THE DISPLAY POSTSCRIPT® SYSTEM: NeXT OVERVIEW

Technical Note #5050

March 6, 1990
PostScript® Developer Support Group

Adobe Systems Incorporated
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400

PN LPS5050

THE DISPLAY POSTSCRIPT® SYSTEM: NeXT OVERVIEW

Technical Note #5050

March 6, 1990
PostScript® Developer Support Group
(415) 961-4111

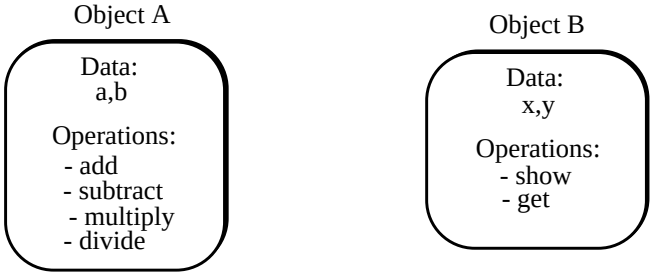
1. INTRODUCTION

This technical note is the first in a series of documents describing programming issues with the Display POSTSCRIPT® system. The examples used in the notes have been developed on the NeXT™ computer and, as a result, are heavily dependent on the NeXT programming environment. This document will provide an overview of that environment.

A number of other, more comprehensive sources of information on these topics is available in the NeXT and Objective-C® documentation. This document is intended to provide a simple basis for the examples and programs that follow in subsequent technical notes.

2. OBJECT-ORIENTED PROGRAMMING

The easiest way to develop applications on the NeXT computer is to take advantage of the object-oriented nature of the development environment. The concept behind object-oriented programming is the organization of an application into distinct objects, an object being a unit that contains both data and operators on the data. Data within an object can only be accessed, in most cases, through the use of an operator within the object. This organization allows for data to be hidden from areas that do not need to access it and tends to simplify the management of data and operations on the data.



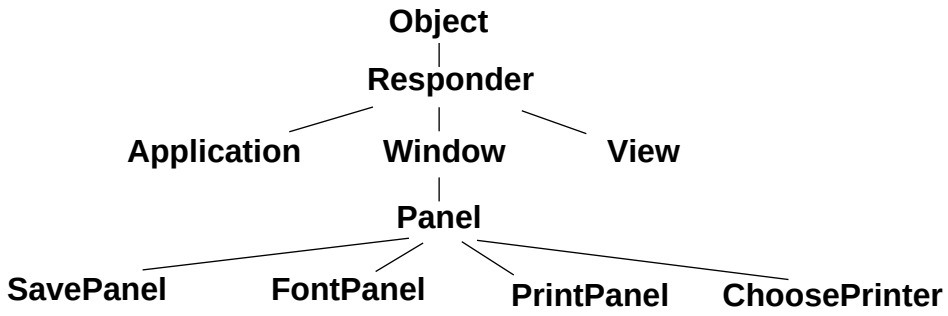
An object is essentially a combination of a structure containing data and of functions that operate on that data. The difference from conventional programming is that the structure and the operations are wholly owned by the object and another object cannot access the data. A message can be sent from one object to another, however, instructing the receiver to perform an operation in the receiver's repertoire. Data necessary for the operation may be passed along with the message.

A piece of data is called an *instance variable* and an operation is called a *method*. Objects with the same instance variable descriptions and methods are members of the same *class*. Different units within a class are called *instances* of the class. Each instance may have different values for the same data types but they share and use the same methods on the data. For example, if *phone* is a class, then two instances of the class would be *John's* phone and *Jane's* phone. In the diagram above, two classes are shown, Object A is one instance of one class and Object B is one instance of another class.

One of the key features to an object-oriented approach is the idea of inheritance. Objects are grouped in a hierarchy with descendants retaining the data types and methods of the ancestor. As a result, a method defined several steps up in the hierarchy chain can be used to operate on a similar and yet distinct object. This approach saves code space and allows for specific objects to be made out of a chain of more general and more abstract objects. In the NeXT environment, a class may have only one *superclass* or, in other words, one immediate parent, but the number in the chain is unlimited.

The methods of any ancestor can be added to or modified by a descendant and so it is easy to use an object as a model and add or change a small number of characteristics. Creating a descendant of an object is called *subclassing* and modifying a method is termed *overriding* a method. An example of subclassing in order to modify a class might be creating a subclass of a **Button** in the NeXT interface in order to turn it into a **LightSwitch**. The general on/off nature of the button would be retained but the visual display and interaction would be overridden and modified.

The following diagram shows the hierarchy within an abbreviated set of NeXT Application Kit objects. **Object** is the topmost class in the hierarchy. **Responder** is simply a class that handles events. **Application**, **Window** and **View** objects all receive events and so all are subclasses of the **Responder** class.



The tree is arranged with the more general and abstract classes nearer the root. Each subclass adds more specific characteristics to the tree. The advantage of going from general to specific in a number of steps is that classes can be added to the tree with very little rework or reordering. For example, a **ChooseModem** could be added as a branch under **Panel** and next to **ChoosePrinter**. The only work that would be needed would be to include the elements of a **ChooseModem** panel that make it unique in regards to a **ChoosePrinter** panel or any other type of panel. Another example would be the addition of an **EPSView u**nder **View**. (**View** is an object specifically designed for drawing on the screen with PostScript language code. An **EPSView** would be an object designed specifically to image

EPS files.) All the characteristics of a **View** would still hold true for an **EPSView** and so none of the objects above would be affected. The only addition would be the subclass **EPSView** and its special elements.

This diagram is a very brief introduction to a hierarchy of objects and touches on only a small portion of the Application Kit objects and their capabilities. The NeXT documentation and later sections provide much more detail in this area. One important item to note is that objects can contain or hold references to other objects without being related to the contained objects. An example from the Application Kit is the case of a **Window** and its **Views**. A **Window** and a **View** are not directly related. They share the same parent but are not directly related. Nonetheless, a **Window** can hold references to many **Views** and can control their display. In fact, a **View** cannot exist on its own; it must be assigned to a **Window**.

3. OBJECTIVE-C

Objective-C is the object-oriented programming language used on the NeXT machine. It is an extension of the C programming language and does not change or affect any standard C language capabilities. Objective-C syntax can be mixed freely with standard C constructs.

3.1 Defining a Class

Developing with Objective-C means first creating classes. Each class is defined using two types of files, an *interface* file (**.h**) and an *implementation* file (**.m**). The interface file contains the syntactical definition of the instance variables and the methods while the implementation file contains the code that implements these methods. Although multiple classes can be declared in the same interface and implementation files, it is customary to declare only one class per set.

The **WorldView** class in the example below is the only class that had to be created for the **HelloWorld** application. The rest of the functionality required for this application is provided by the NeXT Application Kit.

The interface file, **WorldView.h**, defines the new class as a subclass of **View**. **View** is a class provided by NeXT in the Application Kit. Since **WorldView** is a subclass of **View**, it inherits all the instance variables and methods that are defined for **View** and for **View**'s ancestors. Only one additional instance variable is needed, **DrawHello**, and it is defined as a boolean type. (BOOL is a data type defined by the Application Kit that can be assigned the values YES and NO.)

The methods are listed next **– drawHello:**, **clearHello:**, and **drawSelf::**. (The colons signify that arguments will be passed to the methods; each colon signifies one argument. The type of the argument is also included, except when the type is the default type which is an object identifier of type **id**.)

The **@interface** and the **@end** are Objective-C directives that enclose the definition of the class. Class methods are implemented by class objects and are preceded by a plus sign. Instance methods are implemented by instance objects and are preceded by a minus sign. In the case below, the methods are only referenced by instances of the class and so they are marked with a minus sign.

```
/* WorldView.h
 *
 *The instance variable, DrawHello, is toggled between YES and NO and
 * indicates whether the "Hello World" message should be displayed
 * in the view.
 */

#import <appkit/View.h>

/* WorldView is a subclass of View. It inherits */
/* all View's instance variables and methods.  */
@interface WorldView:View
{
  BOOL  DrawHello;
}

- drawHello:sender;
- clearHello:sender;
- drawSelf:(NXRect *)r :(int) count;

@end
```

The implementation file, **WorldView.m**, contains the list of header files from which definitions are used within the methods and then the methods themselves. Note that the syntax of the methods matches the syntax in the interface file. A warning would be generated at compile time were this not the case. Don't be concerned with any of the constructs in the methods. There are still several items that have been left behind the curtain, a few of which will be brought out in a little while.

```
/*WorldView.m
 *
 * WorldView is a subclass of View. It simply displays and clears a
 * message in the view. The [self display]; message calls a method
 * that is defined in the superclass, View. The display method calls
 * the drawSelf:: method after performing some view overhead. The
 * method, "drawSelf::", should not be called directly in order to
 * conform to Application Kit programming rules.
 */

#import "WorldView.h"
#import <dpsclient/wraps.h>

@implementation WorldView

- drawHello:sender
{
  DrawHello = YES;
  [self display];
  return self;
}

- clearHello:sender
{
  DrawHello = NO;
  [self display];
  return self;
}

/* The first two lines clear the view. The remainder display the
 * message in the view. */
- drawSelf:(NXRect *) r: (int) count
{
  PSsetgray(NX_LTGRAY);
  PSrectfill(bounds.origin.x, bounds.origin.y,
      bounds.size.width, bounds.size.height);

  if (DrawHello)
   {
    PSsetgray (NX_BLACK);
    PSmoveto (100.0, 100.0);
    PSselectfont ("Times-Roman", 40.0);
    PSshow ("Hello World");
  }

  return self;
}
@end
```

3.2 Messages

Instead of using only function calls as in conventional C programming, *messages* are used in Objective-C. (Function calls are still used and are quite handy in Objective-C programming but their explanation will be left for the examples.) A message is essentially a call to an object telling it to perform a certain method. (Arguments that the object may need can also be passed in the message.) The message syntax is shown below.

```
[<object id> <method>];--No arguments

[<object id> <method>:<argument>]; --One argument

[<object id> <method>:<argument> <methodcont>:<argument> ...];
--n arguments
```

Methods can return values. In most cases, the return value is the default type, **id**. The square brackets, [ ], simply identify the expression as an Objective-C message. The semi-colon is necessary after the closing bracket because, in this case, the message is a C program statement and all program statements in C must be terminated by a semicolon. All messages may not be program statements. Consider a message in an **if** statement; in this case, a semi-colon would not follow the message. For this example, the method should return a BOOL value.

```
if ([<object id> <method>])
  { <statement> };
```

object id

Each message is sent to a specific object. An *object id* is used to identify objects. The object id should be the first element in the message. An object id is a special C type that is assigned to the object at run-time when it is created by the system. If the id of an object is known by another object, the other object has the ability to send the first object a message. A number of ways exist for an object to learn the id of another object.

method

The *method* is the name defined in the interface file. The name of the method is case sensitive and includes the argument names. A run-time error will result if the method name invoked is different from that defined (provided no other method matches the name invoked). A method name is also referred to as a *selector*.

argument

The colon signifies that an argument is to be passed to the object. The same type of agreement between sender and receiver in regards to number and type of arguments has to occur in a message call as it does in a function call or else similar errors will result. The method name can be broken and separated by colon-argument pairs. The line below illustrates this condition. The name of the method is *replaceSubview:with:*. The argument **viewId1** is inserted after the first colon, a space separates **viewId1** and *with* and then the argument **viewId2** is inserted after the second colon.

```
[objectId replaceSubview:viewId1 with:viewId2];
```

The keys to messaging are 1) knowing the id of the object to send the message to 2) knowing the name of the correct method or selector and 3) knowing what arguments are passed to that method. A mix-up in ids could send the message to an object of another class. In this case, a run-time error would occur if the selector could not be found. A spelling error in the selector would produce a similar error.

3.3 Return Arguments

A message can return a value. The most common return type is the default type which is an object id. All the methods in **WorldView** return an object id indicated by the return of **self** in each method in **WorldView.m**. **Self** is a special object identifier that means the object's own id. Since the default is returned, no type definition has to be included in the message sequence defined in **WorldView.h**. In many cases, the return argument is important and is used by the sender as in the case of nesting messages shown below. In other cases, the return argument is ignored.

If we want a method to return a type other than the default, we would need to include a type definition for the method in the interface and implementation files. In the **if** statement above, the method specified would return a boolean value (it better or we may have some problems). The next line shows a method definition in an interface file that would return an integer.

```
- (int) getIntValue; /* This method returns an integer value. */
```

3.4 Nesting

Messages can be nested together. The inner messages are resolved first. The following example shows a nesting arrangement. In this example, an object, identified by the variable **matrixID**, is sent a message to return the selected cell. The cell id that is returned is then sent a message to return the cell's state. The state is then assigned to the boolean value, **BoolValue**. (**Matrix** and **Cell** are Application Kit objects. What the actual messages mean may become more apparent when they are seen in an example.)

```
BOOL  BoolValue;
BoolValue = [[matrixID selectedCell] state];
```

3.5 Messaging ‘self’ and ‘super’

Two special object identifiers can be used when messaging. The first is **self** and refers to the object's own id. In **WorldView.m**, the line **[self display];** appears twice. The method, **display**, is defined in the **View** class, of which **WorldView** is a subclass. Sending a message to **self** is a common occurrence. It is simply a way to invoke a method within the same object.

The second special identifier is **super**. It refers to the object's parent or superclass. In the case of the **WorldView** class, the superclass would be **View**. The **super** identifier is useful for invoking a method in the superclass when it has been overridden in the subclass. Messaging the method with **self** would invoke the method in the subclass. Messaging with **super** would invoke the method in the superclass. This identifier is handy when the subclass wants to add a characteristic to a method but still perform the functions of the method in the superclass. The example below illustrates this point.

```
- free
{
  [self freeMemory];
  [super free];

  return self;
}
```

A subclass has been created and the free method has been overridden to include a message to **freeMemory**. Since we also want to perform the free operations that appear in the superclasses, we need to message the free method of the superclass with **[super free]**. If **[super free]** were not used, none of the free operations of the superclasses would be performed.

4. NeXT APPLICATION KIT CLASSES

The NeXT Application Kit provides a set of classes that may be used within an application. The examples in the technical notes use many of these classes. The NeXT documentation contains a comprehensive listing of the classes including their instance variables and their methods. This section will provide a simple overview of the classes more commonly used in the examples. Many of the classes listed are not separate and distinct. Rather, a large number are subclasses of other classes.

4.1 Object

The top class in the Application Kit hierarchy is titled **Object**. It is the root of all Objective-C inheritance hierarchies, so all other classes inherit from it. The **Object** class provides its subclasses with a framework for creating, freeing, copying and archiving object — to name just a few of its operations.

4.2 Responder

Responder is an abstract class that forms the basis for command and event processing in the Application Kit. Many of the major classes in the Application Kit inherit from the **Responder** class. Responders are linked in one-way chain. Events are passed through the

chain. Each responder in the chain will look at the event in turn. If the class cannot handle the event, it will be passed on to the next responder in the chain. **Application**, **View**, **Window** and **Control** are subclasses of **Responder**.

4.3 Application

The **Application** class contains some of the methods necessary for a NextStep application. It provides the framework for program execution. Every program has one and only one instance of the **Application** class. Creating an instance of this class connects the program to the Window Server and initializes the PostScript environment. The **Application** object is usually the first object to receive events from the Window Server. It will often pass the events to windows which pass them to views. The events then begin to proceed through the responder chain.

4.4 View

The **View** class provides much of the structure for drawing on the screen. Any graphical object that draws on the screen inherits from **View**. The **HelloWorld** example subclasses **View** with **WorldView** in order to draw "Hello World" on the screen. The **View** class can respond to events. As a result, mouse tracking loops are often placed in **View** subclasses.

4.5 Window

The **Window** class manages a window within an application including the interaction with the Window Server. Methods are available to display, hide, resize and close windows along with a host of other types of operations. A **Window** object can hold references to a number of **Views** and can control their display. Events passed to the window can also be passed to one of these **Views**.

4.6 Control

The **Control** class is the superclass for a number of classes needed for the graphical interface of an application. **Sliders**, **Fields** and **Buttons**, classes also defined in the Application Kit, inherit from control. A control is an abstract entity that is concerned with handling an event. A different and unrelated class, **Cell**, is used to provide the visual aspect to the control.

4.7 Cell

The **Cell** class provides the mechanism to display buttons, icons and text without the overhead of a full **View** subclass. Cells are used by the **Control** class and its subclasses to handle the visual aspects of the control.

***Note**: **Cells** are not a subclass of **Control**. Rather a **Control** object uses a **Cell** or a number of **Cells** to control the visual interaction of displaying, editing, formatting, highlighting and tracking.*

5. INTERFACE BUILDER

Interface Builder is an application provided with the NeXT computer to graphically construct the user interface of an application. Windows can be created and buttons and text fields inserted to quickly develop a large portion of the look of the application.

At some point, NextStep applications will probably have to create separate subclasses of existing classes and compose Objective-C files. These classes and files will contain the core of an application, the part that is distinct from the interface.

The Interface Builder provides a quick way to create many of the common features of an application. In **HelloWorld**, only one class was created, a subclass of **View**. The other classes were Application Kit classes brought in through Interface Builder.

Interface Builder provides the ability to create a variety of objects as well as the ability to *connect* one object to another (have one object message another object in response to a certain event). The objects that can be created and connected are a subset of the classes that are contained in the Application Kit. For example, a slider can be connected to a text field so that the slider will message the text field when the slider has been moved. As a result, the text field can display the value of the slider. Technical Note #5051 will explore the Interface Builder in more detail.

6. SUMMARY

The NeXT programming environment is built around an object-oriented approach. Objective-C, an extension to the C language, is used as the base programming language. The Application Kit is a set of predefined classes that may can be incorporated within an application. These classes provide a solid basis for the development of most applications. Interface Builder is an application provided with the NeXT computer to graphically construct a user interface for an application. Using Interface Builder, windows can be created and buttons and text fields inserted to quickly develop a large portion of the interface of an application.