

 WWDC2013

Advanced Debugging with LLDB

Session 413

Kate Stone

Software Behavioralist

These are confidential sessions—please refrain from streaming, blogging, or taking pictures

What to Expect from This Talk

What to Expect from This Talk

- Emphasis on LLDB as our debugging foundation

Debugging with Xcode

Pacific Heights
Wednesday 2:00PM

What to Expect from This Talk

- Emphasis on LLDB as our debugging foundation

Debugging with Xcode

Pacific Heights
Wednesday 2:00PM

- Tips to streamline the debugging experience

What to Expect from This Talk

- Emphasis on LLDB as our debugging foundation

Debugging with Xcode

Pacific Heights
Wednesday 2:00PM

- Tips to streamline the debugging experience
- LLDB as an investigative tool

What to Expect from This Talk

- Emphasis on LLDB as our debugging foundation

Debugging with Xcode

Pacific Heights
Wednesday 2:00PM

- Tips to streamline the debugging experience
- LLDB as an investigative tool
- Our collective goal: **reliable apps!**

State of LLDB

State of LLDB

- Hundreds of improvements
 - Most stable LLDB ever
 - The debugger in Xcode 5

State of LLDB

- Hundreds of improvements
 - Most stable LLDB ever
 - The debugger in Xcode 5
- Improved data inspection
 - Formatters for more Foundation types
 - Unicode text in C++ types

State of LLDB

- Hundreds of improvements
 - Most stable LLDB ever
 - The debugger in Xcode 5
- Improved data inspection
 - Formatters for more Foundation types
 - Unicode text in C++ types
- Improved expression parser
 - Always up to date with language features
 - Fewer explicit casts required

Best Practices in Debugging

Start well informed



Best Practices in Debugging

Start well informed

- Techniques for avoiding long investigations
 - Assertions
 - Logging
 - Static analysis
 - Runtime memory tools



Best Practices in Debugging

Start well informed

- Techniques for avoiding long investigations
 - Assertions
 - Logging
 - Static analysis
 - Runtime memory tools
- Good unit tests



Best Practices in Debugging

Start well informed



- Techniques for avoiding long investigations
 - Assertions
 - Logging
 - Static analysis
 - Runtime memory tools
- Good unit tests

Unit Testing in Xcode

iTunes
WWDC 2012

- Xcode debug configuration
 - Enables debug information, disables optimization

Best Practices in Debugging

Avoid common mistakes



Best Practices in Debugging

Avoid common mistakes



- Take advantage of LLDB
 - Stop exactly where you want to
 - Customize with data formatters, commands
 - Write debug code without rebuilding

Best Practices in Debugging

Avoid common mistakes



- Take advantage of LLDB
 - Stop exactly where you want to
 - Customize with data formatters, commands
 - Write debug code without rebuilding
- Watch out for side effects
 - Expressions can and will change execution

Best Practices in Debugging

The canonical process



Best Practices in Debugging

The canonical process

- Choose your focus



Best Practices in Debugging

The canonical process

- Choose your focus
- Stop before suspect path



Best Practices in Debugging

The canonical process

- Choose your focus
- Stop before suspect path
- Step through live code



Best Practices in Debugging

The canonical process

- Choose your focus
- Stop before suspect path
- Step through live code
- Inspect data to validate assumptions



Finding Problems

Avoiding long investigations

Sean Callanan

LLDB/Clang Integrator

Debug-Only Assertions

Debug-Only Assertions

- Assertions stop your app in situations that should be impossible

```
NSAssert (_dictionary != nil, @"_dictionary should be initialized");
```

Debug-Only Assertions

- Assertions stop your app in situations that should be impossible

```
NSAssert (_dictionary != nil, @"_dictionary should be initialized");
```

- You can also use them to enforce contracts between components

```
NSAssert ((buffer != nil) || (length == 0),  
          @"empty buffer with nonzero length");
```

Debug-Only Assertions

- Assertions stop your app in situations that should be impossible

```
NSAssert (_dictionary != nil, @"_dictionary should be initialized");
```

- You can also use them to enforce contracts between components

```
NSAssert ((buffer != nil) || (length == 0),  
         @"empty buffer with nonzero length");
```

- **NS_BLOCK_ASSERTIONS** disables assertions in release builds

Debug-Only Assertions

- Assertions stop your app in situations that should be impossible

```
NSAssert (_dictionary != nil, @"_dictionary should be initialized");
```

- You can also use them to enforce contracts between components

```
NSAssert ((buffer != nil) || (length == 0),  
         @"empty buffer with nonzero length");
```

- `NS_BLOCK_ASSERTIONS` disables assertions in release builds

- Make sure your condition doesn't do necessary work!

```
NSAssert(myString = [myDictionary objectForKey:@"key"],  
         @"'key' not in dict");
```

Log Effectively with ASL

Log Effectively with ASL

- Logging lets you review an execution of your code after the fact

Log Effectively with ASL

- Logging lets you review an execution of your code after the fact
- Use ASL log levels to distinguish log severity effectively
 - `ASL_LEVEL_EMERG`
 - `ASL_LEVEL_DEBUG`

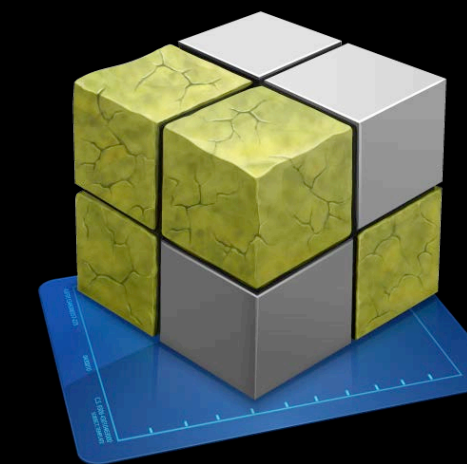
Log Effectively with ASL

- Logging lets you review an execution of your code after the fact
- Use ASL log levels to distinguish log severity effectively
 - `ASL_LEVEL_EMERG`
 - `ASL_LEVEL_DEBUG`
- Use hash tags like `#web` in log messages

Log Effectively with ASL

- Logging lets you review an execution of your code after the fact
- Use ASL log levels to distinguish log severity effectively
 - `ASL_LEVEL_EMERG`
 - `ASL_LEVEL_DEBUG`
- Use hash tags like `#web` in log messages
- Have switches for the heaviest logging (e.g., `NSUserDefaults`)

Validate Your Program with Xcode



- **-Weverything** and the static analyzer find problems as you compile

What's New In LLVM

iTunes
WWDC 2012

What's New in the LLVM Compiler

Pacific Heights
Tuesday 2:00PM

- Guard Malloc catches buffer overruns on the heap
- Zombie Objects catch method calls to freed objects

Advanced Memory Analysis with Instruments

iTunes
WWDC 2010

Stopping Before Problems Occur

Breakpoints at work

Command Syntax

A quick recap

- Commands can have three forms:

▪ Discoverable form	<code>expression --object-description -- foo</code>
▪ Abbreviated form	<code>e -0 -- foo</code>
▪ Alias	<code>po foo</code>

- We will use this notation:

```
po foo
```

```
expression --object-description -- foo
```

Command Syntax

A quick recap

- Commands can have three forms:

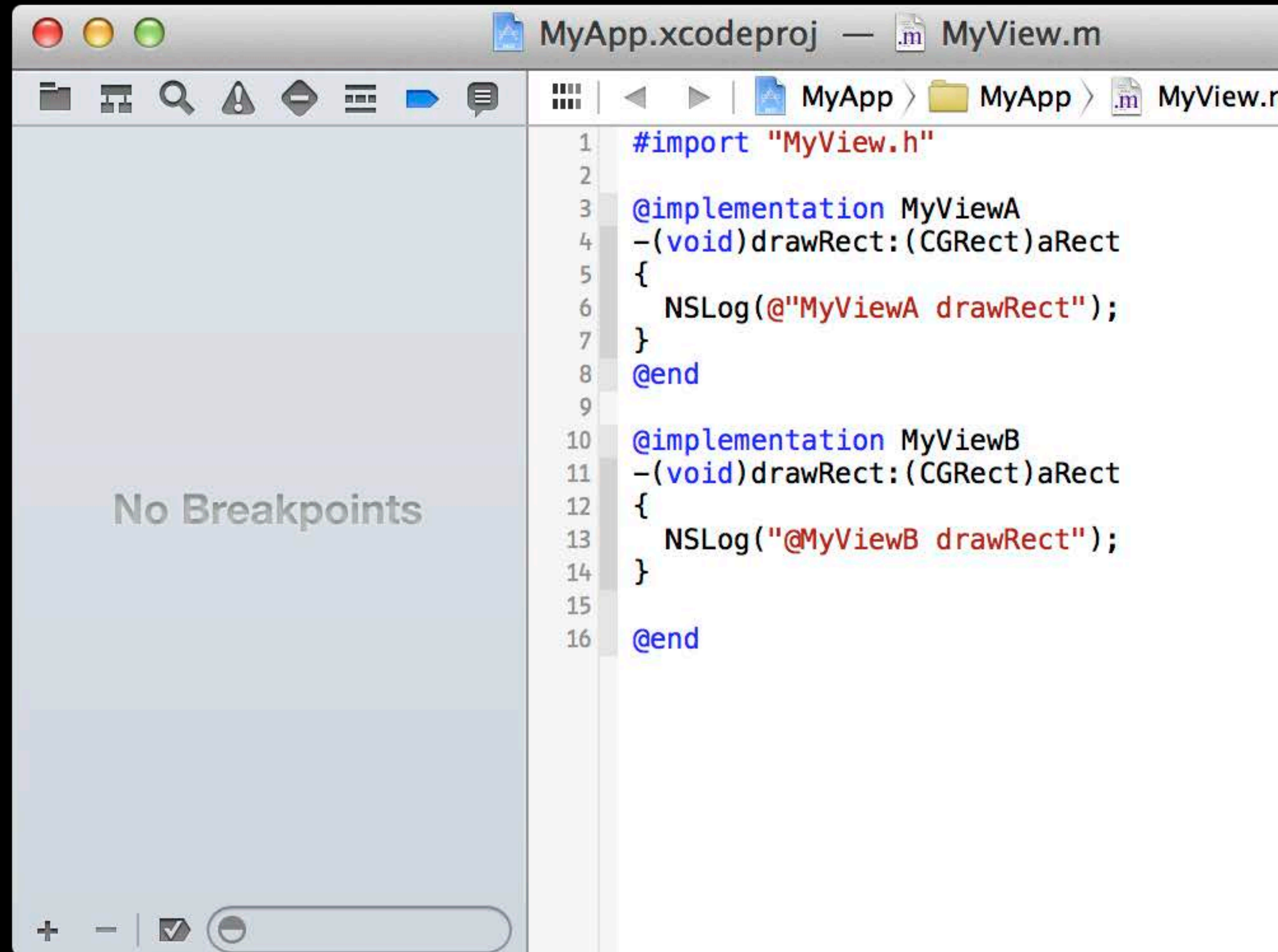
▪ Discoverable form	<code>expression --object-description -- foo</code>
▪ Abbreviated form	<code>e -0 -- foo</code>
▪ Alias	<code>po foo</code>

- We will use this notation:

`po foo` ← Shortest possible form

`expression --object-description -- foo` ← Discoverable form

Common Breakpoint Scenarios



The screenshot shows the Xcode IDE interface. On the left, the Breakpoint Navigator pane is visible, displaying the text "No Breakpoints". The main editor area on the right shows the source code for `MyView.m`. The code includes two implementations of the `drawRect:` method for `MyViewA` and `MyViewB`. The `MyViewA` implementation logs the message `"@MyViewA drawRect"`, and the `MyViewB` implementation logs `"@MyViewB drawRect"`. The code is as follows:

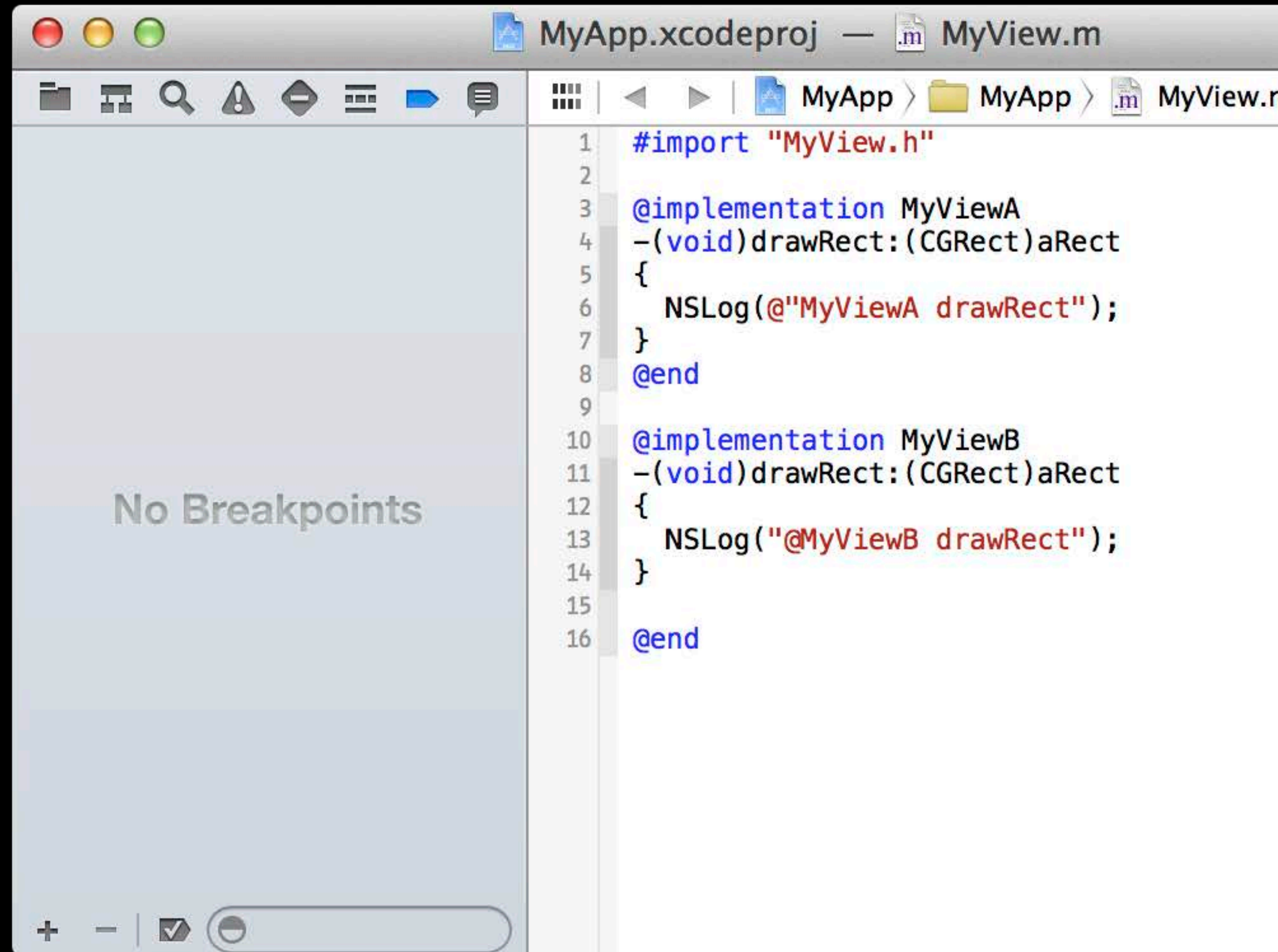
```
1  #import "MyView.h"
2
3  @implementation MyViewA
4  -(void)drawRect:(CGRect)aRect
5  {
6      NSLog(@"MyViewA drawRect");
7  }
8  @end
9
10 @implementation MyViewB
11 -(void)drawRect:(CGRect)aRect
12 {
13     NSLog(@"MyViewB drawRect");
14 }
15
16 @end
```

Common Breakpoint Scenarios

- Stop at a source line:

```
b MyView.m:4
```

```
breakpoint set  
--file MyView.m --line 4
```

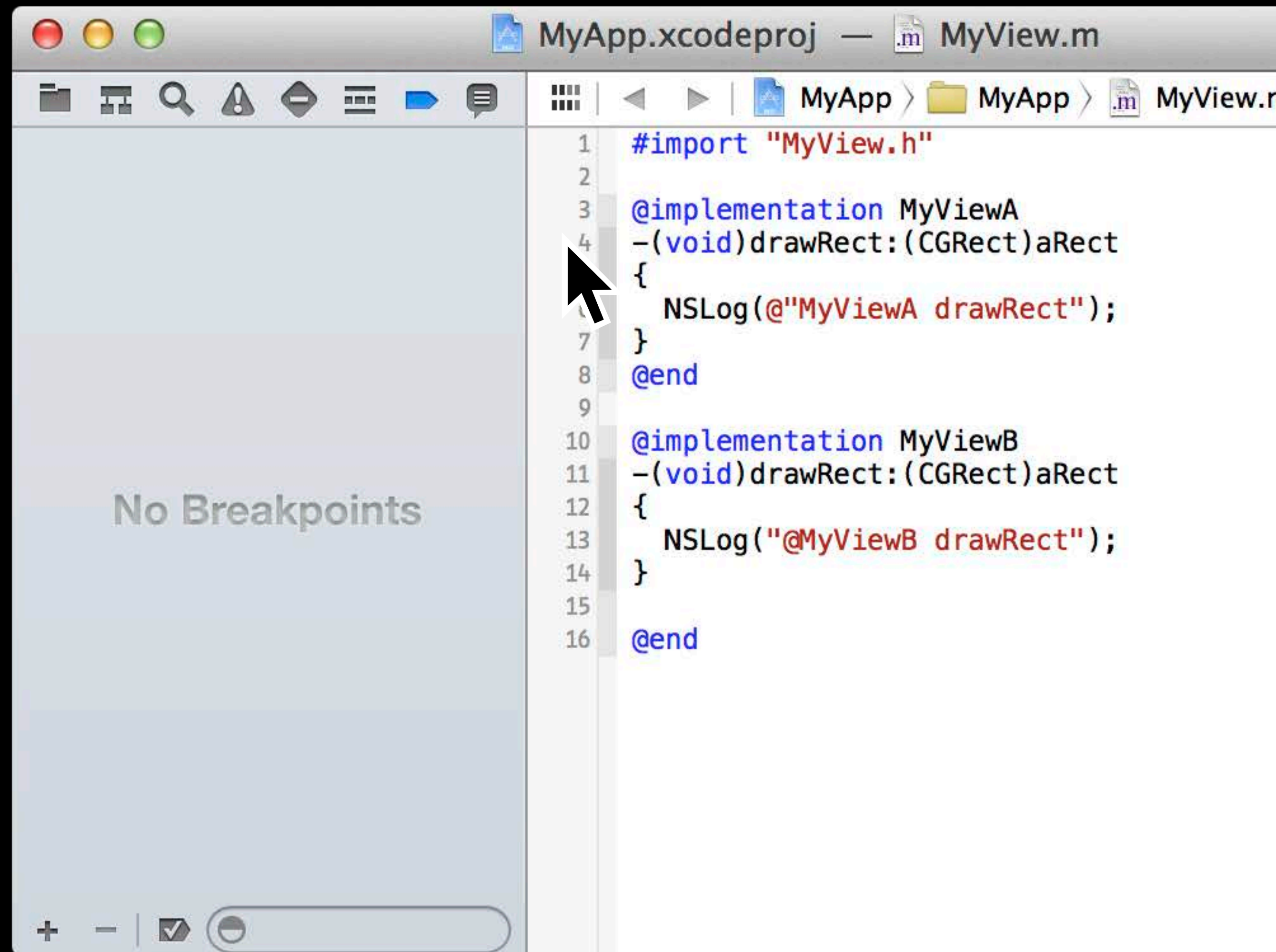


Common Breakpoint Scenarios

- Stop at a source line:

```
b MyView.m:4
```

```
breakpoint set  
--file MyView.m --line 4
```



The screenshot shows the Xcode IDE with a breakpoint set at line 4 of MyView.m. The left pane displays "No Breakpoints". The right pane shows the source code for MyView.m, with a red line indicating the breakpoint at line 4. The code is as follows:

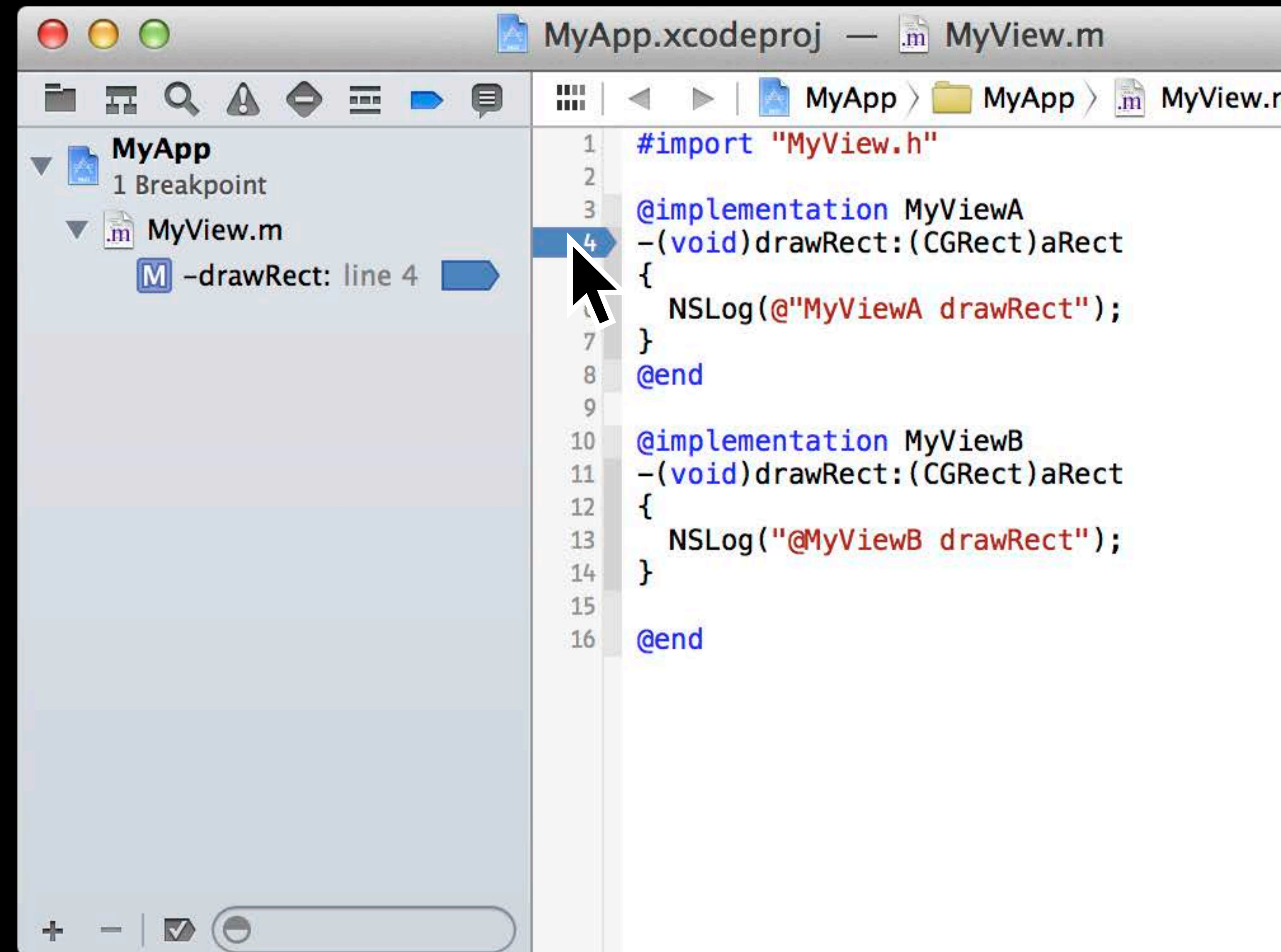
```
1 #import "MyView.h"  
2  
3 @implementation MyViewA  
4 -(void)drawRect:(CGRect)aRect  
5 {  
6     NSLog(@"MyViewA drawRect");  
7 }  
8 @end  
9  
10 @implementation MyViewB  
11 -(void)drawRect:(CGRect)aRect  
12 {  
13     NSLog(@"MyViewB drawRect");  
14 }  
15  
16 @end
```

Common Breakpoint Scenarios

- Stop at a source line:

b MyView.m:4

```
breakpoint set  
--file MyView.m --line 4
```



Common Breakpoint Scenarios

- Stop at a source line:

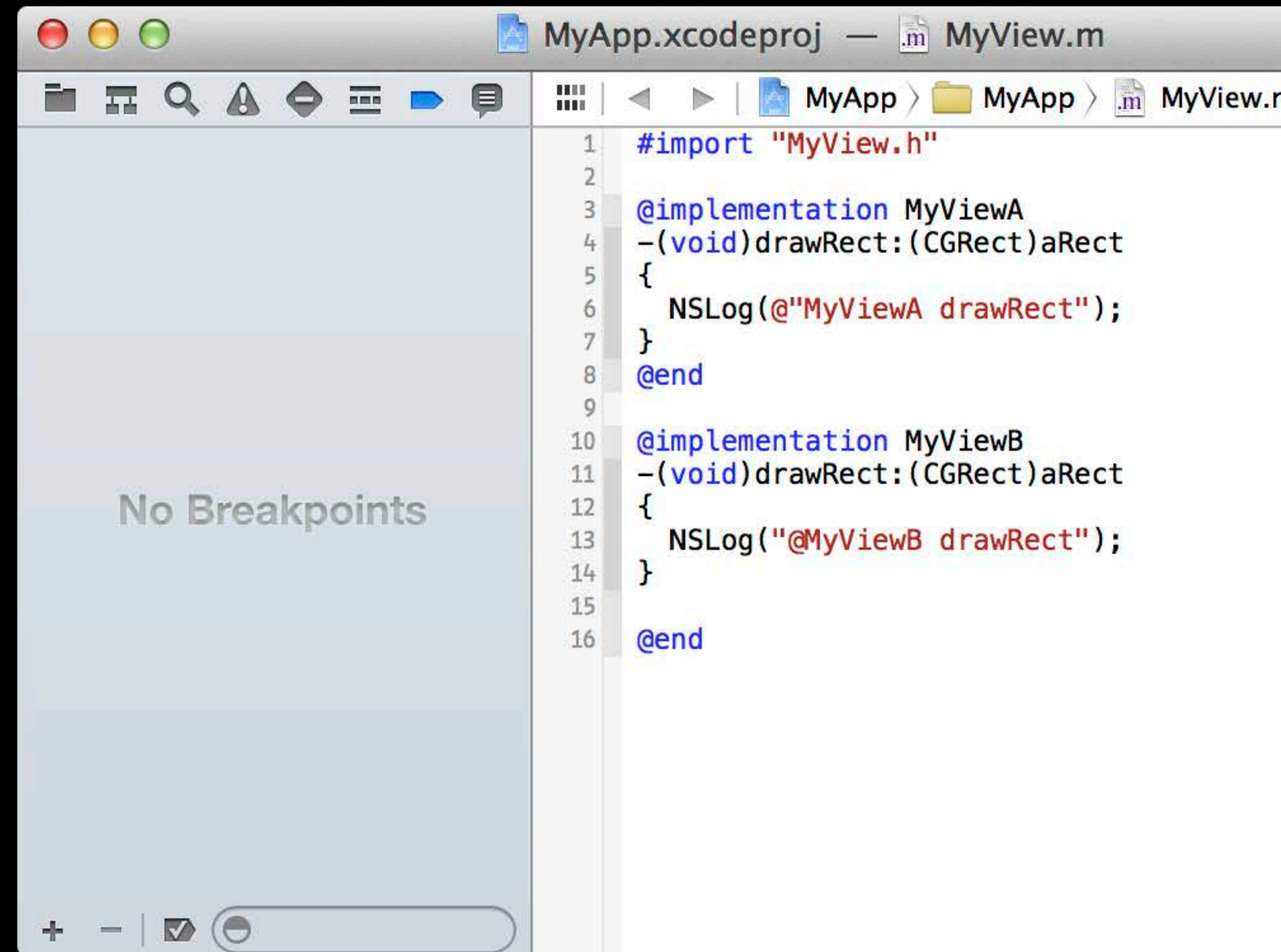
```
b MyView.m:4
```

```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```



Common Breakpoint Scenarios

- Stop at a source line:

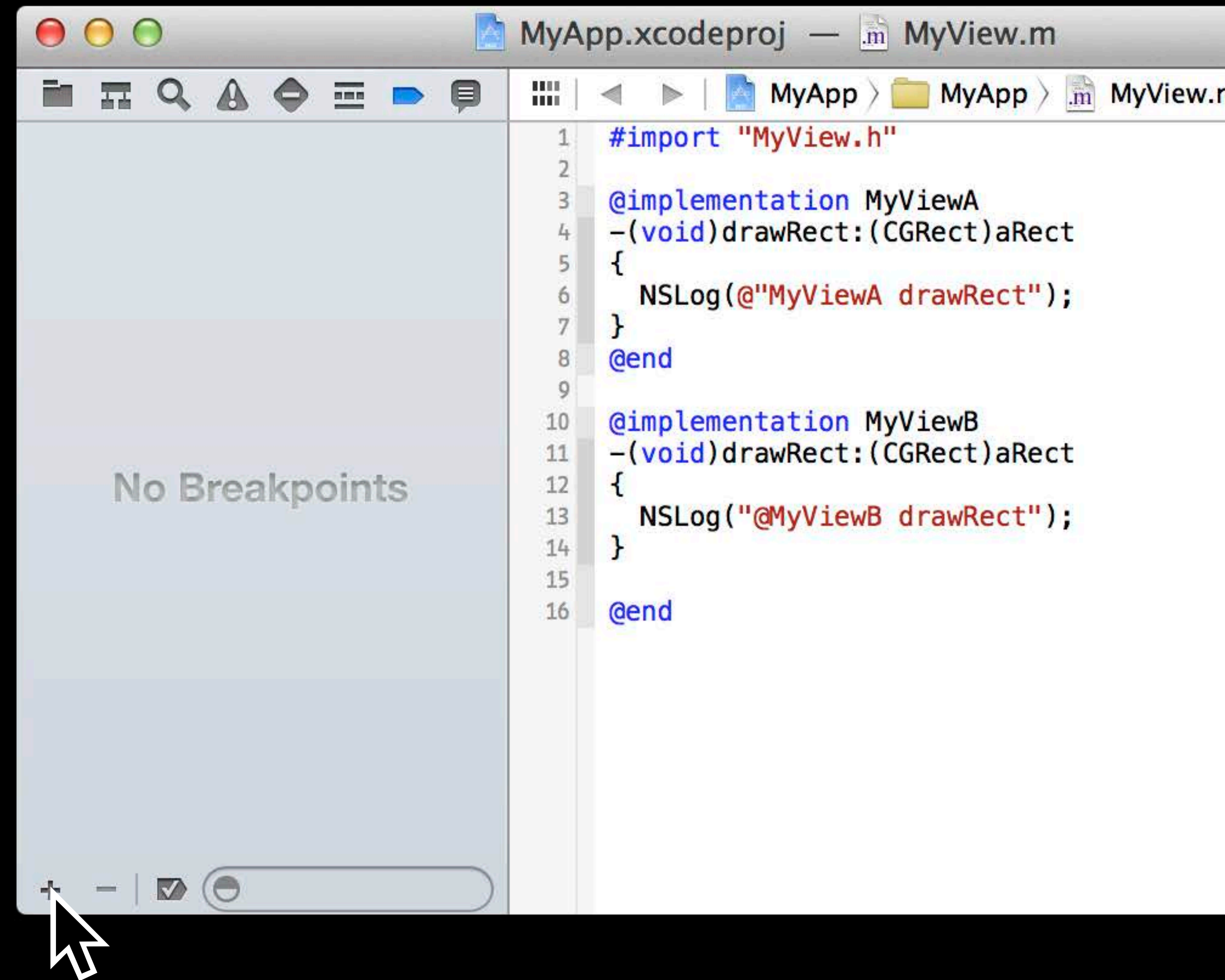
```
b MyView.m:4
```

```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```



Common Breakpoint Scenarios

- Stop at a source line:

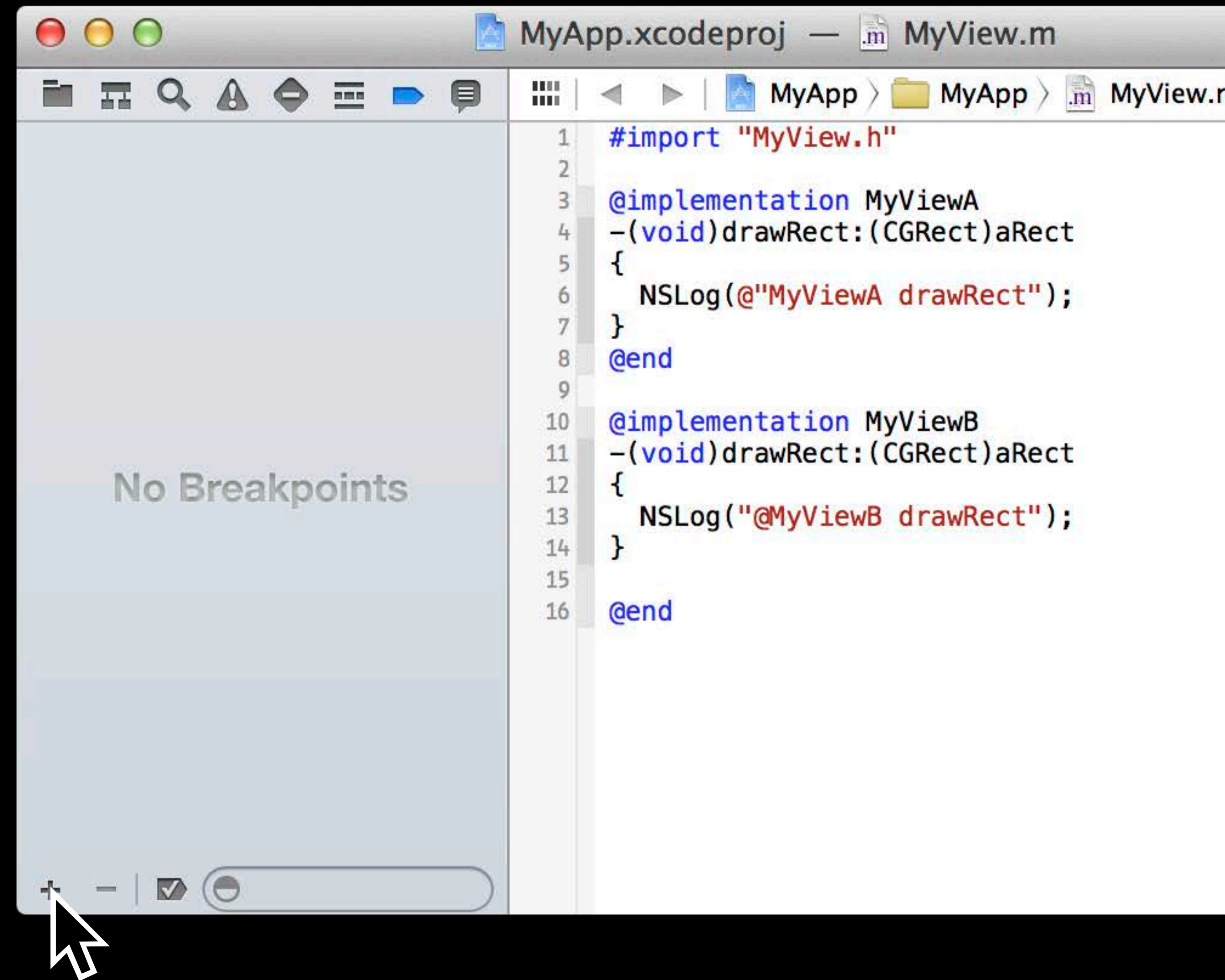
```
b MyView.m:4
```

```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```



Common Breakpoint Scenarios

- Stop at a source line:

```
b MyView.m:4
```

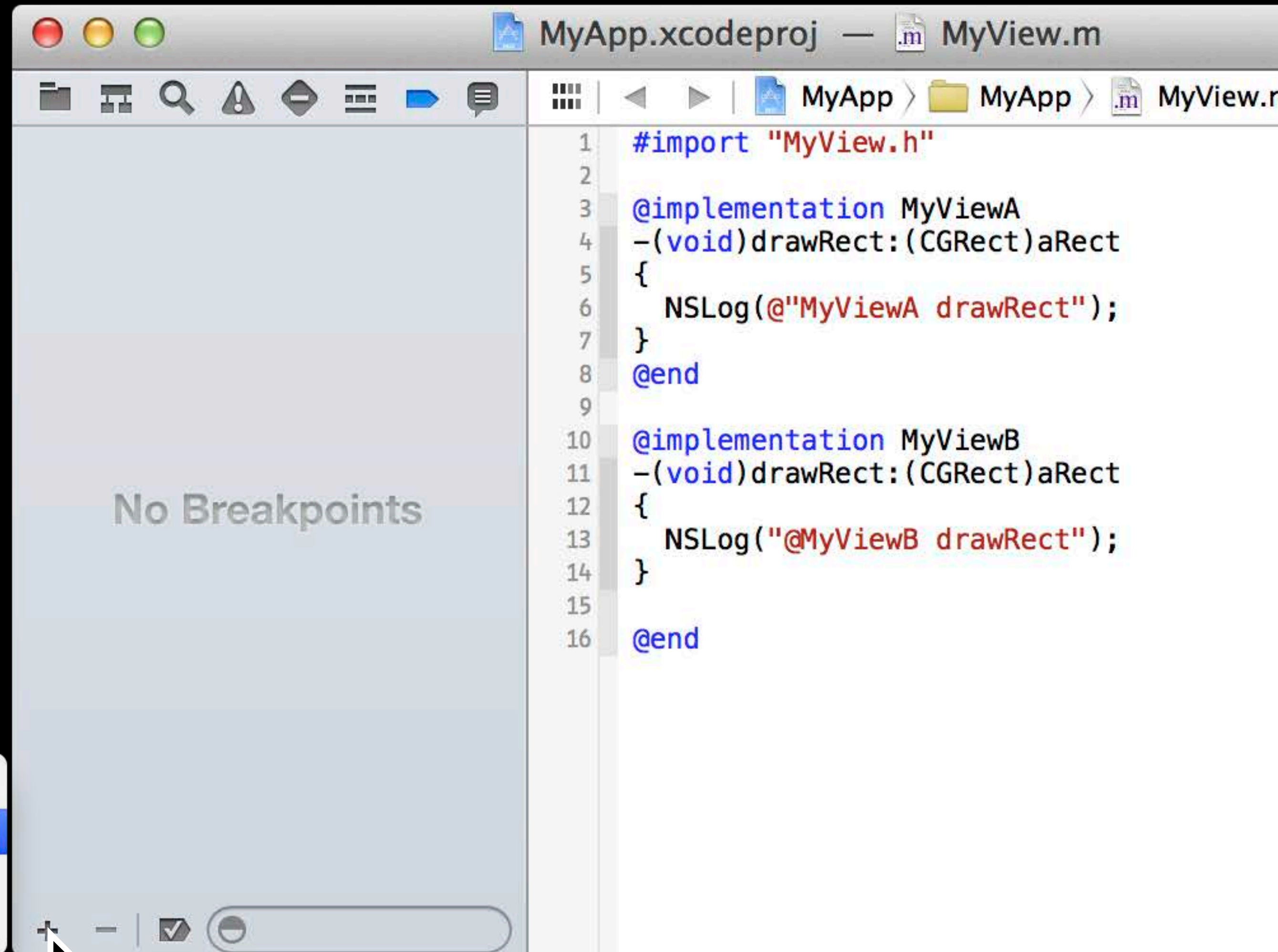
```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```

```
Add Exception Breakpoint...  
Add Symbolic Breakpoint...  
Add OpenGL ES Error Breakpoint...  
Add Test Failure Breakpoint...
```



Common Breakpoint Scenarios

- Stop at a source line:

```
b MyView.m:4
```

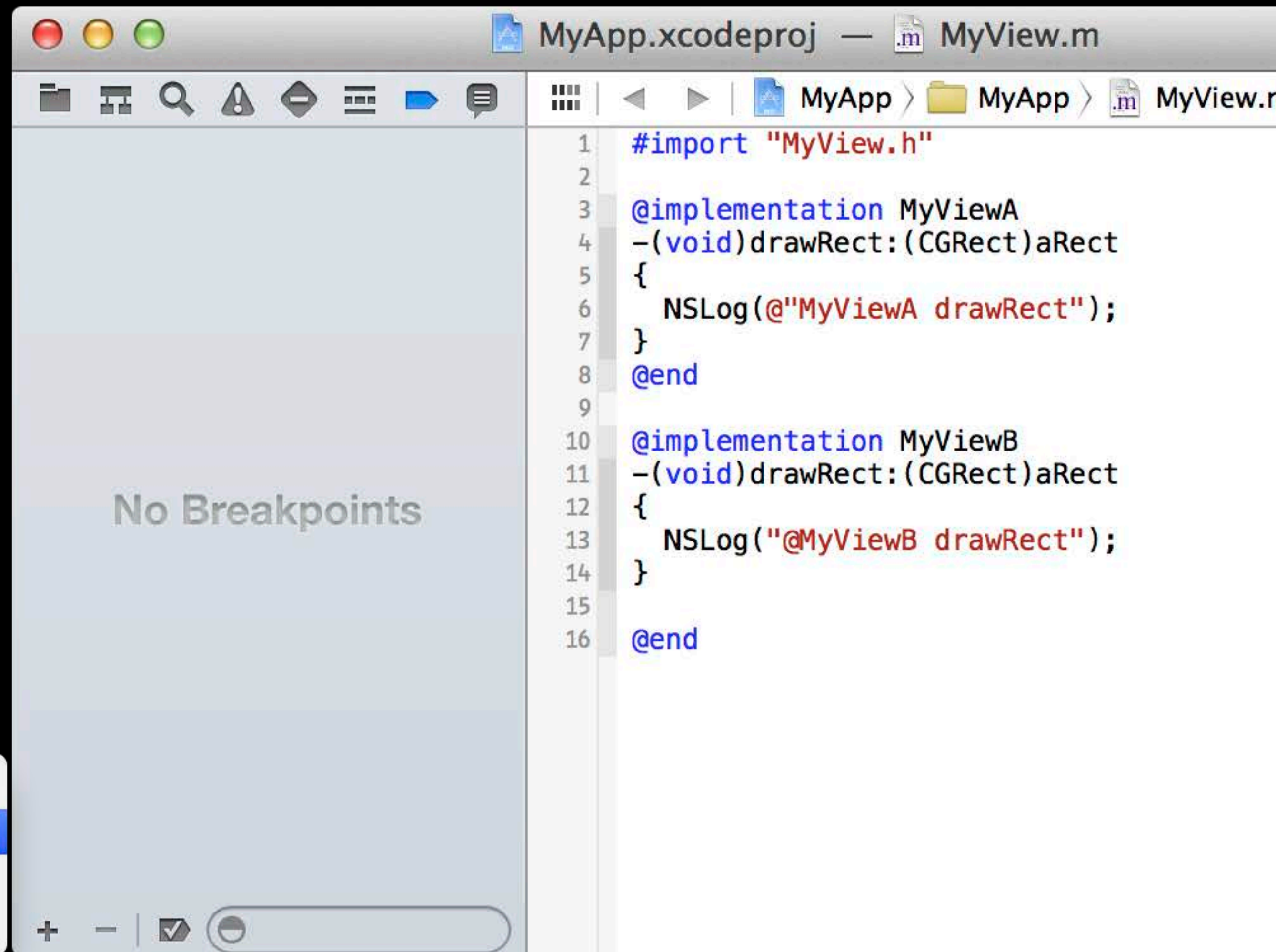
```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```

```
Add Exception Breakpoint...  
Add Symbolic Breakpoint...  
Add OpenGL Error Breakpoint...  
Add Test Failure Breakpoint...
```



Common Breakpoint Scenarios

- Stop at a source line:

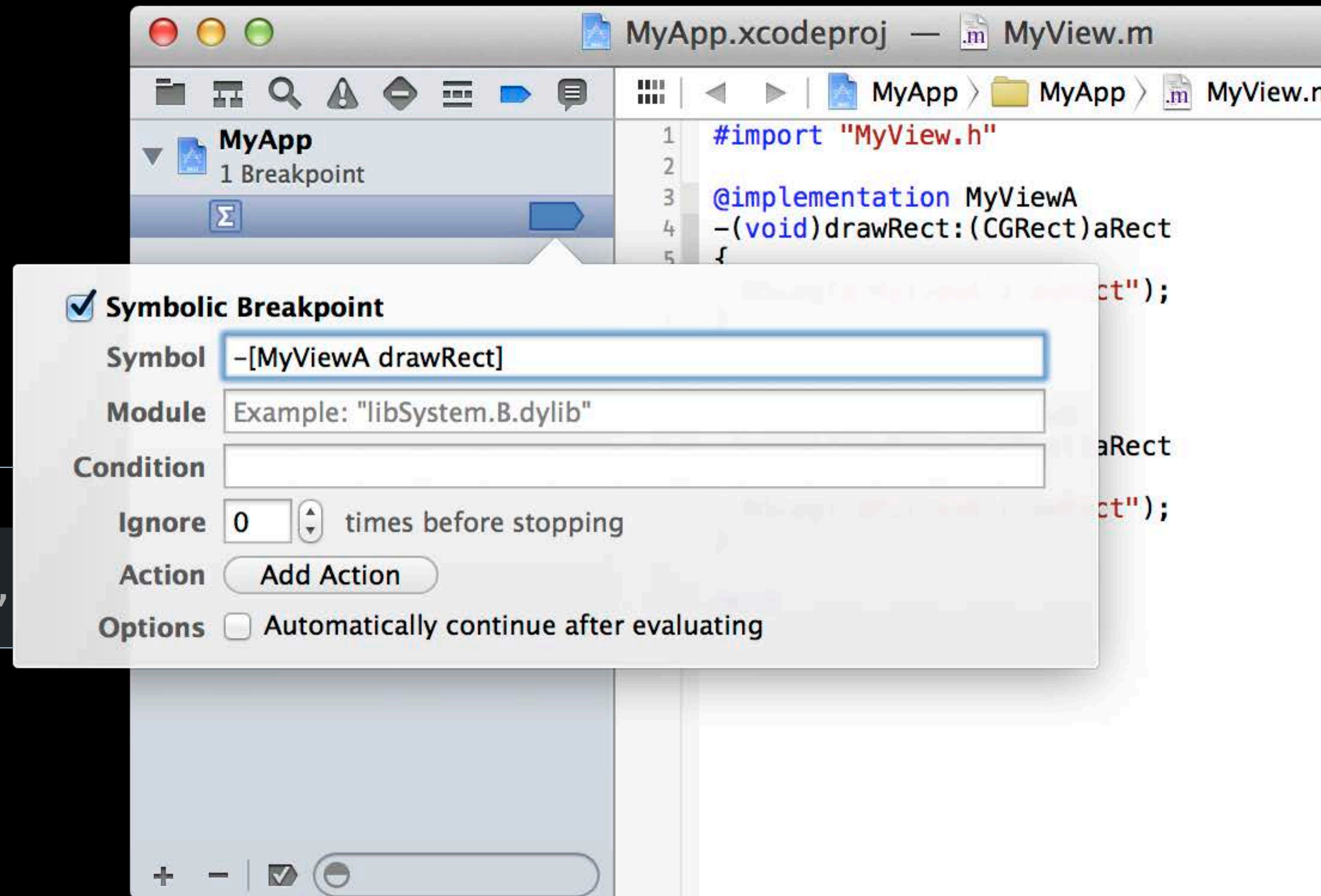
```
b MyView.m:4
```

```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```



Common Breakpoint Scenarios

- Stop at a source line:

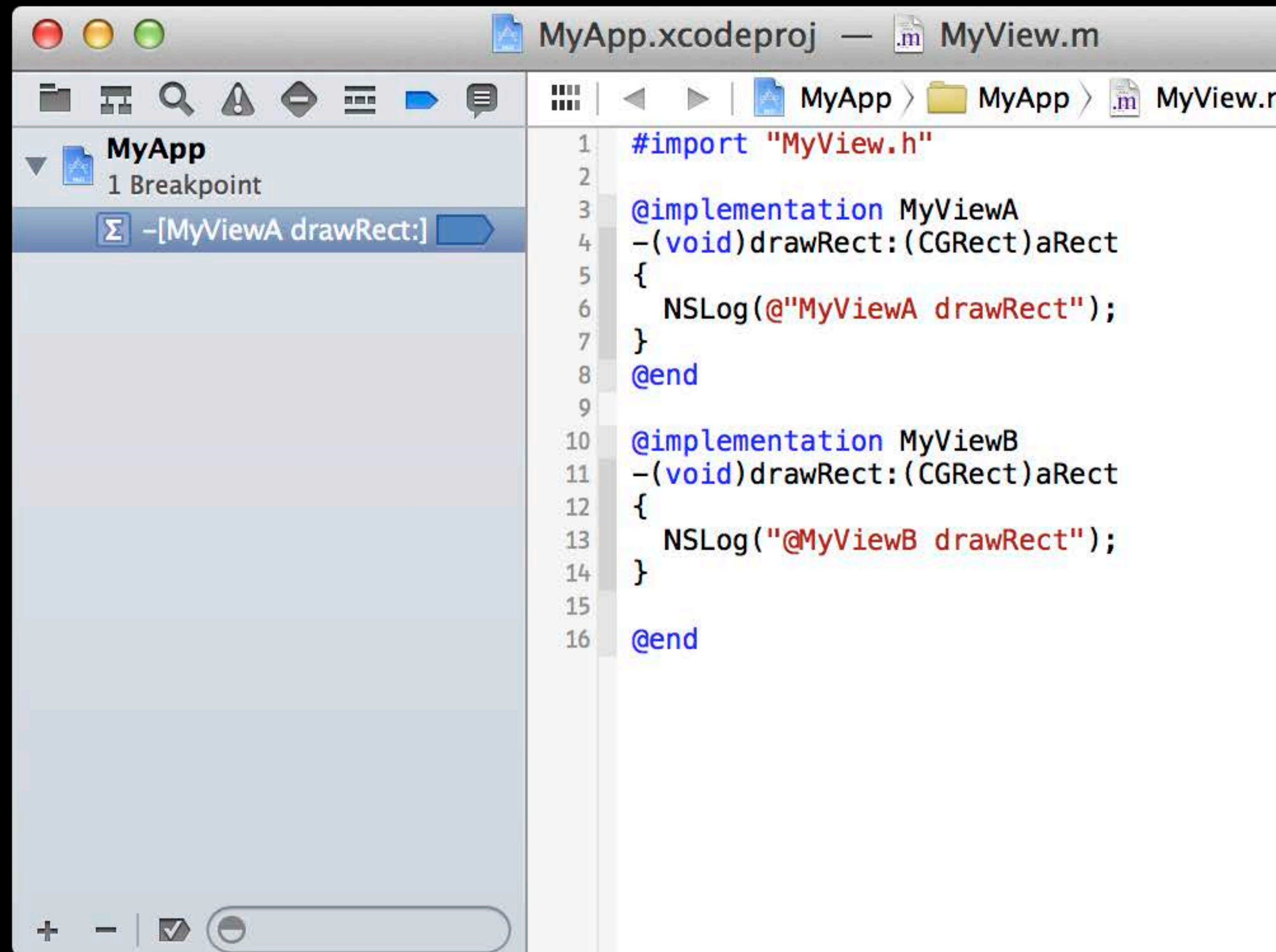
b MyView.m:4

```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

b “-[MyViewA drawRect:]”

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```



Common Breakpoint Scenarios

- Stop at a source line:

```
b MyView.m:4
```

```
breakpoint set  
--file MyView.m --line 4
```

- Stop at a method:

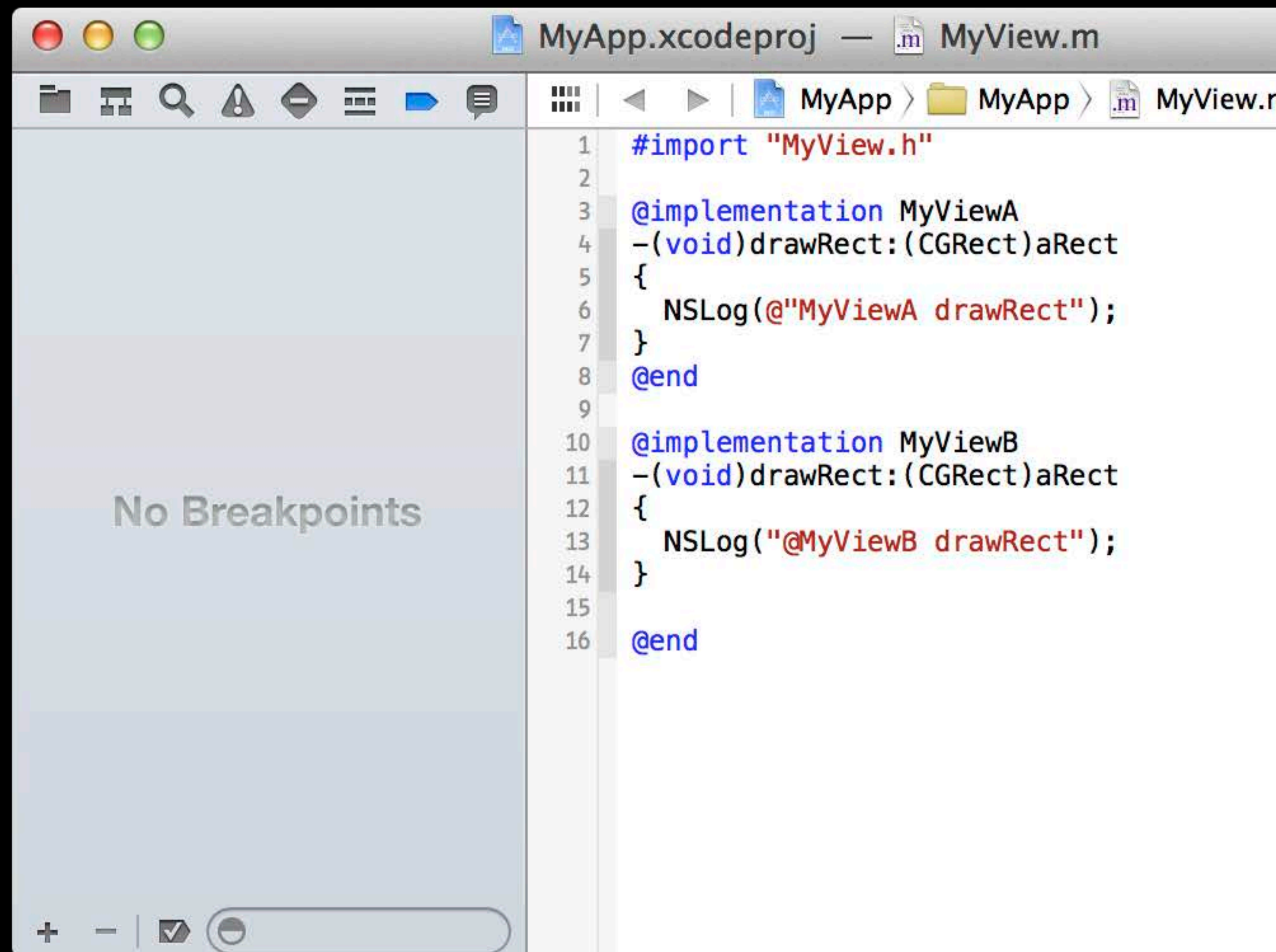
```
b “-[MyViewA drawRect:]”
```

```
breakpoint set  
--name “-[MyViewA drawRect:]”
```

- Stop whenever any object receives a selector:

```
b drawRect:
```

```
breakpoint set  
--selector drawRect:
```

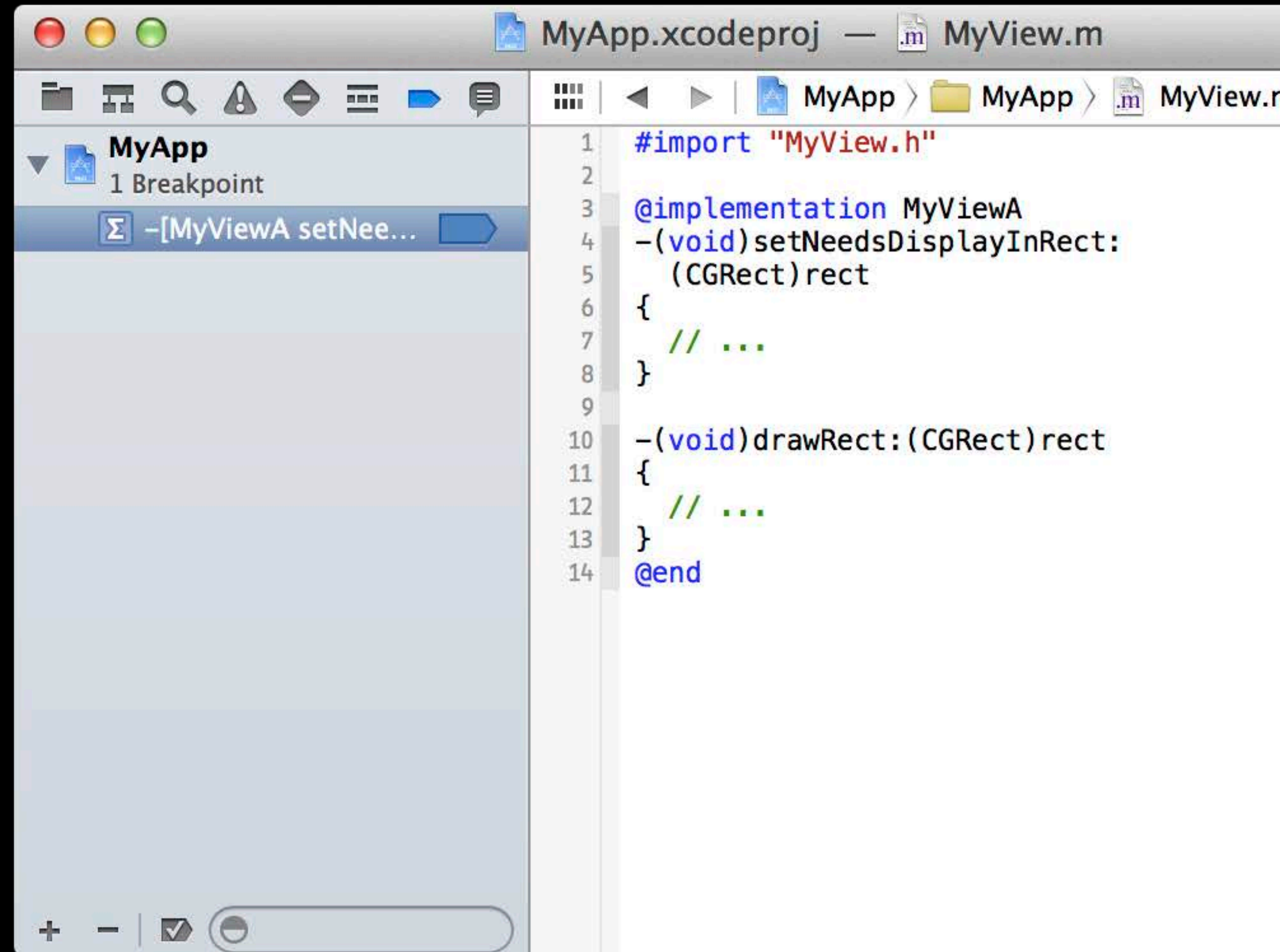


Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

b “[MyViewA
setNeedsDisplayInRect:]”

br co a	breakpoint command add
> p rect	expression rect
> bt	thread backtrace
> c	process continue
> DONE	

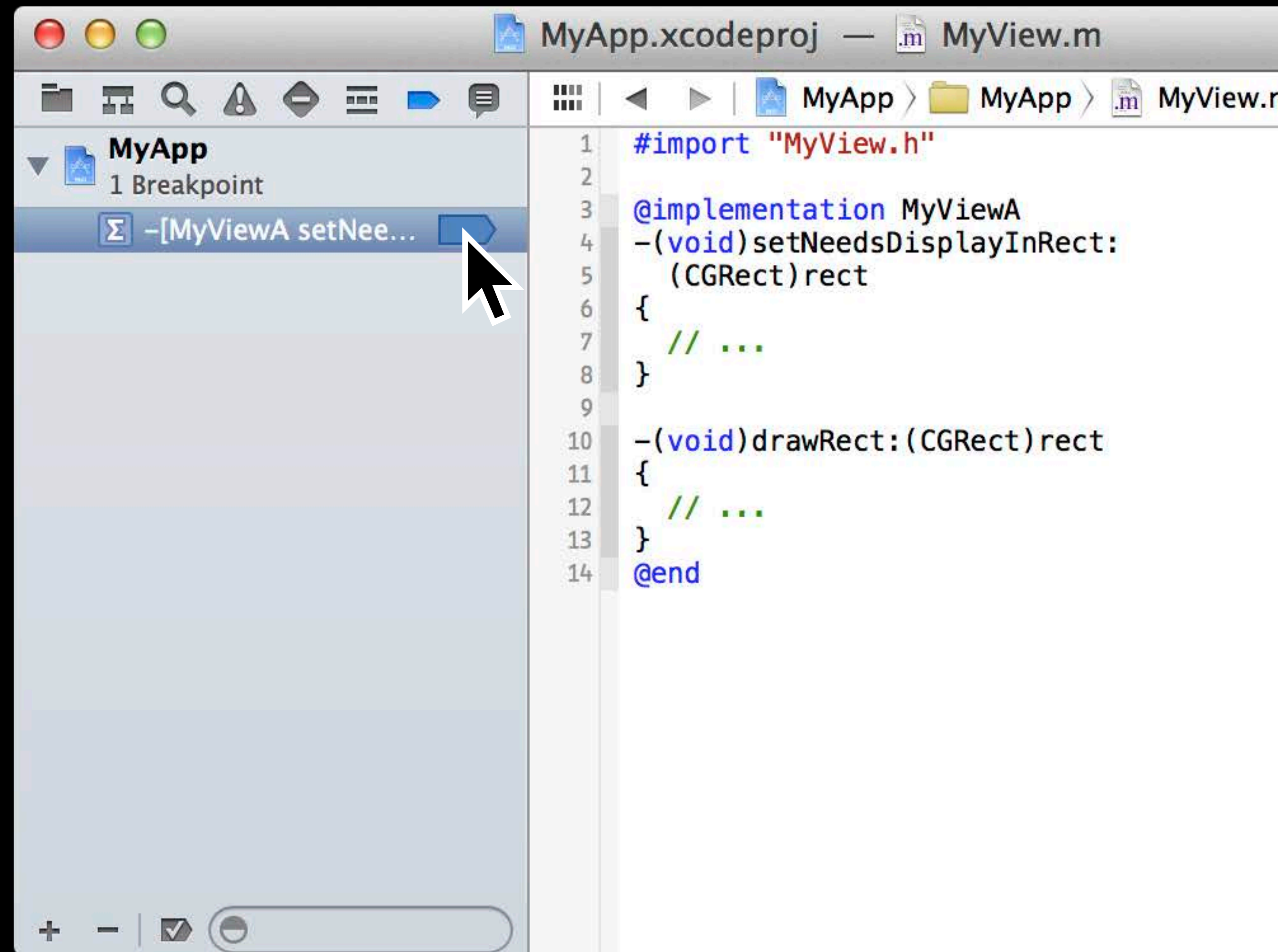


Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

b “[MyViewA
setNeedsDisplayInRect:]”

br co a	breakpoint command add
> p rect	expression rect
> bt	thread backtrace
> c	process continue
> DONE	

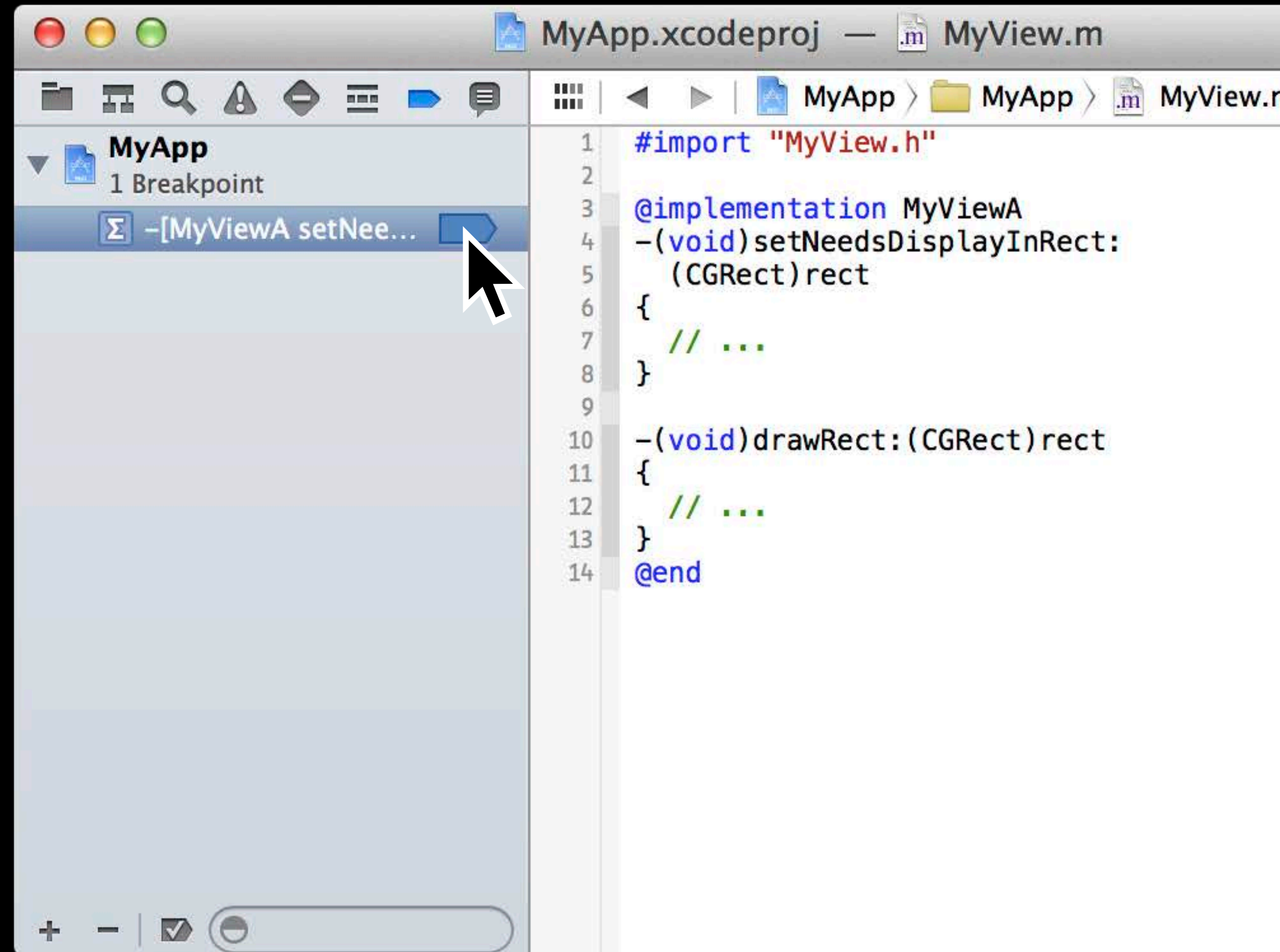


Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

b “[MyViewA
setNeedsDisplayInRect:]”

br co a	breakpoint command add
> p rect	expression rect
> bt	thread backtrace
> c	process continue
> DONE	

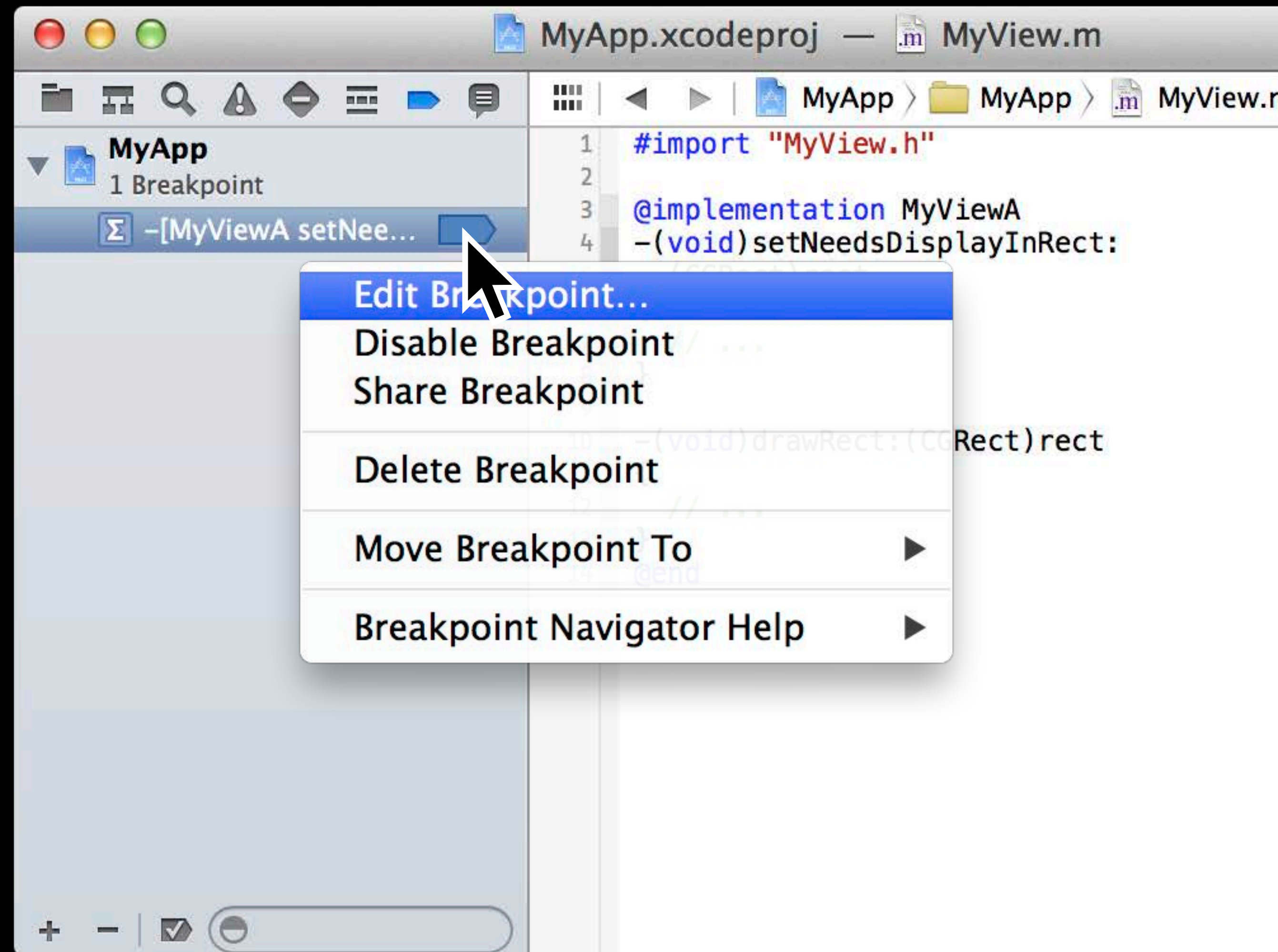


Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

```
b "[MyViewA  
setNeedsDisplayInRect:]"
```

br co a	breakpoint command add
> p rect	expression rect
> bt	thread backtrace
> c	process continue
> DONE	

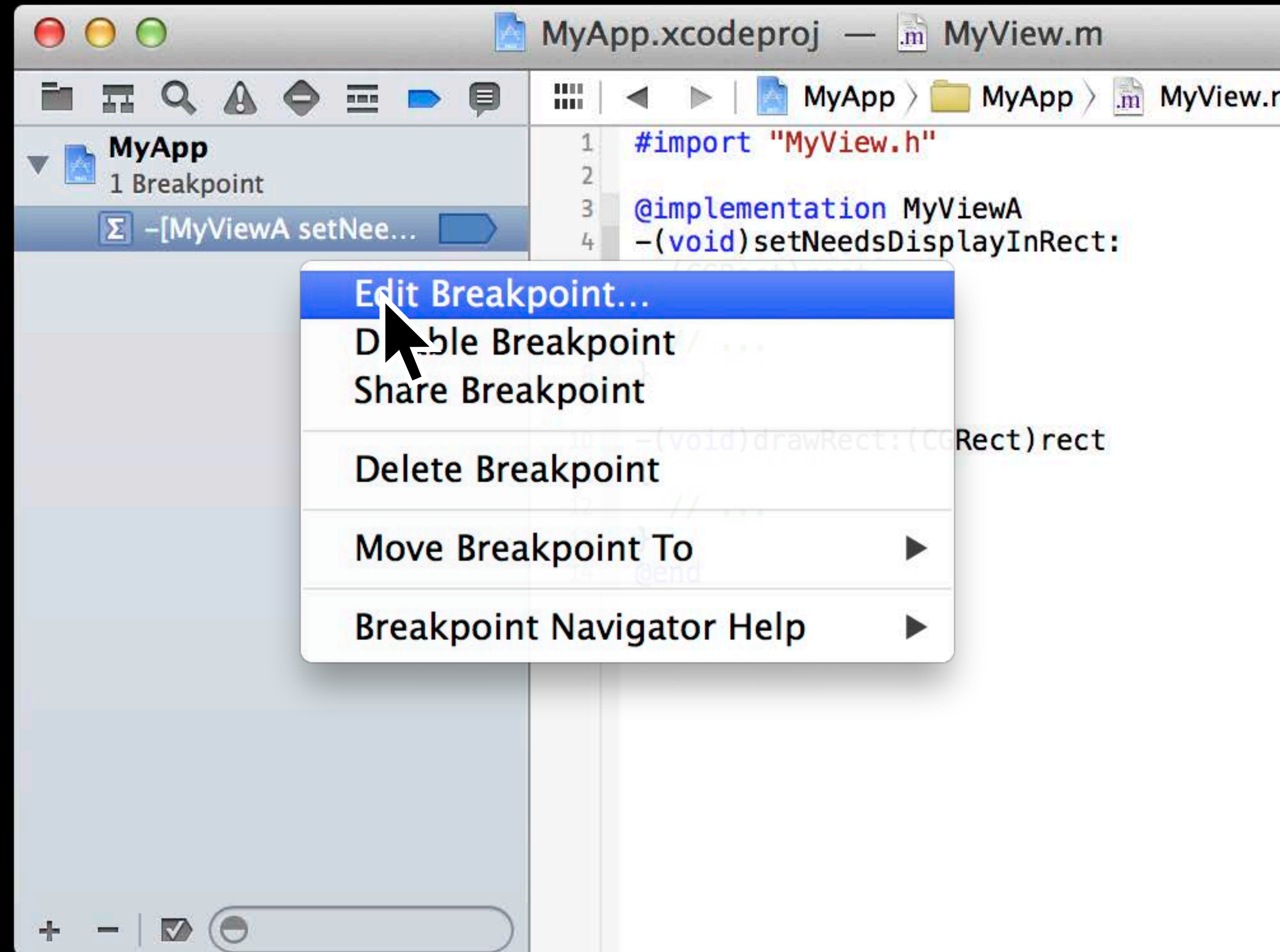


Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

```
b "[MyViewA  
setNeedsDisplayInRect:]"
```

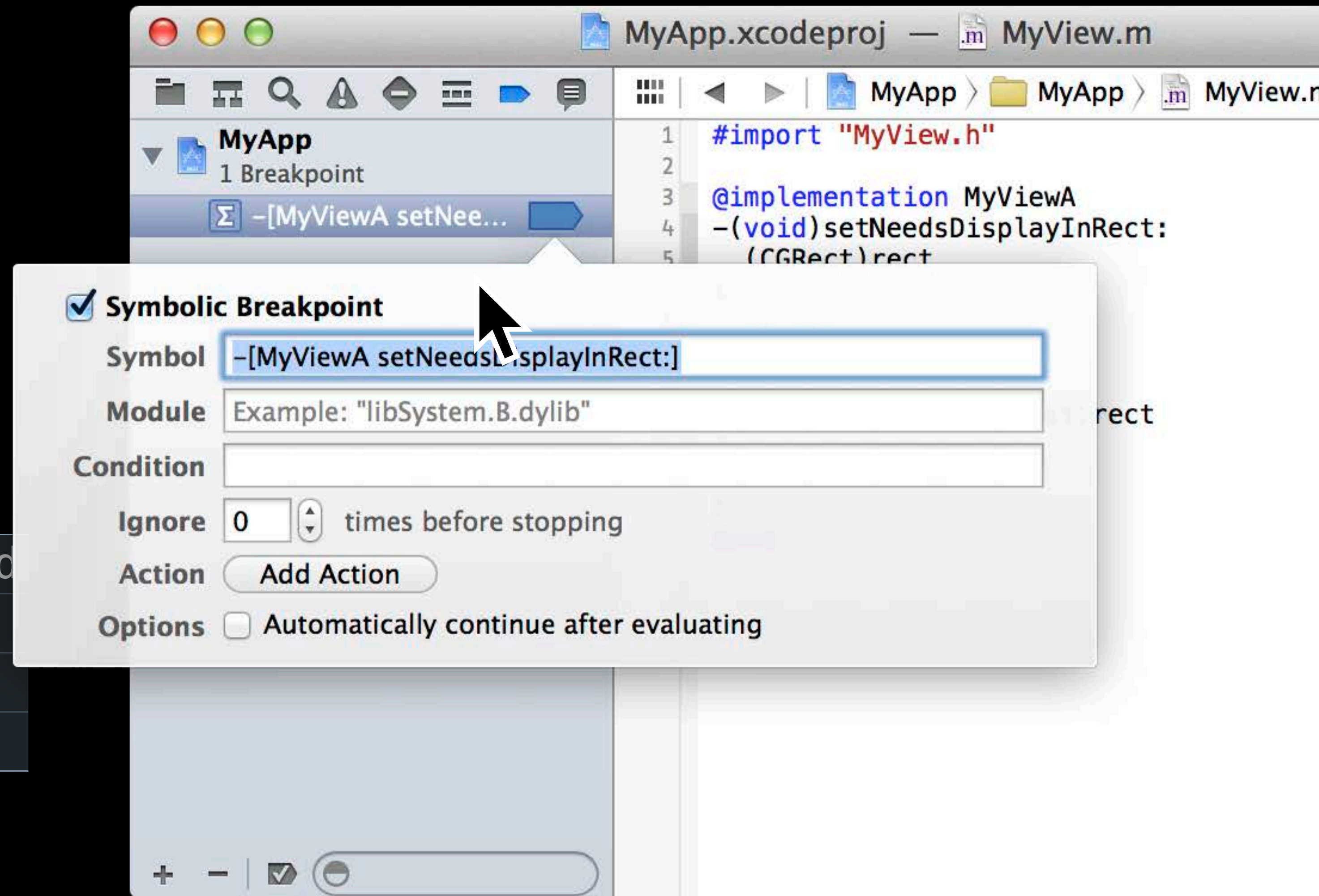
br co a	breakpoint command add
> p rect	expression rect
> bt	thread backtrace
> c	process continue
> DONE	



Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

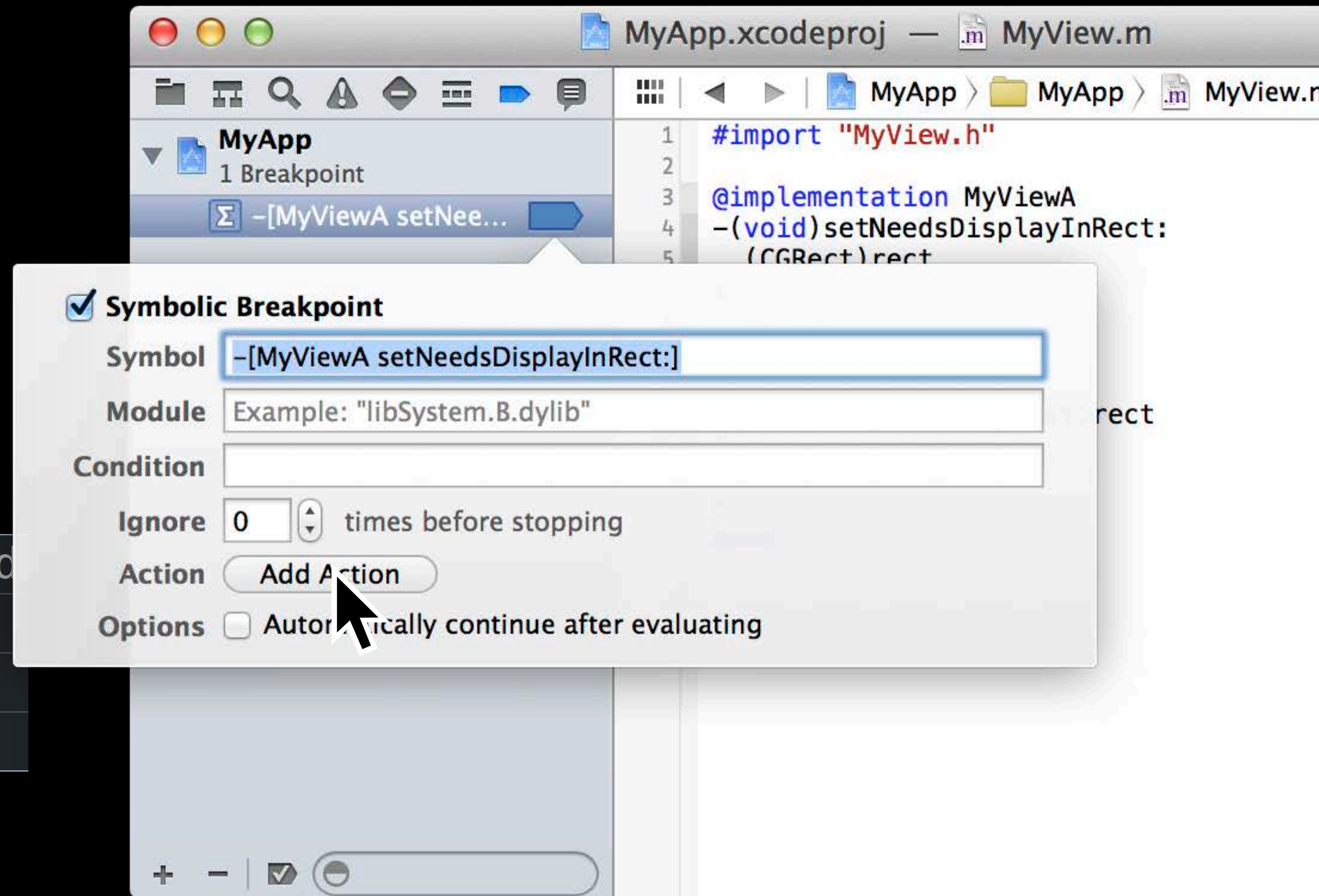
```
b "[MyViewA  
setNeedsDisplayInRect:]"  
br co a breakpoint command add  
> p rect expression rect  
> bt thread backtrace  
> c process continue  
> DONE
```



Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

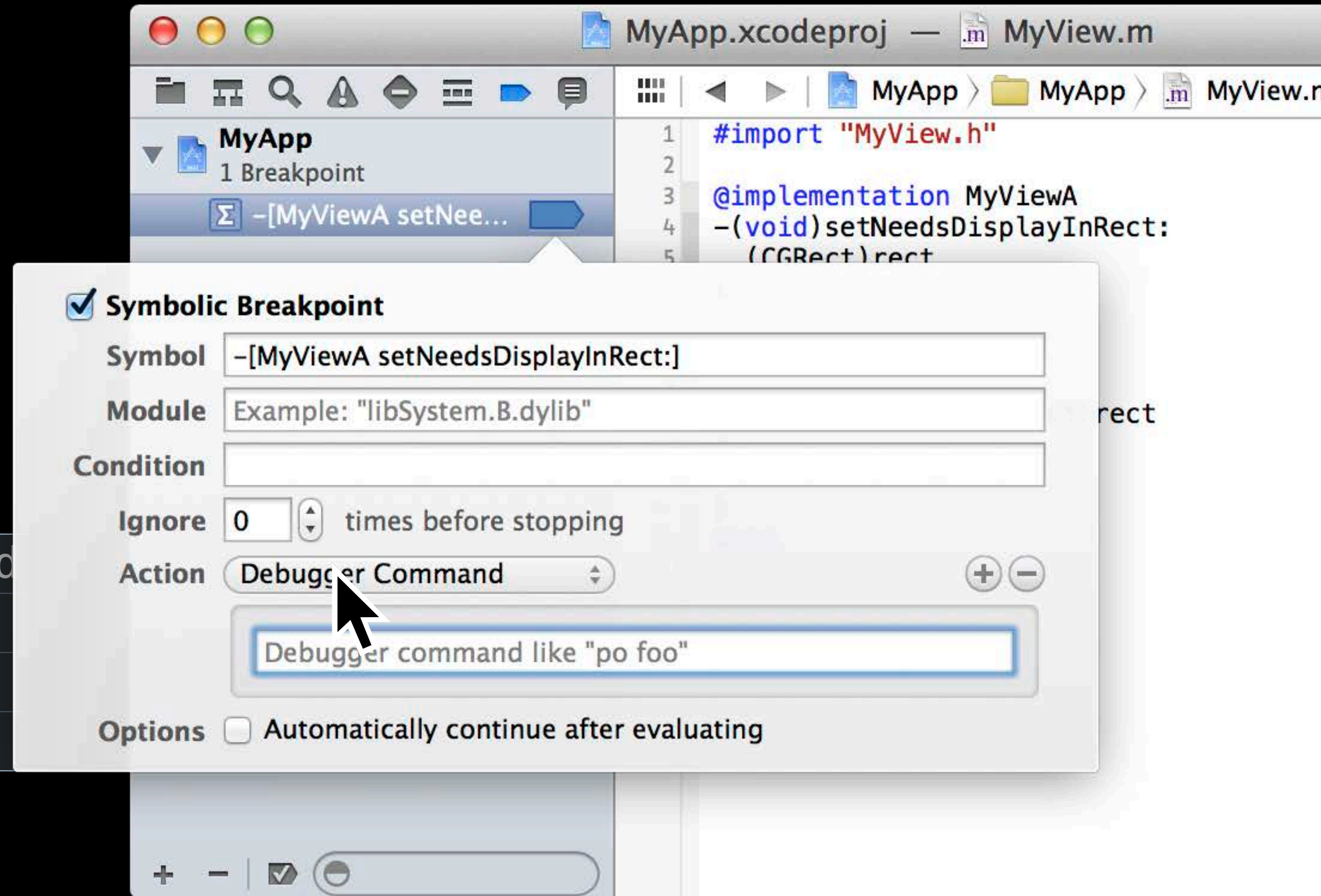
```
b "[MyViewA  
setNeedsDisplayInRect:]"  
br co a breakpoint command add  
> p rect expression rect  
> bt thread backtrace  
> c process continue  
> DONE
```



Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

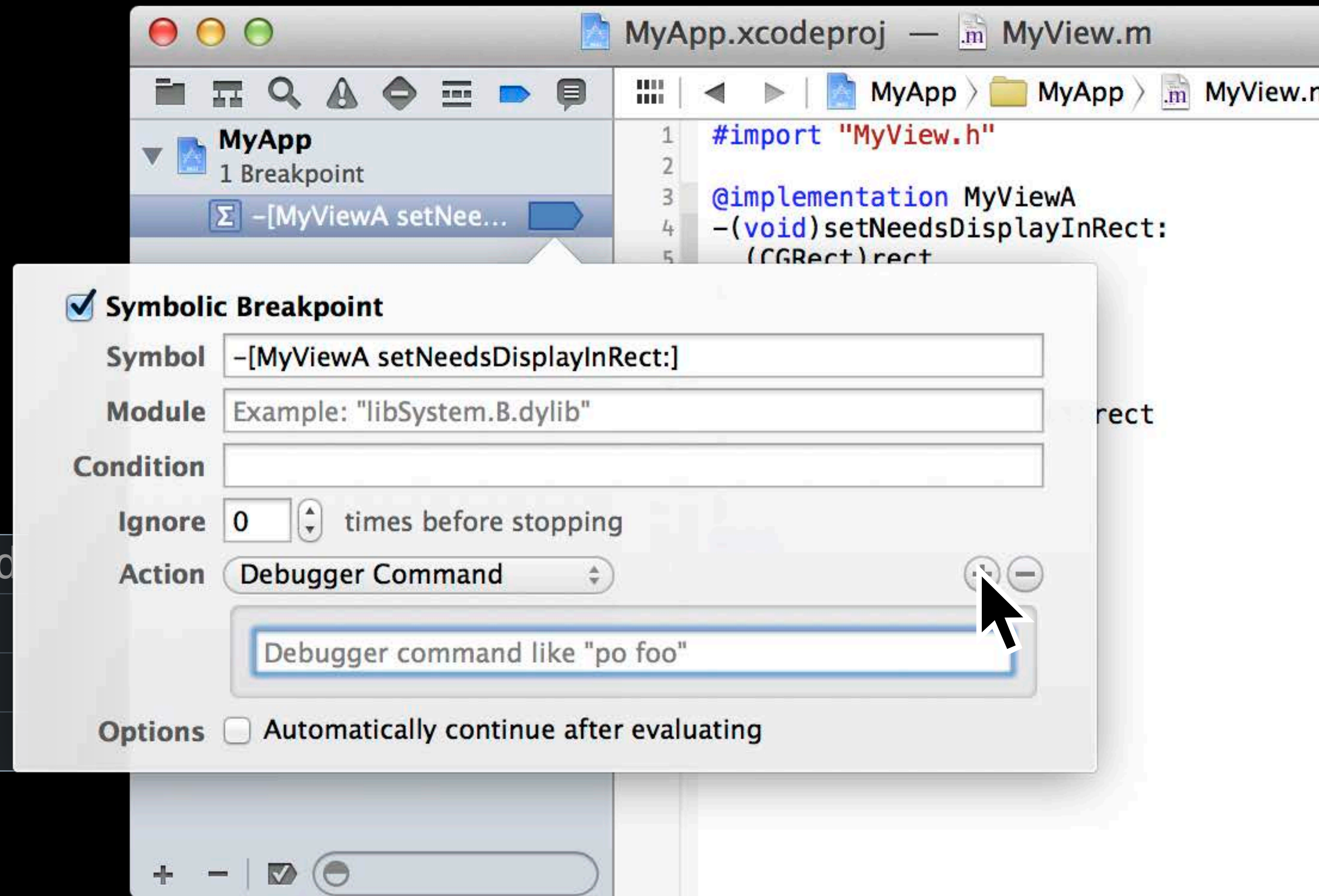
```
b "[MyViewA  
setNeedsDisplayInRect:]"  
br co a breakpoint command add  
> p rect expression rect  
> bt thread backtrace  
> c process continue  
> DONE
```



Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

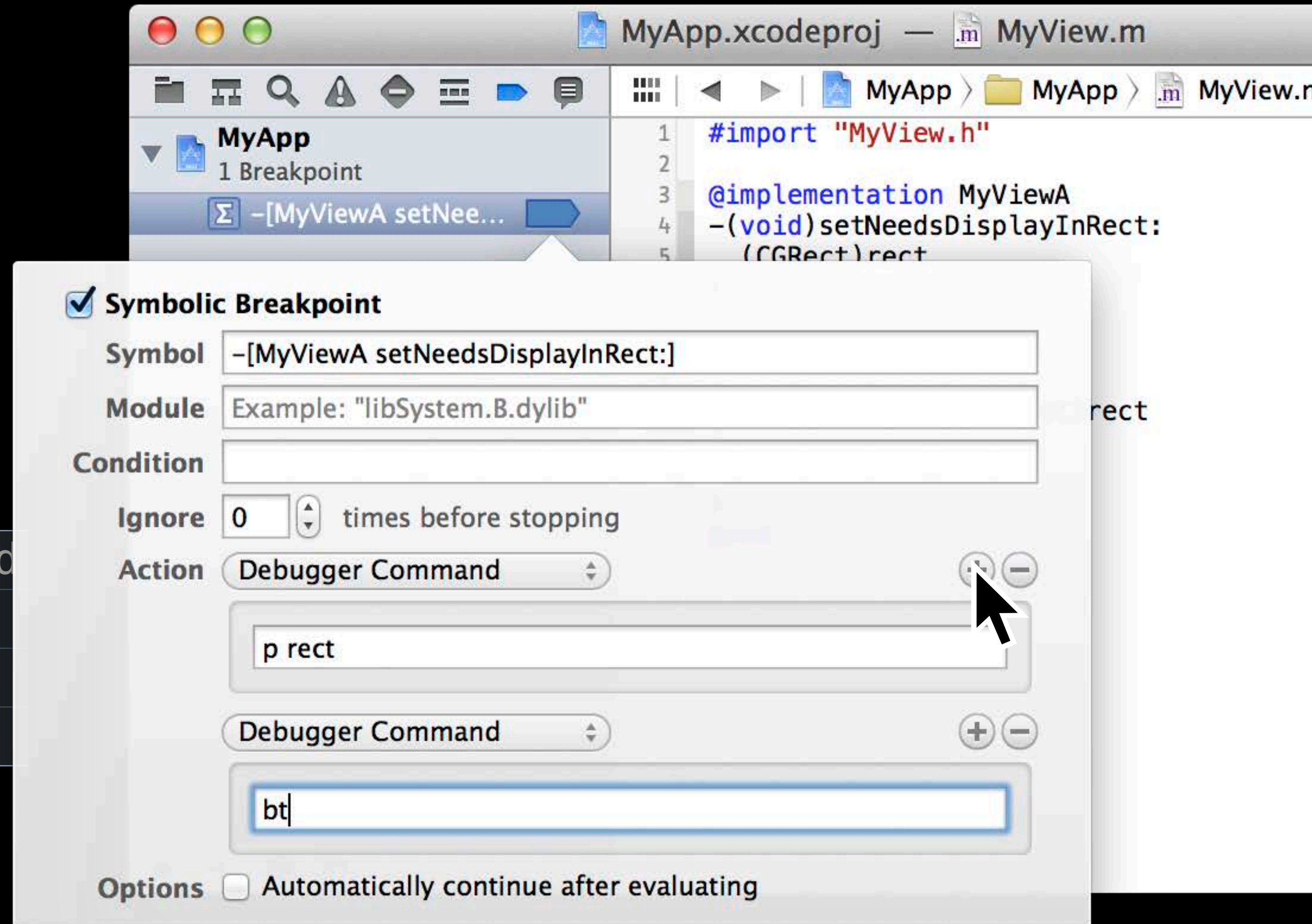
```
b "[MyViewA  
setNeedsDisplayInRect:]"  
br co a breakpoint command add  
> p rect expression rect  
> bt thread backtrace  
> c process continue  
> DONE
```



Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

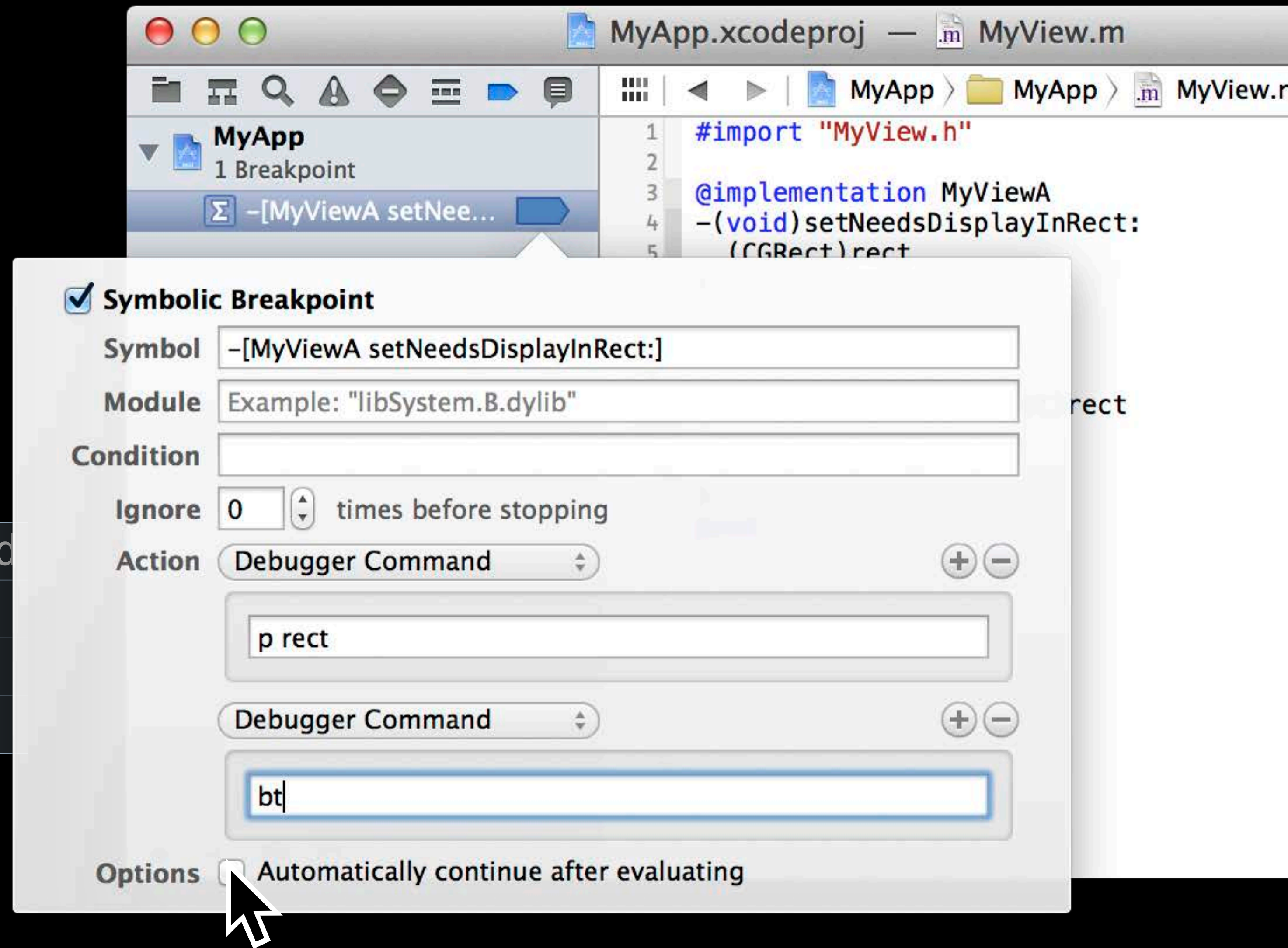
```
b "[MyViewA  
setNeedsDisplayInRect:]"  
br co a breakpoint command add  
> p rect expression rect  
> bt thread backtrace  
> c process continue  
> DONE
```



Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

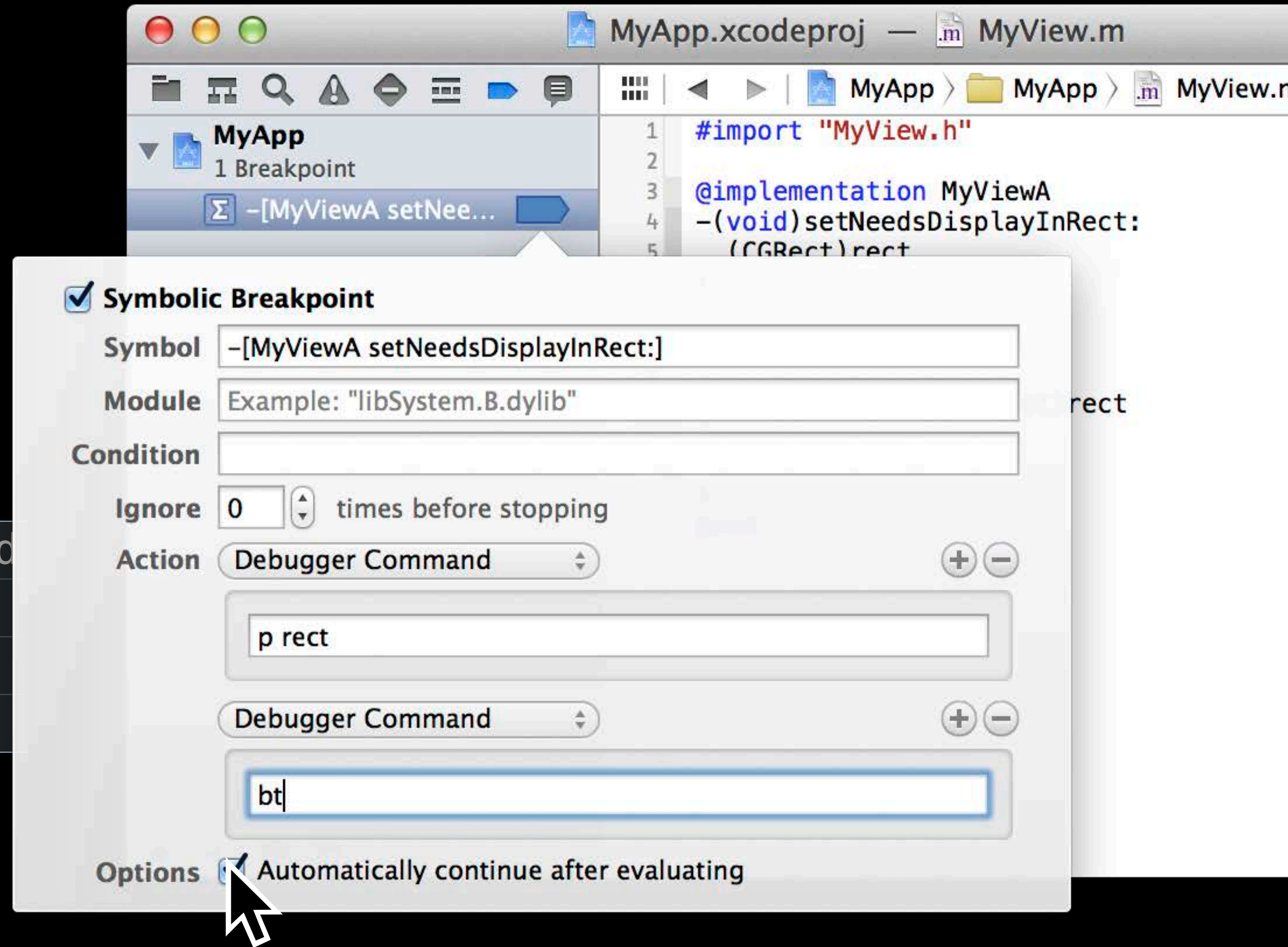
```
b "[MyViewA  
setNeedsDisplayInRect:]"  
br co a breakpoint command add  
> p rect expression rect  
> bt thread backtrace  
> c process continue  
> DONE
```



Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit

```
b "[MyViewA  
setNeedsDisplayInRect:]"  
br co a breakpoint command add  
> p rect expression rect  
> bt thread backtrace  
> c process continue  
> DONE
```



Conditions Focus on Specific Objects

- Use if breakpoints fire too frequently
- Find when a method is called on a specific instance

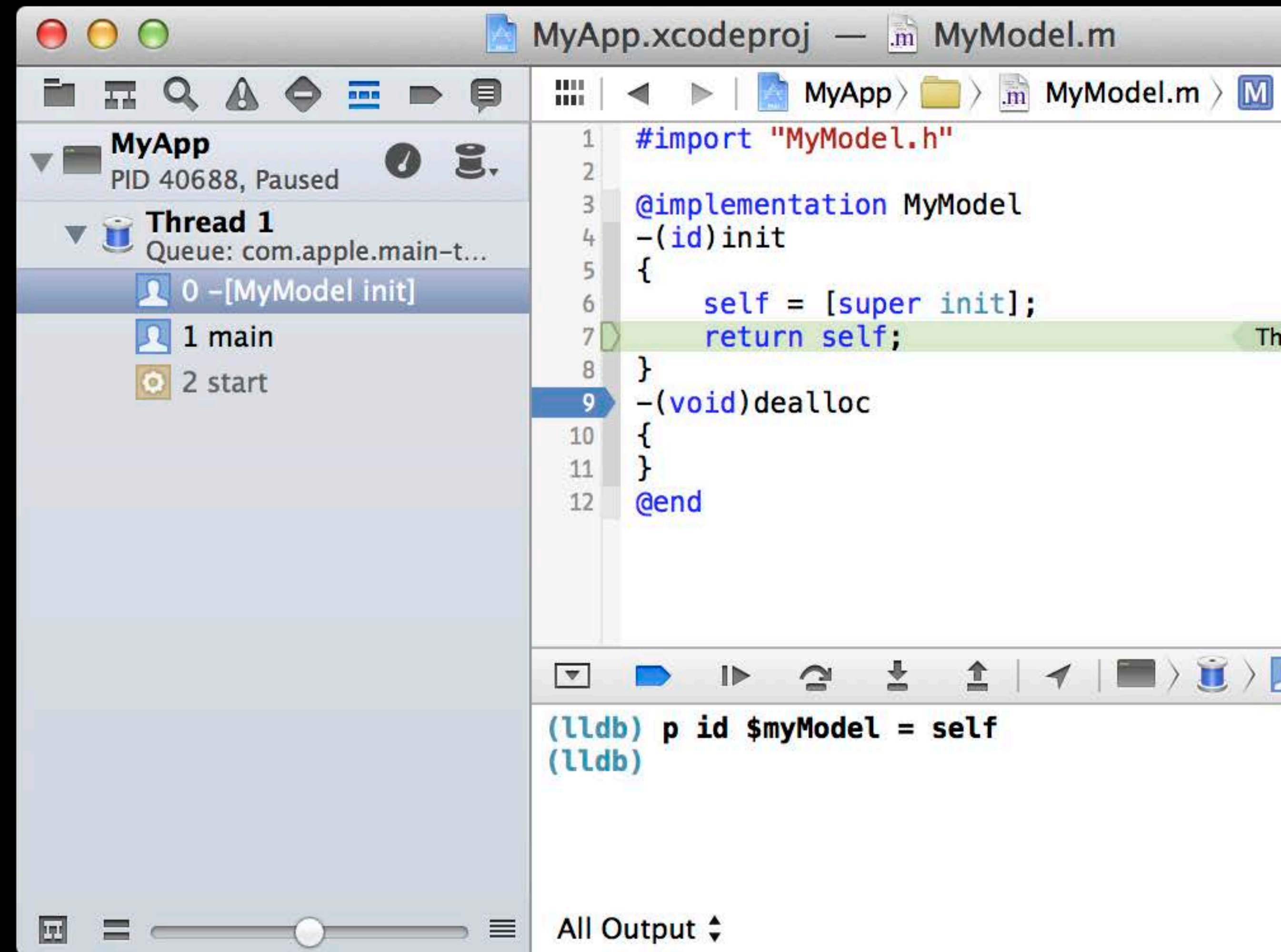
```
p id $myModel = self
```

```
Creates a persistent variable of type id  
expression id $myModel = self
```

```
b “-[MyModel dealloc]”
```

```
br m -c “self == $myModel”
```

```
breakpoint modify  
--condition “self == $myModel”
```



Conditions Focus on Specific Objects

- Use if breakpoints fire too frequently
- Find when a method is called on a specific instance

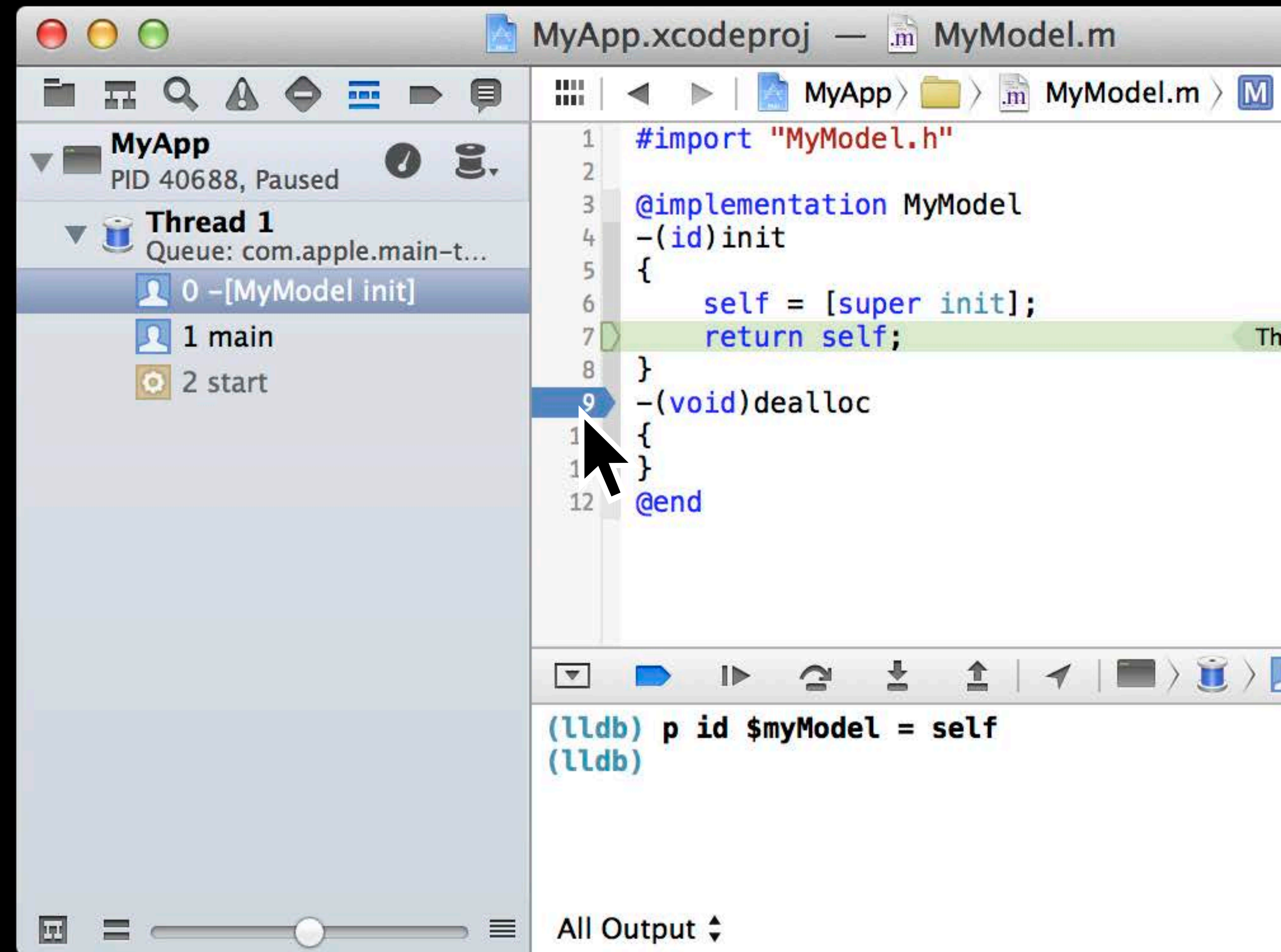
```
p id $myModel = self
```

```
Creates a persistent variable of type id  
expression id $myModel = self
```

```
b "-[MyModel dealloc]"
```

```
br m -c "self == $myModel"
```

```
breakpoint modify  
--condition "self == $myModel"
```



Conditions Focus on Specific Objects

- Use if breakpoints fire too frequently
- Find when a method is called on a specific instance

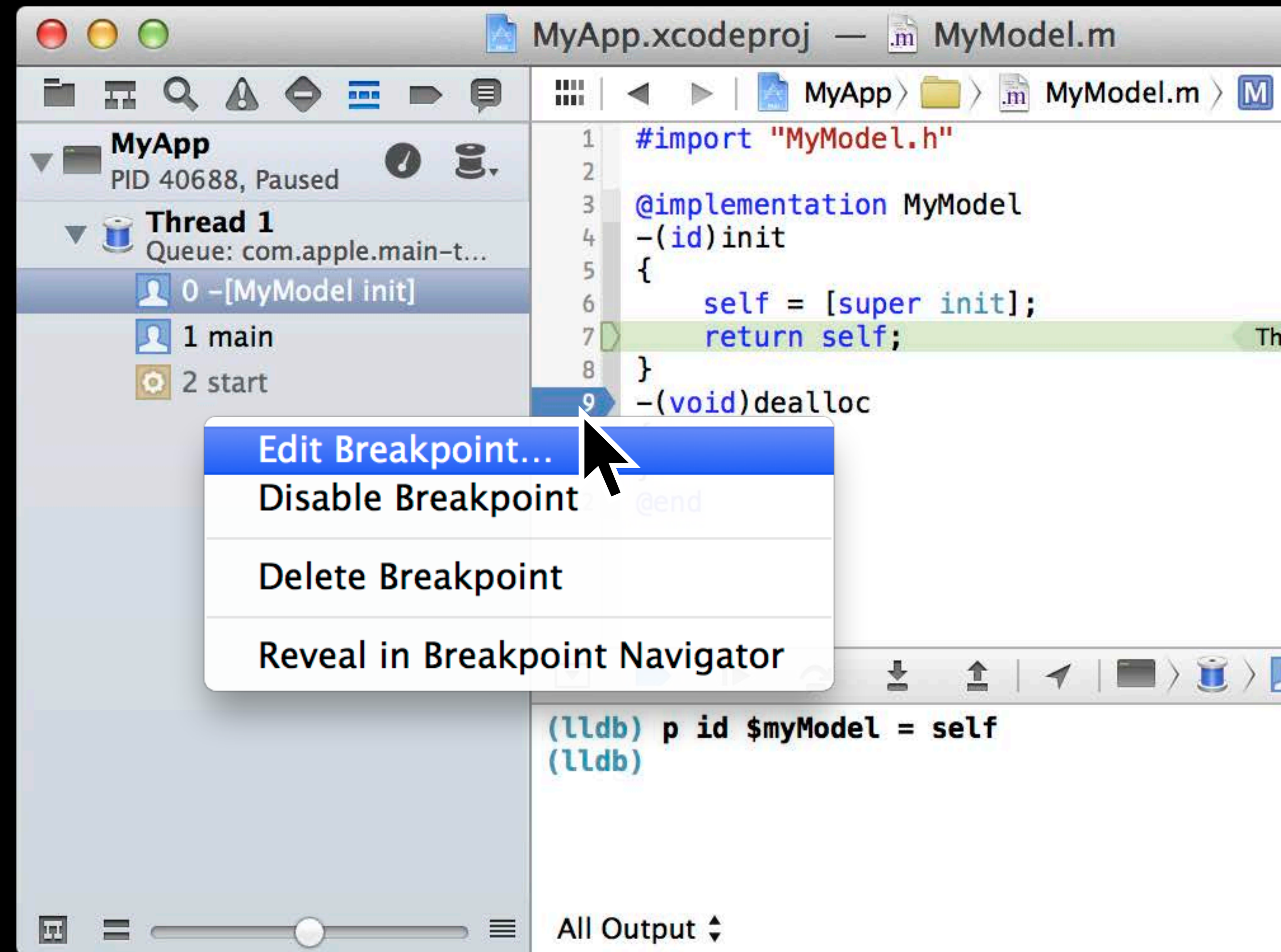
```
p id $myModel = self
```

```
Creates a persistent variable of type id  
expression id $myModel = self
```

```
b "-[MyModel dealloc]"
```

```
br m -c "self == $myModel"
```

```
breakpoint modify  
--condition "self == $myModel"
```



Conditions Focus on Specific Objects

- Use if breakpoints fire too frequently
- Find when a method is called on a specific instance

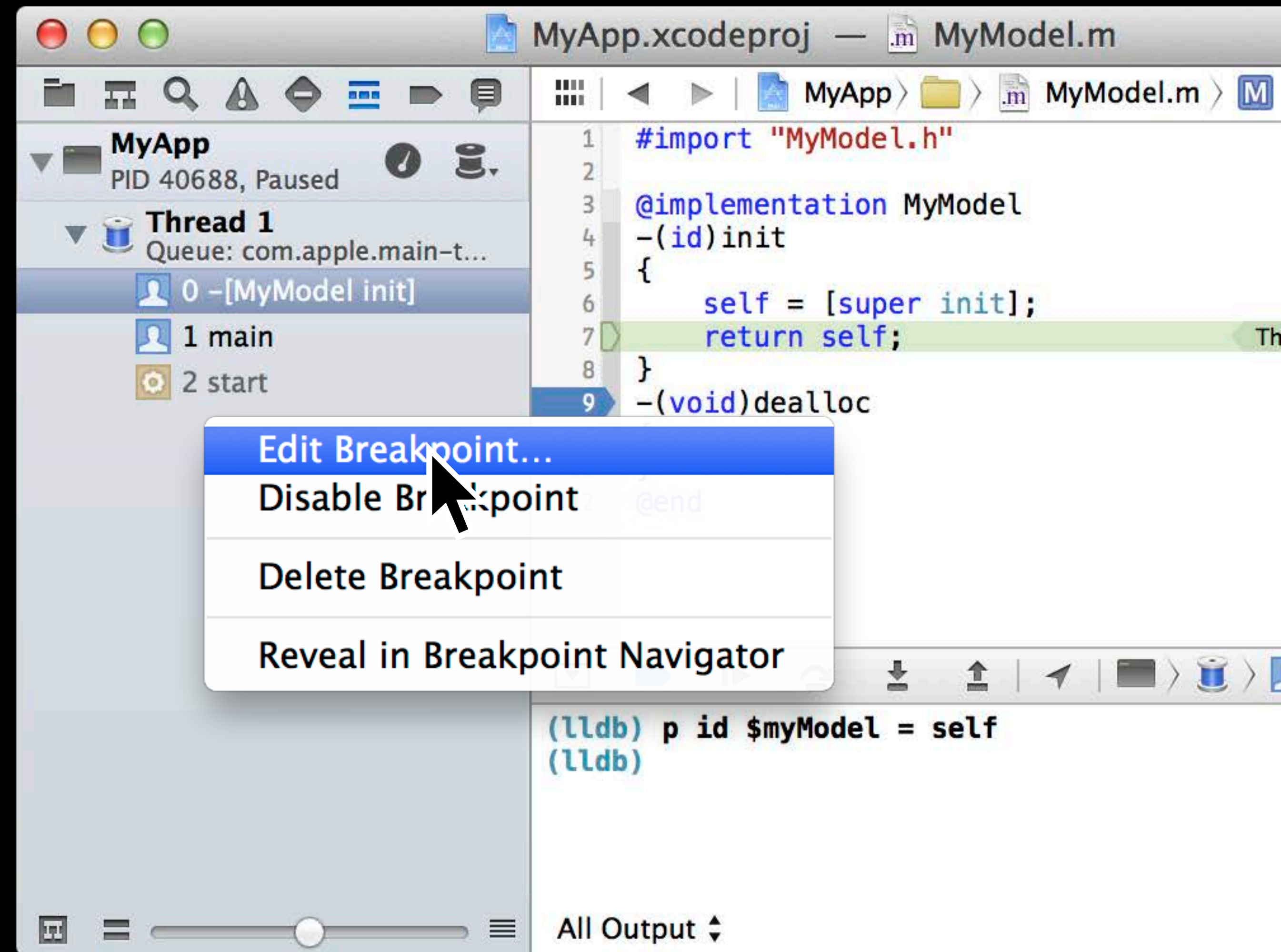
```
p id $myModel = self
```

```
Creates a persistent variable of type id  
expression id $myModel = self
```

```
b “-[MyModel dealloc]”
```

```
br m -c “self == $myModel”
```

```
breakpoint modify  
--condition “self == $myModel”
```



Conditions Focus on Specific Objects

- Use if breakpoints fire too frequently
- Find when a method is called on a specific instance

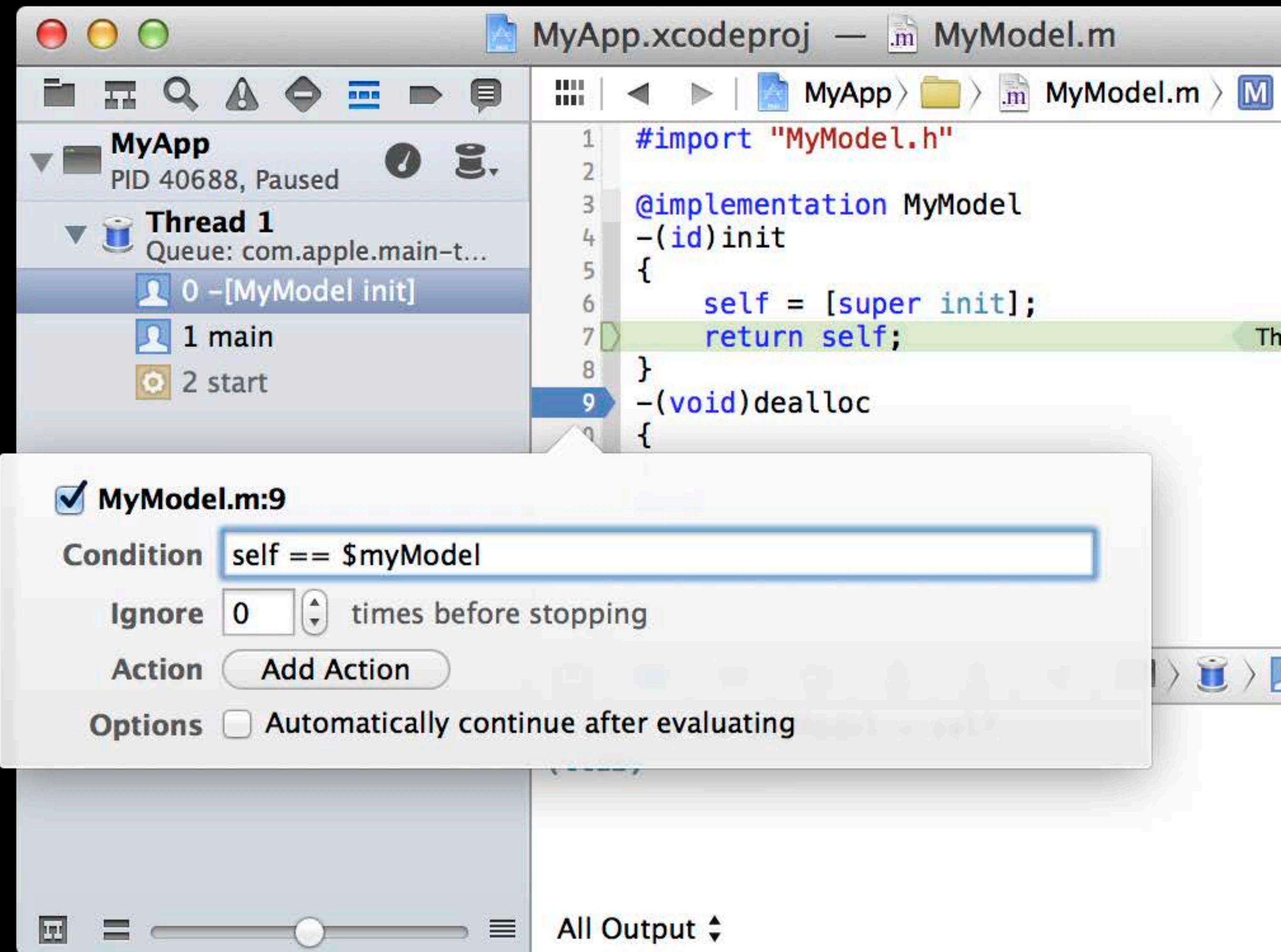
```
p id $myModel = self
```

```
Creates a persistent variable of type id  
expression id $myModel = self
```

```
b “-[MyModel dealloc]”
```

```
br m -c “self == $myModel”
```

```
breakpoint modify  
--condition “self == $myModel”
```



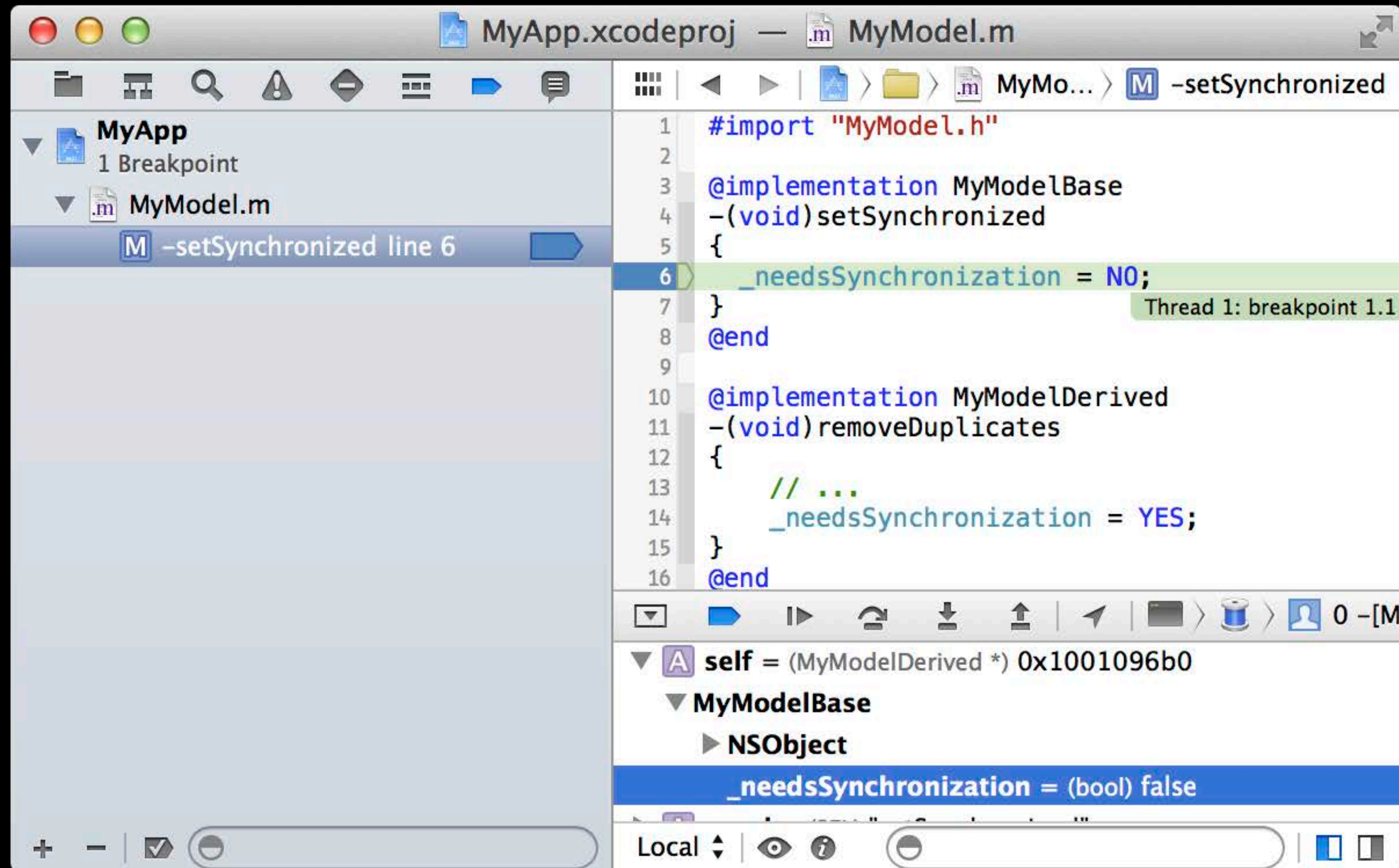
Focus on Memory with Watchpoints

- Someone is changing a value, but all you know is its location
- Watchpoints pause the program if the value is accessed

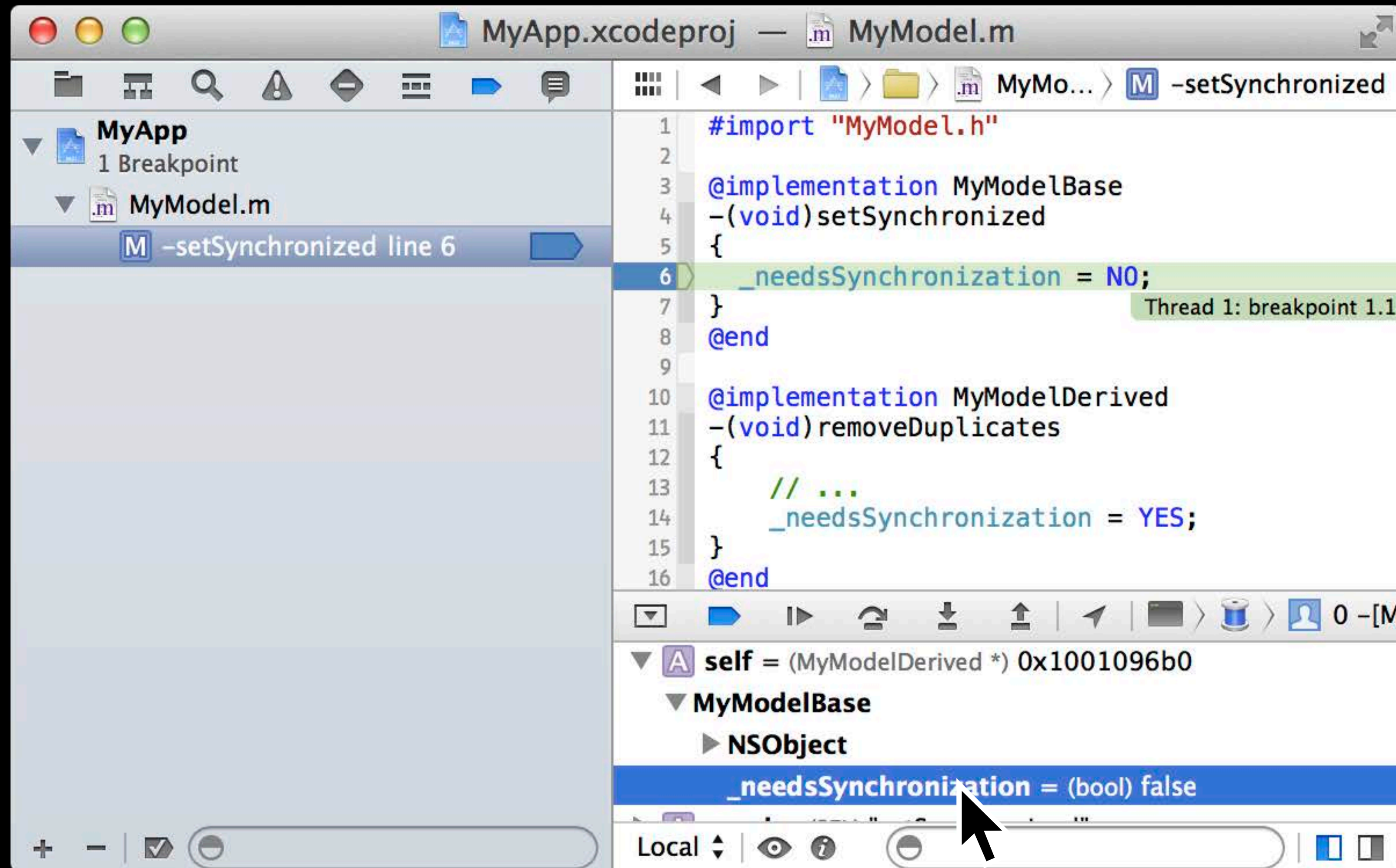
```
w s v self->_needsSynchronization  
watchpoint set variable  
self->_needsSynchronization
```

- Watchpoint resources are limited by CPU
 - 4 on Intel
 - 2 on ARM

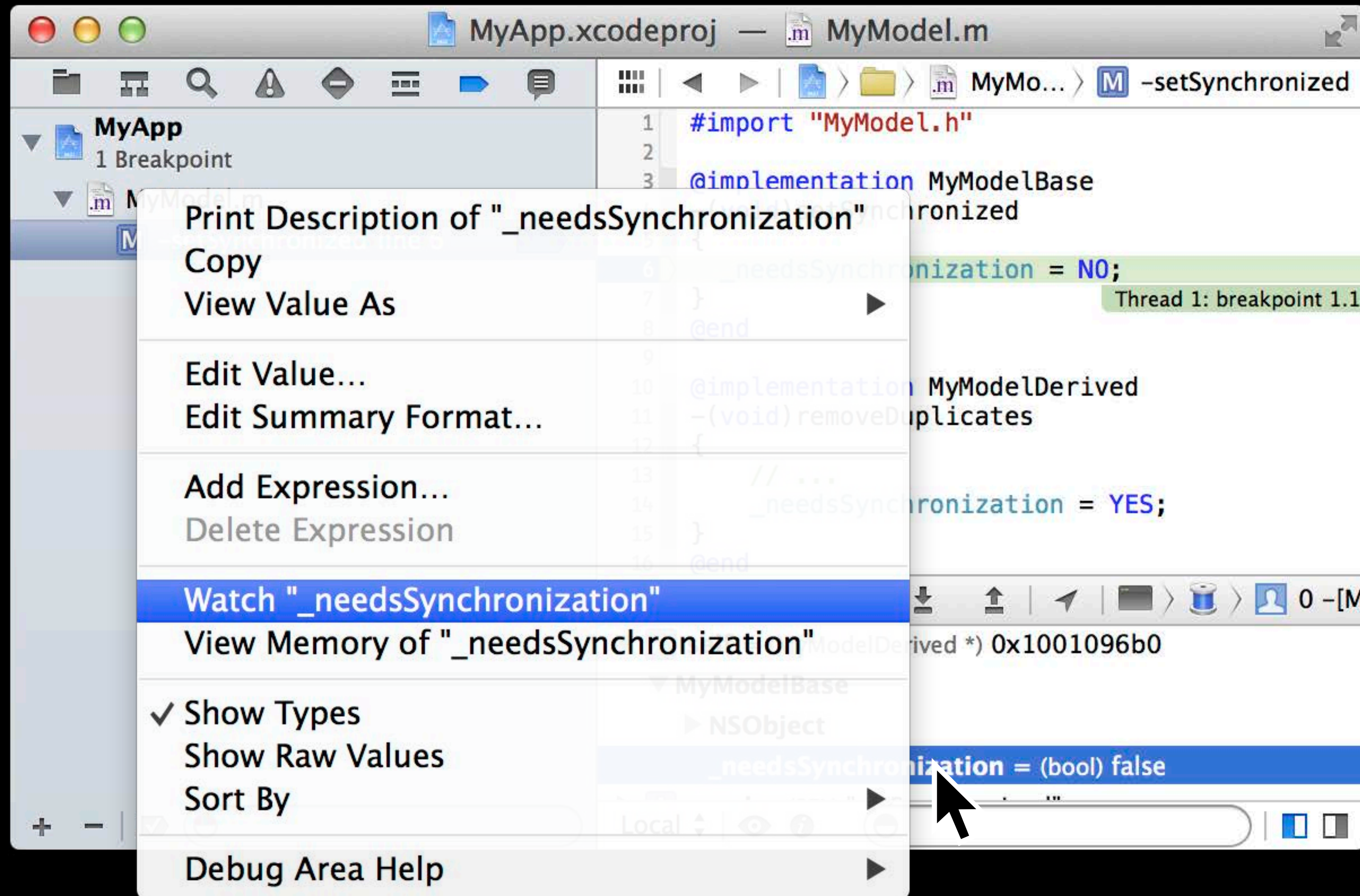
Focus on Memory with Watchpoints



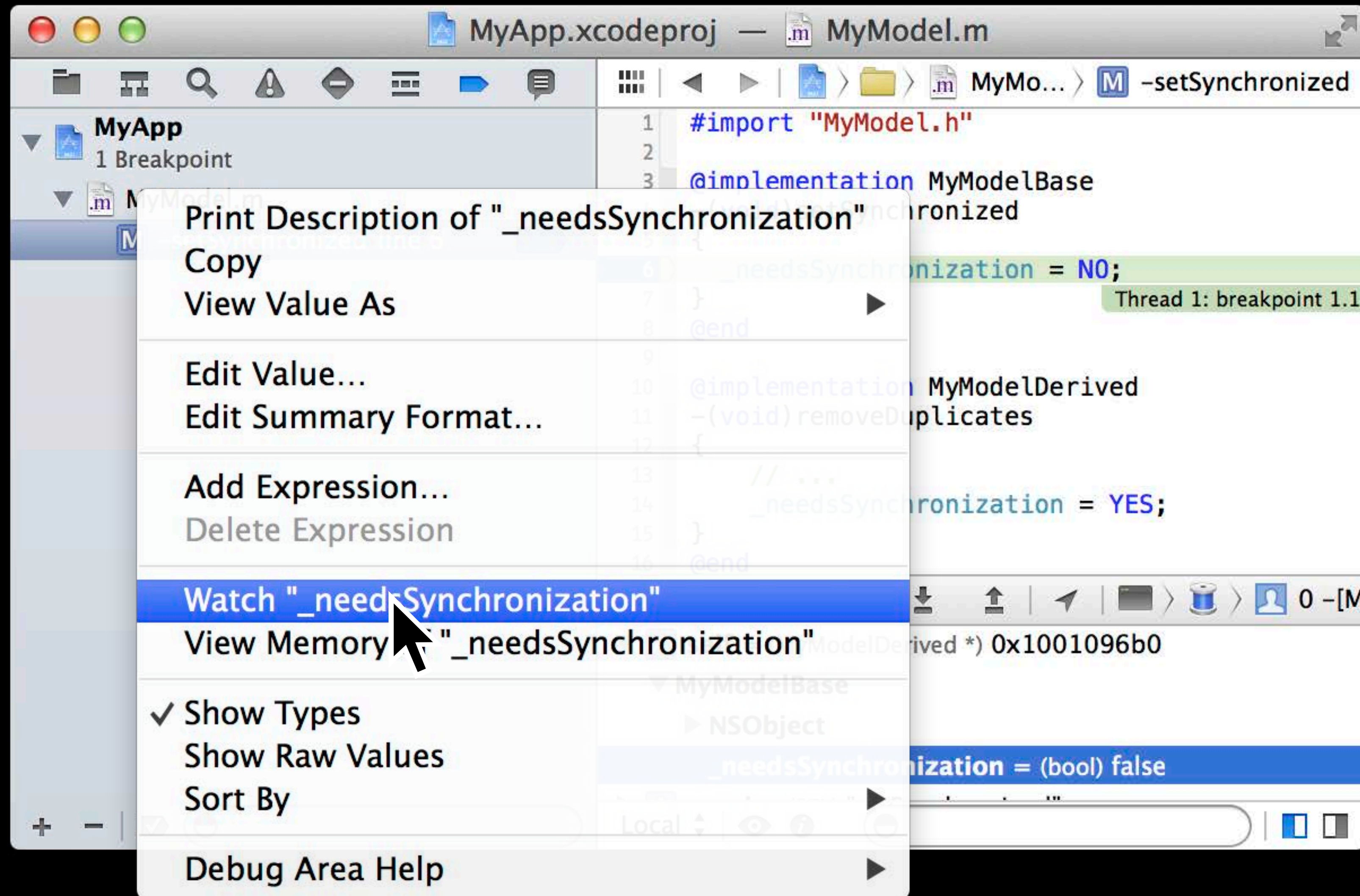
Focus on Memory with Watchpoints



Focus on Memory with Watchpoints



Focus on Memory with Watchpoints



Focus on Memory with Watchpoints

The screenshot displays the Xcode IDE interface for a project named 'MyApp.xcodeproj'. The left sidebar shows a project tree with 'MyApp' containing '1 Breakpoint' and 'MyModel.m' containing '1 Watchpoint'. The watchpoint is named '_needsSynchronization'. The main editor shows the source code for 'MyModel.m' with a watchpoint set on line 6: `_needsSynchronization = NO;`. The right sidebar shows the memory inspector for the current thread, displaying the object `self = (MyModelDerived *) 0x1001096b0` and its superclass `MyModelBase` (which inherits from `NSObject`). The `_needsSynchronization` property is shown as `(bool) false`. A mouse cursor is pointing at the watchpoint configuration area in the left sidebar.

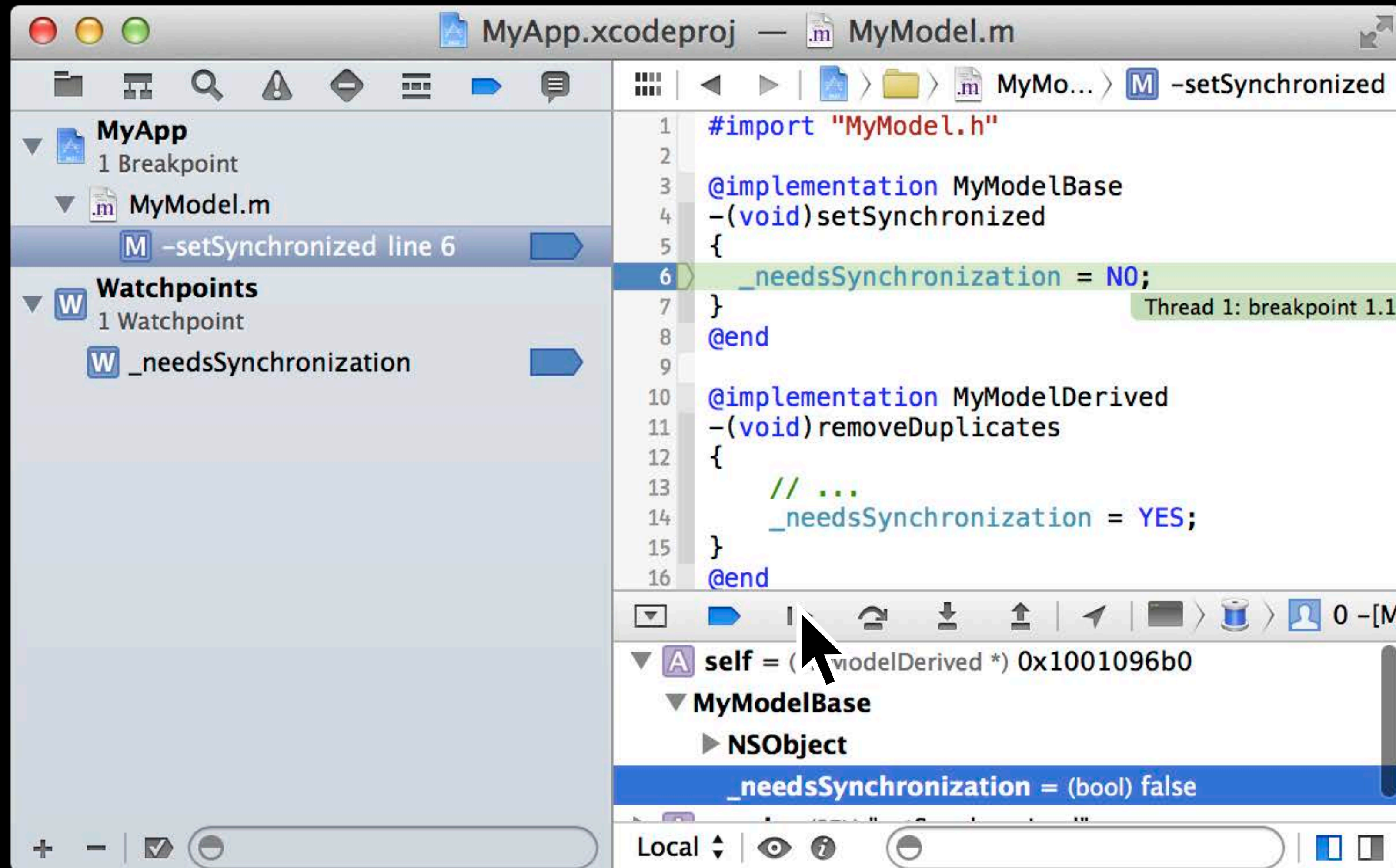
```
1 #import "MyModel.h"
2
3 @implementation MyModelBase
4 -(void)setSynchronized
5 {
6  _needsSynchronization = NO;
7 }
8 @end
9
10 @implementation MyModelDerived
11 -(void)removeDuplicates
12 {
13     // ...
14     _needsSynchronization = YES;
15 }
16 @end
```

Thread 1: breakpoint 1.1

self = (MyModelDerived *) 0x1001096b0

- MyModelBase
 - NSObject
 - `_needsSynchronization = (bool) false`

Focus on Memory with Watchpoints



Focus on Memory with Watchpoints

The screenshot displays the Xcode IDE interface for a project named 'MyApp.xcodeproj'. The main editor window shows the source code for 'MyModel.m'. The code includes a header file '#import "MyModel.h"', an implementation for 'MyModelBase' with a method '-(void) setSynchronized', and an implementation for 'MyModelDerived' with a method '-(void) removeDuplicates'. A watchpoint is set on the memory location of the instance variable '_needsSynchronization' in the 'MyModelBase' class, specifically at line 6 where it is assigned the value 'NO'. The watchpoint is named '_needsSynchronization' and is currently active. The interface also shows a breakpoint set on line 6 of the '-setSynchronized' method. The bottom of the screen shows the 'Local' pane, which displays the current state of the object 'self' as a 'MyModelDerived' instance, with its '_needsSynchronization' property set to '(bool) false'.

```
1 #import "MyModel.h"
2
3 @implementation MyModelBase
4 -(void) setSynchronized
5 {
6     _needsSynchronization = NO;
7 }
8 @end
9
10 @implementation MyModelDerived
11 -(void) removeDuplicates
12 {
13     // ...
14     _needsSynchronization = YES;
15 }
16 @end
```

Thread 1: watchpoint 1

self = (MyModelDerived *) 0x1001096b0

- MyModelBase
 - NSObject
 - _needsSynchronization = (bool) false

Focus on Memory with Watchpoints

The screenshot displays the Xcode IDE interface for a project named 'MyApp.xcodeproj'. The left sidebar shows a project tree with 'MyApp' containing 'MyModel.m'. A watchpoint is set on the memory location '0x1001096b0' for the variable '_needsSynchronization'. The main editor shows the source code for 'MyModel.m' with a watchpoint triggered at line 14, where the value of '_needsSynchronization' is set to 'YES'. The bottom panel shows the memory dump for the watchpoint, indicating the current value is 'YES'.

```
1  #import "MyModel.h"
2
3  @implementation MyModelBase
4  -(void) setSynchronized
5  {
6  _needsSynchronization = NO;
7  }
8  @end
9
10 @implementation MyModelDerived
11 -(void) removeDuplicates
12 {
13     // ...
14     _needsSynchronization = YES;
15 }
16 @end
```

Thread 1: watchpoint 1

Local

Stepping Through Problems

Execution control without surprises

Avoiding Repeated Steps

- Stepping repeatedly over irrelevant code gets old quickly

```
th u 11  
thread until 11
```

- LLDB will stop in one of two cases:
 - At the specified line, if your code goes there; or
 - After the function returns

Avoiding Repeated Steps

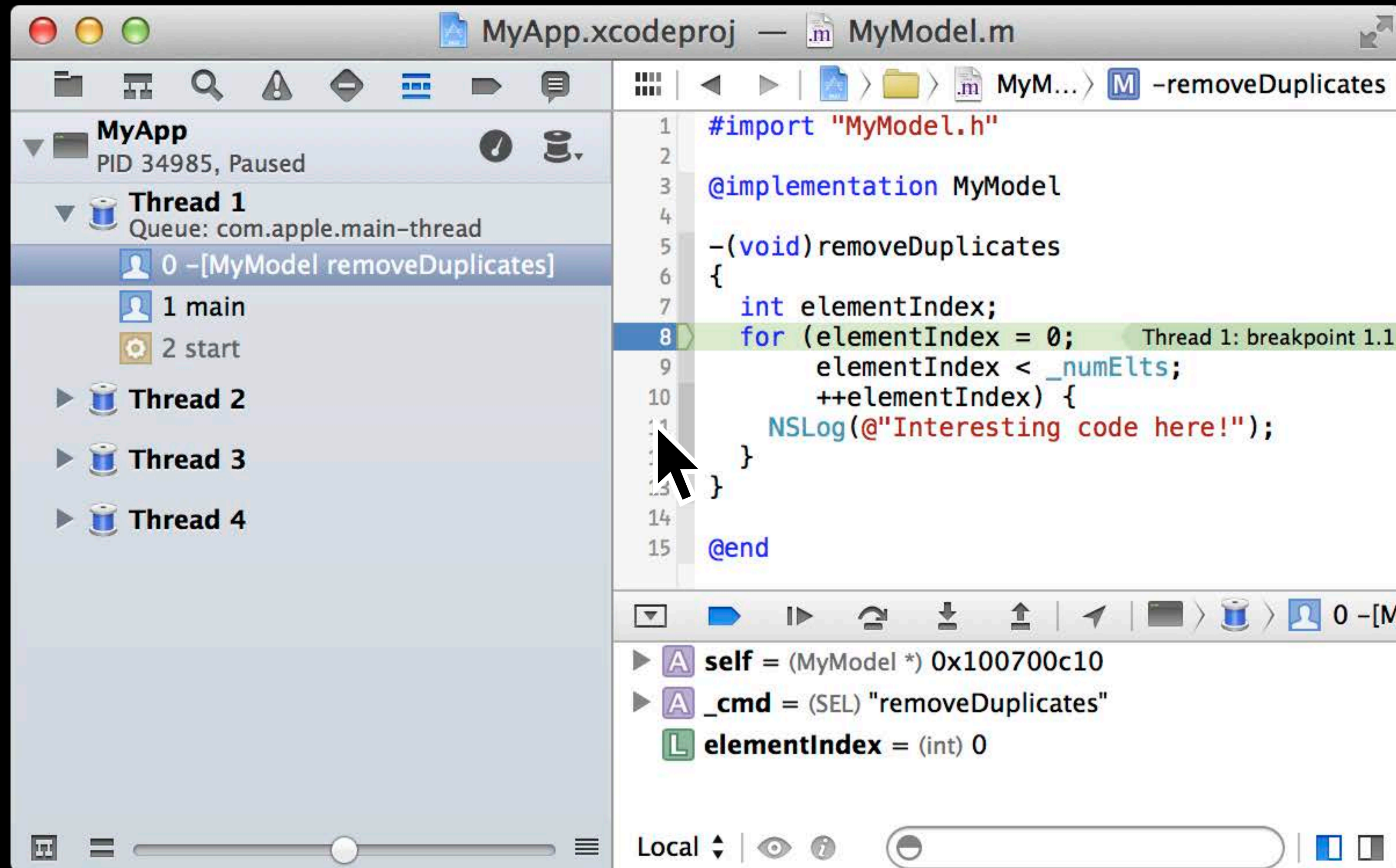
The screenshot shows the Xcode IDE with a project named 'MyApp.xcodeproj' and a file named 'MyModel.m'. The interface is divided into several panels:

- Debugger (Left):** Shows the application state. The main thread is paused at a breakpoint in the method `0 -[MyModel removeDuplicates]`. Other threads (Thread 1, 2, 3, 4) are visible but not selected.
- Code Editor (Center):** Displays the implementation of the `removeDuplicates` method in `MyModel.m`. The code is as follows:

```
1 #import "MyModel.h"
2
3 @implementation MyModel
4
5 -(void)removeDuplicates
6 {
7     int elementIndex;
8     for (elementIndex = 0;
9         elementIndex < _numElts;
10        ++elementIndex) {
11         NSLog(@"Interesting code here!");
12     }
13 }
14
15 @end
```

Line 8 is highlighted, indicating the current execution point.
- Debugger Console (Bottom):** Shows the current state of local variables:
 - `self = (MyModel *) 0x100700c10`
 - `_cmd = (SEL) "removeDuplicates"`
 - `elementIndex = (int) 0`

Avoiding Repeated Steps



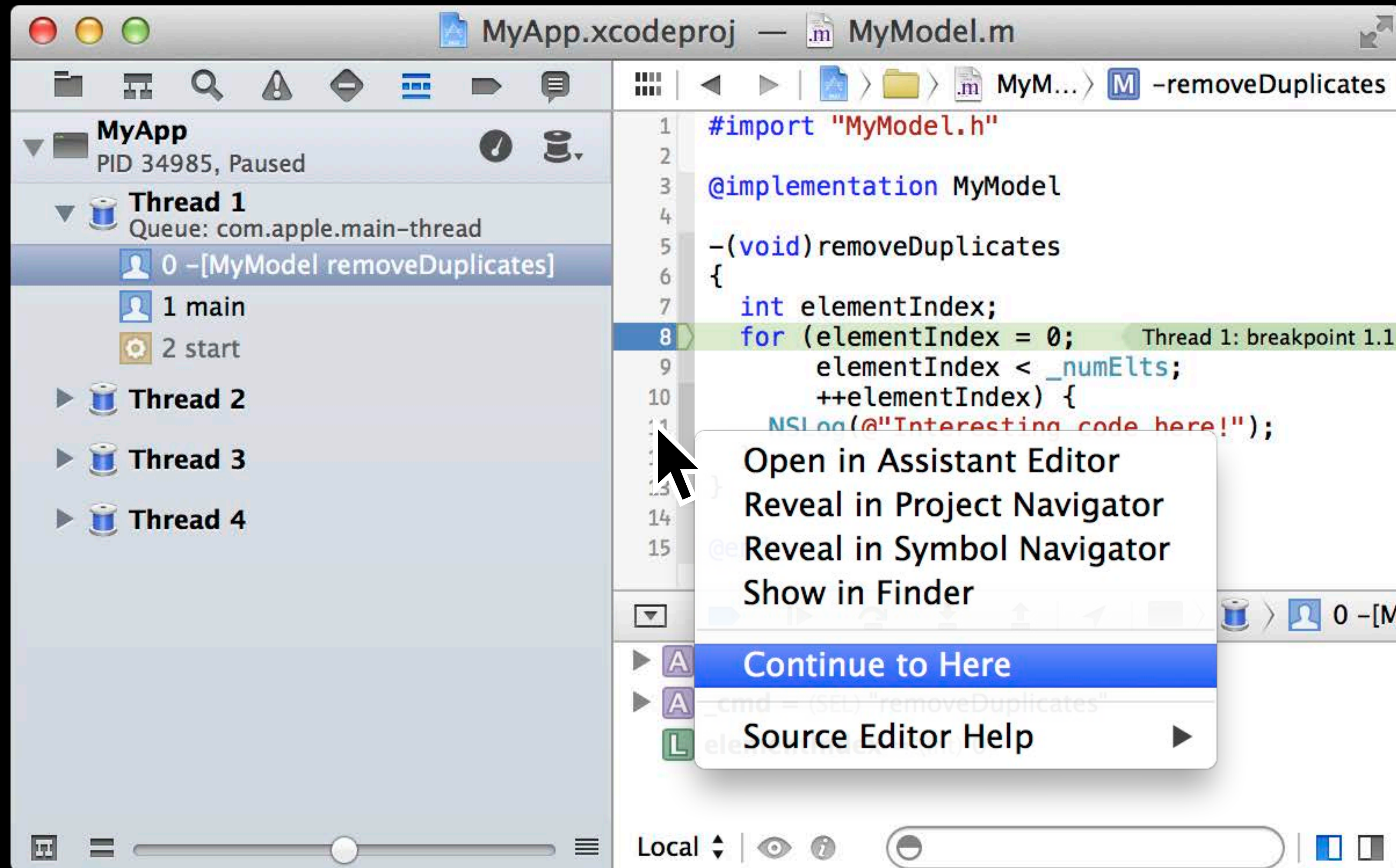
The screenshot shows the Xcode IDE interface. On the left, the 'MyApp' project is expanded to show 'Thread 1' (Queue: com.apple.main-thread) with a breakpoint set on the method call '0 -[MyModel removeDuplicates]'. The main editor displays the source code for 'MyModel.m', with a breakpoint set on line 8: 'for (elementIndex = 0;'. The bottom pane shows the current state of the program, including the method call and the value of 'elementIndex'.

```
1 #import "MyModel.h"
2
3 @implementation MyModel
4
5 -(void)removeDuplicates
6 {
7     int elementIndex;
8     for (elementIndex = 0; Thread 1: breakpoint 1.1
9         elementIndex < _numElts;
10        ++elementIndex) {
11         NSLog(@"Interesting code here!");
12     }
13 }
14
15 @end
```

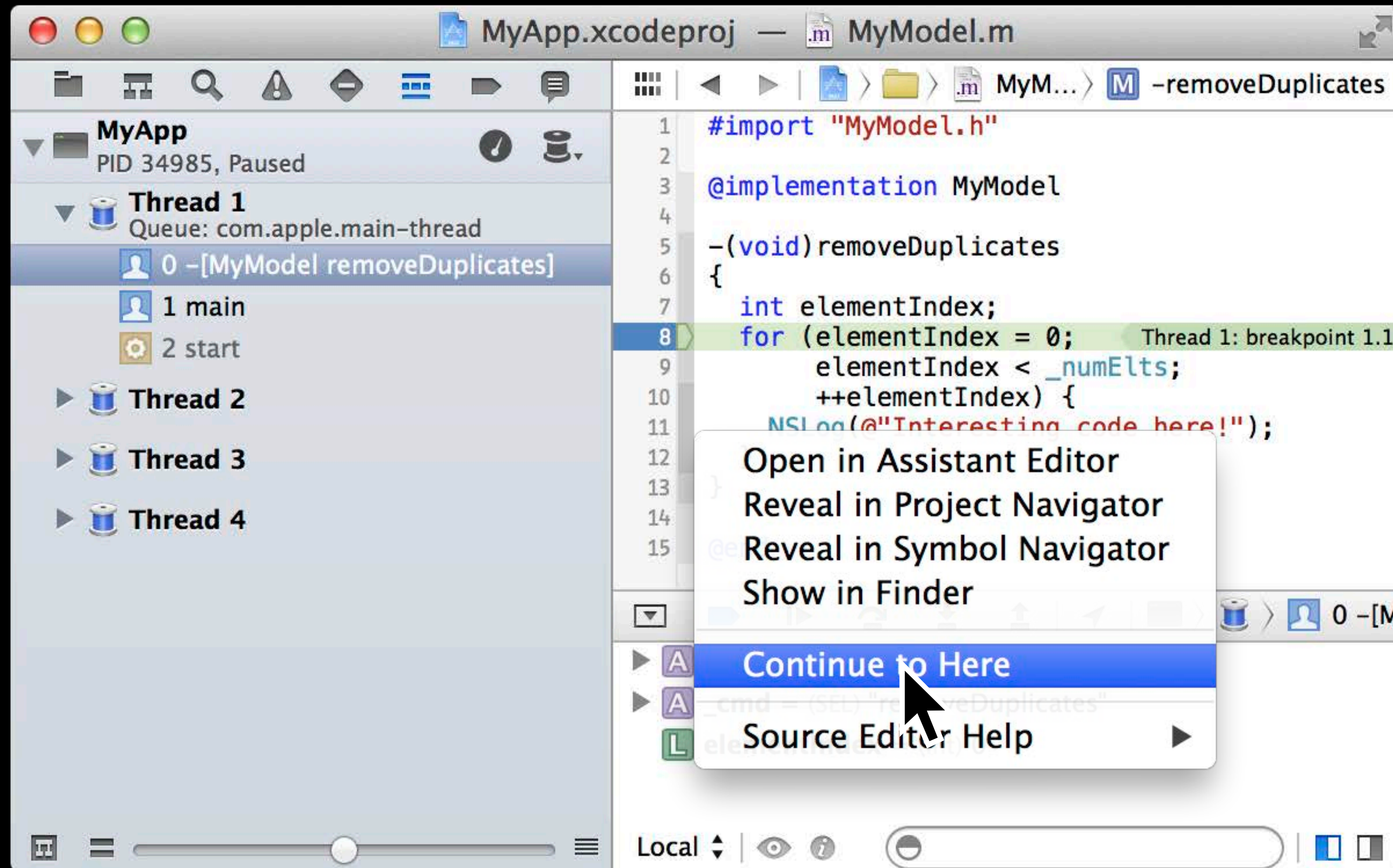
Local

- self = (MyModel *) 0x100700c10
- _cmd = (SEL) "removeDuplicates"
- elementIndex = (int) 0

Avoiding Repeated Steps



Avoiding Repeated Steps



Avoiding Repeated Steps

The screenshot displays the Xcode IDE with a debugger session. The window title is "MyApp.xcodeproj — .m MyModel.m". The left sidebar shows the "MyApp" process (PID 34985, Paused) with a stack of threads. "Thread 1" (Queue: com.apple.main-thread) is selected, and the current step is "0 -[MyModel removeDuplicates]". The main editor shows the implementation of the "removeDuplicates" method in "MyModel.m". The code is as follows:

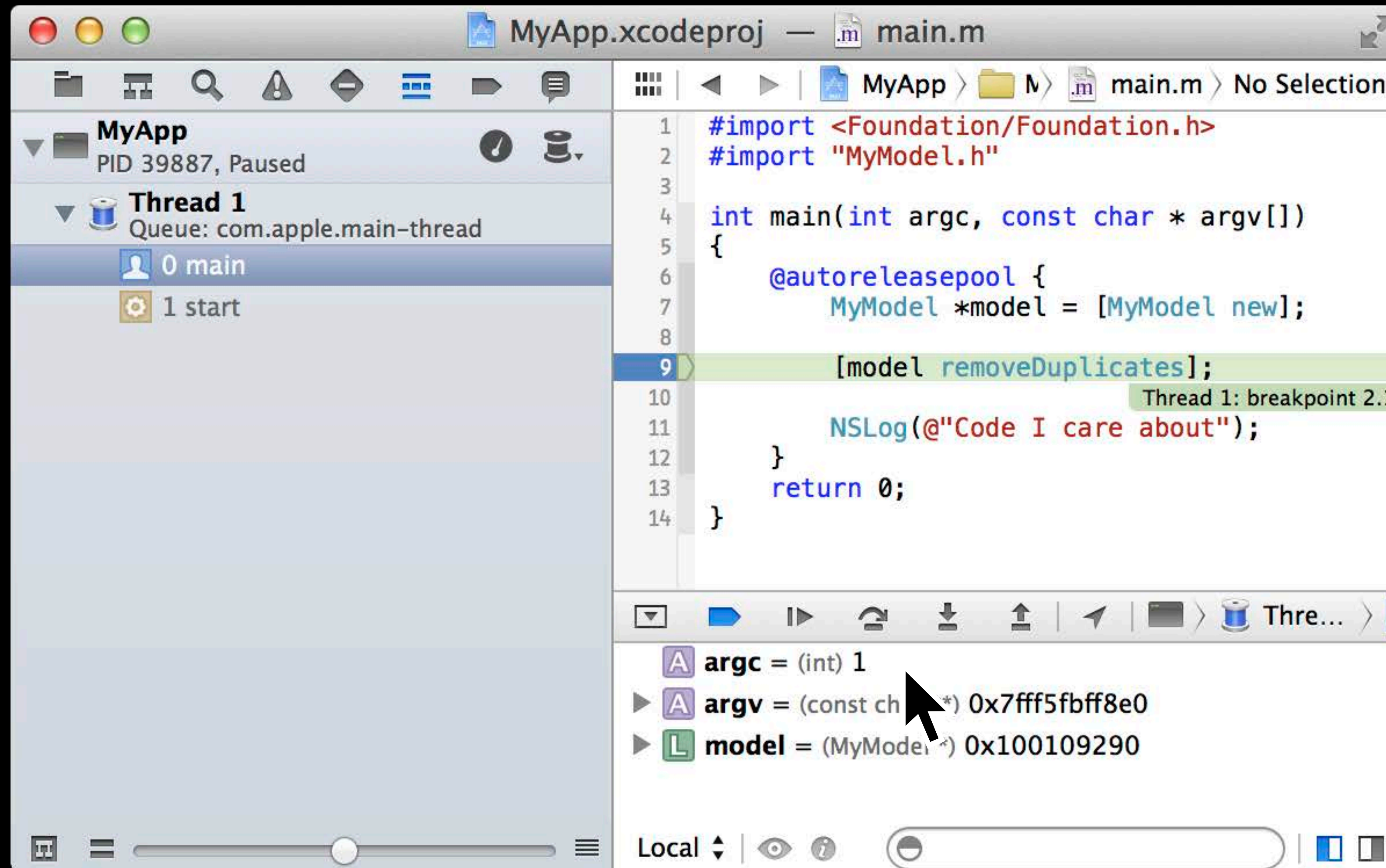
```
1 #import "MyModel.h"
2
3 @implementation MyModel
4
5 -(void)removeDuplicates
6 {
7     int elementIndex;
8     for (elementIndex = 0;
9         elementIndex < _numElts;
10        ++elementIndex) {
11     NSLog(@"Interesting code here!");
12 }
13 }
14
15 @end
```

The debugger is paused at line 11. The bottom debugger console shows the following local variables:

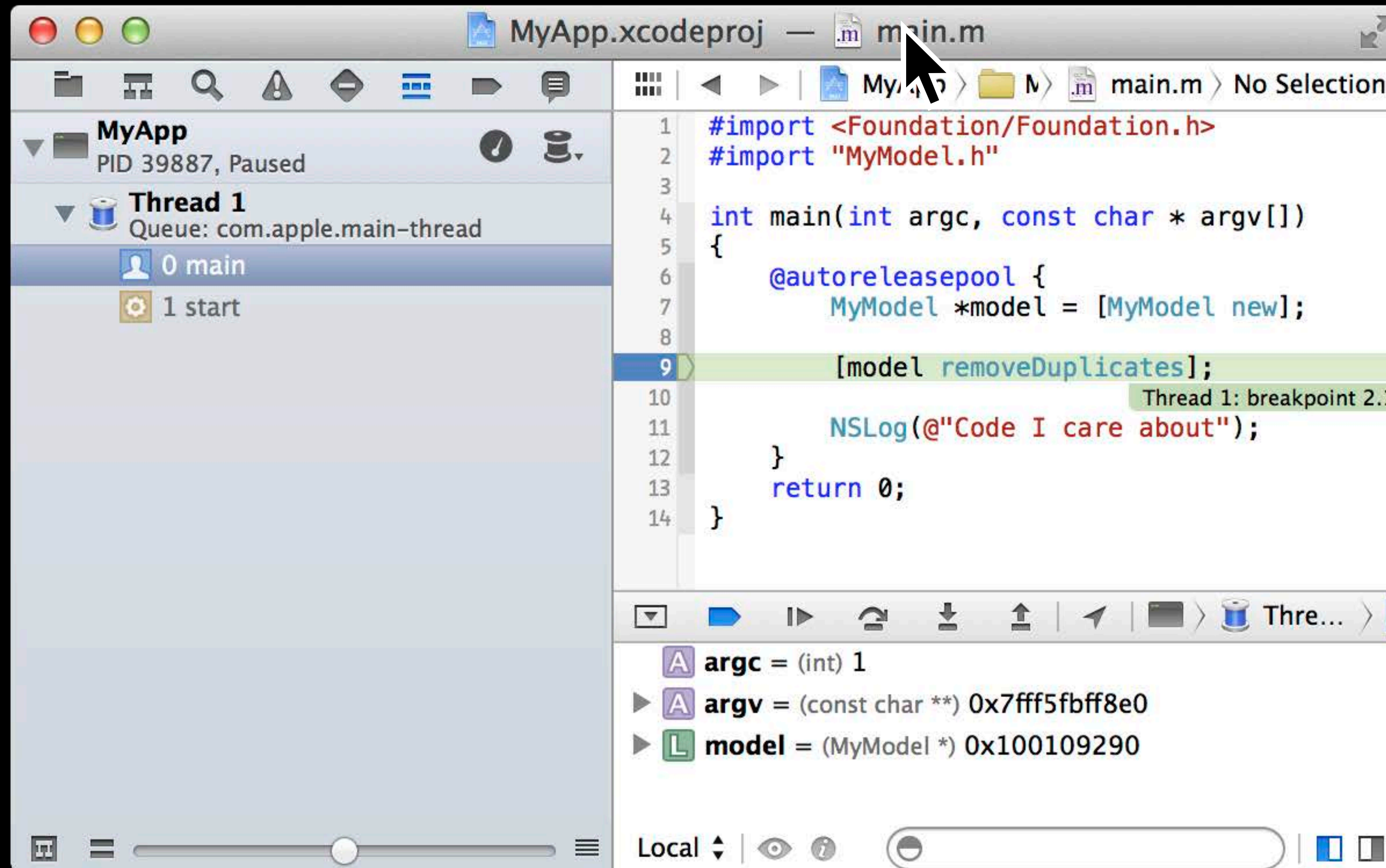
- `self` = (MyModel *) 0x100700c10
- `_cmd` = (SEL) "removeDuplicates"
- `elementIndex` = (int) 0

A green highlight is on the NSLog line, and a tooltip indicates "Thread 1: step until".

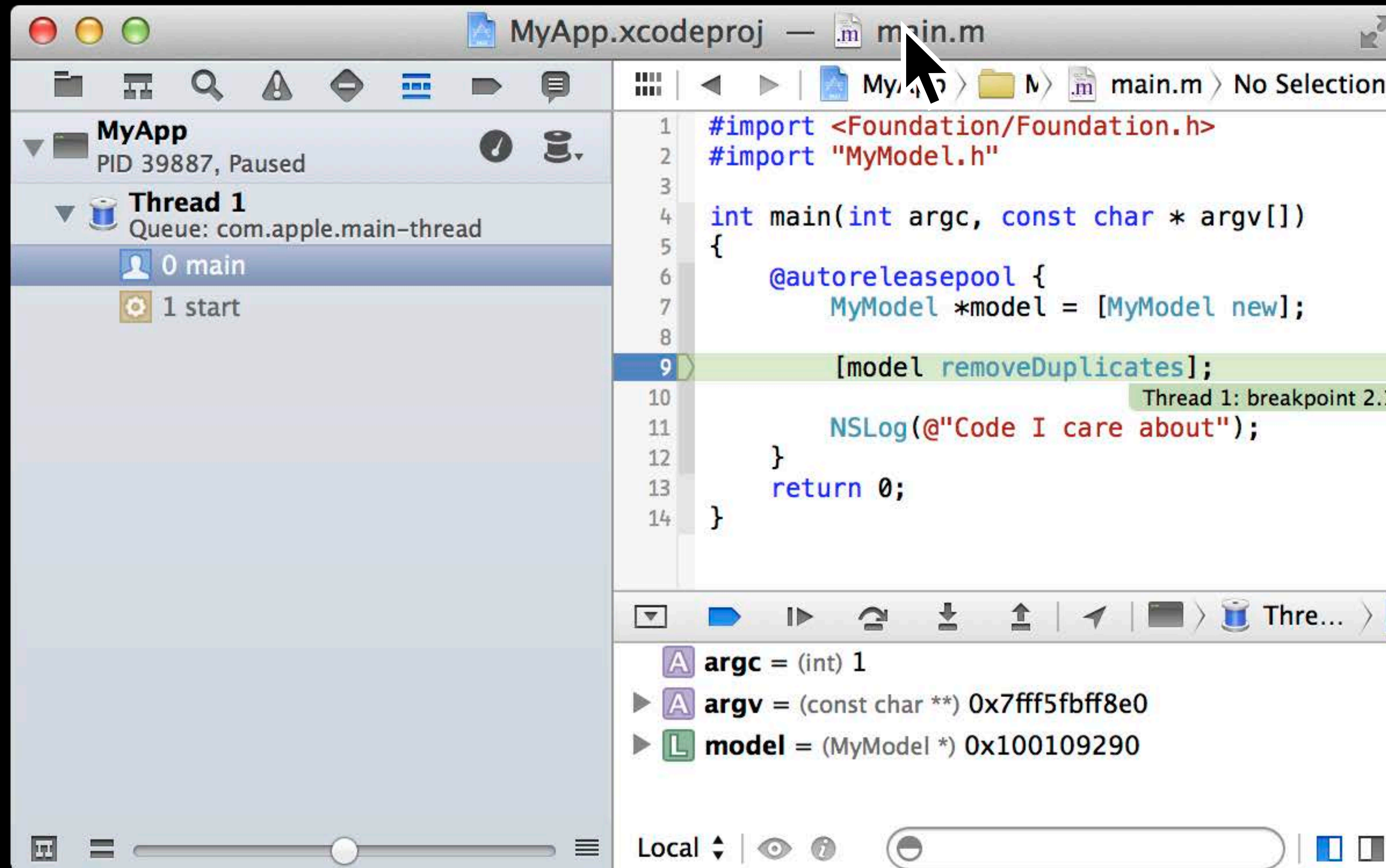
Hitting Breakpoints While Stepping



Hitting Breakpoints While Stepping



Hitting Breakpoints While Stepping



Hitting Breakpoints While Stepping

The screenshot shows the Xcode IDE with a project named 'MyApp.xcodeproj' and a file named 'main.m'. The code in the editor is as follows:

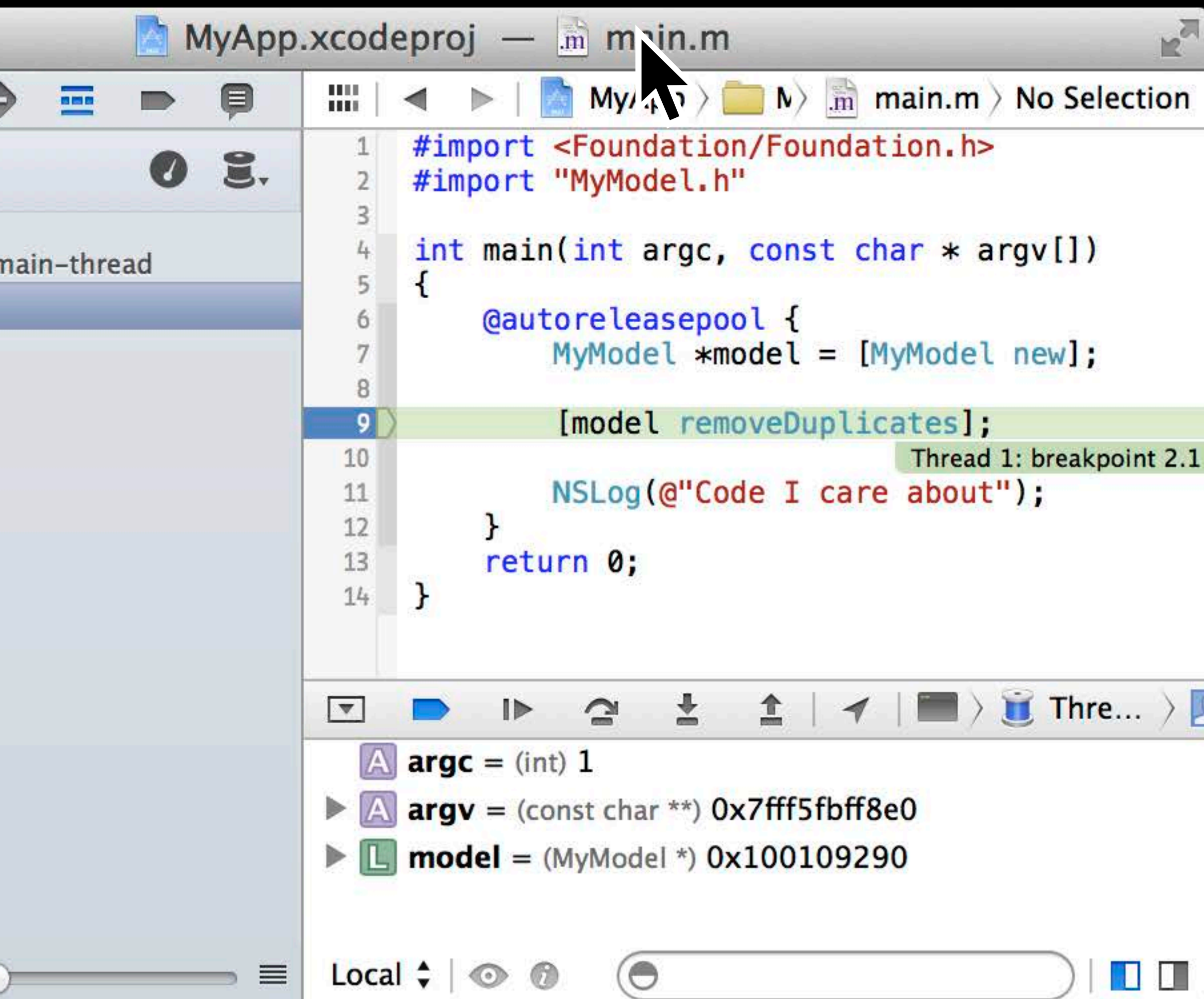
```
1 #import <Foundation/Foundation.h>
2 #import "MyModel.h"
3
4 int main(int argc, const char * argv[])
5 {
6     @autoreleasepool {
7         MyModel *model = [MyModel new];
8
9         [model removeDuplicates];
10
11         NSLog(@"Code I care about");
12     }
13     return 0;
14 }
```

The execution is paused at line 9, which is highlighted in green. A tooltip above the code indicates 'Thread 1: breakpoint 2.1'. The left sidebar shows the 'main-thread' selected. The bottom toolbar contains various debugging controls, including a 'Thread...' button. The 'Local' pane at the bottom displays the following variables:

- A** `argc` = (int) 1
- A** `argv` = (const char **) 0x7fff5fbff8e0
- L** `model` = (MyModel *) 0x100109290

The 'Local' pane also includes a search bar and a zoom slider.

Hitting Breakpoints While Stepping



The screenshot shows the Xcode IDE with a project named 'MyApp.xcodeproj' and a file named 'main.m'. The code in 'main.m' is as follows:

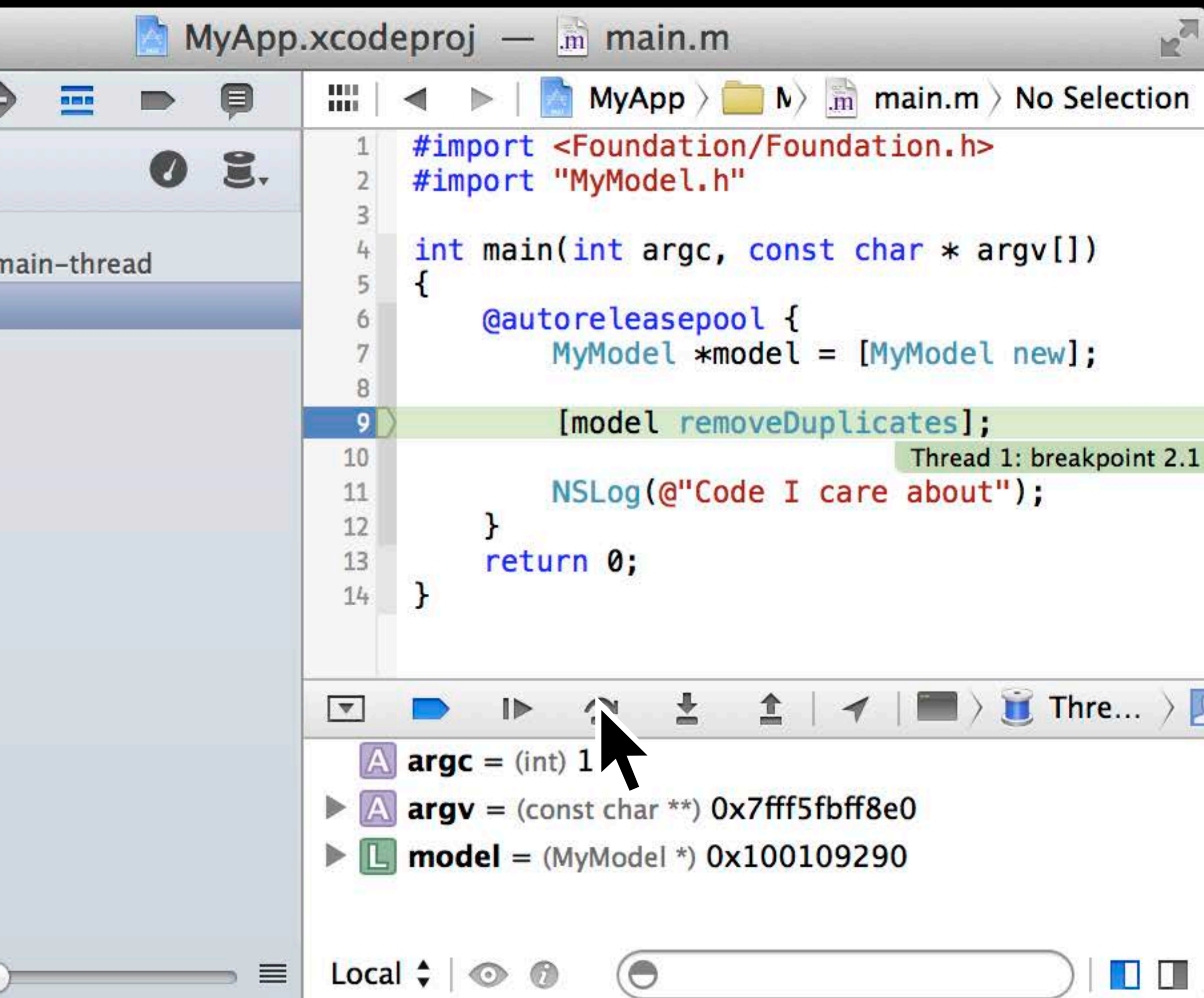
```
1 #import <Foundation/Foundation.h>
2 #import "MyModel.h"
3
4 int main(int argc, const char * argv[])
5 {
6     @autoreleasepool {
7         MyModel *model = [MyModel new];
8
9         [model removeDuplicates];
10
11         NSLog(@"Code I care about");
12     }
13     return 0;
14 }
```

The line `[model removeDuplicates];` at line 9 is highlighted in green, indicating a breakpoint hit. A tooltip above the line reads 'Thread 1: breakpoint 2.1'. The 'main-thread' is selected in the left sidebar. The bottom of the window shows the LLDB console with the following variables:

```
A argc = (int) 1
A argv = (const char **) 0x7fff5fbff8e0
L model = (MyModel *) 0x100109290
```

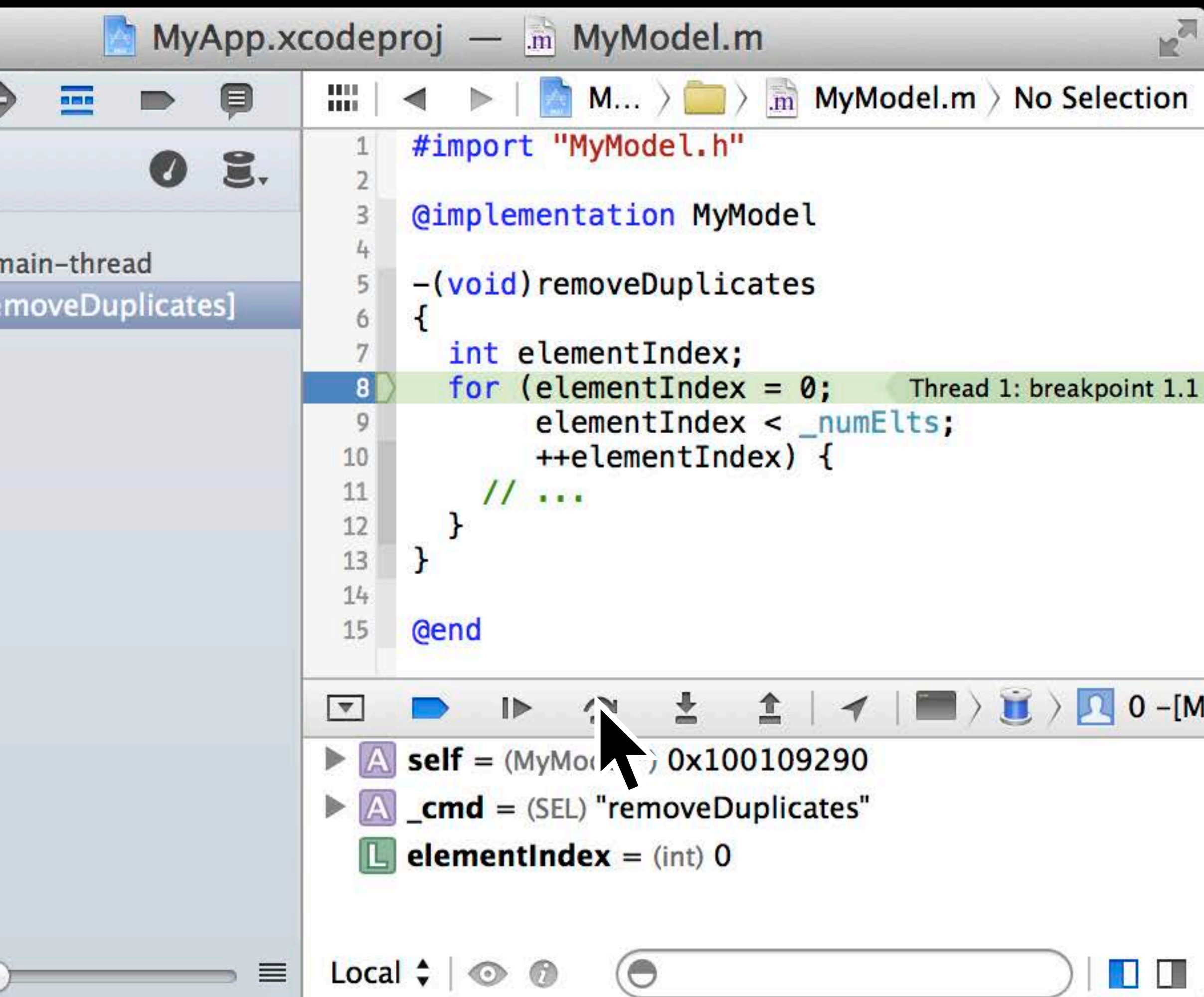
- Stepping can hit breakpoints
- LLDB maintains a stack of things you are doing
 - When you step, LLDB puts it on the stack

Hitting Breakpoints While Stepping



- Stepping can hit breakpoints
- LLDB maintains a stack of things you are doing
 - When you step, LLDB puts it on the stack

Hitting Breakpoints While Stepping



The screenshot shows the Xcode IDE with a project named 'MyApp.xcodeproj' and a file named 'MyModel.m'. The code in the editor is as follows:

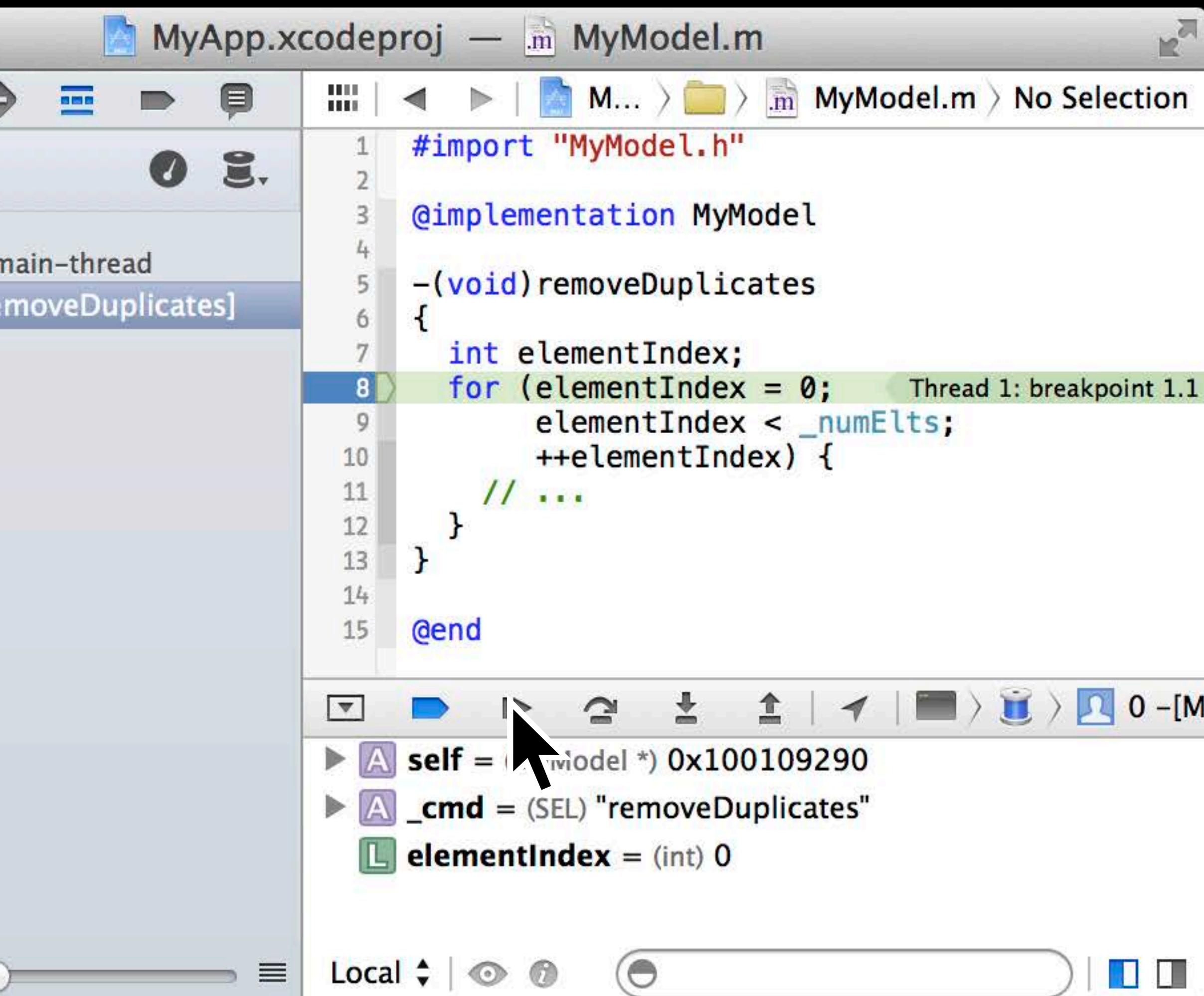
```
1 #import "MyModel.h"
2
3 @implementation MyModel
4
5 -(void)removeDuplicates
6 {
7     int elementIndex;
8     for (elementIndex = 0;
9         elementIndex < _numElts;
10        ++elementIndex) {
11         // ...
12     }
13 }
14
15 @end
```

The breakpoint is set at line 8, and the debugger has stopped at this line. The status bar indicates 'Thread 1: breakpoint 1.1'. The LLDB console shows the following variables:

```
self = (MyModel) 0x100109290
_cmd = (SEL) "removeDuplicates"
elementIndex = (int) 0
```

- Stepping can hit breakpoints
- LLDB maintains a stack of things you are doing
 - When you step, LLDB puts it on the stack
 - If you hit a breakpoint, LLDB remembers the stack...

Hitting Breakpoints While Stepping



The screenshot shows the Xcode IDE with a project named 'MyApp.xcodeproj' and a file named 'MyModel.m'. The code in the editor is as follows:

```
1 #import "MyModel.h"
2
3 @implementation MyModel
4
5 -(void)removeDuplicates
6 {
7     int elementIndex;
8     for (elementIndex = 0;
9         elementIndex < _numElts;
10        ++elementIndex) {
11         // ...
12     }
13 }
14
15 @end
```

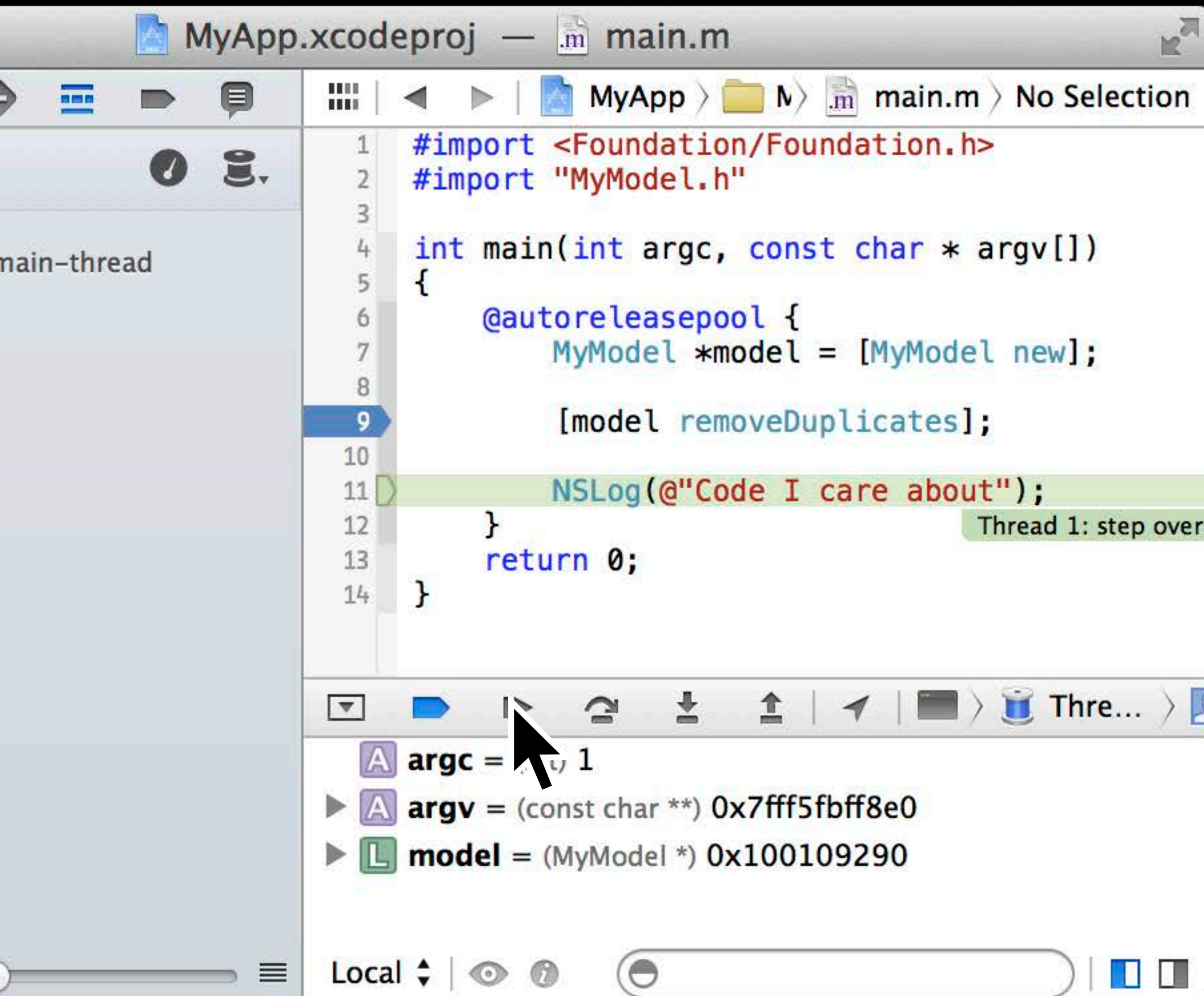
The execution is paused at line 8, which is highlighted in green. A tooltip above the line reads 'Thread 1: breakpoint 1.1'. The LLDB console at the bottom shows the current state of the program:

```
▶ A self = (MyModel *) 0x100109290
▶ A _cmd = (SEL) "removeDuplicates"
L elementIndex = (int) 0
```

The 'Local' pane at the bottom left shows the current scope.

- Stepping can hit breakpoints
- LLDB maintains a stack of things you are doing
 - When you step, LLDB puts it on the stack
 - If you hit a breakpoint, LLDB remembers the stack...

Hitting Breakpoints While Stepping



The screenshot shows the Xcode IDE with a project named 'MyApp.xcodeproj' and a file named 'main.m'. The code in the editor is as follows:

```
1 #import <Foundation/Foundation.h>
2 #import "MyModel.h"
3
4 int main(int argc, const char * argv[])
5 {
6     @autoreleasepool {
7         MyModel *model = [MyModel new];
8
9         [model removeDuplicates];
10
11         NSLog(@"Code I care about");
12     }
13     return 0;
14 }
```

A breakpoint is set on line 9, and the debugger has stepped over it. The console below shows the following variables:

- argc** = 1
- argv** = (const char **) 0x7fff5fbff8e0
- model** = (MyModel *) 0x100109290

- Stepping can hit breakpoints
- LLDB maintains a stack of things you are doing
 - When you step, LLDB puts it on the stack
 - If you hit a breakpoint, LLDB remembers the stack...
 - ...and continuing lets LLDB continue the step

Calling Code by Hand

- What if it's hard to make the code you care about run?
- Call the code using Clang!

```
b “[ModelDerived removeDuplicates]”  
e -i false -- [self removeDuplicates]  
expression --ignore-breakpoints false  
-- [self removeDuplicates]
```

Process 31109 stopped

* thread #1:

-[ModelDerived removeDuplicates]

- Clang runs what you type after `expression` in the process

Calling Code by Hand

- What if it's hard to make the code you care about run?
- Call the code using Clang!

```
b "[ModelDerived removeDuplicates]"  
e -i false -- [self removeDuplicates]  
expression --ignore-breakpoints false  
    -- [self removeDuplicates]
```

← Don't ignore breakpoints!
LLDB does by default

Process 31109 stopped

* thread #1:

-[ModelDerived removeDuplicates]

- Clang runs what you type after `expression` in the process

Inspecting Data to Find Causes

Looking at variables with new eyes

Enrico Granata

LLDB Engineer

Inspecting Data

Inspecting Data

- Inspecting data at the command line

Inspecting Data

- Inspecting data at the command line
- Data formatters

Inspecting Data

- Inspecting data at the command line
- Data formatters
- Opaque data inspection

Inspecting Data at the Command Line

- Several commands
 - Some new
 - Some old
- Which do I use?

Command / Output

When to Use

Command / Output

When to Use

`frame variable`

```
(int) argc = 4  
(char **) argv = 0x1240f0a0
```

Show all my locals

Command / Output	When to Use
<code>frame variable</code> <code>(int) argc = 4</code> <code>(char **) argv = 0x1240f0a0</code>	Show all my locals
<code>expression (x + 35)</code> <code>(int) \$5 = 36</code>	Execute arbitrary code

Command / Output	When to Use
<pre>frame variable (int) argc = 4 (char **) argv = 0x1240f0a0</pre>	Show all my locals
<pre>expression (x + 35) (int) \$5 = 36</pre>	Execute arbitrary code
<pre>p @"Hello" (NSString *) \$6 = @"Hello"</pre>	Compact syntax for expression Allows GDB-style format (p/x)

Command / Output	When to Use
<pre>frame variable (int) argc = 4 (char **) argv = 0x1240f0a0</pre>	Show all my locals
<pre>expression (x + 35) (int) \$5 = 36</pre>	Execute arbitrary code
<pre>p @"Hello" (NSString *) \$6 = @"Hello"</pre>	Compact syntax for expression Allows GDB-style format (p/x)
<pre>po @"Hello" Hello</pre>	Execute arbitrary code, then call the description selector on the result

Inspecting Data at the Command Line

- Several commands
 - Each with a specific use case

“Raw Data” vs. “Data”

“Raw Data” vs. “Data”

- Raw data is not always easy to decipher

“Raw Data” vs. “Data”

- Raw data is not always easy to decipher
 - Too complex

“Raw Data” vs. “Data”

- Raw data is not always easy to decipher
 - Too complex
 - Not your types

“Raw Data” vs. “Data”

- Raw data is not always easy to decipher
 - Too complex
 - Not your types
 - Information overload

“Raw Data”

Life without formatters

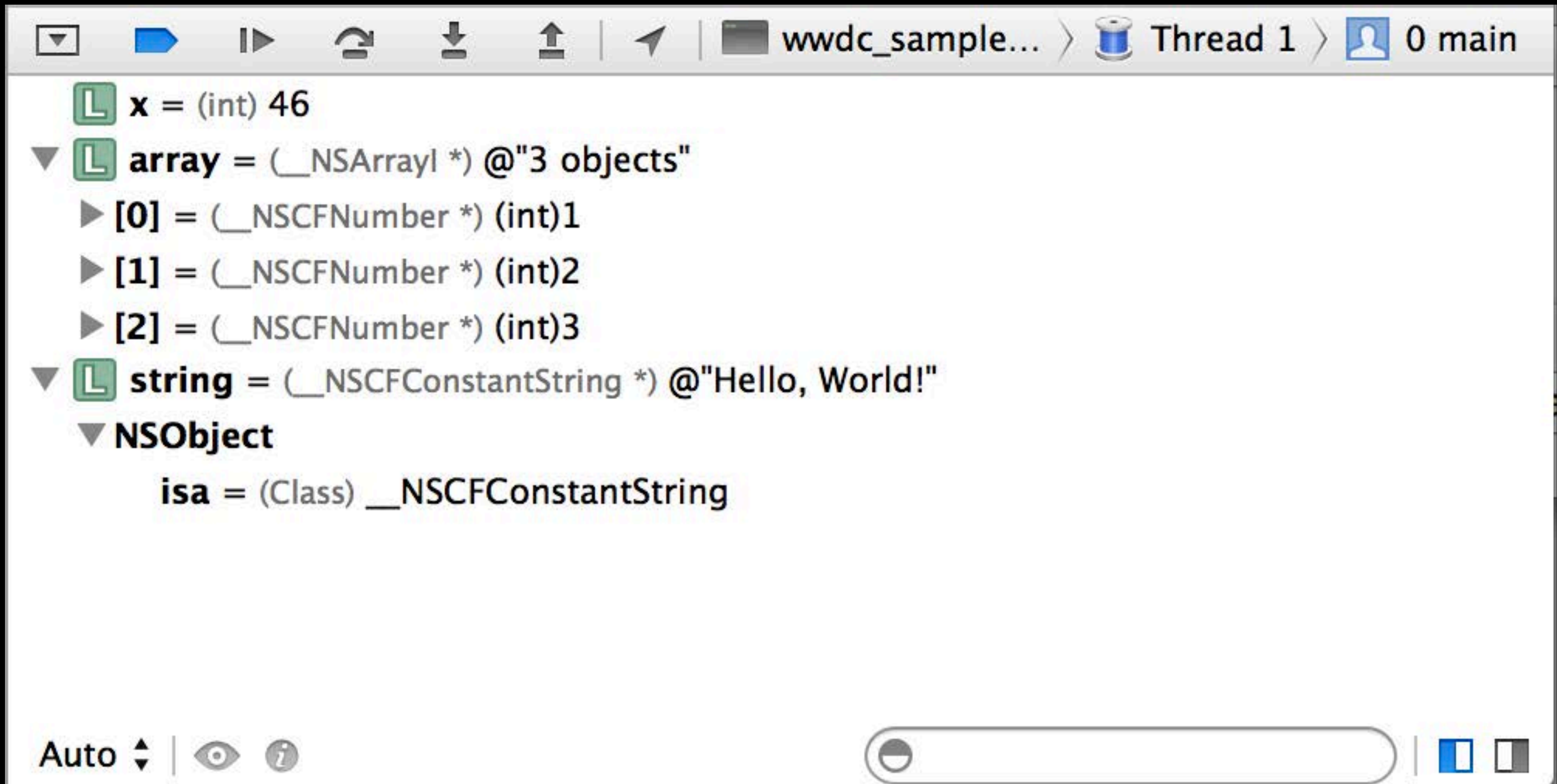
The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. The main area displays the following memory dump:

- `x = (int) 46`
- `array = (__NSArrayI *) 0x100109200`
 - `NSObject`
 - `isa = (Class) 0x7fff7a304480`
- `string = (__NSCFConstantString *) 0x100001090`
 - `NSObject`
 - `isa = (Class) 0x7fff7a3040e8`

At the bottom of the debugger, there is a control bar with the text "Auto" and a double-headed arrow, followed by an eye icon and an information icon. To the right of these icons is a horizontal scrollbar and a window management icon.

"Data"

Life with formatters



The screenshot shows the Xcode debugger interface. The top toolbar includes icons for navigation and execution. The breadcrumb trail indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`.

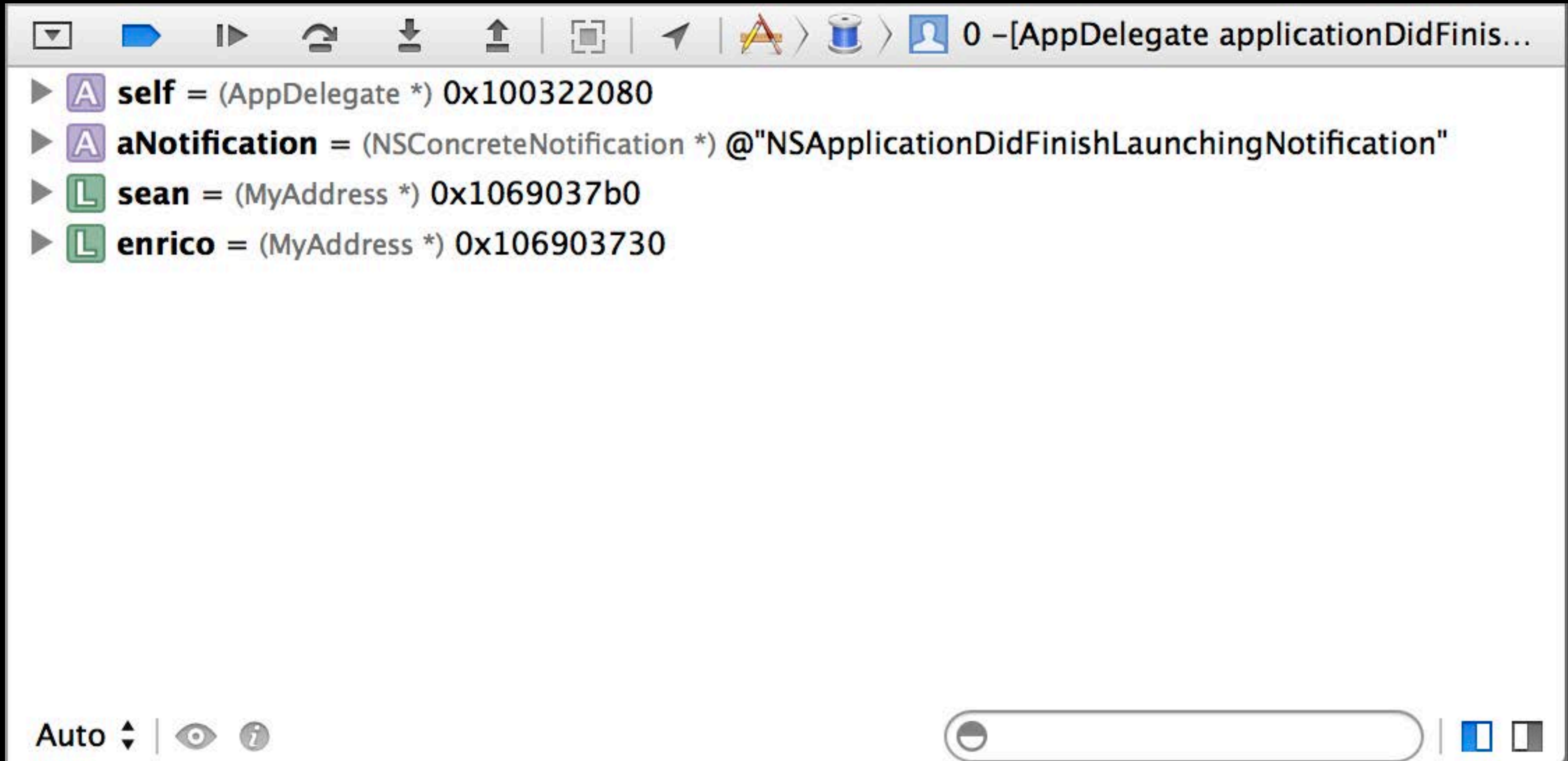
The variable view displays the following data:

- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

At the bottom, there is a control bar with the text "Auto" and a dropdown arrow, followed by an eye icon and an information icon. On the right side, there is a scroll bar and a window management icon.

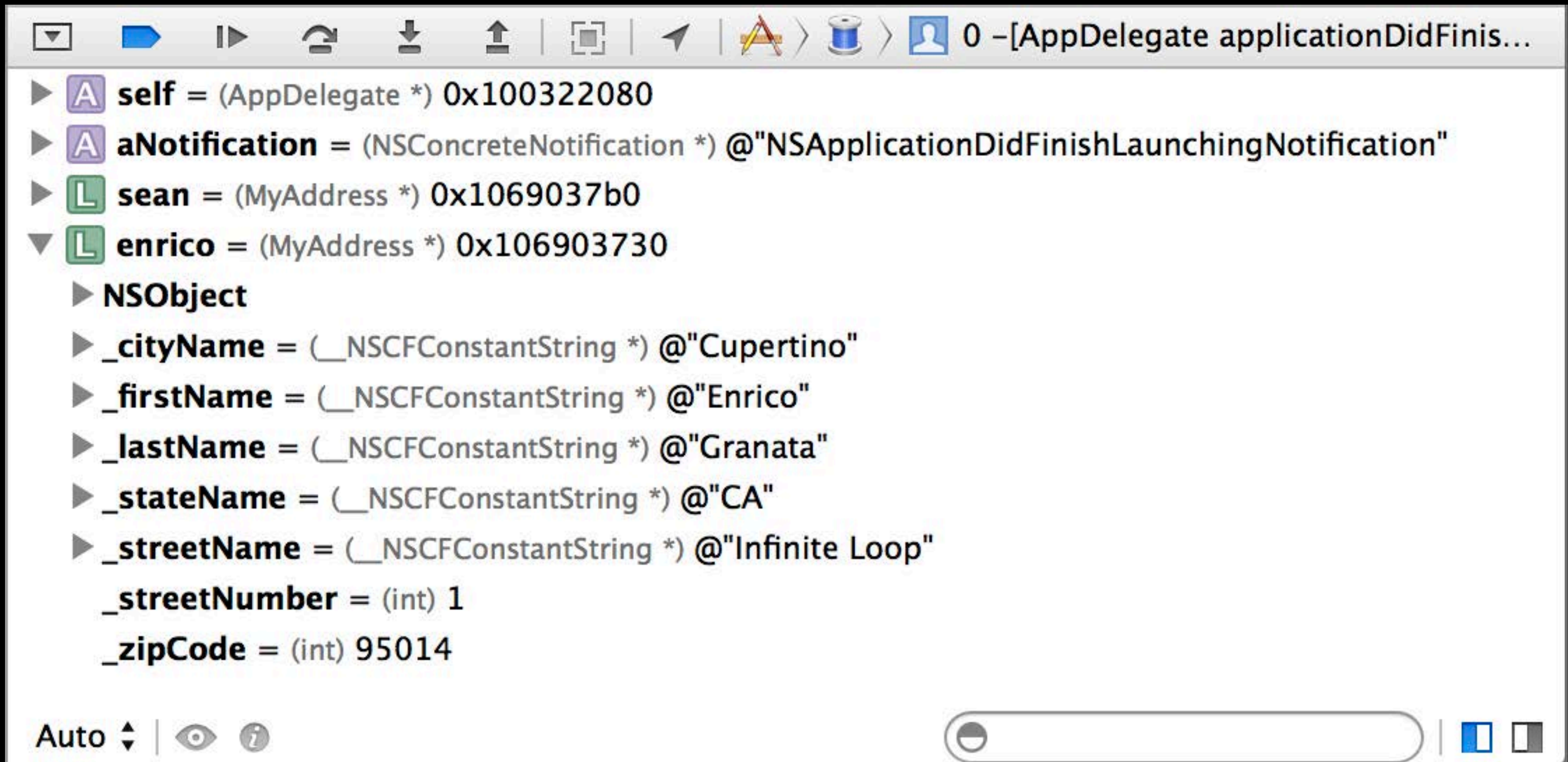
“Raw Data”

Life without formatters



“Raw Data”

Life without formatters



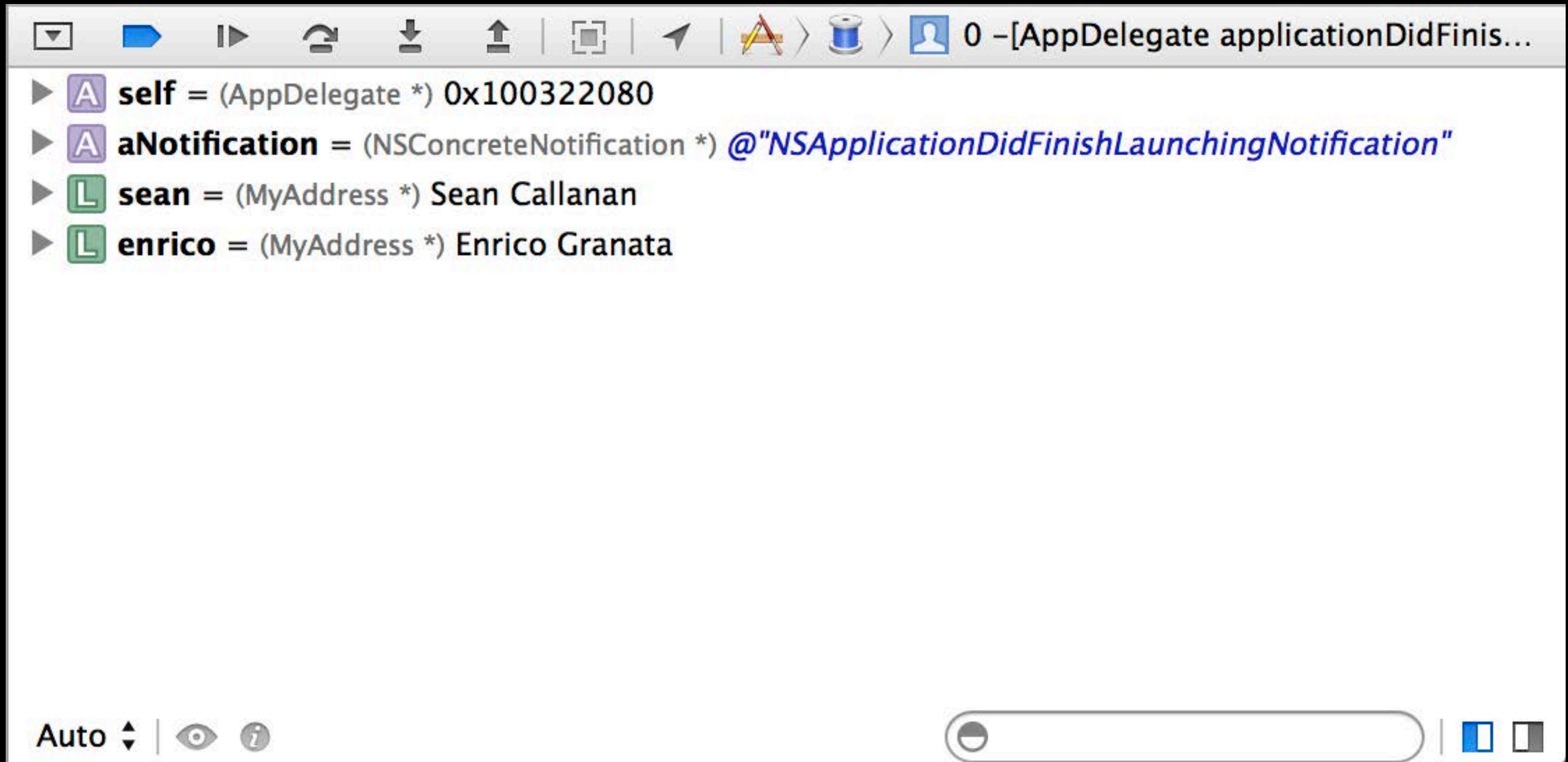
The screenshot shows the Xcode interface with the Debug Console open. The title bar reads "0 -[AppDelegate applicationWillDidFinis...". The console displays the following raw data for variables:

- self** = (AppDelegate *) 0x100322080
- aNotification** = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
- sean** = (MyAddress *) 0x1069037b0
- enrico** = (MyAddress *) 0x106903730
 - NSObject**
 - _cityName** = (__NSCFConstantString *) @"Cupertino"
 - _firstName** = (__NSCFConstantString *) @"Enrico"
 - _lastName** = (__NSCFConstantString *) @"Granata"
 - _stateName** = (__NSCFConstantString *) @"CA"
 - _streetName** = (__NSCFConstantString *) @"Infinite Loop"
 - _streetNumber** = (int) 1
 - _zipCode** = (int) 95014

At the bottom left, there are controls for "Auto" (with a dropdown arrow), a visibility icon, and an information icon. At the bottom right, there is a scrollbar and a window management icon.

"Data"

Life with formatters



The screenshot shows the Xcode interface with the Debug Console open. The console displays the following variable values:

- self** = (AppDelegate *) 0x100322080
- aNotification** = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
- sean** = (MyAddress *) Sean Callanan
- enrico** = (MyAddress *) Enrico Granata

The interface includes a toolbar at the top with various icons for navigation and debugging, and a status bar at the bottom with a scroll bar and window control buttons.

Data Formatters

Data Formatters

- Built-in formatters for system libraries
 - STL
 - CoreFoundation
 - Foundation

Data Formatters

- Built-in formatters for system libraries
 - STL
 - CoreFoundation
 - Foundation
- What we do... you can do too

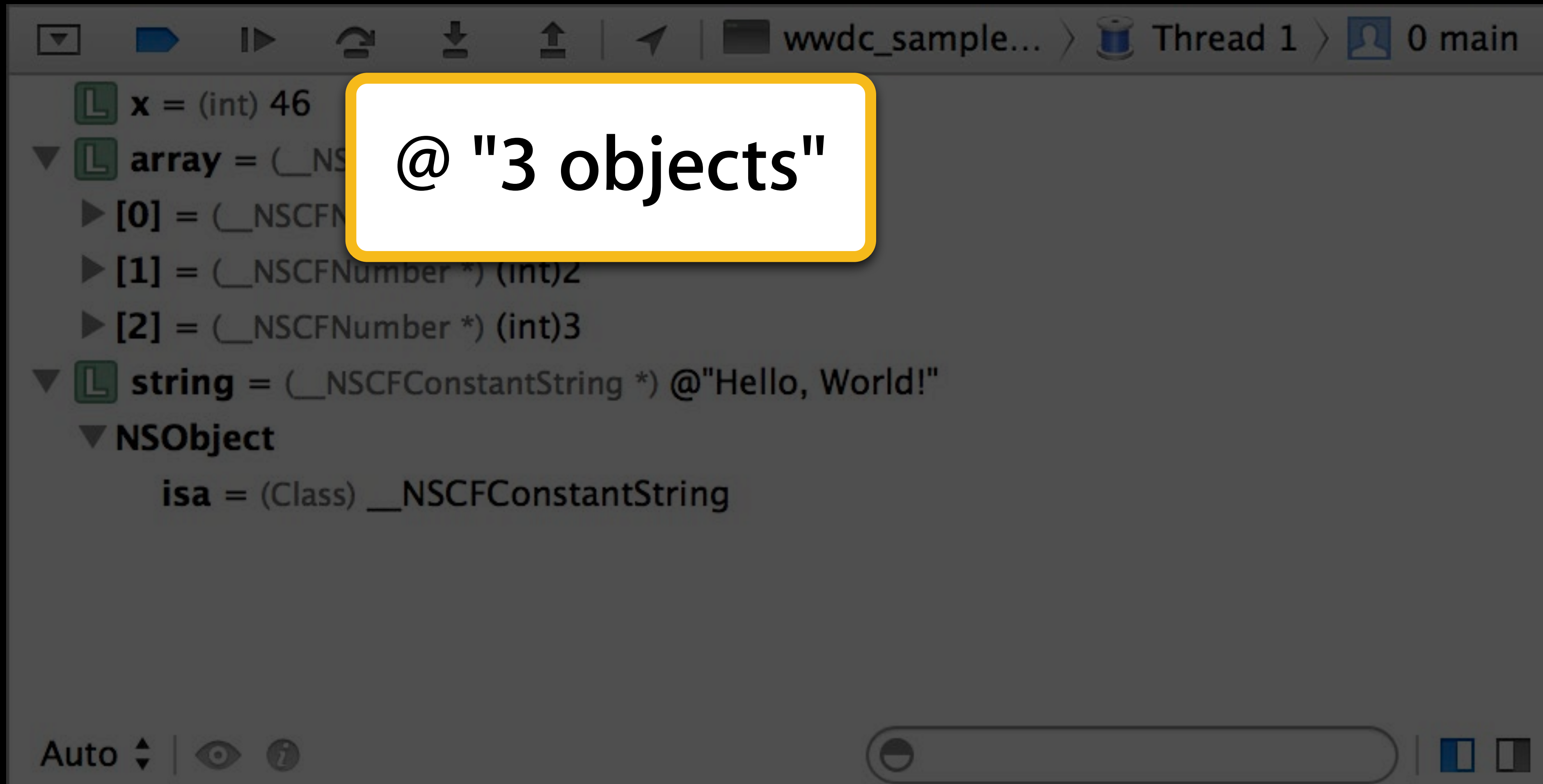
Summaries

wwdc_sample... > Thread 1 > 0 main

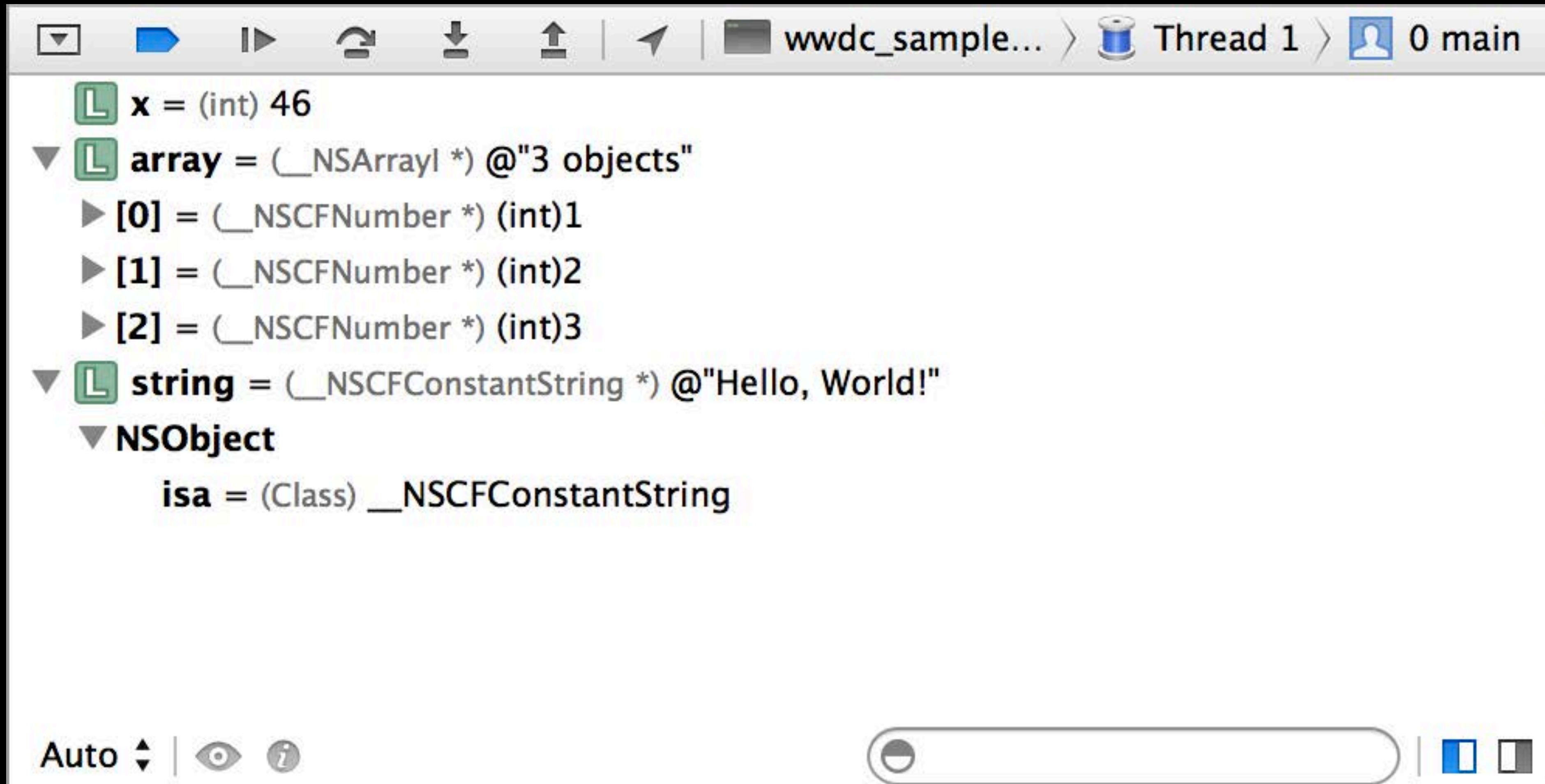
- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

Auto ▾ |

Summaries



Synthetic Children

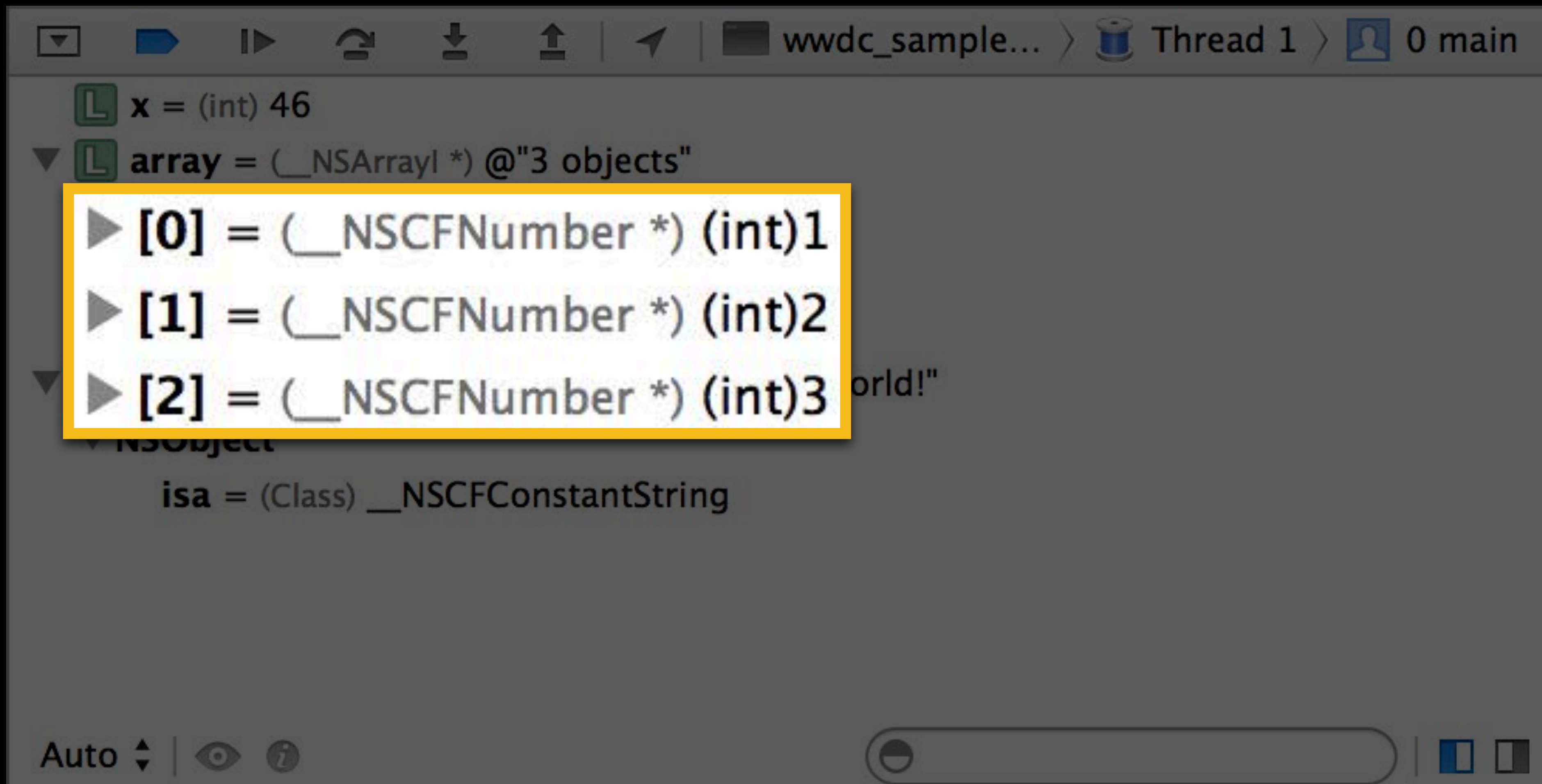


The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. Below this, the variable inspector displays the following information:

- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

At the bottom of the window, there is a control bar with the text "Auto" and a dropdown arrow, followed by an eye icon and an information icon. To the right of these icons is a horizontal scrollbar and a window management icon.

Synthetic Children



How Python Summaries Work

How Python Summaries Work

- Summaries match a type to a Python function
 - Base matching is by type name
 - Refer to LLDB web site for other rules
 - <http://lldb.llvm.org/varformats.html>

How Python Summaries Work

- Summaries match a type to a Python function
 - Base matching is by type name
 - Refer to LLDB web site for other rules
 - <http://lldb.lvm.org/varformats.html>
- The function is called whenever a value is displayed
 - LLDB passes an SBValue to it
 - Part of the LLDB Object Model
 - The function returns a string to be shown

SBValue

The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. Below this, the variable inspector displays the following information:

- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

At the bottom of the interface, there are control elements: an 'Auto' dropdown menu, a visibility icon (eye), an information icon (i), a scroll bar, and a window management icon.

SBValue

The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. A blue callout bubble points to the variable `name`. The variable's value is displayed as a tree structure:

- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

At the bottom of the debugger, there are controls for auto-continue (Auto), visibility (eye icon), and information (i icon). A scrollbar is visible on the right side.

SBValue

The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. The main area displays the following variable values:

- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

A blue callout bubble with the text "type" points to the `__NSCFNumber` type in the array elements.

At the bottom of the debugger, there are controls for "Auto" (with a double-headed arrow), a visibility icon, an information icon, a scroll bar, and window management icons.

SBValue

The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. The main area displays the following variable values:

- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

A blue callout bubble with the text "summary" is positioned over the right side of the variable list.

At the bottom of the debugger, there are controls for "Auto" (with a double-headed arrow), a visibility icon (eye), an information icon (i), a scroll bar, and a window management icon (two overlapping squares).

SBValue

The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. The main area displays the following variable values:

- `x = (int) 46`
- `array = (__NSArrayI *) @"3 objects"`
 - `[0] = (__NSCFNumber *) (int)1`
 - `[1] = (__NSCFNumber *) (int)2`
 - `[2] = (__NSCFNumber *) (int)3`
- `string = (__NSCFConstantString *) @"Hello, World!"`
 - `NSObject`
 - `isa = (Class) __NSCFConstantString`

A blue callout bubble with the text "children" points to the first element of the array, `[0] = (__NSCFNumber *) (int)1`.

At the bottom of the interface, there are controls for "Auto" (with a double-headed arrow), a visibility icon (an eye), an information icon (an 'i'), a scrollbar, and a window management icon (two overlapping squares).

SBValue

The screenshot shows the Xcode debugger interface. At the top, the breadcrumb navigation indicates the current location: `wwdc_sample...` > `Thread 1` > `0 main`. A blue callout bubble labeled `value` points to the selected variable in the variable inspector. The variable is `x`, which is an `NSArray` containing three `NSNumber` objects and one `NSString` object.

```
x = (int) 46
array = (__NSArrayI *) @"3 objects"
  [0] = (__NSCFNumber *) (int)1
  [1] = (__NSCFNumber *) (int)2
  [2] = (__NSCFNumber *) (int)3
string = (__NSCFConstantString *) @"Hello, World!"
  NSObject
    isa = (Class) __NSCFConstantString
```

At the bottom of the debugger, there are controls for `Auto` (with a double-headed arrow), a visibility icon (eye), an information icon (i), a scroll bar, and a window management icon (two overlapping rectangles).

Example

Summarizing an Address

Example

Summarizing an Address

```
def MyAddress_Summary(value, unused):
```

Example


Summarizing an Address

```
def MyAddress_Summary (value, unused) :  
    SBValue
```



Example


Summarizing an Address

```
def MyAddress_Summary(value, unused):  
    firstName = value.GetChildMemberWithName("_firstName")  
    lastName = value.GetChildMemberWithName("_lastName")
```

← SBValue

Example


Summarizing an Address

```
def MyAddress_Summary(value, unused):  
    firstName = value.GetChildMemberWithName("_firstName")  
    lastName = value.GetChildMemberWithName("_lastName")  
    firstNameSummary = firstName.GetSummary()  
    lastNameSummary = lastName.GetSummary()
```

← SBValue

Example


Summarizing an Address

```
def MyAddress_Summary(value, unused):  
    firstName = value.GetChildMemberWithName("_firstName")  
    lastName = value.GetChildMemberWithName("_lastName")  
    firstNameSummary = firstName.GetSummary()  
    lastNameSummary = lastName.GetSummary()  
  
    # process the data as you wish
```

← SBValue

Example

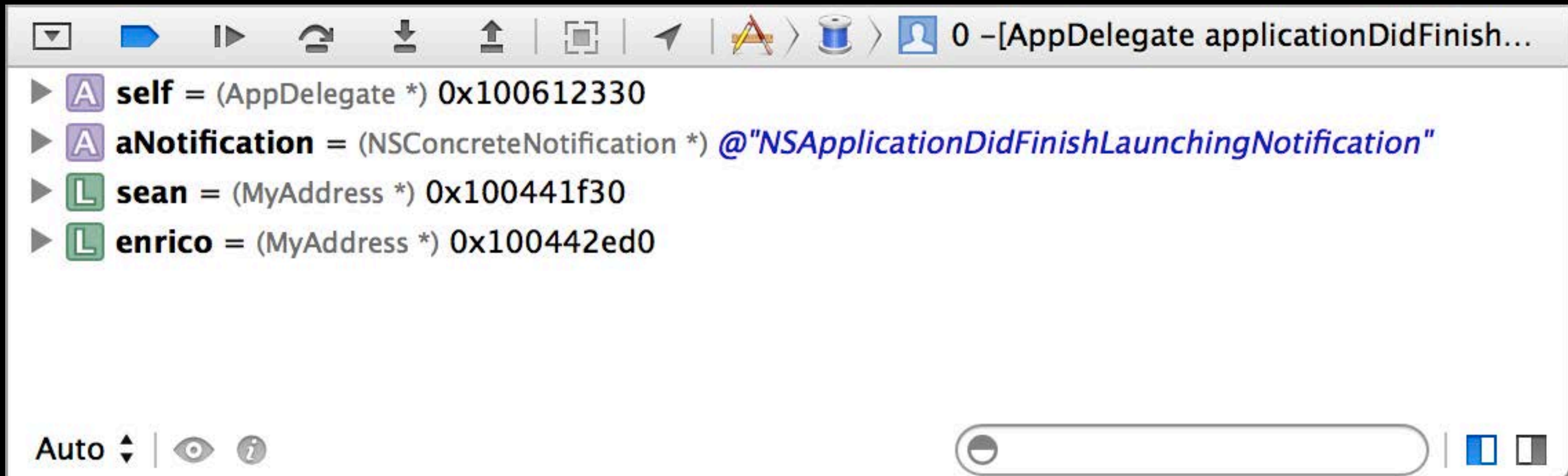
Summarizing an Address

```
def MyAddress_Summary(value, unused):  
    firstName = value.GetChildMemberWithName("_firstName")  
    lastName = value.GetChildMemberWithName("_lastName")  
    firstNameSummary = firstName.GetSummary()  
    lastNameSummary = lastName.GetSummary()  
  
    # process the data as you wish  
    return firstNameSummary + " " + lastNameSummary
```

← SBValue

Example

Summarizing an Address



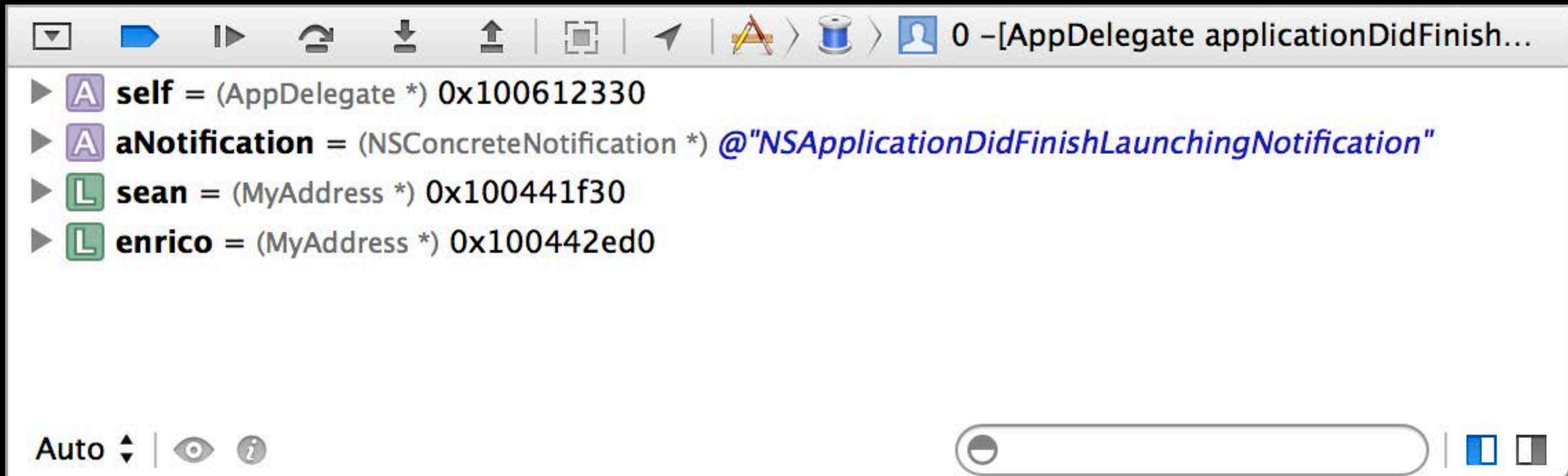
The screenshot shows the Debug Console in Xcode. The title bar indicates the current context is 0 - [AppDelegate applicationDidFinish...]. The console displays four variables:

- self** = (AppDelegate *) 0x100612330
- aNotification** = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
- sean** = (MyAddress *) 0x100441f30
- enrico** = (MyAddress *) 0x100442ed0

At the bottom of the console, there is a control bar with the text "Auto" and a dropdown arrow, an eye icon, an information icon, a scrollbar, and window management icons.

Example

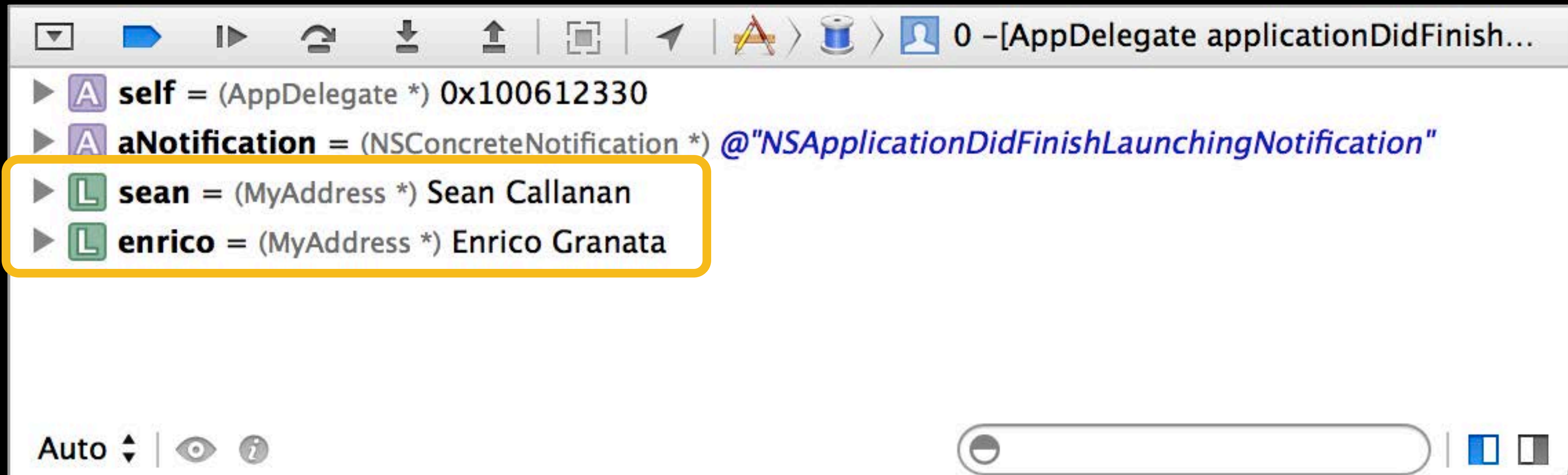
Summarizing an Address



```
ty su a MyAddress -F MyAddress_Summary  
type summary add MyAddress  
--python-function MyAddress_Summary
```

Example

Summarizing an Address



expression for Data Analysis

expression for Data Analysis

- Data types might be opaque
 - You don't have headers...
 - ...but you figured it out anyway

expression for Data Analysis

- Data types might be opaque
 - You don't have headers...
 - ...but you figured it out anyway
- How to see the additional details in the UI?

expression for Data Analysis

```
1 typedef void* Opaque;  
2  
3 Opaque makeOpaque();  
4 int useOpaque(Opaque);  
5 void freeOpaque(Opaque);
```

Opaque.h

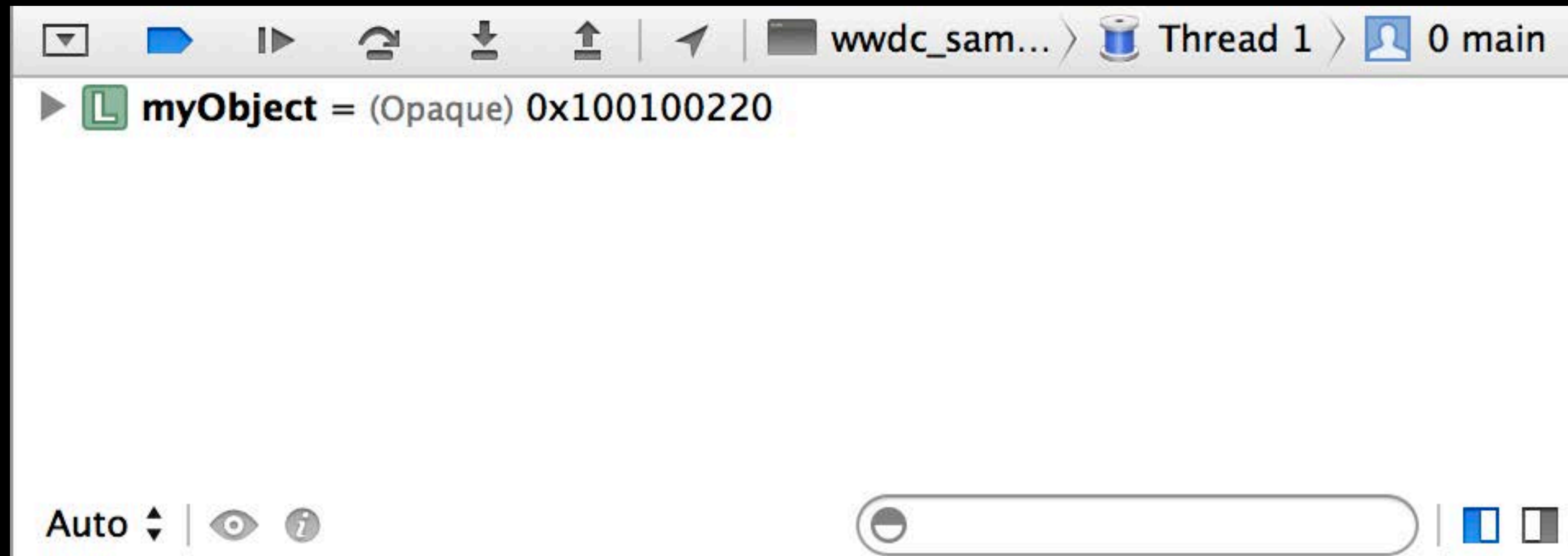
expression for Data Analysis

```
1 typedef void* Opaque;  
2  
3 Opaque makeOpaque()  
4 int useOpaque(Opaque o)  
5 void freeOpaque(Opaque o)
```

```
1 struct ImplOpaque {  
2     int aThing;  
3     float anotherThing;  
4     char* oneMoreThing;  
5 };
```

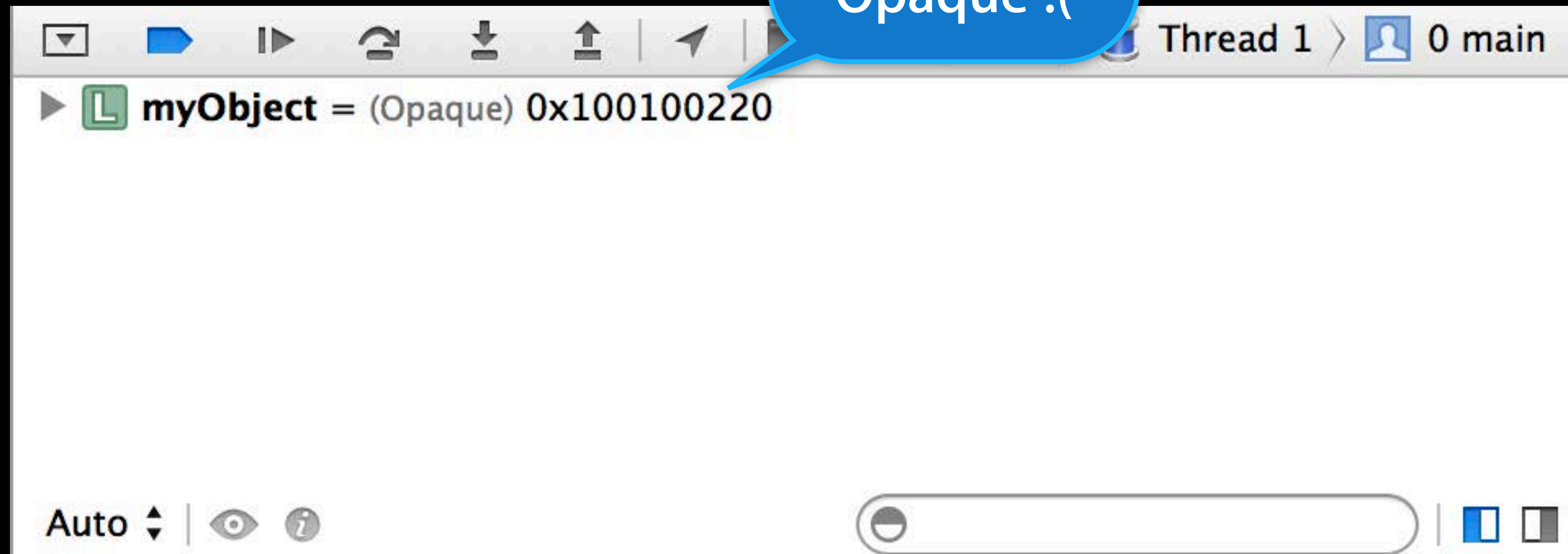
Opaque.cpp

expression for Data Analysis



expression for Data Analysis

It's really
Opaque :(

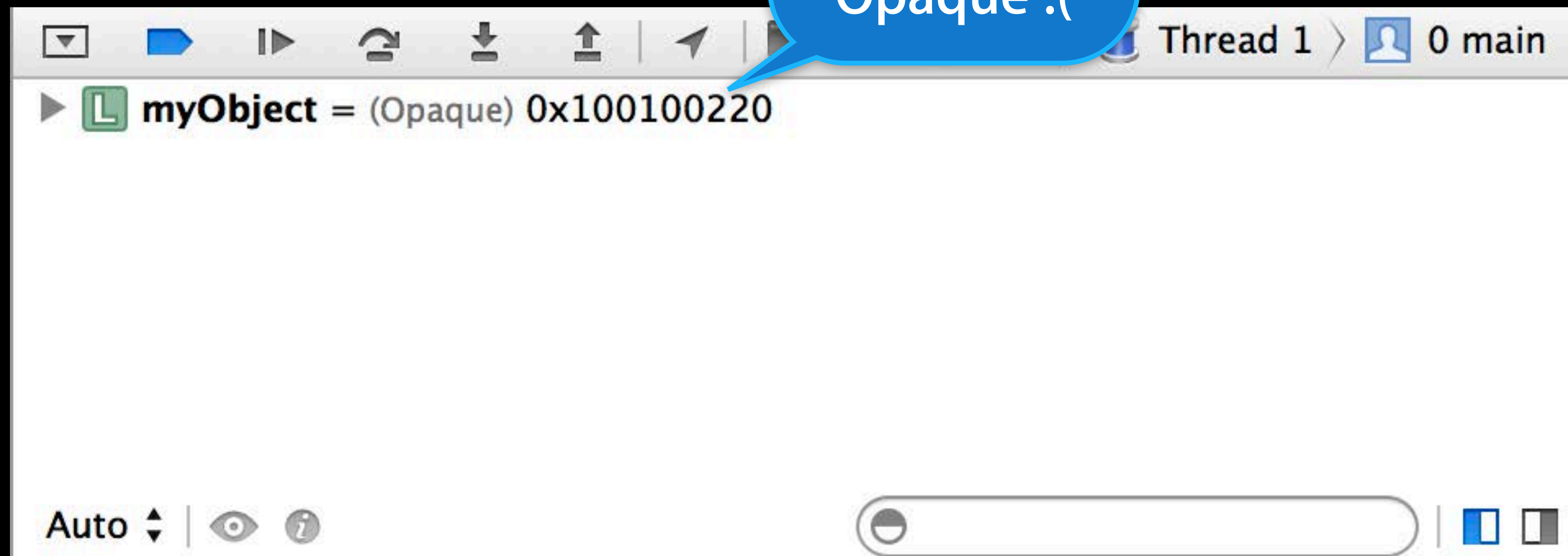


The image shows a screenshot of a debugger's variable window. At the top, there is a toolbar with various icons for navigation and execution. Below the toolbar, the variable `myObject` is displayed with its value `(Opaque) 0x100100220`. The word "Opaque" is highlighted in green. In the top right corner of the window, it says "Thread 1" and "0 main". At the bottom left, there are controls for "Auto" and a visibility icon. At the bottom right, there is a scrollbar and a zoom control.

```
▶ myObject = (Opaque) 0x100100220
```

expression for Data Analysis

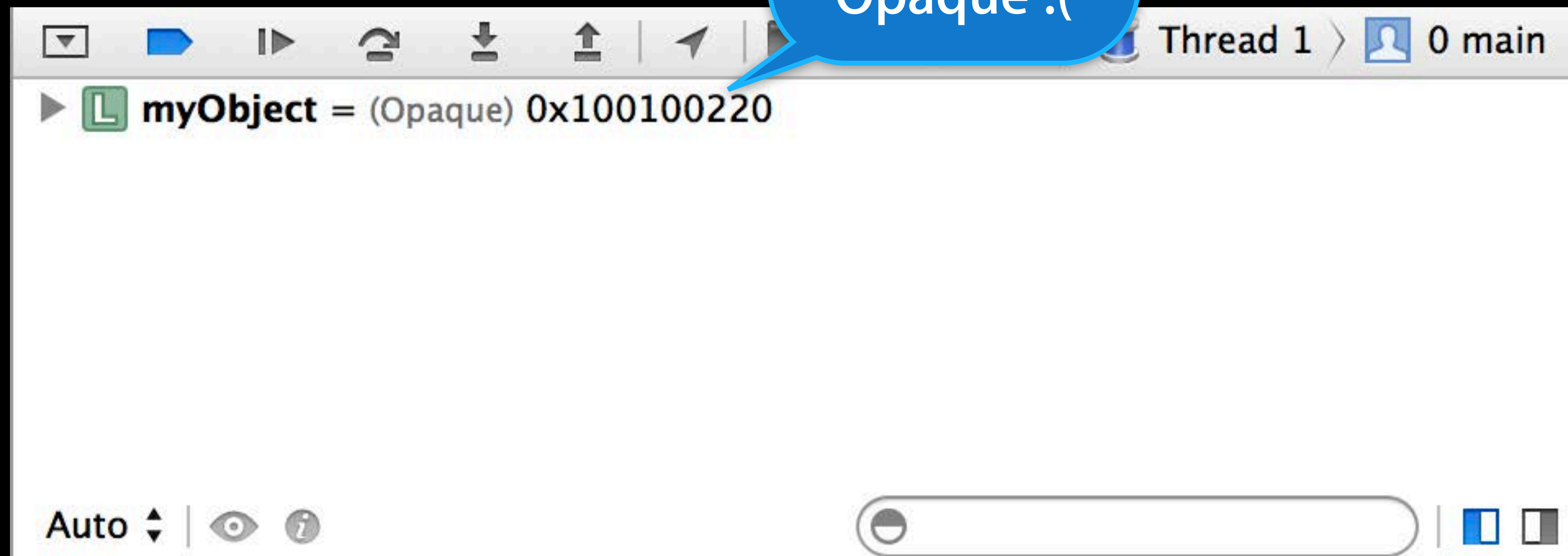
It's really
Opaque :(



```
expression  
struct $NotOpaque {  
    int item1;  
    float item2;  
    char* item3;  
};
```

expression for Data Analysis

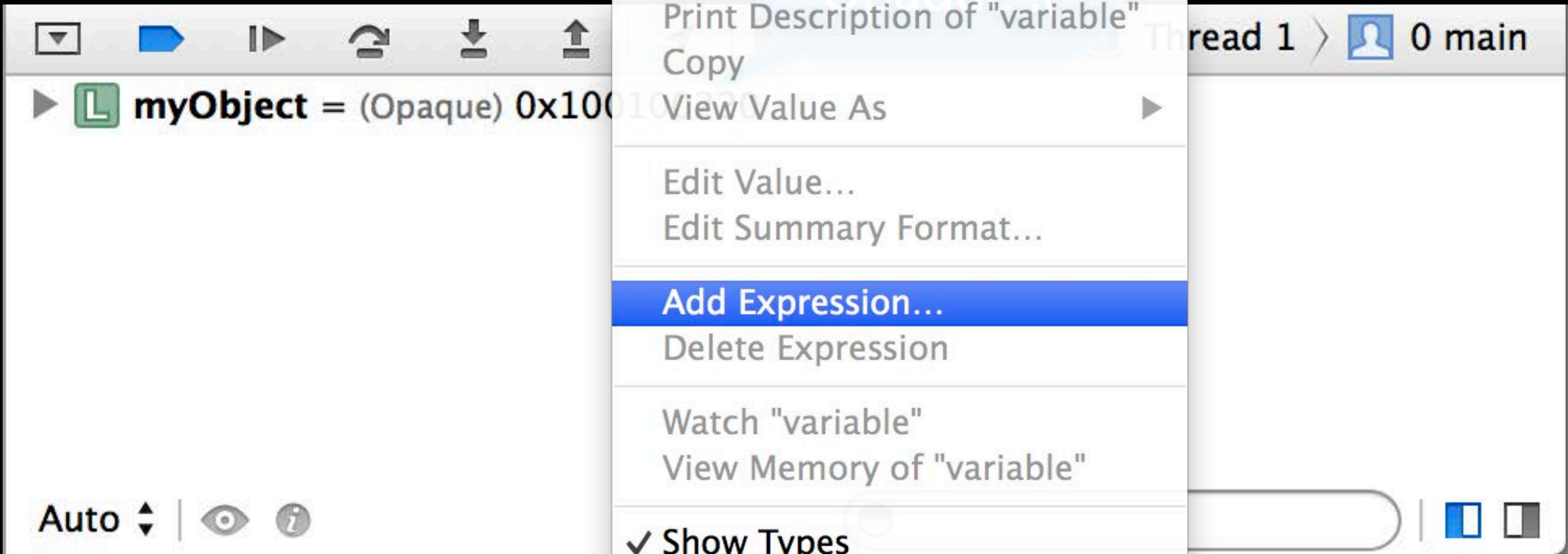
It's really
Opaque :(



```
expression  
struct $NotOpaque { ← Persistent name  
    int item1;  
    float item2;  
    char* item3;  
};
```


expression for Data Analysis

It's really
Opaque!

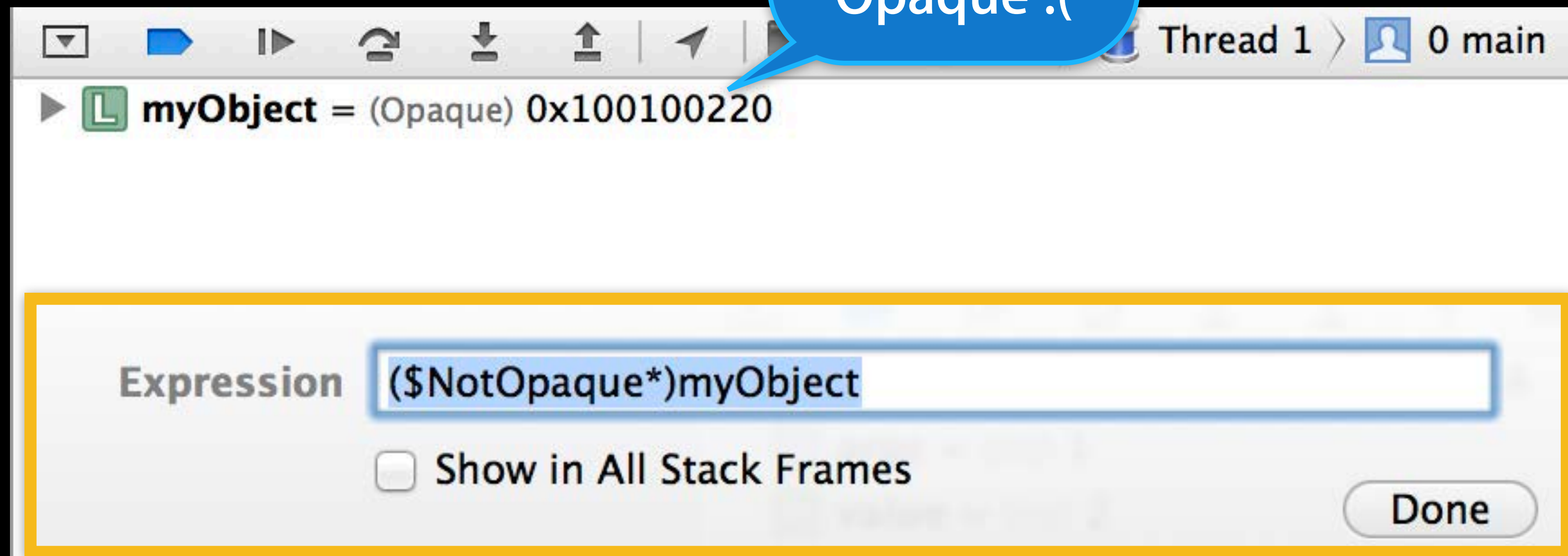


```
expression  
struct $NotOpaque {  
    int item1;  
    float item2;  
    char* item3;  
};
```

Persistent name

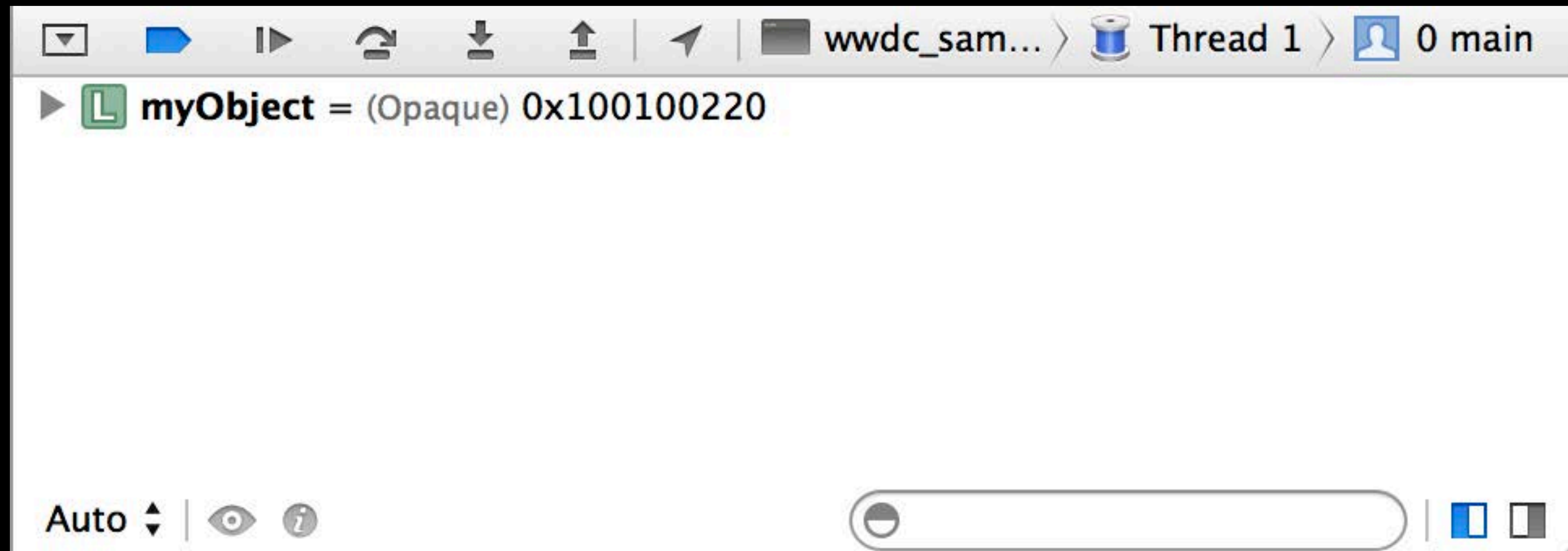
expression for Data Analysis

It's really
Opaque :(

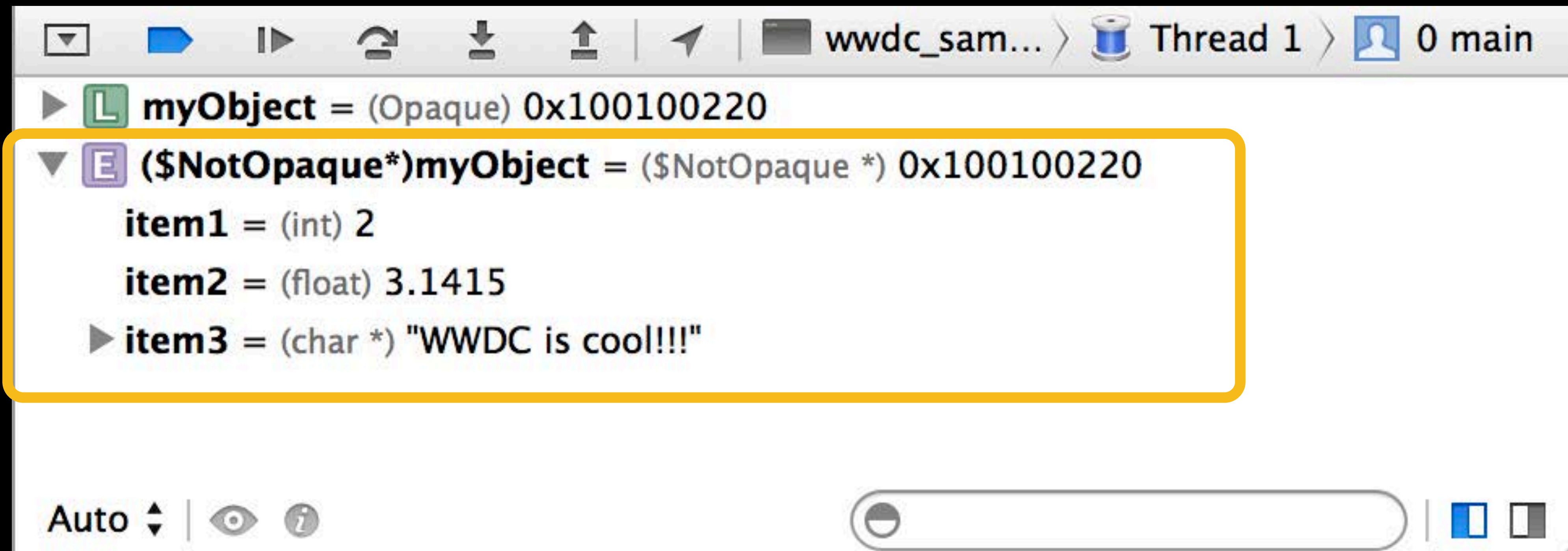


```
expression
struct $NotOpaque { ← Persistent name
    int item1;
    float item2;
    char* item3;
};
```

expression for Data Analysis



expression for Data Analysis



Extending LLDB

Making the debugger your own

Extending LLDB

Extending LLDB

- Custom LLDB commands

Extending LLDB

- Custom LLDB commands
- Breakpoint actions

Extending LLDB

- Custom LLDB commands
- Breakpoint actions
- lldbinit

Custom LLDB Commands

Custom LLDB Commands

- Create new features

Custom LLDB Commands

- Create new features
- Implement your own favorite behavior

Custom LLDB Commands

- Create new features
- Implement your own favorite behavior
- Factor out common logic

Example

Calculate depth of a recursion

Example

Calculate depth of a recursion

- Your program has a recursion

Example

Calculate depth of a recursion

- Your program has a recursion
- You need to know how deep it is

Example

Calculate depth of a recursion

- Your program has a recursion
- You need to know how deep it is
- You could count frames by hand

Example

Calculate depth of a recursion

- Your program has a recursion
- You need to know how deep it is
- You could count frames by hand
 - ...or let LLDB do it

The LLDB Object Model

The LLDB Object Model

- Called "SB" (Scripting Bridge)

The LLDB Object Model

- Called “SB” (Scripting Bridge)
 - Python API

The LLDB Object Model

- Called “SB” (Scripting Bridge)
 - Python API
- Used by Xcode to build its Debugger UI

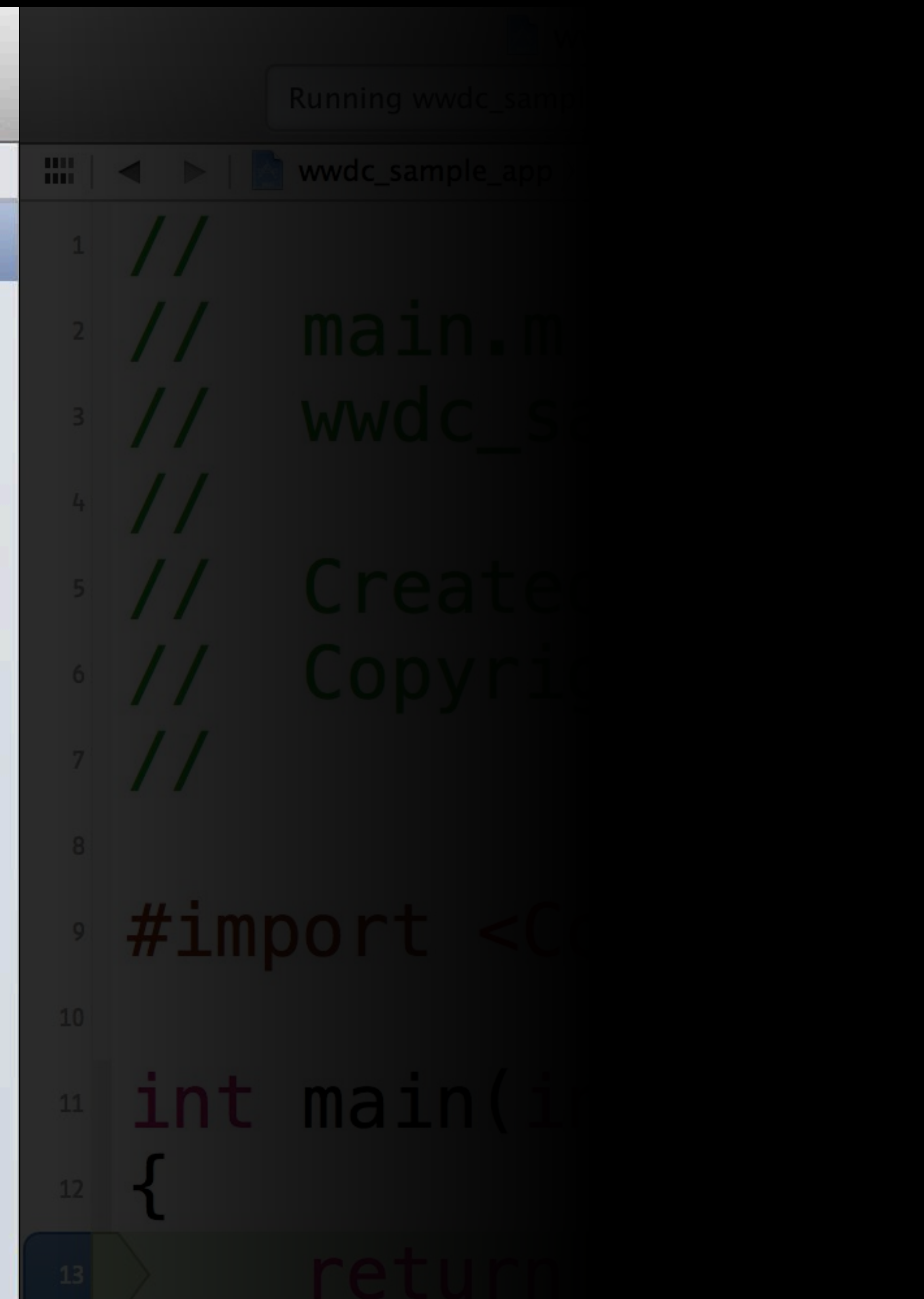
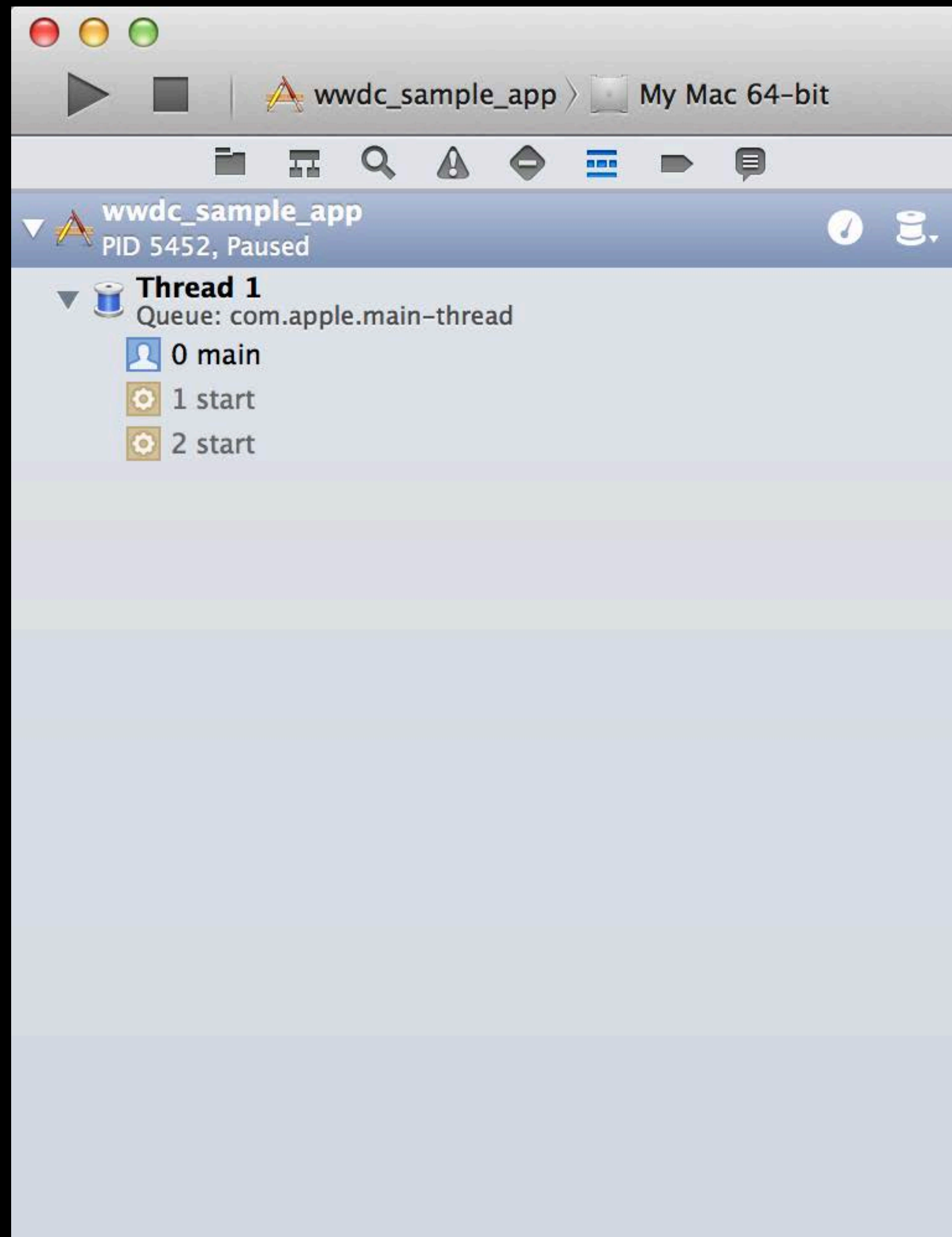
The LLDB Object Model

- Called “SB” (Scripting Bridge)
 - Python API
- Used by Xcode to build its Debugger UI
 - Full power of LLDB available for scripting

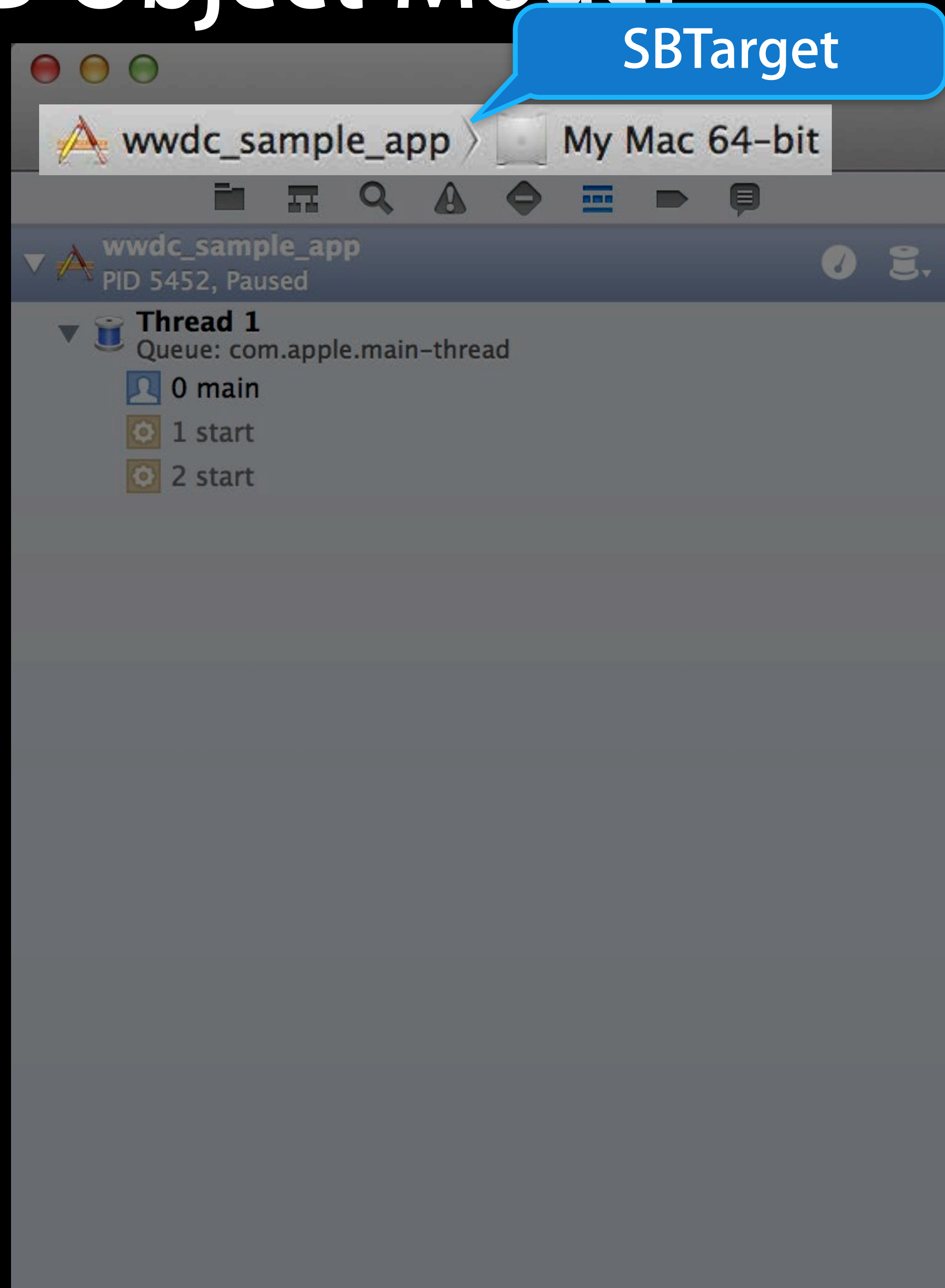
The LLDB Object Model

- Called “SB” (Scripting Bridge)
 - Python API
- Used by Xcode to build its Debugger UI
 - Full power of LLDB available for scripting
- Natural representation of a debugger session

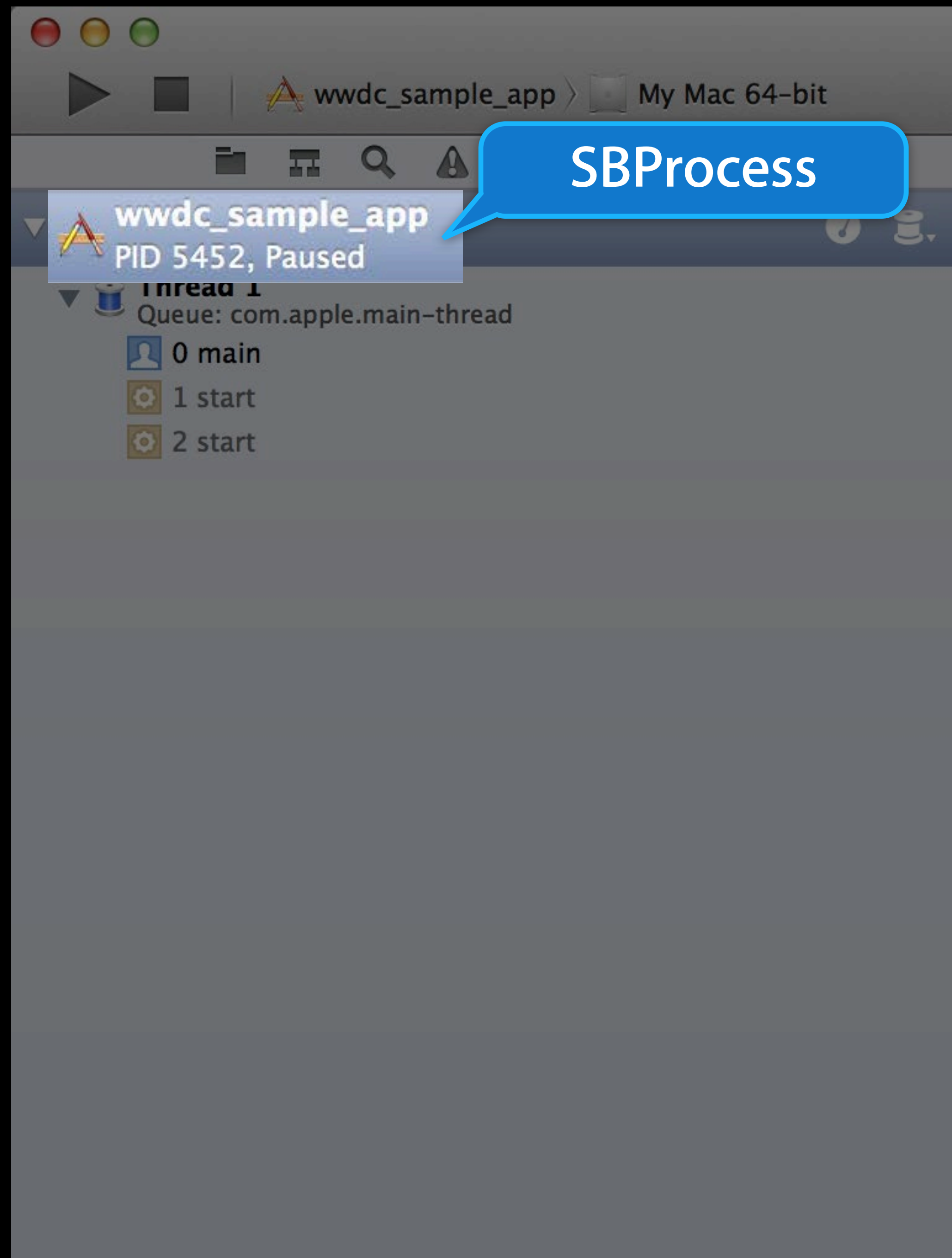
The LLDB Object Model



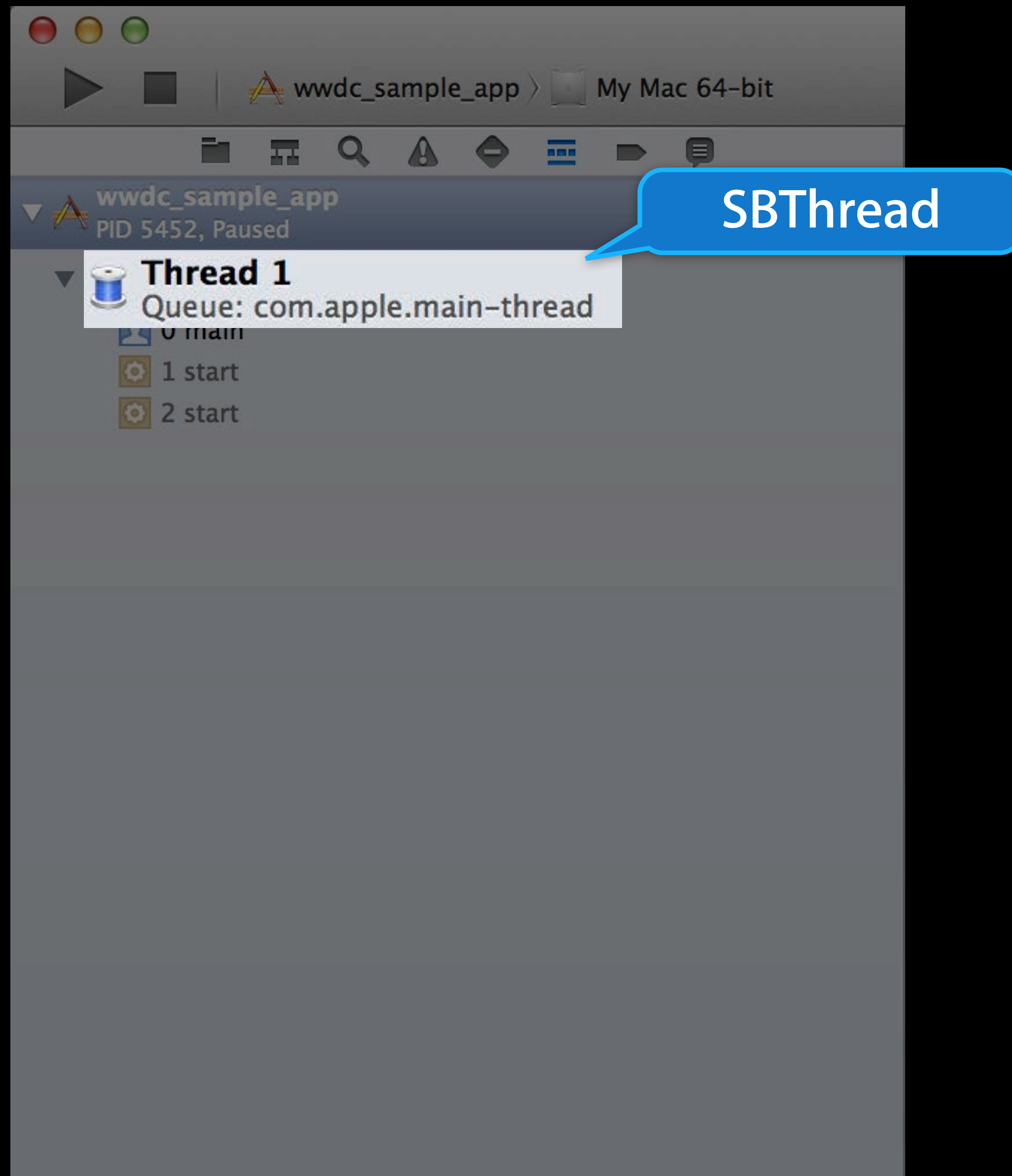
The LLDB Object Model



The LLDB Object Model



The LLDB Object Model



The LLDB Object Model



How Python Commands Work

- Commands associate a name with a Python function
 - The function is invoked whenever the command is typed

How Python Commands Work

- Commands associate a name with a Python function
 - The function is invoked whenever the command is typed

```
def MyCommand_Impl(debugger, user_input, result, unused):
```

How Python Commands Work

- Commands associate a name with a Python function
 - The function is invoked whenever the command is typed



SBDebugger

```
def MyCommand_Impl(debugger, user_input, result, unused):
```


How Python Commands Work

- Commands associate a name with a Python function
 - The function is invoked whenever the command is typed



SBDDebugger

The diagram consists of two callout boxes. The left one is yellow and contains the text 'SBDDebugger'. The right one is purple and contains the text 'Python string'. Both boxes have a tail pointing downwards towards the function signature in the code block below. The 'debugger' parameter in the code is highlighted in yellow, and 'user_input' is highlighted in purple.

Python string

```
def MyCommand_Impl(debugger, user_input, result, unused):
```

How Python Commands Work

- Commands associate a name with a Python function
 - The function is invoked whenever the command is typed

SBDebugger

Python string

SBCommandReturnObject

```
def MyCommand_Impl(debugger, user_input, result, unused):
```

How Python Commands Work

- Commands associate a name with a Python function
 - The function is invoked whenever the command is typed

SBDebugger

Python string

SBCommandReturnObject

```
def MyCommand_Impl(debugger, user_input, result, unused):
```

```
co sc a foo -f foo
command script add foo
--python-function foo
```

Example

Calculate depth of a recursion

Loop over
all frames

Check for
recursion

Display *counter*

Example

Calculate depth of a recursion

Loop over
all frames

```
for frame in thread.frames:  
    # process frame
```

Check for
recursion

Display counter

Example

Calculate depth of a recursion

Utilize LLDB
Object Model

```
thread = debugger.GetSelectedTarget() \
        .GetProcess().GetSelectedThread()
```

Loop over
all frames

```
for frame in thread.frames:  
    # process frame
```

Check for
recursion

Display *counter*

Example

Calculate depth of a recursion

Utilize LLDB
Object Model

```
thread = debugger.GetSelectedTarget() \
        .GetProcess().GetSelectedThread()
```

Loop over
all frames

```
for frame in thread.frames:
    # process frame
```

Check for
recursion

```
if frame.function.name == "MyFunction":
    # update counters
```

Display *counter*

Example

Calculate depth of a recursion

Utilize LLDB
Object Model

```
thread = debugger.GetSelectedTarget() \
        .GetProcess().GetSelectedThread()
```

Loop over
all frames

```
for frame in thread.frames:
    # process frame
```

Check for
recursion

```
if frame.function.name == "MyFunction":
    # update counters
```

Display *counter*

```
print >>result, "depth: " + str(depth)
```


Example

Calculate depth of a recursion

```
def count_depth(thread,signature,max_depth = 0):
    count = 0
    found = False
    for frame in thread:
        frame_name = frame.function.name
        if frame_name != signature:
            if found:
                return count # no indirect recursion
            else:
                pass # dive deeper
        else:
            if found:
                count += 1 # increase counter
            else:
                found = True # now we found it...
                count = 1 # ...start counting
    return count

def Depth_Command_Impl(debugger,user_input,result,unused):
    thread = debugger.GetSelectedTarget().GetProcess().GetSelectedThread()
    name = thread.GetFrameAtIndex(0).function.name
    print >>result,"depth: " + str(count_depth(thread,name,0))
```

Example

Calculate depth of a recursion

The screenshot shows the Thread View in Xcode for the application 'wwdc_sample_app' (PID 5916, Paused). Under 'Thread 1' (Queue: com.apple.main-thread), a list of 23 stack frames is displayed, representing a recursive call stack. The frames are numbered 0 through 22. Frames 0 through 19 are all '-[MyTreeNode traverseWithCallback:]', indicating a deep recursion. Frame 20 is '-[MyTree traverseWithCallback:]', frame 21 is 'main', and frame 22 is 'start'.

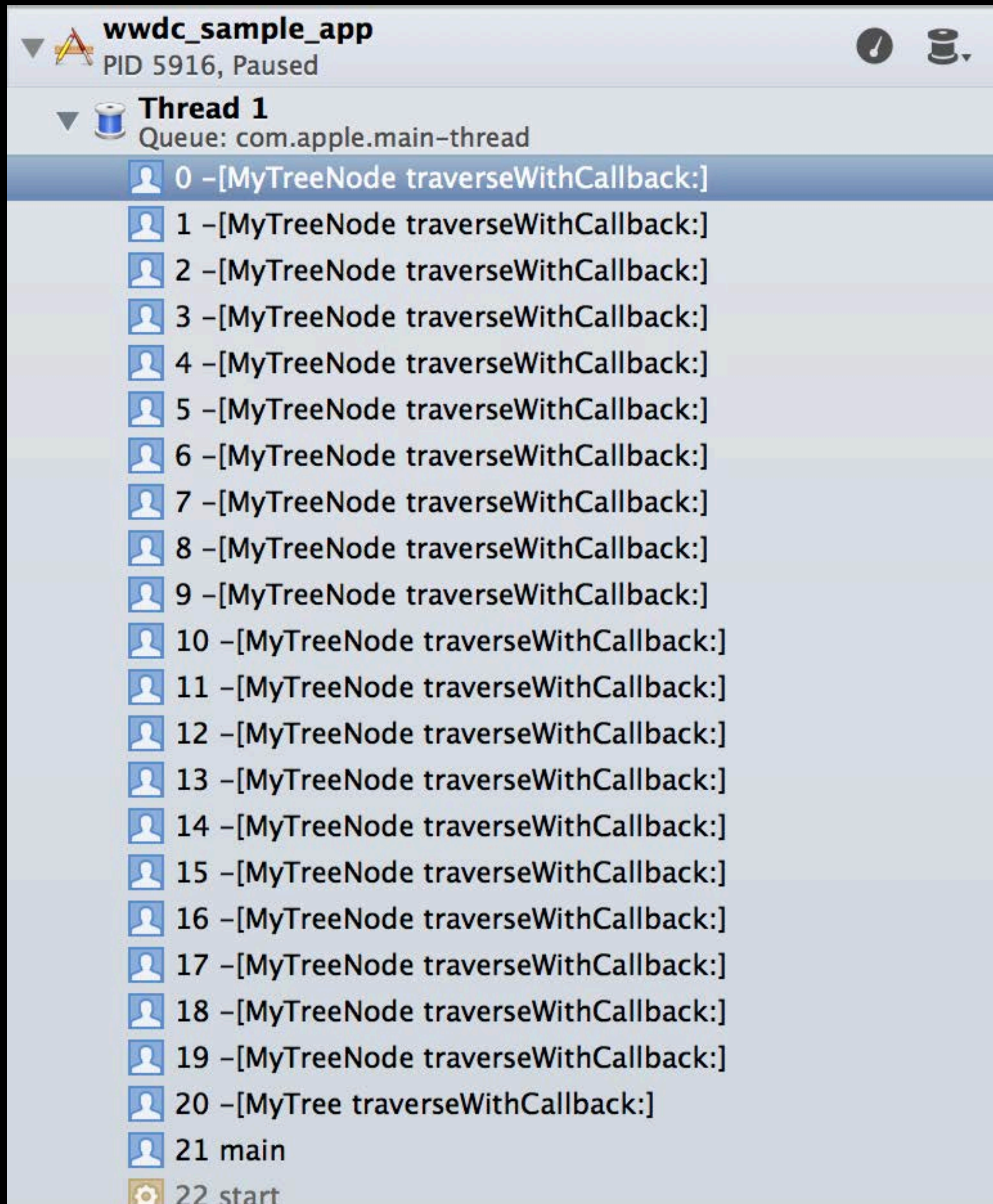
- 0 -[MyTreeNode traverseWithCallback:]
- 1 -[MyTreeNode traverseWithCallback:]
- 2 -[MyTreeNode traverseWithCallback:]
- 3 -[MyTreeNode traverseWithCallback:]
- 4 -[MyTreeNode traverseWithCallback:]
- 5 -[MyTreeNode traverseWithCallback:]
- 6 -[MyTreeNode traverseWithCallback:]
- 7 -[MyTreeNode traverseWithCallback:]
- 8 -[MyTreeNode traverseWithCallback:]
- 9 -[MyTreeNode traverseWithCallback:]
- 10 -[MyTreeNode traverseWithCallback:]
- 11 -[MyTreeNode traverseWithCallback:]
- 12 -[MyTreeNode traverseWithCallback:]
- 13 -[MyTreeNode traverseWithCallback:]
- 14 -[MyTreeNode traverseWithCallback:]
- 15 -[MyTreeNode traverseWithCallback:]
- 16 -[MyTreeNode traverseWithCallback:]
- 17 -[MyTreeNode traverseWithCallback:]
- 18 -[MyTreeNode traverseWithCallback:]
- 19 -[MyTreeNode traverseWithCallback:]
- 20 -[MyTree traverseWithCallback:]
- 21 main
- 22 start

(lldb)

All Output ⇅

Example

Calculate depth of a recursion

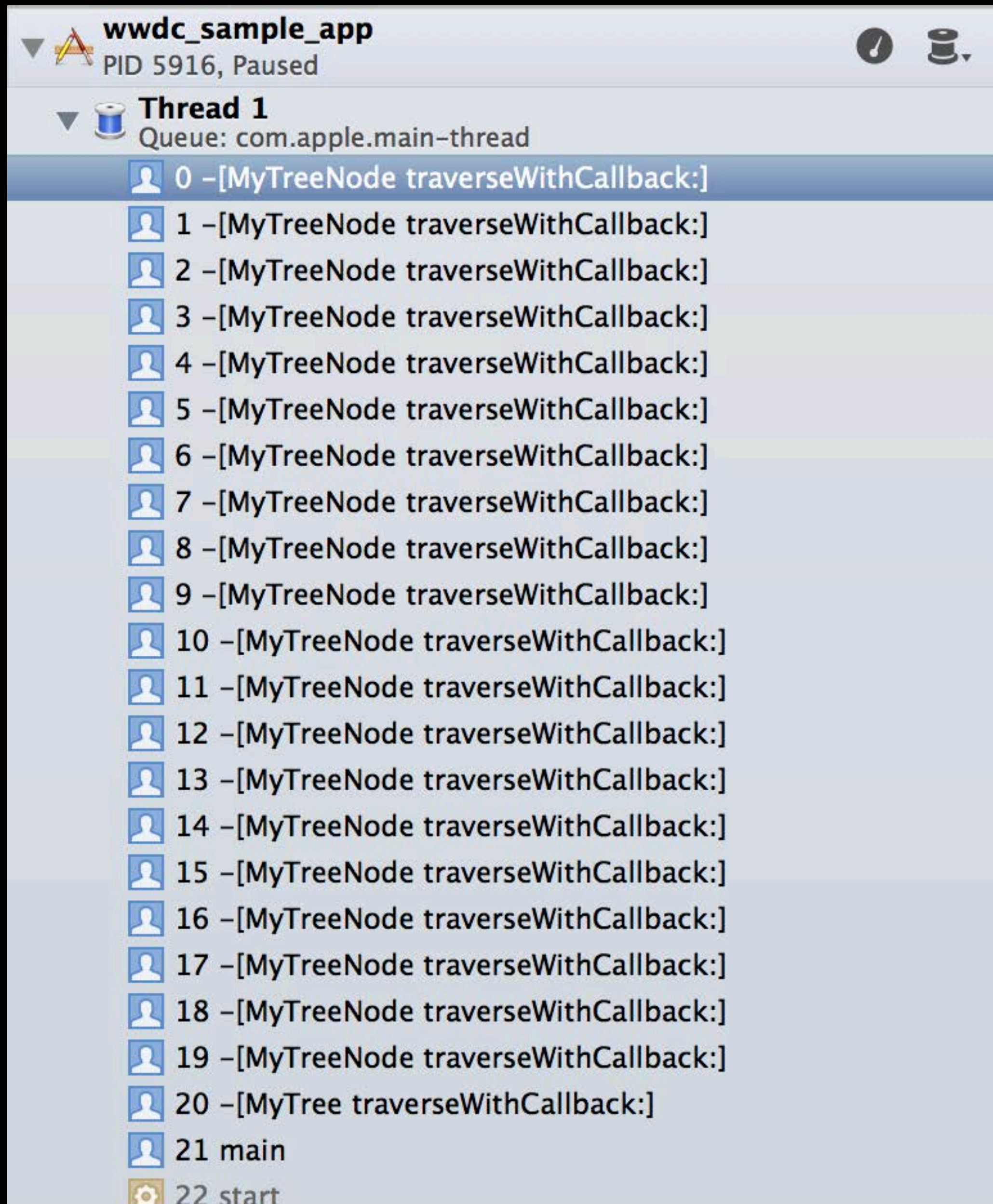


(lldb) depth

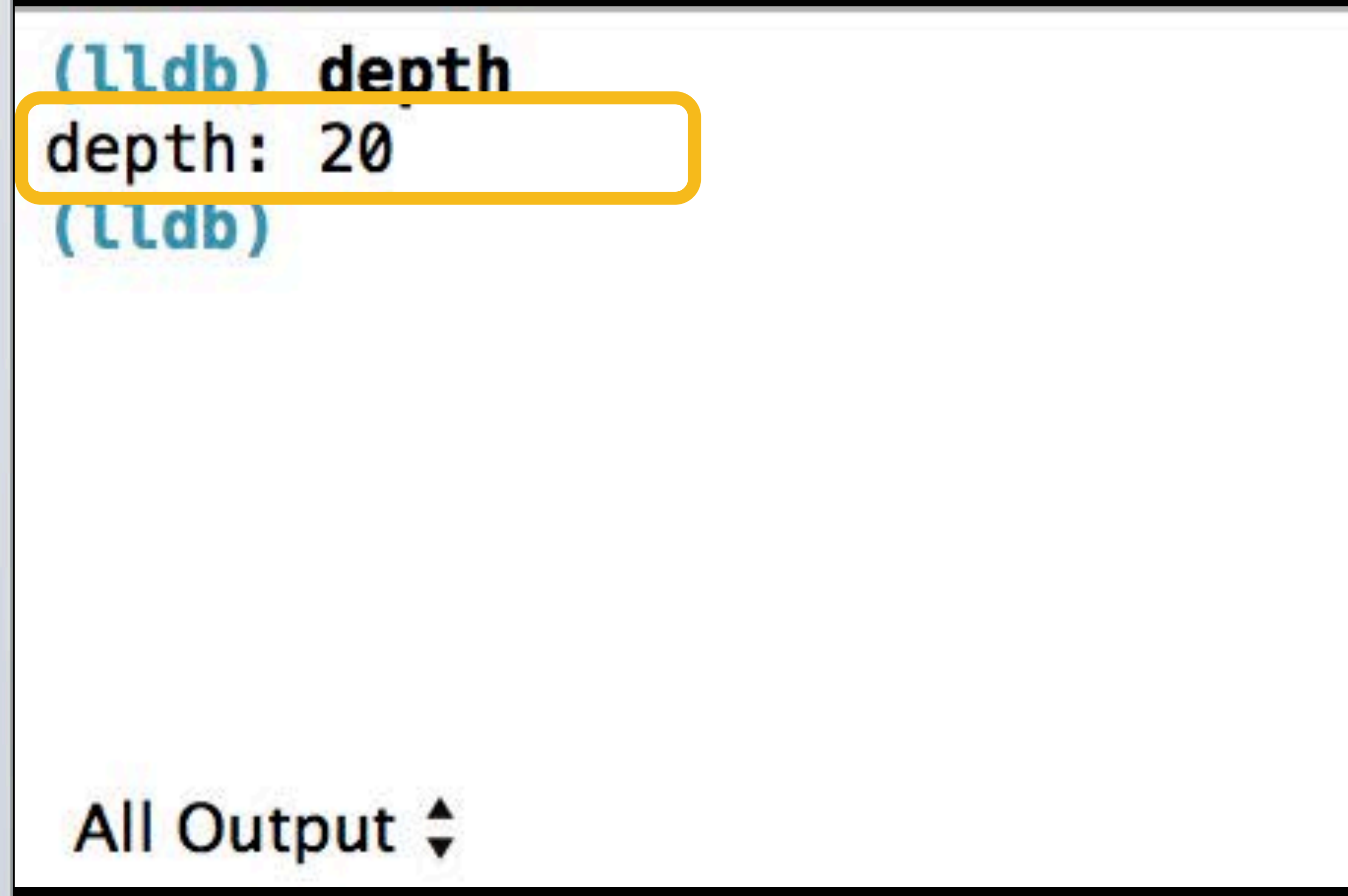
All Output ⇅

Example

Calculate depth of a recursion



The screenshot shows the Call Stack for the application 'wwdc_sample_app' (PID 5916, Paused). The stack is expanded to show 'Thread 1' (Queue: com.apple.main-thread). The stack contains 23 frames, with the top frame being the current function call: '0 -[MyTreeNode traverseWithCallback:]'. The frames below it represent recursive calls, numbered 1 through 20, all with the same method name. Frame 21 is 'main' and frame 22 is 'start'.



The screenshot shows the LLDB console output. The prompt '(lldb) depth' is followed by the output 'depth: 20'. The output is highlighted with a yellow box. Below the output, the prompt '(lldb)' is visible. At the bottom of the console, there is a button labeled 'All Output' with a double-headed arrow icon.

Breakpoint Actions

Breakpoint Actions

- Breakpoints are powerful

Breakpoint Actions

- Breakpoints are powerful
 - But their default behavior is to always stop

Breakpoint Actions

- Breakpoints are powerful
 - But their default behavior is to always stop
- Conditional breakpoints improve a lot

Breakpoint Actions

- Breakpoints are powerful
 - But their default behavior is to always stop
- Conditional breakpoints improve a lot
 - But they can't access the LLDB object model

Breakpoint Actions

- Breakpoints are powerful
 - But their default behavior is to always stop
- Conditional breakpoints improve a lot
 - But they can't access the LLDB object model
- Breakpoint actions allow full program inspection

Breakpoint Actions

- Breakpoints are powerful
 - But their default behavior is to always stop
- Conditional breakpoints improve a lot
 - But they can't access the LLDB object model
- Breakpoint actions allow full program inspection
 - Code + data + object model

How Breakpoint Actions Work

- Breakpoint actions associate a breakpoint with a Python function
 - The function is invoked whenever the breakpoint is hit
 - The function can return False to tell LLDB to continue your program

How Breakpoint Actions Work

- Breakpoint actions associate a breakpoint with a Python function
 - The function is invoked whenever the breakpoint is hit
 - The function can return False to tell LLDB to continue your program

```
def break_on_deep_traversal(frame, location, unused):
```

How Breakpoint Actions Work

- Breakpoint actions associate a breakpoint with a Python function
 - The function is invoked whenever the breakpoint is hit
 - The function can return False to tell LLDB to continue your program



SBFrame

```
def break_on_deep_traversal(frame, location, unused):
```

How Breakpoint Actions Work

- Breakpoint actions associate a breakpoint with a Python function
 - The function is invoked whenever the breakpoint is hit
 - The function can return False to tell LLDB to continue your program



SBFrame

SBBreakpointLocation

```
def break_on_deep_traversal(frame, location, unused):
```

How Breakpoint Actions Work

- Breakpoint actions associate a breakpoint with a Python function
 - The function is invoked whenever the breakpoint is hit
 - The function can return False to tell LLDB to continue your program

SBFrame

SBBreakpointLocation

```
def break_on_deep_traversal(frame, location, unused):
```

```
br co a -s p -F foo 1  
breakpoint command add --script python  
--python-function foo 1
```


Example

Stop if a recursion is more than n levels deep

Example

Stop if a recursion is more than n levels deep

- Your program hangs while doing a recursive task

Example

Stop if a recursion is more than n levels deep

- Your program hangs while doing a recursive task
 - You don't know the exact cause
 - Behavior is hard to reproduce

Example

Stop if a recursion is more than n levels deep

- Your program hangs while doing a recursive task
 - You don't know the exact cause
 - Behavior is hard to reproduce
- Idea!

Example

Stop if a recursion is more than n levels deep

- Your program hangs while doing a recursive task
 - You don't know the exact cause
 - Behavior is hard to reproduce
- Idea!
 - Make a breakpoint action that looks at the call stack
 - Have LLDB stop only when the recursion is getting too deep

Example

Stop if a recursion is more than n levels deep

Count recursion
depth

Break if
counter \geq *threshold*

Example

Stop if a recursion is more than n levels deep

Count recursion
depth



Break if
counter \geq *threshold*

Example

Stop if a recursion is more than n levels deep

Count recursion
depth



Break if
counter \geq *threshold*

```
if count_depth(frame.thread, "MyFunction") < threshold:  
    return False
```

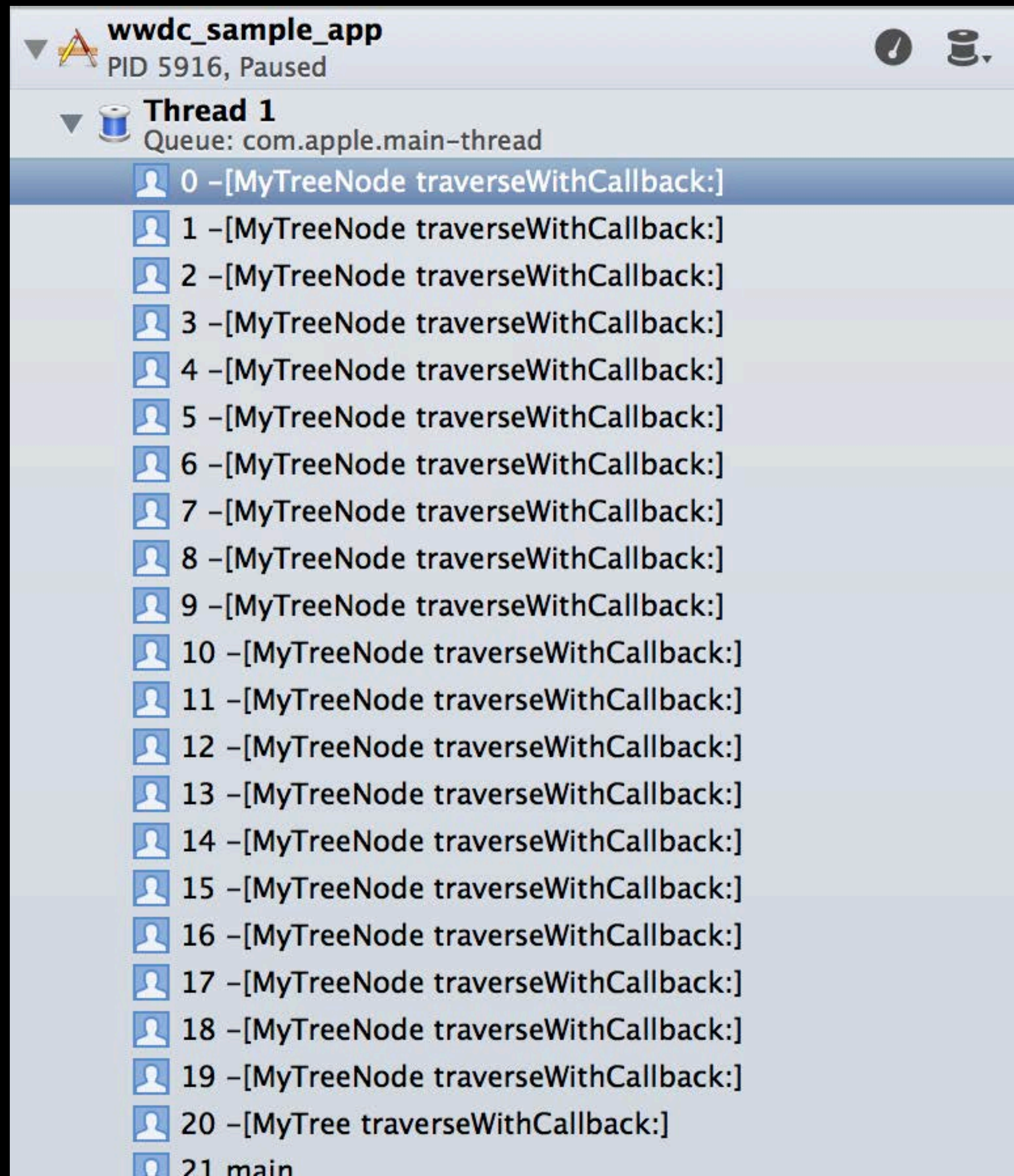

Example

Stop if a recursion is more than n levels deep

```
def break_on_deep_traversal(frame, location, unused):  
    name = "[MyTreeNode traverseWithCallback:]"  
    threshold = 20  
    return count_depth(frame.thread, name, threshold) >= threshold
```

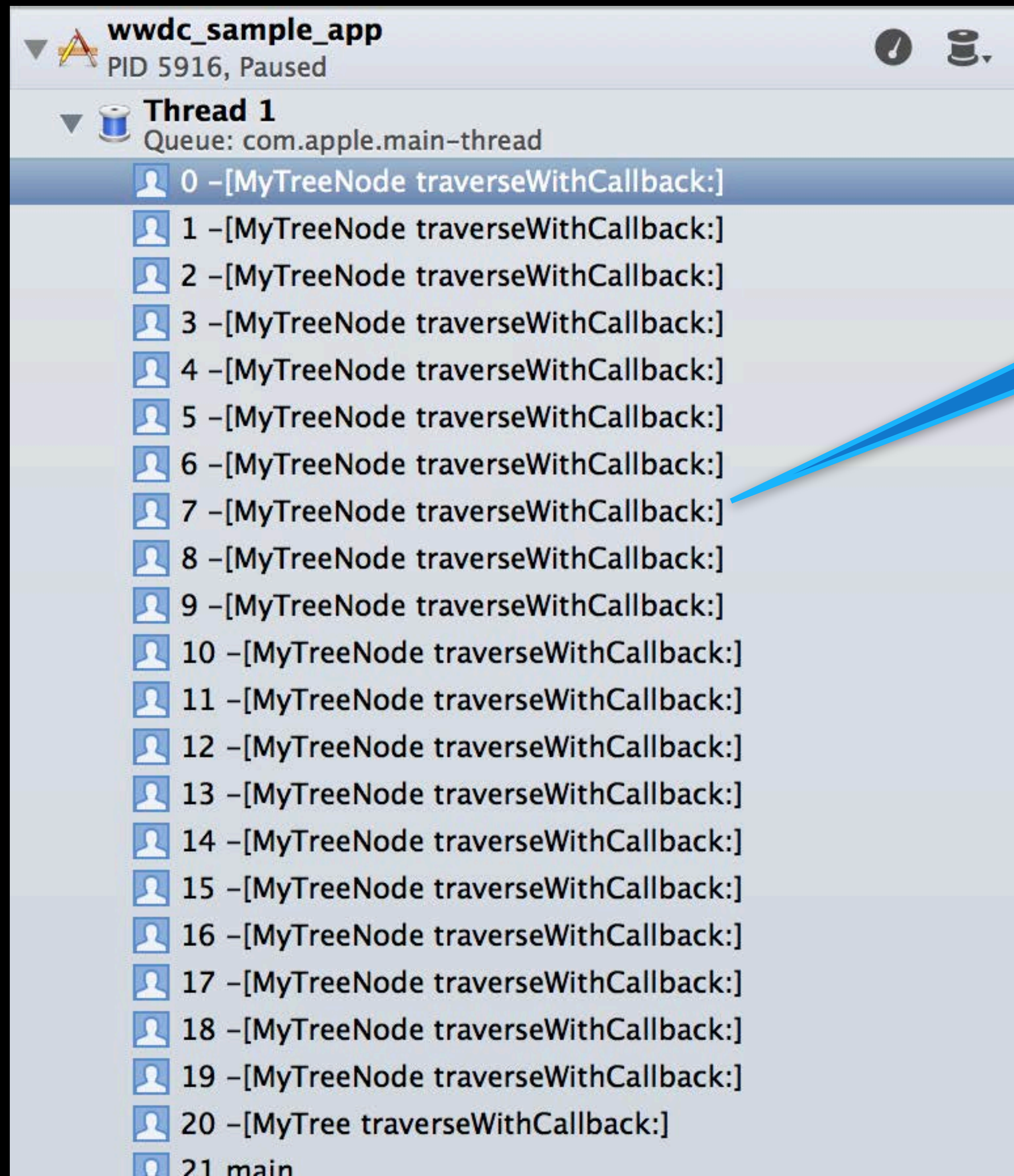
Example

Stop if a recursion is more than n levels deep



Example

Stop if a recursion is more than n levels deep



Stopped at
20th call

Productizing Customizations

Productizing Customizations

- LLDB-specific configuration file

Productizing Customizations

- LLDB-specific configuration file
 - ~/.lldbinit

Productizing Customizations

- LLDB-specific configuration file
 - ~/.lldbinit
- Loaded at debugger startup

Productizing Customizations

- LLDB-specific configuration file
 - ~/.lldbinit
- Loaded at debugger startup
- Useful to tweak debugger settings

Productizing Customizations

- LLDB-specific configuration file
 - ~/.lldbinit
- Loaded at debugger startup
- Useful to tweak debugger settings
 - Or load commonly used scripts

Productizing Customizations

- LLDB-specific configuration file
 - ~/.lldbinit
- Loaded at debugger startup
- Useful to tweak debugger settings
 - Or load commonly used scripts
- Xcode-specific version

Productizing Customizations

- LLDB-specific configuration file
 - `~/.lldbinit`
- Loaded at debugger startup
- Useful to tweak debugger settings
 - Or load commonly used scripts
- Xcode-specific version
 - `~/.lldbinit-Xcode`

Summary

Summary

- LLDB is **the** debugger
 - More efficient
 - New features

Summary

- LLDB is **the** debugger
 - More efficient
 - New features
- Debug effectively
 - Use logging and assertions wisely
 - Set the right breakpoints

Summary

- LLDB is **the** debugger
 - More efficient
 - New features
- Debug effectively
 - Use logging and assertions wisely
 - Set the right breakpoints
- Exploit customization
 - Data formatters provide more meaningful views of data
 - Automate repeated workflows

More Information

Dave DeLong

App Frameworks Evangelist
delong@apple.com

Documentation

LLDB Quick Start

LLDB Website

<http://lldb.llvm.org>

LLDB Help

help / apropos

Apple Developer Forums

<http://devforums.apple.com>

Related Sessions

What's New in Xcode 5

Presidio
Tuesday 9:00AM

Debugging with Xcode

Pacific Heights
Wednesday 2:00PM

Labs

LLDB and Instruments Lab

Tools Lab C
Friday 10:15AM



 WWDC2013