8. Preferences system

By using the GameKit Preferences system, you can allow a player to customize a game. The GameKit's preferences system includes one object, the PreferencesBrain, and several functions to simplify obtaining of preference values. It also has code to handle a multiple-pane inspector-style panel for manipulation of preference values. The Preferences Brain object provided is already set up to record several common preference value that are common to most games, such as starting level, sound on/off, and high score server settings. Nearly all the preferences values supplied by the standard PreferencesBrain object are supported by the GameKit already±meaning that in most cases, you won't need to implement code to make use of the settings. If you decide to make a subclass which adds new preferences values, then you will need to add code to support them, of course. The PreferencesBrain also can register ªunfairº settings which, if used, will prevent a game's final score to be sent to a network high score server. *I would like to make a general Preferences object to put in the DAYMiscKit and then have this object be a subclass of that (one which defines the GameKit standard preferences); this would make a whole lot more sense architecture-wise and would give an object generally useful in other types of apps. Well, when I get around to it¼*

The PreferencesBrain object

The PreferencesBrain object needs to be able to talk to many of the other GameKit objects; as Preferences change, it can alert the GameKit objects of changes in the user's settings. Also, many GameKit objects will query the PreferencesBrain about the current settings of a user's preferences. *(There is little sense to how this is done¼sometimes a message is sent, sometimes a query is done. I really ought to clean this up. The trade-off is that (1) it is silly to store a preference setting multiple times±in the PreferencesBrain* ***and*** *in the objects that use the preference, but (2) querying the object incurs message*

overhead, so multiple copies of the data will run faster. When I clean this up, I'll probably go toward speeding things up even though this would be a poor OOP practice implementation-wise. Any comments? I have noticed that some values make sense to be queried (like, ªIs sound on?º) while others seem to want to invoke messages (like, ªtake the border awayº). Well, I have to think this one through a bit more to come up with a consistent way to handle it. Since it is pretty much transparent to you±it's all automatic and happening internal to the GameKit±whatever I decide will probably not affect you much anyway.)

The preferences values that the PreferencesBrain handles ªout of the boxº are in the following table:

Name	**Type**	**Range**	**Default**
UseServer	boolean	0-1	0
ServerName	string	anything	*
Alerts	boolean	0-1	1
Border	boolean	0-1	1
AutoUnPause	boolean	0-1	1
AutoStart	boolean	0-1	1
SoundOn	boolean	0-1	1
DemoSound	boolean	0-1	1
MusicOn	boolean	0-1	1
StartLevel	integer	1-20	1
GameSpeed	integer	0-2	1
Version	string	anything	0.0
PlayerName	string	anything	User's real name (login name if real unknown)
Key0	string	any char	a
Key1	string	any char	z
Key2	string	any char	l
Key3	string	any char	;
Key4	string	any char	<space>

Note that there are a few preferences, such as background color and background image that are handled directly by the GameView object. *Some of this handling should really be split between the GameView and the PreferencesBrain. A future project, I guess.* The preferences listed above have the specific functions in the GameKit. If you don't want a game to use certain preferences, simply leave the appropriate controls off of the preferences panel and the user will never know about them. (Note that changing the defaults from the command line will still be possible, however.) All the above parameters are basically implemented by the GameKit (except the Key preferences) so you need to do *nothing* in order for your game to make use of them.

The functions of the preferences are described in the paragraphs below:

UseServer	Whether or not the High Score subsystem should use a network server. (If NO, a local server will be used instead.)
ServerName	The name of the machine upon which resides the network high score server. Note that a * means ªthe first machine that you find, if anyº and causes a *very* slow search to be performed to find the machine. If the user knows the actual machine's name or IP address, it should be given.
Alerts	Allows the user to disable the alert panels which pop up; things like warning about quitting when a game is in progress or asking if they really want to abort a game or not.
Border	Allows the border around a GameView to be removed, shrinking the window a little bit. The border normally looks good, but some games±like Columns±can look really nifty if the window doesn't appear to have a border and there is a BackSpace module like Space running behind it.
AutoUnPause	This decides if a game should un pause itself when the game's window becomes key (or is unhidden). Some find this handy, but most can find it disconcerting. *Perhaps I should make the default to be* ***off****. What do you think?*
AutoStart	Should a new game start immediately after the game is finished launching?
SoundOn	Is the sound on of off? This turns the sound section on or off completely, independent of the music section.
DemoSound	Should the game play sound effects and music when in demo mode? This decides.
MusicOn	Whether or not musical scores should be played.
StartLevel	The level that a new game will start at.
GameSpeed	How fast the game will run. *Currently, the GameKit itself doesn't really support this very much; it gives you the number and you decide how to make use of it. When the state machine section is completed, this will probably be built into it.*
Version	Stores the version number of the game. This way, when a new version is played for the first time, the GameBrain will ask the InfoController to pop up a README panel (release notes). Since users never look at release notes, this is a way to try and get them to pay attention. This mechanism keys off of the version number stored in the global string table that the GameKit uses. (See the discussion in the GameBrain and InfoController sections.)
PlayerName	The default name that is placed in the high score panel when a player gets a new high score. This

default changes to whatever the player enters.

Key[0-31] These store the player's choices for keys to control the game. Right now these are stored as single characters, which means that certain special keys such as arrows cannot be used. *In the future I plan to fix this so that a character code and character set are stored. This will allow pretty much any key on the keyboard to be specified.* The PreferencesBrain by default only deals with keys 0-4 (5 total) since many games will only need four or five keys. If you need more keys, override the ±**init** method and set the instance variable **numKeys** to the value needed. To set the default keys used by your game, in ±**init** use something like this to set the value of each default key: strcpy(defkey[keyNumber], "a"); Be sure that you make both these changes *after* the call to super's ±**init** method. As discussed in other sections, it is up to you to respond to the various key presses; the GameKit currently recognizes ªpº as pause and ªnº as new game, and does nothing more with keypresses.

You will note, looking at the class spec sheet, that the PreferencesBrain object has quite a few outlets and action methods that are to all be connected to various controls on the Preferences Panel in the PreferencesPanel.nib file. The easiest thing to do is to use one of the template .nib files provided with the GameKit since it will alredy have all the connections mad for you. If you need to start from scratch for any reason, look at the class spec sheet for specific about the required connections to make things work properly. (Most connections are obvious; it is just difficult to remember to make *all* of them because there are so many.) *Right now the outlets and action methods don't really follow any sort of naming convention, which makes it all very confusing. I will be cleaning this up in the future, so please bear with me here.*

If the above preferences are sufficient for your use, then you only need to drop one of the Preferences Panel prototype .nib files into your application to use the PreferencesBrain object. The only time you will have to do anything special is when you want to change the range allowed for a preference or add your own preference. In this case, you will need to subclass the PreferencesBrain object and tailor one of the PreferencesPanel.nib prototypes to add any extra controls that are needed. The next sections describe how to go about doing this.

Adding new preferences

If you have any ideas of how this process might be made simpler, or of general method prototypes that might be helpful here, please discuss them with me; I'd like to simplify this is much as possible!

It is very easy to add a new preference to the PreferenesBrain object. This is where your understanding of the NEXTSTEP interface objects (mainly controls) will pay off hugely. In order to add a new preference, you should create several new

methods to deal with the preference and should also override a few of the PreferencesBrain methods. The easiest way to describe what to do is to actually step through the addition of a new preference. One example will be described here; the addition of the Big/Small switch found in PacMan. The process of adding a new preference will always follow these steps. (Note that in the case of a preference like the starting level's Slider and TextField combination will require the action method to keep one contol in sync with the other, but in all other respects, the process is identical.)

Big/Small example

In PacMan, there is a radio switch which allows the user to choose between a big and small screen to play the game in. Internally, I have chosen to call this the ªscaleº. This preference setting may have the value of a ª1º or a ª2º. The default is ª1º, which stands for the smaller screen.

To add the preference, the first thing to do is add instance variables to the PreferencesBrain object. To hold the value of the preference, there is an instance variable called ªscaleº which is of type **int**. There is also an **id** which points to the Matrix. In general, a new preference requires a new instance variable to hold its value and as many **id**s as are necessary to point to the NEXTSTEP Control objects which display the value of the preference and allow it to be manipulated.

Next, three methods are added to the PreferencesBrain object. They are perform the following: (1) allow any object to find out the value of the new preference, (2) allow any object to change the value of the preference to a specific number, and (3) an action method which may be called by the NEXTSTEP control to change the setting. These methods are named, respectively, ±**scale**, ±**setScale:**, and ±**scaleChange:**. See the code below to see how I chose to actually implement these methods. Typically, any preference you add to a PreferencesBrain will add three methods like those described above. The second method is usually optional, but the first and third are necessary. Note that in your game's code, in order to take advantage of the new preference, you will need to query the PreferencesBrain object about the preference value at key points; the first method allows for this. In some cases, you may have the third message send another message to some object in your game to immediately alter it's behavior.

The last step to take is to override specific PreferencesBrain methods to add some new functionality. These methods are: ±**readDefaults:**, ±**writeDefaults:**, **±revert:**, and **±refresh**. You override the first two to be sure that you new preference is read from and stored to the defaults database, respectively. The ±**revert** method is used to return your preference back to it's default state, and the ±**refresh** method sets the NEXTSTEP Controls' states so that they properly reflect the values of the internal settings. Below are the .h and .m files for a subclass that adds this single new preference to the PreferencesBrain object. You will notice that this is basically simple, no-brainer code. *For this reason, I may try to come up with a utility app that allows you to create skeleton subclasses by listing preference names, types, ranges, and defaults; it should be really easy to do! Would anyone find this useful?*

Here is the header (.h) file:

```
#import <gamekit/gamekit.h>  // superclass, etc.
```

```
@interface NewPreferencesBrain:PreferencesBrain
{
    id  scaleMatrix;  // buttons to set size of game screen
    int scale;        // holds the screen size internally
}

// new methods
- (int)scale;
- setScale:(int)newScale;
- scaleChange:sender;

// overridden methods:
- readDefaults:sender;
- writeDefaults:sender;
- revert:sender;
- refresh;

@end
```

Here is the implementation (.m) file:

```
#import "NewPreferencesBrain.h"
#define DEFAULTscale 1  // default is a 1 or 2

@implementation PacManPreferencesBrain

- (int)scale { return scale; }

- setScale:(int)newScale
{ // keep the value in the accepted range
        if ((newScale < 1) || (newScale > 2)) return self;
        scale = newScale;
        return self;
}

- scaleChange:sender // sender must be a Matrix of radio buttons with the tags 1=small, 2=big
{
        scale = [[sender selectedCell] tag]; // get the new value
        [gameScreen setScale:scale]; // tell the screen to adjust its size immediately
        return self; // (I assume that GameView is subclassed to respond to setScale:)
}

- readDefaults:sender // get preferences from defaults database
{
    [super readDefaults:sender]; // get the GameKit versions
    scale = getIntPreference("Scale", 1, 2, DEFAULTscale); // a new pref
    return self;
}

- writeDefaults:sender // save preferences in defaults database
```

```
{
    [super writeDefaults:sender];
    putIntPreference("Scale", scale);
    return self;
}

- revert:sender  // return to default values
{
    scale = DEFAULTscale;  // this is the default value of the scale
    return [super revert:sender]; // update the panel
}

- refresh // puts current prefs values into the panel
{
    [super refresh];
    [scaleMatrix selectCellWithTag:scale];
    return self;
}
```

Other than the previous discussion, there are two important things to note in the above code. First, there are several functions which you may use to simplify the reading and writing of preference values. They are defined like this:

```
extern BOOL getBOOLPreference(const char *name, BOOL def);
extern int getIntPreference(const char *name, int min, int max, int def);
extern const char *getStringPreference(const char *name, const char *def);
extern void putIntPreference(const char *name, int value);
extern void putBOOLPreference(const char *name, BOOL value);
extern void putStringPreference(const char *name, const char *value);
```

The functions performed by each are obvious from the names; in each of the retrieval functions, the **def** argument give the default value of the preference if it is not found in the database. In the case of the integer preferences, you can also limit them to always be within a certain range. Use of these functions should make it much easier to add preferences. *Note that right now no defaults vectors are registered before reading preferences, so there is a bit of a performance hit to be taken. It is not necessary to register defaults, but in a future implementation of PreferencesBrain this will be done±at least for the GameKit preferences±so that better performance is achieved. The hit is so small, though, that I do not consider this a very high priority.*

Another important thing to notice is how the ±**revert:** method is overridden. The ±**revert:** method makes the GameScreen check for changes in preferences and updates the Preferences¼ panel, so the call to **super** should follow any reverting done by your code, rather than preceed it, as would normally be done when overriding a method.

Once your PreferencesBrain subclass is complete, the only thing left to do is add the necessary controls to the Preferences¼ panel in the .nib file. The connections will be like the ones already in the .nib. The new outlets in the PreferencesBrain subclass is connected to the appropriate control, and the new controls send action messages to the PreferencesBrain subclass which alter the appropriate preference. In the example above, the **scaleMatrix** outlet connects to the scale buttons Matrix

and the Matrix in turn sends a ±**scaleChange:** message to the NewPreferences Brain object. And that's all you have to do! aside from the code to actually implement the preference somewhere inside of your game.

Another important thing to notice is how the ±**revert:** method is overridden. The ±**revert:** method makes the GameScreen check for changes in preferences and updates the Preferences¼ panel, so the call to **super** should follow any reverting done by your code, rather than preceed it, as would normally be done when overriding a method.
Once you have added one preference to the PreferencesBrain, it is very easy to add more preferences, since this typically involves adding one line of code per preference in each of the overridden methods, as well as a few other simple methods as shown. Remember that it is up to you to query the PreferencesBrain and make use of any new preferences that you add; the GameKit will be blissfully unaware of any changes you make.

Adding panes to the Preferences¼ panel

As you will notice from the class specification sheet, there is support for preferences panels which have multiple panes of controls, like the typical inspector. If you add many preferences, the panel will quickly become very large and crowded. This makes it very difficult for a user to find the preference they want. A better user interface will split the functionality into groups of controls that make sense together. (See Columns for a reasonable example of this.) The PreferencesBrain object currently can deal with three panes. If you only need three panes, then simply use one of the provided .nib file templates which is already set up this way. Note that the StringTable in the .nib actually provides the name for the views as seen on the pop up list at the top of the panel. You can change these strings to anything that makes sense in your application.

If you need more than three panes, they are easy to add. First add an instance variable to point to a view object which contains everything to put on the panel. Typically this will be an unbordered, untitled Box object. (Lay out your controls and then group them together in InterfaceBuilder™; grouping puts a Box object around them.) In Columns, a new view has been added for the options related to the bricks. The instance variable added to the PreferencesBrain object is called **viewBricks**. The only other necessary change is the overriding of the ±**setUpViews** method as follows. You will notice that the name to be placed in the pop up list is taken from the string table, so that localization may be performed easily.

```
- setUpViews
{
    [super setUpViews];
    [viewBricks getFrame:&view[3]]; // keep the frame size so view may be centered properly
    [self addView:viewBricks withName:[strings valueForStringKey:"Bricks"]]; // attach a name to the view
    [self show:[strings valueForStringKey:"Player"]]; // the view that is shown when the panel is first shown
    return self;
}
```

This is all you need to do; you can add as many panes (up to eight) as you need to house all your controls. More than six panes is probably too many, though, since the user will find it difficult to remember where things are. The array **view** holds the sizes of the various views; this array is fixed to allow only eight panes. Slots 0-2 are used by the GameKit views already, so you should use values 3-7. Be sure to use the lowest numbers first or the panes won't be centered the way they should be. (Each pane will require you to add two lines of code as above, and the ±**show:** message at the end is *not* optional±but you can make any other view be the default.)

ªUnfairº preferences

Some of your preferences setting may dramatically affect your game's scoring. They may make it horribly easy to get a high score. In this case, you probably don't want the high score server to be flooded with people who have set everything to be as easy as possible. To aid in this, there is the idea of unfair preferences. Basically, is the preferences setting are deemed to be ªunfairº by the PreferencesBrain, then the score cannot be placed into the network high score panel. It turns out that none of the GameKit preferences are considered unfair, no matter how they are set. So, although this mechanism is in place, unless you write code to trigget it, it will not limit high score entries.

In order to trigger this, simply send a ±**setUnfair** message to the PreferencesBrain. There are two times when it should be triggeed. First is any time that the user changes a Preference on the preferences panel to what you deem to be an unfair setting. Thus, it should be sent from the action methods that you add for a given preference. This makes sure that a game is registered as unfair if a user changes things in the middle of play. The other time to send the ±**setUnfair** message is in an overridden version of the PreferencesBrain ±**startGame** message. Below is a smippet of code which shows an example of how this might be done in your games. This is more or less what you would add to your action messages, too.

```
- startingGame
{
    [super startingGame]; // sets the unfair variable to NO
    if (UNFAIR_PREFERENCES_SETTINGS) [self setUnfair];
    return self;
}
```

This makes sure that a new game registers as unfair if it is started with unfair preferences in effect. Right now, the user is never warned that unfair preferences are in effect, and they are not given the chance to abort the preferences change. In a future implementation, this ability will be added via appropriate alert panels. Unfair preferences do not affect storing local high scores; anybody can store anything in a local high score. Of course, anyone can clear the local table, too, so it's easy to get rid of unfair high scores when you get sick of seeing them.