Creating Objects

- Most of the time, we create objects with alloc and init...

```
NSMutableArray *cards = [[NSMutableArray alloc] init];
CardMatchingGame *game = [[CardMatchingGame alloc] initWithCardCount:12 usingDeck:d];
```

- Or with class methods

```
NSString's + (id)stringWithFormat:(NSString *)format, ...
NSString *moltuae = [NSString stringWithFormat:@"%d", 42];
UIButton's + (id)buttonWithType:(UIButtonType)buttonType;
NSMutableArray's + (id)arrayWithCapacity:(int)count;
NSArray's + (id)arrayWithObject:(id)anObject;
```

- Sometimes both a class creator method and init method exist

`[NSString stringWithFormat:...]` same as `[[NSString alloc] initWithFormat:...]`

Don't be disturbed by this. Using either version is fine.

iOS seems to be moving more toward the alloc/init versions with new API, but is mostly neutral.

Рис. 1

Creating Objects

- You can also ask other objects to create new objects for you

  NSString's `- (NSString *)stringByAppendingString:(NSString *)otherString;`
  NSArray's `- (NSString *)componentsJoinedByString:(NSString *)separator;`
  NSString's & NSArray's `- (id)mutableCopy;`

- But not all objects given out by other objects are newly created

  NSArray's `- (id)lastObject;`
  NSArray's `- (id)objectAtIndex:(int)index;`
  Unless the method has the word "copy" in it, if the object already exists, you get a pointer to it.
  If the object does not already exist (like the first 2 examples at the top), then you're creating.

Рис. 2

nil

- Sending messages to `nil` is (mostly) okay. No code executed.

  If the method returns a value, it will return zero.

  ```
  int i = [obj methodWhichReturnsAnInt]; // i will be zero if obj is nil
  ```

  It is absolutely fine to depend on this and write code that uses this (don't get too cute, though).

  But be careful if the method returns a C struct. Return value is undefined.

  ```
  CGPoint p = [obj getLocation];  // p will have an undefined value if obj is nil
  ```

Рис. 3

Dynamic Binding

- Objective-C has an important type called `id`
  - It means "pointer to an object of unknown/unspecified" type.
  - `id myObject;`
  - Really all object pointers (e.g. `NSString *`) are treated like `id` at runtime.
  - But at compile time, if you type something `NSString *` instead of `id`, the compiler can help you.
  - It can find bugs and suggest what methods would be appropriate to send to it, etc.
  - If you type something using `id`, the compiler can't help very much because it doesn't know much.
  - Figuring out the code to execute when a message is sent at runtime is called "dynamic binding."
- Is it safe?
  - Treating all object pointers as "pointer to unknown type" at runtime seems dangerous, right?
  - What stops you from sending a message to an object that it doesn't understand?
  - Nothing. And your program crashes if you do so. Oh my, Objective-C programs must crash a lot!
  - Not really.
  - Because we mostly use static typing (e.g. `NSString *`) and the compiler is really smart.

Рис. 4

Dynamic Binding

- Static typing

```
NSString *s = @"x";   // "statically" typed (compiler will warn if s is sent non-NSString messges).
id obj = s;           // not statically typed, but perfectly legal; compiler can't catch [obj rank]
NSArray *a = obj;     // also legal, but obviously could lead to some big trouble!
```

Compiler will not complain about assignments between an id and a statically typed variable.

Sometimes you are silently doing this. You have already done so!

```
- (int)match:(NSArray *)otherCards
{
    …
    PlayingCard *otherCard = [otherCards firstObject]; // firstObject returns id!
    …
}
```

Never use "id *" by the way (that would mean "a pointer to a pointer to an object").

Рис. 5

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];
```

No compiler warning.
Perfectly legal since s "isa" Vehicle.
Normal object-oriented stuff here.

Рис. 6

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];
```

Compiler warning!

Would not crash at runtime though.
But only because we know v is a Ship.
Compiler only knows v is a Vehicle.

Рис. 7

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
```

No compiler warning.

The compiler knows that the method `shoot` exists,
so it's not impossible that `obj` might respond to it.
But we have not typed `obj` enough for the compiler to be sure it's wrong.
So no warning.
Might crash at runtime if `obj` is not a `Ship`
(or an object of some other class that implements a `shoot` method).

Рис. 8

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];
```

Compiler warning!

Compiler has never heard of this method.
Therefore it's pretty sure `obj` will not respond to it.

Рис. 9

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
```

Compiler warning.
The compiler knows that `NSString` objects do not respond to `shoot`.
Guaranteed crash at runtime.

Рис. 10

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
```

No compiler warning.
We are "casting" here.
The compiler thinks we know what we're doing.

Рис. 11

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];
```

No compiler warning!
We've forced the compiler to think that the NSString is a Ship.
"All's well," the compiler thinks.
Guaranteed crash at runtime.

Рис. 12

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];
[(id)hello shoot];
```

No compiler warning!
We've forced the compiler to ignore the object type by "casting" in line. "All's well," the compiler thinks. Guaranteed crash at runtime.

Рис. 13

Introspection

- All objects that inherit from `NSObject` know these methods ...
  - `isKindOfClass:` returns whether an object is that kind of class (inheritance included)
  - `isMemberOfClass:` returns whether an object is that kind of class (no inheritance)
  - `respondsToSelector:` returns whether an object responds to a given method
  - It calculates the answer to these questions at runtime (i.e. at the instant you send them).
- Arguments to these methods are a little tricky
  - Class testing methods take a `Class`
  - You get a `Class` by sending the class method `class` to a class (not the instance method `class`).

```
if ([obj isKindOfClass:[NSString class]]) {
    NSString *s = [(NSString *)obj stringByAppendingString:@"xyzzy"];
}
```

Рис. 14

Introspection

- Method testing methods take a selector (SEL)

  Special @selector() directive turns the name of a method into a selector

```
if ([obj respondsToSelector:@selector(shoot)]) {
    [obj shoot];
} else if ([obj respondsToSelector:@selector(shootAt:)]) {
    [obj shootAt:target];
}
```

- SEL is the Objective-C "type" for a selector

```
SEL shootSelector = @selector(shoot);
SEL shootAtSelector = @selector(shootAt:);
SEL moveToSelector = @selector(moveTo:withPenColor:);
```

Рис. 15

Introspection

- If you have a SEL, you can also ask an object to perform it …

Using the performSelector: or performSelector:withObject: methods in NSObject

```
[obj performSelector:shootSelector];
[obj performSelector:shootAtSelector withObject:coordinate];
```

Using makeObjectsPerformSelector: methods in NSArray

```
[array makeObjectsPerformSelector:shootSelector]; // cool, huh?
[array makeObjectsPerformSelector:shootAtSelector withObject:target]; // target is an id
```

In UIButton, - (void)addTarget:(id)anObject action:(SEL)action …;

```
[button addTarget:self action:@selector(digitPressed:) …];
```

Рис. 16

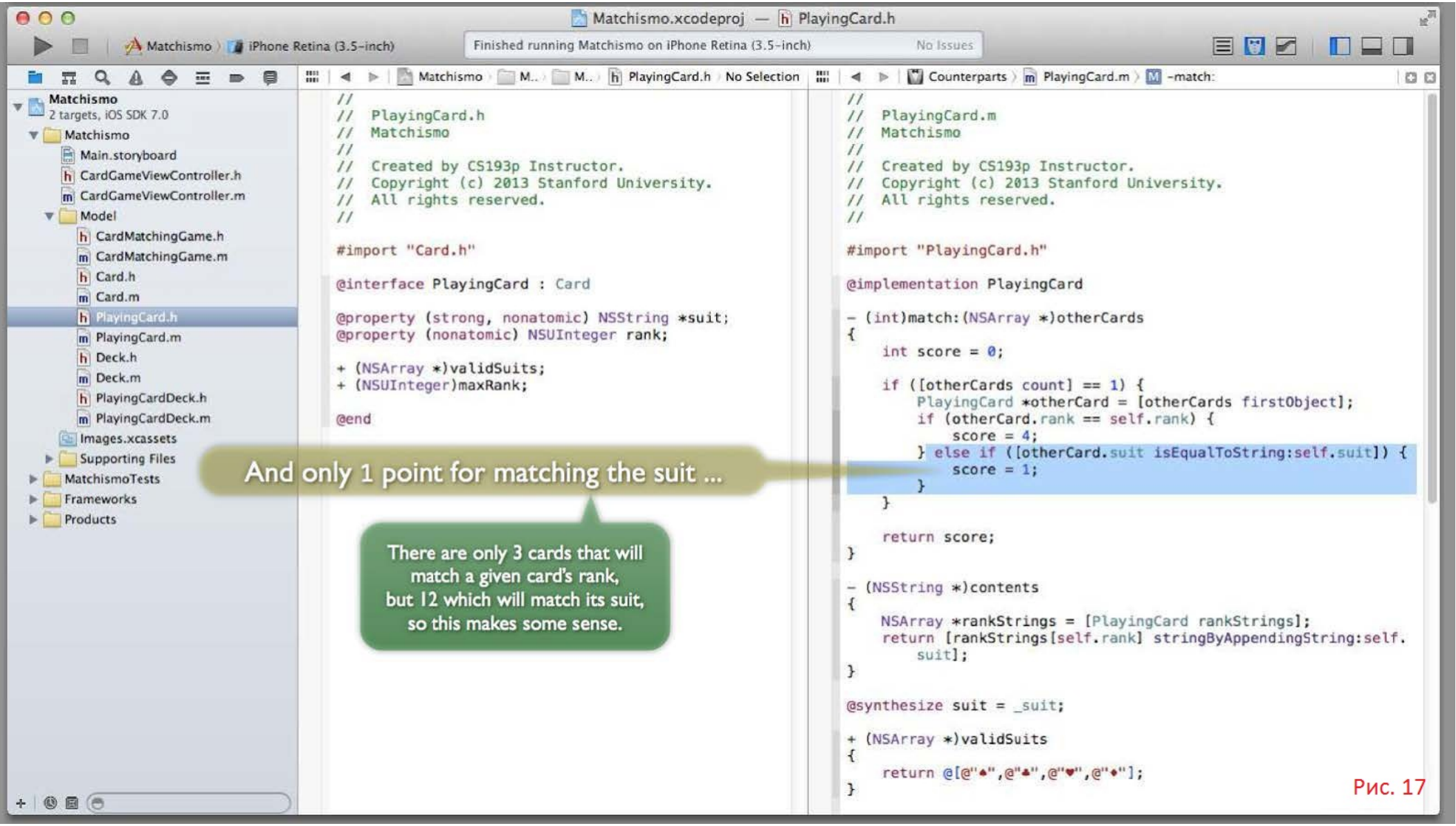

Matchismo.xcodeproj — PlayingCard.h
Matchismo › iPhone Retina (3.5-inch)
Finished running Matchismo on iPhone Retina (3.5-inch)
No Issues
Matchismo
2 targets, iOS SDK 7.0
Matchismo
Main.storyboard
CardGameViewController.h
CardGameViewController.m
Model
CardMatchingGame.h
CardMatchingGame.m
Card.h
Card.m
PlayingCard.h
PlayingCard.m
Deck.h
Deck.m
PlayingCardDeck.h
PlayingCardDeck.m
Images.xcassets
Supporting Files
MatchismoTests
Frameworks
Products
Matchismo › M.. › M.. › PlayingCard.h › No Selection
//
// PlayingCard.h
// Matchismo
//
// Created by CS193p Instructor.
// Copyright (c) 2013 Stanford University.
// All rights reserved.
//
#import "Card.h"
@interface PlayingCard : Card
@property (strong, nonatomic) NSString *suit;
@property (nonatomic) NSUInteger rank;
+ (NSArray *)validSuits;
+ (NSUInteger)maxRank;
@end
Counterparts › PlayingCard.m › -match:
//
// PlayingCard.m
// Matchismo
//
// Created by CS193p Instructor.
// Copyright (c) 2013 Stanford University.
// All rights reserved.
//
#import "PlayingCard.h"
@implementation PlayingCard
- (int)match:(NSArray *)otherCards
{
int score = 0;
if ([otherCards count] == 1) {
PlayingCard *otherCard = [otherCards firstObject];
if (otherCard.rank == self.rank) {
score = 4;
} else if ([otherCard.suit isEqualToString:self.suit]) {
score = 1;
}
}
return score;
}
- (NSString *)contents
{
NSArray *rankStrings = [PlayingCard rankStrings];
return [rankStrings[self.rank] stringByAppendingString:self.
suit];
}
@synthesize suit = _suit;
+ (NSArray *)validSuits
{
return @[@"♠",@"♣",@"♥",@"♦"];
}
And only 1 point for matching the suit ...
There are only 3 cards that will match a given card's rank, but 12 which will match its suit, so this makes some sense.

Рис. 17

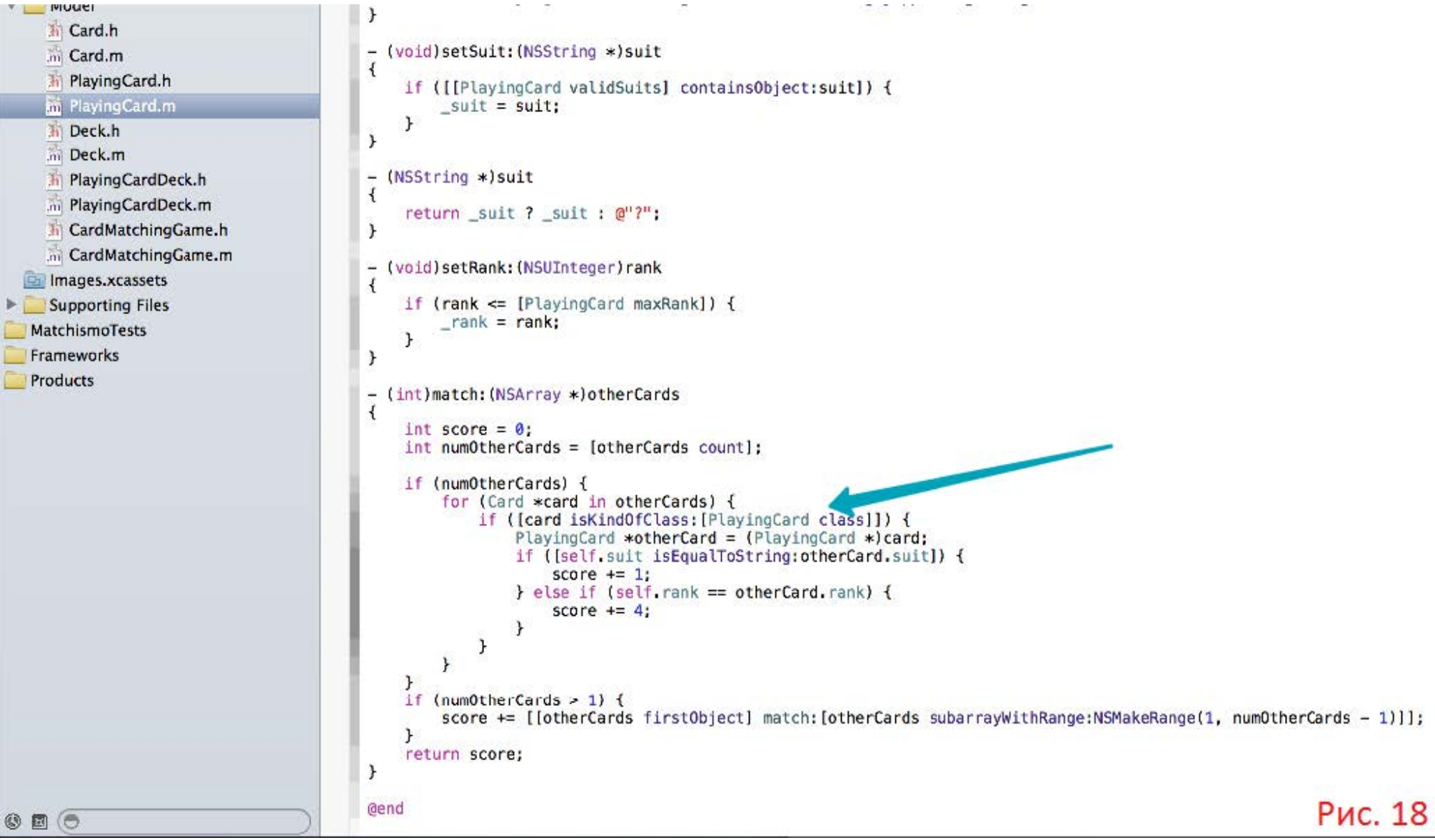

Model
Card.h
Card.m
PlayingCard.h
PlayingCard.m
Deck.h
Deck.m
PlayingCardDeck.h
PlayingCardDeck.m
CardMatchingGame.h
CardMatchingGame.m
Images.xcassets
Supporting Files
MatchismoTests
Frameworks
Products
}
- (void)setSuit:(NSString *)suit
{
if ([[PlayingCard validSuits] containsObject:suit]) {
_suit = suit;
}
}
- (NSString *)suit
{
return _suit ? _suit : @"?";
}
- (void)setRank:(NSUInteger)rank
{
if (rank <= [PlayingCard maxRank]) {
_rank = rank;
}
}
- (int)match:(NSArray *)otherCards
{
int score = 0;
int numOtherCards = [otherCards count];
if (numOtherCards) {
for (Card *card in otherCards) {
if ([card isKindOfClass:[PlayingCard class]]) {
PlayingCard *otherCard = (PlayingCard *)card;
if ([self.suit isEqualToString:otherCard.suit]) {
score += 1;
} else if (self.rank == otherCard.rank) {
score += 4;
}
}
}
}
if (numOtherCards > 1) {
score += [[otherCards firstObject] match:[otherCards subarrayWithRange:NSMakeRange(1, numOtherCards - 1)]];
}
return score;
}
@end

Рис. 18

Foundation Framework

- NSObject

  Base class for pretty much every object in the iOS SDK

  Implements introspection methods discussed earlier.

  `- (NSString *)description` is a useful method to override (it's `%@` in `NSLog()`).
  Example ... `NSLog(@"array contents are %@", myArray);`
  The `%@` is replaced with the results of invoking `[myArray description]`.

  Copying objects. This is an important concept to understand (why mutable vs. immutable?).

```
- (id)copy;          // not all objects implement mechanism (raises exception if not)
- (id)mutableCopy;   // not all objects implement mechanism (raises exception if not)
```

  It's not uncommon to have an array or dictionary and make a `mutableCopy` and modify that.
  Or to have a mutable array or dictionary and `copy` it to "freeze it" and make it immutable.
  Making copies of collection classes is very efficient, so don't sweat doing so.

Foundation Framework

- NSArray

  Ordered collection of objects.
  Immutable. That's right, once you create the array, you cannot add or remove objects.
  All objects in the array are held onto strongly.
  Usually created by manipulating other arrays or with @[].

  You already know these key methods ...

```
- (NSUInteger)count;
- (id)objectAtIndex:(NSUInteger)index; // crashes if index is out of bounds; returns id!
- (id)lastObject;  // returns nil (doesn't crash) if there are no objects in the array
- (id)firstObject;  // returns nil (doesn't crash) if there are no objects in the array
```

  But there are a lot of very interesting methods in this class. Examples ...

```
- (NSArray *)sortedArrayUsingSelector:(SEL)aSelector;
- (void)makeObjectsPerformSelector:(SEL)aSelector withObject:(id)selectorArgument;
- (NSString *)componentsJoinedByString:(NSString *)separator;
```

Рис. 20

Foundation Framework

- **NSMutableArray**

  Mutable version of NSArray.

  Create with alloc/init or …

```
+ (id)arrayWithCapacity:(NSUInteger)numItems; // numItems is a performance hint only
+ (id)array;  // [NSMutableArray array] is just like [[NSMutableArray alloc] init]
```

  NSMutableArray inherits all of NSArray's methods.

  Not just count, objectAtIndex:, etc., but also the more interesting ones mentioned last slide.

  And you know that it implements these key methods as well …

```
- (void)addObject:(id)object;  // to the end of the array (note id is the type!)
- (void)insertObject:(id)object atIndex:(NSUInteger)index;
- (void)removeObjectAtIndex:(NSUInteger)index;
```

Рис. 21

Enumeration

- Looping through members of an array in an efficient manner

  Language support using `for-in`.

  Example: `NSArray` of `NSString` objects

```
NSArray *myArray = ...;
for (NSString *string in myArray) { // no way for compiler to know what myArray contains
    double value = [string doubleValue]; // crash here if string is not an NSString
}
```

  Example: `NSArray` of `id`

```
NSArray *myArray = ...;
for (id obj in myArray) {
    // do something with obj, but make sure you don't send it a message it does not respond to
    if ([obj isKindOfClass:[NSString class]]) {
        // send NSString messages to obj with no worries
    }
}
```

Рис. 22

Foundation Framework

- **NSNumber**

  Object wrapper around primitive types like `int`, `float`, `double`, `BOOL`, `enum`s, etc.

```
NSNumber *n = [NSNumber numberWithInt:36];
float f = [n floatValue]; // would return 36.0 as a float (i.e. will convert types)
```

  Useful when you want to put these primitive types in a collection (e.g. `NSArray` or `NSDictionary`).

  New syntax for creating an `NSNumber` in iOS 6: `@()`

```
NSNumber *three = @3;
NSNumber *underline = @(NSUnderlineStyleSingle); // enum
NSNumber *match = @([card match:@[otherCard]]); // expression that returns a primitive type
```

- **NSValue**

  Generic object wrapper for some non-object, non-primitive data types (i.e. C structs).

  e.g. `NSValue *edgeInsetsObject = [NSValue valueWithUIEdgeInsets:UIEdgeInsetsMake(1,1,1,1)]`

  Probably don't need this in this course (maybe when we start using points, sizes and rects).

Foundation Framework

- NSData

  "Bag of bits." Used to save/restore/transmit raw data throughout the iOS SDK.

- NSDate

  Used to find out the time right now or to store past or future times/dates.
  See also `NSCalendar`, `NSDateFormatter`, `NSDateComponents`.
  If you are going to display a date in your UI, make sure you study this in detail (localization).

- NSSet / NSMutableSet

  Like an array, but no ordering (no `objectAtIndex:` method).
  `member:` is an important method (returns an object if there is one in the set `isEqual:` to it).
  Can union and intersect other sets.

- NSOrderedSet / NSMutableOrderedSet

  Sort of a cross between `NSArray` and `NSSet`.
  Objects in an ordered set are distinct. You can't put the same object in multiple times like array.

Рис. 24

Foundation Framework

- NSDictionary

Immutable collection of objects looked up by a key (simple hash table).
All keys and values are held onto strongly by an NSDictionary.

Can create with this syntax: `@{ key1 : value1, key2 : value2, key3 : value3 }`

```
NSDictionary *colors = @{ @"green" : [UIColor greenColor],
                          @"blue" : [UIColor blueColor],
                          @"red" : [UIColor redColor] };
```

Lookup using "array like" notation ...

```
NSString *colorString = ...;
UIColor *colorObject = colors[colorString]; // works the same as objectForKey: below
```

```
- (NSUInteger)count;
- (id)objectForKey:(id)key; // key must be copyable and implement isEqual: properly
```

NSStrings make good keys because of this.
See NSCopying protocol for more about what it takes to be a key.

Foundation Framework

- NSMutableDictionary

Mutable version of NSDictionary.

Create using alloc/init or one of the + (id)dictionary... class methods.

In addition to all the methods inherited from NSDictionary, here are some important methods ...

```
- (void)setObject:(id)anObject forKey:(id)key;
- (void)removeObjectForKey:(id)key;
- (void)removeAllObjects;
- (void)addEntriesFromDictionary:(NSDictionary *)otherDictionary;
```

Рис. 26

Enumeration

- Looping through the keys or values of a dictionary

Example:

```
NSDictionary *myDictionary = ...;
for (id key in myDictionary) {
    // do something with key here
    id value = [myDictionary objectForKey:key];
    // do something with value here
}
```

Рис. 27

Property List

- The term "Property List" just means a collection of collections
  It's just a phrase (not a language thing). It means any graph of objects containing only:
  `NSArray`, `NSDictionary`, `NSNumber`, `NSString`, `NSDate`, `NSData` (or mutable subclasses thereof)
- An `NSArray` is a Property List if all its members are too
  So an `NSArray` of `NSString` is a Property List.
  So is an `NSArray` of `NSArray` as long as those `NSArray`'s members are Property Lists.
- An `NSDictionary` is one only if all keys and values are too
  An `NSArray` of `NSDictionary`s whose keys are `NSString`s and values are `NSNumber`s is one.
- Why define this term?
  Because the SDK has a number of methods which operate on Property Lists.
  Usually to read them from somewhere or write them out to somewhere. Example:

```
- (void)writeToFile:(NSString *)path atomically:(BOOL)atom;
```

  This can (only) be sent to an NSArray or NSDictionary that contains only Property List objects.

Рис. 28

Other Foundation

- NSUserDefaults

Lightweight storage of Property Lists.

It's basically an `NSDictionary` that persists between launches of your application.

Not a full-on database, so only store small things like user preferences.

Read and write via a shared instance obtained via class method `standardUserDefaults` ...

```
[[NSUserDefaults standardUserDefaults] setArray:rvArray forKey:@"RecentlyViewed"];
```

Sample methods:

```
- (void)setDouble:(double)aDouble forKey:(NSString *)key;
- (NSInteger)integerForKey:(NSString *)key;  // NSInteger is a typedef to 32 or 64 bit int
- (void)setObject:(id)obj forKey:(NSString *)key; // obj must be a Property List
- (NSArray *)arrayForKey:(NSString *)key; // will return nil if value for key is not NSArray
```

Always remember to write the defaults out after each batch of changes!

```
[[NSUserDefaults standardUserDefaults] synchronize];
```

Рис. 29

Other Foundation

- NSRange

C struct (not a class)

Used to specify subranges inside strings and arrays (et. al.).

```
typedef struct {
    NSUInteger location;
    NSUInteger length;
} NSRange;
```

Important `location` value `NSNotFound`.

```
NSString *greeting = @"hello world";  NSString *hi = @"hi";
NSRange r = [greeting rangeOfString:hi];  // finds range of hi characters inside greeting
if (r.location == NSNotFound) { /* couldn't find hi inside greeting */ }
```

`NSRangePointer` (just an NSRange * ... used as an out method parameter).

There are C functions like `NSEqualRanges()`, `NSMakeRange()`, etc.

Рис. 30

Colors

- **UIColor**

An object representing a color.

Initializers for creating a color based on RGB, HSB and even a pattern (UIImage).

Colors can also have alpha (`UIColor *color = [otherColor colorWithAlphaComponent:0.3]`).

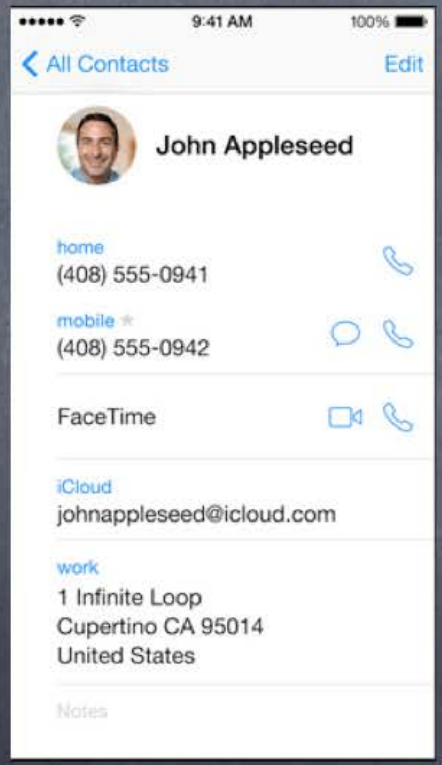
A handful of "standard" colors have class methods (e.g. `[UIColor greenColor]`).
A few "system" colors also have class methods (e.g. `[UIColor lightTextColor]`).

Рис. 31

Fonts

- **UIFont**

  Fonts in iOS 7 are very important to get right.



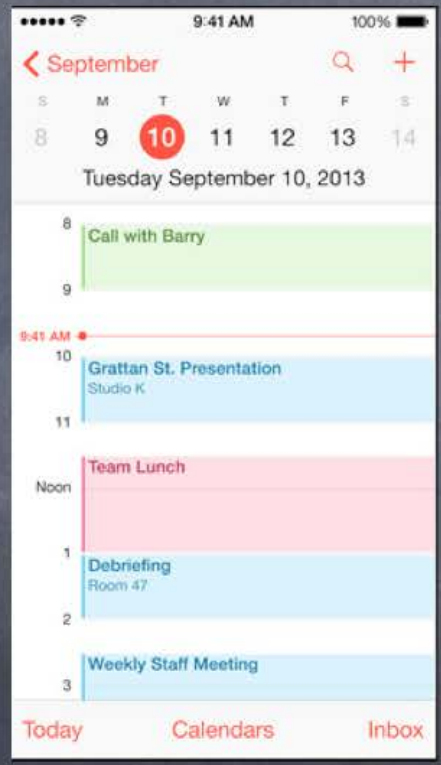


Рис. 32

Fonts

- UIFont

It is best to get a UIFont by asking for the preferred font for a given text style ...

```
UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];
```

Some other styles (see UIFontDescriptor documentation for even more styles) ...

UIFontTextStyleHeadline, UIFontTextStyleCaption1, UIFontTextStyleFootnote, etc.

There are also "system" fonts.

They are used in places like button titles.

```
+ (UIFont *)systemFontOfSize:(CGFloat)pointSize;
+ (UIFont *)boldSystemFontOfSize:(CGFloat)pointSize;
```

You should never uses these for your user's content.

Use preferredFontForTextStyle: for that.

Рис. 33

Fonts

- `UIFontDescriptor`

Fonts are designed by artists.
They aren't always designed to fit any sort of categorization.
Some fonts have Narrow or Bold or Condensed faces, some do not.
Even "size" is sometimes a designed-in aspect of a particular font.

A `UIFontDescriptor` attempts to categorize a font anyway.
It does so by family, face, size, and other attributes.
You can then ask for fonts that have those attributes and get a "best match."

Understand that a best match for a "bold" font may not be bold if there's no such designed face.

Рис. 34

Fonts

- UIFontDescriptor

  You can get a font descriptor from an existing UIFont with this UIFont method …

  ```
  - (UIFontDescriptor *)fontDescriptor;
  ```

  You might well have gotten the original UIFont using preferredFontForTextStyle:.

  Then you might modify it to create a new descriptor with methods in UIFontDescriptor like …

  ```
  - (UIFontDescriptor *)fontDescriptorByAddingAttributes:(NSDictionary *)attributes;
  ```

  (the attributes and their values can be found in the class reference page for UIFontDescriptor)

  You can also create a UIFontDescriptor directly from attributes (though this is rare) using …

  ```
  + (UIFontDescriptor *)fontDescriptorWithFontAttributes:(NSDictionary *)attributes;
  ```

- Symbolic Traits

  Italic, Bold, Condensed, etc., are important enough to get their own API in UIFontDescriptor …

  ```
  - (UIFontDescriptorSymbolicTraits)symbolicTraits;
  - (UIFontDescriptor *)fontDescriptorWithSymbolicTraits:(UIFontDescriptorSymbolicTraits)traits;
  ```

  Some example traits (again, see UIFontDescriptor documentation for more) …

  UIFontDescriptorTraitItalic, UIFontDescriptorTraitBold, UIFontDescriptorTraitCondensed, etc.

Рис. 35

Fonts

- UIFontDescriptor

  Once you have a UIFontDescriptor that describes the font you want, use this UIFont method:

  ```
  + (UIFont *)fontWithDescriptor:(UIFontDescriptor *)descriptor size:(CGFloat)size;
  ```

  (specify size of 0 if you want to use whatever size is in the descriptor)

  You will get a "best match" for your descriptor given available fonts and their faces.

- Example

  Let's try to get a "bold body font" ...

  ```
  UIFont *bodyFont = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];
  UIFontDescriptor *existingDescriptor = [bodyFont fontDescriptor];
  UIFontDescriptorSymbolicTraits traits = existingDescriptor.symbolicTraits;
  traits |= UIFontDescriptorTraitBold;
  UIFontDescriptor *newDescriptor = [existingDescriptor fontDescriptorWithSymbolicTraits:traits];
  UIFont *boldBodyFont = [UIFont fontWithFontDescriptor:newDescriptor size:0];
  ```

  This will do the best it can to give you a bold version of the UIFontTextStyleBody preferred font.

  It may or may not actually be bold.

Рис. 36

Attributed Strings

- **How text looks on screen**
  The font has a lot to do with how text looks on screen.
  But there are other determiners (color, whether it is "outlined", stroke width, underlining, etc.).
  You put the text together with a font and these other determiners using `NSAttributedString`.
- **`NSAttributedString`**
  Think of it as an `NSString` where each character has an `NSDictionary` of "attributes".
  The attributes are things like the font, the color, underlining or not, etc., of the character.
  It is not, however, actually a subclass of `NSString` (more on this in a moment).
- **Getting Attributes**
  You can ask an `NSAttributedString` all about the attributes at a given location in the string.

```
- (NSDictionary *)attributesAtIndex:(NSUInteger)index
                     effectiveRange:(NSRangePointer)range;
```

  The range is returned and it lets you know for how many characters the attributes are identical.
  There are also methods to ask just about a certain attribute you might be interested in.
  `NSRangePointer` is essentially an `NSRange *`. It's okay to pass `NULL` if you don't care.

Рис. 37

Attributed Strings

- NSAttributedString is not an NSString

  It does not inherit from NSString, so you cannot use NSString methods on it.

  If you need to operate on the characters, there is this great method in NSAttributedString …

```
- (NSString *)string;
```

  For example, to find a substring in an NSAttributedString, you could do this …

```
NSAttributedString *attributedString = …;
NSString *substring = …;
NSRange r = [[attributedString string] rangeOfString:substring];
```

  The method `string` is guaranteed to be high performance but is volatile.

  If you want to keep it around, make a `copy` of it.

Attributed Strings

- NSMutableAttributedString

  Unlike NSString, we almost always use mutable attributed strings.

- Adding or setting attributes on the characters

  You can add an attribute to a range of characters …

  ```
  - (void)addAttributes:(NSDictionary *)attributes range:(NSRange)range;
  ```

  … which will change the values of attributes in attributes and not touch other attributes.

  Or you can set the attributes in a range …

  ```
  - (void)setAttributes:(NSDictionary *)attributes range:(NSRange)range;
  ```

  … which will remove all other attributes in that range in favor of the passed attributes.

  You can also remove a specific attribute from a range …

  ```
  - (void)removeAttribute:(NSString *)attributeName range:(NSRange)range;
  ```

- Modifying the contents of the string (changing the characters)

  You can do that with methods to append, insert, delete or replace characters.

  Or call the NSMutableAttributedString method `- (NSMutableString *)mutableString`
  and modify the returned NSMutableString (attributes will, incredibly, be preserved!).

Рис. 39

Attributed Strings

So what kind of attributes are there?

```
@{ NSFontAttributeName :
      [UIFont preferredFontWithTextStyle:UIFontTextStyleHeadline] }
```

Рис. 40

Attributed Strings

So what kind of attributes are there?

```
@{ NSFontAttributeName :
      [UIFont preferredFontWithTextStyle:UIFontTextStyleHeadline]
   NSForegroundColorAttributeName : [UIColor blueColor] }
```

Рис. 41

Attributed Strings

So what kind of attributes are there?

```
@{ NSFontAttributeName :
     [UIFont preferredFontWithTextStyle:UIFontTextStyleHeadline]
  NSForegroundColorAttributeName : [UIColor greenColor] }
```

Be careful with colored text.
Color is one of the primary ways
a user knows what's "clickable."
Be Consistent.

Рис. 42

Attributed Strings

So what kind of attributes are there?

```
@{ NSFontAttributeName :
     [UIFont preferredFontWithTextStyle:UIFontTextStyleHeadline]
  NSForegroundColorAttributeName : [UIColor greenColor],

  NSStrokeWidthAttributeName : @-5,

  NSStrokeColorAttributeName : [UIColor orangeColor] }
```

Negative number means "fill and stroke."
Positive number is stroke only (outline).

Рис. 43

Attributed Strings

So what kind of attributes are there?

```
@{ NSFontAttributeName :
     [UIFont preferredFontWithTextStyle:UIFontTextStyleHeadline]
  NSForegroundColorAttributeName : [UIColor greenColor],

  NSStrokeWidthAttributeName : @-5,

  NSStrokeColorAttributeName : [UIColor redColor],

  NSUnderlineStyleAttributeName : @(NSUnderlineStyleSingle) }
```

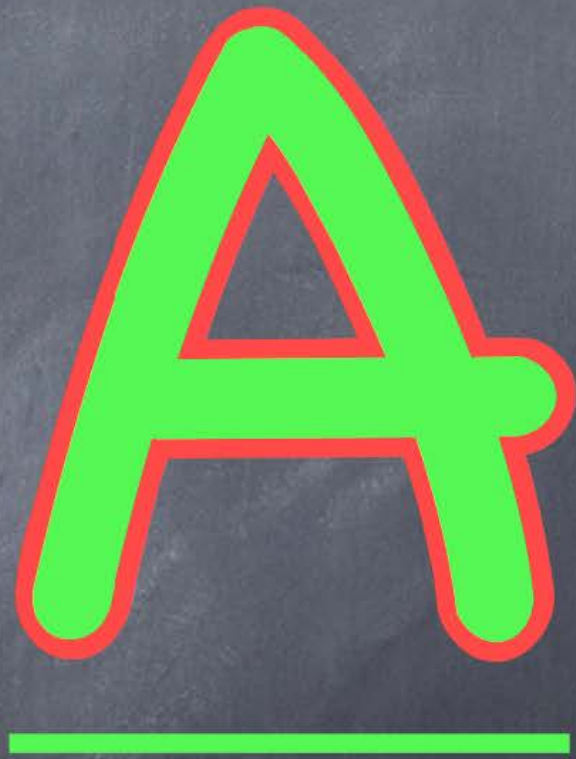


Рис. 44

Attributed Strings

So what kind of attributes are there?

```
@{ NSFontAttributeName :
     [UIFont preferredFontWithTextStyle:UIFontTextStyleHeadline]
  NSForegroundColorAttributeName : [UIColor greenColor],

  NSStrokeWidthAttributeName : @-5,

  NSStrokeColorAttributeName : [UIColor redColor],

  NSUnderlineStyleAttributeName : @(NSUnderlineStyleNone) }
```

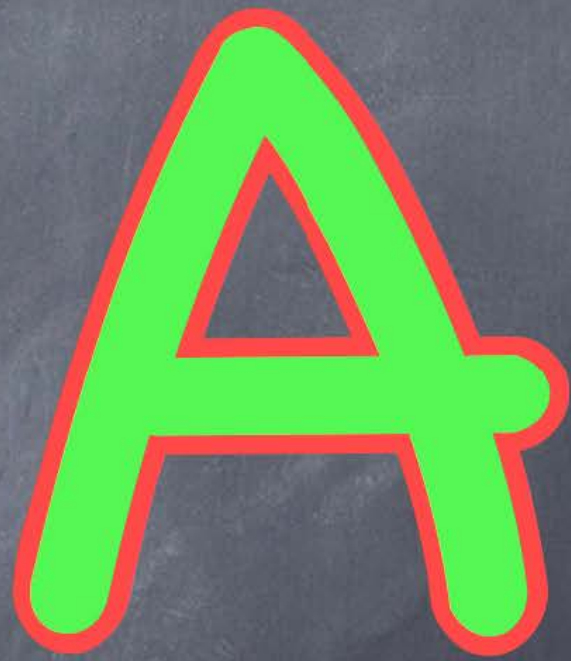


Рис. 45

Attributed Strings

```
UIColor *yellow = [UIColor yellowColor];
UIColor *transparentYellow = [yellow colorWithAlphaComponent:0.3];
 @{ NSFontAttributeName :
      [UIFont preferredFontWithTextStyle:UIFontTextStyleHeadline]
   NSForegroundColorAttributeName : [UIColor greenColor],
   NSStrokeWidthAttributeName : @-5,
   NSStrokeColorAttributeName : [UIColor redColor],
   NSUnderlineStyleAttributeName : @(NSUnderlineStyleNone),
   NSBackgroundColorAttributeName : transparentYellow }
```

You could use transparent colors in other attributes as well.

Рис. 46

NSAttributedString

- Where do attributed strings get used?

  UIButton's `- (void)setAttributedTitle:(NSAttributedString *)title forState:…;`
  UILabel's `@property (nonatomic, strong) NSAttributedString *attributedText;`
  UITextView's `@property (nonatomic, readonly) NSTextStorage *textStorage;`

- UIButton

  Attributed strings on buttons will be extremely use for your homework.

- Drawing strings directly

  Next week we'll see how to draw things directly on screen.
  NSAttributedStrings know how to draw themselves on screen, for example …
  `- (void)drawInRect:(CGRect)aRect;`
  Don't worry about this too much for now. Wait until next week.

UILabel

- **UILabel**

  You have been setting its contents using the NSString property text.
  But it also has a property to set/get its text using an NSAttributedString …

```
@property (nonatomic, strong) NSAttributedString *attributedText;
```

- **Note that this attributed string is not mutable**

  So, to modify what is in a UILabel, you must make a mutableCopy, modify it, then set it back.

```
NSMutableAttributedString *labelText = [myLabel.attributedText mutableCopy];
[labelText setAttributes:…];
myLabel.attributedText = labelText;
```

- **Don't need this very often**

  There are properties in UILabel like font, textColor, etc., for setting look of all characters.
  The attributed string in UILabel would be used mostly for "specialty labels".