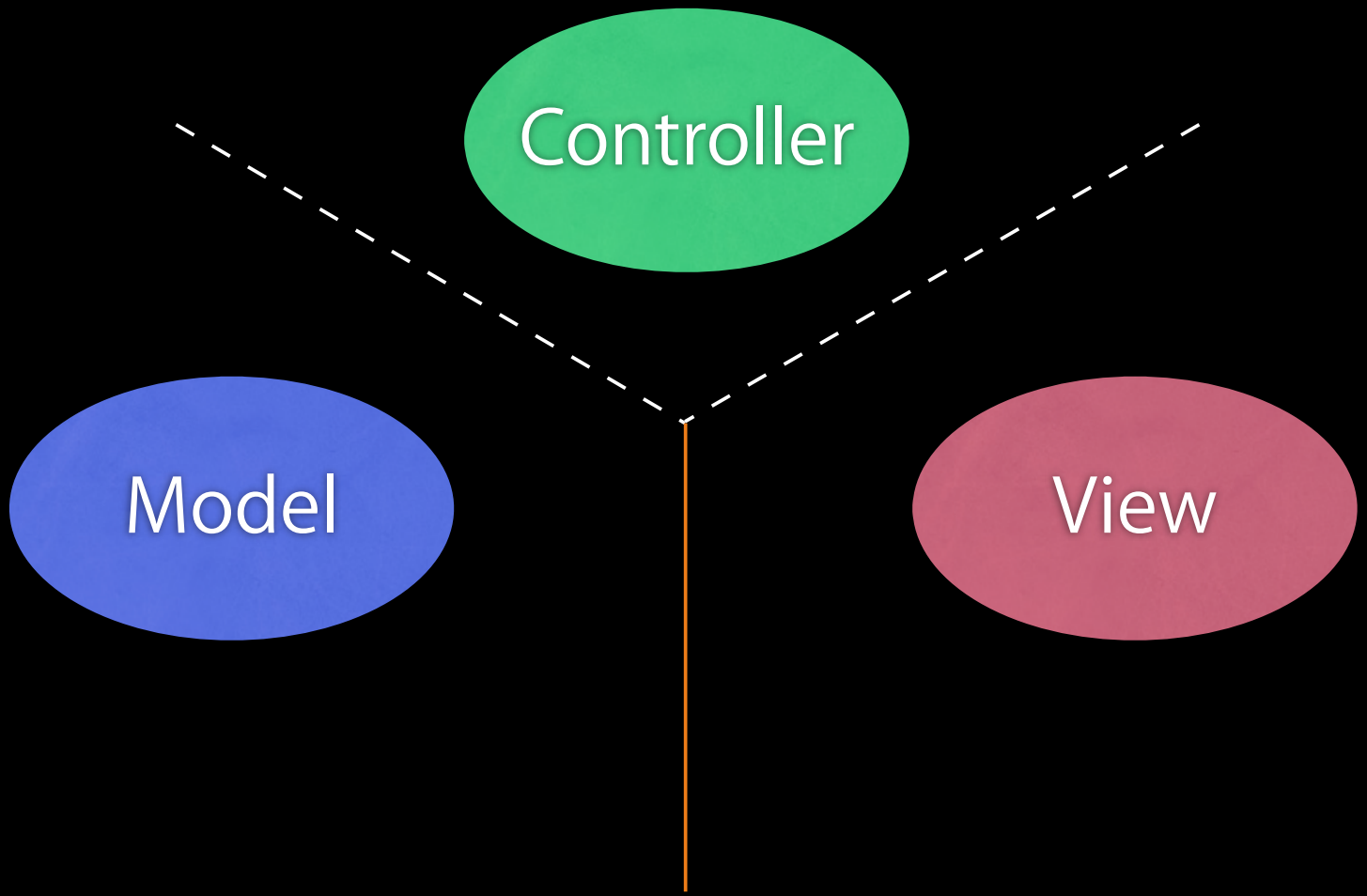
CS193E
Lecture 7

Document-based Applications
NSTableView
Key-Value Coding

Agenda

- Questions?
- Review: delegates, MVC
- Document-based apps
- Table views
- Key Value Coding

Model, View, Controller



Document-based Apps

Basic App Functionality

- Save documents
  - Saving commands: Save, Save As…, Revert
  - Window-modal save panel
- Open documents
  - Recent Documents menu
  - Restrict selection to application's document types
- Handle errors when saving or loading
  - Trying to save a read-only document
  - Trying to save to a read-only directory
- Open multiple files simultaneously
  - Stagger windows nicely to keep things tidy
  - Offer good default document names

Basic App Functionality

- Keep track of changes user has made
  - Prompt to save or discard when closing or quitting
  - Let them undo and redo changes
- Interact well with rest of the system
  - Double click on documents in Finder
  - Drag documents to app icon in the Dock

Cocoa provides this functionality

- Handles most document-based tasks ‘for free’
- You provide app-specific code
- Default behaviors can be customized

Demo

Simple Document-based application

Basic steps to a document-based app

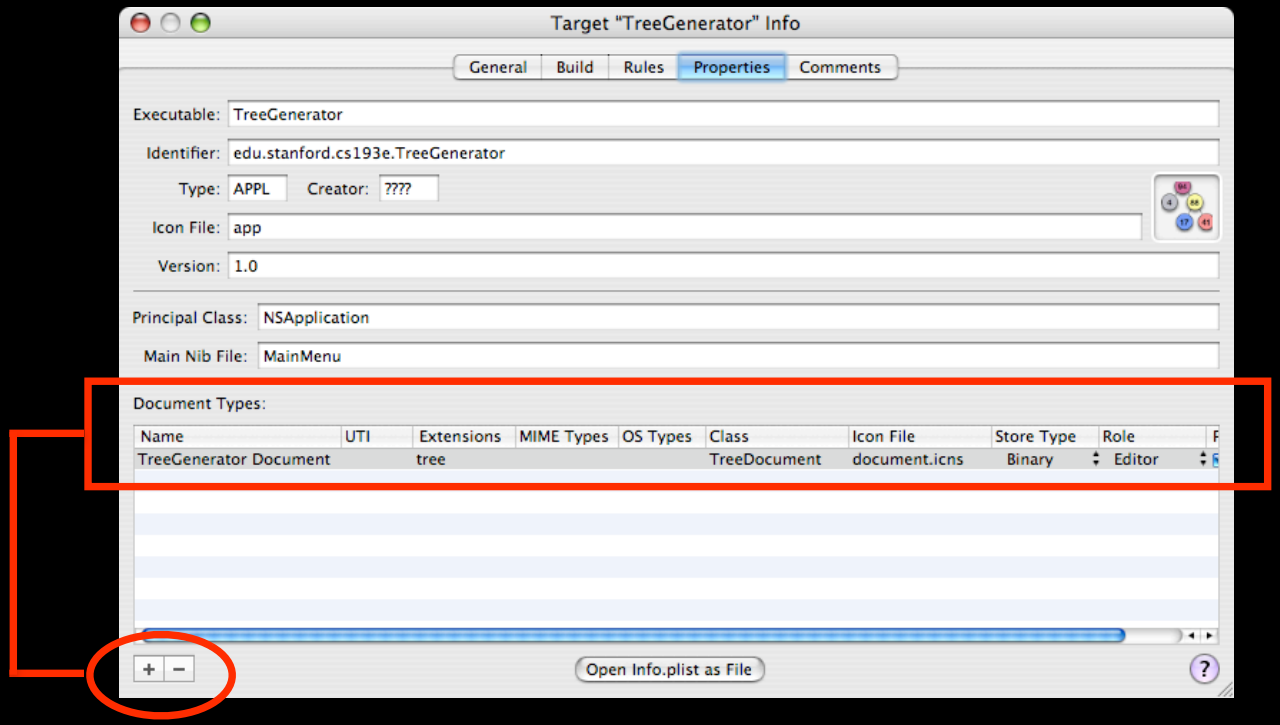
- Create document-based project in Xcode
- Set document type in Xcode
- Create application model
- Implement methods in NSDocument subclass
  - Unarchive loaded data
  - Archive model objects for saving
  - Create new model for new document
  - Implement dealloc

NSDocument Projects

- Start with new "Cocoa Document-based Application" in Xcode
- Creates a MyDocument class for you
  - Subclass of NSDocument, ready to use
  - Interesting methods stubbed out, ready for you to implement
- Separate nibs for main menu and document

Document Types

- In Xcode, bring up Target Info panel
  Menu item: Project > Edit Active Target



Writing Model Classes

- Create classes, typically in Xcode
- Define instance variables
- Write init methods (if needed)
  - Set default state of instance variables
- Write dealloc method (if needed)
  - Release any retained ivars
- Define the API for how to use the object
  - Frequently means write setters/getters
- Implement NSCoding protocol to archive objects into a file

Saving

- Cocoa asks your document for an NSData representing the document

```
- (NSData *)dataOfType:(NSString *)aType error:(NSError **)error
{
    // Create NSData from model objects
    // return nil and an NSError by reference if problem
}
```

- Implement NSCoding in your model object
- Use NSKeyedArchiver to archive your objects into an NSData
- We'll talk about NSError next time

Loading

- Cocoa gives your document an NSData to initialize from

```
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)aType
                                      error:(NSError **)error
{
    // Initialize model objects from contents of NSData
    // Return NO and an NSError by reference if problem

    return YES;
}
```

- Use NSKeyedUnarchiver to unarchive the document's root object, and all related objects
- You will usually set the unarchived model to an instance variable in your NSDocument subclass

Creating New Documents

- Cocoa handles most of the work by initializing NSDocument, loading document nib
- You override `windowControllerDidLoadNib:` and do "new document" initialization there

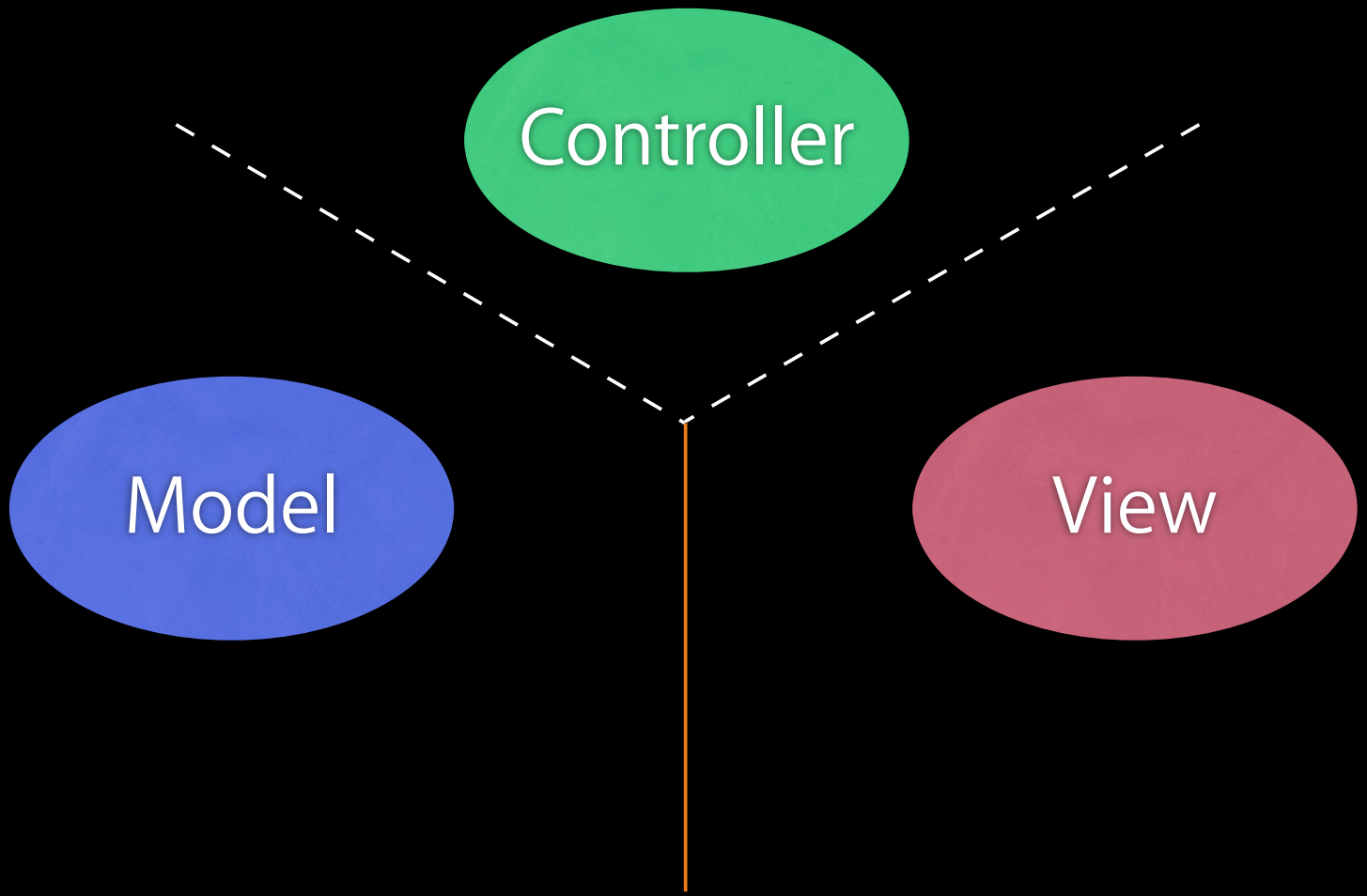
```
- (void)windowControllerDidLoadNib:(NSWindowController *)wc
{
    [super windowControllerDidLoadNib:wc];

    // if an existing model wasn't loaded from disk
    // model ivar still nil.  Make new model for new document
    if (myModel == nil) {
      myModel = [[MyModel alloc] init];
    }
}
```

Document memory management

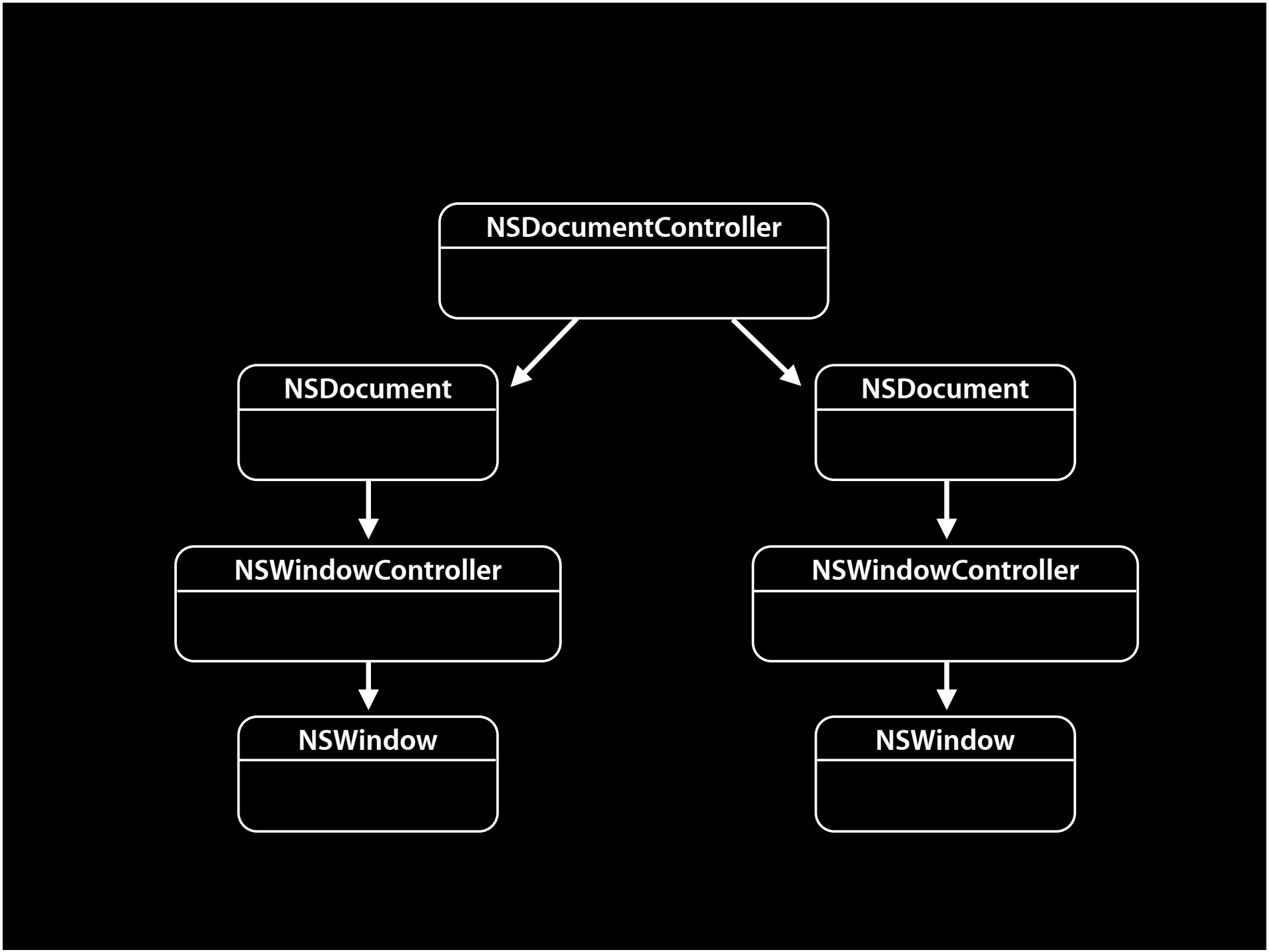
- Your document subclass holds onto the root of your model
  - retaining root object after unarchving
  - alloc'ing new root object for new document
- Don't forget to add a dealloc method

Model, View, Controller



Document-based Application

- Three major classes provide the functionality
  - NSDocumentController
  - NSDocument
  - NSWindowController
- In many cases, you only need to work with NSDocument


NSDocumentController
NSDocument
NSDocument
NSWindowController
NSWindowController
NSWindow
NSWindow

NSDocumentController

- Shared instance per application
- Manages the array of NSDocuments
- Tracks
  - Open documents and maintains uniqueness
  - Recently used documents list (persistent property)
  - Current document and its current directory
  - Edited document status check on Application terminate
- Uses document types plist from Application bundle
  - Definition of valid document types for the application
  - The NSDocument subclass for a given type

NSDocument

- Built-in framework provides most of the functionality
- Handles general app behavior
  - Document open / close / print
  - File / Edit / Window menus
- Reads and writes data to and from files
- Provides hooks for you to implement code that is specific to your document types
- Subclassed to handle your document type

Responder Chain

We've already looked at the simple case

1. The main window's first responder
2. Superviews of the first responder (up to the window's content view)
3. The main window itself
4. The main window's delegate
5. NSApp
6. NSApp's delegate

Responder Chain

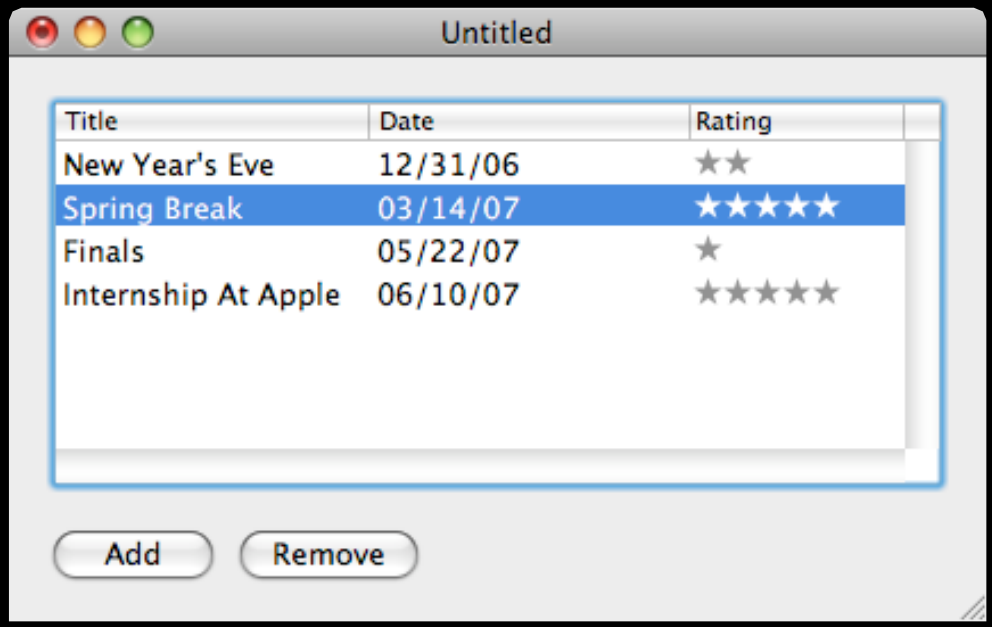
More places to provide custom behavior

1. The main window's first responder
2. Superviews of the first responder (up to the window's content view)
3. The main window itself
4. The main window's delegate
5. The window's NSWindowController
6. The NSDocument object (if different from window's delegate)
7. NSApp
8. NSApp's delegate
9. NSApp's NSDocumentController object

Table Views

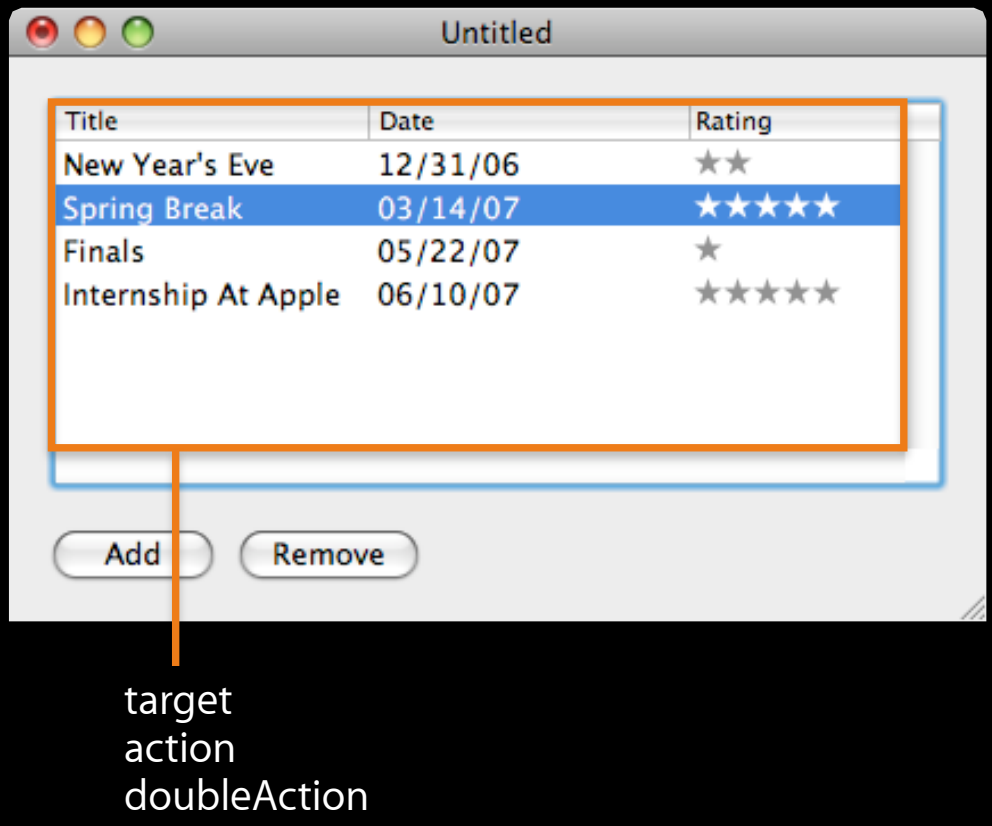
NSTableView

Presents data in a row-based format



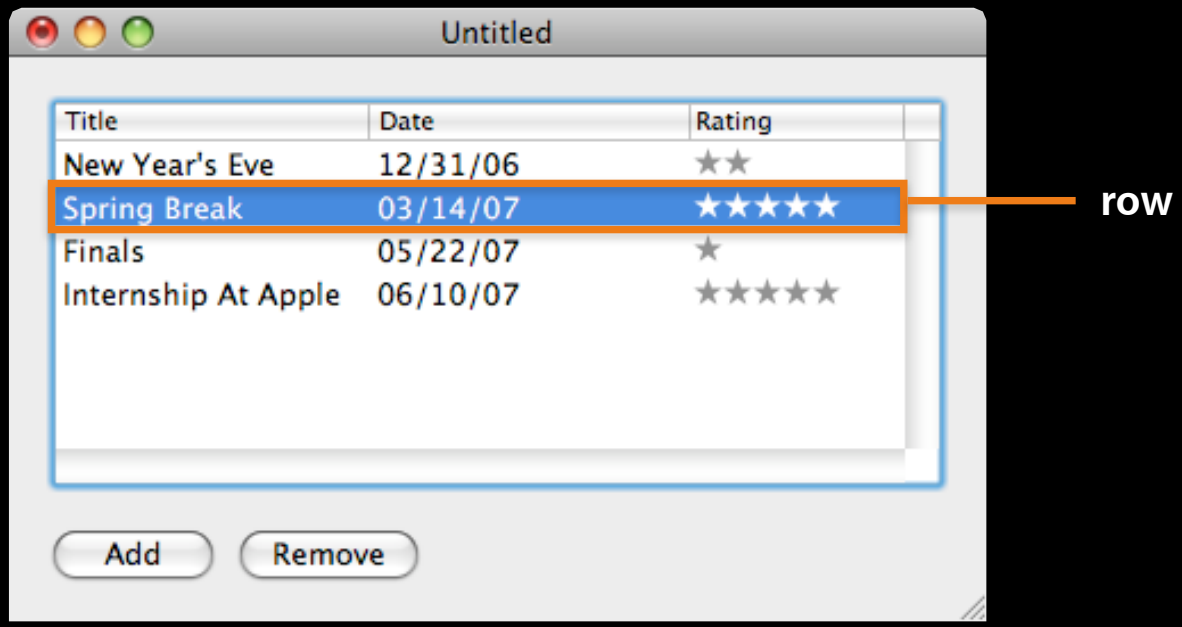
NSTableView

Presents data in a row-based format



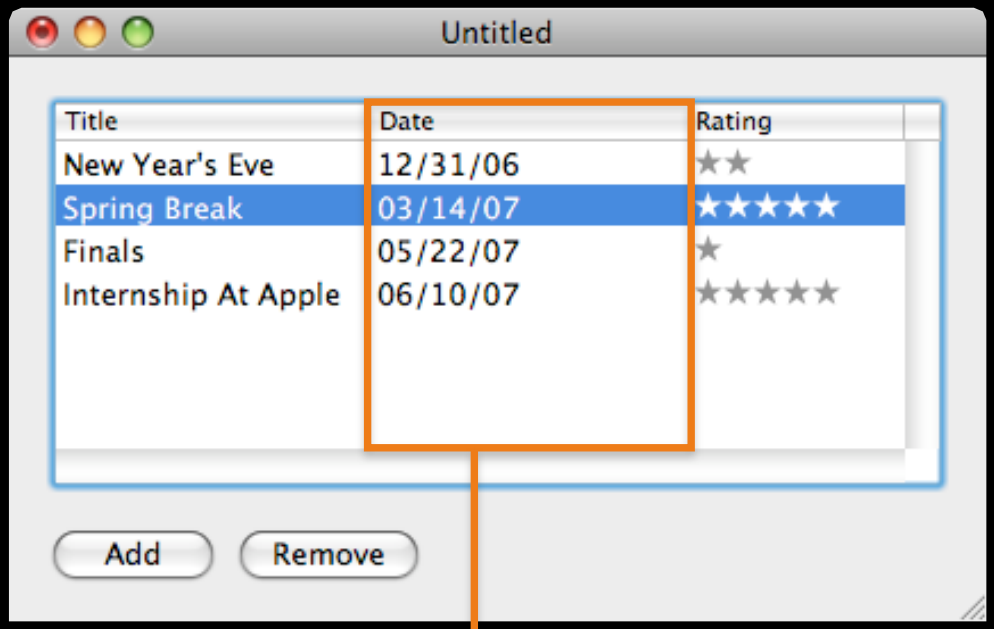
NSTableView

Presents data in a row-based format



NSTableView

Presents data in a row-based format



Commonly used table view methods

Telling the table view to reload data

```
[myTable reloadData];
```

Informing table view that number of rows has changed

```
[myTable noteNumberOfRowsChanged];
```

Finding out the selected row

```
selection = [myTable selectedRow];
```

Programmatically selecting and scrolling

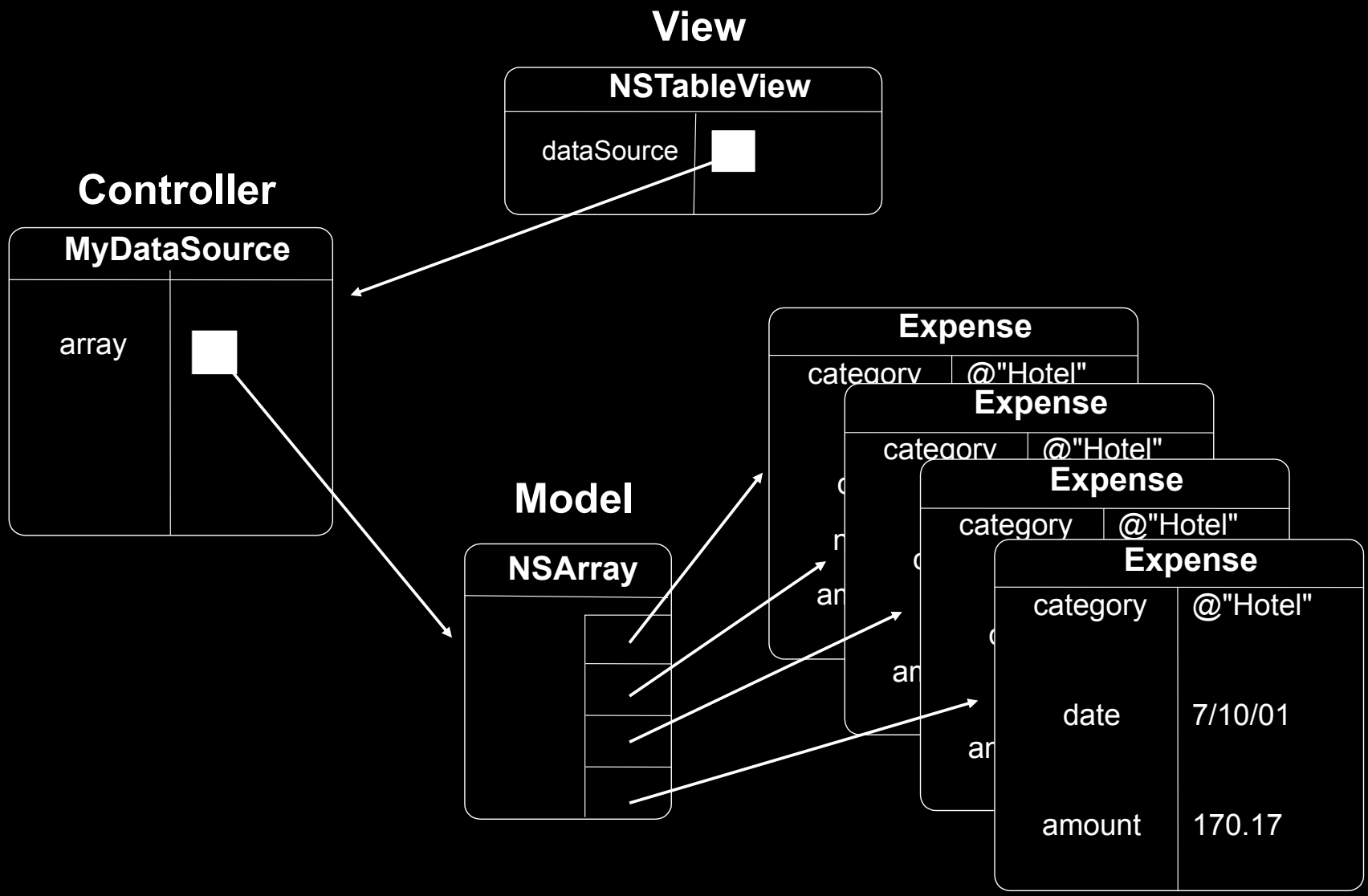
```
[myTable selectRow:row byExtendingSelection:NO];
[myTable scrollRowToVisible:row];
[myTable scrollColumnToVisible:column];
```

Where does a table view get its data?

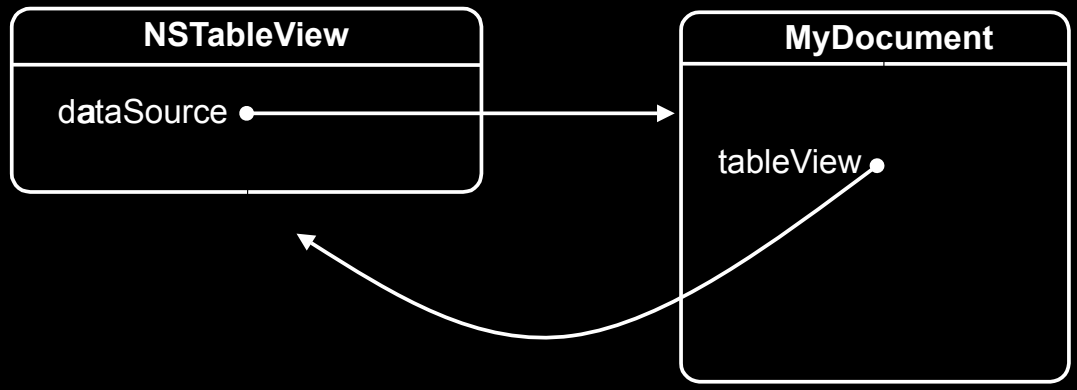
The data source provides the table view with data

- Similar to a delegate, a data source is a helper object
- Data source implements an informal protocol
- Methods are declared in NSTableView.h
- You implement these methods to do the following:
  - get number of rows
  - get value for row and column
  - set value for row and column
  - other data source tasks (drag and drop, for example)

A data source is a controller



Data source methods




```
- (int)numberOfRowsInTableView:(NSTableView *)tableView

- (id)tableView:(NSTableView *)tableView
      objectValueForTableColumn:(NSTableColumn*)column
      row:(int)row

- (void)tableView:(NSTableView *)tableView
      setObjectValue:(id)object
      forTableColumn:(NSTableColumn *)column
      row:(int)row
```

Providing number of rows

```
- (int) numberOfRowsInTableView:
                            (NSTableView *) tableView
 {
      return [expenseArray count];
 }
```

Providing an object value to a table view

```
-(id)tableView:(NSTableView *)tableView
    objectValueForTableColumn:(id)column
    row:(int)row
{
      Expense *expense;
      NSString *identifier;


      expense = [expenseArray objectAtIndex:row];
      identifier = [column identifier];

      if([identifier isEqualToString:@"category"])
         return [expense category];
      else
         return [expense date];
}
```

Updating edited object value

```
- (void)tableView:(NSTableView *)tableView
    setObjectValue:(id)object
    forTableColumn:(NSTableColumn *)column
    row:(int)row
{
    Expense *expense;
    NSString *identifier;

    expense = [expenseArray objectAtIndex:row];
    identifier = [column identifier];
    if ([identifier isEqualToString:@"category"])
            [expense setCategory:object];
    else
            [expense setDate:object];
}
```

NSTableView also has a delegate

- Often you set the same object as data source and delegate
- Delegate methods control table view behavior
- Data source methods focus on providing and affecting data

Table view delegate methods

```
- (BOOL)selectionShouldChangeInTableView:(NSTableView *)
tableView

- (BOOL)tableView:(NSTableView *)aTableView
shouldEditTableColumn:(NSTableColumn*)aTableColumn row:
(int)rowIndex

- (BOOL)tableView:(NSTableView *)aTableView shouldSelectRow:
(int)rowIndex

- (void)tableView:(NSTableView*)tableView
didClickTableColumn:(NSTableColumn *)tableColumn

- (void)tableView:(NSTableView*)tableView
didDragTableColumn:(NSTableColumn *)tableColumn

- (void)tableView:(NSTableView*)tableView
mouseDownInHeaderOfTableColumn:(NSTableColumn*)tableColumn
```

Key Value Coding

Accessing properties using keys

Key Value Coding (KVC)

- Generic way for properties of objects to be accessed (by key)
- Instead of:

  `[shape fillColor]` and `[shape setFillColor:color]`

- One could do:

  `[shape valueForKey:@"fillColor"]` and

  `[shape setValue:newColor forKey:@"fillColor"];`

- Conceptually every object becomes a dictionary

Properties

- KVC allows access to all object "properties"
- Properties are:
  - Attributes: Simple, immutable values like BOOLs, ints, floats, strings… (scalar data types)
  - Relationships: references to other objects which have properties of their own
    - to-one: single object (e.g. outlet in IB, NSWindow's contentView)
    - to-many: one or more objects (e.g. an array of objects, NSView's subviews)

Getting values via KVC

- Given a key, you get a value back (or nil)

```
- (id)valueForKey:(NSString *)key
```

- Scalar types such as BOOL and int are "boxed" automatically in NSNumbers
- Structs such as NSRect are boxed in NSValues.
- Example:

```
NSColor *fillColor = [shape valueForKey:@"fillColor"];
NSNumber *showBorder = [shape valueForKey:@"showBorder"];
```

Setting values via KVC

- Given a value, you set it on an object using

```
- (void)setValue:(id)value forKey:(NSString *)key
```

- Scalar types such as BOOL and int are "unboxed" automatically

```
[shape setValue:[NSColor redColor] forKey:@"fillColor"];
[shape setValue:[NSNumber numberWithBool:YES]
                                    forKey:@"showBorder"];
```

From Keys to Values

- NSObject's implementation of valueForKey: will
  - Search for a public accessor method based on "key". For example, `[person valueForKey:@"firstName"]` will try to find `[person firstName]`
  - Search for an instance variable based on "key". For example, `_firstName` or `firstName`
- If no value is found, an exception is thrown
- There are additional nuances to what is searched, see NSKeyValueCoding.h for details

From Keys to Values

- Setting values works in a similar fashion
  - Search for a set<Key>: method,
    `[person setValue:@"Bob" forKey:@"firstName"]` will try to find
    `[person setFirstName:@"Bob"]`
  - Try to find corresponding instance variable with name _<key> or <key>. For example, `_firstName` or `firstName`.
- If value cannot be set, exception is thrown
- There are additional nuances to what is searched, see NSKeyValueCoding.h for details

Key Paths

- Keys can be chained together to access nested object properties
- For example, if document has a selectedItem property we can access the item's first name:

```
NSString *name;
name = [document
valueForKeyPath:@"selectedItem.firstName"];
```

- Equivalent to:

```
name = [[document selectedItem] firstName];
```

- Corresponding setter methods:

```
[document setValue:name
        forKeyPath:@"selectedItem.firstName"];
```

Pros and Cons

- Provides a very generic way to manipulate objects
  - Can create very flexible, general solutions
  - Can program more generically
- Can be difficult to debug
  - No compiler type checking - everything is typed id
  - Can mistype keys - they are simply strings

Key Value Coding is everywhere

- We can use it to simplify our table view's data source
  - Each table column has an identifier, which can be a key
  - Using KVC, we can get and set column values accordingly
- Enables other technologies yet to be discussed
  - Cocoa Bindings
  - Core Data
  - Scriptability
- Making your model classes KVC compliant is an important step
- See comments in NSKeyValueCoding.h for details

KVC simplifies the data source

```
- (id)tableView:(NSTableView *)tableView
    objectValueForTableColumn:(id)column
    row:(int)row {

      Expense *expense;
      NSString *identifier;


      expense = [expenseArray objectAtIndex:row];
      identifier = [column identifier];

      return [expense valueForKey: identifier];

}
```

KVC simplifies the data source

```
- (void)tableView:(NSTableView *)tableView
    setObjectValue:(id)object
    forTableColumn:(NSTableColumn *)column
    row:(int)row
{
    Expense *expense;
    NSString *identifier;

    expense = [expenseArray objectAtIndex:row];
    identifier = [column identifier];
     [expense setValue:object forKey:identifier];
}
```

Questions?