Chapter 4

CUSTOM VIEWS

CHAPTER 4 CUSTOM VIEWS

Goal

To gain familiarity with custom display and event handling techniques used to implement custom views.

Prerequisites

Exposure to basic view and event concepts including: the view hierarchy, the event loop, and the role of NSResponders in handling mouse and keyboard events.

Objectives

At the end of this section, you will be able to:

- Describe a view in terms of its important attributes—frame, coordinate system, display methods and the designated initializer
- Identify the main techniques for drawing including PostScript functions, wrappers and NSImage handling
- Explain how custom views handle mouse and keyboard events making use of NSEvent data
- List the steps for implementing a custom view and incorporating an instance in an application

Reading

NSView class reference in the Application Kit

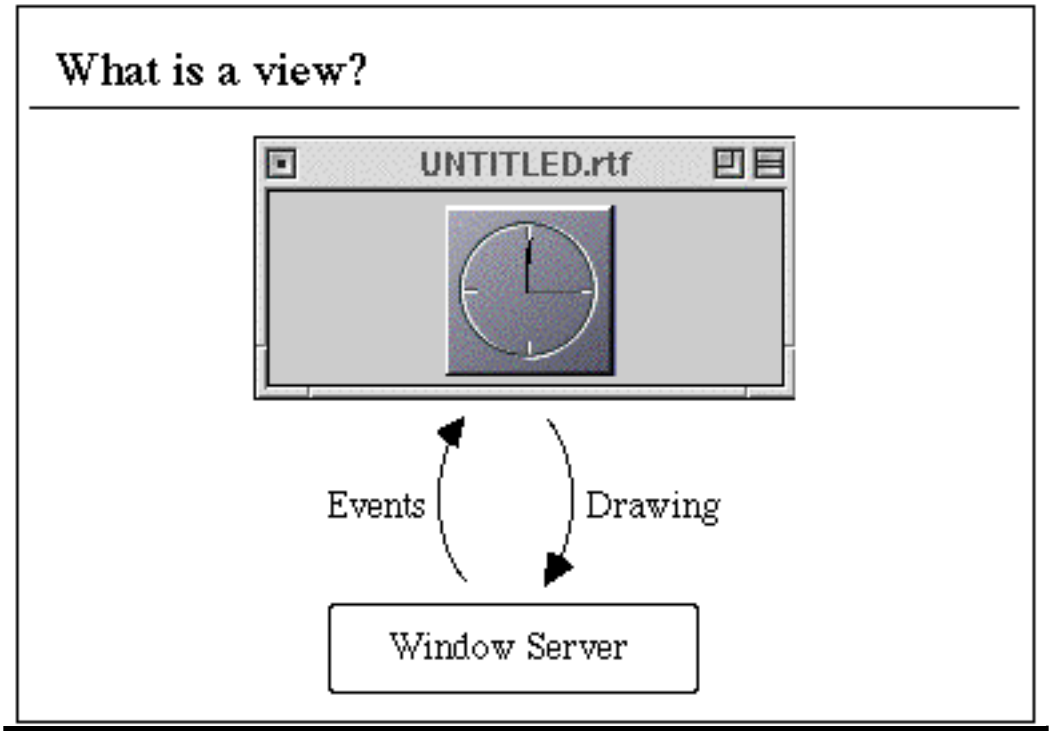
NSResponder class reference in the Application Kit

NSEvent class reference in the Foundation

NSImage class reference in the Application Kit

NSColor class reference in the Application Kit

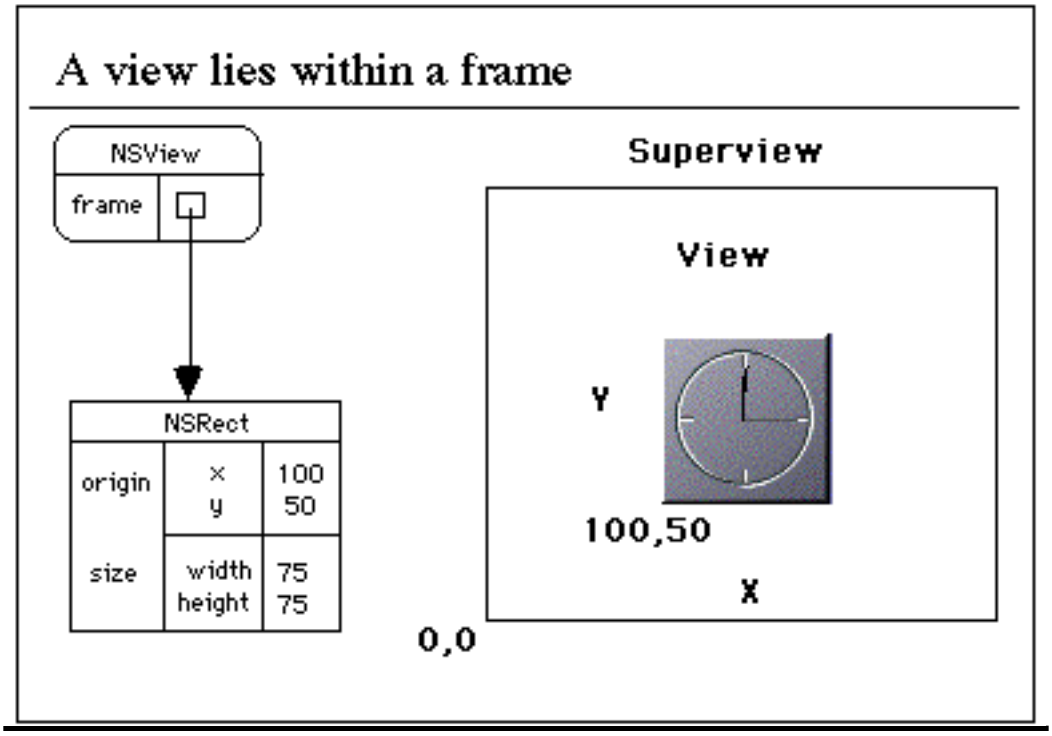
NSString class reference in the Foundation



What is a view?

A view is an area of responsibility within a window. It occupies visible real estate where it displays itself using custom drawing. Anything you can see on a window is, by definition, a view object. Within its boundaries, a view also handles events—mouse clicks and, if first responder, keystrokes. A view is a subclass of NSView and is used to display images, text, or the dynamic appearance of a custom control. Mouse and keyboard event handling is used to implement editable views, drag and drop participants, and controls. NSControl is a subclass of NSView and provides the abstract behavior for implementing target/action.

The Window Server provides all the necessary support for servicing hardware events and queueing them for NSApplication. In conjunction with Application Kit classes—NSApplication, NSWindow and NSView—the Window Server provides the necessary interfaces for a custom view to draw. Like most windowing environments that manage multiple applications on the screen simultaneously, most of this functionality resides in a separate server process; this process is called the Window Server.



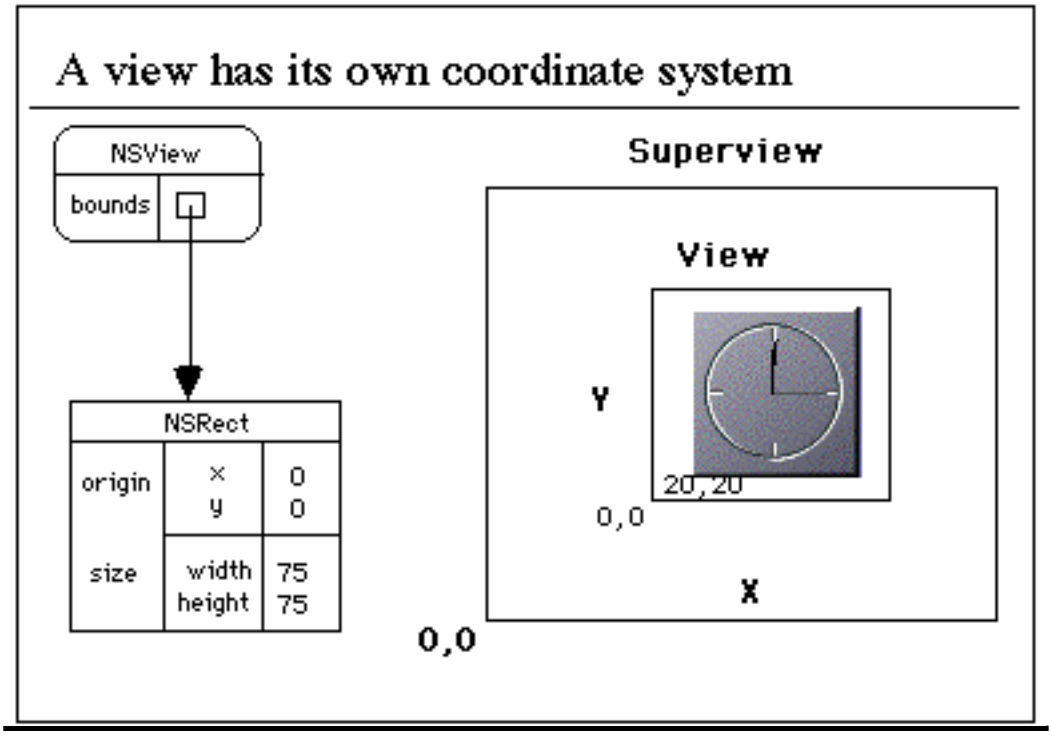
A view lies within a frame

A view lives within its superview and occupies a rectangular region called a frame. This is an attribute of NSView. Every view uses a coordinate system to refer to points within its frame. The origin is the lower left corner and defaults to the x and y coordinates 0,0. In this case, the x and y axes extend in a positive direction to infinity. The frame rectangle is used to create a clipping path so that, despite a theoretically infinite plane, a view cannot scribble outside the lines. A view's frame is described by the NSRect structure and contains:

- origin—x and y coordinates in the super view indicating the view's lower left corner
- Size—width and height

A view's frame is established by NSView's designated initializer, **initWithFrame:**. You typically configure the origin and size of a view with graphical positioning in Interface Builder. The frame can be adjusted at runtime to move the view and even change its size.

The coordinate system points such as x and y are floating point numbers where each whole unit corresponds to approximately 1/72 of an inch.



A view has its own coordinate system

While a view's frame describes where it lives within the superview, it is generally not used by the view itself. A view has its own coordinate system which, by default, has an origin of 0,0. An alternate rectangle is used by the view for addressing points within its frame and this is the called the bounds rectangle. It is another attribute of NSView. When the view addresses points within its bounds, the underlying system takes care that it is mapped to the right place within the superview, within the window and within the screen. Whatever mapping the window server uses, a view can conveniently work with its own simplified world view.

A view object encapsulates the display and event handling of a given region on the window. The NSView abstract superclass encapsulates most of the mechanics of clipping and coordinate system mapping. Your custom view is free to concentrate the specialized details that make it unique and useful.

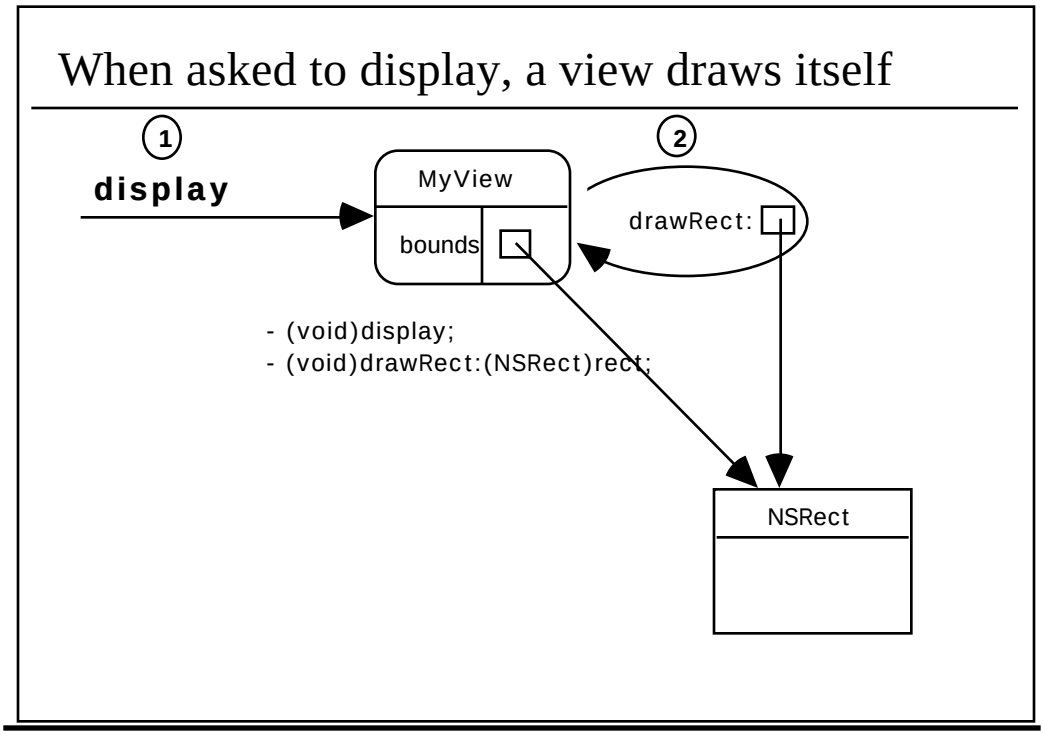
Important view data structures

```
typedef struct _NSPoint {
     float x, y;
} NSPoint;


typedef struct _NSSize {
     float width, height;
} NSSize;


typedef struct _NSRect {
     NSPoint origin;
     NSSize size;
} NSRect;
```

Important view data structures

Most view handling involves a few basic data structures used to describe rectangles and points within them. The coordinate system uses floats. Any point is an x and y pair of which the origin is a fundamental example. A rectangle contains an origin and a description of its size, again two floats, which describes its width and height in the same units.

These are defined in the Foundation kit's **NSGraphics.h** and used extensively by view methods and C functions that provide graphics support.



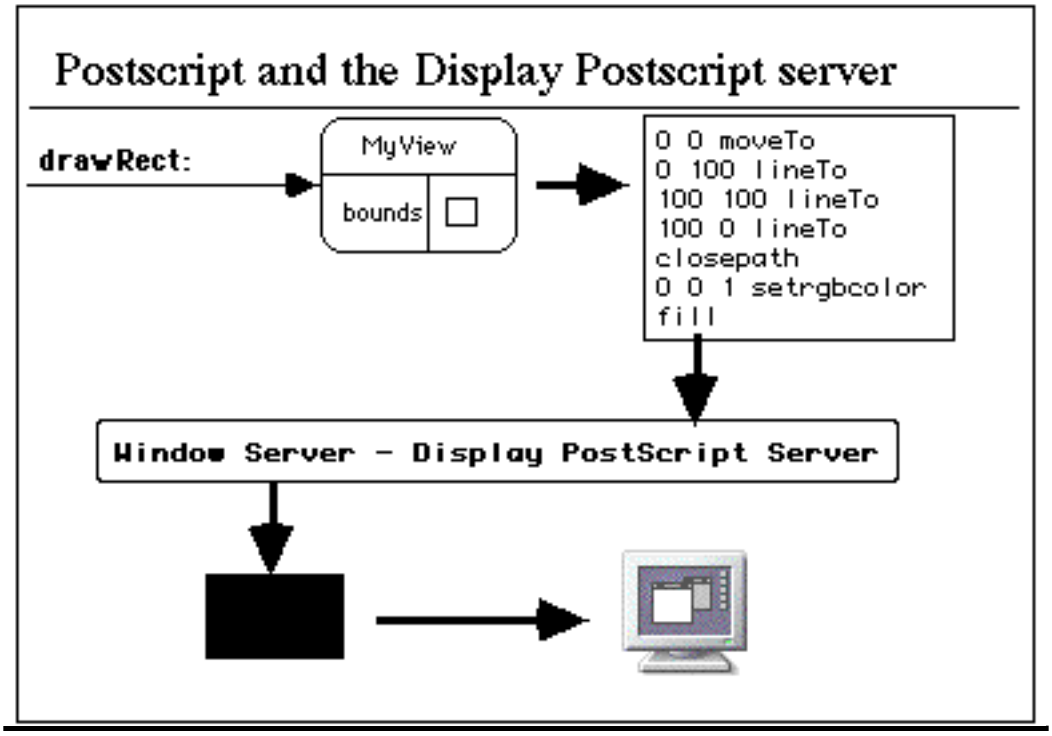
When asked to display, a view draws itself

To display a view, send it the **display** message. It is fully implemented in the NSView superclass to manage all genericWindow Server setup required to put your view in focus—give it exclusive access for drawing. You do not need to override it. NSView concrete subclasses implement their custom drawing behavior in the **drawRect:** method, sent by **display** once the Window Server setup is done. To implement a custom view you must override **drawRect:**. The inherited version from NSView does nothing, leaving the view invisible.

Due to necessary setup in display, you should never call **drawRect:** directly—rather, initiate drawing by sending **display**. Usually the containing objects such as NSWindow and NSApplication send the proper display messages at the necessary time. Another important behavior in **display** is that it will recursively send **display** to all subviews within the view hierarchy. Code inside your view might change state requiring that your view be re-displayed. You can mark your view with **setNeedsDisplay:**, passing YES. NSApplication will display any views and windows that need it after every event. Your view can send itself the **display** message directly in which case drawing happens more or less immediately.

If your view must perform dynamic drawing in methods other than **display** and a call to **display** is unnecessary or inefficient, it can manually perform the required Window Server setup and call **drawRect:** directly:

```
[self lockFocus];
// dynamic drawing here
[[self window] flushWindow];
[self unlockFocus];
```

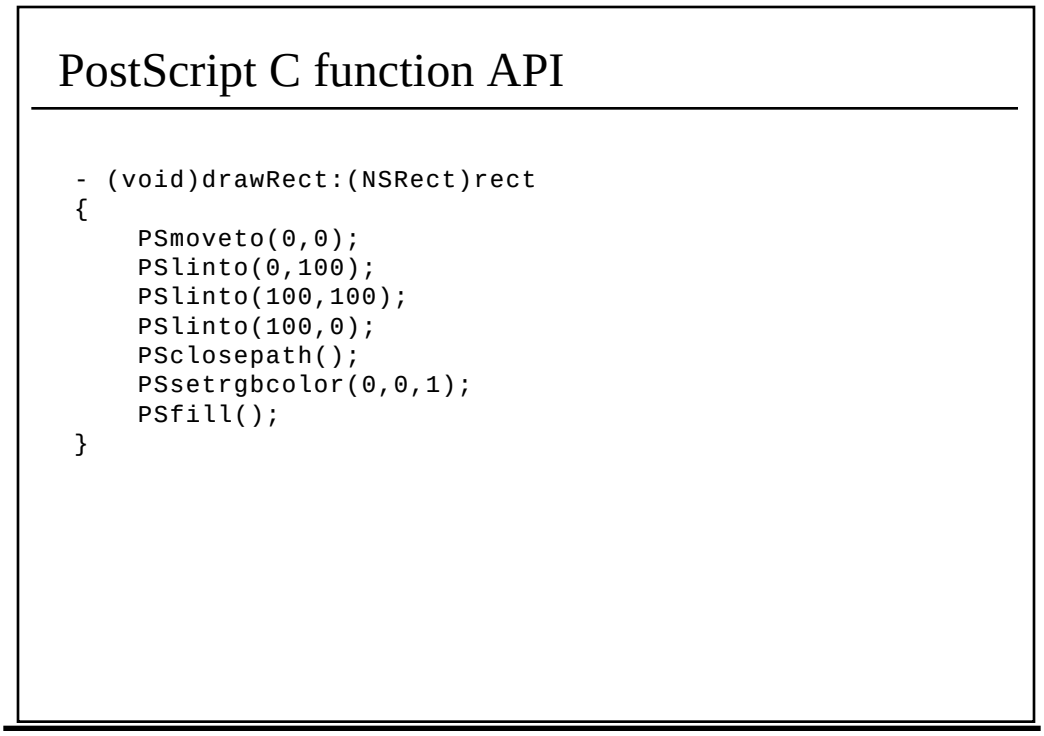


PostScript and the Display PostScript server

To implement custom drawing behavior, your NSView subclass overrides **drawRect:**. How do views implement drawing? The underlying drawing mechanism is the Display PostScriptâ interpreter—part of the Window Server process. It is driven by Display PostScript, a very general and powerful programming language that provides the ability to construct graphics using points, lines, curves, colors, painting, text and even sampled images. Originally developed by Adobeâ, PostScriptâ is a widely used standard with features for describing document layout, suitable for sophisticated formatting and printing.

All display tasks are implemented with PostScript and the Window Server. Applications communicate with the server, sending a stream of PostScript code to do the job. All the Application Kit objects encapsulate their own drawing so you need worry only about your own view. In addition, while it is possible and even common for views to incorporate custom PostScript code, there are a number of convenient object and function API that encapsulate PostScript into a high-level general purpose interface.

If you plan to use custom PostScript code, there are a number of ways to do so.



PostScript C function API

Because your view class is written in Objective-C, you are not able to in-line PostScript code directly. Instead, you might use the provided C function API which essentially includes a wrapper for each PostScript operator. The naming scheme is simple. For any PostScript operator, just prepend "PS"—**moveto** becomes **PSmoveto**(). Real PostScript is stack-based and coded using postfix notation. The operator comes after the operands. The C functions provide a C function calling interface with operands coming after the function.

Each function maps to a single PostScript operator and results in a one-to-one communication with the Window Server process. If your view uses lots of PostScript for drawing, the communications overhead can become quite inefficient. For better performance, you will want to use Post Script wrappers using **pswrap**.

Rectangle convenience functions

NSEraseRect(NSRect rect);

NSRectFill(NSRect rect);

NSDrawGrayBezel(NSRect rect, NSRect clipRect);

NSDrawGroove(NSRect rect, NSRect clipRect);

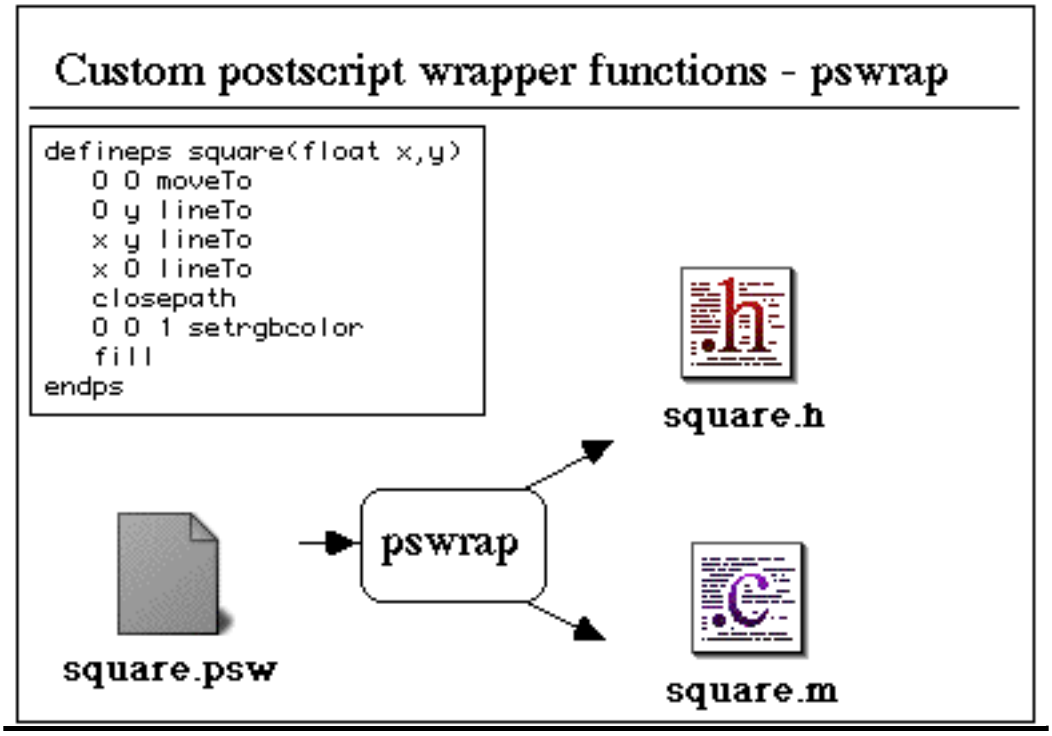
NSDrawButton((NSRect rect, NSRect clipRect);

NSHighlightRect(NSRect rect);

Rectangle convenience functions

There are a number of higher level and more generalized C API for drawing. Each is a wrapper for a block of parameterized PostScript that accomplishes the indicated function. Especially useful for custom controls, they provide you with the ability to clear your entire view, fill it with color, draw traditional control and button borders for a crisp 3-dimensional look, even to highlight your rectangle in a consistent manner.

These are a few useful examples. You will find them and others in the Application Kit's **NSGraphics.h**.



Custom postscript wrapper functions—pswrap

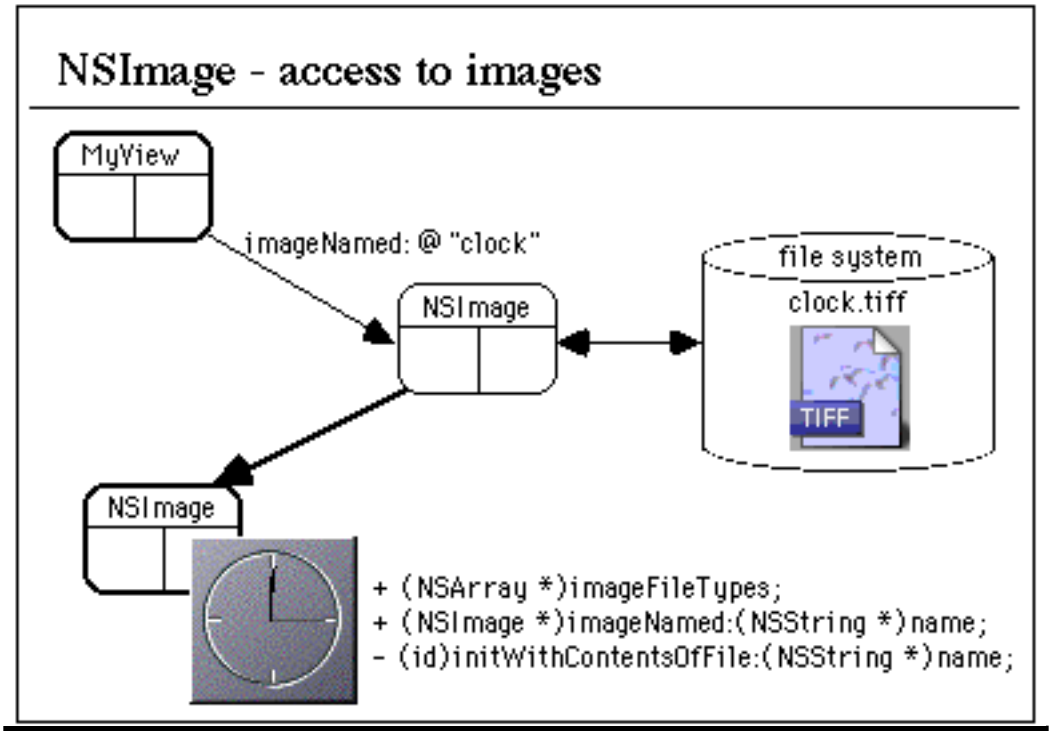
It is likely you will want to implement your own C interfaces of this sort—more general and highly efficient display functions that require multiple PostScript operations, conditionals, loops, and all the general features provided by the full-featured PostScript language. For this you can use PostScript wrappers.

A PostScript wrapper is simply a block of literal PostScript, surrounded by **defineps** and **endps** delimiters to create a parameterized function. One or more wrappers can be stored in a file with the extension **.psw** and added to your project in Project Builder's "Other Sources" suitcase. When building, Project Builder automatically invokes the pswrap utility which generates conventional C .h and .c files. Your custom view can import the header which defines the wrapper prototype, and call it directly as any C function:

```
#import "square.h"

- (void)drawRect:(NSRect)rect
{
    square(rect.origin.x, rect.origin.y);
}
```

By providing a clean C interface complete with parameters, pswrap allows you to create efficient, generalized and reusable display functions.

Why are wrappers more efficient than the individual PostScript C function operators—like PSmoveto()? The literal PostScript inside a wrapper is compiled into binary form and when invoked, is sent to the server in a single request. Anytime you batch multiple data transport operations into a single larger one, you get a bulk discount—you get better performance.



NSImage—access to images

Custom views commonly use images that are not dynamically drawn, but captured once, stored in a file and loaded into the view when necessary. Often called sampled images, they are typically created with one of many different drawing programs and stored using one of many different types of encodings—TIFF, EPS, GIF, and so forth. While the Display PostScript language directly supports image handling, there are also true object-oriented API with the NSImage object. An NSImage instance represents a sampled image and encapsulates all the necessary functionality for using them in your view—loading from a file (or NSData, or Pasteboard etc.), rasterizing in preparation for display and drawing itself at a given point in your view's coordinate system.

» **imageFileTypes**—returns an array of supported image file extensions

» **imageNamed:**—loads the image from the main application bundle

» **initWithContentsOfFile:**—loads the image from an explicit pathname

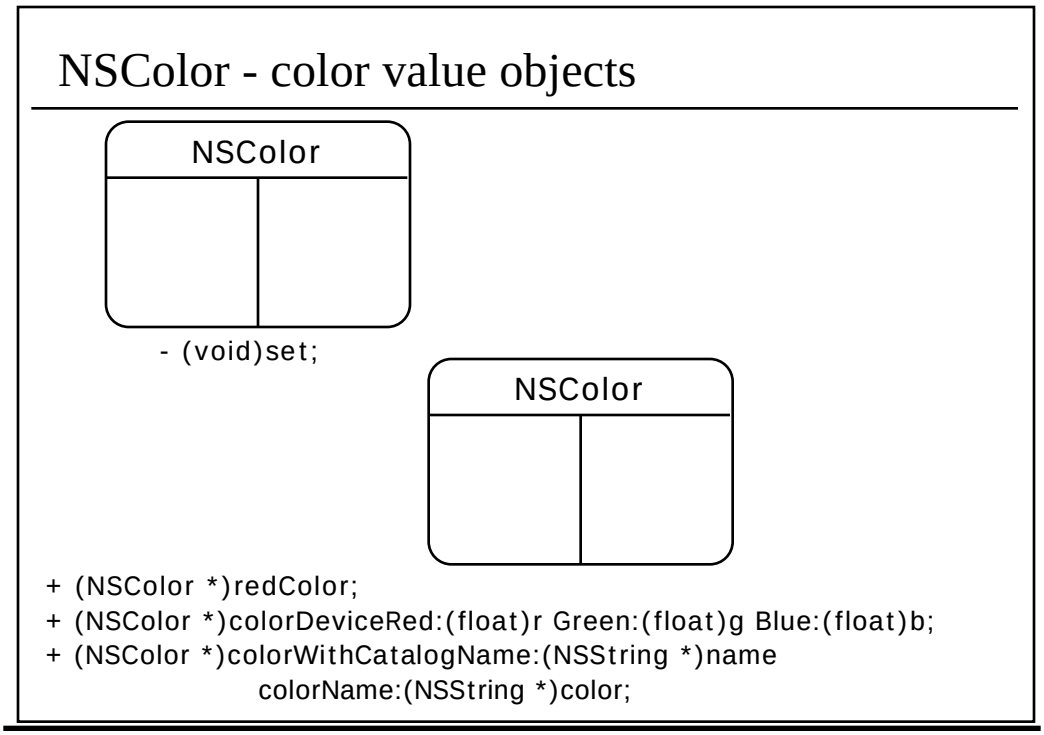
Compositing images in your view

```
- (id)initWithFrame:(NSRect)rect
{
    [super initWithFrame:rect];
    image = [NSImage imageNamed:@"clock"];
    [image retain];
    return self;
}

- (void)drawRect:(NSRect)rect
{
    [image compositeToPoint:rect.origin
        operation:NSCompositeSourceOver];
}
```

Compositing images in your view

NSImage instances can draw themselves in your view. This is called compositing. You supply a point in your coordinate system which indicates the lower left corner of where the image will be composited. You also supply a composite operator. There are 14 different operators that control how two images are mixed together to create a composite—the original underlying image in your view and the new image receiving the composite message. To completely replace the old view contents with a new image, use NSCompositeSourceOver. This is the most commonly used operator.

NSImage also supports dissolving—compositing an image with a variable degree of transparency—with **dissolveToPoint:fraction:**. The floating point fractional value ranges between 0 and 1 and indicates how much of the image should be composited. The smaller the value, the more the previous underlying image will show through. This creates a hybrid picture composed of a mixture between the original image and the new one receiving the dissolve message. By repeatedly dissolving with changing fractional values, the old image appears to dissolve into the new one. This creates a simple and pleasant form of animation.



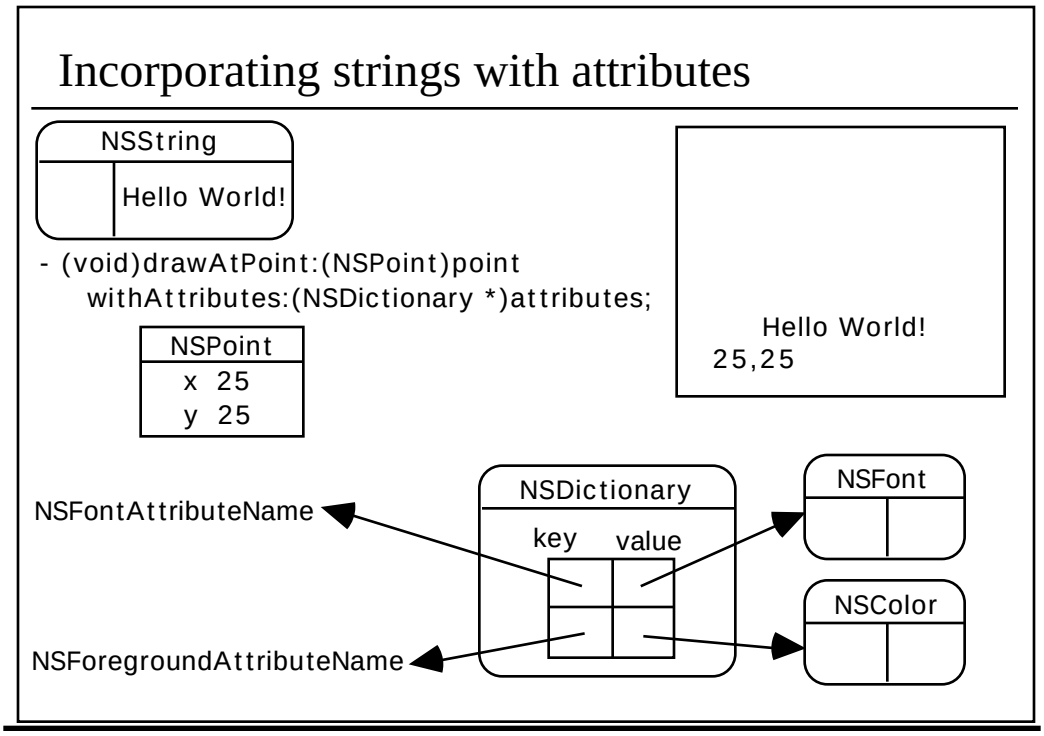
NSColor—color value objects

PostScript drawing allows you to easily create lines, curves, closed shapes, painted or "filled" areas, text and so forth. What determines what color will be used for the drawing or filling? PostScript uses a context that defines the current drawing state. Besides the specific view instance "focused" for exclusive access to the server, the context includes a number of attributes that apply to drawing operations—color, font, orientation and scaling of the coordinate system and so on. While you might manipulate these attributes directly in PostScript code, there is also a higher level object-oriented interface with NSColor.

NSColor has API for obtaining specific color value objects using a variety of different specifications. Certain methods return standard named colors—**redColor**, **blueColor**. Alternate methods allow you to specify colors in terms of red, green and blue values or by requesting a named color from an specific color catalog. Colors can be passed as arguments to drawing functions, or used to set the current color in the PostScript context with **set**. This code fragment that paints your entire view red:

```
[[NSColor colorRed] set];
NSRectFill([self bounds]);
```

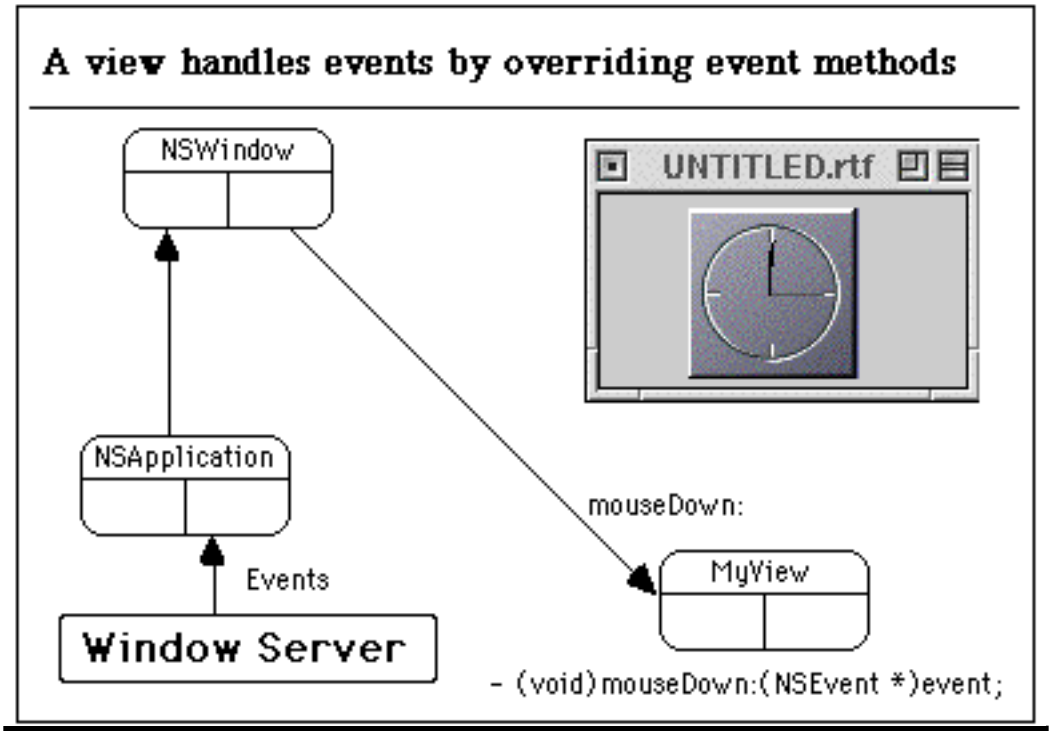
NSFont provides similar ease of handling whether passing a font as an argument or setting the current font in the PostScript context. Note, NSColor and NSFont are considered to be immuttable value objects. They can be archived, unarchived and copied.



Incorporating strings with attributes

Incorporating text in your view is straightforward with Application Kit additions to NSString. NSStrings can draw themselves given a point in your coordinate system, and a set of attributes that apply. Attributes are supplied in a dictionary of key-value pairs. Various attributes are supported by NSString including NSFontAttributeName and NSForegroundColorAttributeName.

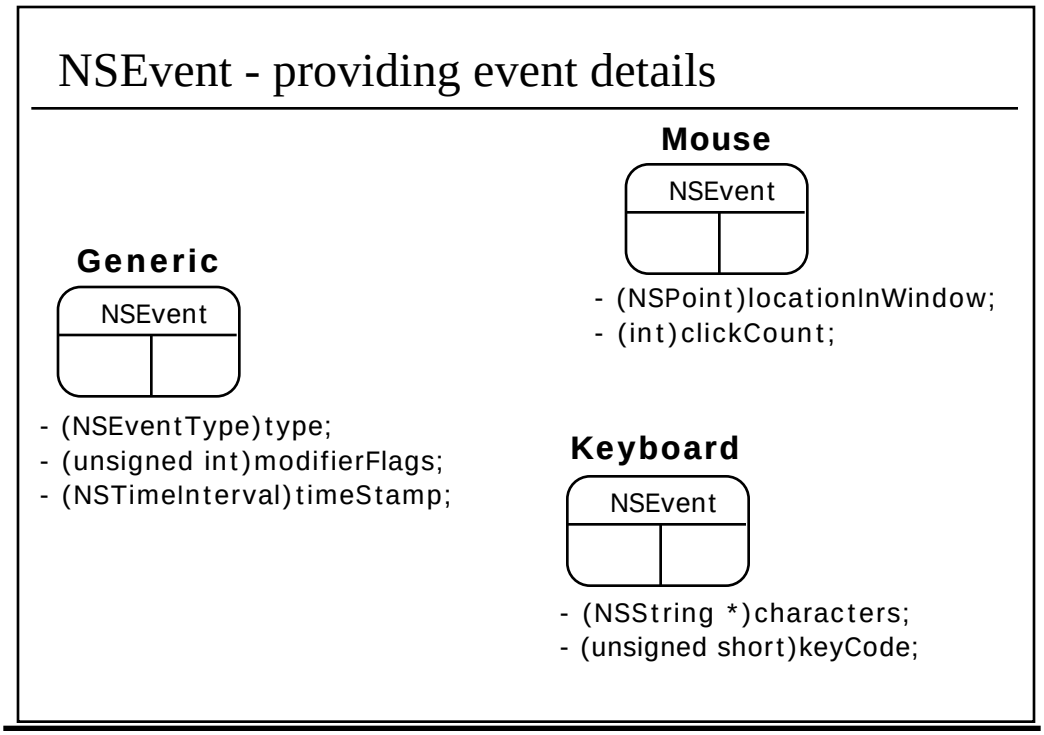
The Foundation includes a class that combines text and attributes in one object, NSAttributedString. It supports the ability to associate different attributes with different ranges within a string down to the granularity of a single character. A widely used object, NSAttributedString can also draw itself in your view using API similar to NSString. NSFormatters can provide their cells with attributed strings to incorporate color or font changes. NSTextView relies on NSTextStorage, a subclass of NSAttributedString used for storing entire rich text documents.



A view handles events by overriding event methods

NSView is a subclass of NSResponder. A view claims a visible area that may also interact with the user, providing control and dynamic behavior through user events. User events such as mouse and keyboard actions are transmitted from the window server to a specific application through its NSApplication instance. From there, the event is mapped to a window instance which in turn locates the appropriate view in its view hierarchy. The view instance is sent a message specific to the event type and given the opportunity to respond. Whether the view is a button, a slider, a text view or a table view, all action begins with a simple message generated as the result of a user action.

NSView is an abstract superclass. Its default implementation for all event messages is to do nothing but pass the event on to its superview. It has no specialized behavior. NSView subclasses override these inherited methods, perhaps only a useful subset, and deal with the event in a specialized way. Buttons use mouse events to trigger target/action responses. Text fields process keyboard events for data input and display. You can create a custom view that is capable of doing the same.



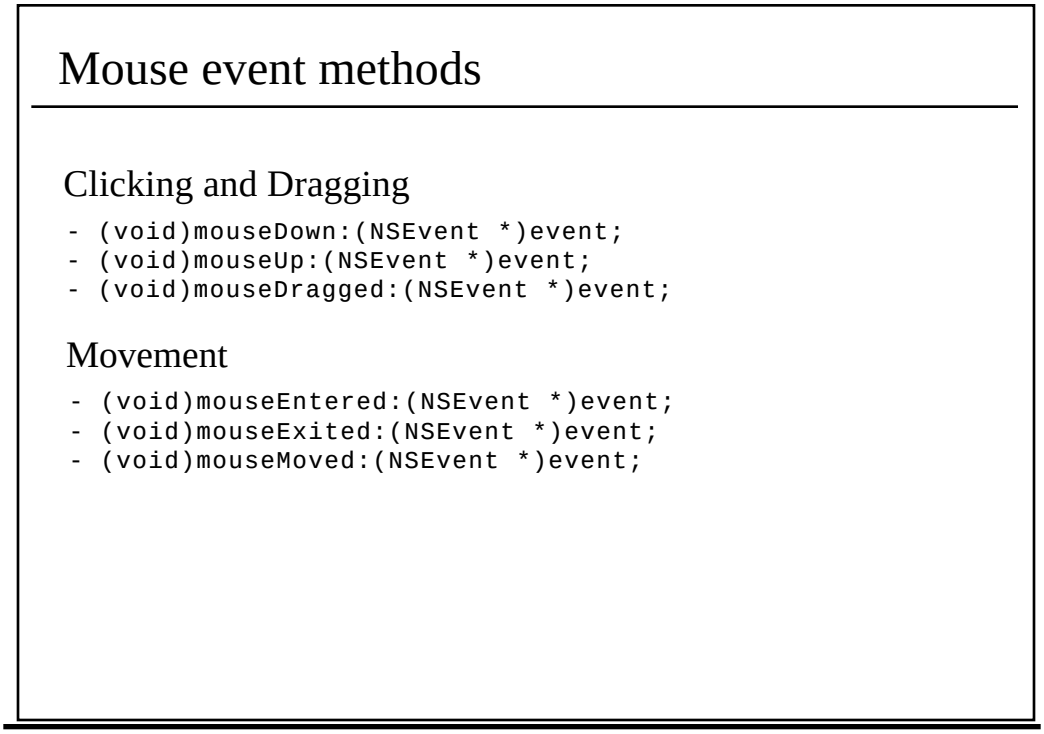
NSEvent—providing event details

Each of the event messages deliver a single argument, an instance of NSEvent. NSEvents are immutable tokens that provide detailed status for a given event, allowing the NSView to find out exactly what happened. Each event has a **type** so that it can be identified independently of the original message that delivered it. The event is timestamped so that a view can track the temporal reality of a sequence of events. Although all the common attributes shown above reside in every event, some make sense only in the context of a mouse or keyboard event.

A useful attribute for mouse events is the location of the mouse in the window's coordinate system. Views such as sliders or custom drawing views make use the mouse's exact location to provide visual feedback and tracking. The view must translate the point to its own coordinate system before using it. Since many views may look at the event before an arbitrary view claims it, it is more efficient to leave the point in the window's terms.

The **clickCount** allows a view to distinguish between a single and double click, useful for implementing target/action with an alternate **doubleAction** selector.

Keyboard events deliver the characters typed as well as the specific key code. Both mouse and keyboard events are likely to use the **modifierFlags** to incorporate special handling for such keys as shift, alt, control, and function keys.



Mouse event methods

Each event generates an NSEvent instance which becomes an argument to a specific message. Your custom view class must override one or more of these in order to specialize its behavior by handling the event. Mouse events fall into two categories:

» Clicking and dragging. Your view gets these events automatically. If you need to distinguish right from left mouse buttons, you may also override **rightMouseDown:** etc.

mouseDown:—right and left buttons distinguished via NSEvent's **type**

mouseUp:—usually mouse clicks generate actions when the mouse is released, not when it is pressed down. Target/action processing happens here.

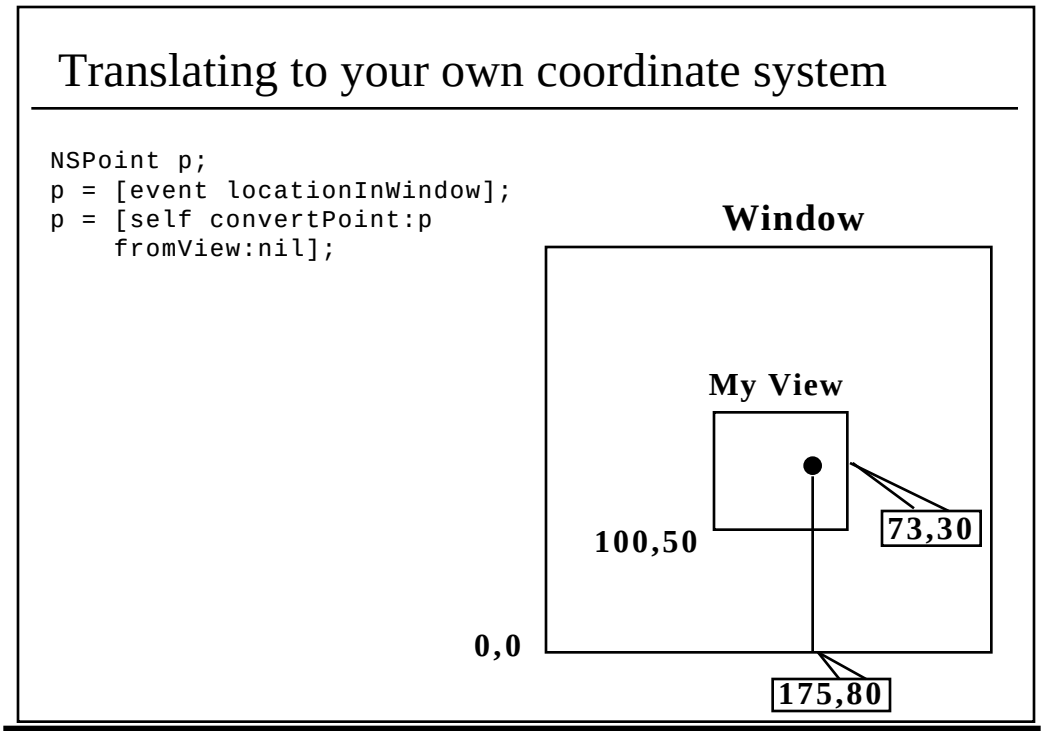
mouseDragged:—sent each time the position of the mouse changes after the initial **mouseDown:** while the mouse button is held down.

» Movement. These events happen much more frequently, and are therefore expensive. Your view must arrange to get these, either by sending its window the **setAcceptsMouseMovedEvents:** or by establishing a tracking rectangle.

mouseEntered:—mouse has entered your view or rectangle

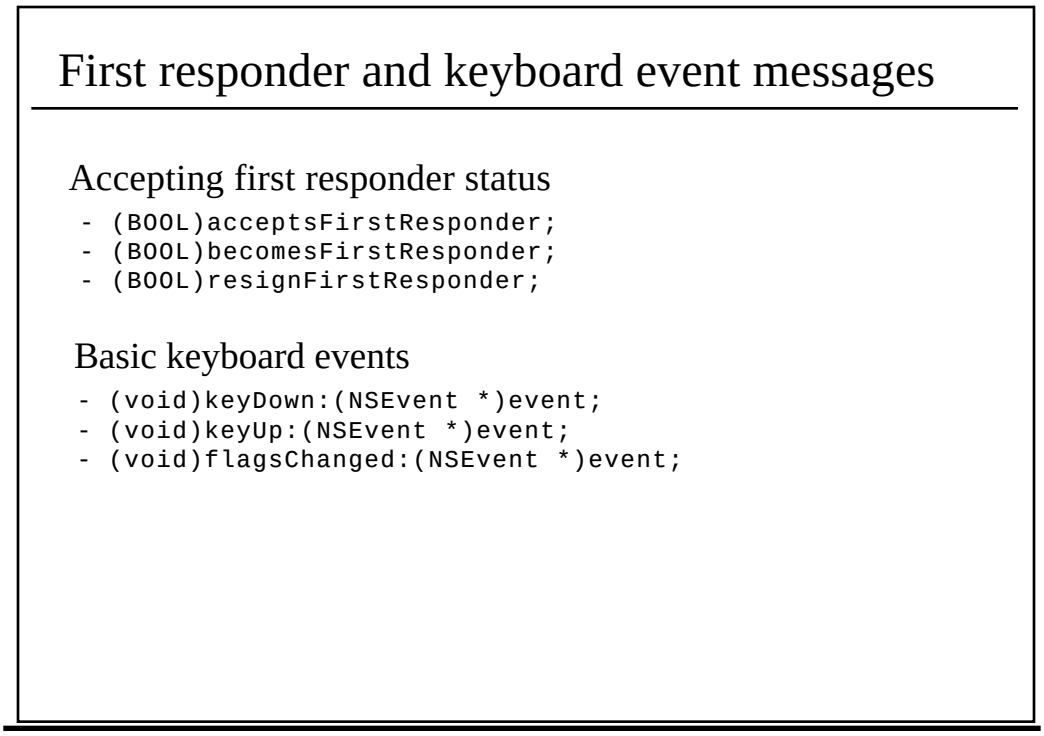
mouseExited:—mouse have left your view or rectangle

mouseMoved:—the mouse moved



Translating to your own coordinate system

When dealing with mouse events, a view may want to know the exact point of the event in its own coordinate system. The NSEvent instance provides the point in the window's coordinate system with **locationInWindow**. NSView implements a method that converts a point to your view's coordinate system from any other view instance—**convertPoint:fromView:**. If the alternate view is nil, the method converts from the window's coordinate system.



First responder and keyboard event messages

In order for an NSView instance to receive keyboard event messages, it must be the first responder. As such, it must identify itself as capable of doing so and possibly participate in a conditional negotiation if it has reason:

» **acceptsFirstResponder**—override to return YES if your view wants keyboard events. By default, this returns NO.

» **becomeFirstResponder**—notifies your view so it can prepare for keyboard events and possibly update its appearance.

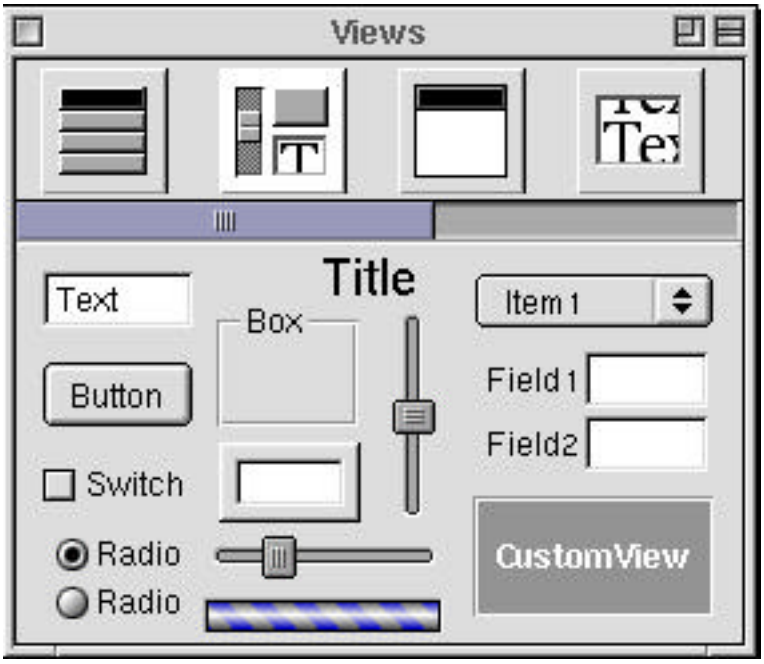
» **resignFirstResponder**—notifies your view that it will now lose first responder status. Your view can refuse e.g. a text field with invalid text.

Once your view is the first responder, it will get these messages:

» **keyDown:**—key press. Use NSEvent **characters**, **keyCode** and **modifierFlags** to identify exactly which keys are involved.

» **keyUp:**—the key has been released.

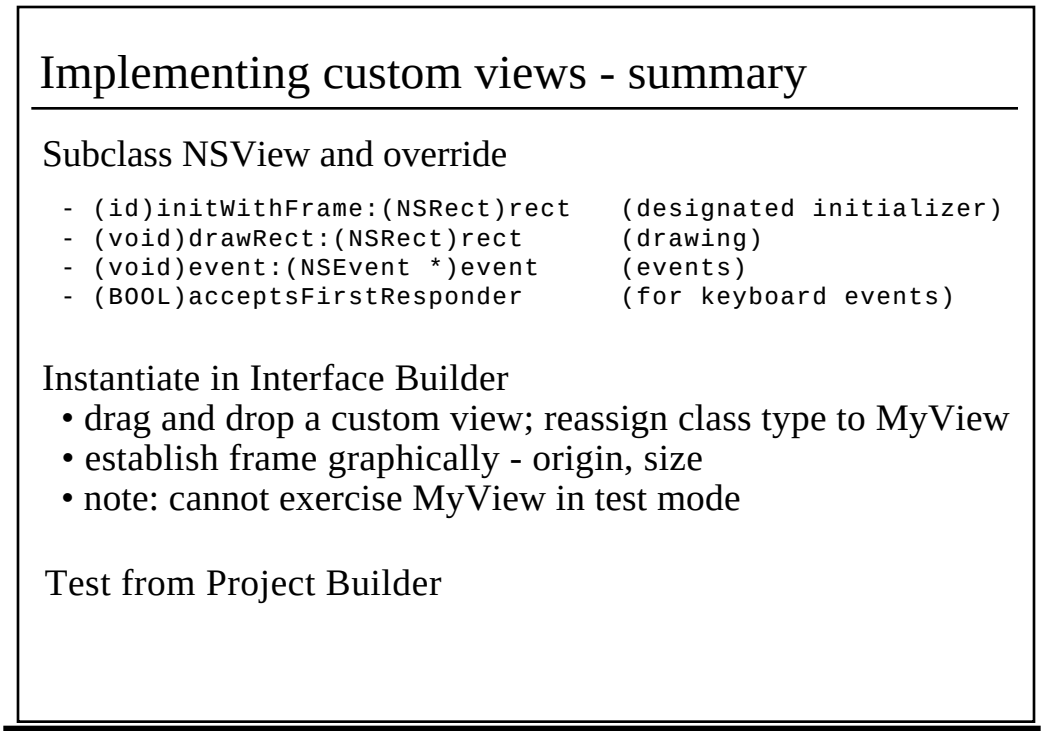
» **flagsChanged:**—check **modifierFlags**, they have changed.



Instantiating custom views in Interface Builder

Custom subclasses of NSView may be instantiated graphically using Interface Builder for ease of placement, sizing, and automatic insertion into the view hierarchy. To do this, drag and drop an instance of “Custom View” found on the Views palette. Make sure you have read your custom class’s interface file into Interface Builder. Using the “Attributes” view of the inspector panel, set the custom view’s type to your custom class. Note that, like any of your custom classes such as controllers and delegates, your custom view code will not be active until you have built and launched from Project Builder. You cannot test your view in Interface Builder test mode. Eventually, you can provide the view on a palette in which case it will display and function properly in test mode.

If your view is a subclass of a concrete Application Kit class such as NSTextField or NSButton, you can drag and drop an instance of these and using the “Custom Class” view on the inspector panel, set its type to your custom class which you have previously read into Interface Builder.



Implementing custom views—summary

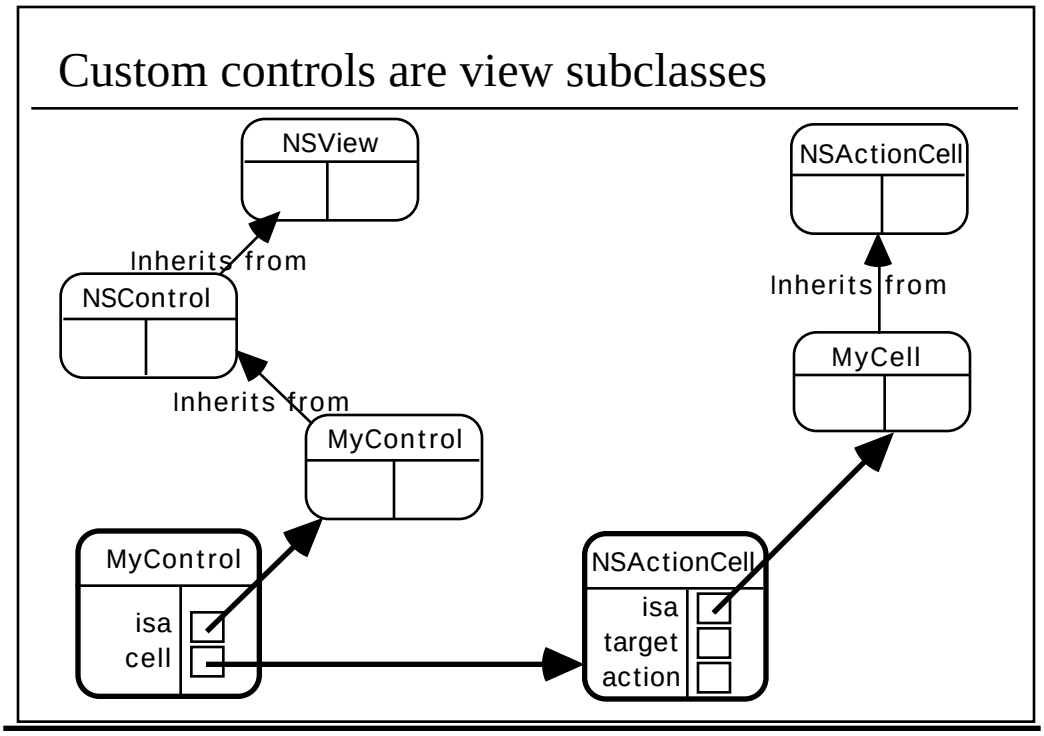
The designated initializer for NSView is **initWithFrame:**. Do you have to override it? Only if your custom subclass needs its own initialization. With non-trivial view subclasses, this is almost always the case.

All concrete views override **drawRect:** since the inherited version does nothing. Without custom drawing code, your view will be invisible.

Which event methods should your view override? Only those it is specifically interested in. The default implementations inherited from NSView pass the message on to the next responder.

To receive keyboard events, your view must accept first responder status. Implement the additional responder methods if your view needs preparation or cleanup as its responder status changes or if it accepts or relinquishes first responder status conditionally.

Use the custom view object on the Interface Builder Views palette to instantiate your custom view. Reassign the class type to your class. You must build and launch the application from Project Builder to test your view's code.



Custom controls are view subclasses

NSTextField and NSButtons are NSViews but more specifically they are instances of NSControl, an abstract yet specialized form of NSView. Remember that NSControl defines the attributes and methods that add target/action behavior to NSView. If you are designing a custom control, you will probably want to subclass NSControl, not NSView. This is required if you wish to make target/action connects graphically with Interface Builder.

In addition, remember that NSControls typically use NSCells. To derive the full flexibility of your custom control—e.g. the ability to create an NSMatrix of your controls or use them in a more elaborate view such as an NSTableView—you will also want to create a custom NSActionCell. It is even possible that you need only design a custom NSControl and reuse a standard cell. Or, it may work the other way around. Lastly, it is possible that you can directly subclass one of the Application Kit's concrete classes, either a control or a cell, to accomplish your needs.

Important ideas from this section

» A view is a visible area of responsibility on a window within an application. It is responsible for two things:

Drawing itself

Handling events that fall within its domain

» A view has basic geometric attributes that control its location, its size and are used to implement both drawing and event handling:

Frame

Coordinate system

» View's draw themselves in **drawRect:** and can use a variety of approaches for accessing the underlying Display PostScript graphics environment:

PostScript C Function API

Convenience Functions

Custom PostScript wrappers

NSImage for sampled images

NSColor, NSString, NSAttributedString and NSFont objects

» View's customize event handling by overriding a useful subset of mouse and/or keyboard methods, gaining access to specifics of each event through the NSEvent object.

Classes featured in this section

» NSView

» NSResponder

» NSEvent

» NSImage

» NSColor

» NSString

» NSAttributedString

» NSFont

REVIEW CUSTOM VIEWS

1. What two main functions does an NSView subclass perform?

2. Name three different features that are useful for implementing custom drawing.

3. What default functionality does your NSView subclass inherit if it does not override a mouse event message?

4. Describe the main steps for instantiating a custom view in your application's interface.

5. When testing your application, you find that your custom view does not draw itself—it is invisible. List some possible problems.