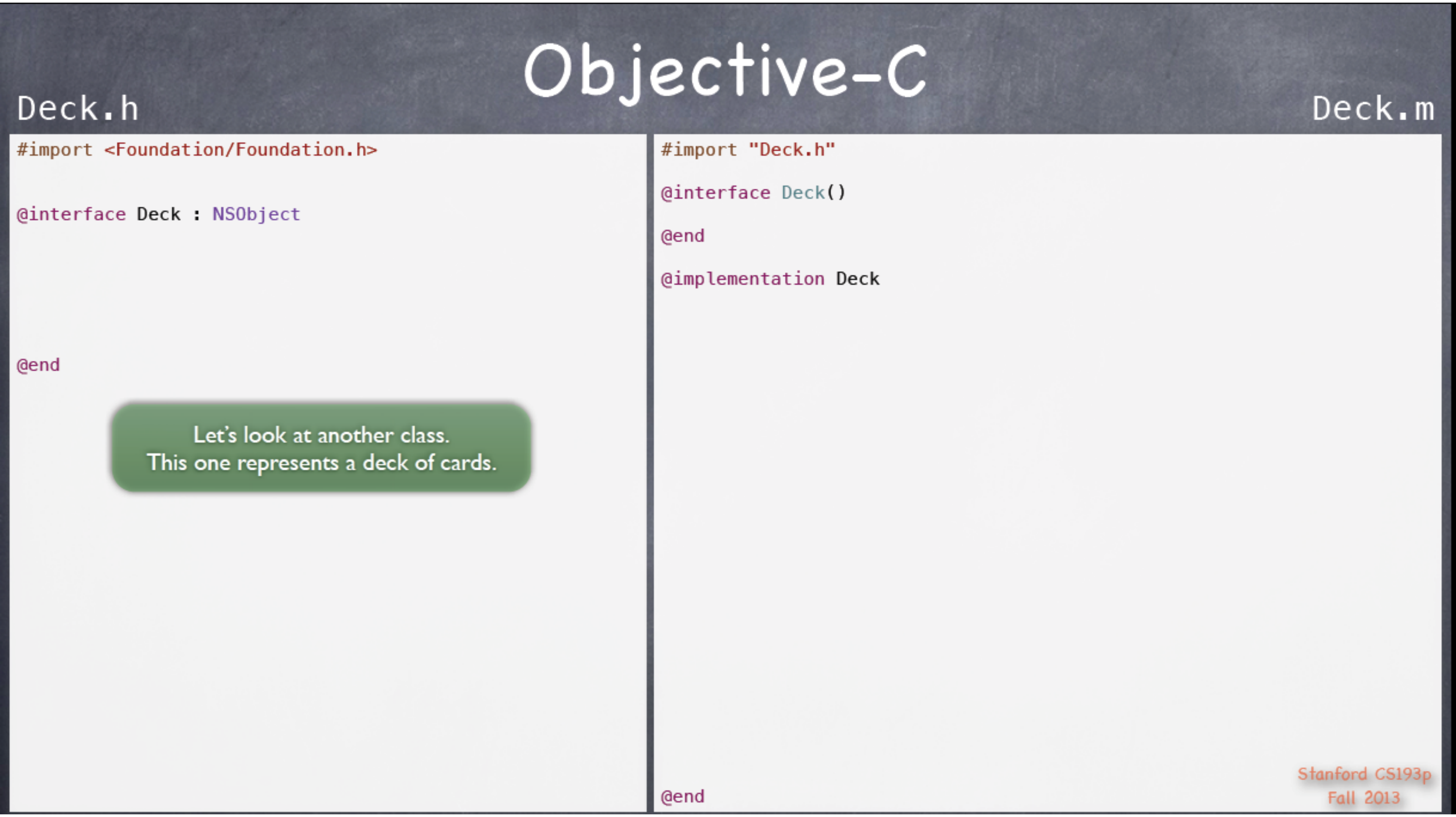
Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card





- (int)match:(NSArray *)otherCards
{
    int score = 0;

    for (Card *card in otherCards) {
        if ([card.contents isEqualToString:self.contents]) {
            score = 1;
        }
    }

    return score;
}




@end
```

We'll implement a very simple match scoring system here which is to score 1 point if ANY of the passed otherCards' contents match the receiving Card's contents.
(You could imagine giving more points if multiple cards match.)

Note the for-in looping syntax here.
This is called "fast enumeration."
It works on arrays, dictionaries, etc.

Deck Class



Objective-C

Deck.h

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


@interface Deck : NSObject




@end
```

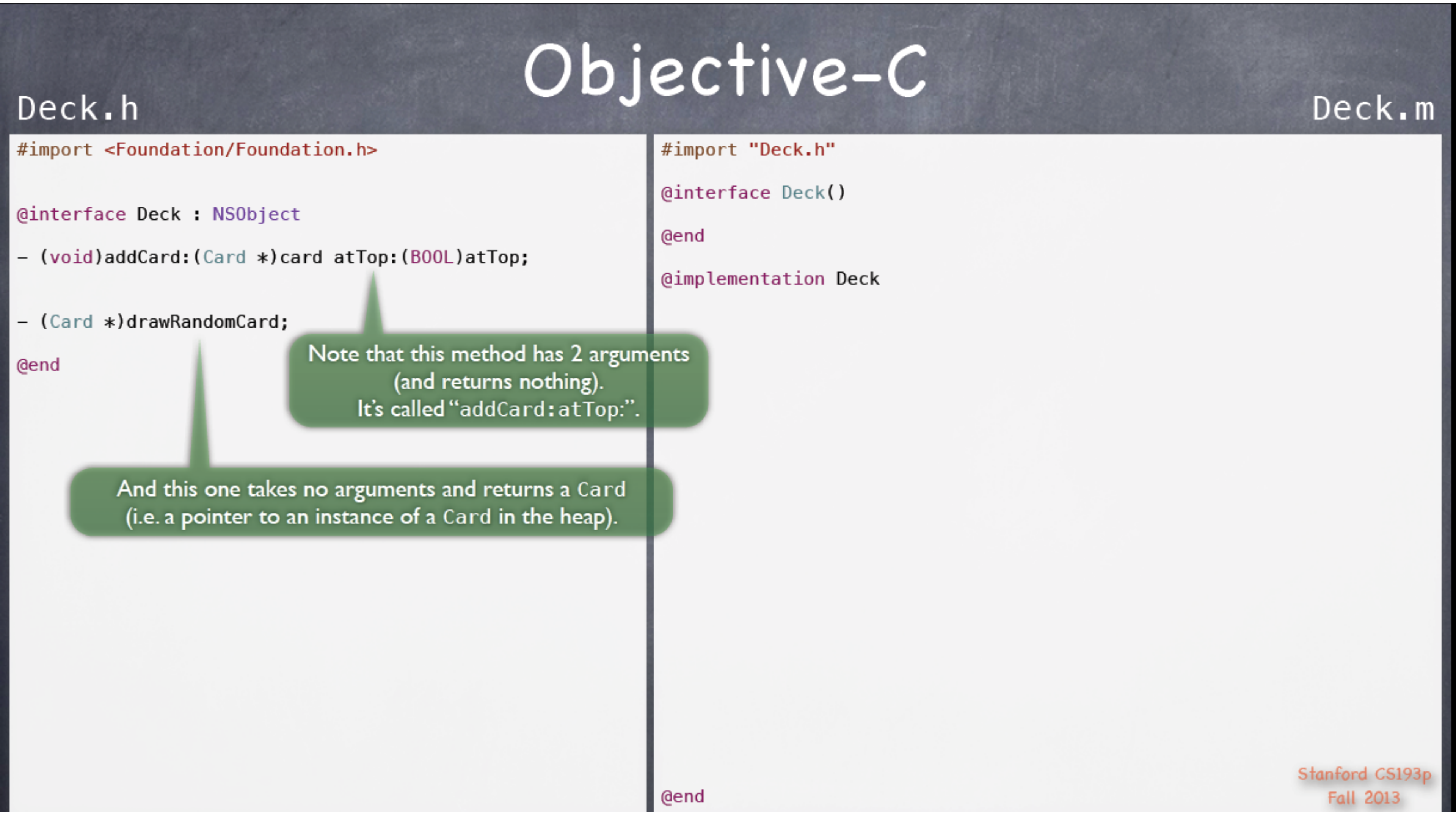
Let's look at another class.
This one represents a deck of cards.

Deck.m

```
#import "Deck.h"

@interface Deck()

@end

@implementation Deck









@end
```

Stanford CS193p
Fall 2013

Methods With Multiple Arguments



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;

- (Card *)drawRandomCard;

@end
```

Note that this method has 2 arguments (and returns nothing).
It's called "addCard:atTop:".

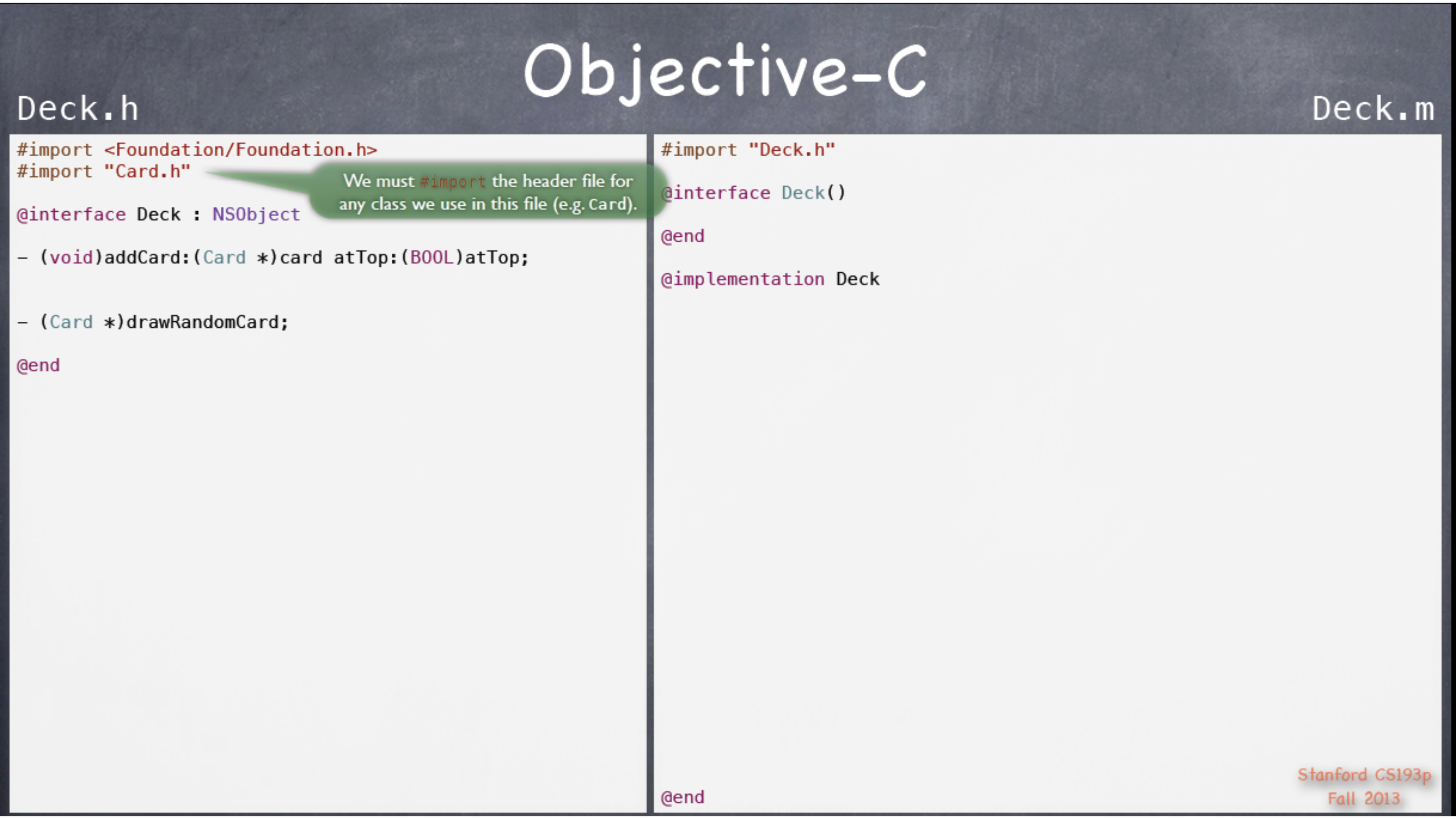
And this one takes no arguments and returns a Card (i.e. a pointer to an instance of a Card in the heap).

Deck.m

```
#import "Deck.h"

@interface Deck()

@end

@implementation Deck




@end
```

Stanford CS193p
Fall 2013

Must Import Card.h



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;


- (Card *)drawRandomCard;

@end
```

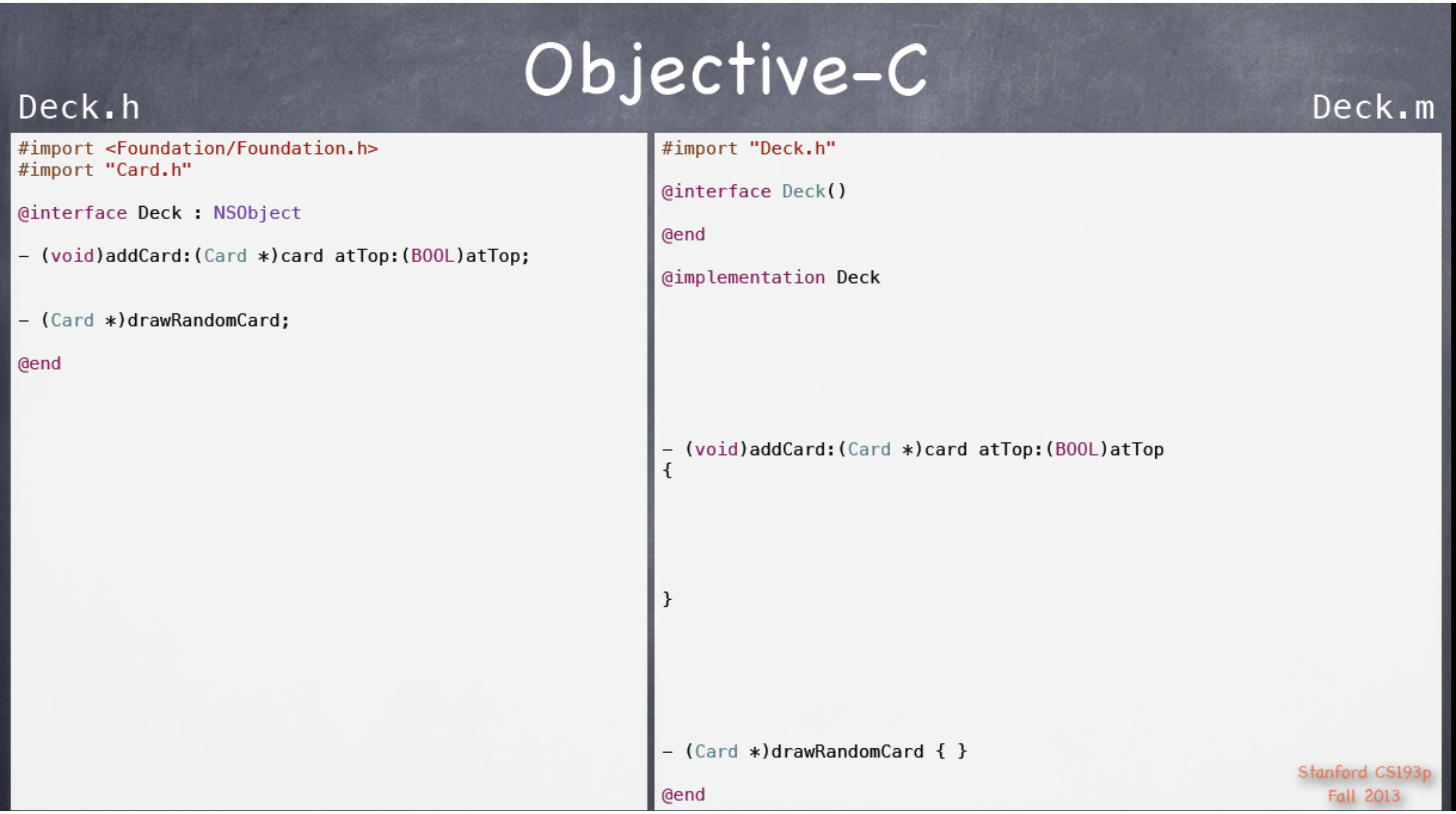
We must #import the header file for any class we use in this file (e.g. Card).

Deck.m

```
#import "Deck.h"

@interface Deck()

@end

@implementation Deck

@end
```

Stanford CS193p
Fall 2013

Define Methods in Deck.m



Objective-C

Deck.h

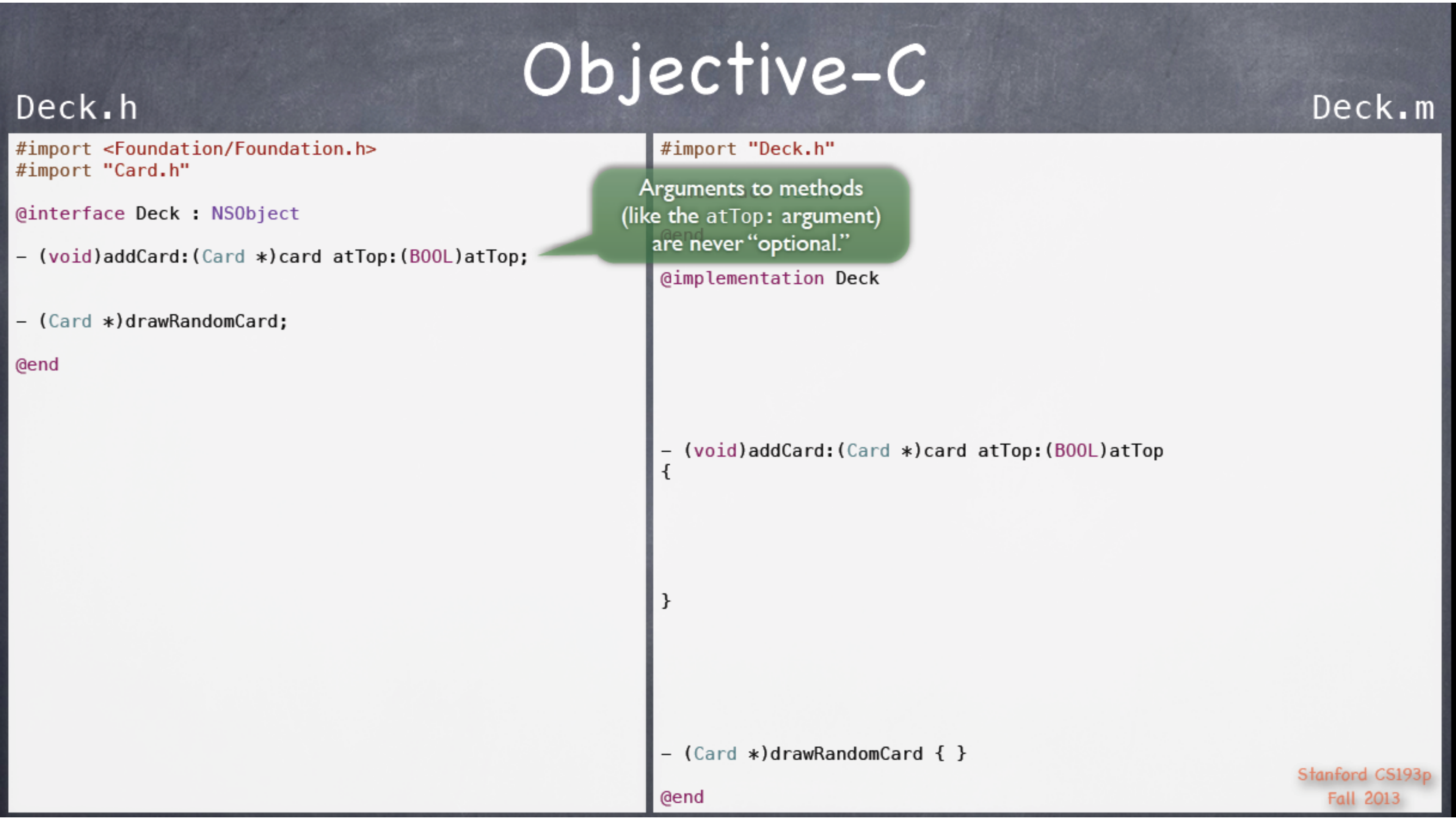
```
#import <Foundation/Foundation.h>
#import "Card.h"

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;


- (Card *)drawRandomCard;

@end
```

Deck.m

```
#import "Deck.h"

@interface Deck()

@end

@implementation Deck







- (void)addCard:(Card *)card atTop:(BOOL)atTop
{






}






- (Card *)drawRandomCard { }

@end
```

Stanford CS193p
Fall 2013

No Optional Arguments



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;


- (Card *)drawRandomCard;

@end
```

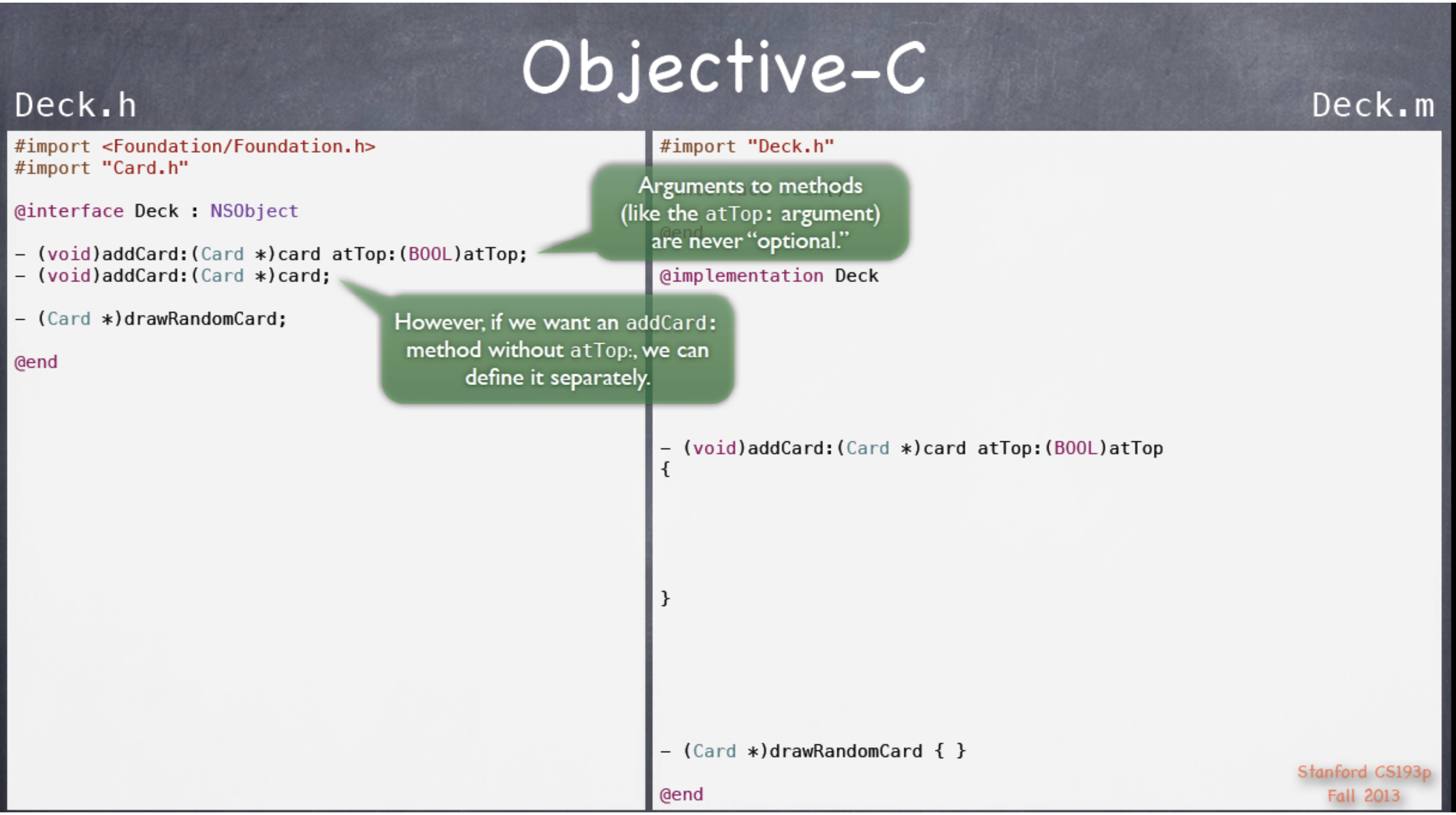
Arguments to methods (like the `atTop:` argument) are never "optional."

Deck.m

```
#import "Deck.h"



@implementation Deck







- (void)addCard:(Card *)card atTop:(BOOL)atTop
{






}








- (Card *)drawRandomCard { }

@end
```

Stanford CS193p
Fall 2013

Can Define A New addCard Method With One Argument



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

Deck.m

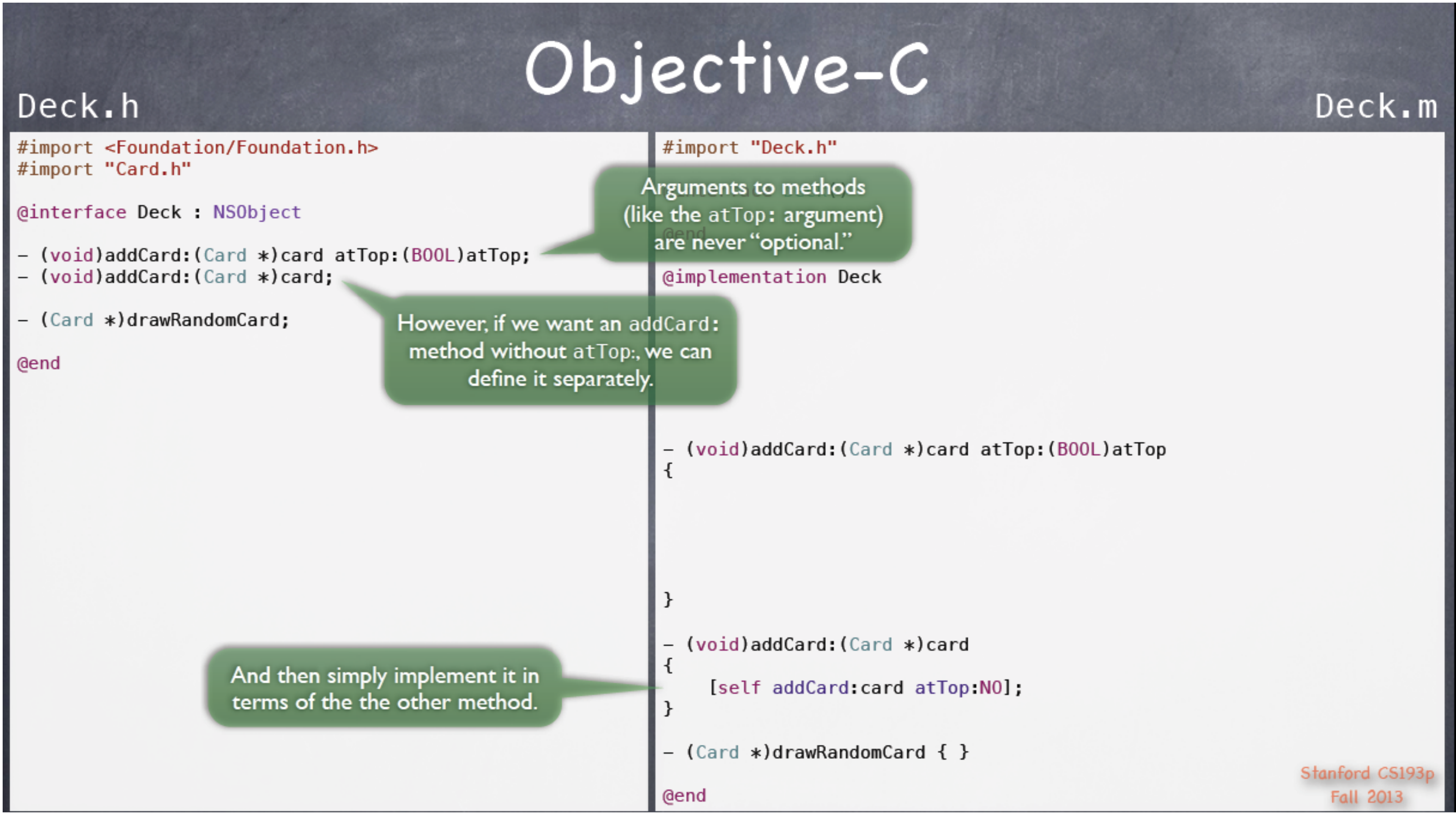
```
#import "Deck.h"

@implementation Deck

- (void)addCard:(Card *)card atTop:(BOOL)atTop
{

}

- (Card *)drawRandomCard { }

@end
```

Arguments to methods (like the atTop: argument) are never "optional."

However, if we want an addCard: method without atTop:, we can define it separately.

Stanford CS193p
Fall 2013

Implement New AddCard Method



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

Deck.m

```
#import "Deck.h"

@implementation Deck

- (void)addCard:(Card *)card atTop:(BOOL)atTop
{

}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

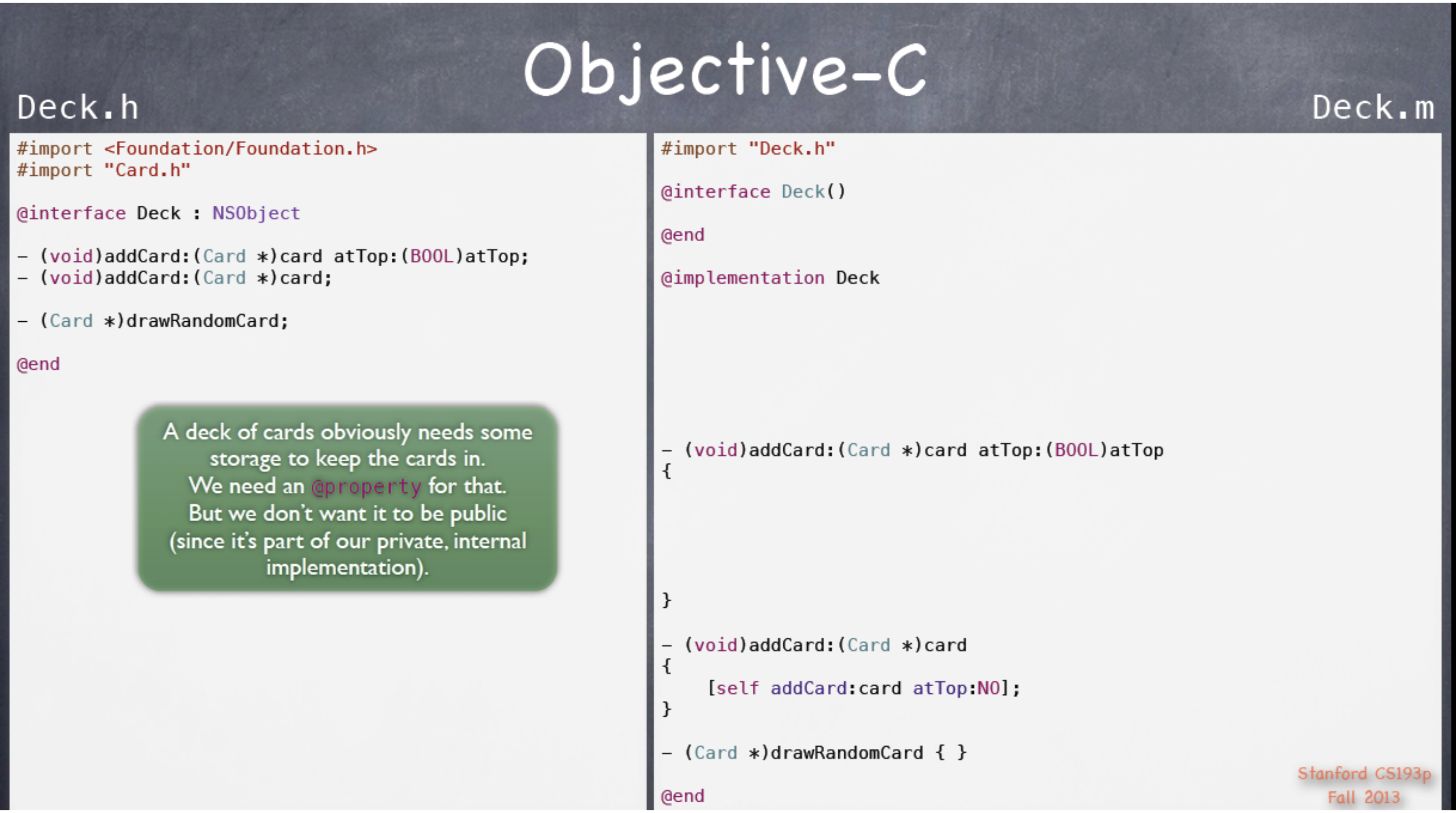
Arguments to methods (like the atTop: argument) are never "optional."

However, if we want an addCard: method without atTop:, we can define it separately.

And then simply implement it in terms of the the other method.

Stanford CS193p Fall 2013

Need Storage To Hold Cards



Objective-C

Deck.h

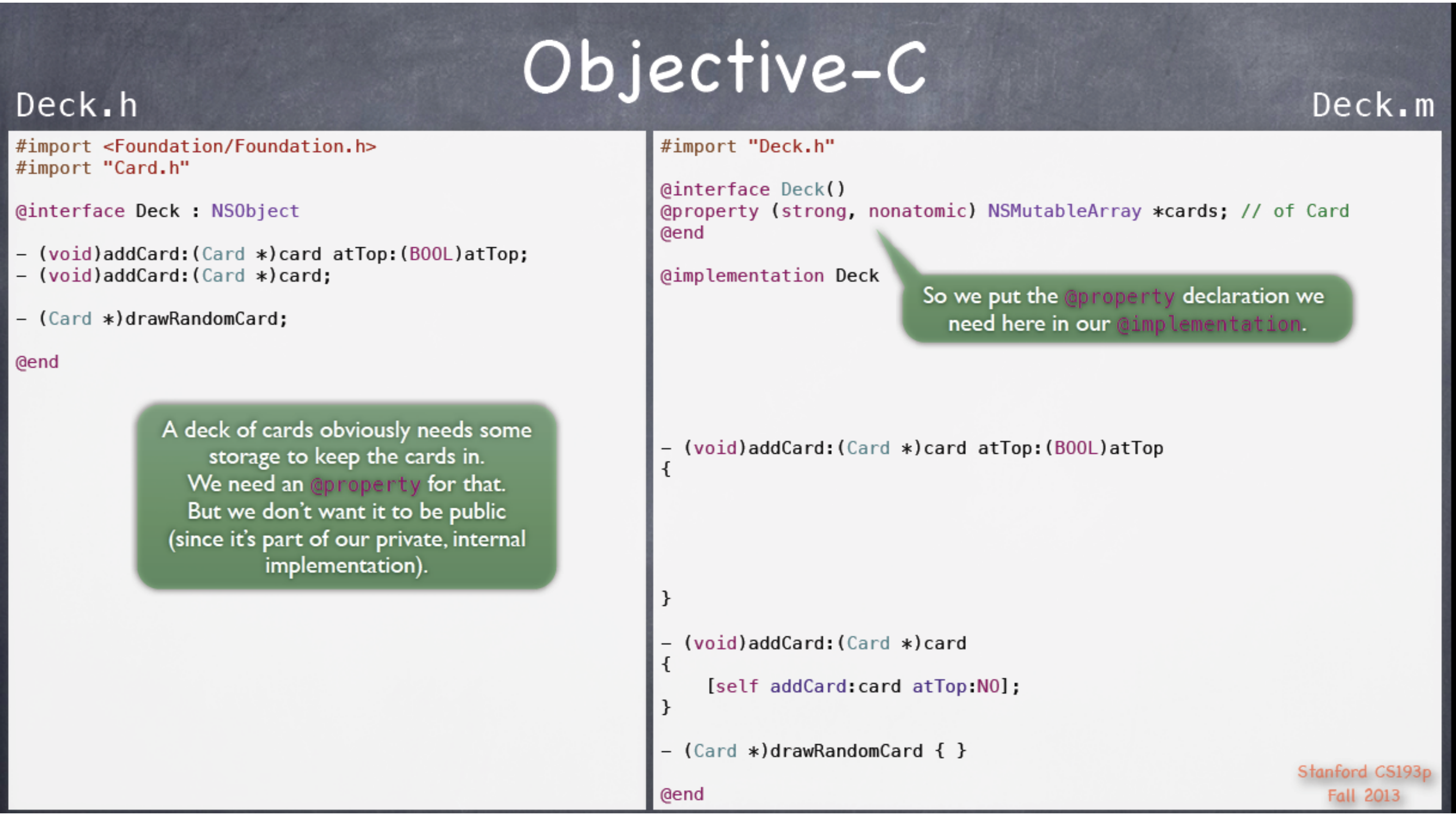
```
#import <Foundation/Foundation.h>
#import "Card.h"

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

A deck of cards obviously needs some storage to keep the cards in.
We need an @property for that.
But we don't want it to be public (since it's part of our private, internal implementation).

Deck.m

```
#import "Deck.h"

@interface Deck()

@end

@implementation Deck








- (void)addCard:(Card *)card atTop:(BOOL)atTop
{






}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

Define The Cards Array As Private Property



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

A deck of cards obviously needs some storage to keep the cards in.
We need an @property for that.
But we don't want it to be public (since it's part of our private, internal implementation).

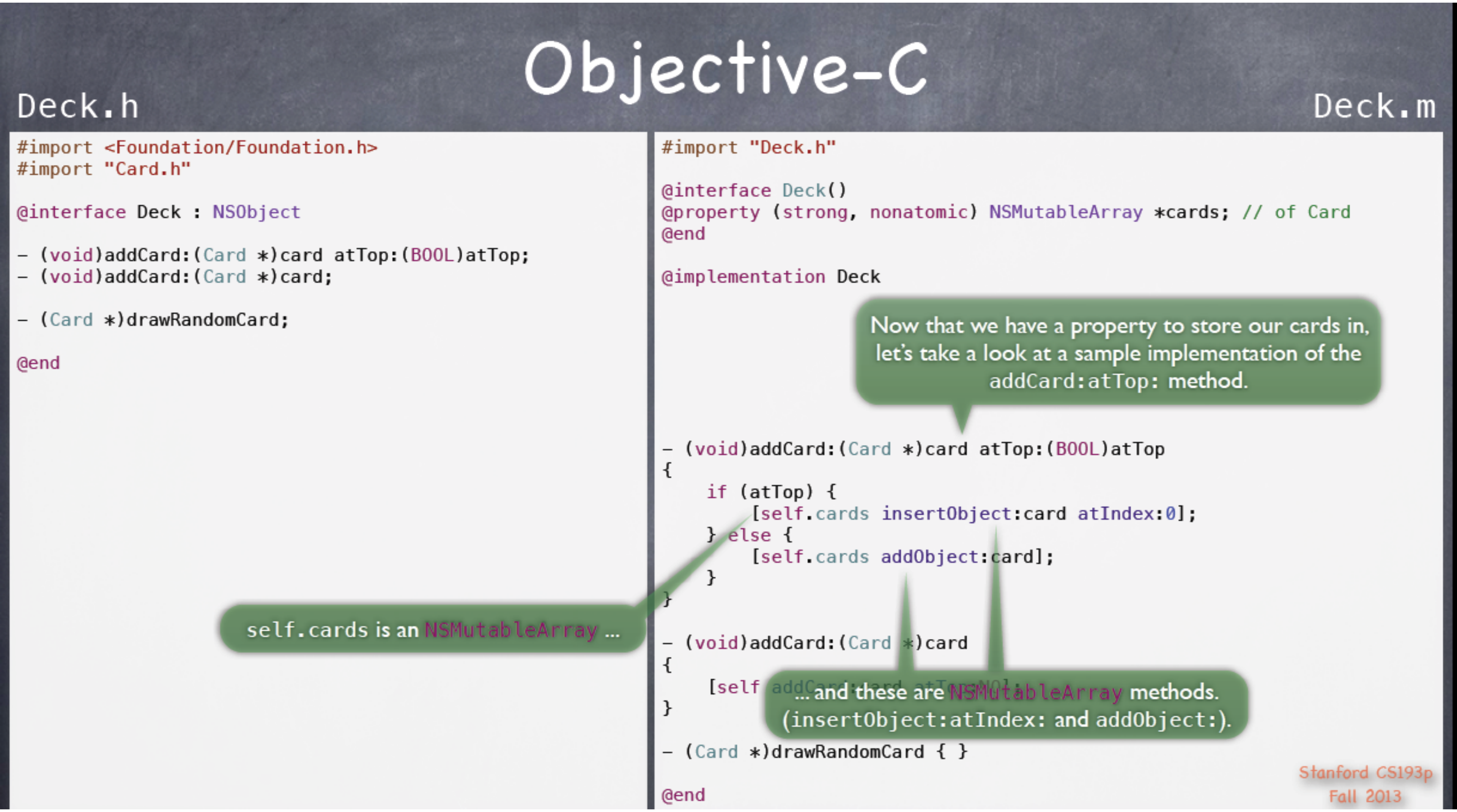
Deck.m

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck









- (void)addCard:(Card *)card atTop:(BOOL)atTop
{






}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

So we put the @property declaration we need here in our @implementation.

Stanford CS193p
Fall 2013

Implement addCard:atTop:



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

Deck.m

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (void)addCard:(Card *)card atTop:(BOOL)atTop
{
    if (atTop) {
        [self.cards insertObject:card atIndex:0];
    } else {
        [self.cards addObject:card];
    }
}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

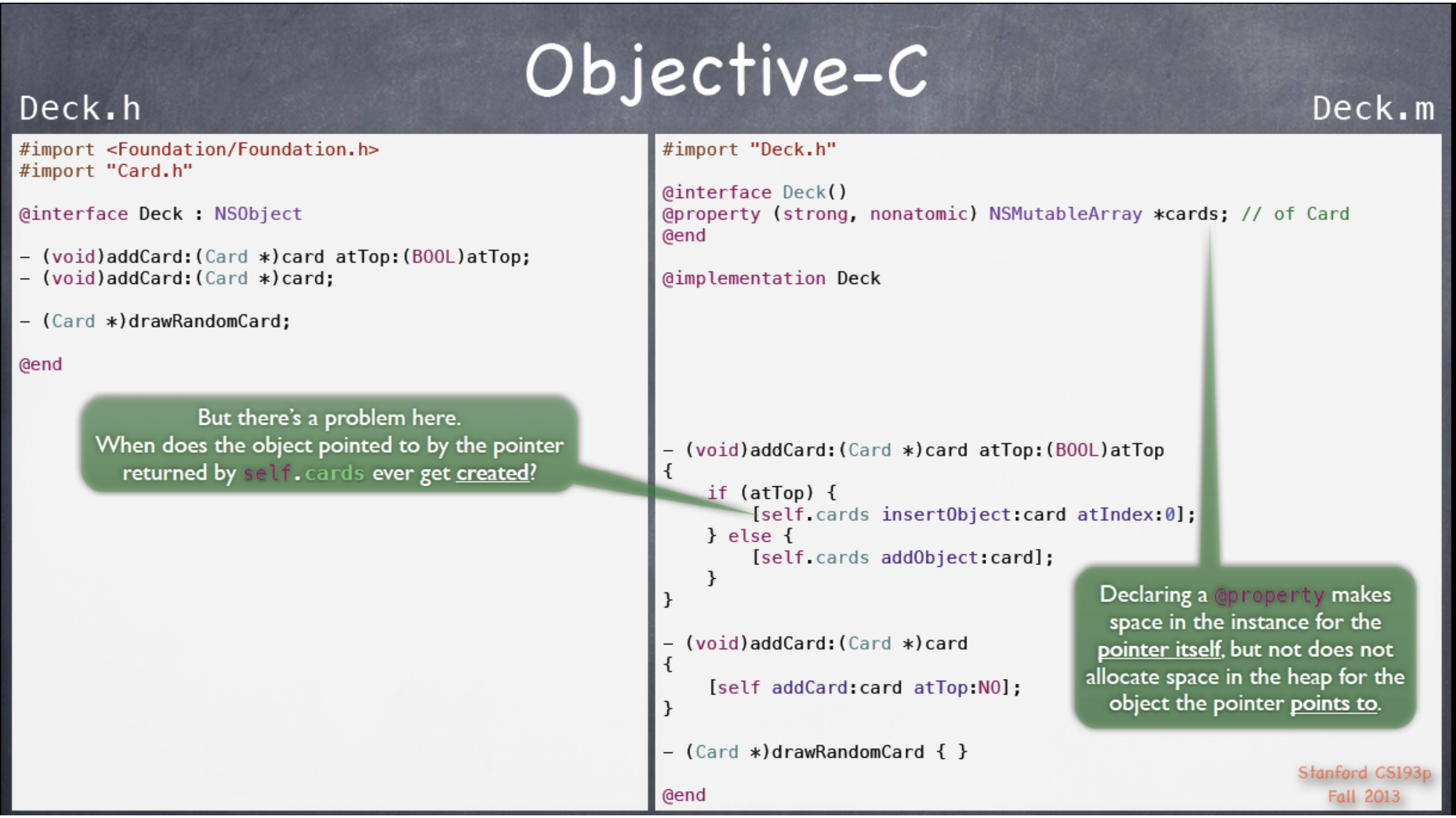
Now that we have a property to store our cards in, let's take a look at a sample implementation of the addCard:atTop: method.

self.cards is an NSMutableArray ...

... and these are NSMutableArray methods. (insertObject:atIndex: and addObject:).

Stanford CS193p
Fall 2013

When Does (cards *) Property Get Allocated?



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

But there's a problem here.
When does the object pointed to by the pointer returned by self.cards ever get created?

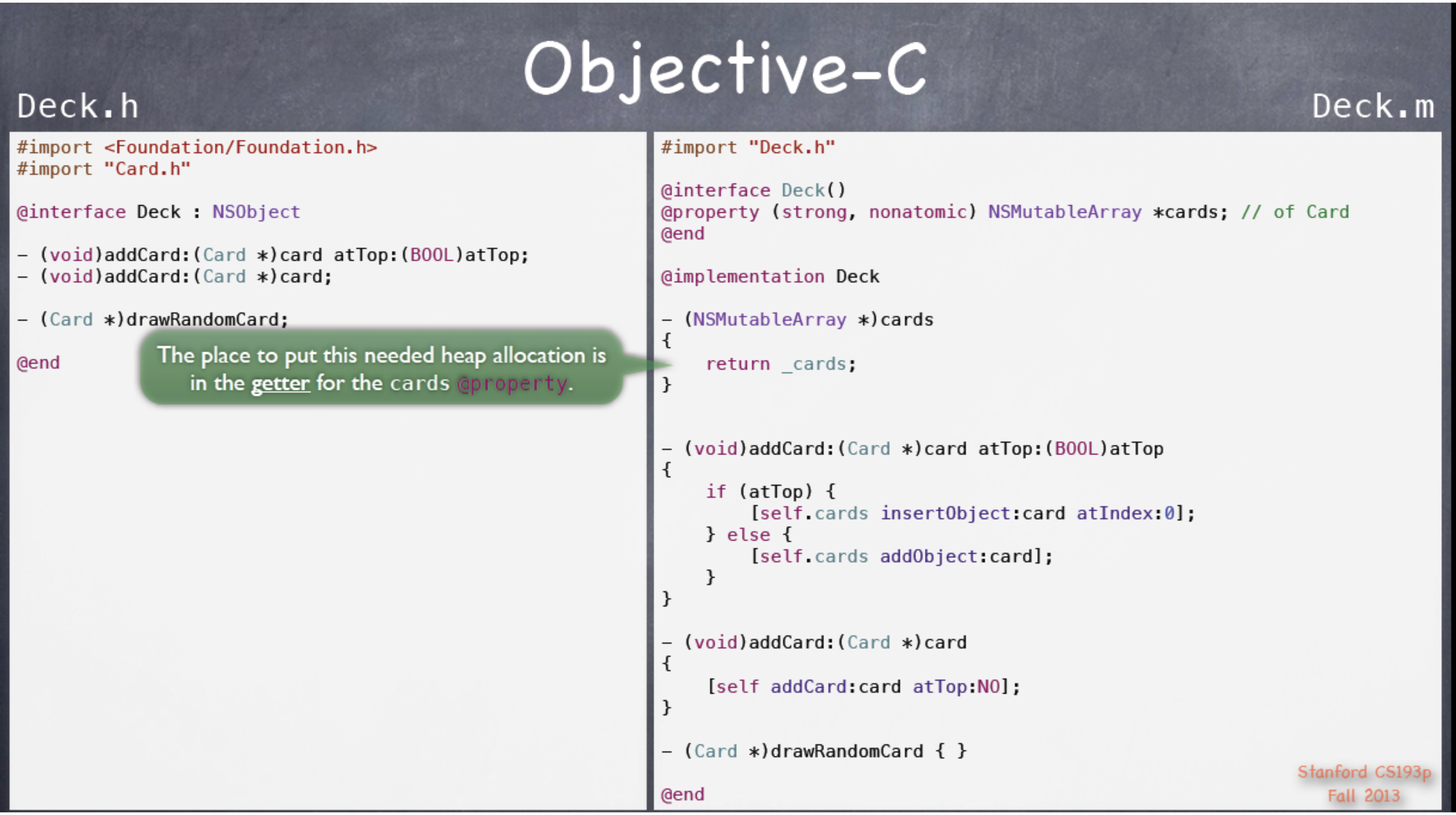
Deck.m

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck









- (void)addCard:(Card *)card atTop:(BOOL)atTop
{
    if (atTop) {
        [self.cards insertObject:card atIndex:0];
    } else {
        [self.cards addObject:card];
    }
}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

Declaring a @property makes space in the instance for the pointer itself, but not does not allocate space in the heap for the object the pointer points to.

Stanford CS193p
Fall 2013

Getter For (cards *) Property



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

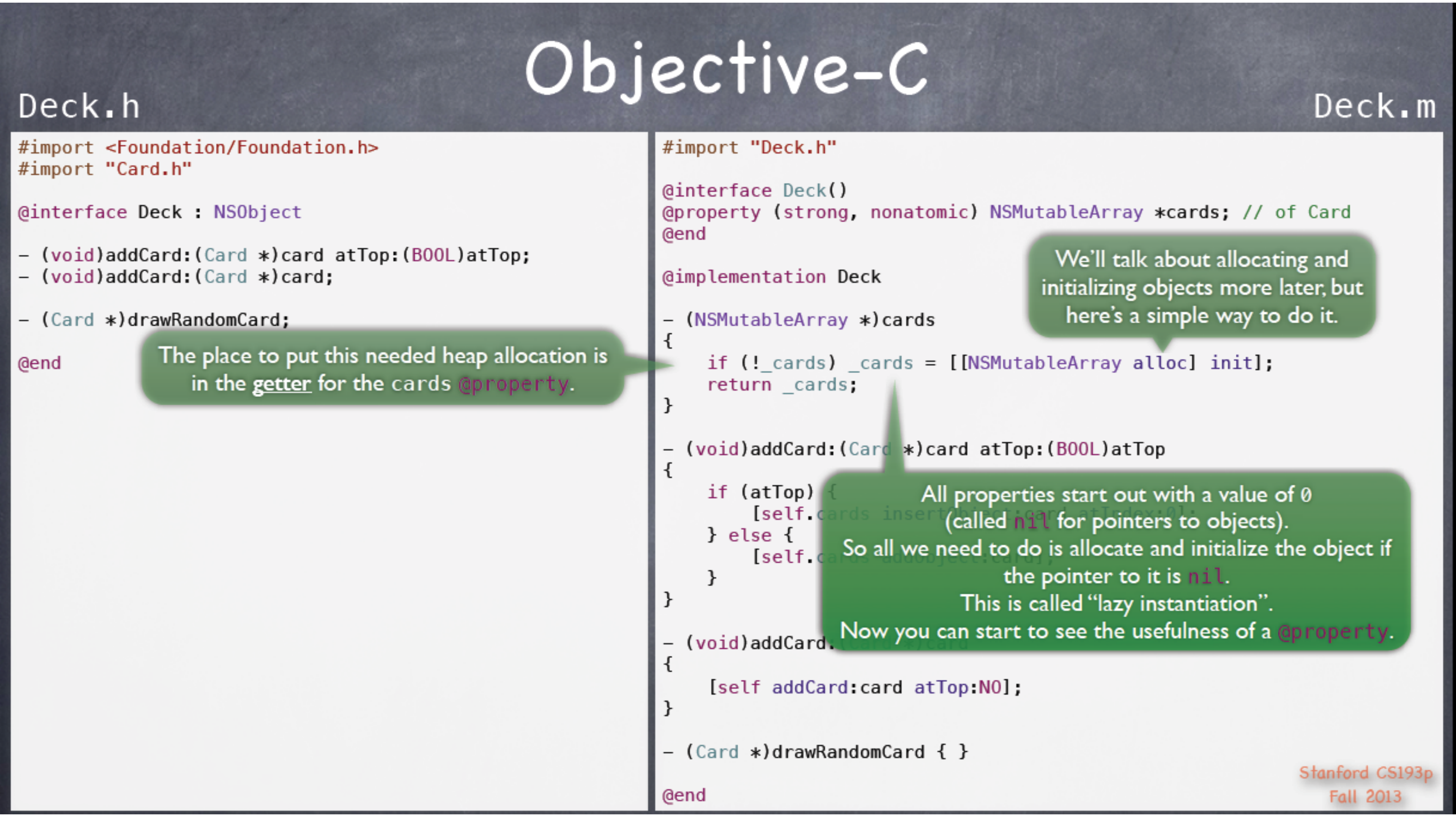
The place to put this needed heap allocation is in the getter for the cards @property.

Deck.m

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (NSMutableArray *)cards
{
    return _cards;
}


- (void)addCard:(Card *)card atTop:(BOOL)atTop
{
    if (atTop) {
        [self.cards insertObject:card atIndex:0];
    } else {
        [self.cards addObject:card];
    }
}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

Stanford CS193p
Fall 2013

Lazy Instantiation In Getter



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

The place to put this needed heap allocation is in the getter for the cards @property.

Deck.m

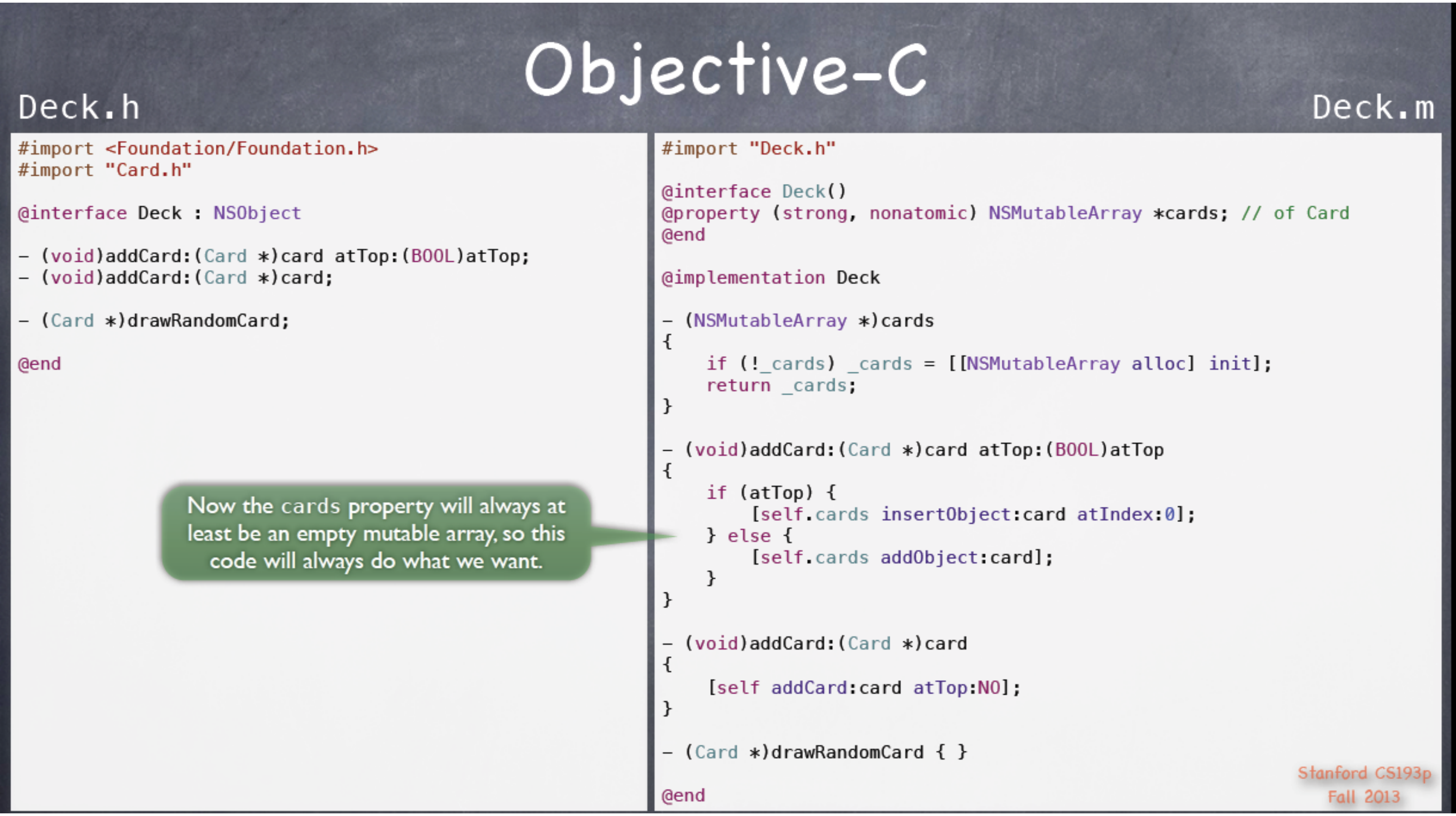
```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (NSMutableArray *)cards
{
    if (!_cards) _cards = [[NSMutableArray alloc] init];
    return _cards;
}

- (void)addCard:(Card *)card atTop:(BOOL)atTop
{
    if (atTop) {
        [self.cards insertObject:card atIndex:0];
    } else {
        [self.cards addObject:card];
    }
}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

We'll talk about allocating and initializing objects more later, but here's a simple way to do it.

All properties start out with a value of 0 (called nil for pointers to objects).
So all we need to do is allocate and initialize the object if the pointer to it is nil.
This is called "lazy instantiation".
Now you can start to see the usefulness of a @property.

Stanford CS193p
Fall 2013

Now addCard:atTop: Will Work



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

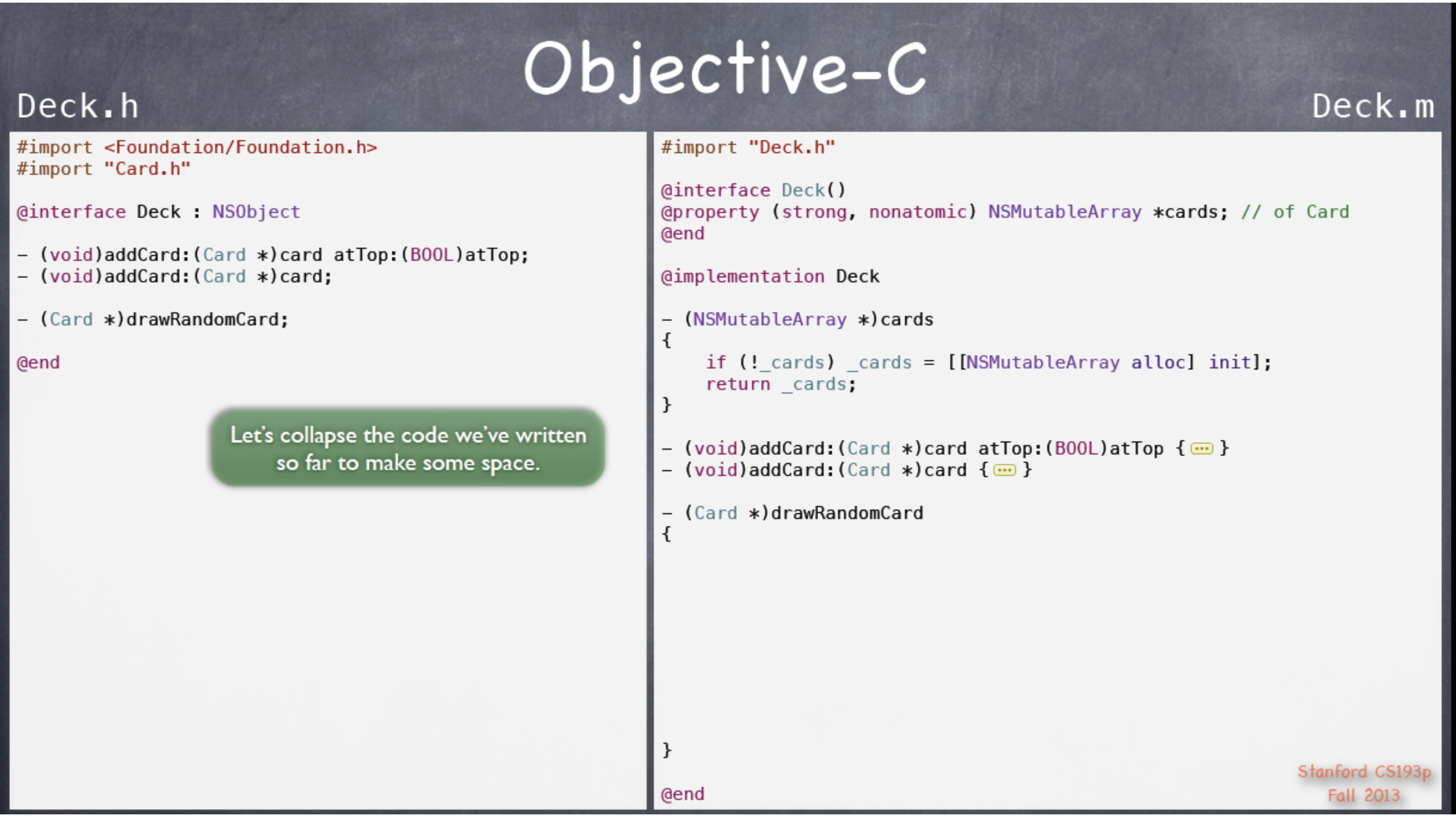
Deck.m

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (NSMutableArray *)cards
{
    if (!_cards) _cards = [[NSMutableArray alloc] init];
    return _cards;
}

- (void)addCard:(Card *)card atTop:(BOOL)atTop
{
    if (atTop) {
        [self.cards insertObject:card atIndex:0];
    } else {
        [self.cards addObject:card];
    }
}

- (void)addCard:(Card *)card
{
    [self addCard:card atTop:NO];
}

- (Card *)drawRandomCard { }

@end
```

Now the cards property will always at least be an empty mutable array, so this code will always do what we want.

Stanford CS193p
Fall 2013

Collapse Code To Make Room



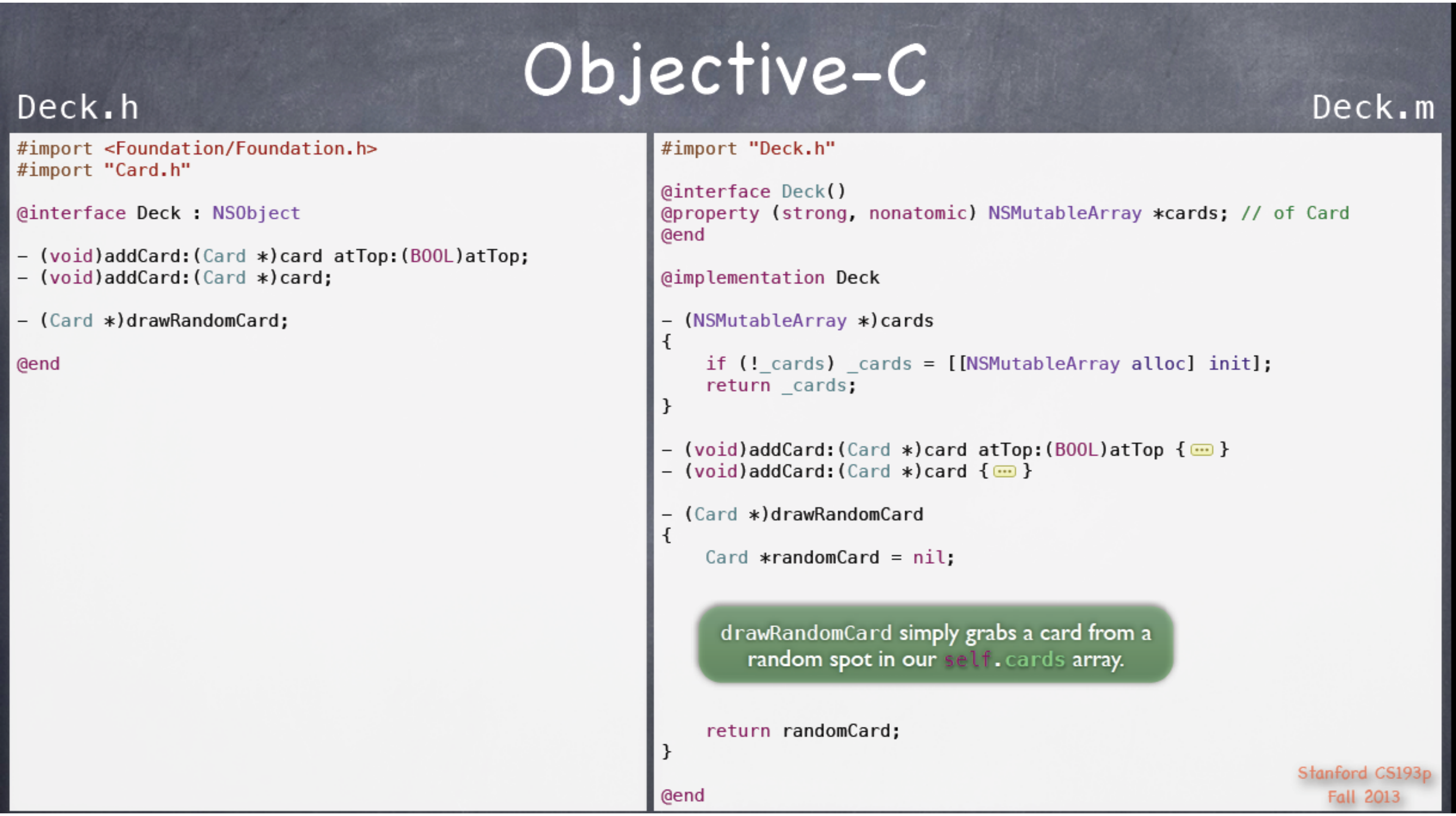
```
#import <Foundation/Foundation.h>
#import "Card.h"

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

Let's collapse the code we've written so far to make some space.

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (NSMutableArray *)cards
{
    if (!_cards) _cards = [[NSMutableArray alloc] init];
    return _cards;
}

- (void)addCard:(Card *)card atTop:(BOOL)atTop {...}
- (void)addCard:(Card *)card {...}

- (Card *)drawRandomCard
{









}

@end
```

Stanford CS193p
Fall 2013

drawRandomCard: Returns A (Card *)



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

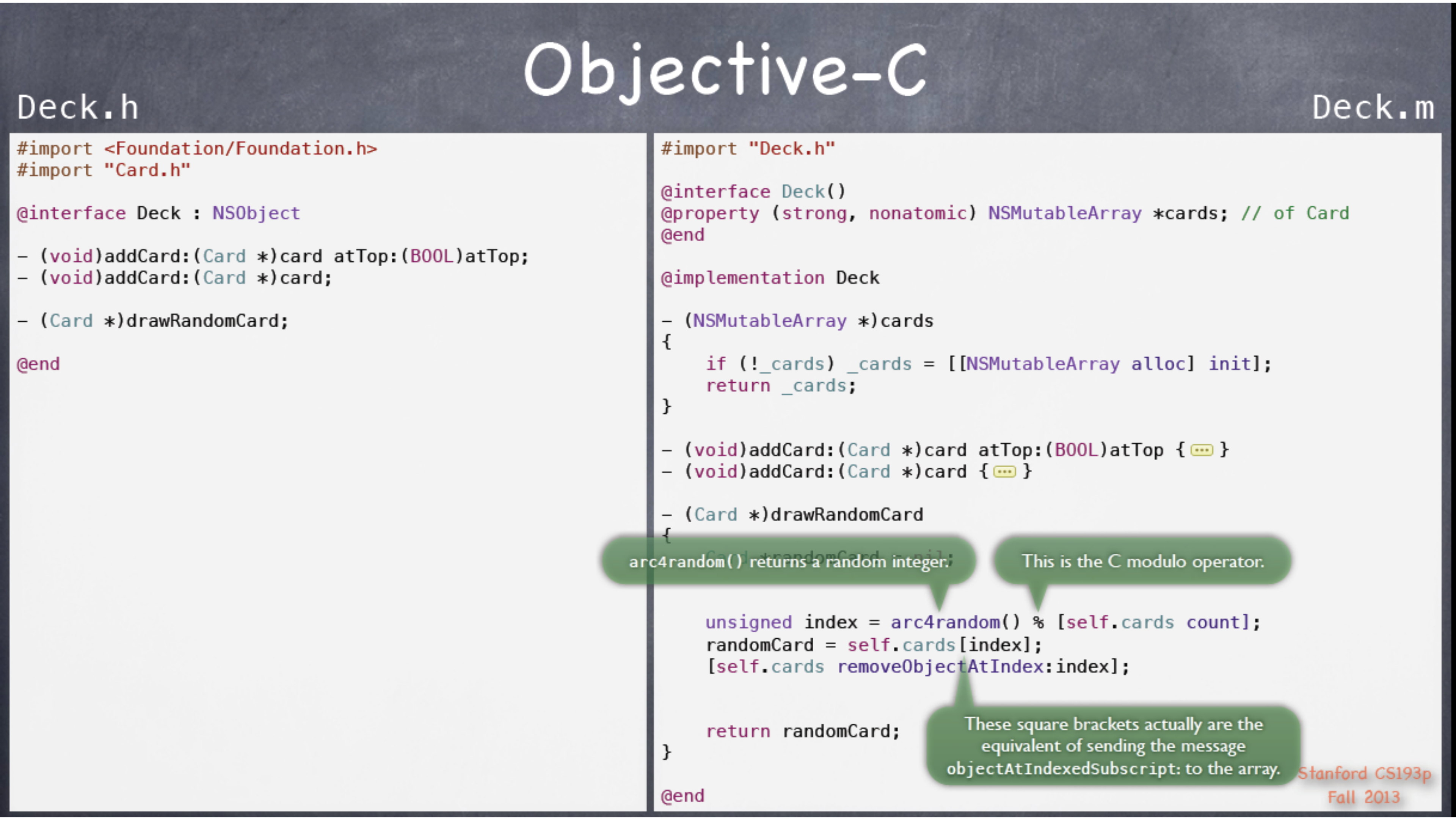
Deck.m

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (NSMutableArray *)cards
{
    if (!_cards) _cards = [[NSMutableArray alloc] init];
    return _cards;
}

- (void)addCard:(Card *)card atTop:(BOOL)atTop {…}
- (void)addCard:(Card *)card {…}

- (Card *)drawRandomCard
{
    Card *randomCard = nil;



    return randomCard;
}

@end
```

drawRandomCard simply grabs a card from a random spot in our self.cards array.

Stanford CS193p
Fall 2013

Implement drawRandomCard:



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

Deck.m

```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (NSMutableArray *)cards
{
    if (!_cards) _cards = [[NSMutableArray alloc] init];
    return _cards;
}

- (void)addCard:(Card *)card atTop:(BOOL)atTop { ... }
- (void)addCard:(Card *)card { ... }

- (Card *)drawRandomCard
{
    Card *randomCard = nil;


    unsigned index = arc4random() % [self.cards count];
    randomCard = self.cards[index];
    [self.cards removeObjectAtIndex:index];


    return randomCard;
}

@end
```

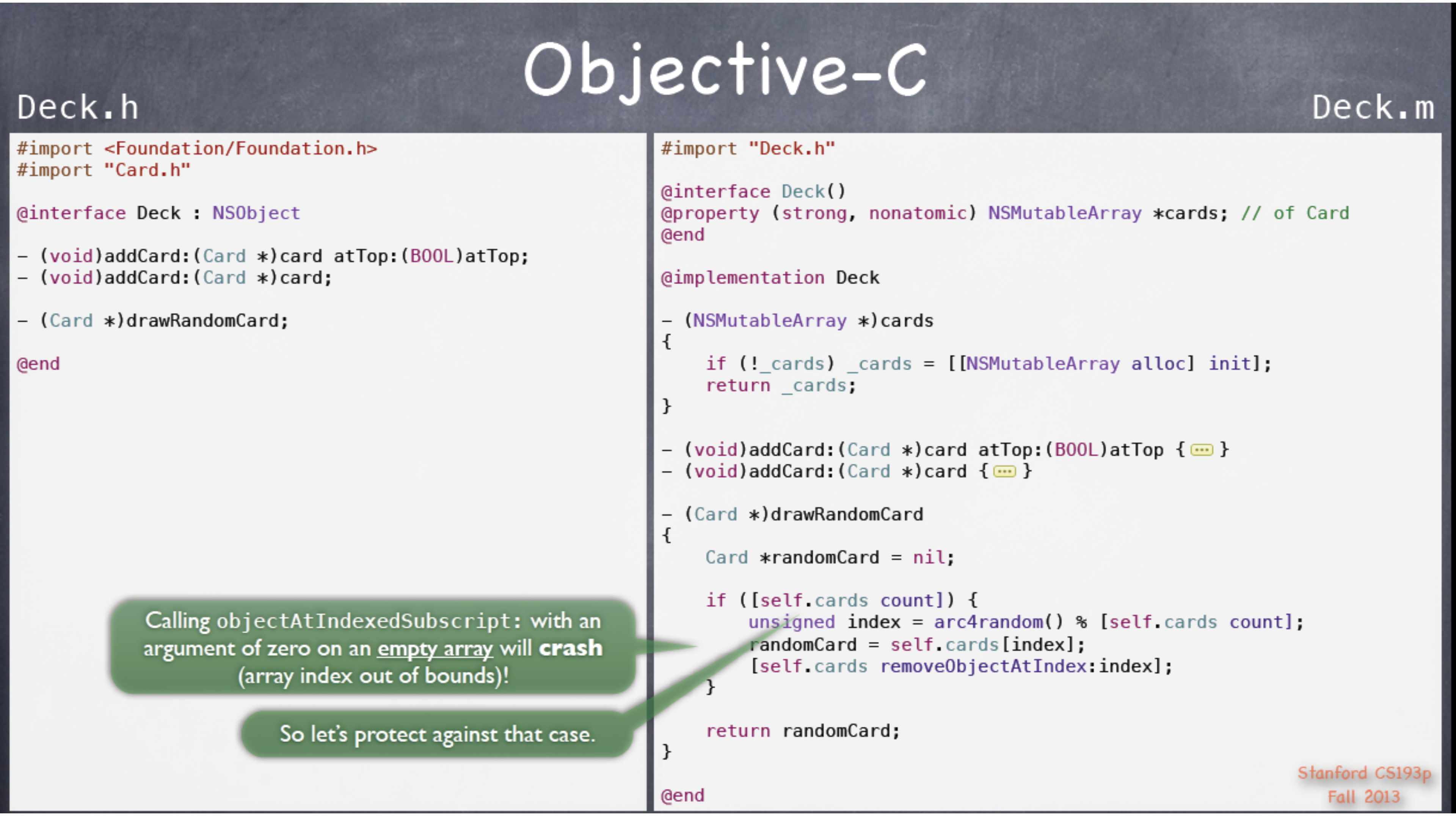
arc4random() returns a random integer.

This is the C modulo operator.

These square brackets actually are the equivalent of sending the message objectAtIndexedSubscript: to the array.

Stanford CS193p
Fall 2013

Protect Against An Empty Array



Objective-C

Deck.h

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

@interface Deck : NSObject

- (void)addCard:(Card *)card atTop:(BOOL)atTop;
- (void)addCard:(Card *)card;

- (Card *)drawRandomCard;

@end
```

Deck.m

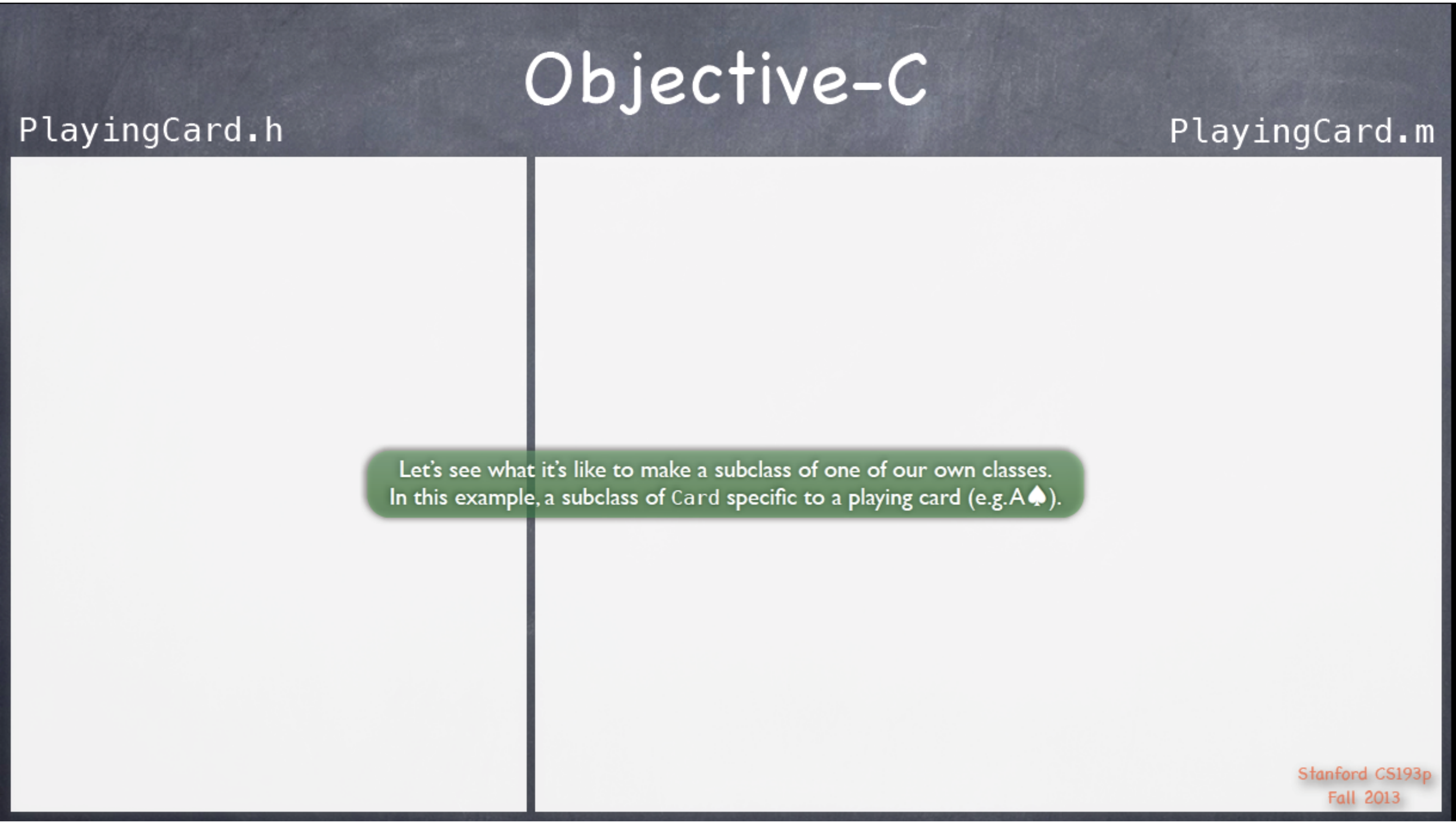
```
#import "Deck.h"

@interface Deck()
@property (strong, nonatomic) NSMutableArray *cards; // of Card
@end

@implementation Deck

- (NSMutableArray *)cards
{
    if (!_cards) _cards = [[NSMutableArray alloc] init];
    return _cards;
}

- (void)addCard:(Card *)card atTop:(BOOL)atTop {...}
- (void)addCard:(Card *)card {...}

- (Card *)drawRandomCard
{
    Card *randomCard = nil;

    if ([self.cards count]) {
        unsigned index = arc4random() % [self.cards count];
        randomCard = self.cards[index];
        [self.cards removeObjectAtIndex:index];
    }

    return randomCard;
}

@end
```

Calling objectAtIndexedSubscript: with an argument of zero on an empty array will **crash** (array index out of bounds)!

So let's protect against that case.

Stanford CS193p
Fall 2013

Create A Subclass of Card



Objective-C

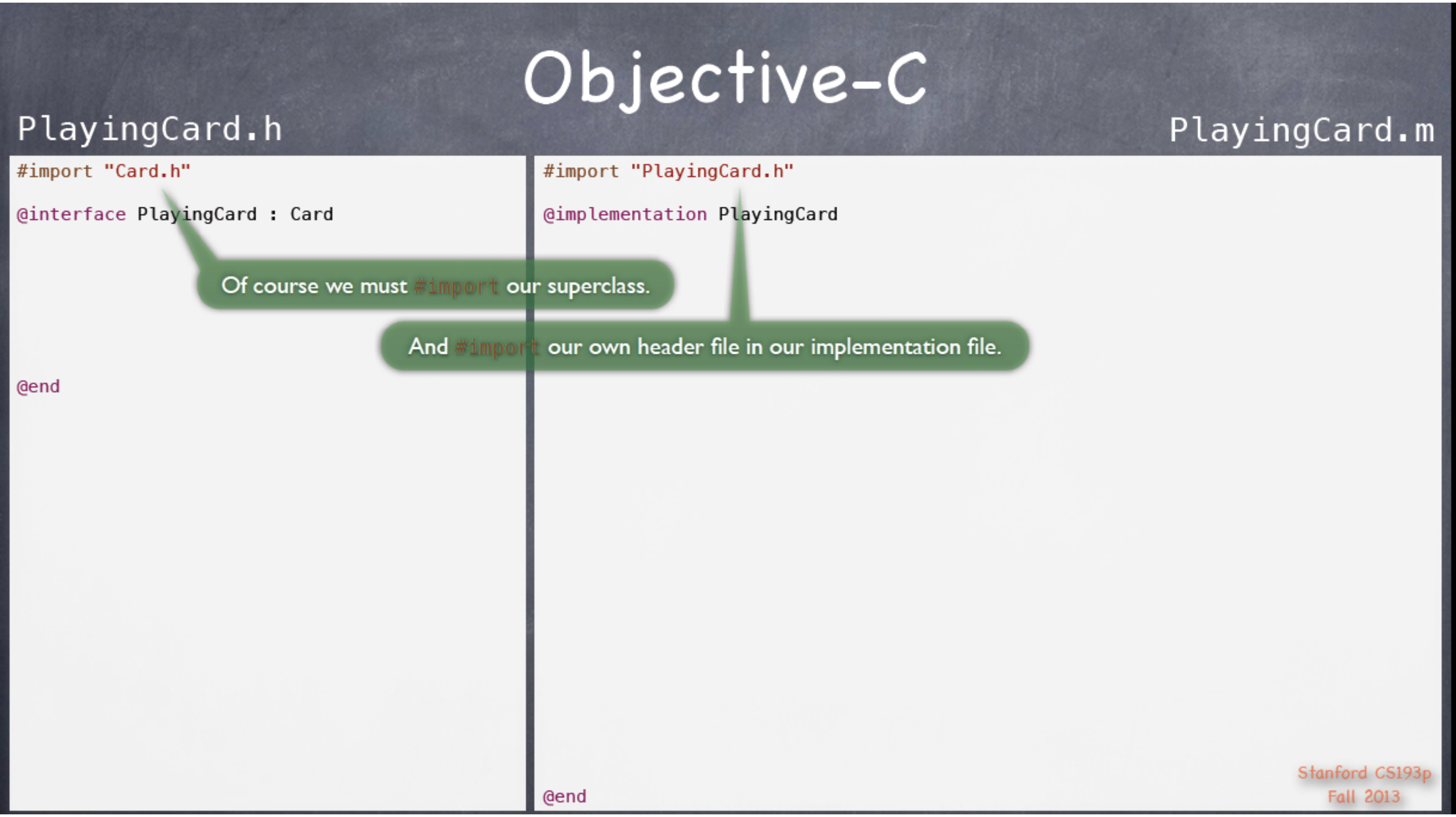
PlayingCard.h

PlayingCard.m

Let's see what it's like to make a subclass of one of our own classes.
In this example, a subclass of Card specific to a playing card (e.g. A♠).

Stanford CS193p
Fall 2013

Make Sure To Have Correct Imports



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card




@end
```

PlayingCard.m

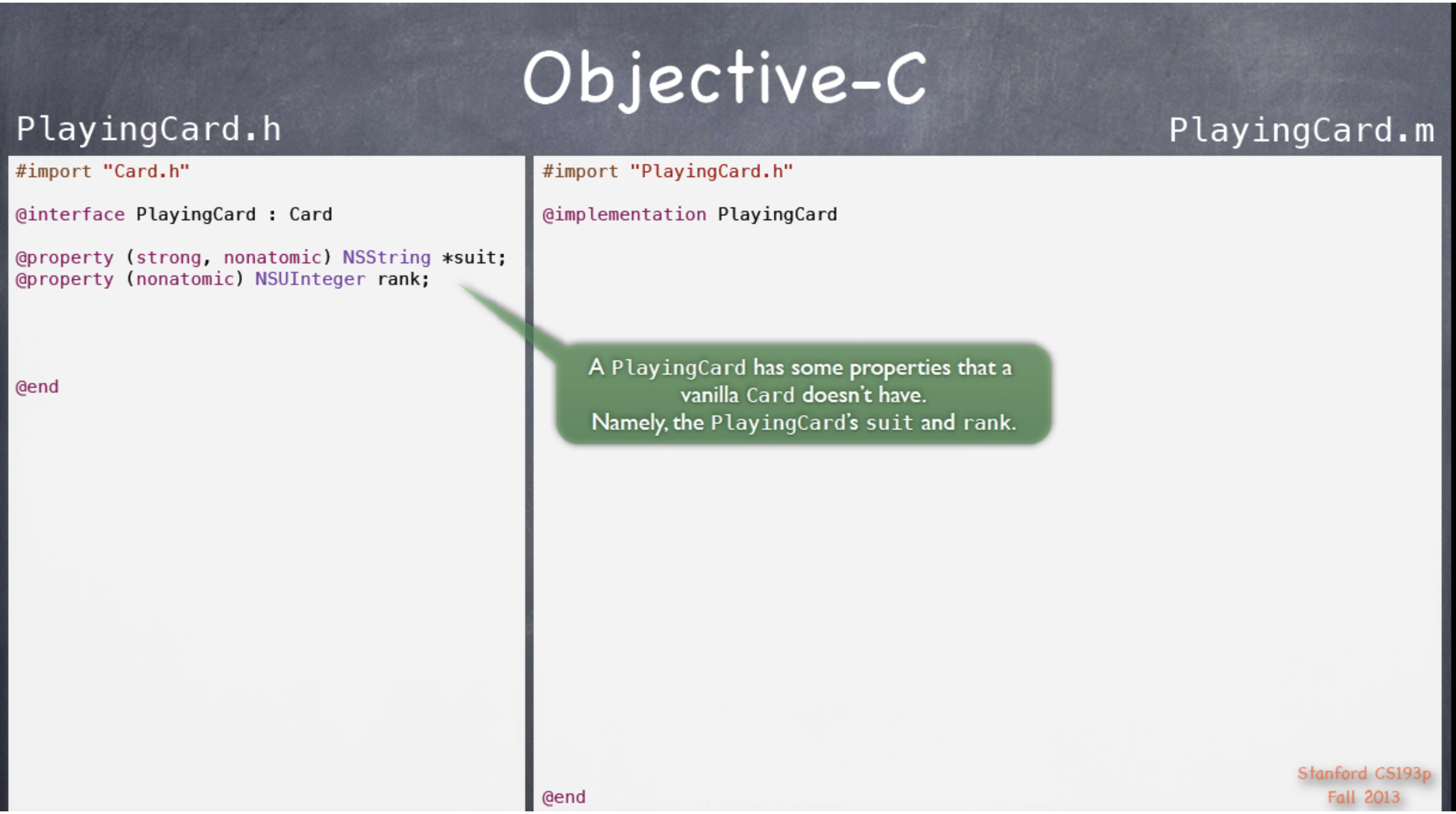
```
#import "PlayingCard.h"

@implementation PlayingCard











@end
```

Of course we must #import our superclass.

And #import our own header file in our implementation file.

Stanford CS193p
Fall 2013

Define The Properties



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

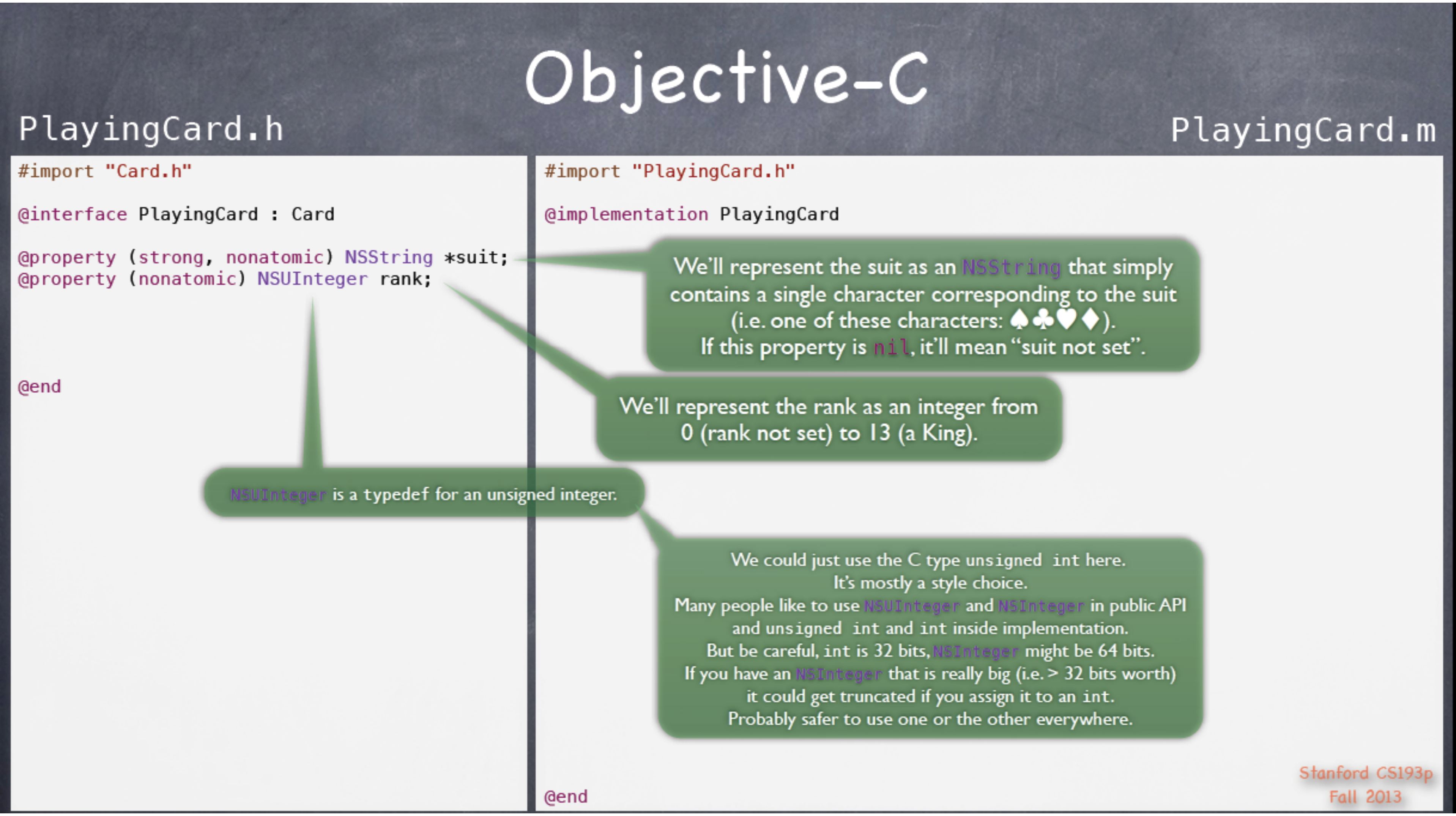
PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

@end
```

A PlayingCard has some properties that a vanilla Card doesn't have.
Namely, the PlayingCard's suit and rank.

Stanford CS193p
Fall 2013

More About PlayingCard Properties



Objective-C

PlayingCard.h

```objc
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

```objc
#import "PlayingCard.h"

@implementation PlayingCard

@end
```

We'll represent the suit as an NSString that simply contains a single character corresponding to the suit (i.e. one of these characters: ♠♣♥♦). If this property is nil, it'll mean "suit not set".

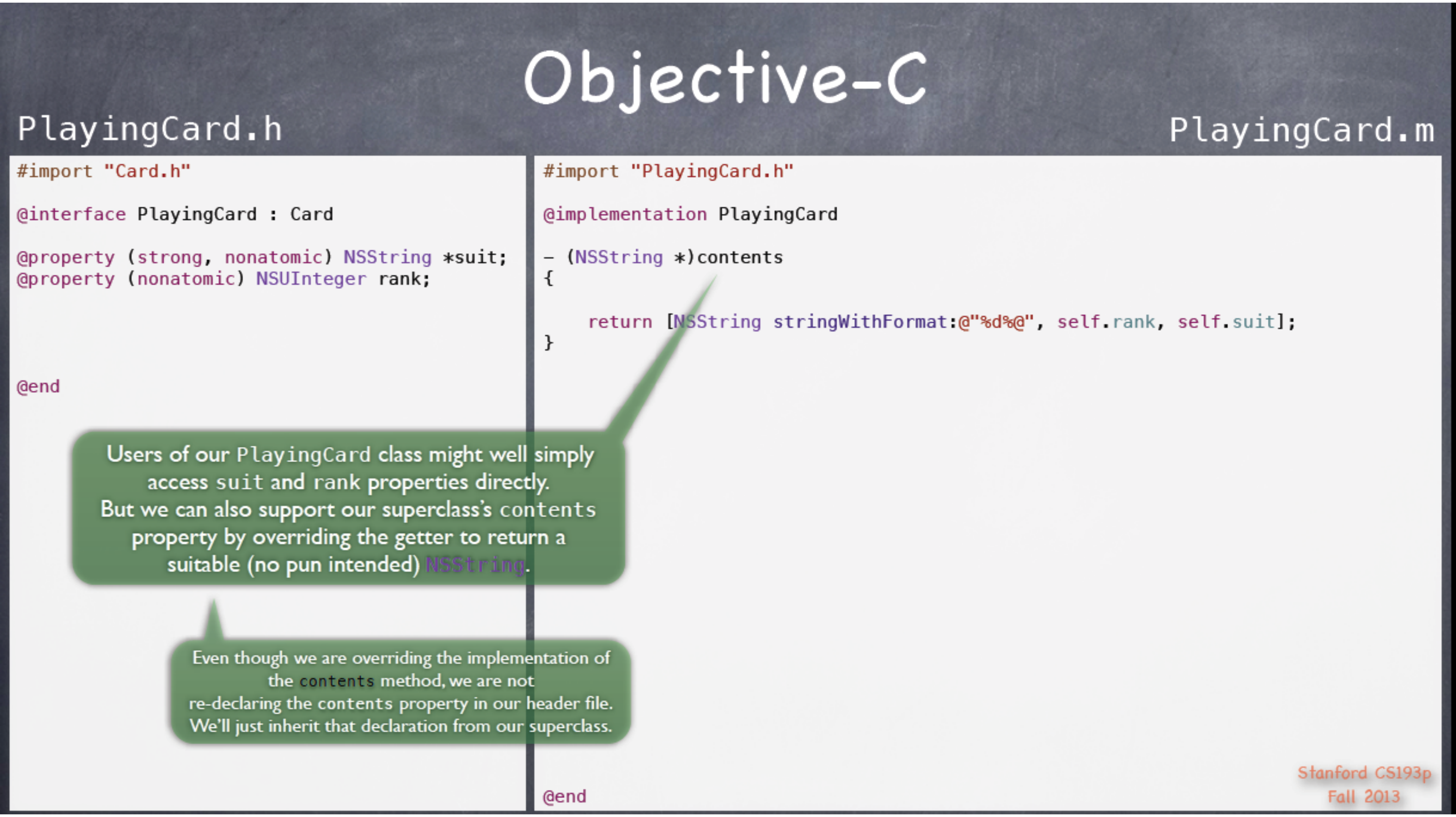
We'll represent the rank as an integer from 0 (rank not set) to 13 (a King).

NSUInteger is a typedef for an unsigned integer.

We could just use the C type unsigned int here.
It's mostly a style choice.
Many people like to use NSUInteger and NSInteger in public API and unsigned int and int inside implementation.
But be careful, int is 32 bits, NSInteger might be 64 bits.
If you have an NSInteger that is really big (i.e. > 32 bits worth) it could get truncated if you assign it to an int.
Probably safer to use one or the other everywhere.

Stanford CS193p
Fall 2013

Override The Getter For contents Property



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

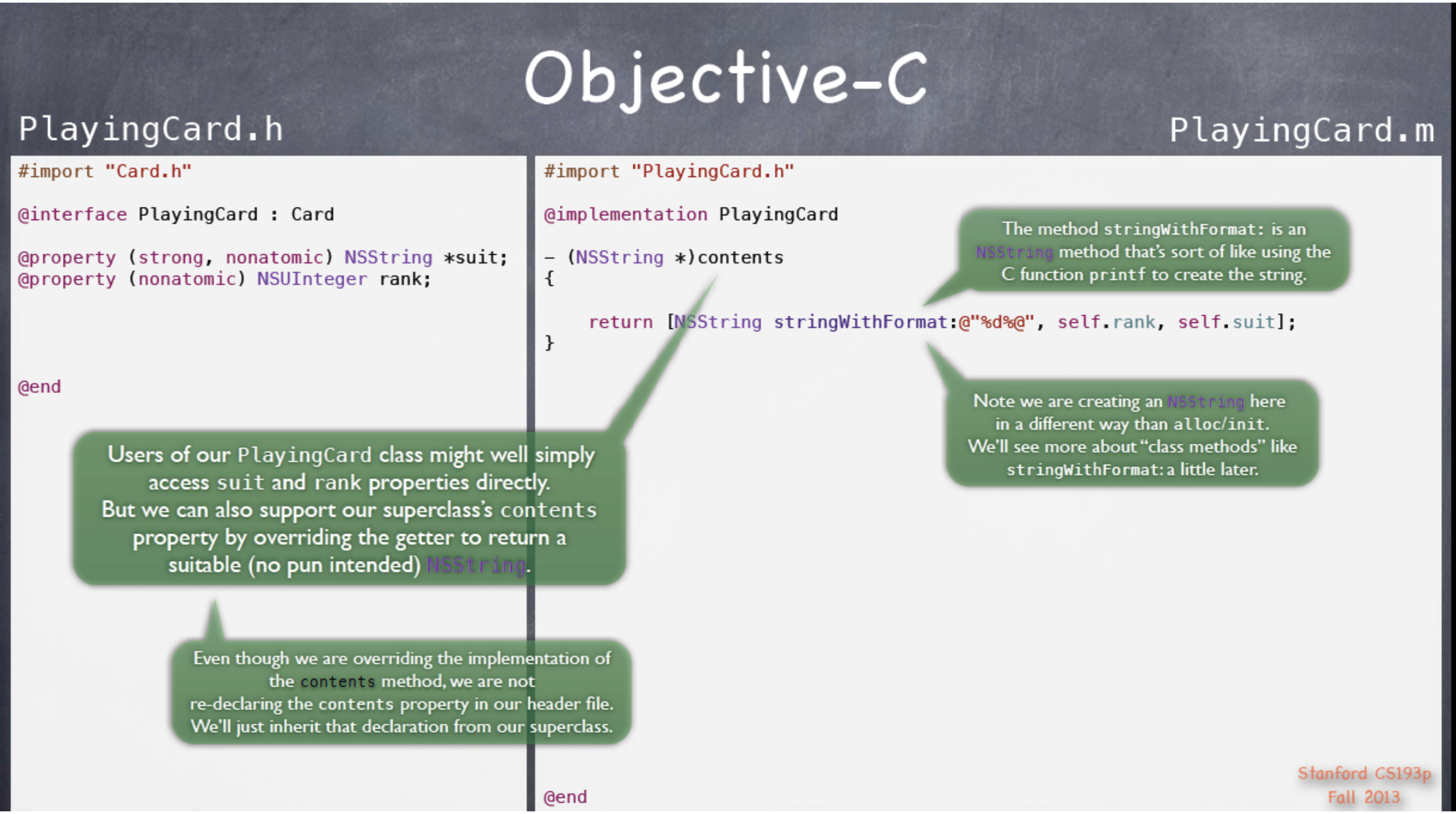
```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{

    return [NSString stringWithFormat:@"%d%@", self.rank, self.suit];
}


@end
```

Users of our PlayingCard class might well simply access suit and rank properties directly.
But we can also support our superclass's contents property by overriding the getter to return a suitable (no pun intended) NSString.

Even though we are overriding the implementation of the contents method, we are not re-declaring the contents property in our header file. We'll just inherit that declaration from our superclass.

Stanford CS193p
Fall 2013

stringWithFormat Method



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    return [NSString stringWithFormat:@"%d%@", self.rank, self.suit];
}

@end
```

The method stringWithFormat: is an NSString method that's sort of like using the C function printf to create the string.

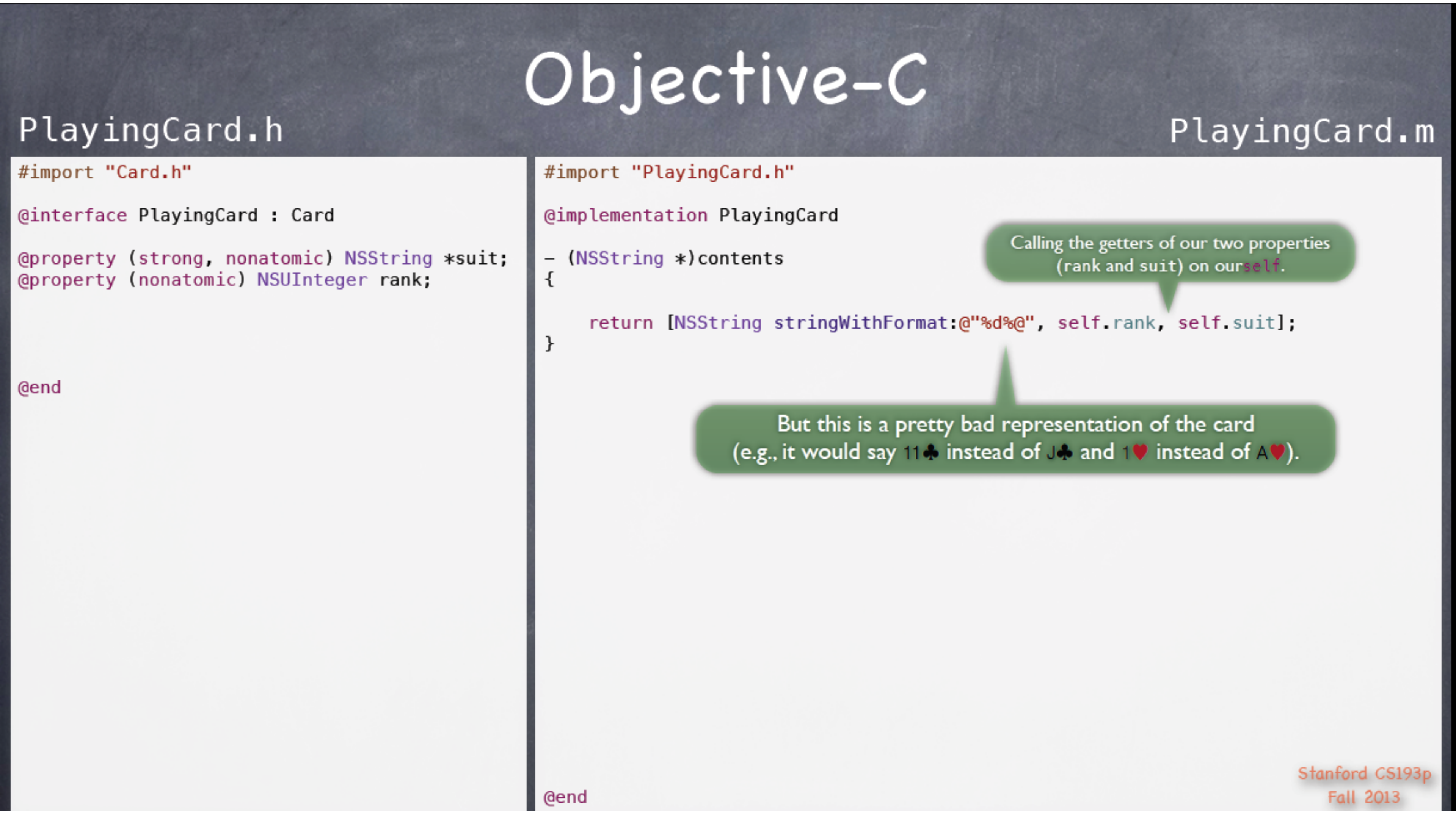
Note we are creating an NSString here in a different way than alloc/init. We'll see more about "class methods" like stringWithFormat: a little later.

Users of our PlayingCard class might well simply access suit and rank properties directly. But we can also support our superclass's contents property by overriding the getter to return a suitable (no pun intended) NSString.

Even though we are overriding the implementation of the contents method, we are not re-declaring the contents property in our header file. We'll just inherit that declaration from our superclass.

Stanford CS193p
Fall 2013

Limitation of Current Implementation



Objective-C

PlayingCard.h

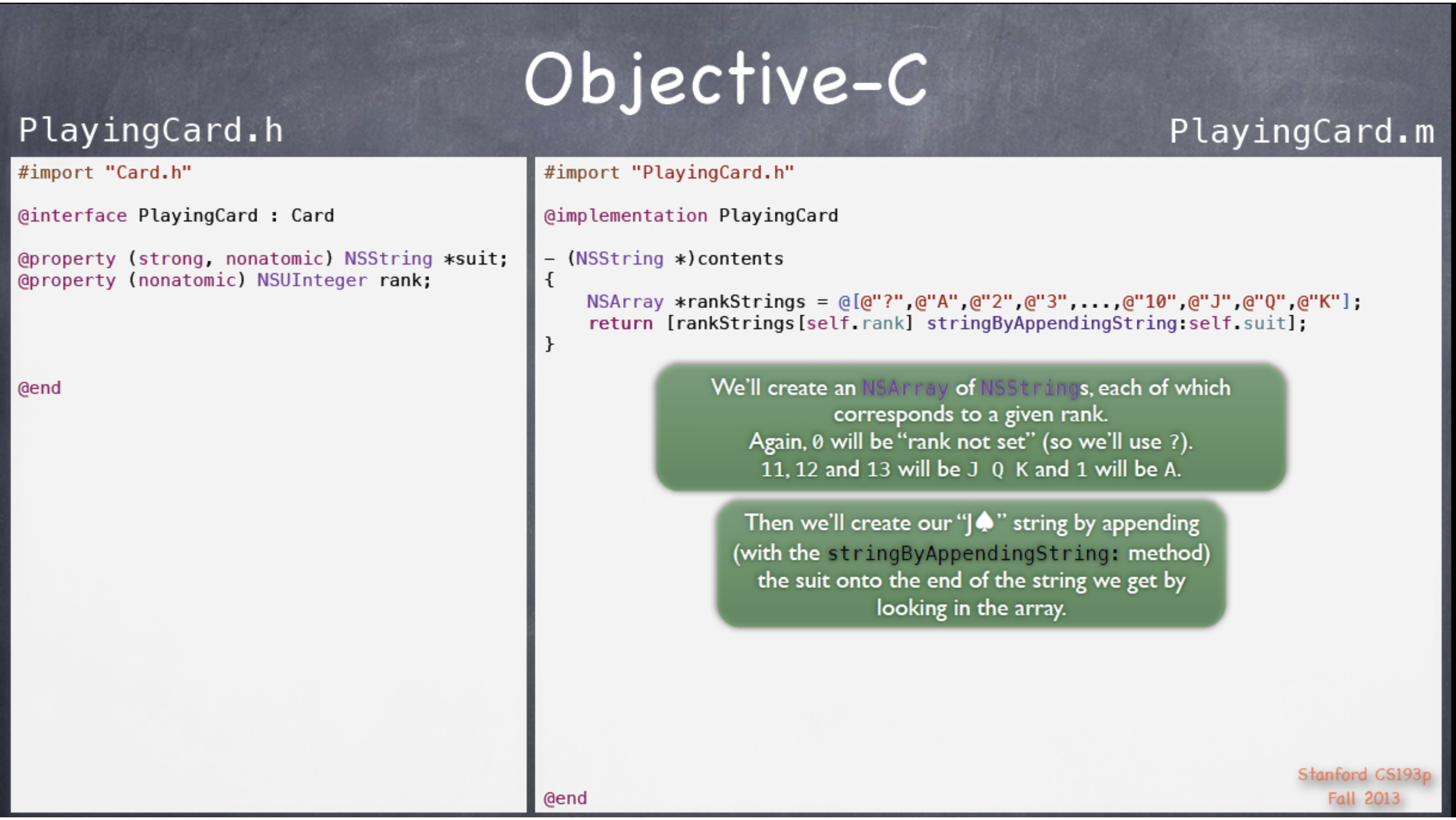
```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{

    return [NSString stringWithFormat:@"%d%@", self.rank, self.suit];
}

@end
```

Calling the getters of our two properties (rank and suit) on ourself.
But this is a pretty bad representation of the card (e.g., it would say 11♣ instead of J♣ and 1♥ instead of A♥).
Stanford CS193p Fall 2013

A Fix To The Problem



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;




@end
```

PlayingCard.m

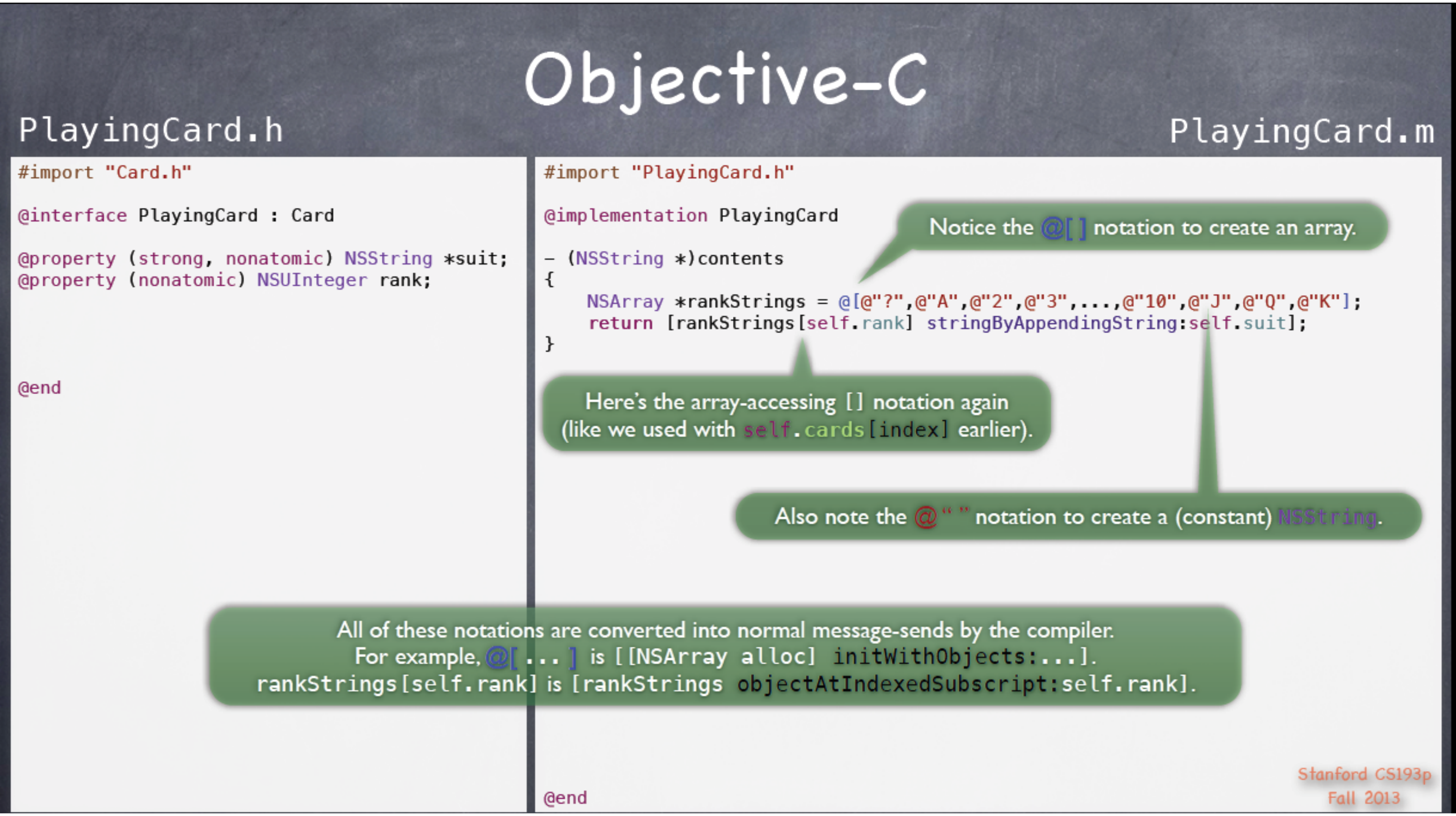
```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@end
```

We'll create an NSArray of NSStrings, each of which corresponds to a given rank.
Again, 0 will be "rank not set" (so we'll use ?).
11, 12 and 13 will be J Q K and 1 will be A.

Then we'll create our "J♠" string by appending (with the stringByAppendingString: method) the suit onto the end of the string we get by looking in the array.

Stanford CS193p
Fall 2013

More About @



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@end
```

Notice the @[ ] notation to create an array.

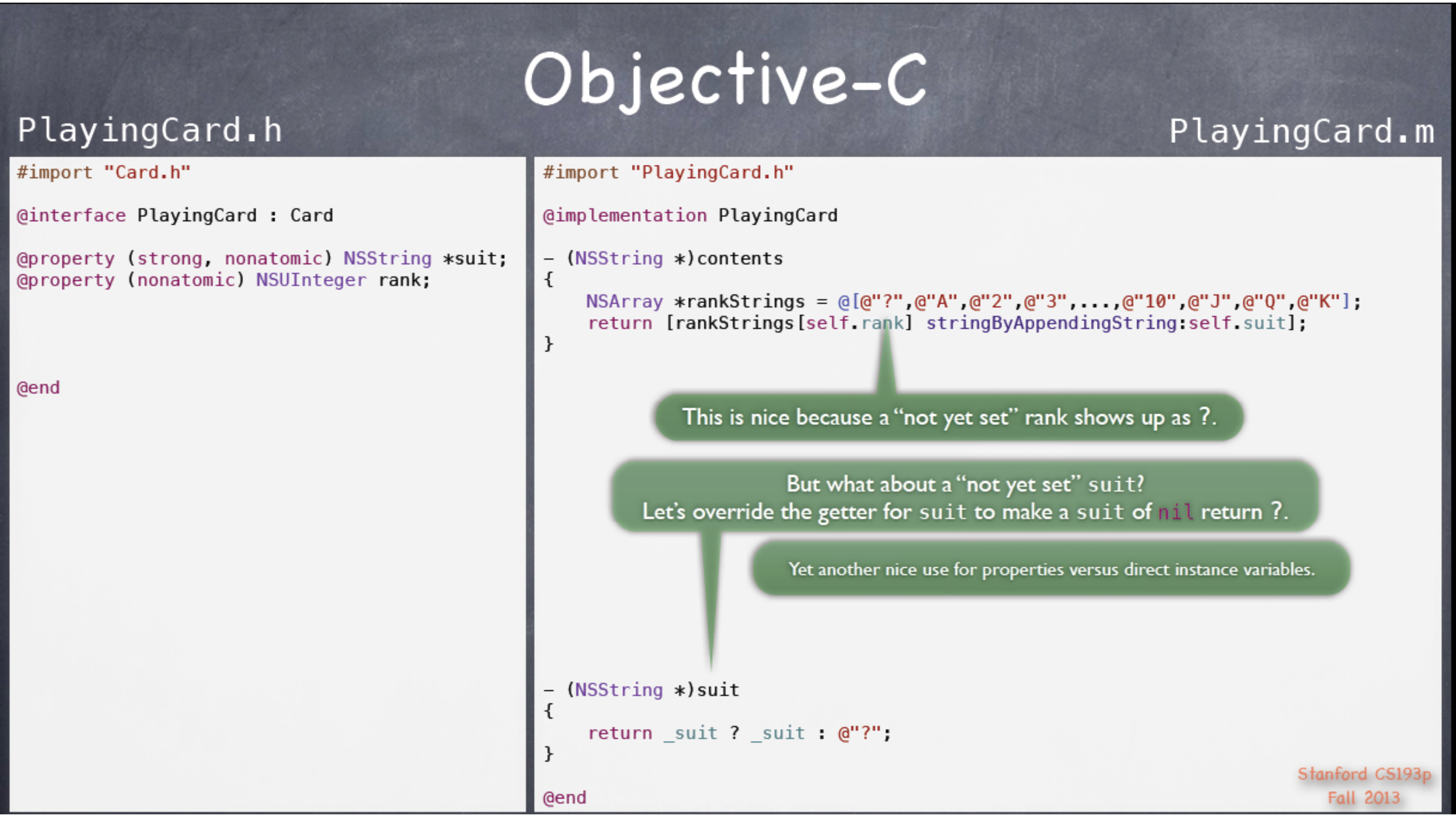
Here's the array-accessing [] notation again (like we used with self.cards[index] earlier).

Also note the @" " notation to create a (constant) NSString.

All of these notations are converted into normal message-sends by the compiler.
For example, @[...] is [[NSArray alloc] initWithObjects:...].
rankStrings[self.rank] is [rankStrings objectAtIndexedSubscript:self.rank].

Stanford CS193p
Fall 2013

Modify suit Getter To Return "?" When Not Set



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

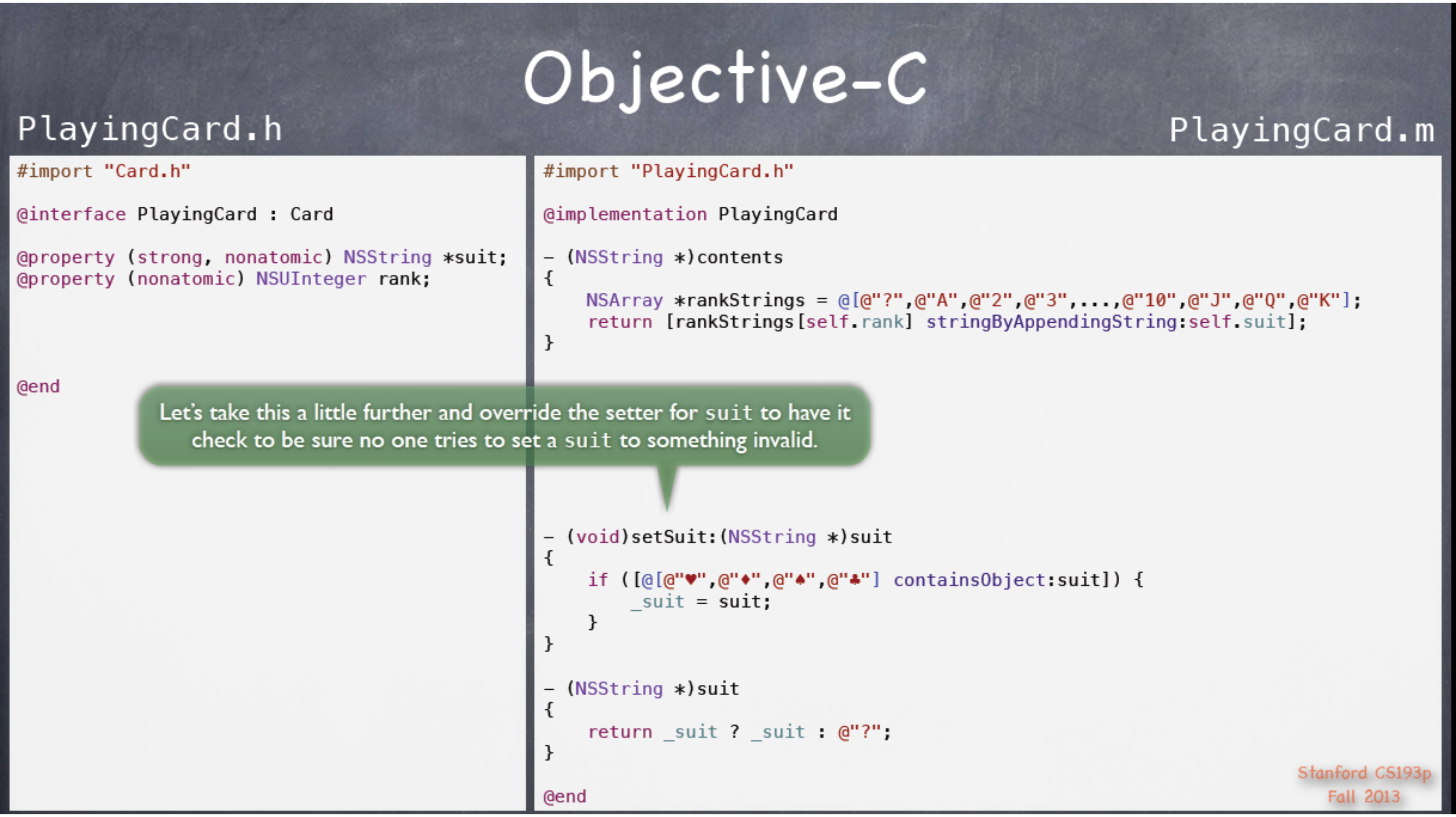
This is nice because a "not yet set" rank shows up as ?.

But what about a "not yet set" suit?
Let's override the getter for suit to make a suit of nil return ?.

Yet another nice use for properties versus direct instance variables.

Stanford CS193p
Fall 2013

Modify suit Setter To Protect Against Invalid Content



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

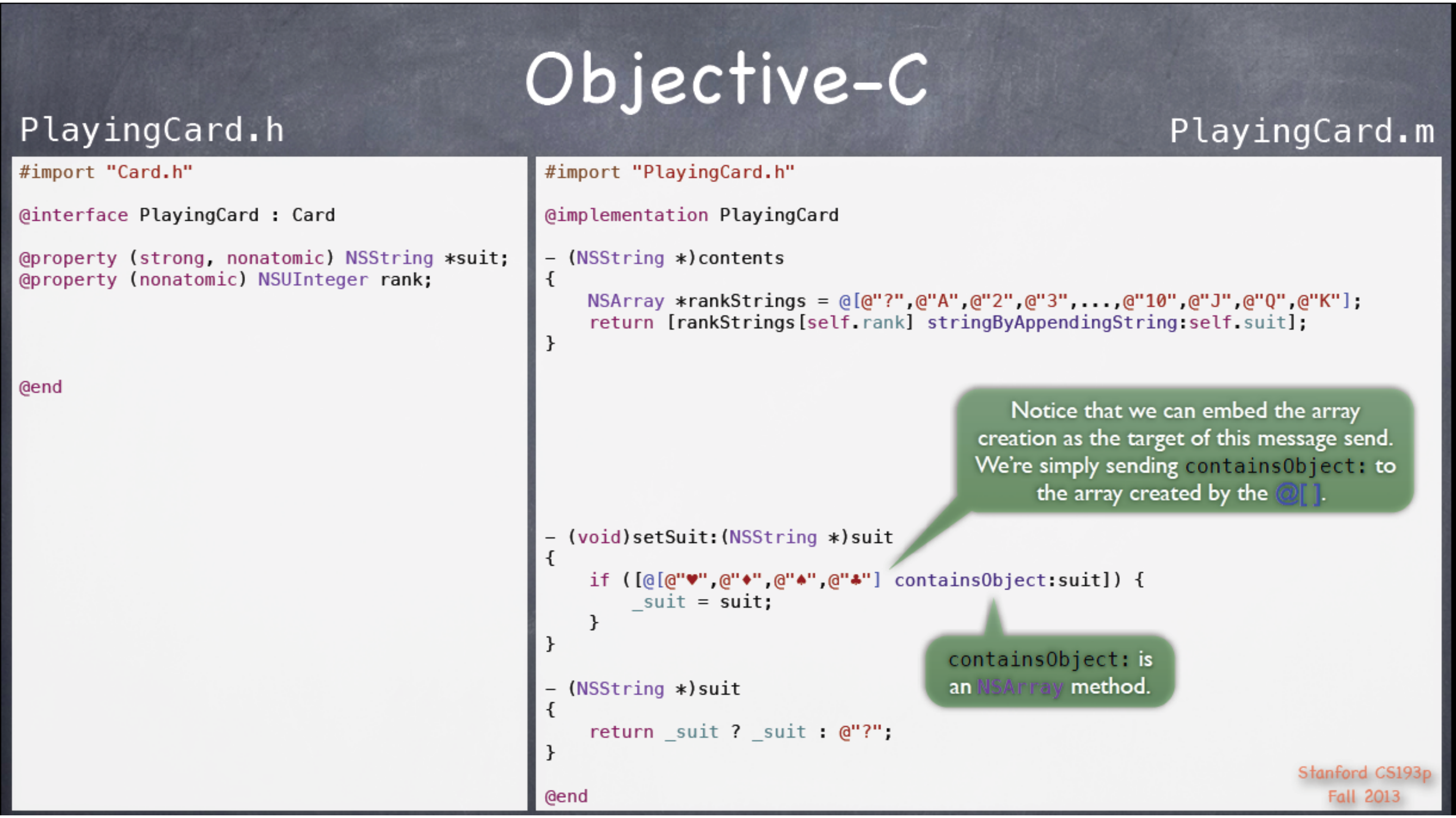
PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

Let's take this a little further and override the setter for suit to have it check to be sure no one tries to set a suit to something invalid.

Stanford CS193p
Fall 2013

Sending Message To An NSArray Created by @[]



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

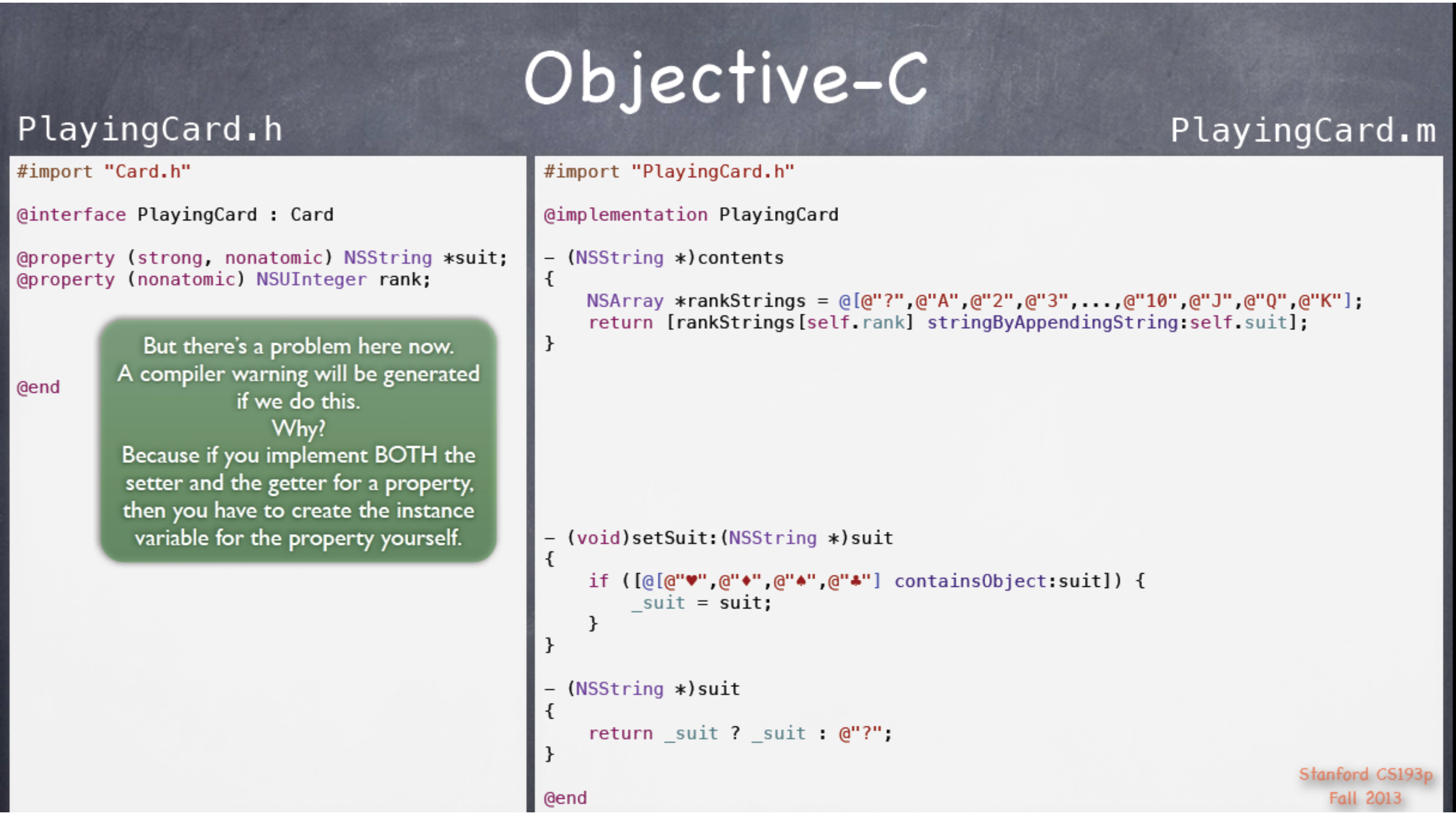
```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

Notice that we can embed the array creation as the target of this message send. We're simply sending `containsObject:` to the array created by the `@[]`.

`containsObject:` is an `NSArray` method.

Stanford CS193p
Fall 2013

Problem With Implementing BOTH The Setter And The Getter For A Property



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;


@end
```

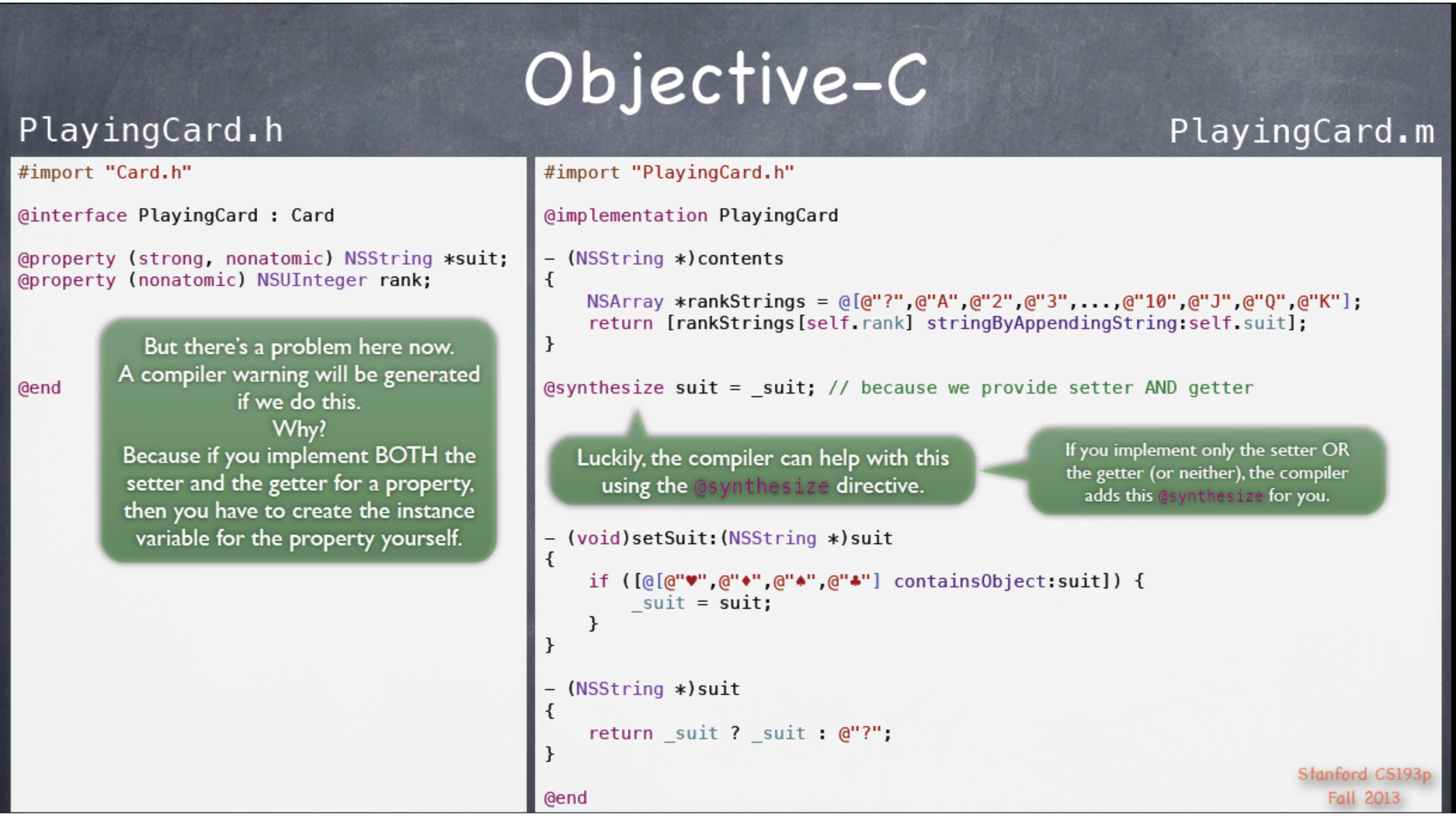
But there's a problem here now.
A compiler warning will be generated if we do this.
Why?
Because if you implement BOTH the setter and the getter for a property, then you have to create the instance variable for the property yourself.

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

Stanford CS193p
Fall 2013

Must Synthesize The suit Property



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

But there's a problem here now.
A compiler warning will be generated if we do this.
Why?
Because if you implement BOTH the setter and the getter for a property, then you have to create the instance variable for the property yourself.

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter
```

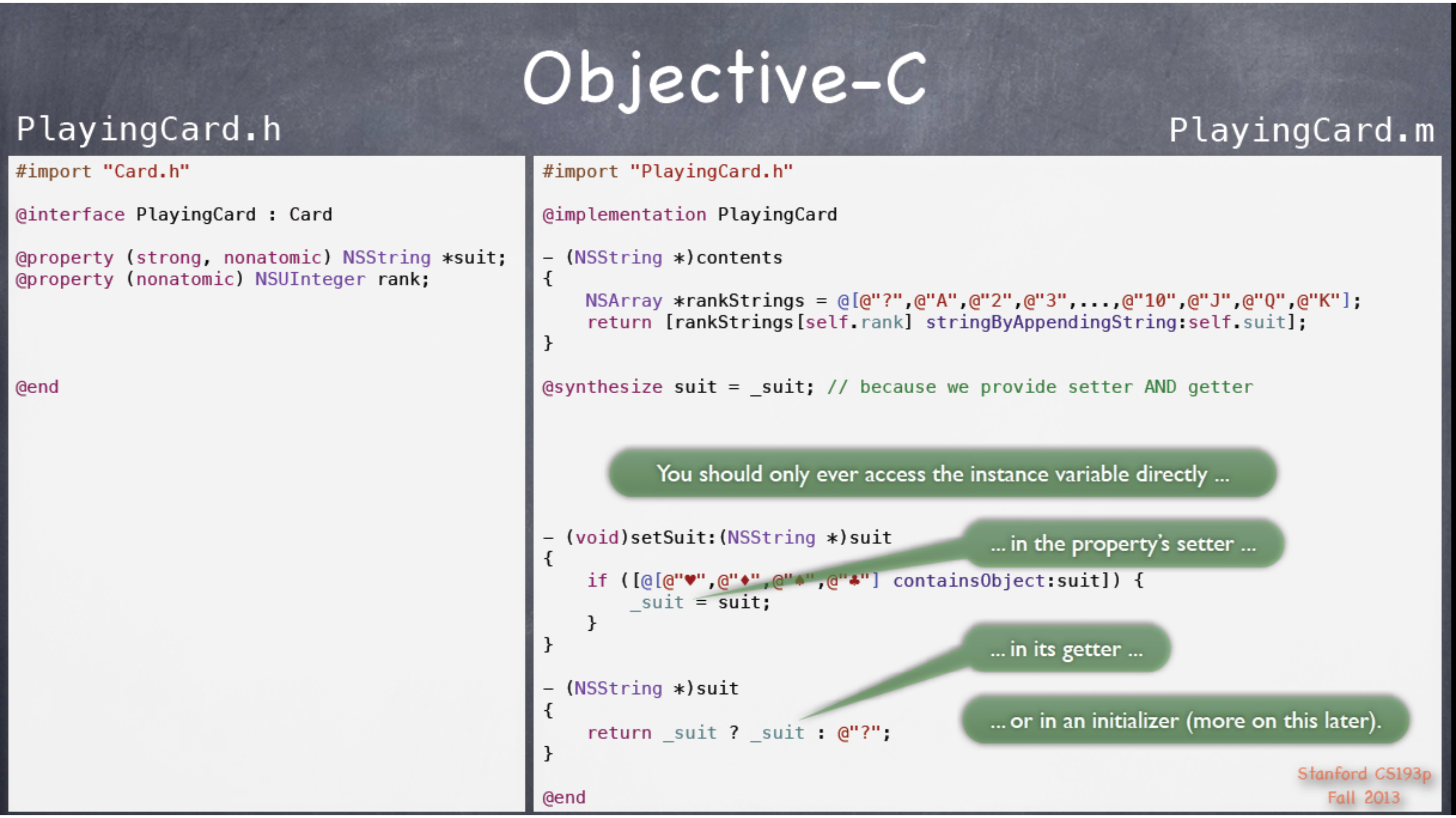
Luckily, the compiler can help with this using the @synthesize directive.

If you implement only the setter OR the getter (or neither), the compiler adds this @synthesize for you.

```
- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

Stanford CS193p
Fall 2013

Only Access _PropertyName From Setter & Getter



Objective-C

PlayingCard.h

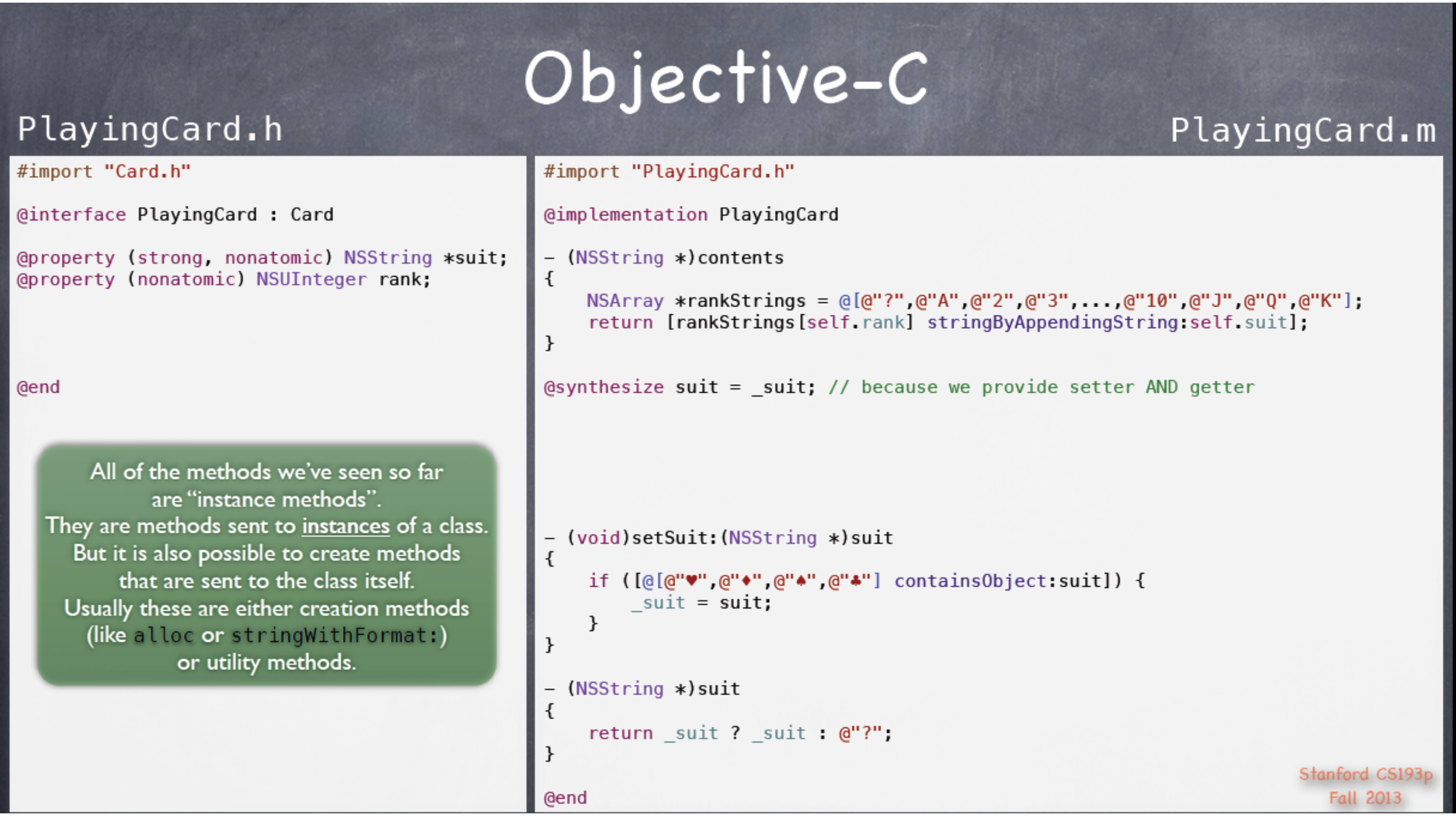
```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter



- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

You should only ever access the instance variable directly ...
... in the property's setter ...
... in its getter ...
... or in an initializer (more on this later).
Stanford CS193p
Fall 2013

Class Methods



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

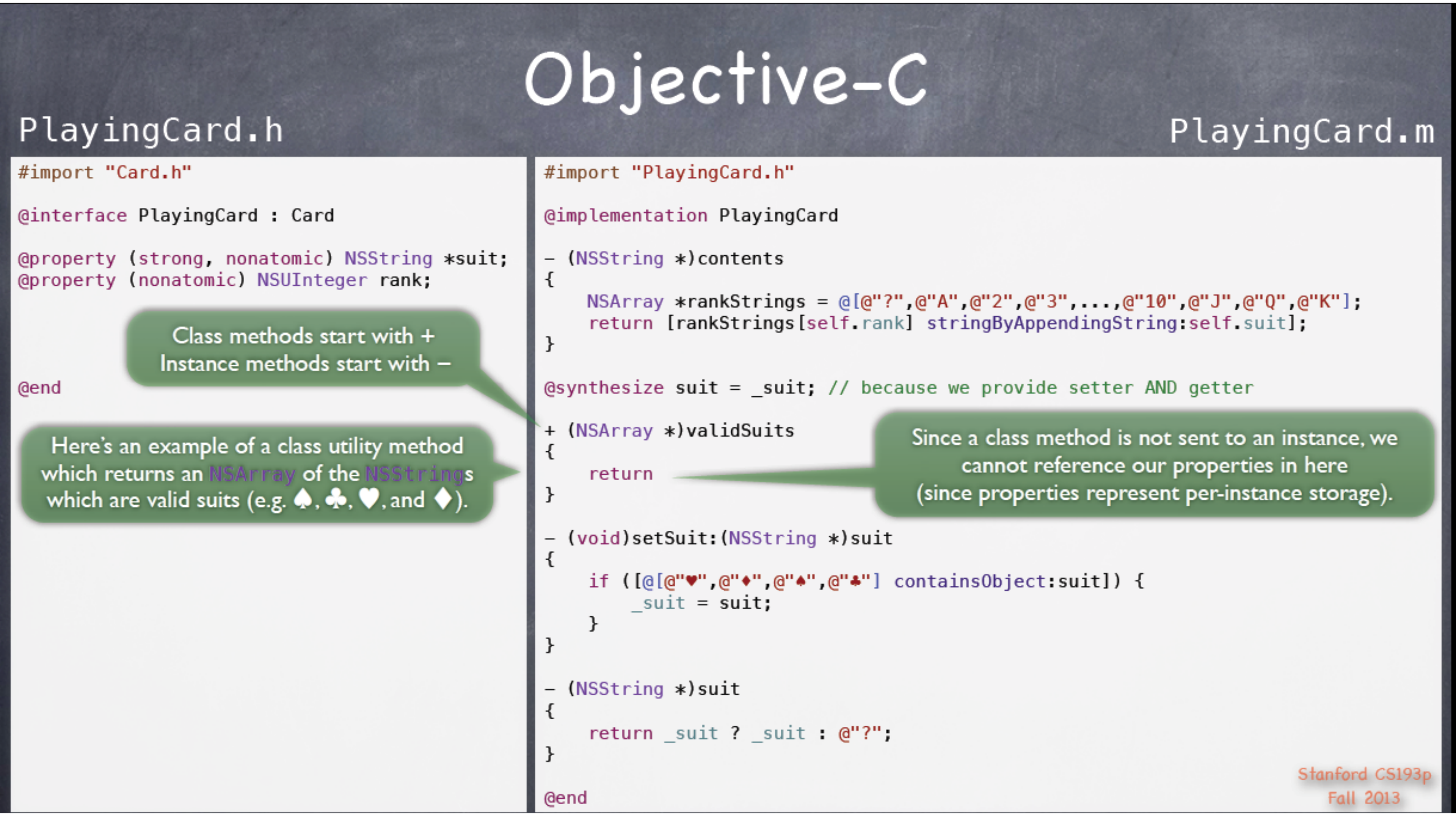
All of the methods we've seen so far are "instance methods".
They are methods sent to <u>instances</u> of a class.
But it is also possible to create methods that are sent to the class itself.
Usually these are either creation methods (like `alloc` or `stringWithFormat:`) or utility methods.

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter





- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

Stanford CS193p
Fall 2013

Our First Class Method



Objective-C

PlayingCard.h

```objc
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;


@end
```

Class methods start with +
Instance methods start with –

Here's an example of a class utility method which returns an NSArray of the NSStrings which are valid suits (e.g. ♠, ♣, ♥, and ♦).

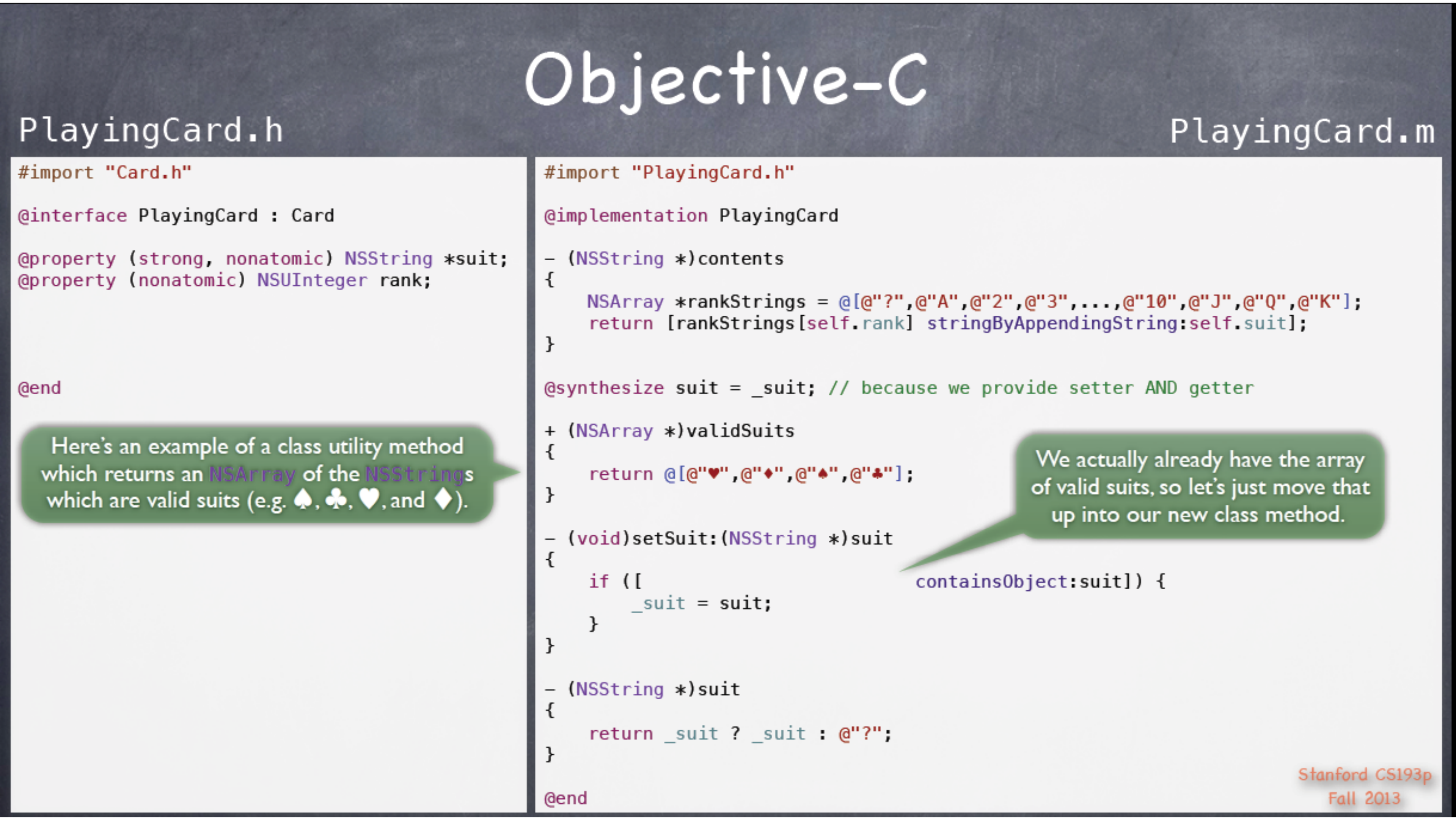
PlayingCard.m

```objc
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter

+ (NSArray *)validSuits
{
    return
}

- (void)setSuit:(NSString *)suit
{
    if ([@[@"♥",@"♦",@"♠",@"♣"] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

Since a class method is not sent to an instance, we cannot reference our properties in here (since properties represent per-instance storage).

Stanford CS193p
Fall 2013

Move The NSArray To The validSuits Method



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;




@end
```

Here's an example of a class utility method which returns an NSArray of the NSStrings which are valid suits (e.g. ♠, ♣, ♥, and ♦).

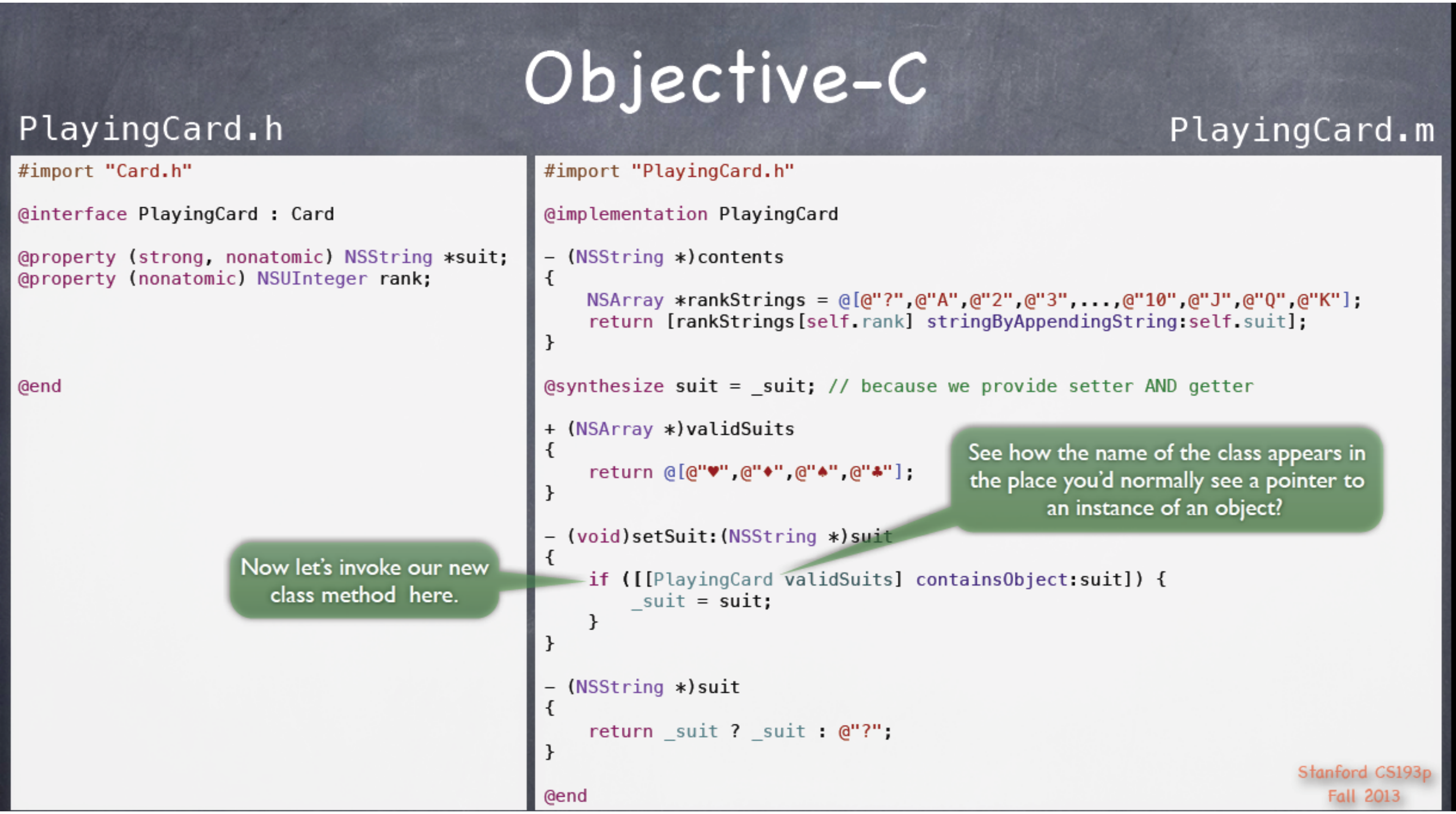
PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter

+ (NSArray *)validSuits
{
    return @[@"♥",@"♦",@"♠",@"♣"];
}

- (void)setSuit:(NSString *)suit
{
    if ([                               containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

We actually already have the array of valid suits, so let's just move that up into our new class method.

Stanford CS193p
Fall 2013

Call validSuits From The Setter



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

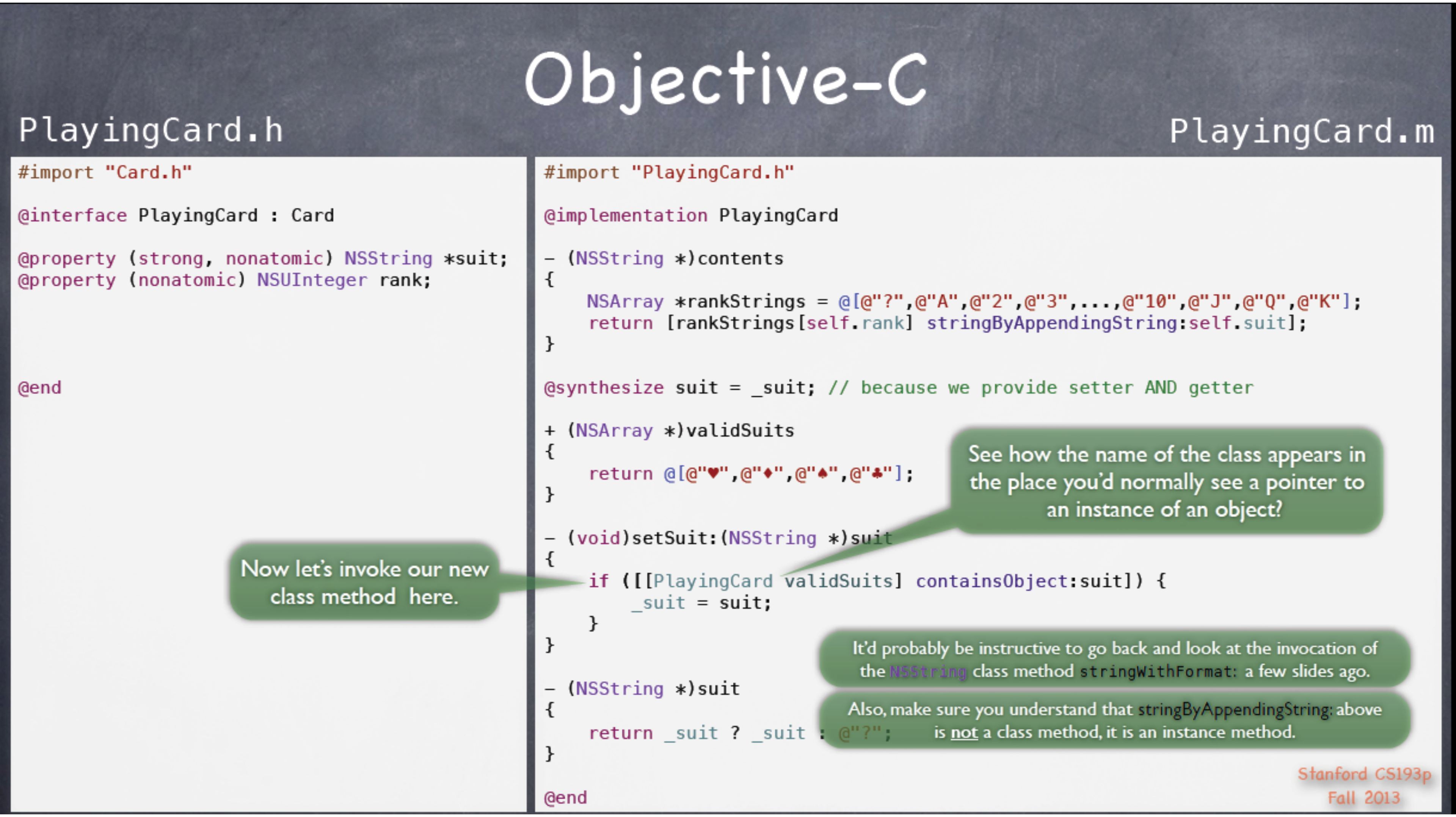
```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter

+ (NSArray *)validSuits
{
    return @[@"♥",@"♦",@"♠",@"♣"];
}

- (void)setSuit:(NSString *)suit
{
    if ([[PlayingCard validSuits] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

See how the name of the class appears in the place you'd normally see a pointer to an instance of an object?

Now let's invoke our new class method here.

Stanford CS193p
Fall 2013

Notice How A Class Method Is Invoked



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;



@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter

+ (NSArray *)validSuits
{
    return @[@"♥",@"♦",@"♠",@"♣"];
}

- (void)setSuit:(NSString *)suit
{
    if ([[PlayingCard validSuits] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

See how the name of the class appears in the place you'd normally see a pointer to an instance of an object?

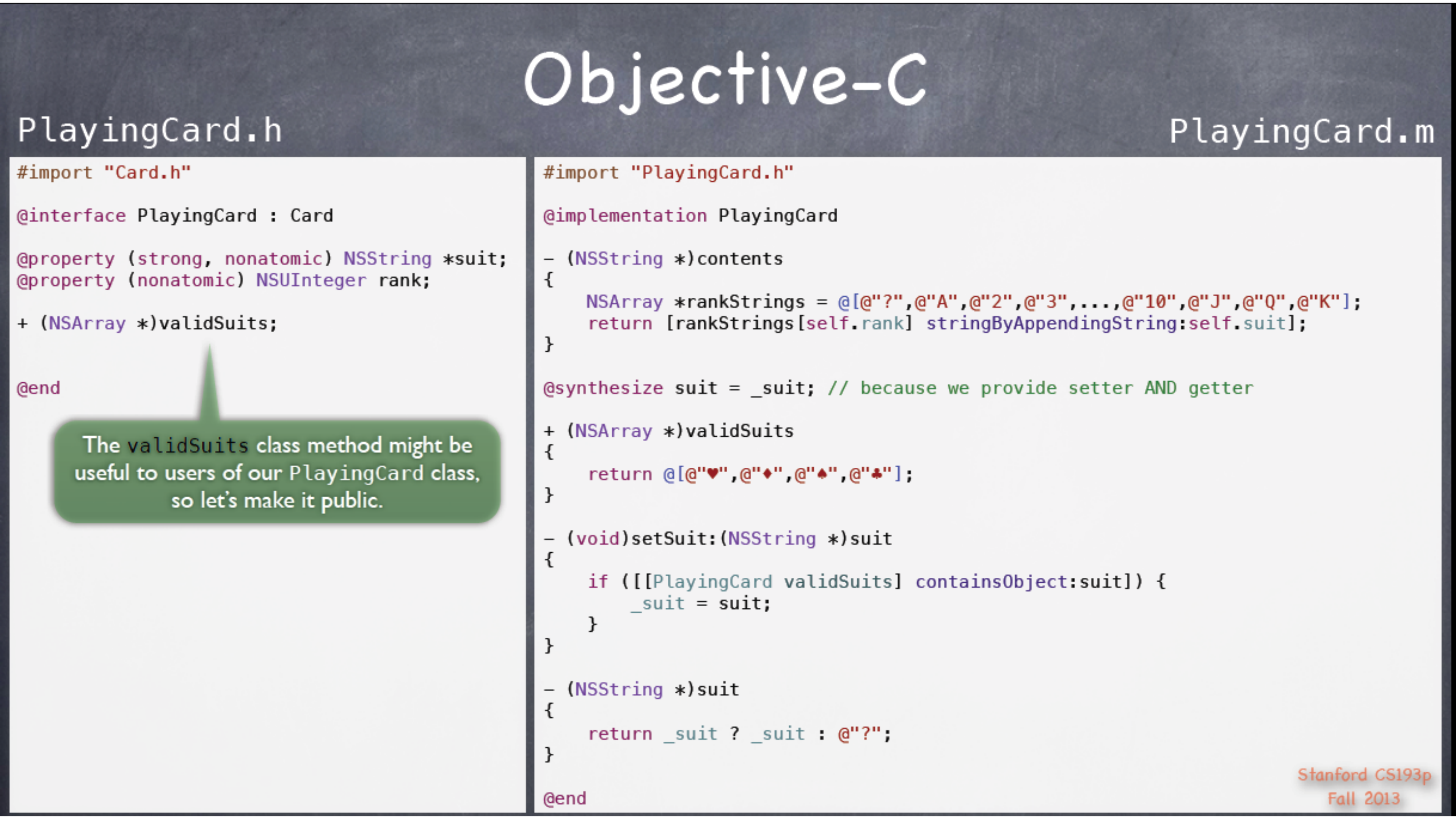
Now let's invoke our new class method here.

It'd probably be instructive to go back and look at the invocation of the NSString class method stringWithFormat: a few slides ago.

Also, make sure you understand that stringByAppendingString: above is not a class method, it is an instance method.

Stanford CS193p
Fall 2013

You Can Make A Class Method Public



Objective-C

PlayingCard.h

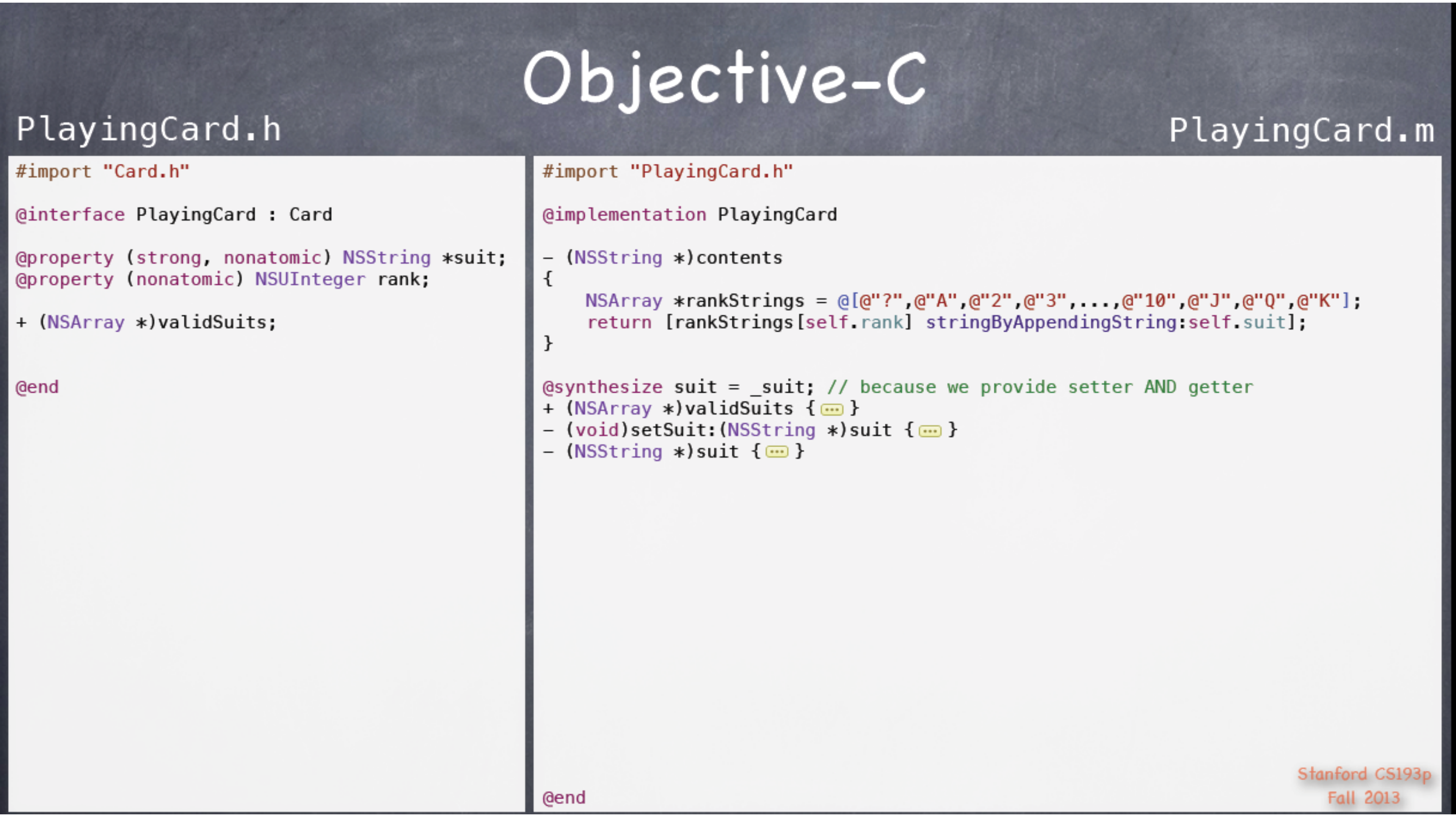
```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;

+ (NSArray *)validSuits;


@end
```

The validSuits class method might be useful to users of our PlayingCard class, so let's make it public.

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter

+ (NSArray *)validSuits
{
    return @[@"♥",@"♦",@"♠",@"♣"];
}

- (void)setSuit:(NSString *)suit
{
    if ([[PlayingCard validSuits] containsObject:suit]) {
        _suit = suit;
    }
}

- (NSString *)suit
{
    return _suit ? _suit : @"?";
}

@end
```

Collapse Methods To Make Room



Objective-C

PlayingCard.h

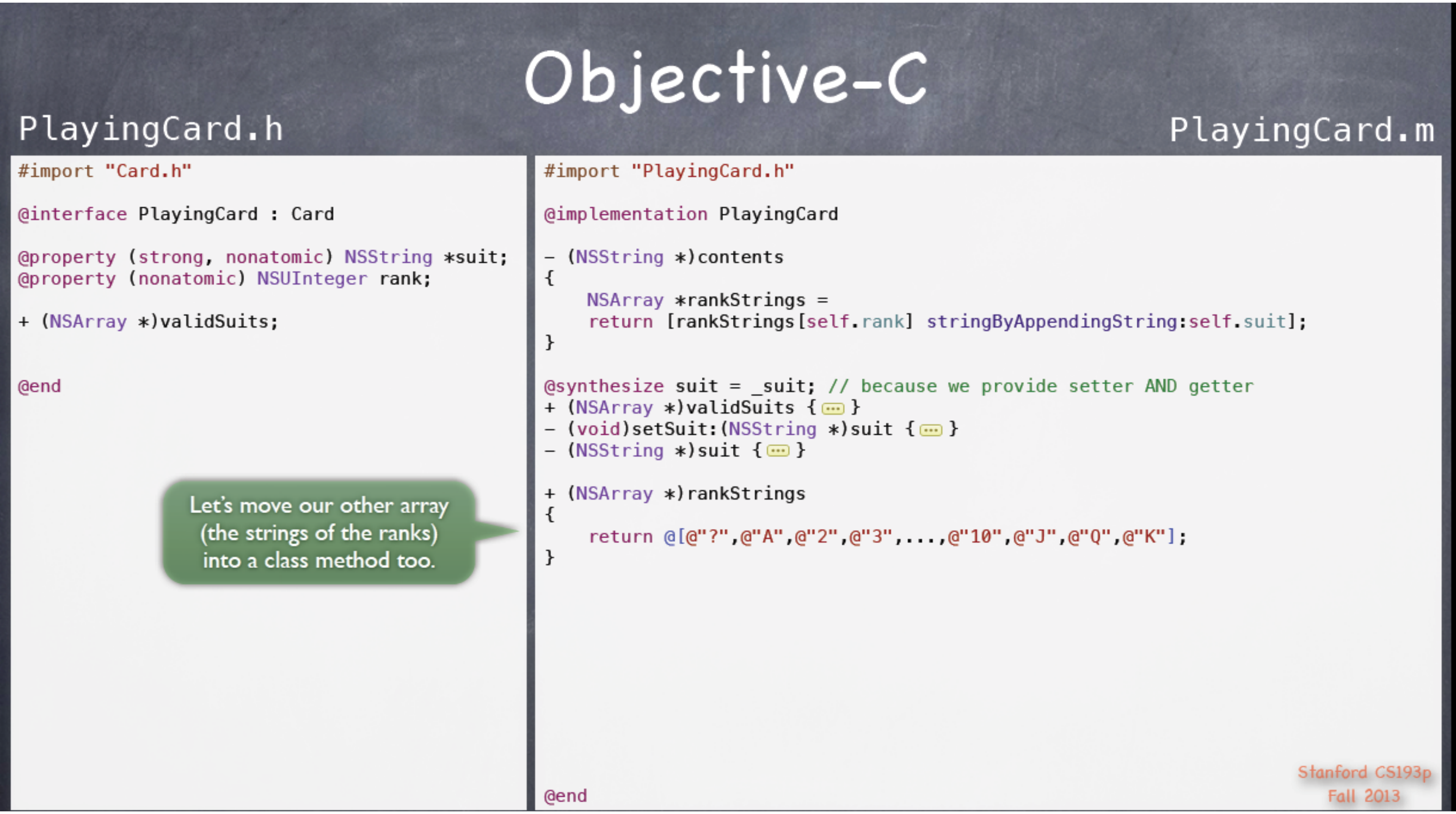
```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;

+ (NSArray *)validSuits;


@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter
+ (NSArray *)validSuits {⋯}
- (void)setSuit:(NSString *)suit {⋯}
- (NSString *)suit {⋯}















@end
```

Stanford CS193p
Fall 2013

Our Second Class Method



Objective-C

PlayingCard.h

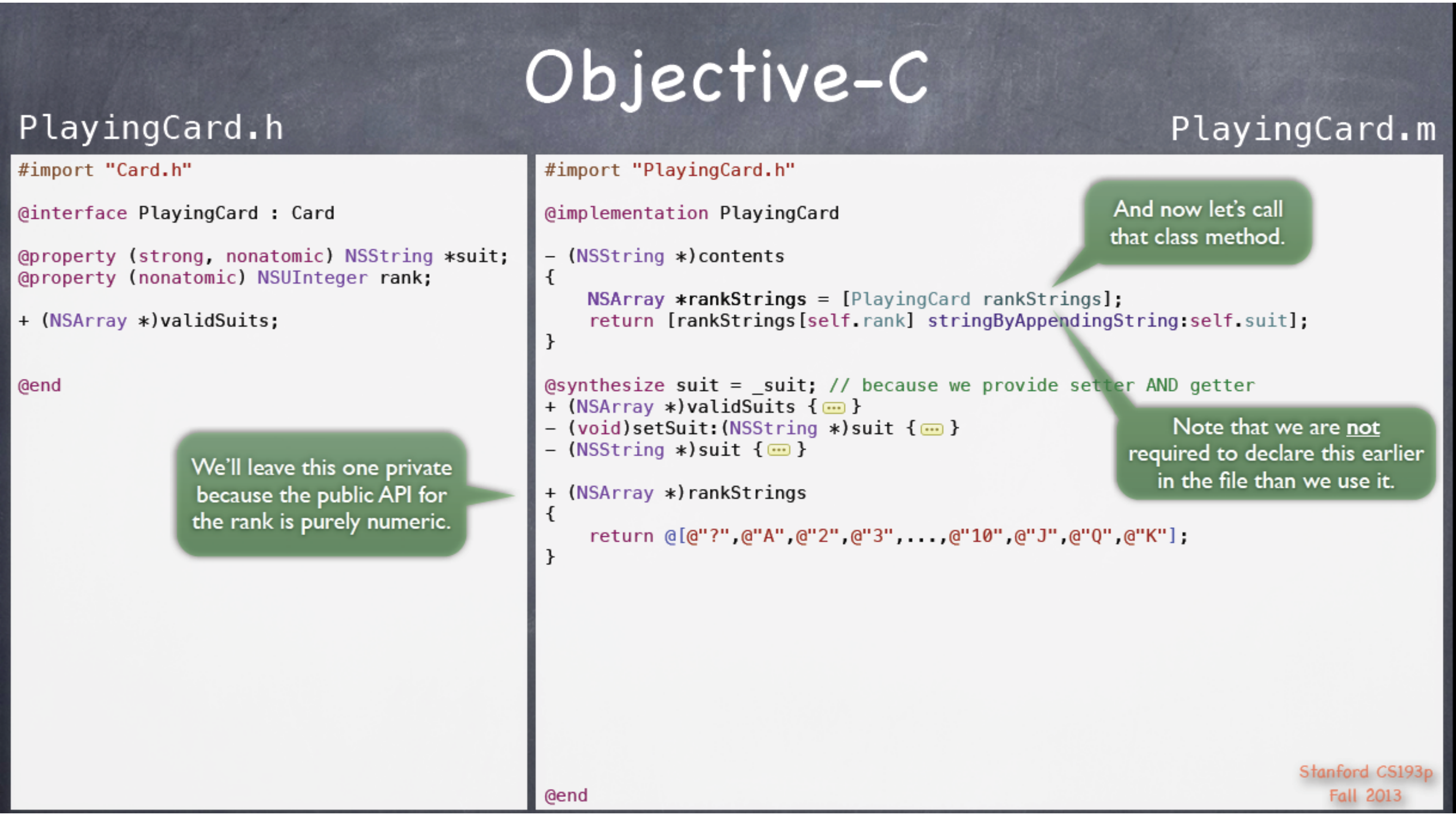
```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;

+ (NSArray *)validSuits;


@end
```

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings =
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter
+ (NSArray *)validSuits {…}
- (void)setSuit:(NSString *)suit {…}
- (NSString *)suit {…}

+ (NSArray *)rankStrings
{
    return @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
}







@end
```

Let's move our other array (the strings of the ranks) into a class method too.

Invoke Our New Class Method



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;

+ (NSArray *)validSuits;


@end
```

We'll leave this one private because the public API for the rank is purely numeric.

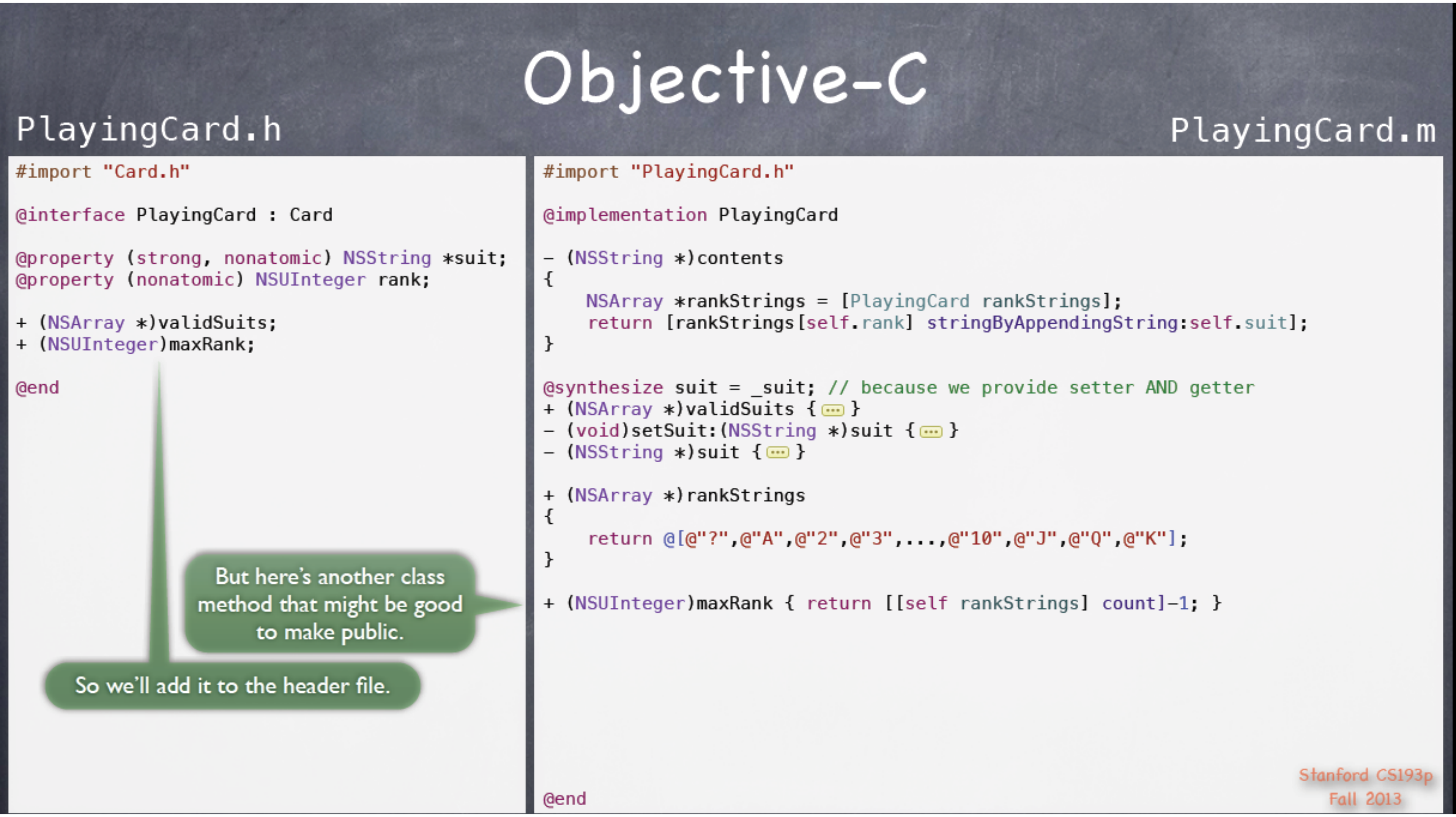
PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = [PlayingCard rankStrings];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter
+ (NSArray *)validSuits {…}
- (void)setSuit:(NSString *)suit {…}
- (NSString *)suit {…}

+ (NSArray *)rankStrings
{
    return @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
}




@end
```

And now let's call that class method.

Note that we are not required to declare this earlier in the file than we use it.

Our Third Class Method Is Also Public



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;

+ (NSArray *)validSuits;
+ (NSUInteger)maxRank;

@end
```

But here's another class method that might be good to make public.

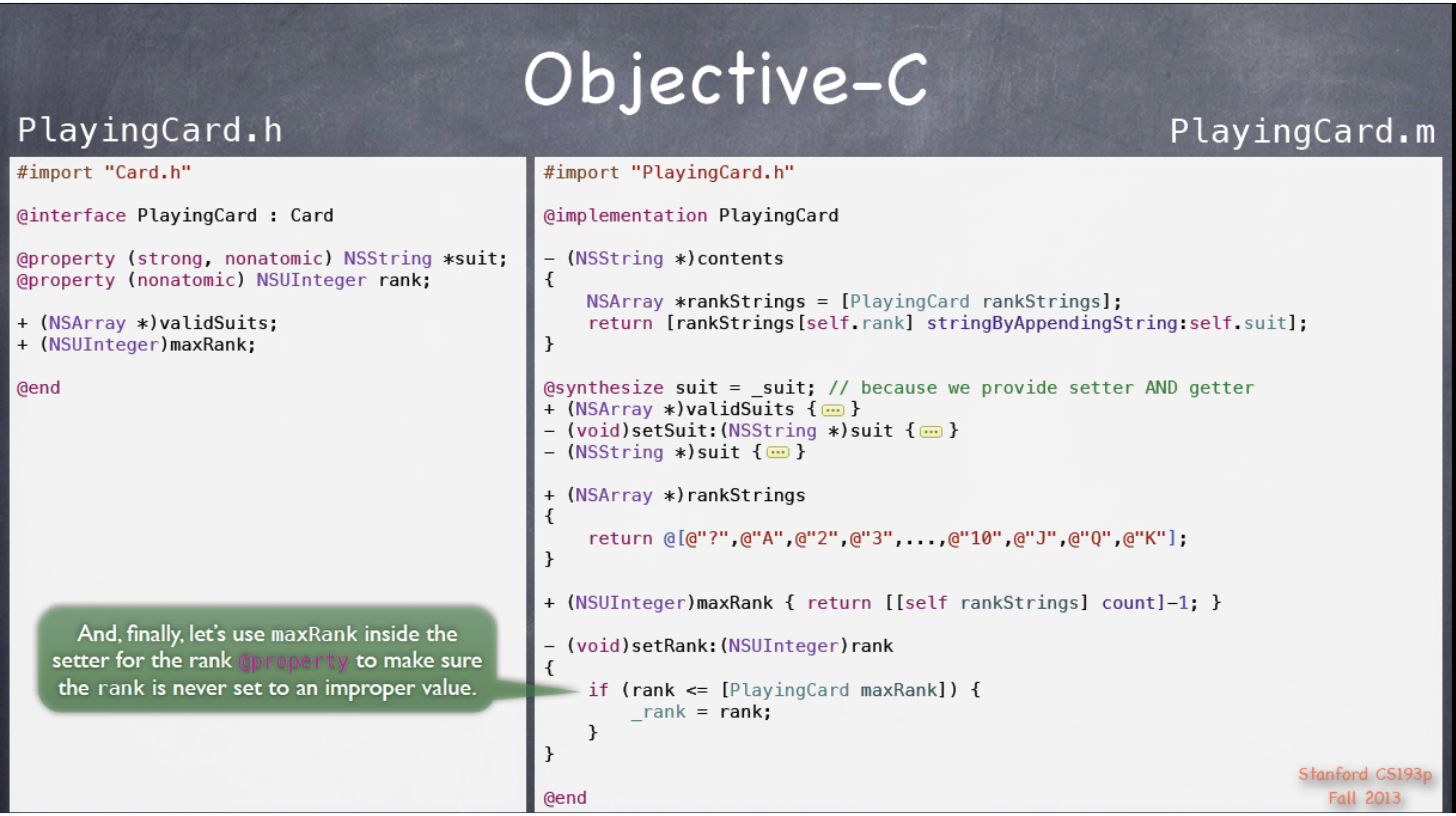
So we'll add it to the header file.

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = [PlayingCard rankStrings];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter
+ (NSArray *)validSuits {…}
- (void)setSuit:(NSString *)suit {…}
- (NSString *)suit {…}

+ (NSArray *)rankStrings
{
    return @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
}

+ (NSUInteger)maxRank { return [[self rankStrings] count]-1; }

@end
```

Stanford CS193p
Fall 2013

Use Our New Class Method In The rank Setter



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;

+ (NSArray *)validSuits;
+ (NSUInteger)maxRank;

@end
```

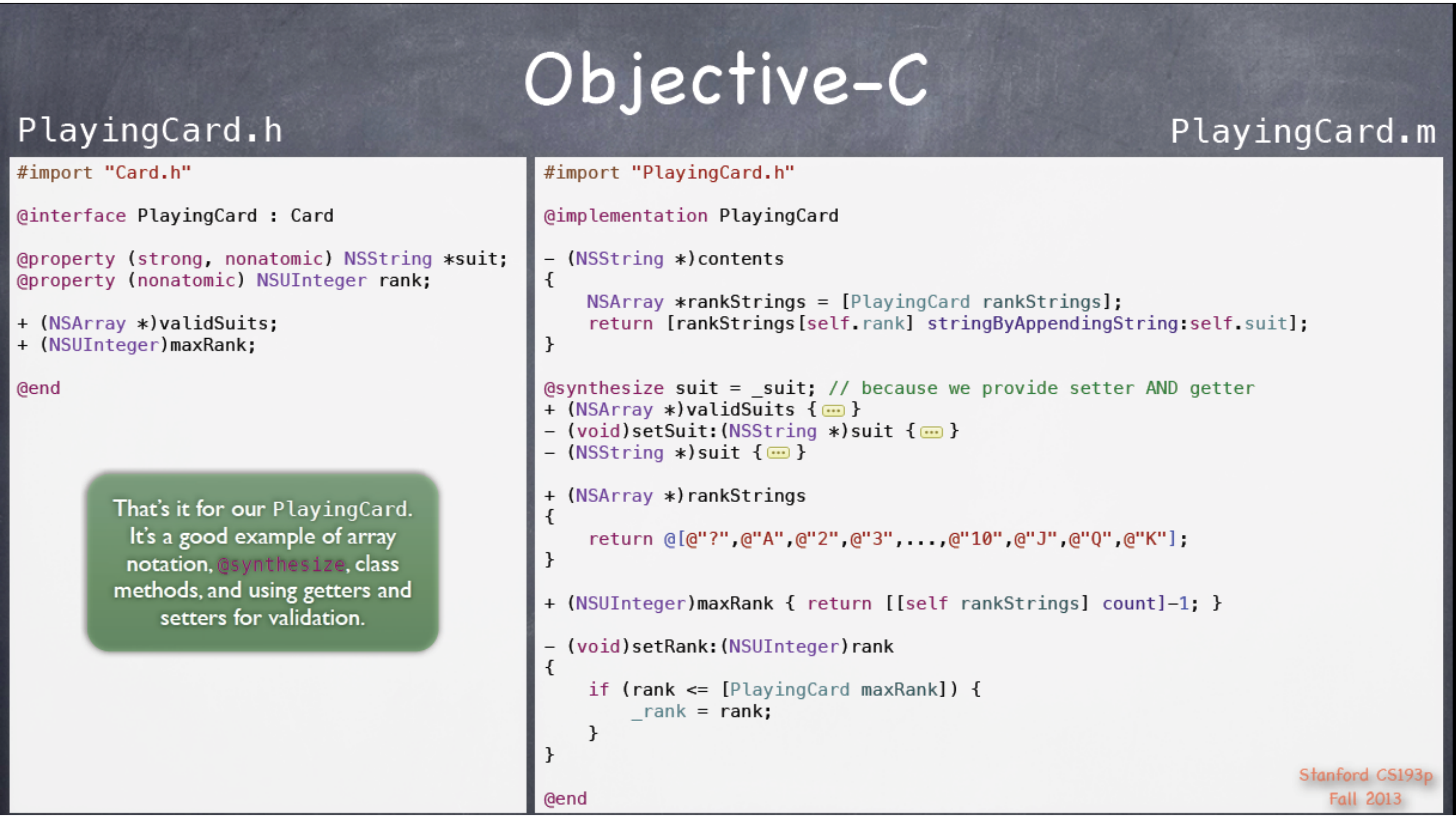
PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = [PlayingCard rankStrings];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter
+ (NSArray *)validSuits {…}
- (void)setSuit:(NSString *)suit {…}
- (NSString *)suit {…}

+ (NSArray *)rankStrings
{
    return @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
}

+ (NSUInteger)maxRank { return [[self rankStrings] count]-1; }

- (void)setRank:(NSUInteger)rank
{
    if (rank <= [PlayingCard maxRank]) {
        _rank = rank;
    }
}

@end
```

And, finally, let's use maxRank inside the setter for the rank @property to make sure the rank is never set to an improper value.

Stanford CS193p
Fall 2013

That's It For PlayingCard



Objective-C

PlayingCard.h

```
#import "Card.h"

@interface PlayingCard : Card

@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;

+ (NSArray *)validSuits;
+ (NSUInteger)maxRank;

@end
```

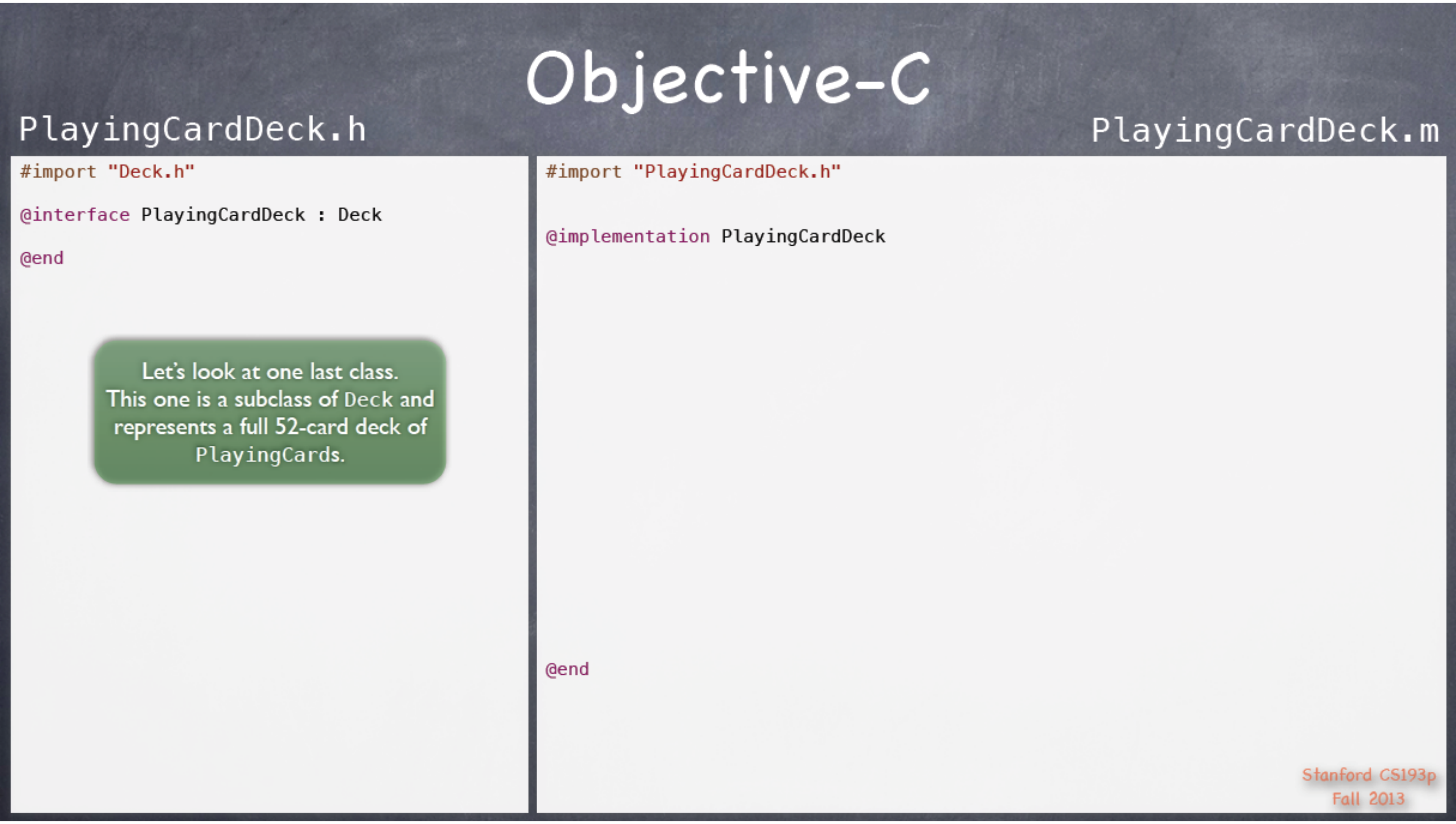
That's it for our PlayingCard. It's a good example of array notation, @synthesize, class methods, and using getters and setters for validation.

PlayingCard.m

```
#import "PlayingCard.h"

@implementation PlayingCard

- (NSString *)contents
{
    NSArray *rankStrings = [PlayingCard rankStrings];
    return [rankStrings[self.rank] stringByAppendingString:self.suit];
}

@synthesize suit = _suit; // because we provide setter AND getter
+ (NSArray *)validSuits {…}
- (void)setSuit:(NSString *)suit {…}
- (NSString *)suit {…}

+ (NSArray *)rankStrings
{
    return @[@"?",@"A",@"2",@"3",...,@"10",@"J",@"Q",@"K"];
}

+ (NSUInteger)maxRank { return [[self rankStrings] count]-1; }

- (void)setRank:(NSUInteger)rank
{
    if (rank <= [PlayingCard maxRank]) {
        _rank = rank;
    }
}

@end
```

Stanford CS193p
Fall 2013

PlayingCardDeck Inherits From Deck



Objective-C

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

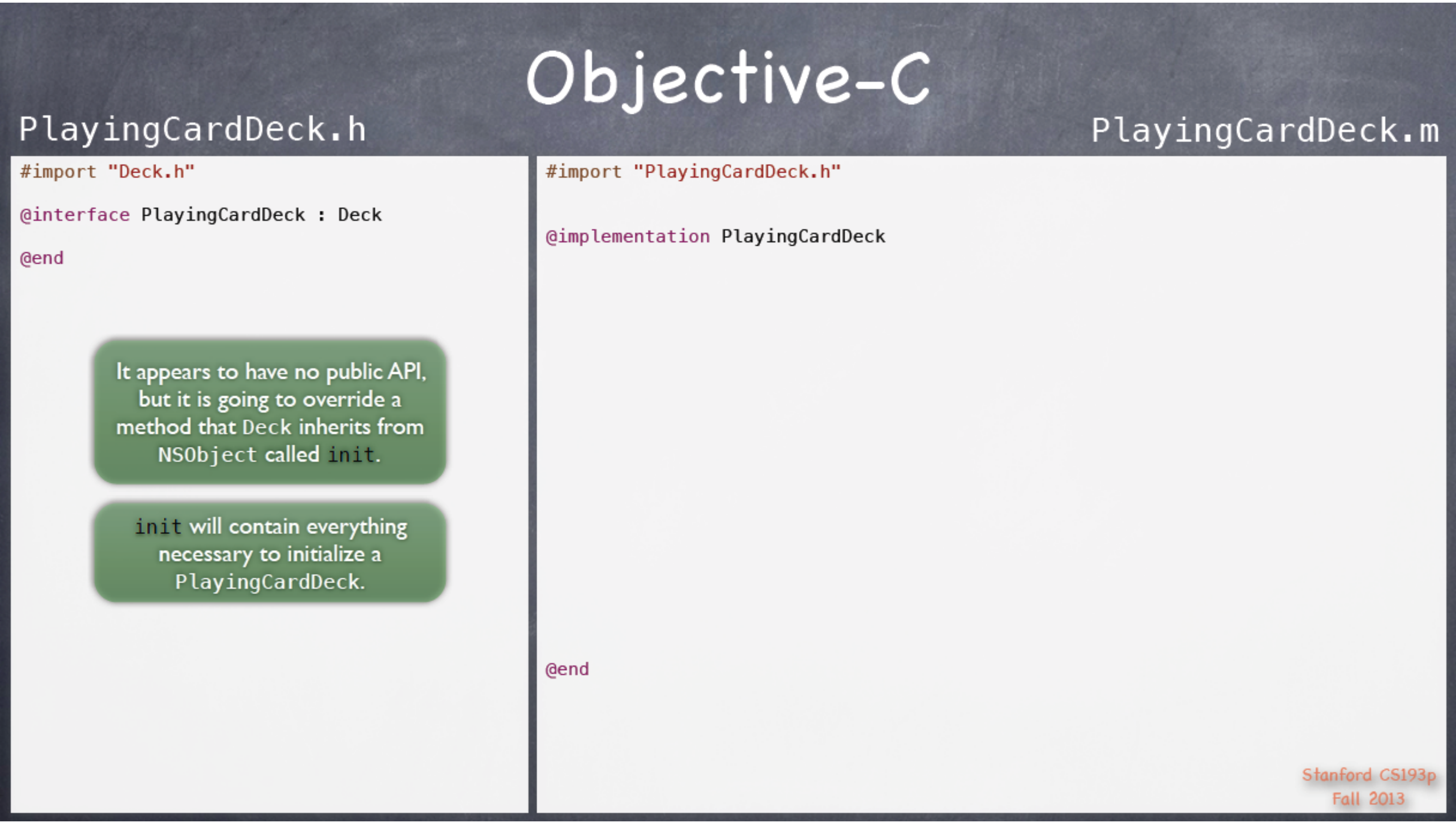
Let's look at one last class. This one is a subclass of Deck and represents a full 52-card deck of PlayingCards.

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck









@end
```

Stanford CS193p
Fall 2013

PlayingCardDeck Overrides init



Objective-C

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

It appears to have no public API, but it is going to override a method that Deck inherits from NSObject called init.

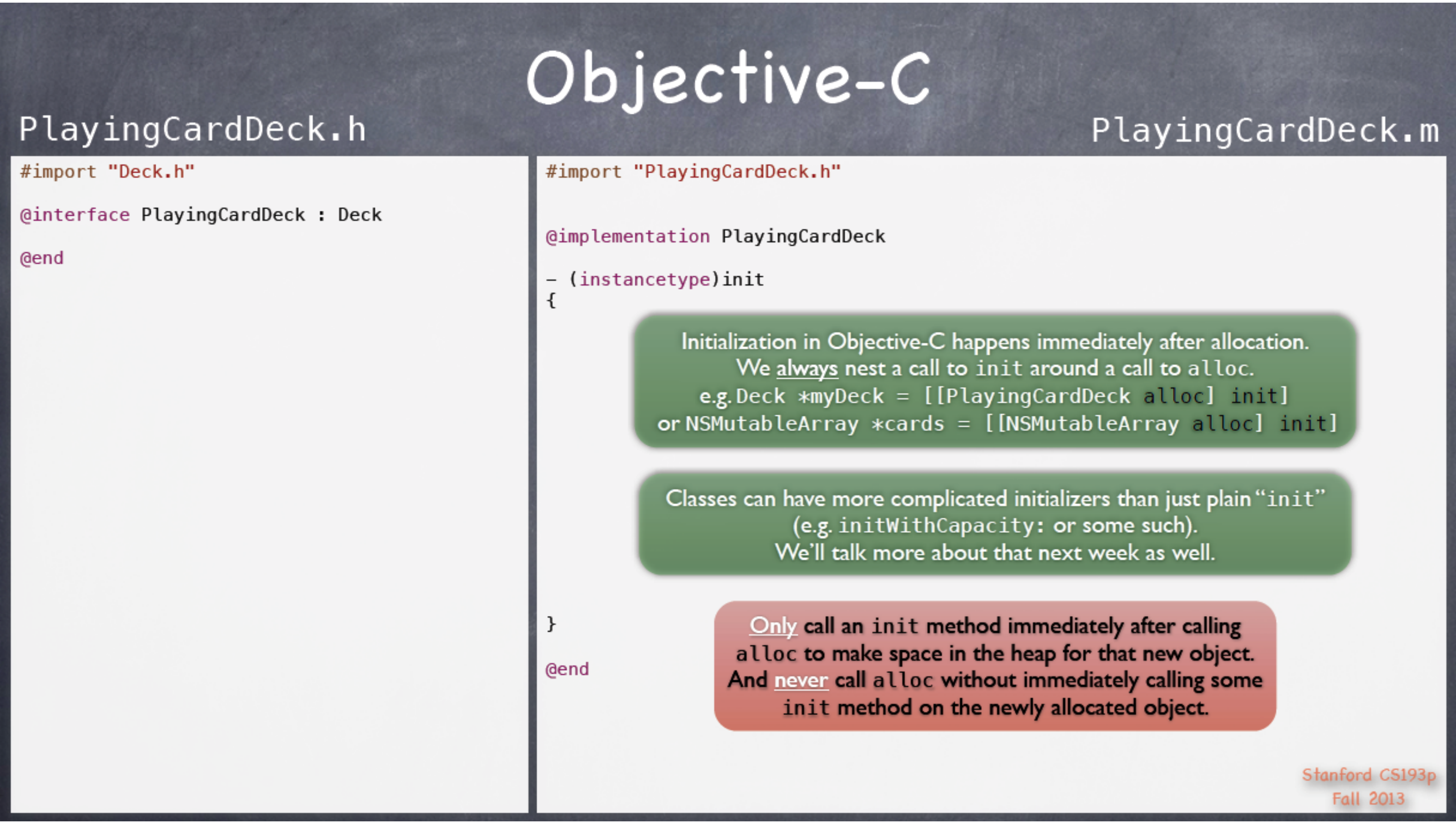
init will contain everything necessary to initialize a PlayingCardDeck.

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck














@end
```

Stanford CS193p
Fall 2013

init



Objective-C

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

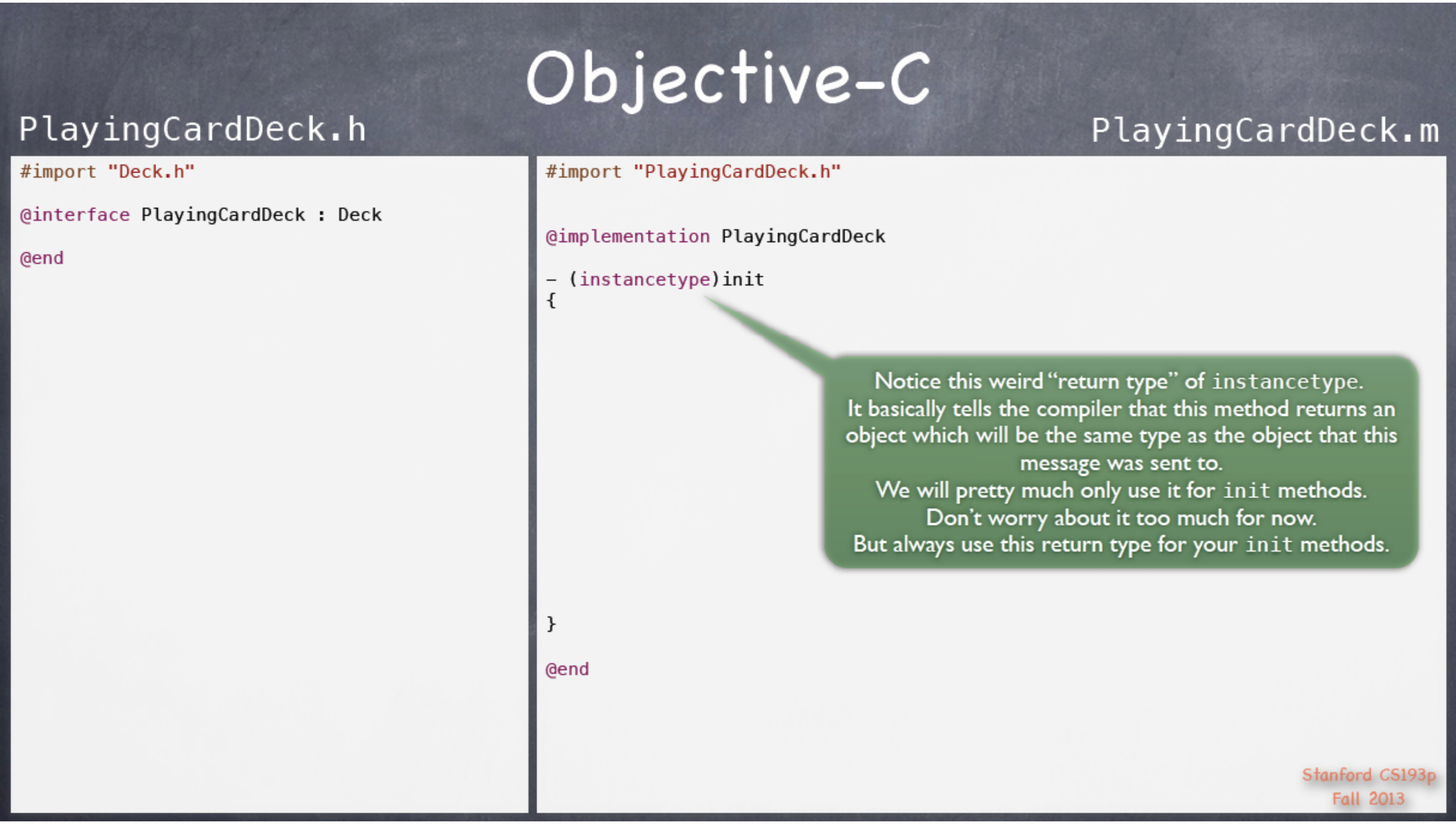
```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck

- (instancetype)init
{

}

@end
```

Initialization in Objective-C happens immediately after allocation.
We <u>always</u> nest a call to `init` around a call to `alloc`.
e.g. `Deck *myDeck = [[PlayingCardDeck alloc] init]`
or `NSMutableArray *cards = [[NSMutableArray alloc] init]`

Classes can have more complicated initializers than just plain "`init`"
(e.g. `initWithCapacity:` or some such).
We'll talk more about that next week as well.

<u>Only</u> call an `init` method immediately after calling `alloc` to make space in the heap for that new object.
And <u>never</u> call `alloc` without immediately calling some `init` method on the newly allocated object.

instancetype



Objective-C

PlayingCardDeck.h

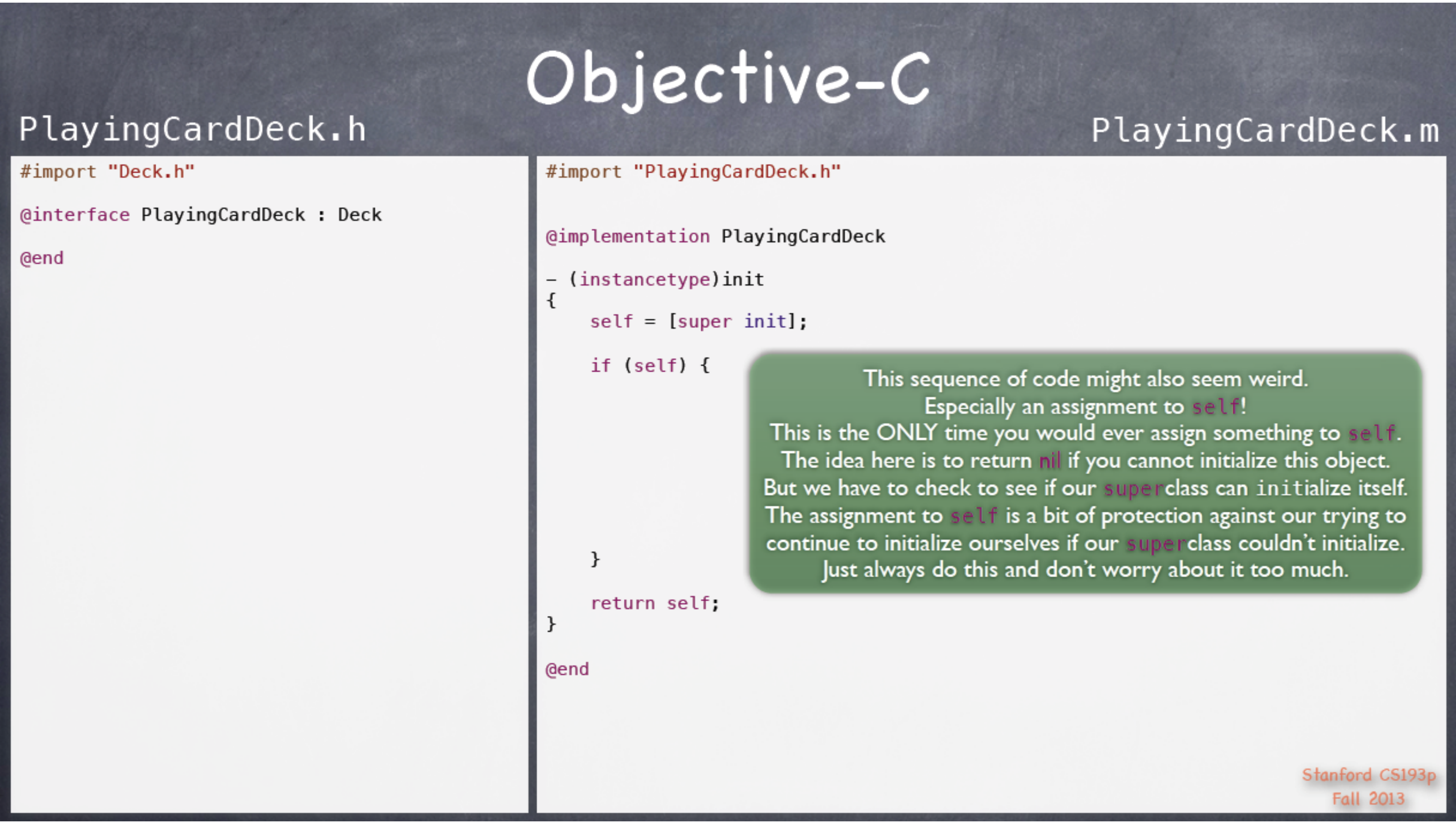
```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck

- (instancetype)init
{










}

@end
```

Notice this weird "return type" of instancetype.
It basically tells the compiler that this method returns an object which will be the same type as the object that this message was sent to.
We will pretty much only use it for init methods.
Don't worry about it too much for now.
But always use this return type for your init methods.

Only Time You Assign To self



Objective-C

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck

- (instancetype)init
{
    self = [super init];

    if (self) {








    }

    return self;
}

@end
```

This sequence of code might also seem weird.
Especially an assignment to self!
This is the ONLY time you would ever assign something to self.
The idea here is to return nil if you cannot initialize this object.
But we have to check to see if our superclass can initialize itself.
The assignment to self is a bit of protection against our trying to continue to initialize ourselves if our superclass couldn't initialize.
Just always do this and don't worry about it too much.

Stanford CS193p
Fall 2013

Objective-C

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck

- (instancetype)init
{
    self = [super init];

    if (self) {









    }

    return self;
}

@end
```

Sending a message to super is how we send a message to ourselves, but use our superclass's implementation instead of our own.
Standard object-oriented stuff.

Iterate Through Suits & Ranks

Objective-C

PlayingCardDeck.h

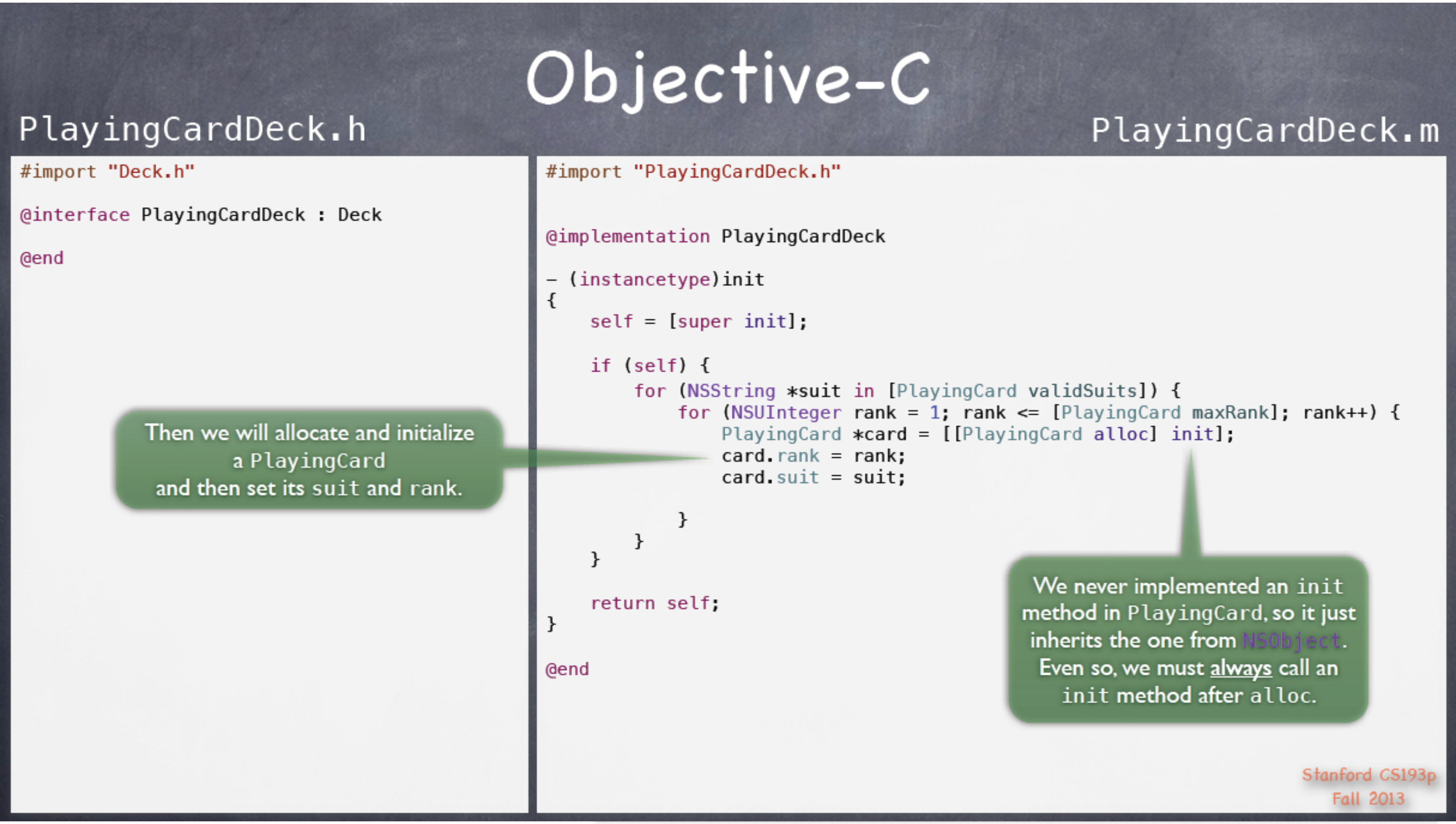
```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

The implementation of init is quite simple. We'll just iterate through all the suits and then through all the ranks in that suit ...

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck

- (instancetype)init
{
    self = [super init];

    if (self) {
        for (NSString *suit in [PlayingCard validSuits]) {
            for (NSUInteger rank = 1; rank <= [PlayingCard maxRank]; rank++) {




            }
        }
    }

    return self;
}

@end
```

alloc & init A PlayingCard



Objective-C

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

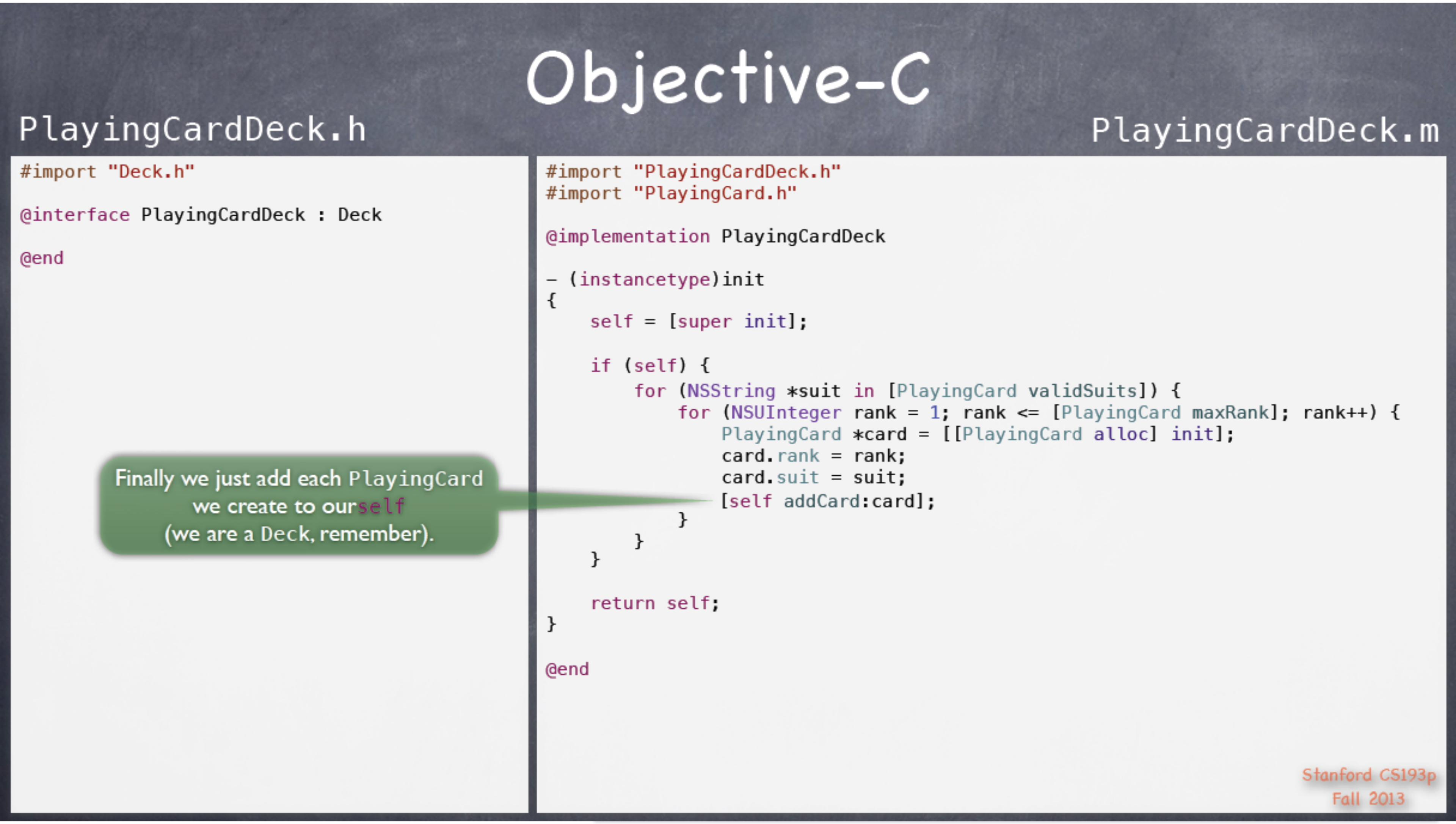
```
#import "PlayingCardDeck.h"


@implementation PlayingCardDeck

- (instancetype)init
{
    self = [super init];

    if (self) {
        for (NSString *suit in [PlayingCard validSuits]) {
            for (NSUInteger rank = 1; rank <= [PlayingCard maxRank]; rank++) {
                PlayingCard *card = [[PlayingCard alloc] init];
                card.rank = rank;
                card.suit = suit;

            }
        }
    }

    return self;
}

@end
```

Then we will allocate and initialize a PlayingCard and then set its suit and rank.

We never implemented an init method in PlayingCard, so it just inherits the one from NSObject. Even so, we must always call an init method after alloc.

Stanford CS193p
Fall 2013

Add The Card To The Deck



Objective-C

PlayingCardDeck.h

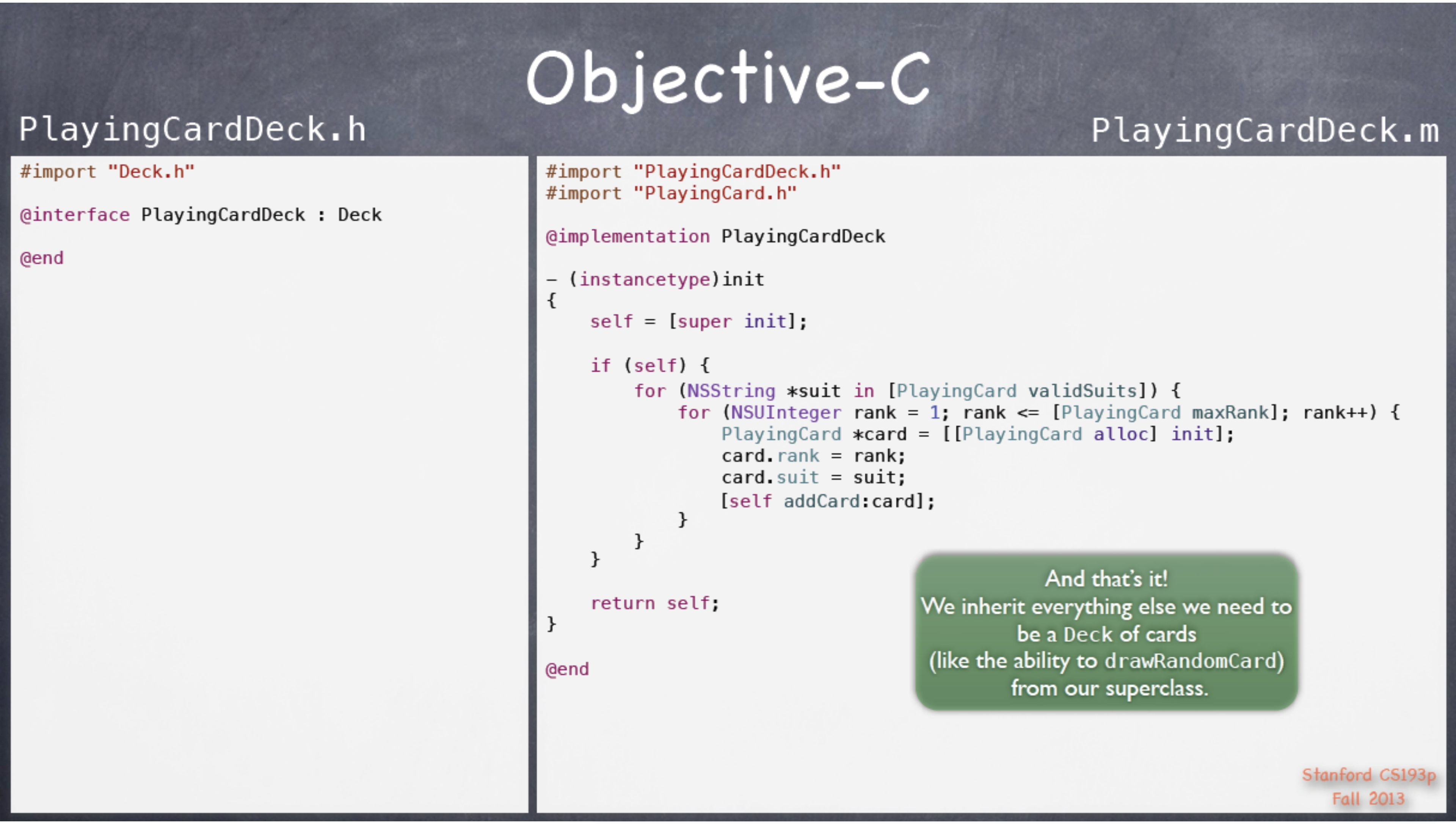
```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"
#import "PlayingCard.h"

@implementation PlayingCardDeck

- (instancetype)init
{
    self = [super init];

    if (self) {
        for (NSString *suit in [PlayingCard validSuits]) {
            for (NSUInteger rank = 1; rank <= [PlayingCard maxRank]; rank++) {
                PlayingCard *card = [[PlayingCard alloc] init];
                card.rank = rank;
                card.suit = suit;
                [self addCard:card];
            }
        }
    }

    return self;
}

@end
```

Finally we just add each PlayingCard we create to ourself (we are a Deck, remember).

That's It For PlayingCardDeck



Objective-C

PlayingCardDeck.h

```
#import "Deck.h"

@interface PlayingCardDeck : Deck

@end
```

PlayingCardDeck.m

```
#import "PlayingCardDeck.h"
#import "PlayingCard.h"

@implementation PlayingCardDeck

- (instancetype)init
{
    self = [super init];

    if (self) {
        for (NSString *suit in [PlayingCard validSuits]) {
            for (NSUInteger rank = 1; rank <= [PlayingCard maxRank]; rank++) {
                PlayingCard *card = [[PlayingCard alloc] init];
                card.rank = rank;
                card.suit = suit;
                [self addCard:card];
            }
        }
    }

    return self;
}

@end
```

And that's it!
We inherit everything else we need to be a Deck of cards
(like the ability to drawRandomCard)
from our superclass.

Key References

All slides in this presentation were imported from:

CS193P: iPhone Application Development.
This course was taught at Stanford University, Fall 2013, by Paul Hegarty.

The course and all of its accompanying material is available on iTunes U.