Beginning Lua with World of Warcraft Addons

Paul Emmerich

Apress®

Beginning Lua with World of Warcraft Addons

ISBN-13 (pbk): 978-1-4302-2371-9

ISBN-13 (electronic): 978-1-4302-2372-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Joohn Choe, Matthew Moodie
Technical Reviewer: Chris Lindboe
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Jim Compton
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositors: Patrick Cunningham and Dina Quan
Proofreader: Lisa Hamilton
Indexer: Carol Burbo
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 4

Working with Game Events

This chapter is about event handlers and how to use them. An *event handler* is a function that is called by the game every time a specific event occurs. World of Warcraft provides a variety of events, ranging from clicks on UI elements to combat-related events. So what we need to do is tell the game that we are interested in an event and provide an event-handler function that should be called when the event occurs. Such a function is also called a *callback function*.

We will build three example mods in this chapter, the first of which will provide mouse-over tooltips for item, quest, and spell links in the chat frame. Figure 4-1 shows the mod we are going to build.

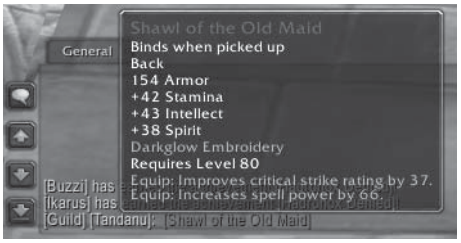

Figure 4-1. *The mod ChatlinkTooltips in action*

The next example mod will be something more abstract, a timing library. However, it provides important functions we are going to need for other examples in the following chapters. This timing library will allow us to schedule a function to be called after a given period of time.

The third mod we will build in this chapter is a fully featured DKP auctioneer that can be used to auction items during your raids. It will make use of the timing library.

For these mods we need to handle certain events, such as the event that occurs when you hover the pointer over a link in a frame. Another event we will need to handle occurs when someone sends you a chat message. However, we need to discuss frames before we can start working with events. What is a frame, and how is it related to events? You need to learn something about frames and events before we start creating our mods.

Using Frames as Event Listeners

A frame is a UI element, such as a button or a text box, but we are not discussing the creation of graphical user interface elements yet. So we will use the very basic frame type that is called simply `Frame` in the following examples. Such a frame is not visible by default, and we are going to use just one feature of this frame: it can be used as an *event listener*, which means it can register events and corresponding handlers to deal with those events. The following line shows how to create such a frame, by using the API function `CreateFrame`:

```
local myFrame = CreateFrame("Frame")
```

You've already seen this function in the last chapter; it creates and returns a frame of the requested type. But what can we do with this frame now? The created `myFrame` table (which is also called an *object*) is basically just an identifier that can be passed to other API functions. These functions (also called *methods*) are always available in the created table, and we must call them with the colon operator.

Different frame types have different methods; a button, for example, has a method that sets the text on the button. A basic frame like we are using here does not have such a method. A list of all available methods for all frame types can be found in Appendix A. We are going to use only a few methods here: `SetScript`, `GetScript`, `HookScript`, and `HasScript`. These methods deal with event handlers, which are also called *script handlers* in World of Warcraft (events are also called scripts). I will use the term "script" from now on to refer to such an event and "script handler" to refer to the event handler callback functions.

Different frame types may have different scripts. For example, a button has the script `OnClick`; other kinds of frames don't have this, because it's very common to handle a button click but not a click on a normal frame. There are other ways to interact with the mouse in frames; there is for example the script handler `OnDragStart` that is invoked when the user tries to drag the frame. We need the following methods.

```
frame:SetScript(script, func)
```

This function registers a script handler that will be called when *`script`* is called. A script is always identified by a string that starts with `On`. The second argument can be either a function or `nil` to remove the script handler. `SetScript` will overwrite any existing script handler for the given script.

The first argument that is passed to that script handler (which is just a function) when the script occurs is always the frame that is responsible for the script. This allows us to assign a function to more than one frame. All following arguments depend on the script type.

```
frame:GetScript(script)
```

This function can be used to get the function assigned to a specific script. It returns `nil` if there is no handler for this script.

```
frame:HookScript(script, func)
```

This function works only if the frame already has a script handler for this script. It will add *`func`* as an additional callback function to *`script`*. Removing the original script handler will also remove all hooks.

```
frame:HasScript(script)
```

This function returns `1` if *`script`* is a valid script for the frame's type, `nil` otherwise. Note that this function cannot be used to check if a script handler for a script is set. You have to use `GetScript` for this.

Now, what scripts could we use now to play around with? If you look at the list of available scripts for normal frames that can be found in Appendix A, you will notice that almost all of them are related to graphical user interface elements. For example, there are scripts like `OnShow`, `OnHide`, `OnDragStart`, and `OnDragStop`. These scripts occur when a frame is used as a window and you open, close, or drag it.

But back to the example tooltips mod I promised at the beginning of the chapter. We don't need to create a frame here as we can use the already existing chat frame. The type of chat frame is `ScrollingMessageFrame`. It provides the script handlers `OnHyperlinkEnter` and `OnHyperlinkLeave`, which are invoked when your mouse hovers over or leaves a link in this frame. A link is any clickable chunk of text in a message frame. So, besides obvious links like items, chat channels and names of players in front of a message are also links. So let's get started building this addon.

Creating ChatlinkTooltips

The first thing we need is a folder in `Interface\AddOns` called `ChatlinkTooltips`. The second step in creating an addon is always to create its TOC file with the name `ChatlinkTooltips.toc`. A TOC file for such an addon could look like this:

```
## Interface: 30100
## Title: Chatlink Tooltips
ChatlinkTooltips.lua
```

Feel free to add additional metadata like a description by using the attribute `Notes`. The next task is to create the file that does the hard work, `ChatlinkTooltips.lua`.

Chat Frames and Their Script Handlers

We need to get references to all the chat frames we have. We need to work with their script handlers, as we need the scripts that occur when the user hovers the mouse pointer over a link in the chat. We can use the `ChatFrame`*`X`* global variables for this. *`X`* is the number of the chat frame we want, so your first chat frame is `ChatFrame1`, the second one is `ChatFrame2`, and so on. Your default chat frame (the one with the edit box to send a message) is also stored in the global variable `DEFAULT_CHAT_FRAME`. Another useful global variable is `NUM_CHAT_WINDOWS`, which is the maximum number of chat frames you can have. This is by default seven, but addons or future patches might change it.

For this addon we need to create a loop that iterates over all chat frames and sets functions that show or hide tooltips as script handlers for link-related scripts. One added complication in this example is that we need to access each `ChatFrame`*`X`* global variable dynamically and in sequence (in other words, we need to access `ChatFrame1` in the first iteration of the loop, `ChatFrame2` in the second iteration of the loop, and so on). There is a function provided by World of Warcraft that returns the value of a global variable with a given name: `getglobal(`*`name`*`)`. The argument *`name`* is a string here, so we can build the name dynamically during each iteration.

The following code shows such a loop and two dummy functions (`showTooltip` and `hideTooltip`) that are used as script handlers. These functions simply print their arguments to the chat frame, so you can see what's going on there. I also created a helper function that tries to hook a script handler if one already exists; otherwise, it sets a new script handler. It is useful to move this functionality into a small function, as we need to do this task for our two script handlers `OnHyperLinkEnter` and `OnHyperLinkLeave`. Here is the code:

```
local function showTooltip(...)
   print(...)
end
local function hideTooltip(...)
   print(...)
end

local function setOrHookHandler(frame, script, func)
   if frame:GetScript(script) then -- check if it already has a script handler...
      frame:HookScript(script, func) -- ...and hook it
   else
      frame:SetScript(script, func) -- set our function as script handler otherwise
   end
end

for i = 1, NUM_CHAT_WINDOWS do
   local frame = getglobal("ChatFrame"..i) -- copy a reference
   if frame then -- make sure that the frame exists
      setOrHookHandler(frame, "OnHyperLinkEnter",  showTooltip)
      setOrHookHandler(frame, "OnHyperLinkLeave",  hideTooltip)
   end
end
```

Let's take a look at the body of the loop. The first line just copies a reference to the frame that this iteration is dealing with into a local variable called `frame`, so we don't have to write `getglobal("ChatFrame"..i)` all the time. The code then checks if the frame really exists; this check should actually never fail, but you can never know which addons your users have installed that might mess with the chat frames. It then calls the helper function `setOrHookHandler`, which sets our script handler if the frame does not have one for this script yet. Otherwise, it hooks our script handler to preserve any existing ones. Note that the default user interface does not use these two script handlers. This check is just to maintain compatibility with chat addons that might already have similar functionality.

Item Links

You can now load this addon in the game, but don't forget to restart the game after creating the files. Then try to hover over a link in your chat frame; you will see a message that looks like the following when your cursor enters or leaves a link:

```
table: 168BAC00 item:40449:3832:3487:3472:0:0:0:0:80 [Valorous Robe of Faith]
```

The first argument is the frame responsible for the script handler:

```
table: 168BAC00
```

You will see what we can do with this frame when we add the tooltip later.

The second argument is the data of the link:

```
item:40449:3832:3487:3472:0:0:0:0:80
```

The data of a link can be any string, and links used by the default UI always consist of separate small strings separated with colons. We will create and use our own links in Chapter 9.

The substring up to the first colon is the type of the link, so this is an `item` link. The following number is the ID (`40449`) of the item, and the next number is the ID (`3832`) of the enchant on the item. The next four numbers (`3487`, `3472`, `0`, `0`) are the item IDs of the socketed jewels. The next number (`0`) is the suffix ID of uncommon items. It is followed by the unique ID (`0`) of an item, but not all items have a unique ID, and you can't get any useful information from such an ID. The last one (`80`) is the level of the player that sent the link, which is used for items that scale with your level.

The third argument is the whole clickable link as it is displayed in the chat frame.

```
[Valorous Robe of Faith]
```

■ **Tip** The item IDs are also used by most item database web sites, like `http://www.wowhead.com`. This allows you to build URLs from item links, for example you can use one of the jewel IDs from the links above to get the link `http://www.wowhead.com/?item=3487`

Using Tooltips

We now have the script and the data we need. The only task that remains is showing the tooltip. We will use the frame `GameTooltip` for this, because it is the default tooltip used by the game. It provides the method `SetHyperlink(`*`linkData`*`)`, which takes the data from a link (the long string with a lot of numbers separated by colons) and sets the content of the tooltip to the corresponding item, spell, or achievement. This method will generate an error message if you provide a link that does not have an associated tooltip, like a player or channel link. Therefore, we need to check the link before passing it to this method. The following code shows the script handlers we need for our mod. These two functions will replace our dummy functions. These functions need to be placed before the loop, as the local variables that hold these functions need to be visible in the loop.

```
local function showTooltip(self, linkData)
   local linkType = string.split(":", linkData)
   if linkType == "item"
   or linkType == "spell"
   or linkType == "enchant"
   or linkType == "quest"
```

```
   or linkType == "talent"
   or linkType == "glyph"
   or linkType == "unit"
   or linkType == "achievement" then
      GameTooltip:SetOwner(self, "ANCHOR_CURSOR")
      GameTooltip:SetHyperlink(linkData)
      GameTooltip:Show()
   end
end

local function hideTooltip()
   GameTooltip:Hide()
end
```

Let's go through this line by line. The function `showTooltip` uses `string.split` to get the first part, which indicates the type of the link. The long expression in the `if` statement checks whether it is a link that can be passed to `SetHyperlink`. But before we call `SetHyperlink`, we need to call the method `SetOwner`. A tooltip is always bound to a frame that owns the tooltip, and hiding the owner will also hide the tooltip. So we set the owner of the tooltip to the current chat frame here. Recall that the first argument that is passed to the script handler (in this case, `self`) is the frame that initiated the call., so `self` is the current chat frame. The second argument to this method is the anchor of the tooltip. We use `ANCHOR_CURSOR` here, which causes the tooltip to stick to your cursor. You will learn more about anchors and frame positioning in the next chapter, when we discuss the creation of frames. The code then calls the `Show` method, which shows the tooltip. The function to hide the tooltip is pretty simple. It just calls the method `Hide` on the tooltip.

The mod is now ready to be used. Note that it also works for spells and names in the combat log, as the combat log is also just a chat frame.

You know how to use script handlers now, but all the script handlers we've worked with thus far are somehow related to the graphical part of the user interface. The most interesting script handlers are not related to the GUI but instead handle the `OnEvent` script. I previously mentioned that scripts are events, so that might sound strange—a script handler for other events? Yes, this script handler has its own events. All events related to gameplaying will invoke the `OnEvent` script handler and pass an event as a string to it. So this is worth a closer look.

OnEvent

We need to discuss another method of the frame object before we can use the `OnEvent` script handler: `frame:RegisterEvent(`*`event`*`)`. This method is used to register a gameplay-related event by this frame. Only events registered with this frame are passed to its `OnEvent` script handler.

In this context I'll use the terms *script* and *script handler* to refer to GUI-related events and event handlers like those we saw in the last section. The terms *event* and *event handler* will refer to the `OnEvent` script handler. These event handlers deal with gameplay events like chat messages, the casting of spells (by you or other players), or entering instances.

Event Handler Basics

The distinction between GUI-related events and gameplay-related events might be confusing just now, but you will get used to it. Let's look at an example that uses the event CHAT_MSG_WHISPER to demonstrate an event handler. This event occurs every time you receive a whisper message. Note that you can whisper to yourself for testing purposes.

```
local frame = CreateFrame("Frame")

local function myEventHandler(self, event, msg, sender)
   print(event, sender, msg)
end

frame:RegisterEvent("CHAT_MSG_WHISPER")
frame:SetScript("OnEvent", myEventHandler)
```

This will show CHAT_MSG_WHISPER followed by the name of the player who sent the whisper and the text of the message in your chat frame every time you receive a whisper message.

The first argument that is received by an event handler is always the frame the handler is attached to (named self in this case). The second argument is the name of the event that occurred (event here). All following arguments depend on the event.

The event handler receives 13 arguments on CHAT_MSG_WHISPER events. You often refer to the third argument as arg1, or "the first argument of the event," as the actual first two arguments are fixed. The most commonly used arguments of the event are the first one (the message), the second (the sender), and the sixth (sender status, such as "AFK" or "DND"). The last (the 13th) argument is a counter for all received chat messages in this session. The meanings of these four arguments are the same for all chat events, including messages in the raid (CHAT_MSG_RAID) or guild (CHAT_MSG_GUILD) chat.

But what about the other nine arguments? Some of them are used in other chat events like CHAT_MSG_CHANNEL. They refer to the chat channel or to the affected player of a channel action like a kick or a ban. You will rarely need to use arguments other than the player who sent the message and the text of the message, so you do not have to worry about all these arguments.

Some arguments are always an empty string or 0. Their exact purpose is unknown; they might have been used for something in an older version. But removing them from the list would shift down the following arguments, breaking compatibility with existing event handlers. So don't wonder if you encounter arguments that don't seem to make sense. You will see how we can quickly extract one of these arguments by using varargs in the next section.

Event Handlers for Multiple Events

Most addons have only one event handler function and one frame that registers all events and passes them to this callback function. So there is exactly one function that has to take care of all events that your addon needs to be informed about. This function might look like the following example:

```
local function myEventHandler(self, event, ...)
   if event == "CHAT_MSG_WHISPER" then
      -- we received a whisper, do something with it here
      local msg, sender = ...
```

```
      print(sender.." wrote "..msg)
   elseif event == "ZONE_CHANGED_NEW_AREA" then
      -- we are in a new zone, do something different
   elseif event == "PARTY_MEMBERS_CHANGED" then
      -- someone joined or left our group, deal with it
   elseif event == "..." then
      -- to be continued...
   end
end
```

Recall the meaning of the three dots in the head of this function: "there are an unknown number of additional arguments." As discussed in Chapter 2, these three dots are called a vararg, and you can use them everywhere that Lua expects a list of values. We can get normal variables from this vararg by writing code like line 4 in the example:

```
local msg, sender = ...
```

It is also possible to cut off the beginning of the vararg by using `select`. For example, if you want to get the sixth argument of a `CHAT_MSG_WHISPER` event (the status of the sender) from the vararg, you don't have to create five dummy variables to get to it. You can just use the following code.

```
local status = select(6, ...)
```

But with that approach, the whole function can quickly become huge and unclear. It is not uncommon to write addons that listen to 20 or more events. So the next step you might take is to split up this function by reducing each part of the `if` block to a single function call. You then have a lot of small functions, each of them dedicated to one event. It could look like this:

```
local function onWhisper(msg, sender)
   -- deal with whispers
end

local function onNewZone()
   -- we are in a new zone
end
-- ... etc

local function myEventHandler(self, event, ...)
   if event == "CHAT_MSG_WHISPER" then
      onWhisper(...)
   elseif event == "ZONE_CHANGED_NEW_AREA" then
      onNewZone(...)
   elseif event == "..." then
      -- etc
   end
end
```

But there is an even smarter way to do this. You can create a table and use the event as a key and store the related handler under this key. This makes splitting up the events really easy, and you don't need the huge `if` construct. Our event handler would then look like this:

```
local eventHandlers = {}
function eventHandlers.CHAT_MSG_WHISPER(msg, sender)
end

function eventHandlers.ZONE_CHANGED_NEW_AREA()
end

local function myEventHandler(self, event, ...)
   return eventHandlers[event](...)
end
```

Another advantage of this solution is that registering a new event with a new event handler at any given time is really easy. Just call the `RegisterEvent` method of your frame and add the function to the table. This solution can also be faster if you have a lot of events. You will see examples of this and more on performance in Chapter 13.

Another script handler that is also worth a closer look is the `OnUpdate` handler.

Using OnUpdate to Build a Timing Library

The `OnUpdate` script handler is always executed before the game renders the user interface. This means that if your game runs at 25 frames per second, this script handler will be executed 25 times per second. The function receives one additional argument, the time elapsed since the last call to it. So we are going to use this script handler to build an addon that provides a function that schedules another function to be called after a certain time. Creating the TOC file is up to you this time.

The Scheduler

What we need is a function that takes a time (in seconds), another function, and a list of arguments. The addon will then call the provided function with those arguments after the specified time. This functionality is needed quite often; we will make use of this addon in later examples. The following code shows this schedule function and a table that stores all scheduled tasks:

```
local tasks = {}

function SimpleTimingLib_Schedule(time, func, ...)
   local t = {...}
   t.func = func
   t.time = GetTime() + time
   table.insert(tasks, t)
end
```

You know that the vararg parameter can be used everywhere Lua expects a list of values, so `{...}` will create a new table and fill its array part with the values stored in vararg. The array part will thus store all arguments that will be passed to the function. The next line stores the

function in the hash table part under the key `func`. The exact time at which the function is executed is stored under the key `time`. The API function `GetTime()` returns the current system uptime with millisecond precision. Adding `time` to this value gives us the moment in which the task should be executed. The `OnUpdate` handler will then check this entry against the current value of `GetTime()`. The last line of the function inserts our new task into the table `tasks`, which stores all tasks.

We now need to write the `OnUpdate` handler that executes a task when it's due. This function iterates over all tasks and checks if a particular task's time is up. We will use the function `unpack` to get the arguments from the table. This function returns all values stored in the array part of a table. Here's the handler code:

```
local function onUpdate()
   for i = #tasks, 1, -1 do
      local val = tasks[i]
      if val.time <= GetTime() then
         table.remove(tasks, i)
         val.func(unpack(val))
      end
   end
end

local frame = CreateFrame("Frame")
frame:SetScript("OnUpdate", onUpdate)
```

Note that the code does not use `ipairs` to traverse the table. Instead it traverses the table backward, by using the numeric `for` loop with a start value of `#tasks`, an end value of 1, and a step of –1. We then store the current entry in the table in a local variable in the loop. Traversing the table forward could cause problems here, as we are using `table.remove` to remove elements while traversing it. This function will move elements after the removed element down by one. So if you remove the current element while traversing over the table with `ipairs`, the next element will be skipped.

The loop saves a reference to the current element and checks whether it should be executed now. If this is the case, the task is removed from the list and executed. It is important to remove it before executing it, as the execution might cause an error. An error cancels the running Lua script. So if you remove it after you execute it and an error occurs, the next time the `OnUpdate` handler is invoked this entry will still be in the table. In this case, the script will try again to execute it; it will fail again, and you'll get the same error over and over.

Another question that might come up is what happens if you have a `nil` value in your arguments? Consider the following call:

```
SimpleTimingLib_Schedule(1, print, 1, nil, 3)
```

You can use `/script` to execute this in-game. The result you get is that 1 is printed to your chat frame after 1 second. The second and third arguments of the scheduled task are ignored by our code because `unpack` only works on the array part of the table, and the third entry will be in the hash table part. The operation is actually slightly more complicated than this because the array part can also contain `nil` values, but only under certain circumstances. But this topic is beyond the scope of this chapter; we will take a closer look at tables in Chapter 13. For now: don't use `nil` in scheduled functions.

Another function that can be useful for such a library is a function that cancels a task; let's call it `SimpleTimingLib_Unschedule`.

The Unscheduler

What we have to do now is traverse the table and remove one or more tasks. The function will take the same arguments as the scheduler, except for the time. It will then remove all tasks that match the given criteria:

```
function SimpleTimingLib_Unschedule(func, ...)
   for i = #tasks, 1, -1 do
      local val = tasks[i]
      if val.func == func then
         local matches = true
         for i = 1, select("#", ...) do
            if select(i, ...) ~= val[i] then
               matches = false
               break
            end
         end
         if matches then
            table.remove(tasks, i)
         end
      end
   end
end
```

Recall that `select("#", ...)` returns the number of arguments stored in the vararg.

The function iterates over the table and uses a second loop to determine if the arguments of the task match our arguments. Note that this loop also matches tasks with additional arguments that were not passed to `SimpleTimingLib_Unschedule` . This allows us to remove more than one scheduled task at once. Let's test this with the following code:

```
SimpleTimingLib_Schedule(1, print, "Foo", 1, 2, 3)
SimpleTimingLib_Schedule(1, print, "Foo", 4, 5, 6)
SimpleTimingLib_Schedule(1, print, "Bar", 7, 8, 9)
SimpleTimingLib_Unschedule(print, "Foo")
```

This will print `"Bar 7 8 9"` after one second as the call to the `Unschedule` function matches the first two calls to the `Schedule` function.

■ **Tip** You can use TinyPad to execute these code fragments in-game.

But this unscheduler comes with a bug in combination with the scheduler. Did you spot it? Look at the following code:

```
local function buggy()
   SimpleTimingLib_Unschedule(print, "Bar")
end

SimpleTimingLib_Schedule(2, print, "Bar")
SimpleTimingLib_Schedule(1, buggy)
```

The table `tasks` will have two entries after this code is executed; the second one is a task to call `buggy`. After one second has passed, the scheduler executes `buggy` and removes the other entry. The table is now empty, as the task that calls `buggy` has already been removed by the `OnUpdate` script handler just before the task was executed. Lua now returns into the loop in the `OnUpdate` handler and the body of the loop will be executed a second time with `i = 1`. It cannot know that the table is empty by now, and it will execute the line `local val = tasks[i]`, setting `val` to `nil`. The next line then tries to get `val.time` and causes an error message, as trying to index a `nil` value is not possible.

Fixing this is easy: we just have to change the line that accesses `val.time` to this:

```
if val and val.time <= GetTime() then
```

This demonstrates how it is important to check that your variables are of the type you expect them to be. We expected `val` to be a table here, but it was `nil` in the previous situation because of a tricky call to the `Unschedule` function.

OnUpdate and Performance

`OnUpdate` can be a very helpful script, but it can also be a real performance-killer. So you have to always be careful when using it. Always keep in mind that it is executed every single frame.

You also should ask yourself if it is really necessary to execute a certain task every frame. If you think something has to be executed as often as possible, it is often sufficient to execute something every few frames or every 0.5 seconds. A common way to restrict how often an `OnUpdate` handler is called is to wrap it with the function I'll present next.

As mentioned previously, the second argument that is passed to the `OnUpdate` script handler is the time that elapsed since the last call to it. We can use this to limit the calls per second.

```
local function onUpdate(self, elapsed)
   -- the update task is here
end

local frame = CreateFrame("Frame")
local e = 0
frame:SetScript("OnUpdate", function(self, elapsed)
   e = e + elapsed
   if e >= 0.5 then
      e = 0
      return onUpdate(self, elapsed)
   end
end)
```

This will invoke the function `onUpdate` twice per second. If you want to invoke it every few frames, you can use the following wrapper function:

```
local frame = CreateFrame("Frame")
local counter = 0
frame:SetScript("OnUpdate", function(...)
   counter = counter + 1
   if counter % 5 == 0 then
      return onUpdate(self, elapsed)
   end
end
```

This will call the function `onUpdate` every five frames.

Using the Timing Library for a DKP Mod

The chapter's final example will use the timing library we just built together with event handlers to build a DKP auctioneer. The addon will be controlled through a simple command-line interface (slash commands). Another useful feature is to allow other players in your raid or guild, such as your officers or raid leaders, to start or stop an auction for an item.

The mod will follow the rules of the DKP system that is used by my guild. All bids that are placed during an auction are hidden, and the player with the highest bid will have to pay the second highest bid plus one DKP at the end. If only one player bids on an item, he pays the lowest possible bid as defined by your DKP rules. If more than one player bids the same amount and this amount is the highest bid, they must roll for the item. You can adjust the addon to match your DKP rules.

Variables and Options

After an auction is started, the addon will post the item and the remaining time to a configurable chat channel. The addon will then accept bids via whispers and post the highest bidders to the chat channel when the time is up.

So let's start with this mod. Building the TOC file should be an easy task for you. Note that you should add the following line to your TOC file:

```
## Dependencies: SimpleTimingLib
```

This makes sure that `SimpleTimingLib` is loaded before the DKP addon is loaded, as we will need this library.

We will start with Lua code to define a few variables that save the current state of the addon as well as the options:

```
local currentItem -- the current item or nil if no auction is running
local bids = {} -- bids on the current item
local prefix = "[SimpleDKP] " -- prefix for chat messages

-- default values for saved variables/options
SimpleDKP_Channel = "GUILD" -- the chat channel to use
SimpleDKP_AuctionTime = 30 -- the time (in seconds) for an auction
```

```
SimpleDKP_MinBid = 15 -- the minimum amount of DKP you have to bid
SimpleDKP_ACL = {} -- the access control list
```

The access control list (ACL) is a list of players who are allowed to control the addon via chat commands. It starts empty, meaning that nobody is allowed to control your addon.

Add the variables `SimpleDKP_Channel`, `SimpleDKP_AuctionTime`, `SimpleDKP_MinBid`, and `SimpleDKP_ACL` to the saved variables in your TOC file. The next function we are going to write is the function that starts an auction. It takes two arguments: the item and the player who requested the auction.

Local Functions and Their Scope

Let's think about this function before we create it. It is a good idea to store this function in a local variable, as we only need to access it from within this file and never from the outside. But what about the scope of this variable? Its scope starts after the variable is created, which means after the statement containing the keyword `local`. So if you have a lot of functions in local variables, a situation like the following might come up:

```
local function foo()
   bar()
end

local function bar()
   -- do something
end
```

The function `foo` tries to call the function `bar` here, but it does not see the local variable `bar`, which is created a few lines later. This means it will try to access a global variable called `bar` and not the local one it should access. So it is a good idea to create all local variables that should be available in the whole file at the beginning of your script. This avoids such problems because the scope of a local variable starts after its declaration; the initialization does not affect it.

If we think about the functions we are going to need, we come up with the following local variables:

```
local startAuction, endAuction, placeBid, cancelAuction, onEvent
```

They have the following meanings:

`startAuction` starts a new auction.

`endAuction` is called when the current auction expires and it sends the results to the chat.

`cancelAuction` can cancel an auction prematurely.

`placeBid` will be called every time someone wants to place a bid on the currently running auction.

`onEvent` will be our event handler.

Starting Auctions

We can now start implementing these functions. The following code shows `startAuction`. Note that all strings used by this function are created outside the function and that the function and its variables are wrapped in a `do-end` block. So these local variables are only visible where they are needed.

Storing strings in variables allows us to change them easily without searching and changing every occurrence of the string. This is also useful when we want to translate our addon later (a topic we will deal with later in the book).

```
do
local auctionAlreadyRunning = "There is already an auction running! (on %s)"
local startingAuction = prefix.."Starting auction for item %s, please place your➥
 bids by whispering me. Remaining time: %d seconds."
local auctionProgress =  prefix.."Time remaining for %s: %d seconds."

function startAuction(item, starter)
   if currentItem then
      local msg = auctionAlreadyRunning:format(currentItem)
      if starter then
         SendChatMessage(msg, "WHISPER", nil, starter)
      else
         print(msg)
      end
   else
      currentItem = item
      SendChatMessage(startingAuction:format(item, SimpleDKP_AuctionTime),➥
SimpleDKP_Channel)
      if SimpleDKP_AuctionTime > 30 then
         SimpleTimingLib_Schedule(SimpleDKP_AuctionTime - 30,➥
SendChatMessage, auctionProgress:format(item, 30), SimpleDKP_Channel)
      end
      if SimpleDKP_AuctionTime > 15 then
         SimpleTimingLib_Schedule(SimpleDKP_AuctionTime - 15,➥
SendChatMessage, auctionProgress:format(item, 15), SimpleDKP_Channel)
      end
      if SimpleDKP_AuctionTime > 5 then
         SimpleTimingLib_Schedule(SimpleDKP_AuctionTime - 5,➥
SendChatMessage, auctionProgress:format(item, 5), SimpleDKP_Channel)
      end
      SimpleTimingLib_Schedule(SimpleDKP_AuctionTime, endAuction)
   end
end
end
```

■ **Caution** Do not use `local function startAuction(item, starter)` here. This would create a new local variable `startAuction`, but we want to use the local variable we created at the beginning of the file.

The function looks long and complicated, but it is pretty simple. It first checks to see if there is an auction running and prints an error message if there is. The error message is whispered to the player who tried to start the auction if it was started remotely. If there is no ongoing auction, it will set the current item to the new item and send a chat message to the configured channel. The next lines are just scheduling status messages to be sent when 30, 15, and 5 seconds remain. The last call in this function schedules `endAuction` to be called at the end.

Ending Auctions

The next function we are going to write is `endAuction`. You will have to adjust this to the DKP system you are using. There are four cases this function has to handle:

- The first three cases are quite simple:
  - There is no bid at all.
  - Only one player bid on the item.
  - More than one player bid and the highest amount is unique.
- The fourth case is that more than one player bid the same amount and it's the highest bid.

The bids will be stored and sorted in the table `bids`, so we have to think of a format for this table before we can write the function to read from it. An entry in the table consists of the player's name and a number representing their bid, so you might be tempted to use the name as the key and the bid as the value. However, this is not a smart approach, because we need sorted tables, and hash tables by design cannot be sorted. If we used a hash table here, we would have to build an array from it and sort the array. So the simpler way to do this is to create a small table with the keys `bid` and `name` for each bid and store these bid tables in the table `bids`. `endAuction` will then sort this table in descending order, so `bids[1].name` will be the winner of the auction.

The following code shows `endAuction` with its helper function to sort the table `bids`:

```
do
local noBids = prefix.."No one wants to have %s :("
local wonItemFor = prefix.."%s won %s for %d DKP."
local pleaseRoll = prefix.."%s bid %d DKP on %s, please roll!"
local highestBidders = prefix.."%d. %s bid %d DKP"

local function sortBids(v1, v2)
   return v1.bid > v2.bid
end
```

```
function endAuction()
   table.sort(bids, sortBids)
   if #bids == 0 then -- case 1: no bid at all
      SendChatMessage(noBids:format(currentItem), SimpleDKP_Channel)
   elseif #bids == 1 then -- case 2: one bid; the bidder pays the minimum bid
     SendChatMessage(wonItemFor:format(bids[1].name, currentItem,➥
SimpleDKP_MinBid), SimpleDKP_Channel)
     SendChatMessage(highestBidders:format(1, bids[1].name, bids[1].bid),➥
SimpleDKP_Channel)
   elseif bids[1].bid ~= bids[2].bid then -- case 3: highest amount is unique
      SendChatMessage(wonItemFor:format(bids[1].name, currentItem, bids[2].bid +
1),➥
SimpleDKP_Channel)
      for i = 1, math.min(#bids, 3) do -- print the three highest bidders
         SendChatMessage(highestBidders:format(i, bids[i].name, bids[i].bid),➥
SimpleDKP_Channel)
      end
   else -- case 4: more than 1 bid and the highest amount is not unique
      local str = "" -- this string holds all players who bid the same amount
      for i = 1, #bids do -- this loop builds the string
         if bids[i].bid ~= bids[1].bid then -- found a player who bid less --> break
            break
         else -- append the player's name to the string
            if bids[i + 2] and bids[i + 2].bid == bid then
               str = str..bids[i].name..", " -- use a comma if this is not the last
            else
               str = str..bids[i].name.." and " -- this is the last player
            end
         end
      end
      str = str:sub(0, -6) -- cut off the end of the string as the loop generates a
      -- string that is too long
      SendChatMessage(pleaseRoll:format(str, bids[1].bid, currentItem),➥
SimpleDKP_Channel)
   end
   currentItem = nil -- set currentItem to nil as there is no longer an
   -- ongoing auction
   table.wipe(bids) -- clear the table that holds the bids
end
end
```

The `if` blocks clearly show the four cases the function handles. The first three cases are pretty easy. The function just has to format a few strings and pass them to `SendChatMessage`; nothing special here. The fourth case has to build a string that contains the names of all players who bid the same amount. The resulting string looks like `"Player1, Player2 and Player 3"` and is built in the loop.

Once it has dealt with the auction, the function sets the variables `currentItem` to `nil` and wipes the table `bids`. The addon is now ready for the next auction. Our next function brings us

closer to the topic of this chapter: events. We need a function that allows players to place a bid via whisper on an item.

Placing Bids

This function needs to handle the event CHAT_MSG_WHISPER and check if an auction is currently running and if the whisper message is a number. It then creates or updates an entry in the table bids.

```
do
   local oldBidDetected = prefix.."Your old bid was %d DKP, your new bid is %d DKP."
   local bidPlaced = prefix.."Your bid of %d DKP has been placed!"
   local lowBid = prefix.."The minimum bid is %d DKP."

   function onEvent(self, event, msg, sender)
      if event == "CHAT_MSG_WHISPER" and currentItem and tonumber(msg) then
         local bid = tonumber(msg)
         if bid < SimpleDKP_MinBid then
            SendChatMessage(lowBid:format(SimpleDKP_MinBid), "WHISPER", nil, sender)
            return
         end
         for i, v in ipairs(bids) do -- check if that player has already bid
            if sender == v.name then
               SendChatMessage(oldBidDetected:format(v.bid, bid), "WHISPER",➥
nil, sender)
               v.bid = bid
               return
            end
         end
         -- he hasn't bid yet, so create a new entry in bids
         table.insert(bids, {bid = bid, name = sender})
         SendChatMessage(bidPlaced:format(bid), "WHISPER", nil, sender)
      end
   end
end
```

The event handler checks if currentItem is set, which would mean that an auction is running. It uses tonumber(msg) to determine if the whisper is a number. The function checks whether the player has already placed a bid and updates it if necessary; otherwise it creates a new entry in bids.

We now need to create a frame and use it as event listener for the CHAT_MSG_WHISPER event and set our event handler function onEvent as script handler for the script OnEvent. Recall that a script handler that handles the OnEvent script is called an event handler, so every event handler is a script handler. Place the following code after the do-end block we inserted above:

```
local frame = CreateFrame("Frame")
frame:RegisterEvent("CHAT_MSG_WHISPER")
frame:SetScript("OnEvent", onEvent)
```

But we still can't test the addon in-game, as there is no way to create an auction. So let's build some slash commands to do that.

Creating Slash Commands

We will register the slash commands `/simpledkp` and `/sdkp`. Table 4-1 shows the commands we are going to build.

Table 4-1. *SimpleDKP slash commands*

Command	Description
`/sdkp start <item>`	Starts an auction for <*item*>.
`/sdkp stop`	Stops the current auction.
`/sdkp channel <channel>`	Sets the chat channel to <*channel*> if <*channel*> is provided. Otherwise it will print the current channel.
`/sdkp time <time>`	Sets the time to <*time*>. If <*time*> is omitted it prints the current setting.
`/sdkp minbid <minbid>`	Sets the lowest bid to <*minbid*>. If <*minbid*> is omitted it prints the current setting.
`/sdkp acl`	Prints the list of players who are allowed to control the addon remotely.
`/sdkp acl add <names>`	Adds <*names*> to the ACL.
`/sdkp acl remove <names>`	Removes <*names*> from the ACL.

The slash command handler is quite long as we have eight commands to deal with. The commands `acl add` and `acl remove` are capable of taking a list of names separated by spaces. We can get the names from this string by using `string.split`, but we don't know how many results we will get. So one possible solution to this is the following code:

```
for i = 1, select("#", string.split(" ", names)) do
   local name = select(i, string.split(" ", names))
   -- do something with the name
end
```

But this will execute `string.split` every time it enters the loop. So the better way is to create a helper function with a vararg. This function can then operate on the vararg, and the splitting will be done only once, when calling the function. So our slash command handler looks like this:

```
SLASH_SimpleDKP1 = "/simpledkp"
SLASH_SimpleDKP2 = "/sdkp"

do
   local setChannel = "Channel is now \"%s\""
   local setTime = "Time is now %s"
   local setMinBid = "Lowest bid is now %s"
   local addedToACL = "Added %s player(s) to the ACL"
```

```
local removedFromACL = "Removed %s player(s) from the ACL"
local currChannel = "Channel is currently set to \"%s\""
local currTime = "Time is currently set to %s"
local currMinBid = "Lowest bid is currently set to %s"
local ACL = "Access Control List:"

local function addToACL(...) -- adds multiple players to the ACL
   for i = 1, select("#", ...) do -- iterate over the arguments
      SimpleDKP_ACL[select(i, ...)] = true -- and add all players
   end
   print(addedToACL:format(select("#", ...))) -- print an info message
end

local function removeFromACL(...) -- removes player(s) from the ACL
   for i = 1, select("#", ...) do -- iterate over the vararg
      SimpleDKP_ACL[select(i, ...)] = nil -- remove the players from the ACL
   end
   print(removedFromACL:format(select("#", ...))) -- print an info message
end

SlashCmdList["SimpleDKP"] = function(msg)
   local cmd, arg = string.split(" ", msg) -- split the string
   cmd = cmd:lower() -- the command should not be case-sensitive
   if cmd == "start" and arg then -- /sdkp start item
      startAuction(msg:match("^start%s+(.+)")) -- extract the item link
   elseif cmd == "stop" then -- /sdkp stop
      cancelAuction()
   elseif cmd == "channel" then -- /sdkp channel arg
      if arg then -- a new channel was provided
         SimpleDKP_Channel = arg:upper() -- set it to arg
         print(setChannel:format(SimpleDKP_Channel))
      else -- no channel was provided
         print(currChannel:format(SimpleDKP_Channel)) -- print the current one
      end
   elseif cmd == "time" then -- /sdkp time arg
      if arg and tonumber(arg) then -- arg is provided and it is a number
         SimpleDKP_AuctionTime = tonumber(arg) -- set it
         print(setTime:format(SimpleDKP_AuctionTime))
      else -- arg was not provided or it wasn't a number
         print(currTime:format(SimpleDKP_AuctionTime)) -- print error message
      end
   elseif cmd == "minbid" then -- /sdkp minbid arg
      if arg and tonumber(arg) then -- arg is set and a number
         SimpleDKP_MinBid = tonumber(arg) -- set the option
         print(setMinBid:format(SimpleDKP_MinBid))
      else -- arg is not set or not a number
         print(currMinBid:format(SimpleDKP_MinBid)) -- print error message
```

```
         end
      elseif cmd == "acl" then -- /sdkp acl add/remove player1, player2, ...
         if not arg then -- add/remove not passed
            print(ACL) -- output header
            for k, v in pairs(SimpleDKP_ACL) do -- loop over the ACL
               print(k) -- print all entries
            end
         elseif arg:lower() == "add" then -- /sdkp add player1, player2, ...
            -- split the string and pass all players to our helper function
            addToACL(select(3, string.split(" ", msg)))
         elseif arg:lower() == "remove" then -- /sdkp remove player1, player2, ...
            removeFromACL(select(3, string.split(" ", msg))) -- split & remove
         end
      end
   end
end
```

The code is long but simple. It basically just deals with setting and retrieving options and is a long `if-then-else` block that processes the different commands. The only part that is slightly more difficult is the part that deals with the ACL. This part splits the string and passes everything except the first two substrings to the helper function that adds or removes the names from the table.

We can now test the mod in-game. It is working pretty well, but there are a few issues left: the `stop` command does not work, as we haven't implement the `cancelAuction` function yet; the ACL is pointless as we haven't implemented remote control yet; and we can see outgoing whispers and placed bids in our chat frames. We don't want to see them. Let's fix these problems.

Canceling Auctions

Canceling auctions is pretty easy. We just need to set the variable `currentItem` to `nil`, wipe the table `bids`, and unschedule all scheduled tasks. Here's the code:

```
do
   local cancelled = "Auction cancelled by %s"
   function cancelAuction(sender)
      currentItem = nil
      table.wipe(bids)
      SimpleTimingLib_Unschedule(SendChatMessage)
      SimpleTimingLib_Unschedule(endAuction)
      SendChatMessage(cancelled:format(sender or UnitName("player")),➥
SimpleDKP_Channel)
   end
end
```

The API function `UnitName(`*`unitId`*`)` returns the name of a unit as it appears in the game. The name `"player"` always refers to you, so `UnitName("player")` returns the name of your character.

■ **Tip** A unit ID is used to identify a unit, for example `target` is your target, `pet` is your pet, and `party1` is the first member of your party. You can find all available unit IDs in Appendix B.

Our cancel function exposes an issue with our timing library: we are unscheduling all calls to `SendChatMessage` here, but other addons that use this library might also have scheduled a call to this function. Therefore, we do not want to unschedule this one. You will see how to fix this issue in Chapter 7 when we update the library.

Remote Control

We want to allow players on your ACL list to create and cancel auctions via chat commands. You can add your guild's officers and raid leaders to this list, so they can start auctions when you are not the master looter in your raid. We want to listen for chat commands in the raid, as well as guild, whisper, and officer chat, so we need to add those events. Go to the line in your code where you registered the event `CHAT_MSG_WHISPER` and add the following lines of code there:

```
frame:RegisterEvent("CHAT_MSG_WHISPER") -- look for this line...
frame:RegisterEvent("CHAT_MSG_RAID") -- ...and insert these four new lines after it
frame:RegisterEvent("CHAT_MSG_RAID_LEADER")
frame:RegisterEvent("CHAT_MSG_GUILD")
frame:RegisterEvent("CHAT_MSG_OFFICER")
```

Our event handler will now also be called for these events. Their first two arguments are the same as they are for whisper messages. So we can just change the code in our event handler to the following code:

```
function onEvent(self, event, msg, sender)
   if event == "CHAT_MSG_WHISPER" and currentItem and tonumber(msg) then
      -- old code here
   elseif SimpleDKP_ACL[sender] then
      -- not a whisper or a whisper that is not a bid
      -- and the sender has the permission to send commands
      local cmd, arg = msg:match("^!(%w+)%s*(.*)")
      if cmd and cmd:lower() == "auction" and arg then
         startAuction(arg, sender)
      elseif cmd and cmd:lower() == "cancel" then
         cancelAuction(sender)
      end
   end
end
```

A player on the access control list can now create an auction by writing `!auction <item>` in the chat. To cancel an auction they must enter `!cancel`. The pattern that allows this might look complicated if you are still getting accustomed to regular expressions, so let's analyze it.

The first character, ^, tells us that it looks for the beginning of the string. This is useful because we don't want to trigger the code if someone writes text like "...!cancel ..." to the chat.

The next character is `!`, which has no special meaning. It is followed by a capture that matches `%w+`, one or more alphanumeric characters. This is the first capture, so the found string will be returned as the first argument and stored in the local variable `cmd`. The next expression is `%s*`, which means we are looking for whitespace, and * means that it matches zero or more. The next expression is our second capture; it matches `.*`, an arbitrary number of arbitrary characters, which matches the rest of the line.

This means we are looking for an exclamation mark at the beginning of the string, followed by a word. This can optionally be followed by whitespace and any other text. The first word is the first return value (`cmd`) and the optional text after the optional whitespace is the second return value (`arg`).

One issue remains: we do not want to see outgoing whispers and incoming bids.

Hiding Chat Messages

Blizzard provides the function `ChatFrame_AddMessageEventFilter(`*`event`*`, `*`func`*`)`, which allows us to set up filters for chat messages. The function takes an event and a function as arguments; *func* is called every time before a chat message of the type *event* is displayed. This function receives the same arguments as the event handler of the corresponding event. The first return value of the function is a boolean that determines if the chat frame should cancel processing of this event. So setting this to `true` will filter the message. The function also has to return all arguments of the event (`arg1` - `arg11` for chat messages) that are also passed to it. This allows you to modify these arguments to change incoming chat messages.

But we only need the simple part of this chat filter API here as we just want to hide certain messages but not modify them. We need to set up filters for the events `CHAT_MSG_WHISPER` and `CHAT_MSG_WHISPER_INFORM`; the former will filter incoming bids, the latter is responsible for outgoing whisper messages. Suitable filter functions look like the following; you can just insert them anywhere in your DKP auctioneer file (preferably at the end):

```
local function filterIncoming(self, event, ...)
   local msg = ... -- get the message from the vararg
   -- return true if there is an ongoing auction and the whisper is a number
   -- followed by all event handler arguments
   return currentItem and tonumber(msg), ...
end

local function filterOutgoing(self, event, ...)
   local msg = ... -- extract the message
   return msg:sub(0, prefix:len()) == prefix, ...
end

ChatFrame_AddMessageEventFilter("CHAT_MSG_WHISPER", filterIncoming)
ChatFrame_AddMessageEventFilter("CHAT_MSG_WHISPER_INFORM", filterOutgoing)
```

The function `filterIncoming` does the same check that is also done by the event handler to determine if an incoming whisper is a bid and does not display the message if it is a bid. The function `filterOutgoing` checks if the message starts with the prefix that was defined at the beginning of our script and swallows the message if it does. Both functions also have to return all arguments of the event handler, even if we didn't change them.

The mod is now ready to be used. It is a fully featured DKP auctioneer that you can easily adjust to match the rules of your DKP system if you use a different one.

Summary

In this chapter you learned how to use frames to listen to scripts and events. This allows us to write addons that respond to events from the game. Virtually every addon makes use of script handlers and event handlers, so this is a very important part of the World of Warcraft API. We have written three powerful example mods in this chapter: The first one added mouse-over tooltips for links in chat frames, the second was a library that allowed us to schedule functions to be called after a certain time has passed, and the third example was a fully featured DKP auctioneer.

All of these addons made extensive use of script handlers and events. We worked with a lot of events, but only a few script handlers. That's because most script handlers are related to the graphical user interface. This leads to our next topic, creating graphical user interface elements in World of Warcraft.