Let’s get started. You have already seen some Lua examples in Chapter 2, so writing a “Hello, World!” mod will be no problem for you.

But this first example mod will do a bit more than just displaying “Hello, World.” It will provide two slash commands: /hwadd <id> <text> and /hwshow <id>. The former adds text to a table that will be saved, while the latter sends the text that was saved under id to a chat channel. Gold farmers might think that the only possible use of this addon is to spam the trade chat with advertisements, as it allows you to send a lot of text with one macro. Another good use for this mod could be to save long boss or battleground instructions, for example, in a single macro. You can then easily send them to the raid or battleground chat.

To run a script like the one we’re going to create, just type the following slash command in World of Warcraft:

The command /script <text> compiles <text> as a new chunk and executes it.

A chunk is a block of Lua code with its own scope. So if you define a local variable in a /script command, it will not be visible in the next /script command.

You already know the print function, but here it has one small difference from standard Lua: World of Warcraft writes the output to the default chat frame and not to your standard output. So this will show “Hello, World” in your default chat frame.

That’s it? Yes. Basically that is it. But this is more a simple macro-like script, and I promised that you are going to learn how to write addons, so let’s create your first real addon. This example was just to show that Lua in World of Warcraft works just like normal Lua. We can now start creating a Lua script that is loaded as an addon by the game

Let’s create the addon I described at the beginning of the chapter. The first thing we need to decide before we can start creating the mod is where to put our Lua files

You might already know that you need to put addons in the World of Warcraft Interface\AddOns folder to install them. This folder is probably located in C:\Program Files\World of Warcraft\Interface\AddOns on your file system. So let’s create the folder HelloWorld inside that folder. This is the place where we will create all the files for our “Hello, World” mod. But have you ever looked in the AddOns folder? There are many different file extensions. Which types of files can be used by an addon

World of Warcraft provides API functions to handle the following file types: Lua scripts (.lua), XML (a data description language, which you will learn more about in Chapter 5) files (.lua), textures (images that can be used by addons) (.tga and .blp), sounds (.wav and .mp3), fonts (.ttf) and models (3D objects) (.m2). You will be able to use all default files that come with the World of Warcraft client in your addon. This includes the background music, spoken text, textures, models, and much more. And if that is not enough, you can just provide your own sounds, textures, and so on with your addon

The World of Warcraft API provides a virtual file system, which means you can use remote files as if they were on your local disk. This allows you to access all files in your Interface folder and all files in the MPQ archives

MPQ is the archive format used by all Blizzard games. Think of it as something like a zip file. You will see more about MPQ files later in this chapter

This virtual file system is built when you start the client, so the game won’t recognize new files until you restart it. Whenever you add a new file throughout this book, you’ll have to restart the game; simply reloading your interface (by typing\_/script ReloadUI() or simply /reload if you have the addon DevTools installed) won’t work

When working with World of Warcraft, all paths to files are relative to your World of Warcraft folder, and files from MPQ archives will be mapped into that root folder. There is no way to access any file outside the installation folder. So a path name of a sound file could look like this

The first example references a file that is included in your addon. The second references a file in the archive expansion-speech-locale.mpq. (You will learn how to browse and extract these MPQ archives later in this chapter.

Playing a sound file is a simple and good example that uses a path, so let’s try this first. The game provides a simple function that takes the file name of a sound file and plays it: Lh]uOkqj`Beha$beha%. Test it by running this command

e Since you can play any sound file that is in your Interface\Addons, it is very easy to write an in-game MP3 player. You just need to copy all your MP3s into that folder and provide the addon with a list (for example, as a table) of your files because there is no function that returns a list of files in a directory. For example, if you put all your MP3s in the folder Interface\AddOns\MyMusic, you can run the following command to play one in-game:

But the file type we need for our addon is obviously a .lua file, as we want to code Lua now. We need to tell the game somehow to execute the Lua script. We also need to provide the game some additional information about our addon, such as its name. The file type we are going to use for this is called a TOC file. Let’s see what this is

A TOC (table of contents) file contains metadata, or “data about data,” describing an addon along with a list of Lua and XML files to load. There are a few basic attributes like the name of the addon and the version of World of Warcraft it was made for. But it is also possible to define custom attributes and store arbitrary data in a TOC file. So let’s create a TOC file for our “Hello, World!” mod

NNote World of Warcraft reads the TOC files only once while starting. So if you change anything in a .toc file you will have to restart your game. Remember, reloading your interface won’t work

We have already created the folder HelloWorld inside Interface\AddOns, so let’s create our TOC file here. World of Warcraft tries to load a TOC file with the same name as the folder, so we create HelloWorld.toc as shown here

There are two types of commands in this TOC file: lines starting with ## contain metadata. All other lines are files that will be executed to load the addon

We are defining the following metadata for our addon

This tells the game that our addon is written for World of Warcraft 3.1.x. If this version number is lower than the current interface version, World of Warcraft will report that the addon is outdated or even incompatible

This defines the name of our addon, so it will show up as “Hello, World!” in the list of installed addons

An example of localization, this is a special line that will only be executed if you are using a German client. It overwrites the old title we defined before, so our addon will show up as “Hallo, Welt!” in German clients. You can append any locale to any metadata name to localize your addon. A locale consists of a language code followed by a country code, so deDE stands for German-Germany. The locale of English clients is always enUS, even if you have the European version

This will be shown in the tooltip of the addon.

There are many metadata attributes that are recognized by World of Warcraft; Table 3-1 shows them

A comma-separated list of addons that are required for this addon. The addon will load after all dependencies are loaded, and it won’t load if one of its dependencies is missing or disabled

A comma-separated list of addons that will be loaded before the addon if they are installed and enabled

0 (default) or 1. A load-on-demand addon will not be loaded when logging in. You can load the addon by calling LoadAddOn(addon) in another addon

Requires LoadOn. A comma-separated list of addons. The addon will be loaded if one of these addons is loaded

A comma-separated list of addons. This will set LoadOnDemand to 1 if one of the given addons is enabled

Requires LoadOnDemand = 1. A comma-separated list of addons. The addon will be loaded if one of these addons is loaded

A comma-separated list of global variables that will be saved to \WTF\Account\<account name>\SavedVariables\<addon>.lua when logging out

This does basically the same as SavedVariables, but it will save everything on a per-character basis to WTF\Account\<account name>\<server>\<character>\SavedVariables\<addon>.lua

Either enabled or disabled. Determines whether the addon will be enabled or disabled by default

1 or 0. Secure addons are digitally signed by Blizzard. There is no way to get such a signature at the moment, so this attribute can only be used by Blizzard’s addons

You can also use your own metadata by using attributes starting with x=.The API function GetAddOnMetadata(addon, attribute) can be used to get the metadata. Note that this function also works for addons that are not loaded yet. So the TOC file is the place to define when a load-on-demand should be loaded. You don’t have to append the locale when retrieving metadata; this is done automatically

Let’s look at an example. Deadly Boss Mods’ module for the instance “The Eye of Eternity” is a TOC file that makes extensive use of custom metadata

I removed all localized attributes except for the German ones; the real file is longer as each localization requires four additional lines

The file defines all metadata that is needed by DBM to build the list of available mods without loading them. The most important attribute is X-DBM-Mod-LoadZone, which instructs DBM to load the addon when you enter the specified zone. This is a key part of how DBM’ load-on-demand structure works. The real power of metadata is that it is available before the mod is loaded

We have created a TOC file, but we now need to add Lua code to do cool stuff. World of Warcraft provides us a lot of functions that can be used to work with the interface.

You have already seen a few API functions: print, PlaySoundFile, LoadAddOn, and GetAddOnMetadata. All API functions are available in global variables, so they are available everywhere: in Lua files, in XML files (they can embed Lua code, which we will learn more about in Chapter 5), as well as in command line scripts. These functions have self-explanatory names and arguments. You can find a complete list of WoW-specific functions on the WoWWiki page, http://www.wowiki.com/API. This book does not contain a list of all available functions for two reasons: it would be comparatively long as there are many functions and it would be outdated after the next patch.. All Lua functions described in Chapter 2 are also available, unless mentioned otherwise

There are a few different types of functions. Most common are the “real” API functions, so called because they are written in C and made available through Lua’s C API. An example of such a function is PlaySoundFile. There is no Lua file in the default UI that defines this function. Instead, it is directly provided by the underlying interface API, and in most cases it’s not possible to implement this type of function using Lua

You can also find a lot of functions in the Lua files of the default UI; most of these functions start with the prefix UI. An example of this second type of function is UIFrameFadeOut(frame, time) which fades out a frame by reducing its alpha value (the opacity of a frame) over time

Remember that you can always use /script <lua code> to execute Lua code in-game. Use this to play around with some API functions to see how they work. For example, you can test UIFrameFadeOut by calling it with these arguments:

This will fade out your minimap to be invisible after 1 second. But how do you get it back? And what is behind the global variable Minimap?

This brings us to the next type of API function: the object-oriented part of the interface API. All frames are objects; they can be created by calling CreateFrame(frameType) or by using XML files. You will see how to create these frames and the available types in Chapter 5. A frame is basically just a table that holds a userdata value and a lot of methods. Recall that you have to use the colon operator to call methods and that methods are basically just functions stored in tables. The thing to remember here is that frames contain many methods. So the third type of API function is a real API function that is stored in a frame and expects to be called with the colon operator. Let’s call this type widget or frame functions, as they are closely related to graphical user interface elements

For now we’ll just look at the basics of the object-oriented API. You will learn more about frames and the object-oriented model behind them in the next two chapters. Let’s look at a method that is available for all frames: SetAlpha(value). It sets the transparency of a frame (a number between 0 and 1), and all its methods are called by using the colon operator. Try this to get your minimap back

A GetAlpha() method is also available; it returns the current alpha value. You can find a complete list of all available methods and frame types in Appendix A

But back to our Hello World mod. We have an addon that has the required metadata to be recognized by World of Warcraft, but it’s just a TOC file. Now we need to add Lua code to our file to do something with those API functions. We already have the line HelloWorld.lua in our TOC file. Whenever you try to load HelloWorld, World of Warcraft tries to find this file in the folder HelloWorld, so let’s create this file

The game will now execute the Lua code in that file when you log in. So writing print(“Hello, World!”) in that file will display “Hello, World!” in your chat frame after logging in. But that’s useless to us; we need something interactive. Let’s create some slash commands

I told you earlier that you need to restart the game after adding a new file. It turns out you don’t need to restart it after changes to an existing file. Just type

Creating custom slash commands is easy. There is a table that stores all slash command handler functions, available as SlashCmdList in the global namespace. The slash command handler is then associated with a specific slash command like /hwadd. This function is then called every time the user enters this slash command. The function receives the string after the actual slash command. So typing /hwadd test string will call our handler with the argument “test string”

To create the mod I discussed at the start of the chapter, we need two custom slash commands: one that stores a new string and another that retrieves a string and sends it to the chat. There are two steps to creating a slash command. First, we have to choose a unique identifier for our slash command handler that will be used in the next step as a key in the table SlashCmdList. Our identifier for /hwadd is HELLO\_WORLD\_ADD, and our function will be the value associated with this key. We also need to define the actual slash commands we want to use and associate them with this identifier. This is also done using global variables, SLASH\_identifier1 to SLASH\_identifierN, to associate more than one slash command with a command handler. In our case we use SLASH\_HELLO\_WORLD\_ADD1 and SLASH\_HELLO\_WORLD\_ADD2 to associate two slash commands with our command handler.

The second step is to store the slash command handler in the table SlashCmdList by using our identifier as a key. This handler is a function that takes one argument. Remember that the slash commands we want to use are /hwadd <id> <text> and /hwshow <id>. This means we need two slash command handlers. The code is pretty simple

If you haven’t created the file HelloWorld.lua yet, do it now. Then add the following code

Let’s go through this file line by line. At the beginning we initialize two variables. The global variable DahhkSknh`[Patp will hold a table with our strings that are stored when the user types /hwadd. The local variable channel will contain the chat channel we want to use. Then we create our first slash command with the ID HELLO\_WORLD\_ADD. The function with the key HELLO\_WORLD\_ADD in SlashCmdList will be called every time we enter /hwadd <text> or /helloworldadd\_<text> with msg = <text>

The first line of the HELLO\_WORLD\_ADD handler parses the user input. This is done by using regular expressions; if you haven’t already read the section about them in Chapter 2, read it now. First, %S is the complement of %s; since %s represents all whitespace characters, %S stands for all non-whitespace characters. The code + means “at least one.” The brackets around the expression define it as a capture, so the matching string will be the first return value of the i]p\_d method. So id will be the substring of msg beginning with the first non-whitespace character and ending with the first whitespace. The second expression stands for the separating space between the two arguments, and the third expression (the second capture) is the rest of the string

This pattern will work for all imaginable user inputs like “foo bar”. The i]p\_d method returns nil if the pattern does not match the given string, so our next task is to test whether id and text are not nil. A valid input will set id and text, so we can use the lowercase version of id as key in HelloWorld\_Text and set the value of this field to the given text. If the pattern didn’t match, the input the function fails silently; but it’s easy to add an error message for invalid inputs

The second slash command handler is quite simple: it takes the user input, converts it to lowercase, and uses it as a key in HelloWorld. If there was some text to use as a key (in other words, if the user provided an ID), we call the function SendChatMessage(msg, channel, language, target) to send a chat message with the previously stored text. Only the first argument is required. The channel argument can be one of the following values: SAY (default value), YELL, EMOTE, WHISPER, PARTY, RAID, BATTLEGROUND, RAID WARNING, GUILD or OFFICER. The language argument is the language to use; nil will use the character’s default language. Finally, target is needed only for channel = WHISPER

One problem remains: the table is not persistent, meaning that your texts will be gone when you log out. So we need to save them somewhere. We can use the metadata attribute you’ve already seen, SavedVariables, to save a value. Just add this line to our TOC file

(Don’t forget to restart your game after updating the TOC file.) What exactly does this line do

When you log in, World of Warcraft loads your mod, which then initializes HelloWorld\_Text with a new, empty table. After the mod is loaded, the game tries to load the saved variables for this mod. If HelloWorld\_Text has not been saved yet, because you are using the mod for the first time, nothing happens. The global variable HelloWorld\_Text will not be set to nil in this case. If HelloWorld\_Text is saved, it will set the global variable to the saved value, dropping the old empty table. What’s important to note here is that the initialization of a saved variable while loading a mod defines a default value. When you log out, the game gets the value of HelloWorld\_Text and saves it if it’s not nil

But these default values can also cause problems. Imagine you have an addon that stores its options in a table. The default value could look like this

If a new user of this addon installs this version, there’s no problem. The user has no saved variables, so she gets all four options. But if you update your addon, World of Warcraft overwrites the global variable MyAddOn Options, dropping the new default option table. Option4 will be nil because your old saved value does not have this field. This will cause error messages like “attempt to perform arithmetic on a nil value” if the mod assumes that Option4 is a number and tries to add it to another value. It is always important to keep your mod updatable, as you might want to add additional features in a later version. Future patches also might break your mod if Blizzard decides to change API functions. You will then have to fix your mod and release a new version of it

There are a few possible solutions for this issue. One is to rename the global variable, but this method has a lot of disadvantages: you have to replace all occurrences of the variable, the users will lose their old settings every time you add a new option, and it is poor coding style if you have to rename all occurrences of this variable. A solution that might be suitable for small addons is to add a simple check whether an option that should be set is nil. You can then set the default value in the if block. But this can quickly become confusing, and it’s also not the best coding style

The best solution is to keep a separate table with the default values. The saved table will then store only those options that were changed by the user. We will see a smart way to implement this later in this book. This solution also has the benefit that you can always access the default value of an option to reset it. We will be using this solution for all examples that have saved options in this book

We have created our first World of Warcraft addon, and I have mentioned that the default UI is also written like an addon. So you might wonder how Blizzard wrote their code. To conclude the chapter, let’s take a look

The default UI is stored in MPQ archives, and there are several ways to get that code. The simplest way is to use Blizzard’s Interface AddOn Kit, introduced in Chapter 1, to extract it. It’s easy to use and extracts all Lua, XML, and TOC files as well as all textures used by the default UI

You can also use an MPQ viewer or editor like MPQ Editor, also introduced in Chapter 1, to browse all MPQ archives. These MPQ editors are more powerful than Blizzard’s interface tool. Such an editor can be used get the paths of available sound files, models, and textures, and to extract some Lua and XML files that Blizzard’s tool does not extract (such as the code used by the login screen)

NAttention MPQ Editor is a very powerful tool that also allows you to modify MPQ archives. But don’t do this. It can break your game, and it’s against the EULA

You can also use the web site http://wowcompares.com, which has detailed diffs for all patches

A diff shows all differences between two versions of a file; this allows you to spot changes in patches

The code from the MPQ archives can be helpful if you are unsure how an API function works or to figure out changes after a patch, since Blizzard provides little or no documentation. You might notice a few strange things; for example, the code uses semicolons everywhere. Most addon authors don’t use semicolons at all, as semicolons are not necessary in Lua in most cases

The code is easily readable and has a few helpful comments. Some comments are definitely worth reading

At this point you know how to work with Lua, you know the basics about the World of Warcraft API, and you can create addons that are recognized and loaded by World of Warcraft. So it’s time to create your first useful addon in the next chapter