4

In this chapter:
- *Windows*
- *Views*
- *Messaging*

Windows, Views, and Messages

A window serves as a program's means of communicating with the user. In order to provide information to a user, a window needs to be able to draw either text or graphics. And in order to receive information from a user, a window needs to be aware of user actions such as mouse button clicks or key presses. Views make both these modes of communication possible. All drawing takes place in views. And views are recipients of messages that are transmitted from the Application Server to the program in response to user actions. All three of these topics—windows, views, and messages—can be discussed individually, and this chapter does just that. To be of real use, though, the interaction of these topics must be described; this chapter of course does that as well.

Windows

Your program's windows will be objects of a class, or classes, that your project derives from the `BWindow` class. The `BWindow` class is one of many classes in the Interface Kit—the largest of the Be kits. Most other Interface Kit class objects draw to a window, so they expect a `BWindow` object to exist—they work in conjunction with the window object.

Because it is a type of `BLooper`, a `BWindow` object runs in its own thread and runs its own message loop. This loop is used to receive and respond to messages from the Application Server. In this chapter's "Messaging" section, you'll see how a window often delegates the handling of a message to one of the views present in the window. The ever-present interaction of windows, views, and messages accounts for the combining of these three topics in this chapter.

Window Characteristics

A window's characteristics—its size, screen location, and peripheral elements (close button, zoom button, and so forth)—are all established in the constructor of the `BWindow`-derived class of the window.

BWindow constructor

A typical `BWindow`-derived class constructor is often empty:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    :BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}
```

The purpose of the constructor is to pass window size and window screen location on to the `BWindow` constructor. In this next snippet, this is done by invoking the `MyHelloWindow` constructor, using the `BRect` parameter `frame` as the first argument in the `BWindow` constructor:

```
MyHelloWindow   *aWindow;
BRect            aRect(20, 30, 250, 100);

aWindow = new MyHelloWindow(aRect);
```

It is the `BWindow` constructor that does the work of creating a new window. The four `BWindow` constructor parameters allow you to specify the window's:

- Size and screen placement
- Title
- Type or look
- Behavioral and peripheral elements

The `BWindow` constructor prototype, shown here, has four required parameters and an optional fifth. Each of the five parameters is discussed following this prototype:

```
BWindow(BRect       frame,
        const char  *title,
        window_type type,
        ulong       flags,
        ulong       workspaces = B_CURRENT_WORKSPACE)
```

Window size and location (frame argument)

The first `BWindow` constructor parameter, `frame`, is a rectangle that defines both the size and screen location of the window. The rectangle's coordinates are relative to the screen's coordinates. The top left corner of the screen is point (0, 0), and coordinate values increase when referring to a location downward or

rightward. For instance, the lower right corner of a 640×480 screen has a screen coordinate point of (639, 479). Because the initialization of a `BRect` variable is specified in the order left, top, right, bottom; the following declaration results in a variable that can be used to create a window that has a top left corner fifty pixels from the top of the user's screen and seventy pixels in from the left of that screen:

```
BRect  frame(50, 70, 350, 270);
```

The width of the window based on `frame` is determined simply from the delta of the first and third `BRect` initialization parameters, while the height is the difference between the second and fourth. The above declaration results in a rectangle that could be used to generate a window 301 pixels wide by 201 pixels high. (The "extra" pixel in each direction is the result of zero-based coordinate systems.)

The frame coordinates specify the content area of a window—the window's title tab is not considered. For titled windows, you'll want to use a top coordinate of at least 20 so that none of the window's title tab ends up off the top of the user's screen.

If your program creates a window whose size depends on the dimensions of the user's screen, make use of the `BScreen` class. A `BScreen` object holds information about one screen, and the `BScreen` member functions provide a means for your program to obtain information about this monitor. Invoking `Frame()`, for instance, returns a `BRect` that holds the coordinates of the user's screen. This next snippet shows how this rectangle is used to determine the width of a monitor:

```
BScreen  mainScreen(B_MAIN_SCREEN_ID);
BRect    screenRect;
int32    screenWidth;

screenRect = mainScreen->Frame();
screenWidth = screenRect.right - screenRect.left;
```

As of this writing, the BeOS supports only a single monitor, but the above snippet anticipates that this will change. The Be-defined constant `B_MAIN_SCREEN_ID` is used to create an object that represents the user's main monitor (the monitor that displays the Deskbar). Additionally, the width of the screen can be determined by subtracting the left coordinate from the right, and the height by subtracting the top from the bottom. On the main monitor, the `left` and `top` fields of the `BRect` returned by `Frame()` are 0, so the `right` and `bottom` fields provide the width and height of this screen. When an additional monitor is added, though, the `left` and `top` fields will be non-zero; they'll pick up where the main screen "ends."

Window title

The second `BWindow` constructor argument, `title`, establishes the title that is to appear in the window's tab. If the window won't display a tab, this parameter

value is unimportant—you can pass `NULL` or an empty string (`""`) here (though you may want to include a name in case your program may eventually access the window through scripting.

Window type

The third `BWindow` constructor parameter, `type`, defines the style of window to be created. Here you use one of five Be-defined constants:

`B_DOCUMENT_WINDOW`
: Is the most common type, and creates a nonmodal window that has a title tab. Additionally, the window has right and bottom borders that are thinner than the border on its other two sides. This narrower border is designed to integrate well with the scrollbars that may be present in such a window.

`B_TITLED_WINDOW`
: Results in a nonmodal window that has a title tab.

`B_MODAL_WINDOW`
: Creates a modal window, a window that prevents other application activity until it is dismissed. Such a window is also referred to as a dialog box. A window of this type has no title tab.

`B_BORDERED_WINDOW`
: Creates a nonmodal window that has no title tab.

`B_FLOATING_WINDOW`
: Creates a window that floats above (won't be obscured by) other application windows.

There's another version of the `BWindow` constructor that has two parameters (`look` and `feel`) in place of the one `type` parameter discussed above. The separate look and feel parameters provide a means of more concisely stating just how a window is to look and behave. The single type parameter can be thought of as a shorthand notation that encapsulates both these descriptions. Refer to the `BWindow` class section of the Interface Kit chapter of the Be Book for more details (and a list of Be-defined `look` and `feel` constants).

Window behavior and elements

The fourth `BWindow` constructor argument, `flags`, determines a window's behavior (such as whether the window is movable) and the window's peripheral elements (such as the presence of a title tab or zoom button). There are a number of Be-defined constants that can be used singly or in any combination to achieve the desired window properties. To use more than a single constant, list each and

separate them with the OR (|) operator. The following example demonstrates how to create a window that has no zoom button or close button:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    :BWindow(frame, windowName, B_TITLED_WINDOW, B_NOT_ZOOMABLE | B_NOT_
CLOSABLE)
{
}
```

If you use 0 (zero) as the fourth parameter, it serves as a shortcut for specifying that a window include all the characteristics expected of a titled window. Default windows are movable, resizable, and have close and zoom buttons:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    :BWindow(frame, windowName, B_TITLED_WINDOW, 0)
{
}
```

The following briefly describes many of the several Be-defined constants available for use as the fourth parameter in the `BWindow` constructor:

`B_NOT_MOVABLE`
: Creates a window that cannot be moved—even if the window has a title tab. By default, a window with a title tab is movable.

`B_NOT_H_RESIZABLE`
: Generates a window that can't be resized horizontally. By default, a window can be resized both horizontally and vertically.

`B_NOT_V_RESIZABLE`
: Generates a window that can't be resized vertically. By default, a window can be resized both horizontally and vertically.

`B_NOT_RESIZABLE`
: Creates a window that cannot be resized horizontally or vertically.

`B_NOT_CLOSABLE`
: Results in a window that has no close button. By default, a window with a title tab has a close button.

`B_NOT_ZOOMABLE`
: Results in a window that has no zoom box. By default, a window with a title tab has a zoom box.

`B_NOT_MINIMIZABLE`
: Defines a window that cannot be minimized (collapsed). By default, a window can be minimized by double-clicking on the window's title bar.

`B_WILL_ACCEPT_FIRST_CLICK`

Results in a window that is aware of mouse button clicks in it—even when the window isn't frontmost. By default, a window is aware only of mouse button clicks that occur when the window is the frontmost, or active, window.

Workspace

The `BWindow` constructor has an optional fifth parameter, `workspaces`, that specifies which workspace or workspaces should contain the new window. Desktop information such as screen resolution and color depth (number of bits of color data per pixel) can be adjusted by the user. Different configurations can be saved to different workspaces. Workspaces can be thought of as virtual monitors to which the user can switch. Under different circumstances, a user may wish to display different types of desktops. By omitting this parameter, you tell the `BWindow` constructor to use the default Be-defined constant `B_CURRENT_WORKSPACE`. Doing so means the window will show up in whatever workspace is currently selected by the user. To create a window that appears in all of the user's workspaces, use the Be-defined constant `B_ALL_WORKSPACES` as the fifth parameter to the `BWindow` constructor.

You can find out more about workspaces from the user's perspective in the BeOS User's Guide, and from the programmer's perspective in the `BWindow` constructor section of the Interface Kit chapter of the Be Book.

Accessing Windows

Fortunately for you, the programmer, the Be operating system takes care of much of the work in keeping track of your application's windows and the user's actions that affect those windows. There will be times, however, when you'll need to directly manipulate one or all of your program's windows. For instance, you may want to access the frontmost window to draw to it, or access all open windows to implement a Close All menu item.

The Application Server keeps a list that holds references to an application's open windows. The list indices begin at 0, and continue integrally. The windows aren't entered in this list in any predefined order, so you can't rely on a particular index referencing a particular window. You can, however, use the `BApplication` member function `WindowAt()` to find any given window.

Accessing a window using WindowAt()

`WindowAt()` accepts a single argument, an integer that serves as a window list index. Calling `WindowAt()` returns the `BWindow` object this index references. A call to `WindowAt()` returns the first window in the list:

```
BWindow  *aWindow;

aWindow = be_app->WindowAt(0);
```

From Chapter 1, *BeOS Programming Overview*, you know that the Be-defined global variable `be_app` always points to the active application, so you can use it anywhere in your code to invoke a `BApplication` member function such as `WindowAt()`.

When `WindowAt()` is passed a value that is an out-of-bounds index, the routine returns `NULL`. You can use this fact to create a simple loop that accesses each open window:

```
BWindow  *theWindow;
int32    i = 0;

while (theWindow = be_app->WindowAt(i++)) {
   // do something, such as close theWindow
}
```

The preceding loop starts at window 0 in the window list and continues until the last window in the list is reached.

A good use for the `WindowAt()` loop is to determine the frontmost window. The `BWindow` member function `IsFront()` returns a `bool` (Boolean) value that indicates whether a window is frontmost. If you set up a loop to cycle through each open window and invoke `IsFront()` for each returned window, the frontmost window will eventually be encountered:

```
BWindow  *theWindow;
BWindow  *frontWindow = NULL;
int32    i = 0;

while (theWindow = be_app->WindowAt(i++)) {
   if (theWindow->IsFront())
      frontWindow = theWindow;
}
```

In the preceding snippet, note that `frontWindow` is initialized to `NULL`. If no windows are open when the loop runs, `frontWindow` will retain the value of `NULL`, properly indicating that no window is frontmost.

Frontmost window routine

With the exception of `main()`, all the functions you've encountered to this point have been part of the BeOS API—they've all been Be-defined member functions of Be-defined classes. Your nontrivial projects will also include application-defined member functions, either in classes you define from scratch or in classes you derive from a Be-defined class. Here I provide an example of this second category of application-defined routine. The `MyHelloApplication` class is derived from the Be-defined `BApplication` class. This version of `MyHelloApplication` adds a new application-defined routine to the class declaration:

```
class MyHelloApplication : public BApplication {

   public:
                  MyHelloApplication();
      BWindow *   GetFrontWindow();
};
```

The function implementation is familiar to you—it's based on the previous snippet that included a loop that repeatedly calls `AtWindow()`:

```
BWindow * MyHelloApplication::GetFrontWindow()
{
   BWindow  *theWindow;
   BWindow  *frontWindow = NULL;
   int32    i = 0;

   while (theWindow = be_app->WindowAt(i++)) {
      if (theWindow->IsFront())
         frontWindow = theWindow;
   }
   return frontWindow;
}
```

When execution of `GetFrontWindow()` ends, the routine returns the `BWindow` object that is the frontmost window. Before using the returned window, typecast it to the `BWindow`-derived class that matches its actual type, as in:

```
MyHelloWindow  *frontWindow;

frontWindow = (MyHelloWindow *)GetFrontWindow();
```

With access to the frontmost window attained, any `BWindow` member function can be invoked to perform some action on the window. Here I call the `BWindow` member function `MoveBy()` to make the frontmost window jump down and to the right 100 pixels in each direction:

```
frontWindow->MoveBy(100, 100);
```

Frontmost window example project

I've taken the preceding `GetFrontWindow()` routine and included it in a new version of MyHelloWorld. To test out the function, I open three MyHelloWorld windows, one directly on top of another. Then I call `GetFrontWindow()` and use the returned `BWindow` reference to move the frontmost window off the other two. The result appears in Figure 4-1.

```
MyHelloApplication::MyHelloApplication()
   : BApplication("application/x-vnd.dps-mywd")
{
   MyHelloWindow  *aWindow;
   BRect          aRect;
   MyHelloWindow  *frontWindow;

   aRect.Set(20, 30, 250, 100);
   aWindow = new MyHelloWindow(aRect);
   aWindow = new MyHelloWindow(aRect);
   aWindow = new MyHelloWindow(aRect);

   frontWindow = (MyHelloWindow *)GetFrontWindow();
   if (frontWindow)
      frontWindow->MoveBy(100, 100);
}
```

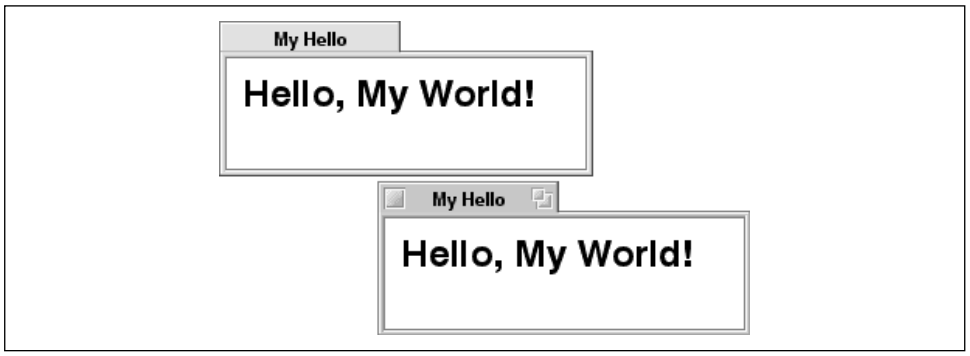

Figure 4-1. The result of running the FrontWindow program

Notice that before working with the returned window reference, I verify that it has a non-`NULL` value. If no windows are open when `GetFrontWindow()` is invoked, that routine returns `NULL`. In such a case, a call to a `BWindow` member function such as `MoveBy()` will fail.

The `MyHelloWindow` class doesn't define any of its own member functions—it relies on `BWindow`-inherited functions. So in this example, I could have declared `frontWindow` to be of type `BWindow` and omitted the typecasting of the returned `BWindow` reference. This code would still work:

```
...
BWindow  *frontWindow;
```

```
...
frontWindow = GetFrontWindow();
   if (frontWindow)
      frontWindow->MoveBy(100, 100);
}
```

But instead of working with the returned reference as a `BWindow` object, I opted to typecast it to a `MyHelloWindow` object. That's a good habit to get into—the type of window being accessed is then evident to anyone looking at the source code listing. It also sets up the returned object so that it can invoke any `BWindow`-derived class member function. A `BWindow` object knows about only `BWindow` functions, so if I define a `SpinWindow()` member function in the `MyHelloWindow` class and then attempt to call it without typecasting the `GetFrontWindow()`-returned `BWindow` reference, the compiler will complain:

```
BWindow  *frontWindow;

frontWindow = GetFrontWindow();
if (frontWindow)
   frontWindow->SpinWindow();    // compilation error at this line
```

The corrected version of the above snippet looks like this:

```
MyHelloWindow  *frontWindow;

frontWindow = (MyHelloWindow *)GetFrontWindow();
if (frontWindow)
   frontWindow->SpinWindow();    // compiles just fine!
```

Windows and Data Members

Defining a `GetFrontWindow()` or some similar member function to locate a window is one way to access a window. If you have only one instance of any given window class in your program, though, you should consider using a technique that stores window references in data members in the application object.

Defining a window object data member in the application class

For each type of window in your application, you can add to the class definition a private data member of the window class type. Consider a program that displays two windows: an input window for entering a mathematical equation, and an output window that displays a graph of the entered equation. If such a program defines `BWindow`-derived classes named `EquationWindow` and `GraphWindow`, the `BApplication`-derived class could include two data members. As shown below, Be convention uses a lowercase *f* as the first character of a data member name:

```
class MathApp : public BApplication {

   public:
                     MathApp();
```

```
                        ...
    private:
        EquationWindow  *fEquationWindow;
        GraphWindow     *fGraphWindow;
};
```

For the MyHelloWorld project, the `MyHelloApplication` class is defined as:

```
class MyHelloApplication : public BApplication {

    public:
                        MyHelloApplication();

    private:
        MyHelloWindow  *fMyWindow;
};
```

Storing a window object in the data member

In past examples, I created an instance of a window by declaring a local window variable in the application constructor, then using that variable in a call to the window's class constructor:

```
MyHelloWindow  *aWindow;
...
aWindow = new MyHelloWindow(aRect);
```

With the new technique, there's no need to use a local variable. Instead, assign the object returned by the window constructor to the window data member. The new version of the `MyHelloApplication` class defines an `fMyWindow` data member, so the result would be:

```
fMyWindow = new MyHelloWindow(aRect);
```

Here's how the new version of the `MyHelloApplication` constructor looks:

```
MyHelloApplication::MyHelloApplication()
    : BApplication("application/x-vnd.dps-mywd")
{
    BRect  aRect;

    aRect.Set(20, 30, 250, 100);
    fMyWindow = new MyHelloWindow(aRect);
}
```

Once created, the new window can be accessed from any application member function. For instance, to jump the window across part of the screen requires only one statement:

```
fMyWindow->MoveBy(100, 100);
```

Window object data member example projects

This chapter's `MyHelloWorld` project consists of the new version of the `MyHelloApplication` class—the version that includes an `fMyWindow` data member. The executable built from this project is indistinguishable from that built from prior versions of the project; running the program results in the display of a single window that holds the string "Hello, My World!"

The WindowTester project picks up where MyHelloWorld leaves off. Like MyHelloWorld, it includes an `fMyWindow` data member in the `MyHelloApplication` class. The WindowTester version of the `MyHelloApplication` class also includes a new application-defined member function:

```
class MyHelloApplication : public BApplication {

   public:
                     MyHelloApplication();
      void           DoWindowStuff();

   private:
      MyHelloWindow  *fMyWindow;
};
```

After creating a window and assigning it to the `fMyWindow` data member, the `MyHelloApplication` constructor invokes `DoWindowStuff()`:

```
MyHelloApplication::MyHelloApplication()
   : BApplication("application/x-vnd.dps-mywd")
{
   BRect  aRect;

   aRect.Set(20, 30, 250, 100);
   fMyWindow = new MyHelloWindow(aRect);

   DoWindowStuff();
}
```

I've implemented `DoWindowStuff()` such that it glides the program's one window diagonally across the screen:

```
void  MyHelloApplication::DoWindowStuff()
{
   int16  i;

   for (i=0; i<200; i++) {
      fMyWindow->MoveBy(1, 1);
   }
}
```

Feel free to experiment by commenting out the code in `DoWindowStuff()` and replacing it with code that has `fMyWindow` invoke `BWindow` member functions other than `MoveBy()`. Refer to the `BWindow` section of the Interface Kit chapter of the Be Book for the details on such `BWindow` member functions as `Close()`, `Hide()`, `Show()`, `Minimize()`, `ResizeTo()`, and `SetTitle()`.

Views

A window always holds one or more views. While examples up to this point have all displayed windows that include only a single view, real-world Be applications make use of windows that often consist of a number of views. Because all drawing must take place in a view, everything you see within a window appears in a view. A scrollbar, button, picture, or text lies within a view. The topic of drawing in views is significant enough that it warrants its own chapter—Chapter 5, *Drawing*. In this chapter, the focus will be on how views are created and accessed. Additionally, you'll get an introduction to how a view responds to a message.

A view is capable of responding to a message sent from the Application Server to a `BWindow` object and then on to the view. This messaging system is the principle on which controls such as buttons work. The details of working with controls are saved for Chapter 6, *Controls and Messages*, but this chapter ends with a discussion of views and messages that will hold you over until you reach that chapter.

Accessing Views

You've seen that a window can be accessed by storing a reference to the window in the `BApplication`-derived class (as demonstrated with the `fMyWindow` data member) or via the BeOS API (through use of the `BApplication` member function `WindowAt()`). A similar situation exists for accessing a view.

Views and data members

Just as a reference to a window can be stored in an application class data member, a reference to a view can be stored in a window class data member. The MyHelloWorld project defines a single view class named `MyHelloView` that is used with the project's single window class, the `MyHelloWindow` class. Here I'll add a `MyHelloView` reference data member to the `MyHelloWindow` class:

```
class MyHelloWindow : public BWindow {

   public:
```

```
                        MyHelloWindow(BRect frame);
      virtual bool      QuitRequested();

   private:
      MyHelloView  *fMyView;
};
```

Using this new technique, a view can be added to a new window in the window's constructor, much as you've seen in past examples. The `MyHelloWindow` constructor creates a new view, and a call to the `BWindow` member function `AddChild()` makes the view a child of the window:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyHelloView(frame, "MyHelloView");
   AddChild(fMyView);

   Show();
}
```

The window's view can now be easily accessed and manipulated from any `MyHelloWindow` member function.

View data member example projects

This chapter's NewMyHelloWorld project includes the new versions of the `MyHelloWindow` class and the `MyHelloWindow` constructor—the versions developed above. Once again, performing a build on the project results in an executable that displays a single "Hello, My World!" window. This is as expected. Using a data member to keep track of the window's one view simply sets up the window for easy access to the view—it doesn't change how the window or view behaves.

The ViewDataMember project serves as an example of view access via a data member—the `fMyView` data member that was just added to the NewMyHelloWorld project. Here's how the ViewDataMember project defines the `MyHelloWindow` class:

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual bool      QuitRequested();
      void              SetHelloViewFont(BFont newFont, int32 newSize);

   private:
      MyHelloView  *fMyView;
};
```

The difference between this project and the previous version is that this project uses the newly added `SetHelloViewFont()` member function to set the type and size of the font used in a view. In particular, the project calls this routine to set the characteristics of the font used in the `MyHelloView` view that the `fMyView` data member references. Here's what the `SetHelloViewFont()` implementation looks like:

```
void MyHelloWindow::SetHelloViewFont(BFont newFont, int32 newSize)
{
   fMyView->SetFont(&newFont);
   fMyView->SetFontSize(newSize);
}
```

`SetFont()` and `SetFontSize()` are `BView` member functions with which you are familiar—they're both invoked from the `MyHelloView AttachedToWindow()` function, and were introduced in Chapter 2, *BeIDE Projects*.

To change a view's font, `SetHelloViewFont()` is invoked by a `MyHelloWindow` object. To demonstrate its use, I chose to include the call in the `MyHelloWindow` constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyHelloView(frame, "MyHelloView");
   AddChild(fMyView);

   BFont  theFont = be_plain_font;
   int32  theSize = 12;
   SetHelloViewFont(theFont, theSize);

   Show();
}
```

The call to `SetHelloViewFont()` results in the about-to-be shown window having text characteristics that include a font type of plain and a font size of 12. Figure 4-2 shows the results of creating a new window. While `SetHelloViewFont()` is a trivial routine, it does the job of demonstrating view access and the fact that characteristics of a view can be changed at any time during a program's execution.

Figure 4-2. The ViewDataMember window displays text in a 12-point plain font

A More Practical Use For SetHelloViewFont()

Attaching a view to a window by calling `AddChild()` automatically invokes the view's `AttachedToWindow()` routine to take care of any final view setup. Recall that the `MyHelloView` class overrides this `BView` member function and invokes `SetFont()` and `SetFontSize()` in the `AttachedToWindow()` implementation:

```
void MyHelloView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);
}
```

So it turns out that in the above version of the `MyHelloWindow` constructor, the view's font information is set twice, almost in succession. The result is that when the view is displayed, the last calls to `SetFont()` and `SetFontSize()` are used when drawing in the view, as shown in Figure 4-2.

Because this example project has very few member functions (intentionally, to keep it easily readable), I'm limited in where I can place a call to `SetHelloViewFont()`. In a larger project, a call to `SetHelloViewFont()` might be invoked from the code that responds to, say, a button click or a menu item selection. After reading Chapter 6 and Chapter 7, *Menus*, you'll be able to easily try out one of these more practical uses for a routine such as `SetHelloViewFont()`.

Accessing a view using FindView()

When a view is created, one of the arguments passed to the view constructor is a string that represents the view's name:

```
fMyView = new MyHelloView(frame, "MyHelloView");
```

The `MyHelloView` class constructor invokes the `BView` constructor to take care of the creation of the view. When it does that, it in turn passes on the string as the second argument, as done here:

```
MyHelloView::MyHelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

If your code provides each view with a unique name, access to any particular view can be easily gained by using the `BWindow` member function `FindView()`. For instance, in this next snippet a pointer to the previously created view with the name "MyHelloView" is being obtained. Assume that the following code is called

from within a MyHelloApplication member function, and that a window has already been created and a reference to it stored in the MyHelloApplication data member fMainWindow:

```
MyHelloView  *theView;

theView = (MyHelloView *)fMainWindow->FindView("MyHelloView");
```

FindView() returns a BView object. The above snippet typecasts this BView object to one that matches the exact type of view being referenced—a MyHelloView view.

FindView() example project

The FindByName project does just that—it finds a view using a view name. This project is another version of this chapter's MyHelloWorld. Here I keep track of the program's one window using a data member in the MyHelloApplication class. A reference to the program's one view isn't, however, stored in a data member in the MyHelloWindow class. Instead, the view is accessed from the window using a call to FindView(). Here's the MyHelloWindow constructor that creates a view named "MyHelloView" and adds it to a new window:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   MyHelloView  *aView;

   frame.OffsetTo(B_ORIGIN);
   aView = new MyHelloView(frame, "MyHelloView");
   AddChild(aView);

   Show();
}
```

The MyHelloWindow member function QuitRequested() has remained unchanged since its introduction in Chapter 1. All it did was post a B_QUIT_REQUESTED and return true. I'll change that by adding a chunk of code. Figure 4-3 shows how the program's window looks just before closing.

```
bool MyHelloWindow::QuitRequested()
{
   MyHelloView  *aView;
   bigtime_t    microseconds = 1000000;

   aView = (MyHelloView *)FindView("MyHelloView");
   if (aView) {
      aView->MovePenTo(BPoint(20, 60));
      aView->DrawString("Quitting...");
      aView->Invalidate();
   }
```

```
        snooze(microseconds);

        be_app->PostMessage(B_QUIT_REQUESTED);
        return(true);
    }
```

Figure 4-3. The FindByName program adds text to a window before closing it

The new version of `QuitRequested()` now does the following:

- Accesses the view named "MyHelloView."
- Calls a few `BView` member functions to draw a string and update the view.
- Pauses for one second.
- Closes the window and quits.

Several lines of code are worthy of further discussion.

The "Accessing a view using FindView()" section in this chapter demonstrates the use of `FindView()` from an existing window object:

```
MyHelloView  *theView;

theView = (MyHelloView *)fMainWindow->FindView("MyHelloView");
```

This latest example demonstrates the use of `FindView()` from within a window member function. The specific object `FindView()` acts on is the one invoking `QuitRequested()`, so unlike the above example, here no `MyHelloWindow` object variable precedes the call to `FindView()`:

```
MyHelloView  *aView;

aView = (MyHelloView *)FindView("MyHelloView");
```

With a reference to the `MyHelloView` object, `QuitRequested()` can invoke any `BView` member function. `MovePenTo()` and `DrawString()` are functions you've seen before—they also appear in the `MyHelloView` member function `Draw()`. `Invalidate()` is new to you. When a view's contents are altered—as they are here with the writing of the string "Quitting..."—the view needs to be updated before the changes become visible onscreen. If the changes are made while the view's window is hidden, then the subsequent act of showing that window brings

on the update. Here, with the window showing and frontmost, no update automatically occurs after the call to `DrawString()`. The `BView` member function `Invalidate()` tells the system that the current contents of the view are no longer valid and require updating. When the system receives this update message, it immediately obliges the view by redrawing it.

Finally, the `snooze()` function is new to you. The BeOS API includes a number of global, or nonmember, functions—`snooze()` is one of them. A global function isn't associated with any class or object, so once the `BApplication`-defined object is created in `main()`, it can be called from any point in a program. The `snooze()` function requires one argument, the number of microseconds for which execution should pause. The parameter is of type `bigtime_t`, which is a `typedef` equivalent to the `int64` datatype. Here, the first call to `snooze()` pauses execution for one million microseconds, or one second, while the second call pauses execution for fifty thousand microseconds, or one-twentieth of one second:

```
bigtime_t  microseconds = 1000000;

snooze(microseconds);
snooze(50000);
```

In this book I'll make occasional use of a few global functions. In particular, you'll see calls to `snooze()` and `beep()` in several examples. You'll quickly recognize a function as being global because it starts with a lowercase character. A global function is associated with one of the Be kits, so you'll find it documented in the *Global Functions* section of the appropriate kit chapter in the Be Book. For instance, `snooze()` puts a thread to sleep, so it's documented in the thread-related chapter of the Be Book, the Kernel Kit chapter. The `beep()` global function plays the system beep. Sound (and thus the `beep()` function) is a topic covered in the Media Kit chapter of the Be Book.

View Hierarchy

A window can hold any number of views. When a window holds more than one, the views fall into a hierarchy.

Top view

Every window contains at least one view, even if none is explicitly created and added with calls to `AddChild()`. That's because upon creation, a window is always automatically given a top view—a view that occupies the entire content area of the window. Even if the window is resized, the top view occupies the

entire window content. A top view exists only to serve as an organizer, or container, of other views. The other views are added by the application. Such an application-added view maps out a window area that has its own drawing characteristics (such as font type and line width), is capable of being drawn to, and is able to respond to messages.

Application-added views and the view hierarchy

Each view you add to the window falls into a window view hierarchy. Any view that is added directly to the window (via a call to the `BWindow` member function `AddChild()`) falls into the hierarchy just below the top view. Adding a few views to a window in this way could result in a window and view hierarchy like those shown in Figure 4-4.

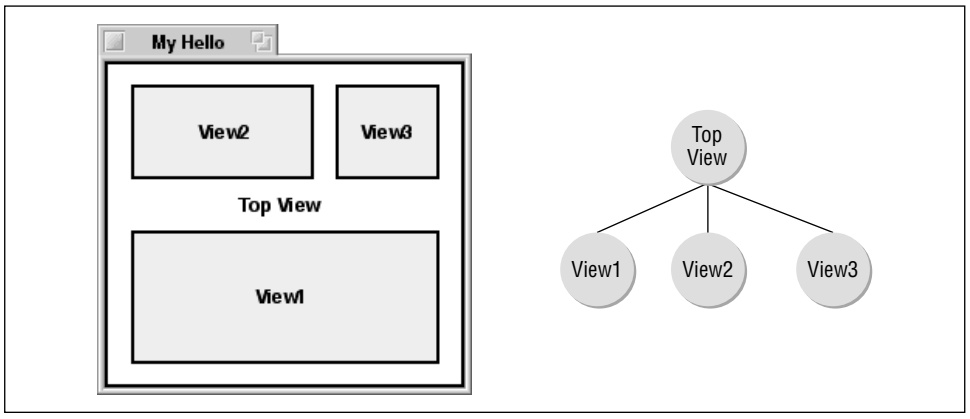

Figure 4-4. A window with three views added to it and that window's view hierarchy

When a view is added to a window, there is no visible sign that the view exists. So in Figure 4-4, the window's views—including the top view—are outlined and are named. The added views have also been given a light gray background. While the view framing, shading, and text have been added for clarity, you could in fact easily create a window that highlighted its views in this way. You already know how to add text to a view using `DrawString()`. Later in this chapter you'll see how to draw a rectangle in a view. And in Chapter 5 you'll see how to change the background color of a view.

The views you add to a window don't have to exist on the same hierarchical level; they can be nested one inside another. Figure 4-5 shows another window with three views added to the top view. Here, one view has been placed inside another.

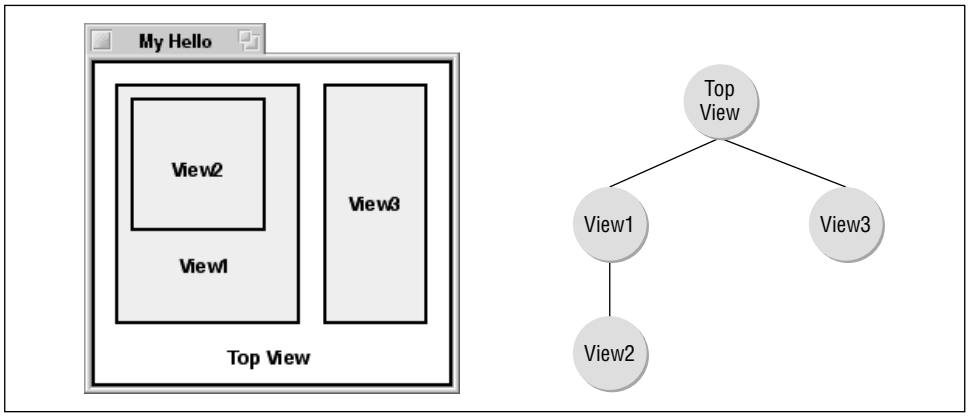

Figure 4-5. A window with nested views added to it and that window's view hierarchy

To place a view within another, you add the view to the container view rather than to the window. Just as the `BWindow` class has an `AddChild()` member function, so does the `BView` class. This next snippet shows a window constructor that creates two views. The first is 200 pixels by 300 pixels in size, and is added to the window. The second 150 pixels by 150 pixels, and is added to the first view.

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "Nested Views", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   BRect        viewFrame;
   MyHelloView  *view1;
   MyHelloView  *view2;

   viewFrame.Set(30, 30, 230, 330);
   view1 = new MyHelloView(viewFrame, "MyFirstView");
   AddChild(view1);

   viewFrame.Set(10, 10, 160, 160);
   view2 = new MyHelloView(viewFrame, "MySecondView");
   view1->AddChild(view2);

   Show();
}
```

Multiple views example project

Later in this chapter you'll see a few example projects that place two views of type `MyHelloView` in a window. Having the views be the same type isn't required, of course—they can be different class types. The TwoViewClasses project defines a view named `MyDrawView` and adds one such view to a window, along with an instance of the `MyHelloView` class with which you're already familiar. Figure 4-6 shows the window that results from running the TwoViewClasses program.

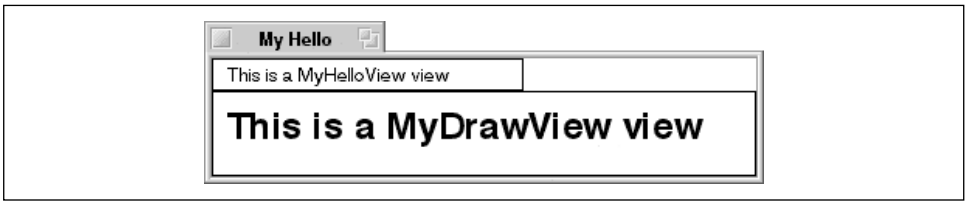

Figure 4-6. A window that holds two different types of views

In keeping with the informal convention of placing the code for a class declaration in its own header file and the code for the implementation of the member functions of that class in its own source code file, the TwoViewClasses project now has a new source code file added to it. Figure 4-7 shows the project window for this project.

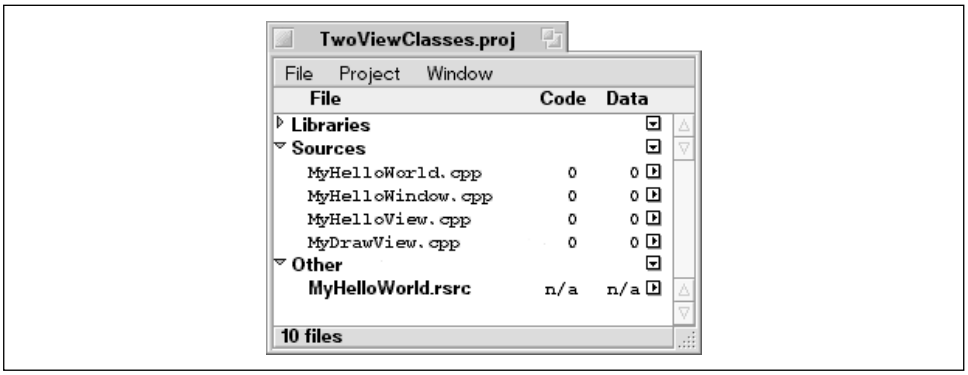

Figure 4-7. The TwoViewClasses project window shows the addition of a new source code file

I haven't shown a project window since Chapter 2, and won't show one again. I did it here to lend emphasis to the way in which projects are set up throughout this book (and by many other Be programmers as well).

I created the new class by first copying the *MyHelloView.h* and *MyHelloView.cpp* files and renaming them to *MyDrawView.h* and *MyDrawView.cpp*, respectively. My intent here is to demonstrate that a project can derive any number of classes from the `BView` class and readily mix them in any one window. So I'll only make a couple of trivial changes to the copied `MyHelloView` class to make it evident that this is a new class. In your own project, the `BView`-derived classes you define may be very different from one another.

With the exception of the class name and the name of the constructor, the MyDrawView class declaration is identical to the MyHelloView class declaration. From the *MyDrawView.h* file, here's that declaration:

```
class MyDrawView : public BView {

public:
                  MyDrawView(BRect frame, char *name);
virtual  void     AttachedToWindow();
virtual  void     Draw(BRect updateRect);
};
```

Like the MyHelloView constructor, the MyDrawView constructor is empty:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

The MyDrawView member function AttachedToWindow() sets up the view's font and font size. Whereas the MyHelloView specified a 12-point font, the MyDrawView specifies a 24-point font:

```
void MyHelloView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);
}
```

Except for the text drawn to the view, the MyDrawView member function Draw() looks like the MyHelloView version of this function:

```
void MyDrawView::Draw(BRect)
{
   BRect frame = Bounds();
   StrokeRect(frame);

   MovePenTo(BPoint(10, 30));
   DrawString("This is a MyDrawView view");
}
```

To create a further contrast in the way the two views display text, I turned to the MyHelloView and made one minor modification. In the AttachedToWindow() member function of that class, I changed the font set by SetFont() from be_bold_font to be_plain_font. Refer to Figure 4-6 to see the difference in text appearances in the two views.

In order for a window to be able to reference both of the views it will hold, a new data member has been added to the MyHelloWindow class. In the *MyHelloWindow.h* header file, you'll find the addition of a MyDrawView data member named fMyDrawView:

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual  bool     QuitRequested();

   private:
      MyHelloView       *fMyView;
      MyDrawView        *fMyDrawView;
};
```

In the past the `MyHelloWindow` constructor created and added a single view to itself. Now the constructor adds a second view:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.Set(0, 0, 200, 20);
   fMyView = new MyHelloView(frame, "MyHelloView");
   AddChild(fMyView);

   frame.Set(0, 21, 350, 300);
   fMyDrawView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyDrawView);

   Show();
}
```

Both views have been added directly to the window (to the top view), rather than to another view, so both views are on the same level in the window's view hierarchy. The `Draw()` function of each view type includes code to frame the view, so you can easily see the results of any view size changes you might make to the views here in the `MyHelloWindow` constructor.

Coordinate System

In order to specify where a window is to be placed on the screen and where a view is to be placed within a window, a coordinate system is required.

Global coordinate system

To allow a programmer to reference any point on the computer screen, Be defines a coordinate system that gives every pixel a pair of values: one for the pixel's distance from the left edge of the screen and one for the pixel's distance from the top of the screen. Figure 4-8 points out a few pixels and their corresponding coordinate pairs.

For display devices, the concept of fractional pixels doesn't apply. Consider a window that is to have its top left corner appear 100 pixels from the left edge of the screen and 50 pixels from the top of the screen. This point is specified as (100.0,

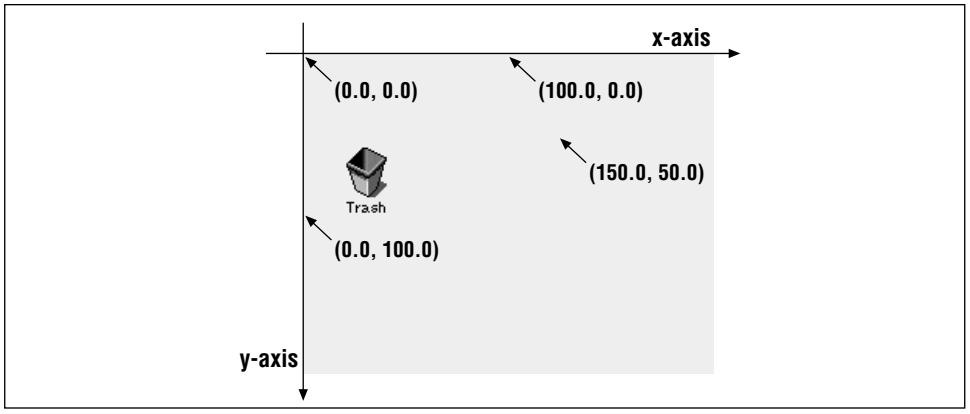

Figure 4-8. The global coordinate system maps the screen to a two-dimensional graph

50.0). If the point (100.1, 49.9) is used instead, the result is the same—the window's corner ends up 100 pixels from the left and 50 pixels from the top of the screen.

The above scenario begs the question: if the coordinates of pixel locations are simply rounded to integral values, why use floating points at all? The answer lies in the current state of output devices: most printers have high resolutions. On such a device, one coordinate unit doesn't map to one printed dot. A coordinate unit is always 1/72 of an inch. If a printer has a resolution of 72 dots per inch by 72 dots per inch (72 dpi × 72 dpi), then one coordinate unit would in fact translate to one printed dot. Typically printers have much higher resolutions, such as 300 dpi or 600 dpi. If a program specifies that a horizontal line be given a height of 1.3 units, then that line will occupy one row of pixels on the screen (the fractional part of the line height is rounded off). When that same line is sent to a printer with a resolution of 600 dpi, however, that printer will print the line with a height of 11 rows. This value comes from the fact that one coordinate unit translates to 8.33 dots (that's 1/72 of 600). Here there is no rounding of the fractional coordinate unit, so 1.3 coordinate units is left at 1.3 units (rather than 1 unit) and translates to 11 dots (1.3 times 8.33 is 10.83). Whether the line is viewed on the monitor or on hardcopy, it will have roughly the same look—it will be about 1/72 inch high. It's just that the rows of dots on a printer are denser than the rows of pixels on the monitor.

Window coordinate system

When a program places a view in a window, it does so relative to the window, not to the screen. That is, regardless of where a window is positioned on the screen when the view is added, the view ends up in the same location within the content area of the window. This is possible because a window has its own

coordinate system—one that's independent of the global screen coordinate system. The type of system is the same as the global system (floating point values that get larger as you move right and down)—but the origin is different. The origin of a window's coordinate system is the top left corner of the window's content area.

When a program adds a view to a window, the view's boundary rectangle values are stated in terms of the window's coordinate system. Consider the following window constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   MyHelloView  *aView;
   BRect        viewFrame(20.0, 30.0, 120.0, 130.0);

   aView = new MyHelloView(viewFrame, "MyHelloView");
   AddChild(aView);

   Show();
}
```

The coordinate systems for the window and the view are different. The window's size and screen placement, which are established by the `BRect` variable `frame` that is passed to the constructor, are expressed in the global coordinate system. The view's size and placement, established by the local `BRect` variable `viewFrame`, are expressed in the window coordinate system. Regardless of where the window is placed, the view `aView` will have its top left corner at point (20.0, 30.0) within the window.

In all previous examples, the arguments to a `BRect` constructor, or to the `BRect` member function `Set()`, were integer values, such as (20, 30, 120, 130). Since none of the examples were concerned with high precision printouts, that technique worked fine. It also may have been comforting to you if you come from a Mac or Windows programming background, where rectangle boundaries use integral values. Now that we've seen the true nature of the BeOS coordinate system, however, we'll start—and continue—to use floating point values.

View coordinate system

When a program draws in a view, it draws relative to the view, not to the window or the screen. It doesn't matter where a window is onscreen, or where a view is within a window—the drawing will take place using the view's own coordinate system. Like the global and window coordinate systems, the view coordinate system is one of floating point coordinate pairs that increase in value from left to right

and from top to bottom. The origin is located at the top left corner of the view. Consider this version of the `MyHelloView` member function `Draw()`:

```
void MyHelloView::Draw(BRect)
{
   MovePenTo(BPoint(10.0, 30.0));
   DrawString("Hello, My World!");
}
```

The arguments in the call to the `BView` member function `MovePenTo()` are local to the view's coordinate system. Regardless of where the view is located within its window, text drawing will start 10 units in from the left edge of the view and 30 units down from the top edge of the view.

Figure 4-9 highlights the fact that there are three separate coordinate systems at work in a program that displays a window that holds a view.

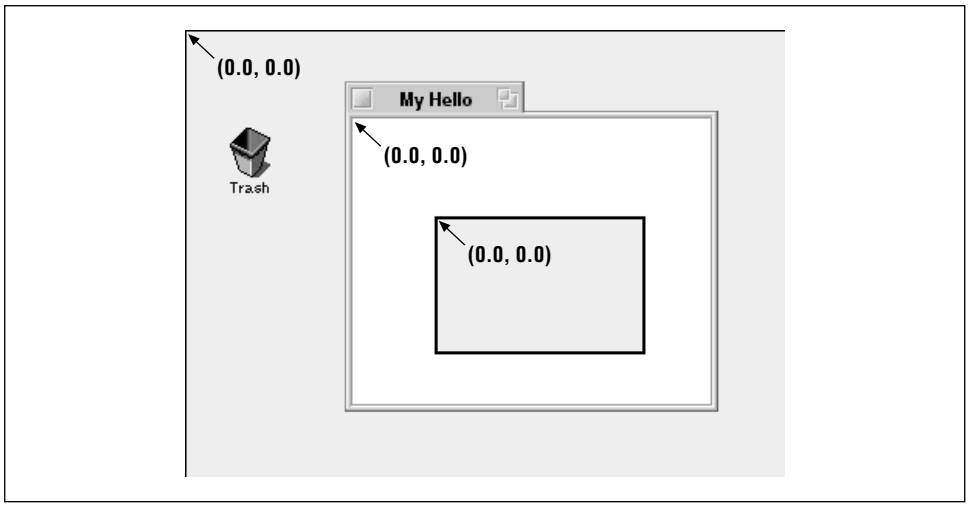

Figure 4-9. The screen, windows, and views have their own coordinate systems

Coordinate system example projects

To determine the size of a view in its own coordinate system (whether the view resides in a window or within another view), begin by invoking the `BView` member function `Bounds()`. In this chapter's OneView project, a call to this function has been added to the MyHelloView member function `Draw()`. One other `BView` member function call has been added too—a call to `StrokeRect()`. This routine draws a rectangle at the coordinates specified by the `BRect` argument passed to it:

```
void MyHelloView::Draw(BRect)
{
   BRect frame = Bounds();
   StrokeRect(frame);
```

```
    MovePenTo(BPoint(10.0, 30.0));
    DrawString("Hello, My World!");
}
```

Since the rectangle returned by the `Bounds()` function call is relative to the view's own coordinate system, the `left` and `top` fields are always 0.0. The `right` and `bottom` fields reveal the view's width and height, respectively.

To find a view's boundaries relative to the window or view it resides in, call the `BView` member function `Frame()`. The rectangle returned by a call to `Frame()` has `left` and `top` fields that indicate the view's distance in and down from the window or view it resides in.

The OneView project creates a single `MyHelloView` view and adds it to a window. These steps take place in the `MyHelloWindow` constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
    frame.OffsetTo(B_ORIGIN);
    fMyView = new MyHelloView(frame, "MyHelloView");

    AddChild(fMyView);
    Show();
}
```

Now that you know about the different coordinate systems, setting up the view rectangle might make more sense to you. In the above snippet, the `BRect` parameter `frame` holds the coordinates of the window. These coordinates directly define the screen placement of the window and indirectly define the size of the window (subtract `frame.left` from `frame.right` to get the window's width, and subtract `frame.top` from `frame.bottom` to get the window's height). Calling the `BRect` member function `OffsetTo()` with `B_ORIGIN` as the parameter shifts these coordinates so that each of the `frame.left` and `frame.top` fields has a value of 0.0. The overall size of the `frame` rectangle itself, however, doesn't change—it is still the size of the window. It just no longer reflects the screen positioning of the window. Next, the view that is to be added to the window is created. The view is to be positioned in the window using the window's coordinate system, so if the view is to fit snugly in the window, the view must have its top left corner at the window's origin. The `frame` rectangle that was initially used to define the placement and size of the window can now be used to define the placement and size of the view that is to fill the window.

When the `BWindow` member function `Show()` is invoked from the window constructor, the window is drawn to the screen and the view's `Draw()` function is automatically called to update the view. When that happens, the view is outlined—the `Draw()` function draws a line around the perimeter of the view. Figure 4-10 shows the result of creating a new window in the OneView project.

Because the window's one view is exactly the size of the content area of the window, the entire content area gets a line drawn around it.

Figure 4-10. Drawing a rectangle around the OneView window's view

The OneSmallView project is very similar to the OneView project—both draw a frame around the one view that resides in the program's window. To demonstrate that a view doesn't have to occupy the entire content area of a window, the OneSmallView project sets up the view's boundary rectangle to be smaller than the window. This is done in the `MyHelloWindow` constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.Set(100.0, 80.0, 250.0, 120.0);
   fMyView = new MyHelloView(frame, "MyHelloView");
   AddChild(fMyView);

   Show();
}
```

Here the line that offsets the window boundary rectangle (the `BRect` parameter `frame`) has been replaced by one that calls the `BRect` member function `Set()` to reset all the values of the `frame` rectangle. Figure 4-11 shows the resulting window. Note that a view is aware of its own boundaries, so that when you try to draw (or write) beyond a view edge, the result is truncated.

Figure 4-11. When a view is too small for the window content

Messaging

As discussed in Chapter 1, the Application Server communicates with (serves) an application by making the program aware of user actions. This communication is done in the form of *system messages* sent from the server to the application. Mes-

sages are received by a window and, often, passed on to a view in that window. The BeOS shoulders most of the responsibility of this communication between the Application Server, windows, and views. Your application (typically a view in a window in your application) is responsible for performing some specific action in response to a message.

System Messages

A system message is sent from the Application Server to a `BLooper` object. Both the `BApplication` and `BWindow` classes are derived from `BLooper`, so objects of these two classes (or objects of classes derived from these two classes) can receive messages. The Application server is responsible for directing a system message to the appropriate type of object.

The message loop of a program's `BWindow`-derived object receives messages that hold information about user actions. If the user typed a character, that character may need to be entered into a window. If the user clicked a mouse button, that click may have been made while the cursor was over a button in the window. The system message types of these two user actions are `B_KEY_DOWN` and `B_MOUSE_DOWN`. Such `BWindow`-directed system messages are referred to as *interface messages*.

The message loop of a program's `BApplication`-derived object receives messages that pertain to the application itself (as opposed to messages that pertain to a window or view, which are sent to a `BWindow`-derived object). If the user chooses the About menu item present in most programs, the program dispatches to the application object a message of type `B_ABOUT_REQUESTED`. Such `BApplication`-directed system messages are referred to as *application messages*.

See the Application Kit chapter of the Be Book for a description of all of the application messages, and the Interface Kit chapter for a description of all the interface messages.

System message dispatching

When a system message reaches a looper object (such as the application object or a window object), that object handles, or dispatches, the message by automatically invoking a virtual hook function. Such a function is declared `virtual` so that your own derived classes can override it in order to reimplement it to match your program's specific needs. In that sense, you're "hooking" your own code onto the system code.

Each system message has a corresponding hook function. For the three system messages mentioned above (B_ABOUT_REQUESTED, B_KEY_DOWN, and B_MOUSE_DOWN), those functions are AboutRequested(), KeyDown(), and MouseDown(). The application object itself handles a B_ABOUT_REQUESTED message by calling the BApplication member function AboutRequested(). A window object, on the other hand, passes a B_KEY_DOWN or B_MOUSE_DOWN message on to the particular view object to which the message pertains. This view object then invokes the BView member function KeyDown() or MouseDown() to handle the message.

Types of hook functions

For some system messages, the hook function defined by the Be class takes care of all the work suggested by the message. For instance, a click on a window's zoom button results in a B_ZOOM message being sent to the affected window. The receiving of this message automatically brings about the execution of the BWindow member function Zoom(). This hook function is fully implemented, meaning that you need to add no code to your project in order to support a click in a window's zoom button.

All hook functions are declared virtual, so your code can override even fully implemented ones. Unless your application needs to perform some nonstandard action in response to the message, though, there's no need to do so.

For other system messages, the hook function is implemented in such a way that the most common response to the message is handled. A program may override this type of hook function and reimplement it in such a way that the new version handles application-specific needs. This new application-defined version of the hook function may also call the original Be-defined BView version of the routine in order to incorporate the default actions of that BView version. An example of this type of hook function is ScreenChanged(), which is invoked in response to a B_SCREEN_CHANGED message. When the user changes the screen (perhaps by altering the monitor resolution), the application may need to make special adjustments to an open window. After doing that, the application-defined version of ScreenChanged() should invoke the BView-defined version of this routine so that the standard screen-changing code that's been supplied by Be can execute too.

Finally, for some system messages, the hook function implementation is left to the application. If an application is to respond to user actions that generate messages of types such as B_KEY_DOWN and B_MOUSE_DOWN, that application needs to override BView hook functions such as KeyDown() and MouseDown().

Interface messages

A system message directed at the application object is an application message, while a system message directed at a window object is an interface message. Responding to user actions is of great importance to a user-friendly application, so the remainder of this chapter is dedicated to illustrating how a project goes about doing this. In particular, I'll discuss the handling of two of the interface messages (`B_KEY_DOWN` and `B_MOUSE_DOWN`). Summarized below are several of the interface messages; refer to the Interface Kit chapter of the Be Book for a description of each of the 18 message types.

`B_KEY_DOWN`
: Goes to the active window in response to the user pressing a character key. The recipient window invokes the `BView` hook function `KeyDown()` of the affected view. The affected view is typically one that accepts text entry, such as a view of the yet-to-be-discussed `BTextControl` or `BTextView` classes. An example of handling a `B_KEY_DOWN` message is presented later in this chapter.

`B_KEY_UP`
: Is sent to the active window when the user releases a pressed character key. The recipient window invokes the `BView` hook function `KeyUp()` of the affected view. Typically, a program responds to a `B_KEY_DOWN` message and ignores the `B_KEY_UP` message that follows. In other words, the program doesn't override the `BView` hook function `KeyUp()`.

`B_MOUSE_DOWN`
: Is sent to the window over which the cursor was located at the time of the mouse button click. The window that receives the message calls the `BView` hook function `MouseDown()` of the view the cursor was over at the time of the mouse button click.

`B_MOUSE_UP`
: Reaches the window that was affected by a `B_MOUSE_DOWN` message when the user releases a pressed mouse button. The `MouseDown()` hook function that executes in response to a `B_MOUSE_DOWN` message often sufficiently handles a mouse button click, so a `B_MOUSE_UP` message is often ignored by a program. That is, the program doesn't override the `BView` hook function `MouseUp()`.

`B_MOUSE_MOVED`
: Is sent to a window when the user moves the cursor over the window. As the user drags the mouse, repeated `B_MOUSE_MOVED` messages are issued by the Application Server. As the cursor moves over one window to another, the window to which the messages are sent changes. When the mouse is moved over the desktop rather than a window, a `B_MOUSE_MOVED` message is sent to the Desktop window of the Tracker.

Mouse Clicks and Views

When a window receives a B_MOUSE_DOWN message from the Application Server, the window object (without help from you) determines which of its views should respond. It is that view's MouseDown() hook function that is then invoked.

The ViewsMouseMessages project includes a MouseDown() routine with the MyHelloView class in order to make the program "mouse-click aware." The ViewsMouseMessages program displays a single window that holds two framed MyHelloView views. Clicking the mouse while the cursor is over either view results in the playing of the system beep.

The mechanism for responding to a mouse click has already been present in every example project in this book, so there's very little new code in the ViewsMouseMessages project. The ViewsMouseMessages program, and every other program you've seen in this book, works as follows: when the user clicks the mouse button while the cursor is over a window, the Application Server sends a B_MOUSE_DOWN message to the affected window, causing it to invoke the affected view's MouseDown() hook function. The MyHelloView class is derived from the BView class, and the BView class defines its version of MouseDown() as an empty function. So unless the MyHelloView class overrides MouseDown(), it inherits this "do-nothing" routine. In all previous examples, a mouse button click while the cursor was over a view resulted in the execution of this empty routine—so effectively the mouse button click was ignored. The ViewsMouseMessages project overrides MouseDown() so that a mouse button click with the cursor over a view now results in something happening. Here's the latest version of the MyHelloView class definition, with the addition of the MouseDown() declaration:

```
class MyHelloView : public BView {

   public:
                     MyHelloView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);
      virtual void   MouseDown(BPoint point);
};
```

The one MouseDown() parameter is a BPoint that is passed to the routine by the Application Server. This point parameter holds the location of the cursor at the time the mouse button was clicked. The values of the point are in the view's coordinate system. For example, if the cursor was over the very top left corner of the view at the time of the mouse click, the point's coordinates would be close to (0.0, 0.0). In other words, both point.x and point.y would have a value close to 0.0.

To verify that the B_MOUSE_DOWN message has worked its way to the new version of MouseDown(), the implementation of MouseDown() sounds the system beep:

```
void MyHelloView::MouseDown(BPoint point)
{
   beep();
}
```

Recall that `beep()` is a global function that, like the `snooze()` routine covered earlier in this chapter, can be called from any point in your project's source code.

Key Presses and Views

In response to a `B_MOUSE_DOWN` message, a window object invokes the `MouseDown()` function of the affected view. For the window object, determining which view is involved is simple—it chooses whichever view object is under the cursor at the time of the mouse button click. This same test isn't made by the window in response to a `B_KEY_DOWN` message. That's because the location of the cursor when a key is pressed is generally insignificant. The scheme used to determine which view's `KeyDown()` hook function to invoke involves a focus view.

Focus view

A program can make any view the focus view by invoking that view's `MakeFocus()` function. For a view that accepts typed input (such as `BTextControl` or a `BTextView` view), the call is made implicitly when the user clicks in the view to activate the insertion bar. Any view, however, can be made the focus view by explicitly calling `MakeFocus()`. Here a click of the mouse button while the cursor is over a view of type `MyHelloView` makes that view the focus view:

```
void MyHelloView::MouseDown(BPoint point)
{
   MakeFocus();
}
```

Now, when a key is pressed, the `KeyDown()` hook function of the last clicked-on view of type `MyHelloView` will automatically execute.

Because a `MyHelloView` view doesn't accept keyboard input, there is no obvious reason to make a view of this type the focus view. We haven't worked with many view types, so the above example must suffice here. If you're more comfortable having a reason for making a `MyHelloView` accept keyboard input, consider this rather contrived scenario. You want the user to click on a view of type `MyHelloView` to make it active. Then you want the user to type any character and have the view echo that character back—perhaps in a large, bold font. Including the above `MouseDown()` routine in a project suffices to make the view the focus view when clicked on. Now a `MyHelloView` `KeyDown()` routine can be written to examine the typed character, clear the view, and draw the typed character.

KeyDown() example project

The ViewsKeyMessages project adds to the ViewsMouseMessages project to create a program that responds to both mouse button clicks and key presses. Once again, a mouse button click while the cursor is over a view results in the sounding of the system beep. Additionally, ViewsKeyMessages beeps twice if the Return key is pressed and three times if the 0 (zero) key is pressed.

To allow a `MyHelloView` view to respond to a press of a key, the `BView` hook function `KeyDown()` needs to be overridden:

```
class MyHelloView : public BView {

   public:
                     MyHelloView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);
      virtual void   MouseDown(BPoint point);
      virtual void   KeyDown(const char *bytes, int32 numBytes);
};
```

The first `KeyDown()` parameter is an array that encodes the typed character along with any modifier keys (such as the Shift key) that were down at the time of the key press. The second parameter tells how many bytes are in the array that is the first parameter. As with all hook functions, the values of these parameters are filled in by the system and are available in your implementation of the hook function should they be of use.

The `KeyDown()` routine responds to two key presses: the Return key and the 0 (zero) key. Pressing the Return key plays the system beep sound twice, while pressing the 0 key plays the sound three times:

```
void MyHelloView::KeyDown(const char *bytes, int32 numBytes)
{
   bigtime_t  microseconds = 1000000;

   switch ( *bytes ) {

      case B_RETURN:
         beep();
         snooze(microseconds);
         beep();
         break;

      case '0':
         beep();
         snooze(microseconds);
         beep();
         snooze(microseconds);
         beep();
         break;
```

```
            default:
               break;
         }
      }
```

There are a number of Be-defined constants you can test bytes against; `B_RETURN` is one of them. The others are: `B_BACKSPACE`, `B_LEFT_ARROW`, `B_INSERT`, `B_ENTER`, `B_RIGHT_ARROW`, `B_DELETE`, `B_UP_ARROW`, `B_HOME`, `B_SPACE`, `B_DOWN_ARROW`, `B_END`, `B_TAB`, `B_PAGE_UP`, `B_ESCAPE`, `B_FUNCTION_KEY`, and `B_PAGE_DOWN`. For a key representing an alphanumeric character, just place the character between single quotes, as shown above for the 0 (zero) character.

Notice that calls to the global function `snooze()` appear between calls to the global function `beep()`. The `beep()` routine executes in its own thread, which means as soon as the function starts, control returns to the caller. If successive, uninterrupted calls are made to `beep()`, the multiple playing of the system beep will seem like a single sound.

Only the focus view responds to a key press, so the ViewsKeyMessages program needs to make one of its two views the focus view. I've elected to do this in the `MyHelloView MouseDown()` routine. When the user clicks on a view, that view becomes the focus view:

```
void MyHelloView::MouseDown(BPoint point)
{
   beep();

   MakeFocus();
}
```

When the user then presses a key, the resulting `B_KEY_DOWN` message is directed at that view. Since the views are derived from the `BView` class, rather than a class that accepts keyboard input, a typed character won't appear in the view. But the view's `KeyDown()` routine will still be called.