A library is an addon that provides a set of functions that can be used by other addons. The addon SimpleTimingLib we wrote in Chapter 4 is a library. But there are a few issues involved when working with libraries. Let’s assume you wrote an addon that depends on that library and you want to publish it. You can either include the library in your zip file or tell the users to install the library themselves.

The first solution, including the library, exposes its weakness when a user who already has this library tries to install your package. He will overwrite the version of the library he already had with your version of the library. But what happens if he had a newer version of the library? And what happens if he had an older version, but another addon that uses the library is incompatible with the newer version? This solution almost certainly breaks the user’s interface as soon as there are different versions of your library around. And the user currently cannot even distinguish an older version of a library from the newer one, as it doesn’t have a version number

The other solution, telling the user to download a specific version of the required library from somewhere else, would work perfectly if your addon is designed to be a plugin for another, larger framework or library. Let’s assume you have written a boss mod for a new instance, and you’ve used Deadly Boss Mods for this. You will then publish this boss mod somewhere (such as on curse.com) with a note like “This is a plugin for Deadly Boss Mods; you need to have DBM from http://www.deadlybossmods.com installed in order to use this boss mod.” Your users will certainly understand that a boss mod can be based on a boss mod creation framework and therefore requires DBM.

But there are a lot of mods that make use of ten or more libraries out there. You will see such mods in this chapter. It is quite an effort for the user to download by hand all the libraries your mod requires. And a lot of people out there will fail at this task and send you an email saying “your mod doesn’t work; I’m always getting an error about a missing dependency!”

We will see in this chapter how this problem can be solved in World of Warcraft by using embedded libraries, which are embedded in your addon’s main folder and not as separate addons.

An embedded library is a library that is embedded in your addon, meaning it will be in the same folder as your addon. Such a library usually consists of a single Lua file that is simply added to your TOC file. You can then just pack your addon folder into a zip file and publish it together with all libraries it makes use of

What happens if I have two addons that embed the same library? Will I have this library twice? The answer to that question is both yes and no. You will have the library twice on your hard disk, but each version will be loaded only once. But our libraries currently do not even have version numbers, so two different versions of OeilhaPeiejcHe^ would not be distinguishable; and the library needs to be able to determine whether it is already loaded or not. What would be a good version number for a library

A version number for a library usually consists of two parts: its major version, which is basically the name followed by a normal version number, for example LibFoo-1. A change to this version number indicates a major change to the library. The second part of this version number is its minor version, which is increased with every minor change to the library; this minor version is just represented as a number and could be, for example, 9. Later you will see how we can set these version numbers of a library

Let’s assume we have an embedded library that is called LibFoo. We have the following addons with the following versions of LibFoo installed

Addons are always loaded in alphabetical order if they are not load-on-demand. This means that the first addon that will be loaded is Addon1. This addon then loads version 1.0 of LibFoo. The next addon that is loaded is Addon2, which also embeds version 1.0 of LibFoo. The situation now becomes interesting, as there is already an older version of the same library loaded, as indicated by the minor version numbers

LibFoo-1.0 of Addon2 now detects that there is already an older version of it; you will soon see how we can implement this functionality. It now has to replace this older version of the library with the new one. This new version must be compatible with the old library, as other addons might rely on this old version

The next addon is Addon3, which tries to load yet another minor version of LibFoo-1.0: the minor version 2, which is older than the currently loaded version 3. This version of LibFoo-1.0, now detects that there is already a newer version (3) of itself, and it just does nothing. To instruct the program to do nothing you could use a large if-block around all code in your file, but there is a better solution: using return outside a function. Recall that a Lua file is internally handled as a huge function that is executed once when the addon is loaded. Calling return outside of a function in the file is like calling return in the function that represents the file. This means that this return call cancels loading the file. The following code shows what such a return statement could look like

This stops loading the file if dontLoadMe is true. We will see examples for a library and how we can handle this detection later in this chapter

The last addon that is loaded is Addon4. It loads a new major version of this library. This version contains a lot of changes and is not compatible with older versions of the library; it has the major version LibFoo-1.1 for this reason. It does not replace the old 1.0 version of the library if it is loaded; it is treated as a completely different library because of possible compatibility issues. You will have two versions of this library loaded in your memory from this point on

All addons that depend on LibFoo-1.0, will use LibFoo-1,0, with the minor version 3, and all addons that require LibFoo-1.1 will then use LibFoo-1.1 with the minor version 9. This works for all addons and all libraries that are backward-compatible within one major version.

But what happens if a library is not 100% backward-compatible within a major version? Or what happens if an addon only works with a library because of a bug in that library? A newer version of the library now fixes that bug, and the old addon breaks. This should not happen, and it can actually be quite hard to maintain complete backward compatibility in a new version. The simple solution to this issue is that the major version is increased with every major change that might cause such issues. The minor version is only increased for minor changes that do not affect the functionality of the library

Another disadvantage is that embedded libraries might slow down the loading process of your addons, especially if you have a lot of different versions of a library installed. It might be replaced multiple times while loading the UI

The whole problem with libraries and different versions of them that are shared by multiple addons or programs is not entirely solved. This is not only true for World of Warcraft; every system that makes use of libraries that are shared between many programs has the same problems. For example, in Windows this problem is known as “DLL Hell.” You know what I’m talking about if you’ve ever had a missing DLL (dynamic link library) file under Windows

Embedded libraries are quite a good solution for this problem, but they are far from being perfect

Let’s look at how these embedded libraries can be implemented. There is basically one main library that acts as a helper library for libraries: LibStub

LibStub is an embedded library that manages other embedded libraries. You can always download the latest version of it on its project page on wowace.com: <http://www.wowace.com/projects/libstub/>

It provides three functions that are accessible by addons and libraries: LibStub: NewLibrary(major, minor), LibStub:GetLibrary(major, silent), and LibStub: IterateLibraries(). The last is just a helper function that can be used in the generic bkn loop to iterate over all libraries that are currently loaded

The method NewLibrary creates a new library and registers it. The first argument to this function is a string that acts as unique identifier for the library. This identifier is referred to as the major version of this library. In our previous example with LibFoo, this could be LibFoo-1.0, for LibFoo-1.0, and LibFoo-1.1 for LibFoo-1.1. The second argument to this function is a number that represents the minor version of the library. This number should be increased with every single change to that library to ensure that a newer version always replaces older versions

LibStub then checks whether a library with the same major version already exists. It returns a new empty table if that is not the case. This empty table will then be used by the library as a namespace, meaning that all functions and variables of the library will be put in this table. If there is already such a library loaded with a newer minor version, it returns nil. The library’s loading process should then be canceled, as this means that it is outdated and a newer version is already loaded

The third case is that there is already a library with the same major version loaded but the old minor version is lower than the minor version of the new library. LibStub then returns this old library (which is just a table) followed by the minor version of the old library. The new library can now overwrite all functions and other values in the table that was used by the old version of the library. This process completely replaces the old version of the library with your new version

The last method is LibStub:GetLibrary(major, silent): it returns the requested library followed by its minor version and throws an error message if that library doesn’t exist. The second optional argument can be set to pnqa to suppress that error message

LibStub is quite simple and short, so we can take a look at its source code here. Note that it is also an embedded library, which means it also has to take care of replacing old versions of itself and must be able to cancel the load process. Recall that \_G is the table that holds all global variables; this means that LibStub will be stored in the global variable LibStub in the following code

The major version of LibStub here is just LibStub, and it is used as a global variable to store the library. The minor version here is 2. It checks whether there is already an older version of LibStub loaded and does nothing (by using a large if-block) if this old version exists and the minor version of the old instance of LibStub is greater than or equal to the minor version from this instance

Otherwise, it initializes LibStub with a table that holds all libraries and one that holds the minor versions of all libraries. It will just use the table (namespace) that was used by an older version of LibStub if it is overwriting an old version. The library then defines its three methods

The metatable assignment in the last line just allows you to use LibStub(“libname”) instead of LibStub:GetLibrary(“libname”)

最后一行中的metatable赋值允许你使用LibStub(“libname”)而不是LibStub:GetLibrary(“libname”)

创建我们自己的嵌入库(Creating Our Own Embedded Library)

Let’s update our SimpleTimingLib to use LibStub. We need to set up a new addon that acts as a test for our library to check whether it works. We need this test addon because an embedded library cannot be used without the addon it’s embedded in. The folder SimpleTimingLib, which currently contains the library, will no longer be required after we finish this embedded version. We can then later embed a working version of this library in our other addons that currently make use of SimpleTimingLib. Let’s call our test addon SimpleTimingLibTest

Create a folder called SimpleTimingLibTest for the library and place the following TOC file in it

We are using the field OptionalDeps here. This ensures that our addon is loaded after the addon SimpleTimingLib-1.0, if such an addon exists, allowing us to create stand-alone versions of our library later. Such a stand-alone version has the advantage that it is loaded just once during the loading process. The disadvantage is that the user has to take care of updating the stand-alone version of the library. The solution to this is using an updater for your addons, but there is currently no updater available that can extract the embedded libraries from an addon. The old WowAce updater had this feature, but its successor, the Curse Client, does not provide this functionality yet. It is planned for a future version and might already be available by the time you read this

The next three lines deal with loading the required files. It is common to use the subfolder libs to place all libraries there. Note that the order in which the files are listed here is important: the first thing that must be loaded is LibStub, as it is required by both our library and our test file. The next file that needs to be loaded is our library, as it is required by our test file and depends on LibStub. The last file is our simple script that tests whether the library works properly; it requires SimpleTimingLib-1.0, and LibStub

Many addons embed a lot of libraries, so this list in your TOC file can quickly become quite long and unclear. But using XML, there is a simple way to separate the libraries from the files that belong to your addon. Recall that XML files can embed Lua code by using the element <Script>. This means you can create an XML file with the sole purpose of loading other Lua files. The following code shows an XML file that can be used instead of the two libraryrelated entries in the previous TOC file

Such an XML file is often called embeds.xml. It is then added to the TOC file instead of all the lib/libname/libname.lua entries

The next thing we need is LibStub. Take the file LibStub.lua from the download package and put it in the folder SimpleTimingLibTest\libs\LibStub to load it in your addon

We can now create our library in the file libs\SimpleTimingLib-1.0\SimpleTimingLib-1.0.lua. The first thing we need to do in this file is create a new library, by calling the method NewLibrary of the library LibStub. The following code shows the beginning of our library. It tries to create the new library and cancels the execution if LibStub returns nil there. This happens when the same version or a newer one is already loaded

The beginning of our file looks like this

The local variables MAJOR and MINOR hold the major and minor version of our library; they are visible in the whole file. We can now just copy and paste all the code from the old library, except for the following line:

Our library will then just use the LibStub library (which is also just a table) instead of the old table in the global variable. The declarations of methods like the following work just fine

In fact, the whole library works just fine with LibStub. We can even keep our two global functions we have left in for compatibility reasons. They will simply be overwritten when a new version of SimpleTimingLib is loaded over the current one

Problems with Overwriting Old Versions

But there is also a problem when there is a new minor version that overwrites the old one. We currently keep all tasks in a local table, and there is no way to access this table from the outside. This also means that if our library has to overwrite an old version of itself, it would lose all old tasks. But it is easy to prevent this by storing this table in the library. The problem is this line

Replace it with the following two lines to solve this issue

This stores a reference to the task table in the field tasks of the library. The library will use the old table if there is already an older version of the library loaded or create a new one if it is being loaded for the first time. The local variable tasks also stores a reference to this table; I’m using this table here just so that we do not need to change anything else in the code

But there is a second, similar problem: what happens to the frame if we upgrade the library during runtime? It is also simply stored in a local variable and is not accessible from the outside. Its OnUpdate script handler would continue to be called after the upgrade. This is actually no problem at all. But it comes with a small performance hit if the library is overwritten multiple times while loading the interface, so we want to recycle this frame

with these lines

This creates the frame only if it doesn’t already exist. SimpleTimingLib.frame holds the frame that was used by the old version of the library—the one that is about to be overwritten if there is such an old version.

That’s it. Our library is now fully functional and can be embedded. But another nice feature would be to preserve its ability to act as a stand-alone library. Many people prefer stand-alone libraries over embedded libraries because using a lot of embedded libraries might slow down the loading process of the UI. It is easy to create a library that can be used as a stand-alone library as well as an embedded library. We just need to add a TOC file

Create a TOC file called SimpleTimingLib-1.0.toc in the folder libs\SimpleTimingLib-1.0. This TOC file will be ignored by the game if you use the library as an embedded library, as it is in a subfolder of an addon and not in an addon’s main folder.

Suitable content for this TOC file looks like this

Note that the stand-alone version of this library still needs to embed LibStub. This means you have to copy and paste the folder LibStub from SimpleTimingLibTest\libs to SimpleTimingLibTest\libs\SimpleTimingLib-1.0. This version of LibStub will not be loaded if you use the library as an embedded library, as the TOC file that references it will not be loaded in this case

Our library can now be used as a stand-alone version; just move the folder SimpleTimingLib-1.0, from SimpleTimingLibTest\libs to your addon folder. Thanks to our TOC file, World of Warcraft will recognize this folder as a separate addon and load it together with LibStub

But how can we be sure that the library actually works? We wanted to write a small test addon that makes use of the library. Let’s write that test code now

The place to put our test code is the file SimpleTimingLibTest.lua. The following code creates a new instance of our timing library and creates a simple slash command handler. This slash command is /stltest <time> <lua code> (for SimpleTimingLibTest); it is very similar to +o\_nelp. It takes a number (time) as an argument, which is followed by Lua code to be executed after time seconds have passed.

In the SimpleTimingLibTest file we use the Lua function loadstring(“lua code”), which takes Lua code as a string and creates from it a function that can be executed. This function should always be used very carefully, as there might be a security issue if you execute a string (which might come from another player in the worst case) as code. It is also quite slow as it has to compile the string as Lua code. Because of these issues it should only be used if there is no other way to implement a given functionality

This prints test to your default chat frame after three seconds. You can also test our library as a stand-alone library or you can embed the library in another addon with another minor version and confirm that the upgrading works. You will see that it is no problem and the library continues to work properly. This great flexibility is one of the main advantages of embedded libraries and LibStub

We can now write our own libraries, but it is more interesting to use existing libraries. They allow you to add a lot of exciting functionality to your addon with little work. Let’s look at a famous addon framework that can be included as an embedded library with LibStub: Ace3

Ace3 is a framework that provides a lot of useful functionality for your addon. It is designed in a modular way, which means it consists of many small embeddable libraries. You only have to embed those libraries you need in your addon

This section is not meant to be a reference for the whole Ace framework, so we will only look at a few particularly interesting and important parts of this framework. You can learn more about Ace3 at <http://www.wowace.com/projects/ace3/>

A Simple Hello, World Addon with Ace3

Create a new folder with the name HelloAce in your addon folder and create the file HelloAce.lua with the following contents there:

We are using the XML way of embedding the libraries here, as Ace addons often use many libraries because of the modular structure of Ace3. The main file of an Ace addon is usually called Core.lua, so let’s use this name and create a file with the name Core.lua in our HelloAce folder

We will make use of the following parts of Ace3 in our Hello, World mod: AceAddon-3.0, AceConsole-3.0, AceConfig-3.0, and AceDB-3.0. This means we have to download the latest version of Ace3 from http://www.wowace.com/projects/ace3/. This download contains all Ace3 libraries as well as LibStub, which is required by Ace3. You can either install the whole download package as a complete stand-alone version that bundles all libraries or copy and paste only certain libraries into your addon. We will only use a few libraries, so there is no need to install the whole Ace3 framework as a stand-alone package. We just copy the four subfolders we need into our addon and use them as embedded libraries

Create the folder libs in your HelloAce folder and copy the following folders from the Ace3 download package there: AceAddon-3.0, AceConsole-3.0, AceConfig-3.0, and AceDB-3.0. You also need to copy the folder LibStub, which contains LibStub, into your libs folder, as LibStub is required by the Ace libraries

Libraries can also have dependencies. AceConfig-3.0, depends on CallbackHandler-1.0, and AceGUI-3.0. So we will also need to copy these two libraries into the libs folder of our addon

We can now build our file embeds.xml. If you have peeked into a few folders that are included in Ace3.zip, you might have noticed that some Ace libraries consist of multiple Lua files, for example AceGUI-3.0, comes with many Lua files for the different widgets it can create. But we do not have to add them all by hand to our embeds.xml, as each one of the Ace libraries has its own XML file that includes all Lua and XML files it needs. We just have to add that XML file. Recall that you can use the element Include in an XML file to reference another XML file

We can now build the file embeds.xml, which includes the four Ace libraries we want to use, the dependency LibStub, and the two dependencies of AceConfig-3.0:

Note that the order here is important. The first included library line needs to be LibStub, as it is required by all other libraries. And it is also important to CallbackHandler-1.0, and AceGUI before loading AceConfig.

Our next task is creating the real code of our addon in the file Core.lua

All Ace3 addons follow an object-oriented model. Our addon is an instance of AceAddon-3.0, which means we call a constructor to create the object we will be working with. The constructor is the method NewAddon of the library AceAddon-3.0.

The first argument to that constructor is a string that is used as the identifier of our new addon; we will call our addon just HelloWorld. The following arguments can reference certain other Ace3 libraries, which will be used as what are called mixins. We will add AceConsole-3.0, as a mixin in our example but not AceBD-3,0, or AceConfig-3.0. You will see how to use these two libraries later

A mixin adds the methods of the library that is mixed in to our AceAddon object. This means that these mixins basically just work like the multiple inheritance we saw in Chapter 6. We add AceConsole, as a mixin; it provides a few methods related to command line interfaces like registering slash commands or printing messages.

Our constructor then looks like this and can be added as the first line to Core.lua:

HelloWorld now contains a normal table with an Ace3-specific metatable attached to it. There are a few important callback methods you can place in that table. One of these is OnInitialize; this method will be called after the addon and all its saved variables have been loaded. This is basically an event handler for the event ADDON\_LOADED, with the first argument being the name of your addon. We can, for example, place our code that prints “Hello, World” there

There are two other callback methods: OnEnable and OnDisable; they are called when your addon is enabled or disabled, respectively. You can disable or enable an Ace3 addon by calling its methods Disable and Enable. A disabled Ace3 addon won’t receive any events; you will see how we can add events to Ace3 addons later when we discuss AceEvent-3.0,

You might have asked why we are just using print in OnInitialize in our example. Isn’t there an Ace3 method that deals with printing messages? Yes there is one. The print method is provided by AceConsole-3.0, and is just called Print. It prints a message to your default chat frame with your addon’s identifier as a prefix. This allows us to create a fully functional Hello, World addon that uses Ace3 for everything if we change the line that says print(“Hello, World”) to the following

HelloWorld:Print(“Hello, World”)

The method Print is available only because we added AceConsole-3.0, as a mixin in our constructor. But you don’t have to use these multiple-inheritance-style mixins here; you can also use the methods of AceConsole-3.0, directly, as in the following code

We don’t get our addon’s name as a prefix if we call Print directly, and it is easier to add all methods of AceConsole-3.0, to our AceAddon object. So you might want to add AceConsole-3.0, to almost all of your Ace3 addons as a mixin, as printing messages is always useful

Our Hello, World addon in Ace3 is now complete. But that was too easy, and it seems pointless to use Ace3 here. So let’s build a more complex addon

Do you remember our first Hello, World addon we wrote in Chapter 3? It could do a lot more: we had a slash command that allowed us to add text that was saved in a table under a given key and another slash command allowed us to retrieve the stored text and send it to a chat channel. Let’s build that with Ace3 so you can learn more about how Ace3 works

We will also place all the code of this advanced “Hello, World” addon in the file Core.lua. The constructor for the addon stays the same, so we can start the file with the following line:

We now need to add slash commands. AceConsole-3.0, can be used to create slash commands.

Our original advanced “Hello, World” addon had two slash commands: /hwadd to add text to the database and /hwshow to retrieve saved text and send it to a channel. Let’s call our new slash commands /ahwadd and /ahwshow (ahw for Ace Hello World) to avoid compatibility issues with our old addon

The method RegisterChatCommand(cmd, func) can be used to create a slash command in Ace. The first argument, cmd, is the slash command without the leading slash, and the second argument, func, can either be a function or the name of a method in your addon object that will be called when the user enters this slash command. Add the following lines to Core.lua to register our slash commands

Caution. Do not add a leading slash when registering a slash command with AceConsole-3.0. If you do so, you will create a slash command that needs to be typed with two slashes in the game, and that’s probably not what you expected to get

Ace3 will now try to call the method AddText in the table HelloWorld when you type /ahwadd and ShowText when you type /ahwshow. It is possible to use the same method for multiple commands to add aliases. If you want to have the alias /ahelloworldadd for /ahwadd, you can add the following line

We now need to implement the methods AddText and ShowText. They need to access saved variables. You might have guessed it—we can use AceDB-3.0, to handle saved variables in an Ace addon

Ace-3.0, is not added as a mixin to our addon like AceConsole-3.0, as it does not provide methods that can be used on AceAddon-3.0, objects. But it also follows an object-oriented design model and provides a constructor method that is called to initialize a new saved table. This saved table is usually stored in the field db of our addon object, but you can put it wherever you want. Note that the saved variables of an AceDB-3.0, instance are just normal saved variables; that means they are not available before the ADDON\_LOADED event for our addon has occurred. So we cannot place the creation of our AceDB just anywhere in our code. We have to wait for this event

We previously saw the callback method OnInitialize of the AceAddon-3.0, object; this method is called when our addon is fully initialized and its variables have been loaded. So this is a good place to create or load our AceDB

We create a new saved table by calling the method New of AceBD-3.0, which takes one mandatory argument that is used as a global variable in which the saved table will be stored. The AceDB constructor checks whether there is already a saved table from a previous session and restores it if such a table exists; otherwise, it creates a new, empty table in this global variable. Most Ace3 addons use the identifier of the addon (here HelloWorld) followed by DB for a global saved variable. Let’s follow that convention and name our saved table HelloWorldDB

The second argument to this constructor can optionally be a table that contains the default values for the table. We will use two fields in our saved table: the first one is called channel, and it stores the chat channel the messages will be sent to; the second one is texts, a table that stores our strings

Such an AceDB-3.0, default table basically works like the default table we built in Chapter 7; it’s just easier to use here, as this functionality is provided by the library. We don’t need to worry about metatables and the recursive function to apply that metatable here. We can just use the provided methods of the library. You can read the source code of AceDB-3.0, if you are curious how it works internally

We now have to create the defaults table somewhere and store it in a (preferably local) variable. The following code shows such a table and the call to the AceDB-3.0, constructor in the new OnInitialize method of our addon (you can delete the old Hello, World OnInitialize method):

Note that self is the hidden argument that is added by the colon notation. It will always be the addon object (here HelloWorld) when this method is called by Ace3

You might have also wondered what the purpose of the subtable global is. If you try to add your options directly to the default table, you will get an error message saying that you tried to use an invalid data type. This error message does not refer to a normal Lua data type, it refers to the subtable global, which is one of eight AceDB data types. But what does global mean here, and what are the other seven data types? And will that subtable (or data type) also be present in the resulting AceDB

One might think that we can just use self.db = value to store a value and self.db.key to retrieve it. But AceDB-3.0, provides a more powerful way than accessing HelloWorld.db directly. The library provides eight subtables, which are called the data types (like the type global we saw in the defaults table earlier) in that table and are used for the actual data. These data types are always tables that hold the actual values we want to save

Different data types store different sets of values based on the character the user is currently playing. For example, the data type class stores options based on the character’s class. We could, for example, define a default value in the subtable class and, if the user changes it while playing a priest, she will see this change on all her alts that are also priests. But if she plays a hunter alt, she will still see the default value or her hunter-specific setting

The following data types are provided by AceDB-3.0

This table will be saved on a per-character basis and can be used instead of the TOC metadata SavedVariablesPerCharacter

\_h]oo: This allows you to save variables on a per-class basis. This is useful for mods that heavily depend on the class of the player

This saves options based on the realm and faction of the player; this means that a horde character will see a different factionrealm subtable than an alliance character on the same server

This allows you to manage setting profiles in your addon. You can switch from one profile to another by calling the method addon.db:SetProfile(identifier), where identifier is a string that holds the name of that profile; this method will create a new, empty profile if identifier does not exist yet. Please read the API documentation of AceDB-3.0 on http://old.wowace.com/wiki/AceDB-3.0\_API\_Documentation for more information about managing profiles

We just need the simplest data type of them, global. We don’t want any per-character options and we also don’t need profiles in a simple Hello, World addon. This means we are going to place all of our strings with their identifiers in HelloWorld.db.global.text, and we save the channel the texts are sent to in HelloWorld.db.global.channel.

But we haven’t told the game to actually save a variable yet; we have to add this global variable to our TOC file or AceDB-3.0, won’t be able to save anything. Add this line to your TOC file to complete our persistent table

Don’t forget that you always have to restart your game for it to recognize a change to your TOC file

AceDB-3.0, provides more methods than those we looked at here; for example there are a few methods that are related to profiles, like copying or deleting profiles and listing all available profiles that the user can switch to. But this chapter is not meant to be a reference. Please refer to the AceDB-3.0 API documentation at http://old.wowace.com/wiki/AceDB-3.0\_API\_Documentation if you want to use these advanced features of AceDB in your addon

We haven’t implemented the methods that actually deal with our slash commands yet: AddText and ShowText. They receive, just like normal slash command handlers, the text the user typed in. We can recycle the code from the old non-Ace version of our addon for both methods. AddText looks like this

The regular expression is the same pattern as we used in the old example: a word (everything except for spaces) followed by one or more spaces and one or more arbitrary characters at the end. The first word is our identifier, and the rest of the string minus the separating spaces is the text we want to store. If that pattern matches the entered text, the entry is added to our saved option; otherwise we print a short message that tells the user how to use this addon

HelloWorld:ShowText(msg) is also easy. It reads the option from the saved table and sends it to the chat; otherwise it displays an error message

Our old mod didn’t have an option that stored the channel the message was sent to. The channel was a constant that was stored in a local variable, and there was no way to change it— except for editing the file. Our Ace version of this addon is designed better here: we saved this option, so we now need a way to modify it. We could, of course, just add another slash command handler and parse the string and set the option by hand. But there is a more powerful way to implement options in Ace3 addons: using AceConfig-3.0,

Configuration with AceConfig-3.0

There is an extremely powerful way to describe the options of your addons in Ace3, by using what is called an AceOptions table. This table contains an abstract description of each option your mod has; it is not just a slash command handler. AceConfig-3.0, can create either a slash command from that table or a graphical user interface. You will see how to build various GUIs from it later in this chapter

An AceOptions table is a hierarchic structure. Each option table describes a single option, but certain option types may contain additional option tables. These additional tables can then also be of a type that contains even more subtables

The field tpye in such a table contains a string that describes the type of the option. The simplest type is execute, which just executes a function when the slash command that corresponds to that option is entered. If an addon creates a GUI from such an option table, this will show up as a button. The function to execute has to be stored in the field func of the option table. The field name can be used to set the name of the option and the field desc for a longer description. The name would be shown on the button in a GUI while a description could, for example, be used as tooltip by an addon that builds a configuration GUI from such a table. But how your configuration menu will actually look is up to the addon that interprets the table and builds the GUI; we just describe our option here. Such a table could look like the following

func can also be a string here, and is then interpreted as the name of a method in your addon. Your AceConfig-3.0, doesn’t know which addon this option table is related to, so it doesn’t know in which table the method you are referring to is stored. You also have to add the table representing your AceAddon object (or any other table that holds the method you want to call) in the field handler of the option table.

The fields name and desc can be used by all option tables regardless of the type. But additional types can add additional fields that are required for that type. One particularly interesting type is group, which stands for a group of options that are visually grouped together in a GUI; for example, you might have a border around all options belonging to the same group or put all such options in a separate tab in a tab-based configuration menu. But that is also up to the configuration addon that creates the GUI

The type group needs the field args, which holds an arbitrary number of other option tables. args is a hash table; the key is used as short identifier for the option and the value is an AceOptions table. The identifier is used as sub-slash command in a command-line interface. A subtable of this field args of such a group can also be another table of the type group that also contains a third group, and so on