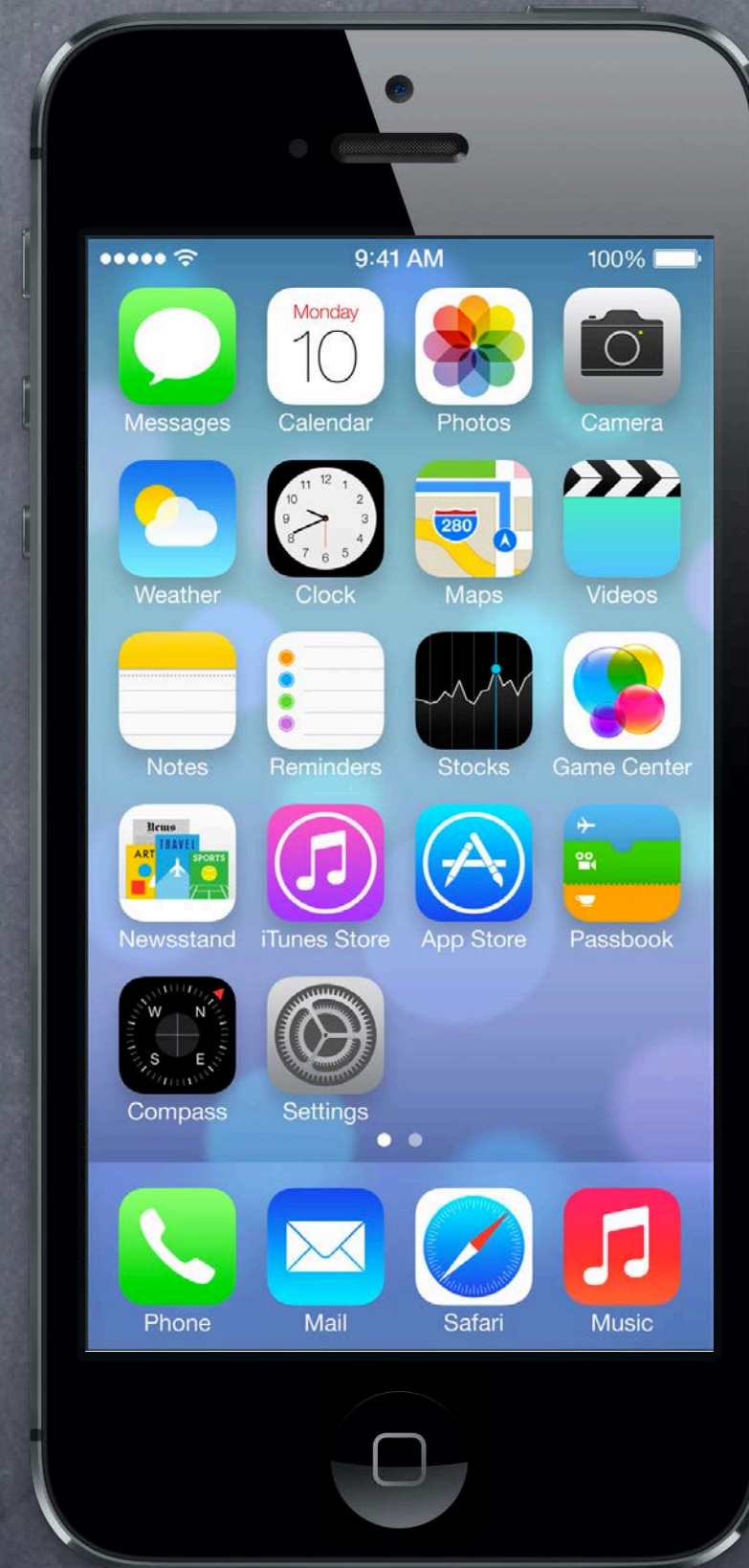


# Stanford CS193p

Developing Applications for iOS  
Fall 2013-14



# Today

- **Protocols**

How to make id a little bit safer.

- **Blocks**

Passing a block of code as an argument to a method.

- **Animation**

Dynamic Animator

View property animation

- **Demo**

Dropit!

# Protocols

- The problem with `id` ...

Obviously it's hard to communicate your intent with `id`.

What do you want callers of this method to pass (or what are you returning) exactly?

- Introspection

Helps occasionally, but not a "primary programming methodology."

And it doesn't help with communicating your intent at all (it's more of a runtime thing).

- Protocols

A syntactical modification of `id`, for example, `id <MyProtocol> obj`.

`MyProtocol` would then be defined to be a list of methods (including `@propertys`).

The variable `obj` now can point to an object of any class, but that it implements known methods.

Not all the methods in a protocol have to be required, but still, you'll know what's expected.

Let's look at the syntax ...

# Protocols

## • Declaring a `@protocol`

Looks a lot like `@interface` (but there's no corresponding `@implementation`)

```
@protocol Foo
```

```
- (void)someMethod;
```

```
- (void)methodWithArgument:(BOOL)argument;
```

```
@property (readonly) int readonlyProperty; // getter (only) is part of this protocol
```

```
@property NSString *readwriteProperty; // getter and setter are both in the protocol
```

```
- (int)methodThatReturnsSomething;
```

```
@end
```

All of these methods are required. Anyone implementing this protocol must implement them all.

# Protocols

## • Declaring a `@protocol`

Looks a lot like `@interface` (but there's no corresponding `@implementation`)

```
@protocol Foo
```

```
- (void)someMethod;
```

```
@optional
```

```
- (void)methodWithArgument:(BOOL)argument;
```

```
@property (readonly) int readonlyProperty; // getter (only) is part of this protocol
```

```
@property NSString *readwriteProperty; // getter and setter are both in the protocol
```

```
- (int)methodThatReturnsSomething;
```

```
@end
```

Now only the first one is required.

You can still say you implement Foo even if you only implement someMethod.

# Protocols

## • Declaring a `@protocol`

Looks a lot like `@interface` (but there's no corresponding `@implementation`)

```
@protocol Foo
```

```
- (void)someMethod;
```

```
@optional
```

```
- (void)methodWithArgument:(BOOL)argument;
```

```
@required
```

```
@property (readonly) int readonlyProperty; // getter (only) is part of this protocol
```

```
@property NSString *readwriteProperty; // getter and setter are both in the protocol
```

```
- (int)methodThatReturnsSomething;
```

```
@end
```

Now all of them except `methodWithArgument:` are required.

# Protocols

## • Declaring a `@protocol`

Looks a lot like `@interface` (but there's no corresponding `@implementation`)

```
@protocol Foo <Xyzy>
```

```
- (void)someMethod;
```

```
@optional
```

```
- (void)methodWithArgument:(BOOL)argument;
```

```
@required
```

```
@property (readonly) int readonlyProperty; // getter (only) is part of this protocol
```

```
@property NSString *readwriteProperty; // getter and setter are both in the protocol
```

```
- (int)methodThatReturnsSomething;
```

```
@end
```

Now all of them except `methodWithArgument:` are required.

Now you can only say you implement `Foo` if you also implement the methods in `Xyzy` protocol.

# Protocols

## • Declaring a `@protocol`

Looks a lot like `@interface` (but there's no corresponding `@implementation`)

```
@protocol Foo <Xyzy, NSObject>
```

```
- (void)someMethod;
```

```
@optional
```

```
- (void)methodWithArgument:(BOOL)argument;
```

```
@required
```

```
@property (readonly) int readonlyProperty; // getter (only) is part of this protocol
```

```
@property NSString *readwriteProperty; // getter and setter are both in the protocol
```

```
- (int)methodThatReturnsSomething;
```

```
@end
```

Now all of them except `methodWithArgument:` are required.

Now you can only say you implement `Foo` if you also implement the methods in `Xyzy` protocol.

Now you would have to implement both the `Xyzy` protocol and the `NSObject` protocol (what's that!?).



# Protocols

- `@protocol NSObject`

Has things like `class`, `isEqual:`, `isKindOfClass:`, `description`, `performSelector:`, etc.

Not uncommon to add this requirement when declaring a protocol.

Then you can rely on using introspection and such on the object obeying the protocol.

Of course, the class `NSObject` implements the protocol `NSObject` (so it comes for free!).

# Protocols

- Where do `@protocol` declarations go?

In header files.

It can go in its own, dedicated header file.

Or it can go in the header file of the class that is going to require its use.

Which to do?

If the `@protocol` is only required by a particular class's API, then put it there, else put it in its own header file.

Example: The `UIScrollViewDelegate` protocol is defined in `UIScrollView.h`.

# Protocols

- Okay, I have a `@protocol` declared, now what?

Now classes can promise to implement the protocol in their `@interface` declarations.

Okay to put in `private @interface` if they don't want others to know they implement it.

- Example:

```
#import "Foo.h" // importing the header file that declares the Foo @protocol
@interface MyClass : NSObject <Foo> // MyClass is saying it implements the Foo @protocol
    (do not have to declare Foo's methods again here, it's implicit that you implement it)
@end
```

`@end`

... or ("or" not "and"... it's one or the other, private or public, not both) ...

```
@interface MyClass() <Foo>
```

`@end`

```
@implementation MyClass
```

```
// in either case, you had better implement Foo's @required methods here!
```

`@end`

# Protocols

- The class must now implement all non-@optional methods

Or face the wrath of the compiler if you do not (but that's the only wrath you'll face).

No warning if you don't implement @optional methods.

@optional is more a mechanism to say: "hey, if you implement this, I'll use it."

(i.e. caller will likely use introspection to be sure you actually implement an @optional method)

@required is much stronger: "if you want this to work, you must implement this."

(very unlikely that the caller would use introspection before invoking @required methods)

# Protocols

- Okay, so now what?

We have protocols.

We have classes that promise to implement them.

Now we need variables that hold pointers to objects that make that promise.

- Examples ...

```
id <Foo> obj = [[MyClass alloc] init]; // compiler will love this (due to previous slides)
```

```
id <Foo> obj = [NSArray array]; // compiler will not like this one bit!
```

- Can also declare arguments to methods to require a protocol

```
- (void)giveMeFooObject:(id <Foo>)anObjectImplementingFoo;
```

```
@property (nonatomic, weak) id <Foo> myFooProperty; // properties too!
```

If you call these and pass an object which does not implement Foo ... compiler warning!

# Protocols

- Just like static typing, this is all just compiler-helping-you stuff  
It makes no difference at runtime.
- Think of it as documentation for your method interfaces  
It's a powerful way to leverage the `id` type.

# Protocols

- #1 use of protocols in iOS: **delegates** and **dataSources**

Often when an object in iOS wants something important and non-generic done, it may delegate it. It does this through a property on that iOS object that is specified with a certain protocol.

```
@property (nonatomic, weak) id <UISomeObjectDelegate> delegate;
```

```
@property (nonatomic, weak) id <UISomeObjectDataSource> dataSource;
```

Note that it is a **weak** (or worse) **@property**, by the way (more on that soon).

You may implement your own delegates too (we'll see that later in the course).

This is an alternative to subclassing to provide non-generic behavior.

You use delegation when you want to be "blind" to the class of the implementing object (MVC).

- **dataSource** and Views

Complex UIView classes commonly have a dataSource because Views cannot own their data!

- Other uses of protocols

Declaring what sorts of things are "animatable" (mostly UIView, but other things too).

We'll see other uses as the quarter progresses.

# Blocks

## • What is a **block**?

A block of code (i.e. a sequence of statements inside `{}`).

Usually included "in-line" with the calling of method that is going to use the block of code.

Very smart about local variables, referenced objects, etc.

## • What does it look like?

Here's an example of calling a method that takes a **block** as an argument.

```
[aDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {  
    NSLog(@"value for key %@ is %@", key, value);  
    if ([@"ENOUGH" isEqualToString:key]) {  
        *stop = YES;  
    }  
}];
```

This `NSLog()`s every **key** and **value** in `aDictionary` (but stops if the **key** is "ENOUGH").

## • Blocks start with the magical character caret `^`

Then (optional) return type, then (optional) arguments in parentheses, then `{`, then code, then `}`.



# Blocks

- Can use local variables declared before the **block** inside the **block**

```
double stopValue = 53.5;
[aDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
    NSLog(@"value for key %@ is %@", key, value);
    if ([@"ENOUGH" isEqualToString:key] || ([value doubleValue] == stopValue)) {
        *stop = YES;
    }
}];
```

- But they are read only!

```
BOOL stoppedEarly = NO;
double stopValue = 53.5;
[aDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
    NSLog(@"value for key %@ is %@", key, value);
    if ([@"ENOUGH" isEqualToString:key] || ([value doubleValue] == stopValue)) {
        *stop = YES;
        stoppedEarly = YES; // ILLEGAL
    }
}];
```

# Blocks

- Unless you mark the local variable as `__block`

```
__block BOOL stoppedEarly = NO;
```

```
double stopValue = 53.5;
```

```
[aDictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {  
    NSLog(@"value for key %@ is %@", key, value);  
    if ([@"ENOUGH" isEqualToString:key] || ([value doubleValue] == stopValue)) {  
        *stop = YES;  
        stoppedEarly = YES; // this is legal now  
    }  
}];  
if (stoppedEarly) NSLog(@"I stopped logging dictionary values early!");
```

- Or if the "variable" is an instance variable

But we only access instance variables (e.g. `_display`) in setters and getters.  
So this is of minimal value to us.

# Blocks

- So what about objects which are messaged inside the **block**?

```
NSString *stopKey = [@"Enough" uppercaseString];
__block BOOL stoppedEarly = NO;
double stopValue = 53.5;
[adictionary enumerateKeysAndObjectsUsingBlock:^(id key, id value, BOOL *stop) {
    NSLog(@"value for key %@ is %@", key, value);
    if ([stopKey isEqualToString:key] || ([value doubleValue] == stopValue)) {
        *stop = YES;
        stoppedEarly = YES; // this is legal now
    }
}];
if (stoppedEarly) NSLog(@"I stopped logging dictionary values early!");
```

**stopKey** will automatically have a **strong** pointer to it until the **block** goes out of scope. This is obviously necessary for the **block** to function properly.

# Blocks

## • Creating a “type” for a variable that can hold a **block**

**Blocks** are kind of like “objects” with an unusual syntax for declaring variables that hold them. Usually if we are going to store a **block** in a variable, we **typedef** a type for that variable, e.g.,  
`typedef double (^unary_operation_t)(double op);`

This declares a type called “**unary\_operation\_t**” for variables which can store a **block**.

(specifically, a **block** which takes a **double** as its only argument and returns a **double**)

Then we could declare a variable, square, of this type and give it a value ...

```
unary_operation_t square;  
square = ^(double operand) { // the value of the square variable is a block  
    return operand * operand;  
}
```

And then use the variable **square** like this ...

```
double squareOfFive = square(5.0); // squareOfFive would have the value 25.0 after this
```

(It is not mandatory to **typedef**, for example, the following is also a legal way to create **square** ...)

```
double (^square)(double op) = ^(double op) { return op * op; };
```

# Blocks

- We could then use the `unary_operation_t` as follows ...

For example, you could have a property which is an array of blocks ...

```
@property (nonatomic, strong) NSMutableDictionary *unaryOperations;
```

Then implement a method like this ...

```
typedef double (^unary_operation_t)(double op);
```

```
- (void)addUnaryOperation:(NSString *)op whichExecutesBlock:(unary_operation_t)opBlock {  
    self.unaryOperations[op] = opBlock;  
}
```

Note that the **block** can be treated somewhat like an object (e.g., adding it to a dictionary).

Later, we could use an operation added with the method above like this ...

```
- (double)performOperation:(NSString *)operation onOperand:(double)operand  
{  
    unary_operation_t unaryOp = self.unaryOperations[operation];  
    if (unaryOp) {  
        double result = unaryOp(operand);  
    }  
    ...  
}
```

# Blocks

## • We don't always typedef

When a **block** is an argument to a method and is used immediately, often there is no typedef.

Here is the declaration of the dictionary enumerating method we showed earlier ...

```
- (void)enumerateKeysAndObjectsUsingBlock:(void (^)(id key, id obj, BOOL *stop))block;
```

No "name" for the type appears here.

The syntax is exactly the same as the typedef except that the name of the typedef is not there.

For reference, here's what a typedef for this argument would look like this ...

```
typedef void (enumeratingBlock)(id key, id obj, BOOL *stop);
```

(i.e. the underlined part is not used in the method argument)

This ("block") is the keyword for the argument (e.g. the local variable name for the argument inside the method implementation).

# Blocks

- Some shorthand allowed when defining a block

If there are no arguments to the `block`, you do not need to have any parentheses.

Consider this code ...

```
[UIView animateWithDuration:5.0 animations:^() {  
    view.opacity = 0.5;  
}];
```

# Blocks

- Some shorthand allowed when defining a block

If there are no arguments to the **block**, you do not need to have any parentheses. Consider this code ...

```
[UIView animateWithDuration:5.0 animations:^{  
    view.opacity = 0.5;  
}]
```

No arguments to this block.

No need for the **()** then.



# Blocks

- Some shorthand allowed when defining a block

If there are no arguments to the `block`, you do not need to have any parentheses. Consider this code ...

```
[UIView animateWithDuration:5.0 animations:^(  
    view.opacity = 0.5;  
)];
```

Also, return type can usually be inferred from the `block`, in which case it is optional.

```
NSSet *mySet = ...;  
NSSet *matches = [mySet objectsPassingTest:^(BOOL(id obj, ...) {  
    return [obj isKindOfClass:[UIView class]];  
});
```

Return type is clearly a BOOL.

# Blocks

- Some shorthand allowed when defining a block

If there are no arguments to the **block**, you do not need to have any parentheses. Consider this code ...

```
[UIView animateWithDuration:5.0 animations:^(  
    view.opacity = 0.5;  
)];
```

Also, return type can usually be inferred from the **block**, in which case it is optional.

```
NSSet *mySet = ...;  
NSSet *matches = [mySet objectsPassingTest:^(id obj, ...) {  
    return [obj isKindOfClass:[UIView class]];  
}];
```

Return type is clearly a BOOL.

So no need for the **BOOL** declaration here.

# Blocks

## • How **blocks** sort of act like objects

It turns out **blocks** can be stored inside other objects (in properties, arrays, dictionaries, etc.). But they act like objects only for the purposes of storing them (their only “method” is **copy**).

For example, if you had a class with the following property ...

```
@property (nonatomic, strong) NSMutableArray *myBlocks; // array of blocks
```

... you could do the following ...

```
[self.myBlocks addObject:^(  
    [self doSomething];  
)];
```

... neat!

By the way, you invoke a **block** that is in the array like this ...

```
void (^doit)(void) = self.myBlocks[0];  
doit();
```

But there is danger lurking here ...

# Blocks

## Memory Cycles (a bad thing)

We said that all objects referenced inside a **block** will stay in the heap as long as the **block** does (in other words, **blocks** keep a **strong** pointer to all objects referenced inside of them).

In the example above, **self** is an object reference in this **block** ...

```
[self.myBlocks←addObject:^ {  
    [self doSomething];  
}];
```

Thus the **block** will have a **strong** pointer to **self**.

But notice that **self** also has a **strong** pointer to the **block** (it's in its `myBlocks` array)!

**This is a serious problem.**

Neither **self** nor the **block** can ever escape the heap now.

That's because there will always be a **strong** pointer to both of them (each other's pointer).

This is called a memory "cycle."

# Blocks

## Memory Cycles Solution

You'll recall that local variables are always **strong**.

That's fine because when they go out of scope, they disappear, so the **strong** pointer goes away.

It turns out there's a way to declare that a local variable is **weak**. Here's how ...

```
__weak MyClass *weakSelf = self; // even though self is strong, weakSelf is weak
```

Now if we reference weakSelf inside the **block**, then the **block** will not strongly point to self ...

```
[self.myBlocks addObject:^ {  
    [weakSelf doSomething];  
}];
```

Now we no longer have a cycle (self still has a **strong** pointer to the **block**, but that's okay).

As long as someone in the universe has a **strong** pointer to this self, the **block's** pointer is good.

And since the **block** will not exist if self does not exist (since myBlocks won't exist), all is well!

# Blocks

- When do we use **blocks** in iOS?
  - Enumeration (like we saw above with NSDictionary)
  - View Animations (we'll talk about that next)
  - Sorting (sort this thing using a **block** as the comparison method)
  - Notification (when something happens, execute this **block**)
  - Error handlers (if an error happens while doing this, execute this **block**)
  - Completion handlers (when you are done doing this, execute this **block**)
- And a super-important use: Multithreading
  - With Grand Central Dispatch (GCD) API
  - We'll talk about that later in the course
- More about **blocks**
  - Search "blocks" in Xcode documentation.

# Animation

- ◉ **Animating views**

  - Animating specific properties.

  - Animating a group of changes to a view "all at once."

  - Physics-based animation.

- ◉ **Animation of View Controller transitions**

  - Beyond the scope of this course, but fundamental principles are the same.

- ◉ **Core Animation**

  - Underlying powerful animation framework (also beyond the scope of this course).

# Animation

- Animation of important UIView properties

The changes are made immediately, but appear on-screen over time (i.e. not instantly).

UIView's class method(s) `animationWithDuration:...`

- Animation of the appearance of arbitrary changes to a UIView

By flipping or dissolving or curling the entire view.

UIView's class method `transitionWithView:...`

- Dynamic Animator

Specify the "physics" of animatable objects (usually UIViews).

Gravity, pushing forces, attachments between objects, collision boundaries, etc.

Let the physics happen!



# UIView Animation

- Changes to certain UIView properties can be animated over time
  - frame
  - transform (translation, rotation and scale)
  - alpha (opacity)
- Done with UIView class method and blocks
  - The class method takes animation parameters and an animation **block** as arguments.
  - The animation block contains the code that makes the changes to the UIView(s).
  - Most also have a “completion **block**” to be executed when the animation is done.
  - The changes inside the **block** are made immediately (even though they will appear “over time”).

# UIView Animation

- Animation class method in UIView

```
+ (void)animateWithDuration:(NSTimeInterval)duration  
    delay:(NSTimeInterval)delay  
    options:(UIViewAnimationOptions)options  
    animations:(void (^)(void))animations  
    completion:(void (^)(BOOL finished))completion;
```

- Example

```
[UIView animateWithDuration:3.0  
    delay:0.0  
    options:UIViewAnimationOptionBeginFromCurrentState  
    animations:^(myView.alpha = 0.0; }  
    completion:^(BOOL fin) { if (fin) [myView removeFromSuperview]; }];
```

This would cause myView to “fade” out over 3 seconds (starting immediately).

Then it would remove myView from the view hierarchy (but only if the fade completed).

If, within the 3 seconds, someone animated the alpha to non-zero, the removal would not happen.

# UIView Animation

## • Another example

```
if (myView.alpha == 1.0) {
    [UIView animateWithDuration:3.0
        delay:2.0
        options:UIViewAnimationOptionBeginFromCurrentState
        animations:^( myView.alpha = 0.0; )
        completion:nil];
    NSLog(@"alpha is %f.", myView.alpha);
}
```

This would also cause myView to “fade” out over 3 seconds (starting in 2 seconds in this case). The `NSLog()` would happen immediately (i.e. not after 3 or 5 seconds) and would print “alpha is 0.” In other words, the animation block’s changes are executed immediately, but the animation itself (i.e. the visual appearance of the change to `alpha`) starts in 2 seconds and takes 3 seconds.

# UIView Animation

## • UIViewAnimationOptions

<code>BeginFromCurrentState</code>	<code>// interrupt other, in-progress animations of these properties</code>
<code>AllowUserInteraction</code>	<code>// allow gestures to get processed while animation is in progress</code>
<code>LayoutSubviews</code>	<code>// animate the layout of subviews along with a parent's animation</code>
<code>Repeat</code>	<code>// repeat indefinitely</code>
<code>Autoreverse</code>	<code>// play animation forwards, then backwards</code>
<code>OverrideInheritedDuration</code>	<code>// if not set, use duration of any in-progress animation</code>
<code>OverrideInheritedCurve</code>	<code>// if not set, use curve (e.g. ease-in/out) of in-progress animation</code>
<code>AllowAnimatedContent</code>	<code>// if not set, just interpolate between current and end state image</code>
<code>CurveEaseInEaseOut</code>	<code>// slower at the beginning, normal throughout, then slow at end</code>
<code>CurveEaseIn</code>	<code>// slower at the beginning, but then constant through the rest</code>
<code>CurveLinear</code>	<code>// same speed throughout</code>

# UIView Animation

- Sometimes you want to make an entire view modification at once

By flipping view over `UIViewAnimationOptionsTransitionFlipFrom{Left,Right,Top,Bottom}`

Dissolving from old to new state `UIViewAnimationOptionsTransitionCrossDissolve`

Curling up or down `UIViewAnimationOptionsTransitionCurl{Up,Down}`

Just put the changes inside the animations block of this UIView class method ...

```
+ (void)transitionWithView:(UIView *)view
    duration:(NSTimeInterval)duration
    options:(UIViewAnimationOptions)options
    animations:(void (^)(void))animations
    completion:(void (^)(BOOL finished))completion;
```

# UIView Animation

- Animating changes to the view hierarchy is slightly different

Animate swapping the replacement of one view with another in the view hierarchy.

```
+ (void)transitionFromView:(UIView *)fromView
    toView:(UIView *)toView
    duration:(NSTimeInterval)duration
    options:(UIViewAnimationOptions)options
    completion:(void (^)(BOOL finished))completion;
```

Include `UIViewAnimationOptionShowHideTransitionViews` if you want to use the `hidden` property. Otherwise it will actually remove `fromView` from the view hierarchy and add `toView`.

# Dynamic Animation

- A little different approach to animation than above

Set up physics relating animatable objects and let them run until they resolve to stasis

Easily possible to set it up so that stasis never occurs, but that could be performance problem

- Steps

Create a UIDynamicAnimator

Add UIDynamicBehaviors to it (gravity, collisions, etc.)

Add UIDynamicItems (usually UIViews) to the UIDynamicBehaviors

That's it! Things will instantly start happening.

# Dynamic Animator

- Create a UIDynamicAnimator

```
UIDynamicAnimator *animator = [[UIDynamicAnimator alloc] initWithReferenceView:aView];
```

If animating views, all views must be in a view hierarchy with reference view at the top.

- Create and add UIDynamicBehaviors

```
e.g., UIGravityBehavior *gravity = [[UIGravityBehavior alloc] init];
```

```
[animator addBehavior:gravity];
```

```
e.g., UICollisionBehavior *collider = [[UICollisionBehavior alloc] init];
```

```
[animator addBehavior:collider];
```



# Dynamic Animator

- Add UIDynamicItems to a UIDynamicBehavior

```
id <UIDynamicItem> item1 = ...;
id <UIDynamicItem> item2 = ...;
[gravity addItem:item1];
[collider addItem:item1];
[gravity addItem:item2];
```

The items have to implement the UIDynamicItem protocol ...

```
@protocol UIDynamicItem
@property (readonly) CGRect bounds;
@property (readwrite) CGPoint center;
@property (readwrite) CGAffineTransform transform;
@end
```

UIView implements this @protocol.

If you change center or transform while animator is running, you must call UIDynamicAnimator's  
- (void)updateItemUsingCurrentState:(id <UIDynamicItem>)item;

# Behaviors

- **UIGravityBehavior**

```
@property CGFloat angle;  
@property CGFloat magnitude; // 1.0 is 1000 points/s/s
```

- **UICollisionBehavior**

```
@property UICollisionBehaviorMode collisionMode; // Items, Boundaries, Everything (default)  
- (void)addBoundaryWithIdentifier:(NSString *)identifier forPath:(UIBezierPath *)path;  
@property BOOL translatesReferenceBoundsIntoBoundary;
```

- **UIAttachmentBehavior**

```
- (instancetype)initWithItem:(id <UIDynamicItem>)item attachedToAnchor:(CGPoint)anchor;  
- (instancetype)initWithItem:(id <UIDynamicItem>)i1 attachedToItem:(id <UIDynamicItem>)i2;  
- (instancetype)initWithItem:(id <UIDynamicItem>)item offsetFromCenter:(CGPoint)offset ...  
@property (readwrite) CGFloat length; // distance between attached things (settable!)
```

Can also control damping and frequency of oscillations.

```
@property (readwrite) CGPoint anchorPoint; // can be reset at any time
```

# Behaviors

- UISnapBehavior

- (instancetype)initWithItem:(id <UIDynamicItem>)item snapToPoint:(CGPoint)point;

- Imagine four springs at four corners around the item in the new spot.

- You can control the damping of these “four springs” with @property CGFloat damping;.

- UIPushBehavior

- @property UIPushBehaviorMode mode; // Continuous or Instantaneous

- @property CGVector pushDirection;

- @property CGFloat magnitude/angle; // magnitude 1.0 moves a 100x100 view at 100 pts/s/s

# Behaviors

- `UIDynamicItemBehavior`

Controls the behavior of items as they are affected by other behaviors. Any item added to this behavior (with `addItem:`) will be affected.

```
@property BOOL allowsRotation;
```

```
@property BOOL friction;
```

```
@property BOOL elasticity;
```

```
@property CGFloat density;
```

Can also get information about items ...

```
- (CGPoint)linearVelocityForItem:(id <UIDynamicItem>)item;
```

```
- (CGFloat)angularVelocityForItem:(id <UIDynamicItem>)item;
```

If you have multiple `UIDynamicItemBehaviors`, you will have to know what you are doing.

# Behaviors

- `UIDynamicBehavior`

Superclass of behaviors.

You can create your own subclass which is a combination of other behaviors.

Usually you override `init` method(s) and `addItem(s):` and `removeItem(s):` to do ...

```
- (void)addChildBehavior:(UIDynamicBehavior *)behavior;
```

This is a good way to encapsulate a physics behavior that is a composite of other behaviors.

You might also have some API which helps your subclass configure its children.

- All behaviors know the `UIDynamicAnimator` they are part of

They can only be part of one at a time.

```
@property UIDynamicAnimator *dynamicAnimator;
```

And the behavior will be sent this message when its animator changes ...

```
- (void)willMoveToAnimator:(UIDynamicAnimator *)animator;
```

# Behaviors

- UIDynamicBehavior's `action` property

Every time the behavior is applied, the `block` set with this UIDynamicBehavior property is called ...

```
@property (copy) void (^action)(void);
```

(i.e. it's called `action`, it takes no arguments and returns nothing)

You can set this to do anything you want.

But it will be called a lot, so make it very efficient.

If the `action` refers to properties in the behavior itself, watch out for memory cycles.

# Demo

## 👁 Dropit

Drop squares, collect them at the bottom of the screen, then blow them up!

## 👁 What to look for ...

UIDynamicAnimator and UIDynamicItem @protocol

UIGravityBehavior

UICollisionBehavior

UIDynamicItemBehavior (basically physics configuration)

Composite Behaviors (UIDynamicBehavior subclass)

Flying UIViews using animateWithDuration:...

Animation completion **blocks**

UIDynamicAnimator's delegate (reacting to pauses in dynamic animation)

UIAttachmentBehavior

Adding an action **block** to a behavior

Observing the behavior of items (elapsed animation time, linear velocity, etc.)

UICollisionBehavior's collisionDelegate

# Coming Up

- **Wednesday**

Continuation of demo.  
Autolayout

- **Friday**

Still hoping to get University Developer Program up and running.

- **Homework**

Due a week from today.

- **Next Week**

Scroll View

Table View

Collection View