Examining the Call Stack

While a program is running, it stores information about what it's doing in a data structure known as a *call stack*. Each time a method is called, the program pushes a new *stack frame* on top of the call stack, which contains the following: the arguments passed to the method, if any, the local variables of the method, if any, and the address to return to after the method call finishes.

When the program stops at a breakpoint, you can interact with the debugger to examine the state of the current stack frame. This allows you to reason about the behavior of a method and how it interacts with other parts of a program.

In addition to getting information about the current stack frame, you can interact with the debugger to examine the entire call stack of the current thread, and of other threads used by the program.

Getting Information About the Current Frame

By entering the `frame info` command, you get the location of the current frame in code, including the source file and line number.

```
(lldb) frame info
frame #0: 0x0000000100001277 Validation`(string="Sw0rdf!sh") -> Bool).
(containsSymbol #1)(String) -> Bool + 23 at Validation.swift:8
```

Inspecting Variables

You use the `frame variable` (`f v`) command to get a list of all the variables in the stack frame.

```
(lldb) frame variable
(String) name = "Anton"
(Greeter.Greeter) self = 0x0000000100502920 {
  acquaintances = ([0] = "Anton")
}
```

To get information about an individual variable, you use the `frame variable` command, passing the variable name as an argument.

```
(lldb) frame variable name
(String) name = "Anton"
```

Evaluating Expressions

One of the most powerful features of LLDB is the ability to evaluate expressions from within a debug session.

In the example used in the Quick Tour of LLDB chapter, the `expression` (`e`) command is used to modify the state of a stored property to change the final output of the program.

```
(lldb) expression -- acquaintances.insert("Mei")
(lldb) expression -- acquaintances.remove("Anton")
(String?) $R1 = "Anton"
```

The `expression` (`e`) command evaluates the passed argument as an expression in the target language. For example, when debugging a Swift program, you can evaluate Swift code as if it were a read-eval-print (REPL) loop in the context of the current stack frame. This is a powerful way to introspect variables at different points during execution.

Printing Modes

When inspecting values during a debug session, it is important to understand the difference between the `frame variable` and `expression` commands.

`frame variable` (**`f v`**)	**`expression --`** (`p`)	**`expression -O --`** (**`po`**)
Does not run code	Runs your code	Runs your code
Uses LLDB formatters	Uses LLDB formatters	Adds code to format objects

Using the `frame variable` (`f v`) command doesn't run any code, and therefore does not produce any side effects. Accessing a property or evaluating the result of calling a method can often change the state of a program in a way that obscures the problem that you are attempting to debug, so using the `frame variable` (`f v`) command is the safest option for cursory investigation.

The `expression` command is aliased to both `p` and `po`, which are frequently used operations when debugging. The difference between the two is that `p` uses the built-in LLDB data formatters, whereas `po` calls code provided by the developer that creates a representation of that object, such as the `debugDescription` method in Swift. If no custom representation is available, the `po` command falls back on the representation provided by the `p` command.

Note: Use `p` when you want to use default LLDB formatting, and use `po` when you want the implementation of the type to control presentation.

Getting a Backtrace

A *backtrace* is a list of the currently active function calls. By using the `thread backtrace` (`bt`) command, you can reason more clearly about the chain of events that caused the program to be in its current state.

Calling the `thread backtrace` command with no arguments produces a backtrace of the current thread.

```
(lldb) thread backtrace
* thread #1: tid = 0x1288be3, 0x0000000100001a98
Greeter`Greeter.hasMet(name="Anton", self=0x0000000101200190) -> Bool + 24 at
Greeter.swift:5, queue = 'com.apple.main-thread', stop reason = step in
    frame #0: 0x0000000100001a98 Greeter`Greeter.hasMet(name="Anton",
self=0x0000000101200190) -> Bool + 24 at Greeter.swift:5
  * frame #1: 0x0000000100001be4 Greeter`Greeter.greet(name="Anton",
self=0x0000000101200190) -> () + 84 at Greeter.swift:9
```

```
    frame #2: 0x00000001000019eb Greeter`main + 155 at Greeter.swift:20
    frame #3: 0x00007fff949d05ad libdyld.dylib`start + 1
    frame #4: 0x00007fff949d05ad libdyld.dylib`start + 1
```

You can pass an integer as an argument to the `thread backtrace` command to limit the number of frames that are displayed.

Alternatively, calling the `thread backtrace` command with `all` as the argument produces a full backtrace of all threads.

Listing Threads

Programs often execute code across multiple threads. To get a list of all of the current threads in a process, you use the `thread list` command.

```
(lldb) thread list
Process 96461 stopped
* thread #1: tid = 0x1384af1, 0x000000010000111f main`sayHello() -> () + 15 at
main.swift:2, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
```