



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

Unit Testing in Xcode

iTunes
WWDC 2012

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

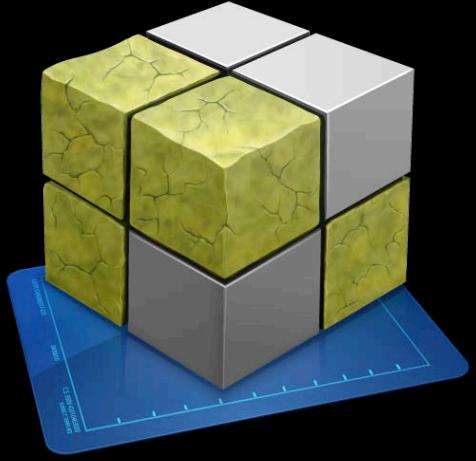
Log Effectively with ASL

- 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	expression --object-description -- foo
▪ Abbreviated form	e -0 -- foo
▪ Alias	po foo

- We will use this notation:

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

Command Syntax

A quick recap

- Commands can have three forms:

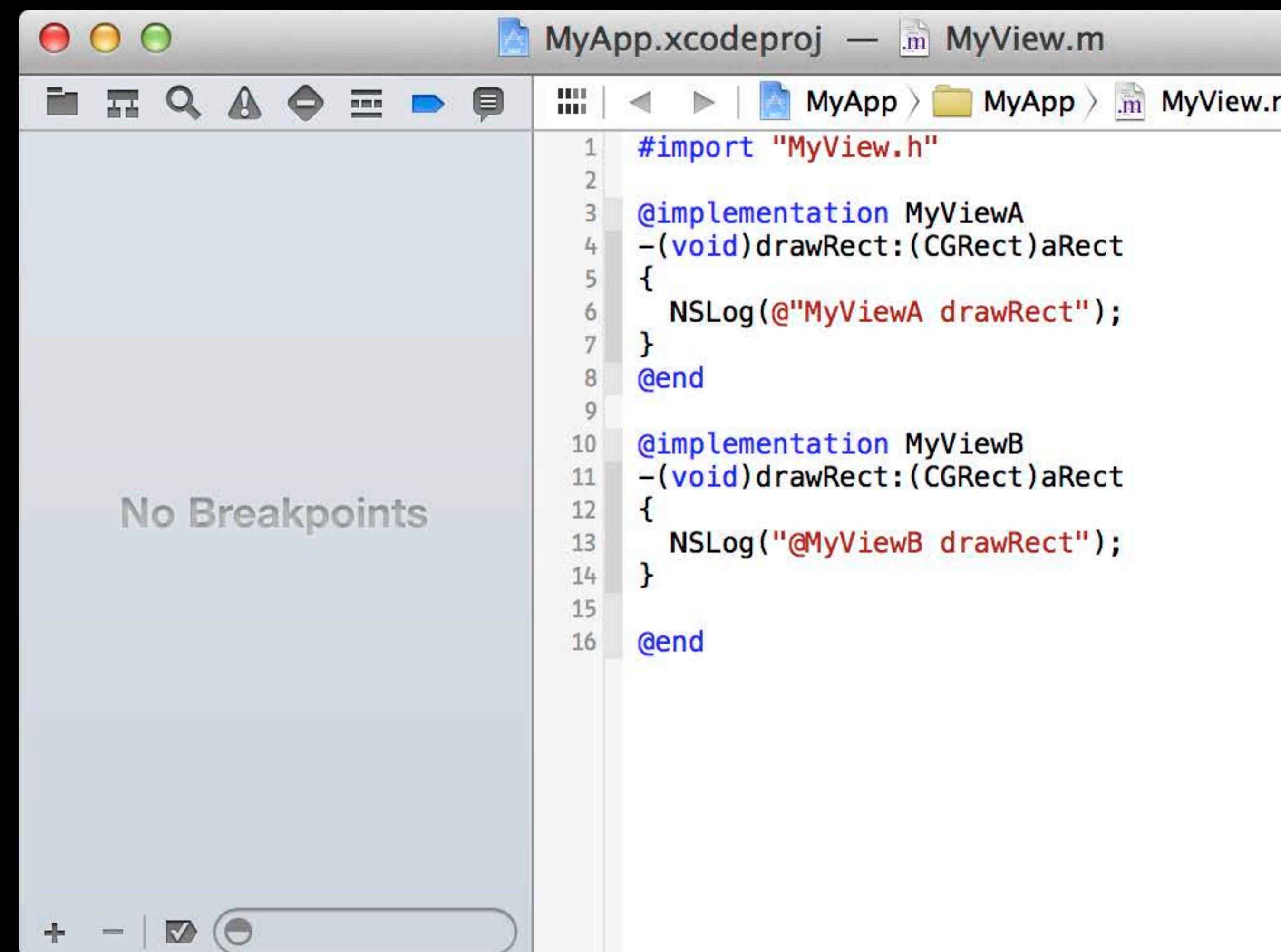
▪ Discoverable form	expression --object-description -- foo
▪ Abbreviated form	e -0 -- foo
▪ Alias	po foo

- We will use this notation:

po foo ← Shortest possible form

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

Common Breakpoint Scenarios



The screenshot shows a Xcode interface with a code editor window. The title bar reads "MyApp.xcodeproj — MyView.m". The code editor displays the following Objective-C code:

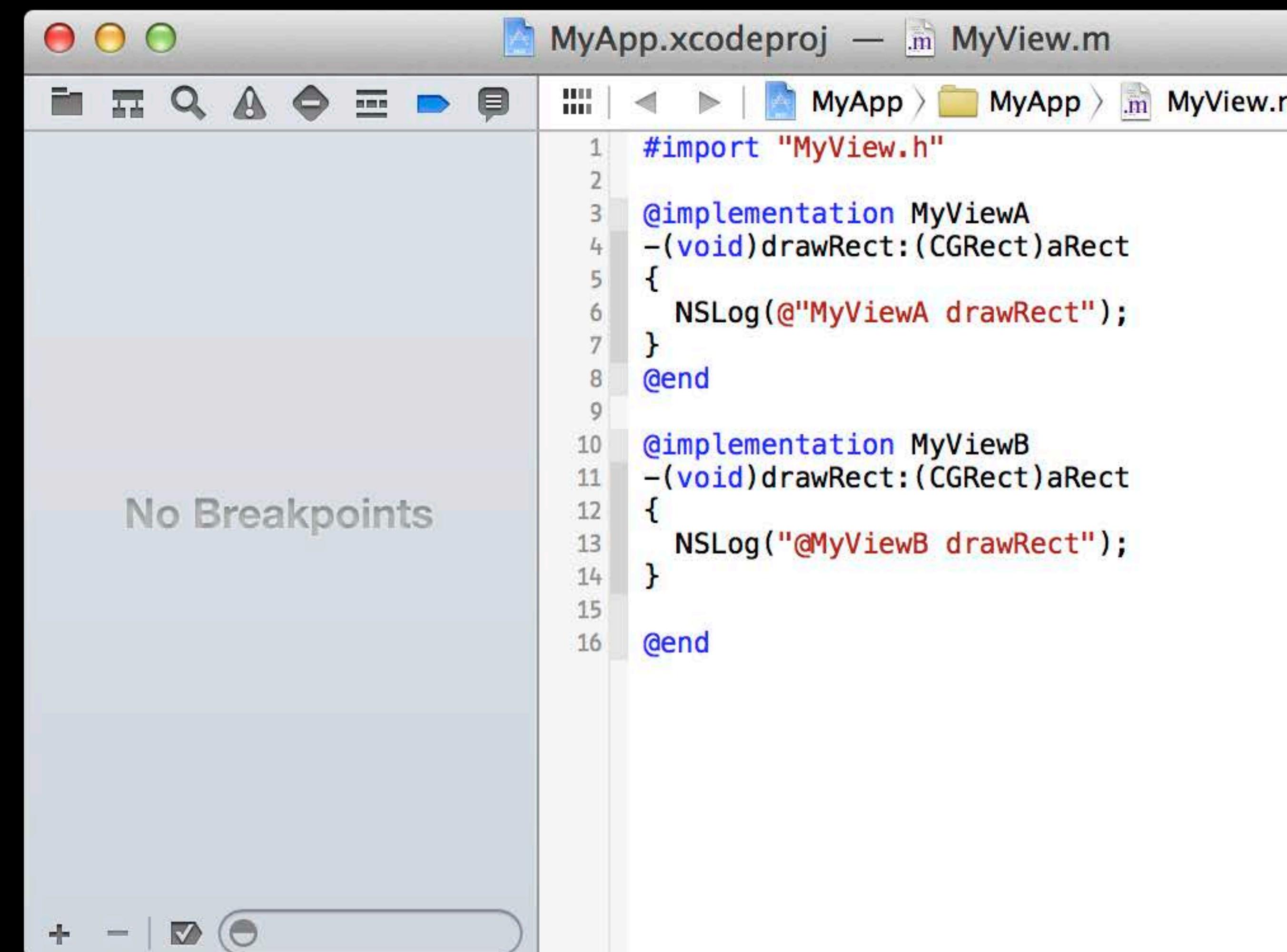
```
1 #import "MyView.h"
2
3 @implementation MyViewA
4 -(void)drawRect:(CGRect)aRect
5 {
6     NSLog(@"MyViewA drawRect");
7 }
8 @end
9
10 @implementation MyViewB
11 -(void)drawRect:(CGRect)aRect
12 {
13     NSLog(@"MyViewB drawRect");
14 }
15 @end
16
```

The code editor has a sidebar on the left with the message "No Breakpoints". At the bottom of the code editor, there is a toolbar with icons for file operations and a search field.

Common Breakpoint Scenarios

- Stop at a source line:

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



The screenshot shows the Xcode interface with a project named "MyApp.xcodeproj" and a file "MyView.m" selected. The code editor displays the following implementation of `drawRect:` for two different view classes:

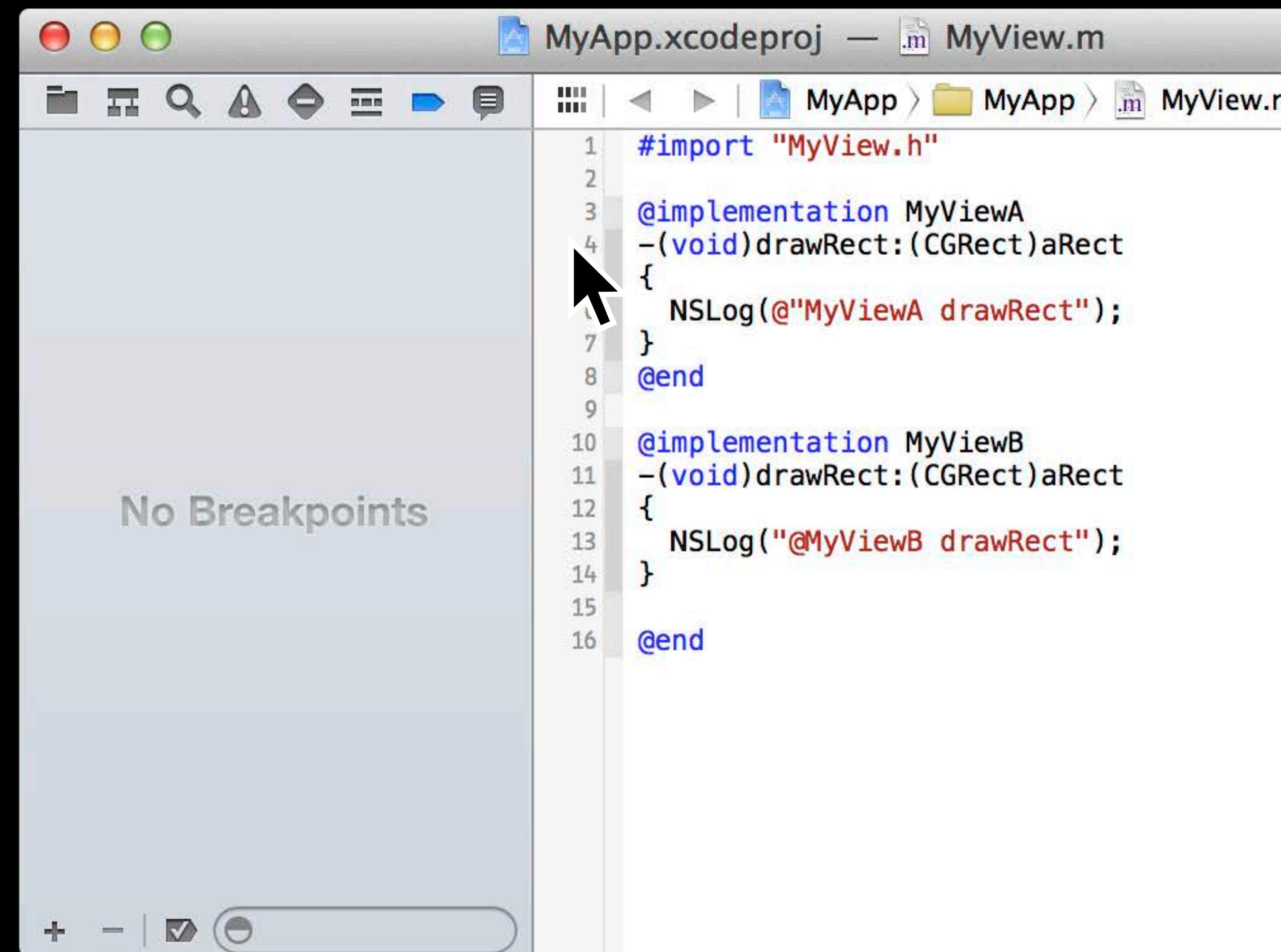
```
#import "MyView.h"  
  
@implementation MyViewA  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewA drawRect");  
}  
@end  
  
@implementation MyViewB  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewB drawRect");  
}  
@end
```

The code editor has a status bar at the bottom with icons for zooming and search.

Common Breakpoint Scenarios

- Stop at a source line:

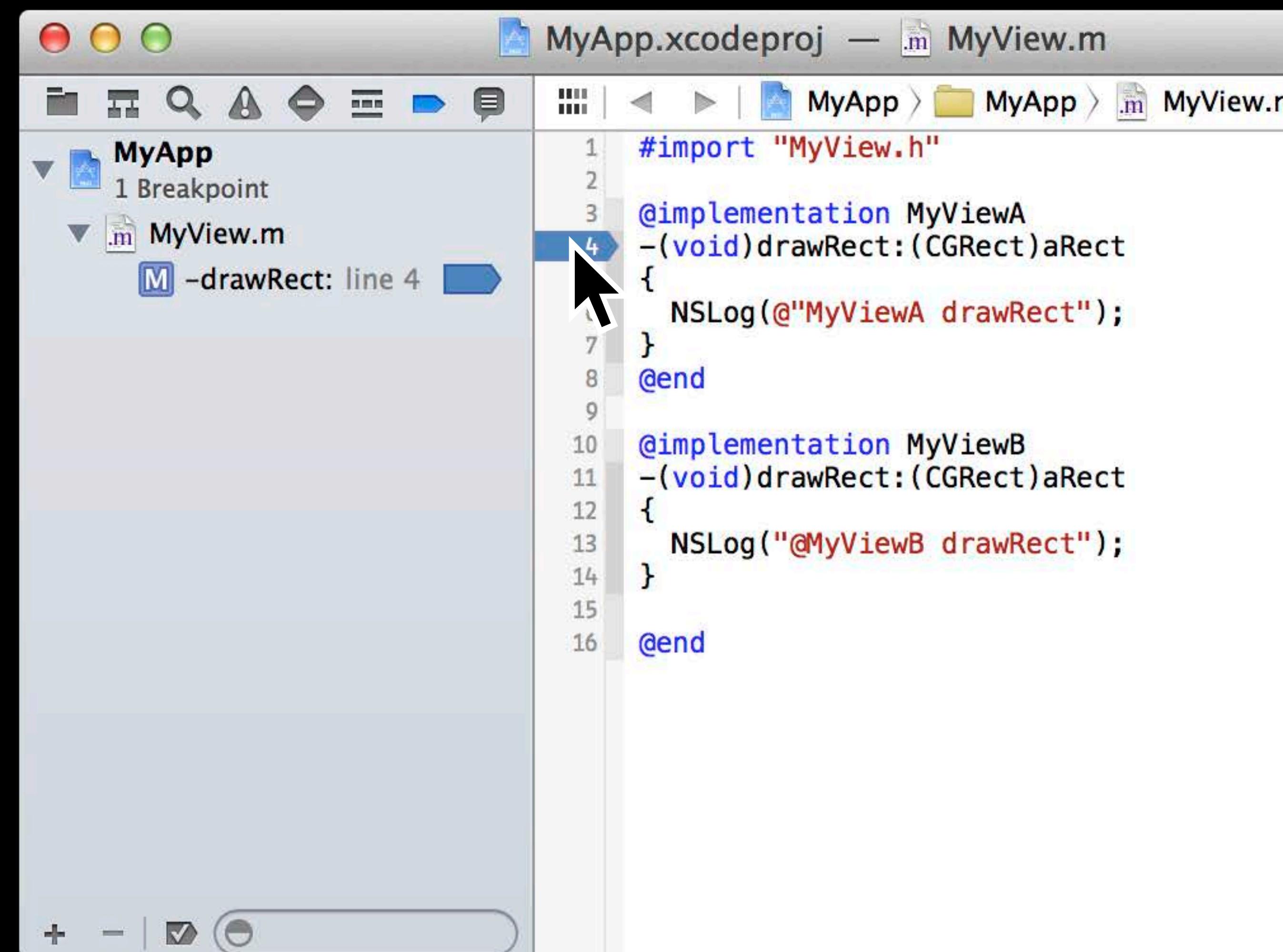
```
b MyView.m:4  
breakpoint set  
--file MyView.m --line 4
```



Common Breakpoint Scenarios

- Stop at a source line:

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



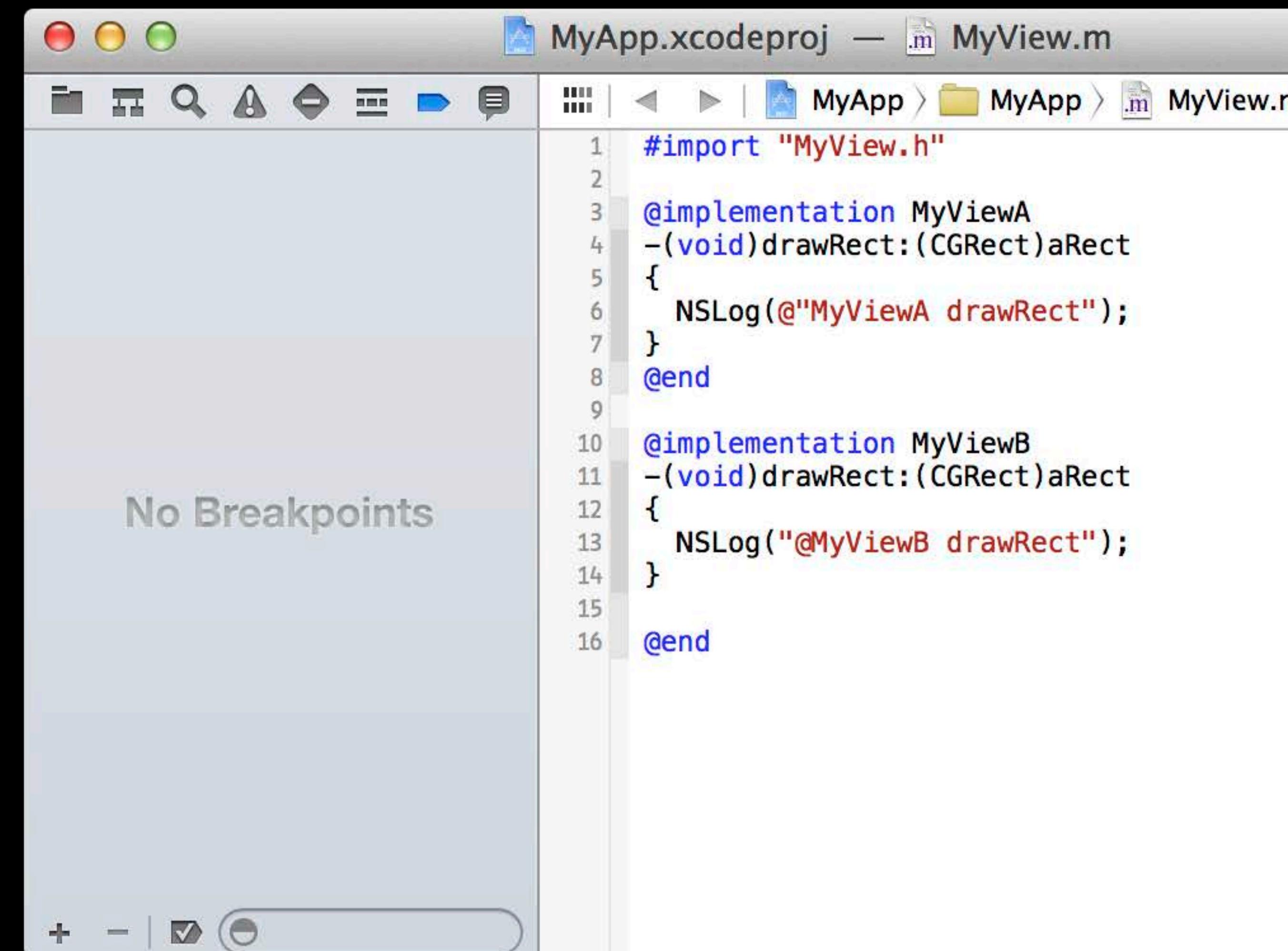
Common Breakpoint Scenarios

- 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:]”
```



```
#import "MyView.h"  
@implementation MyViewA  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewA drawRect");  
}  
@end  
  
@implementation MyViewB  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewB drawRect");  
}  
@end
```

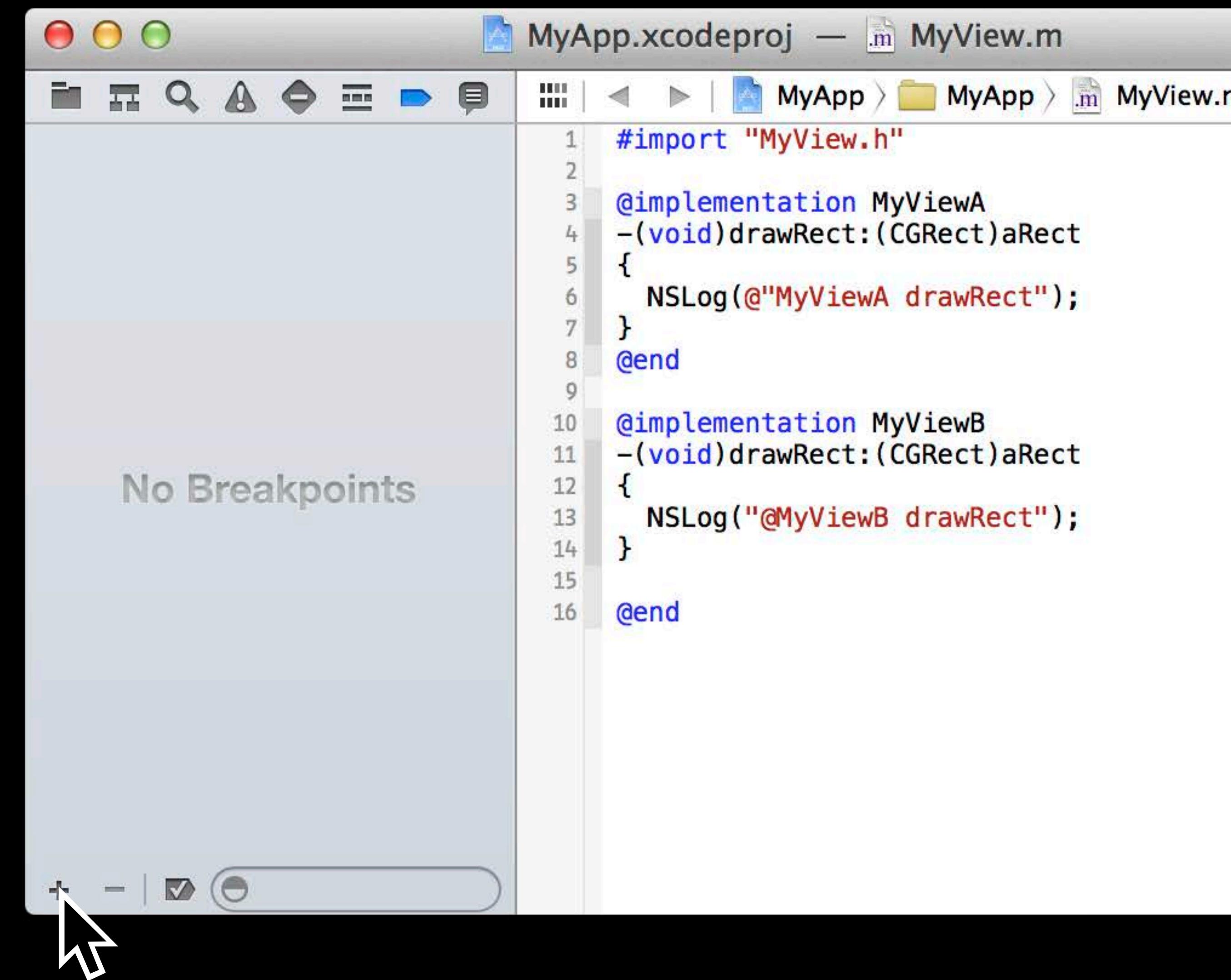
Common Breakpoint Scenarios

- 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:]”
```



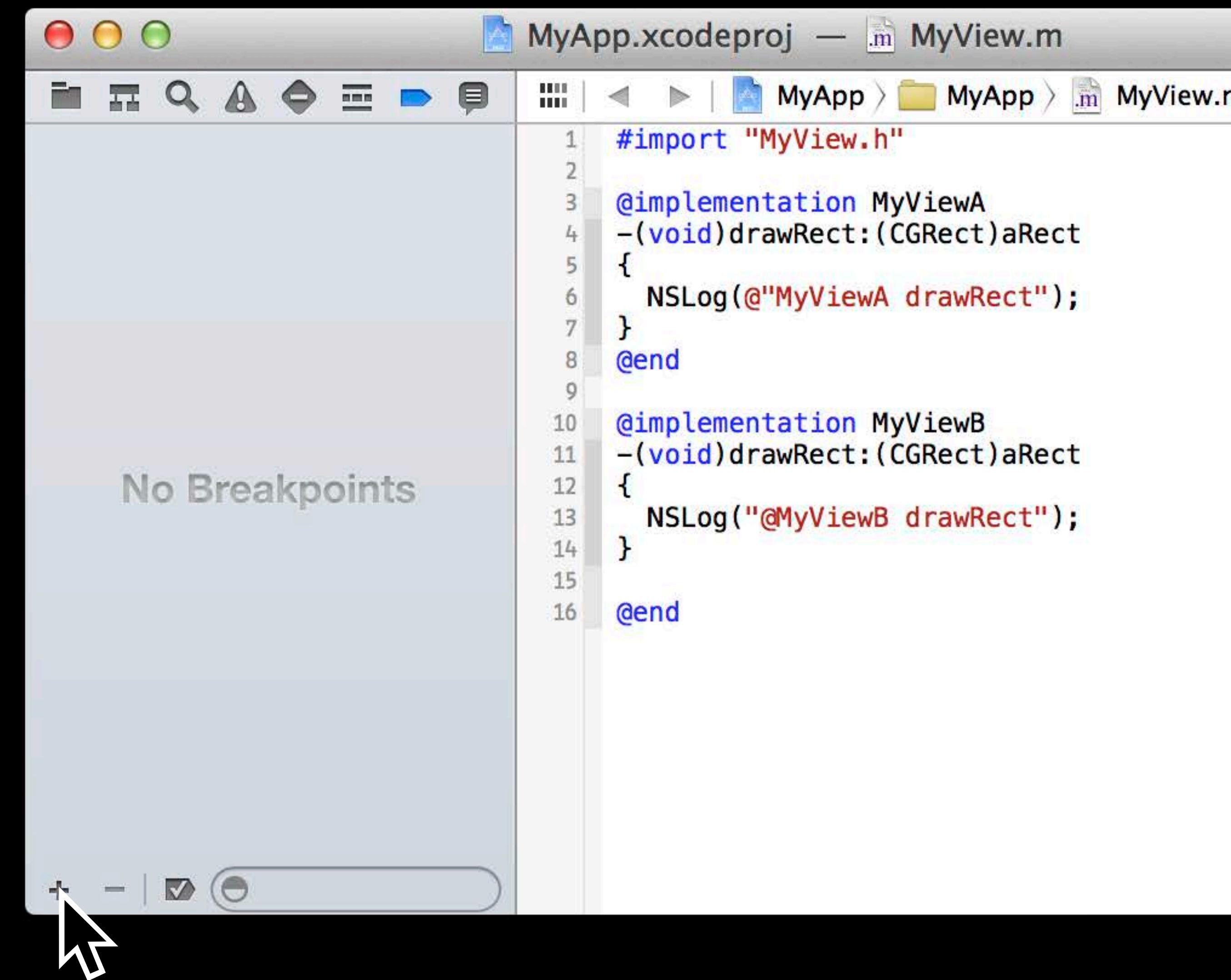
Common Breakpoint Scenarios

- 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:]”
```



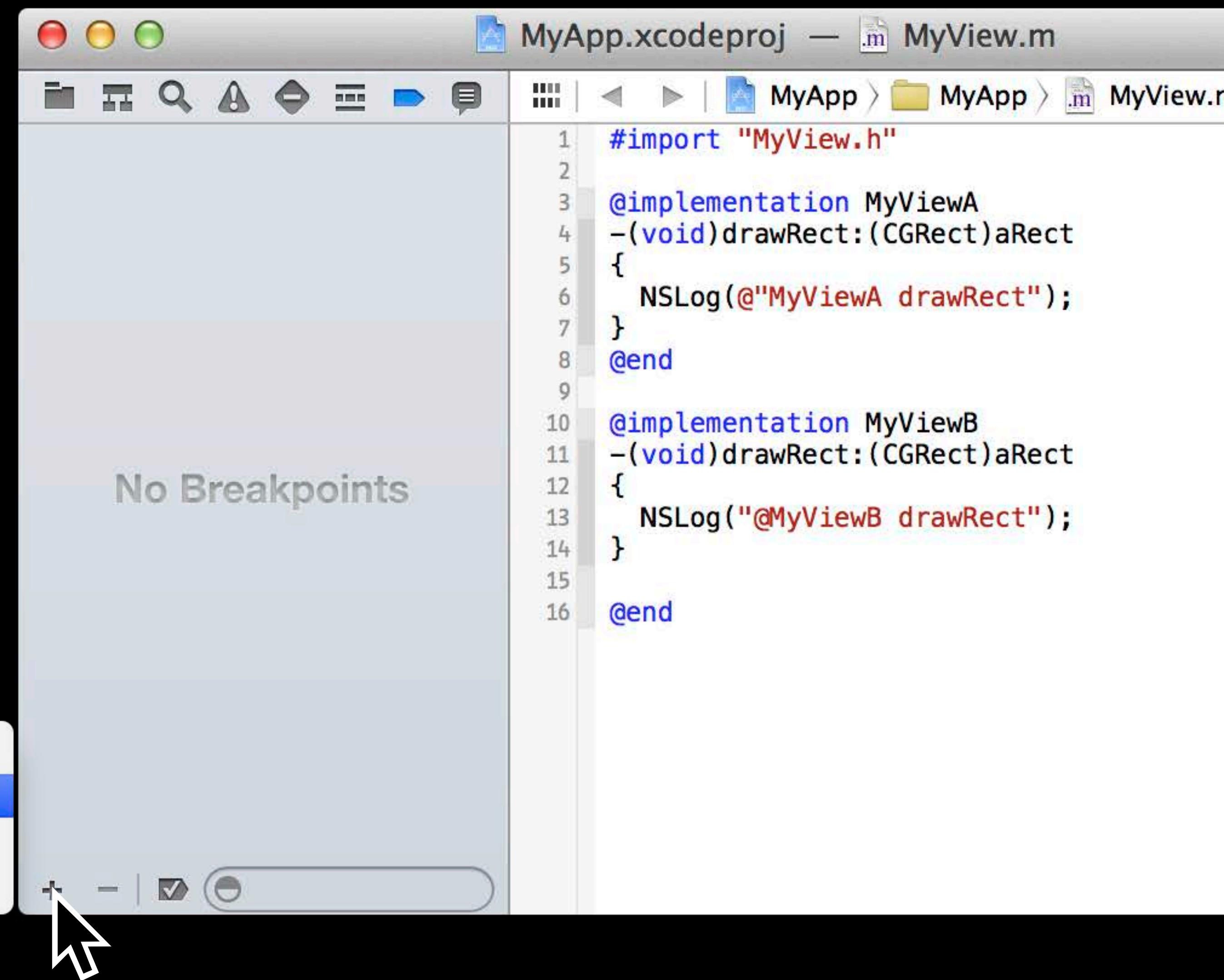
Common Breakpoint Scenarios

- 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:]”
```

The screenshot shows the Xcode interface. The top bar displays "MyApp.xcodeproj" and "MyView.m". The main area shows the code for MyView.m:

```
#import "MyView.h"  
@implementation MyViewA  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewA drawRect");  
}  
@end  
  
@implementation MyViewB  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewB drawRect");  
}  
@end
```

The left side of the interface shows a list of breakpoints with the message "No Breakpoints". At the bottom, there is a floating menu with the following options:

- Add Exception Breakpoint...
- Add Symbolic Breakpoint...** (highlighted)
- Add OpenGL Error Breakpoint...
- Add Test Failure Breakpoint...

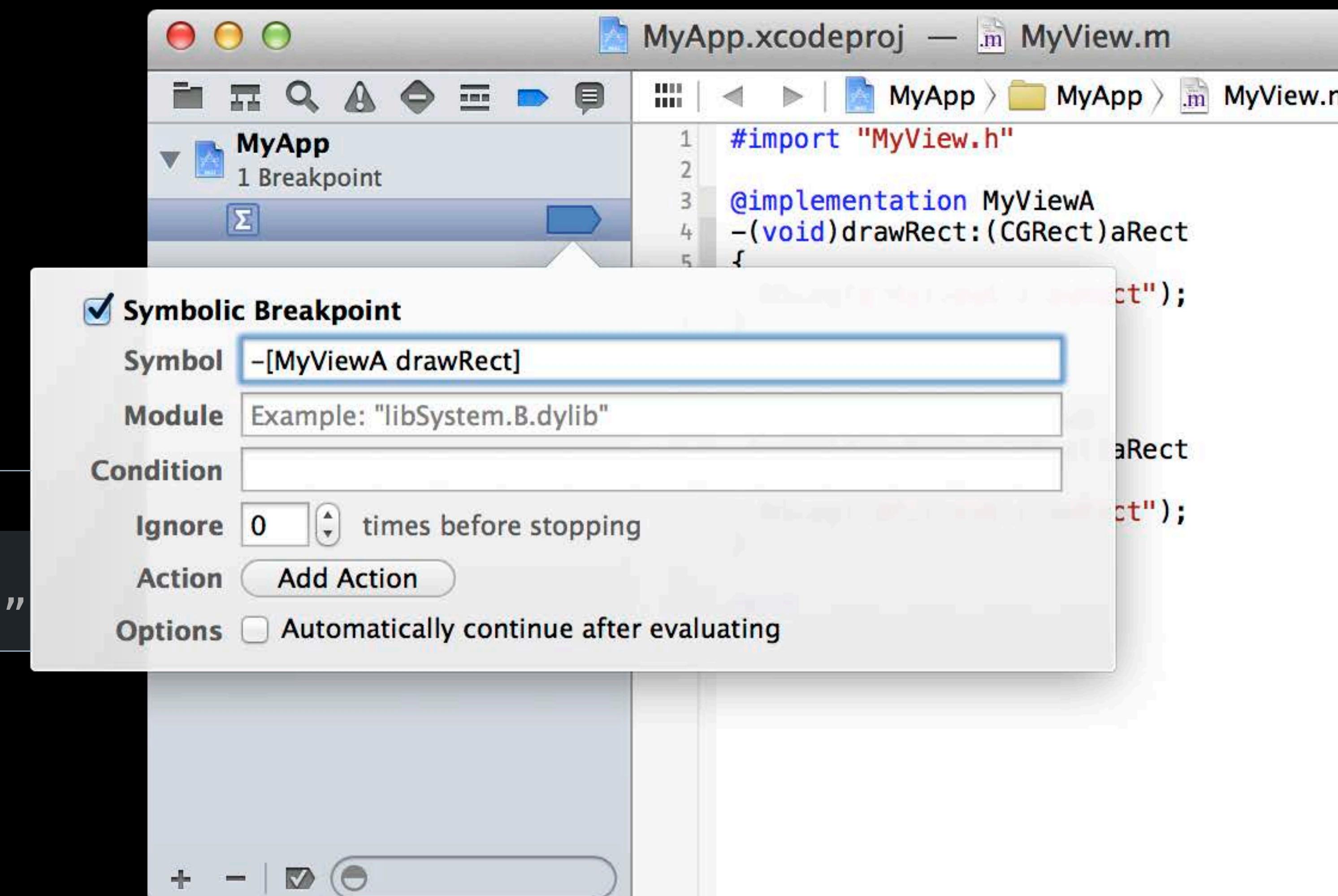
Common Breakpoint Scenarios

- 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:]”
```



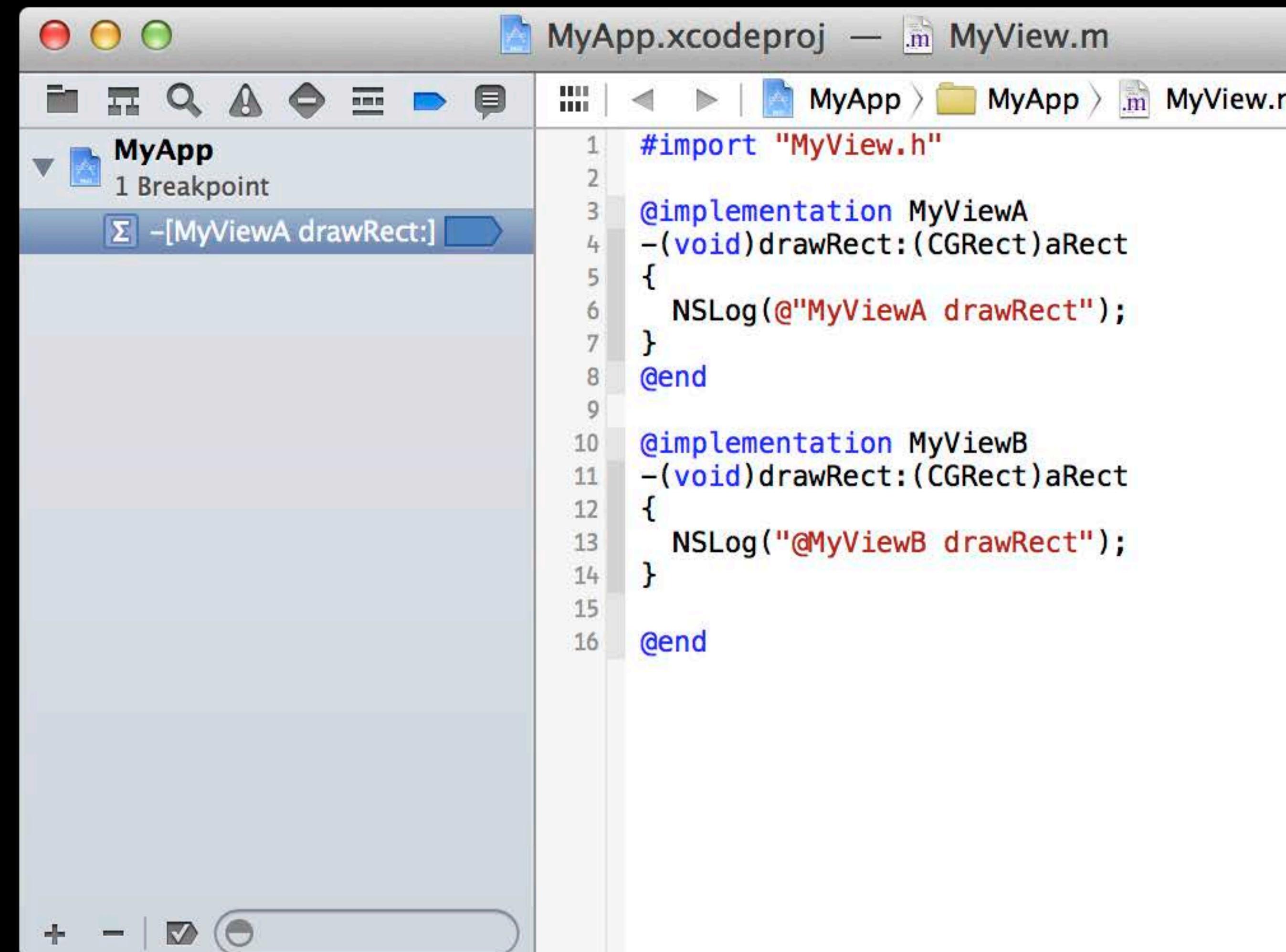
Common Breakpoint Scenarios

- 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:]”
```



The screenshot shows the Xcode interface with the title bar "MyApp.xcodeproj — MyView.m". The left sidebar shows a project named "MyApp" with one breakpoint listed. The main editor area displays the code for "MyView.m". The code contains two methods: `-[MyViewA drawRect:]` and `-[MyViewB drawRect:]`. Both methods log their respective names to the console.

```
#import "MyView.h"  
@implementation MyViewA  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewA drawRect");  
}  
@end  
  
@implementation MyViewB  
-(void)drawRect:(CGRect)aRect  
{  
    NSLog(@"MyViewB drawRect");  
}  
@end
```

Common Breakpoint Scenarios

- Stop at a source line:

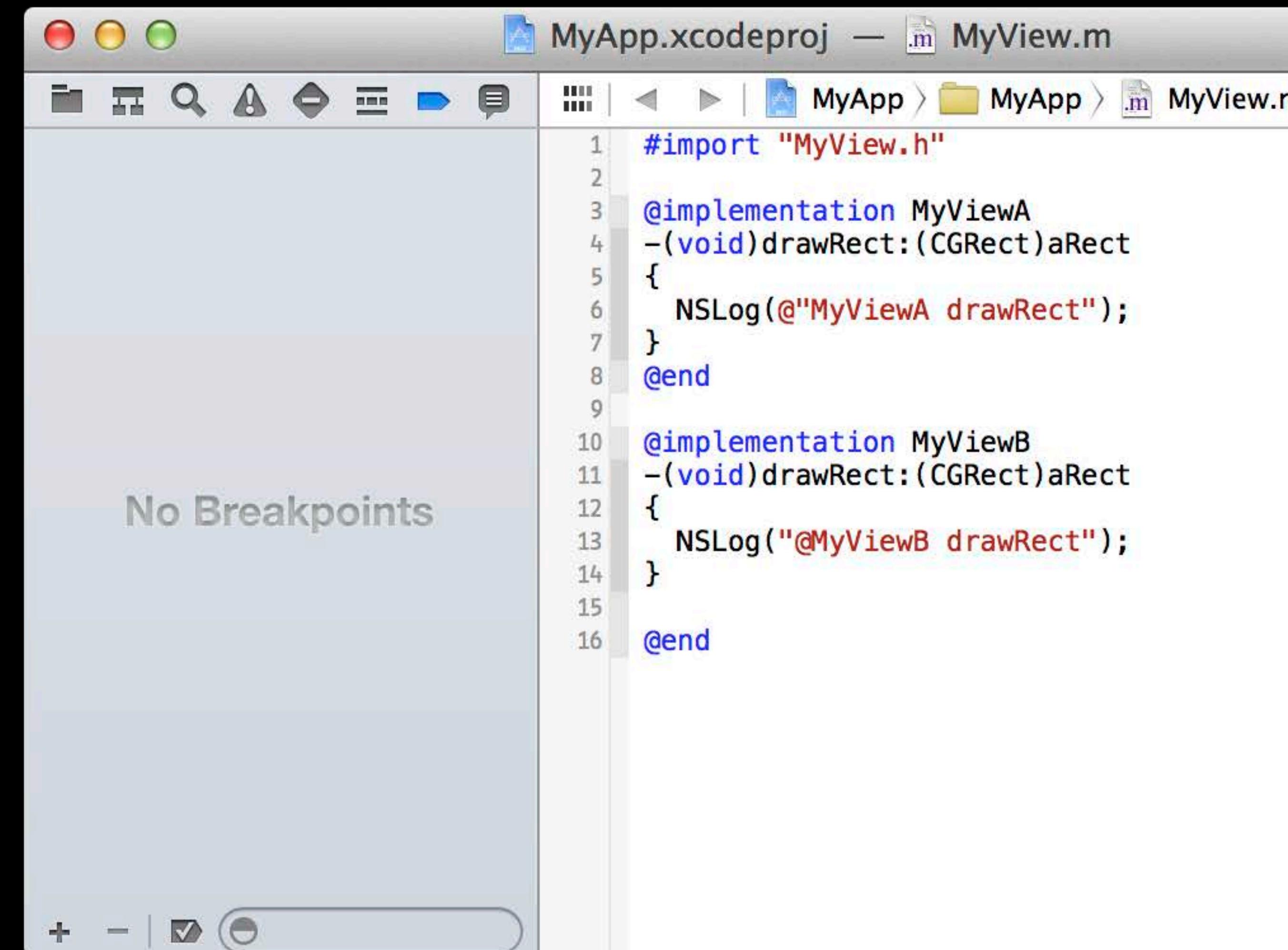
```
b MyView.m:4  
breakpoint set  
--file MyView.m --line 4
```

- 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
```

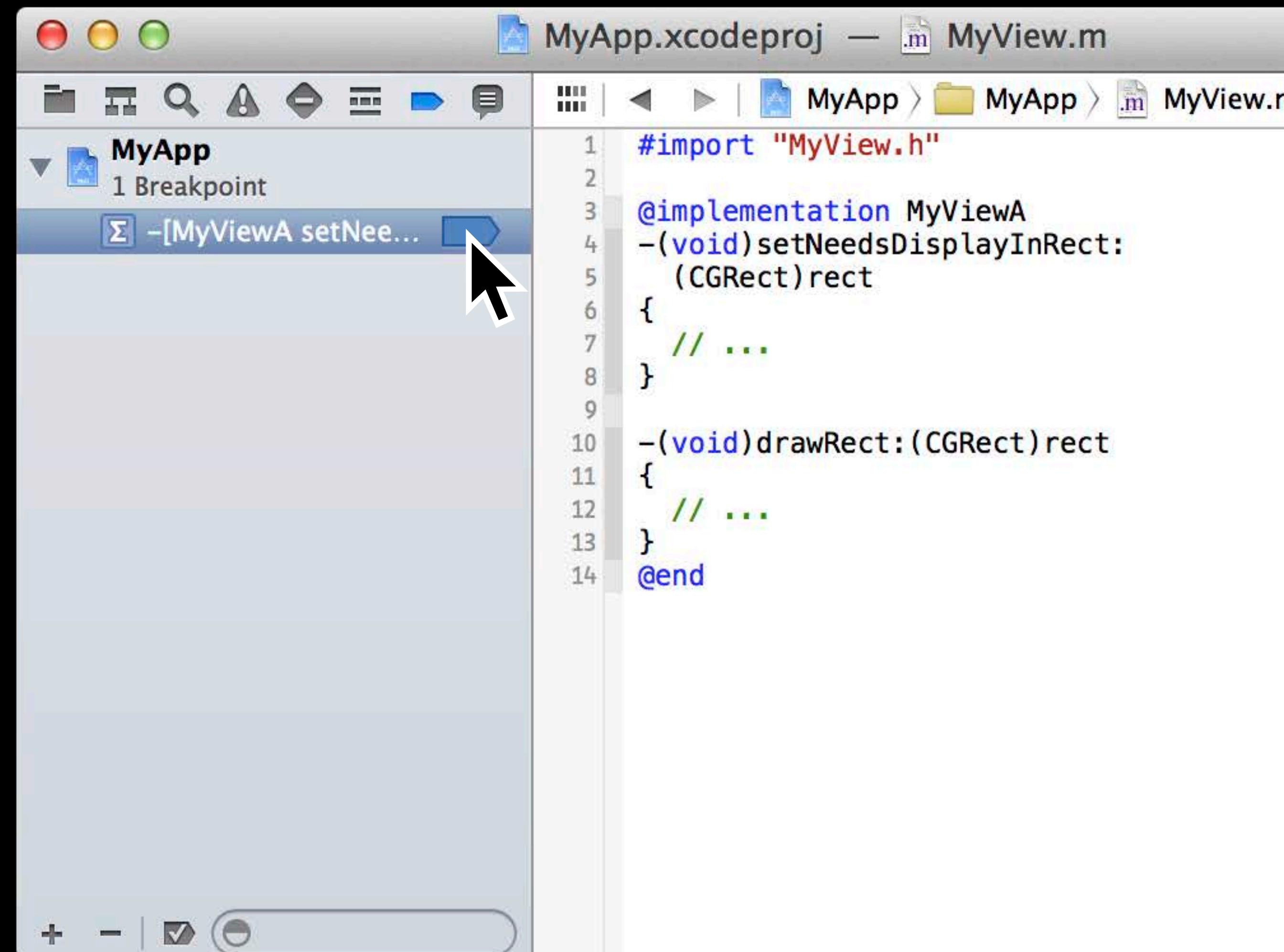
The screenshot shows the Xcode interface with a project named "MyApp.xcodeproj" and a file "MyView.m" open. A breakpoint is set at line 3, specifically on the line containing `-[MyViewA setNee...`. The terminal window below shows the history of commands entered to set this breakpoint:

```
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
```



The screenshot shows the Xcode interface with the title bar "MyApp.xcodeproj — MyView.m". The left sidebar shows the project structure with "MyApp" selected, containing one breakpoint. The main editor area displays the code for "MyView.m":

```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
    (CGRect)rect
{
    // ...
}

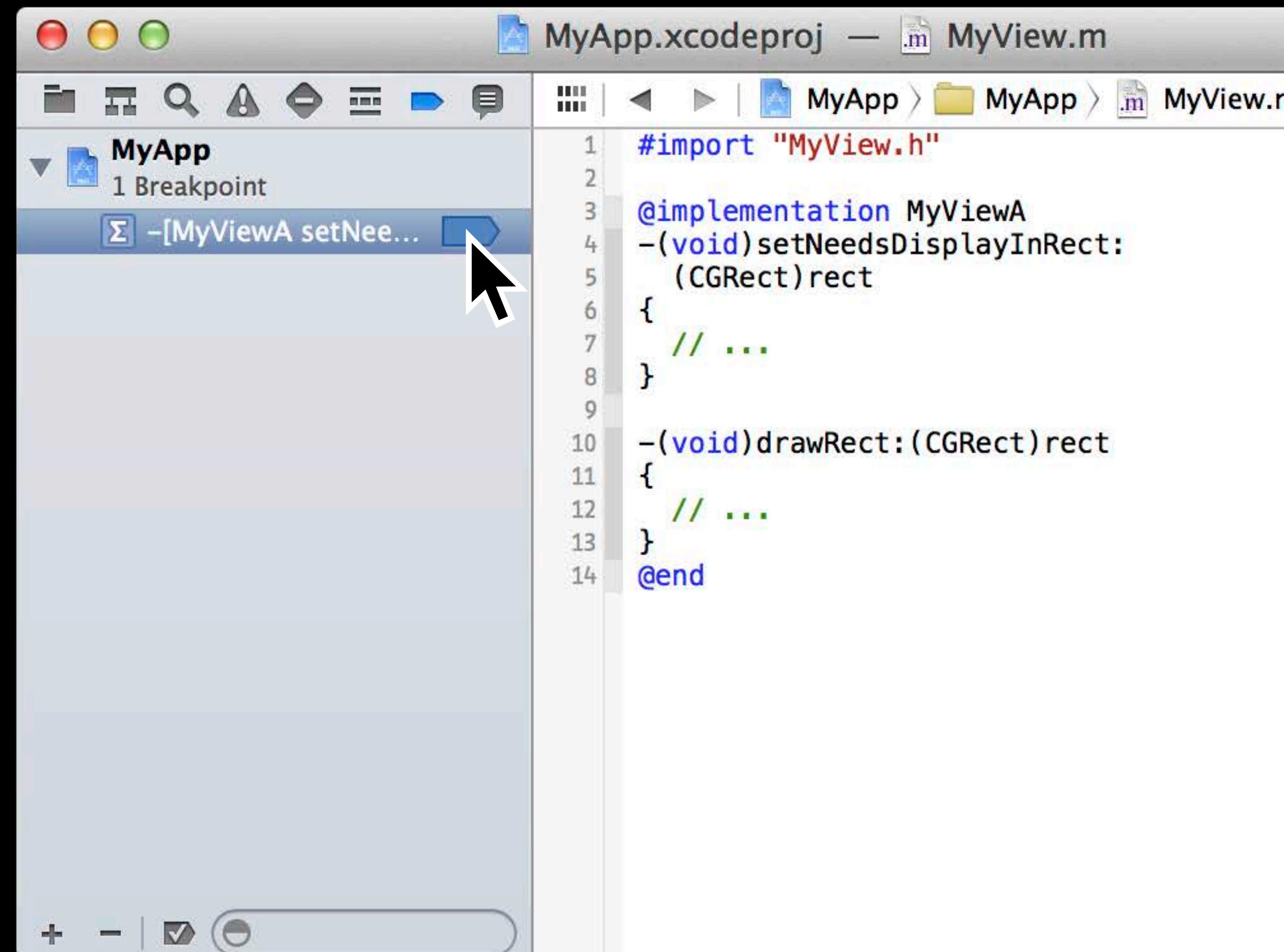
-(void)drawRect:(CGRect)rect
{
    // ...
}
@end
```

A mouse cursor is hovering over the breakpoint in the sidebar.

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
```



The screenshot shows the Xcode interface with the title bar "MyApp.xcodeproj — MyView.m". The left sidebar shows the project structure with "MyApp" selected, containing one breakpoint. The main editor area displays the code for "MyView.m":

```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
    (CGRect)rect
{
    // ...
}

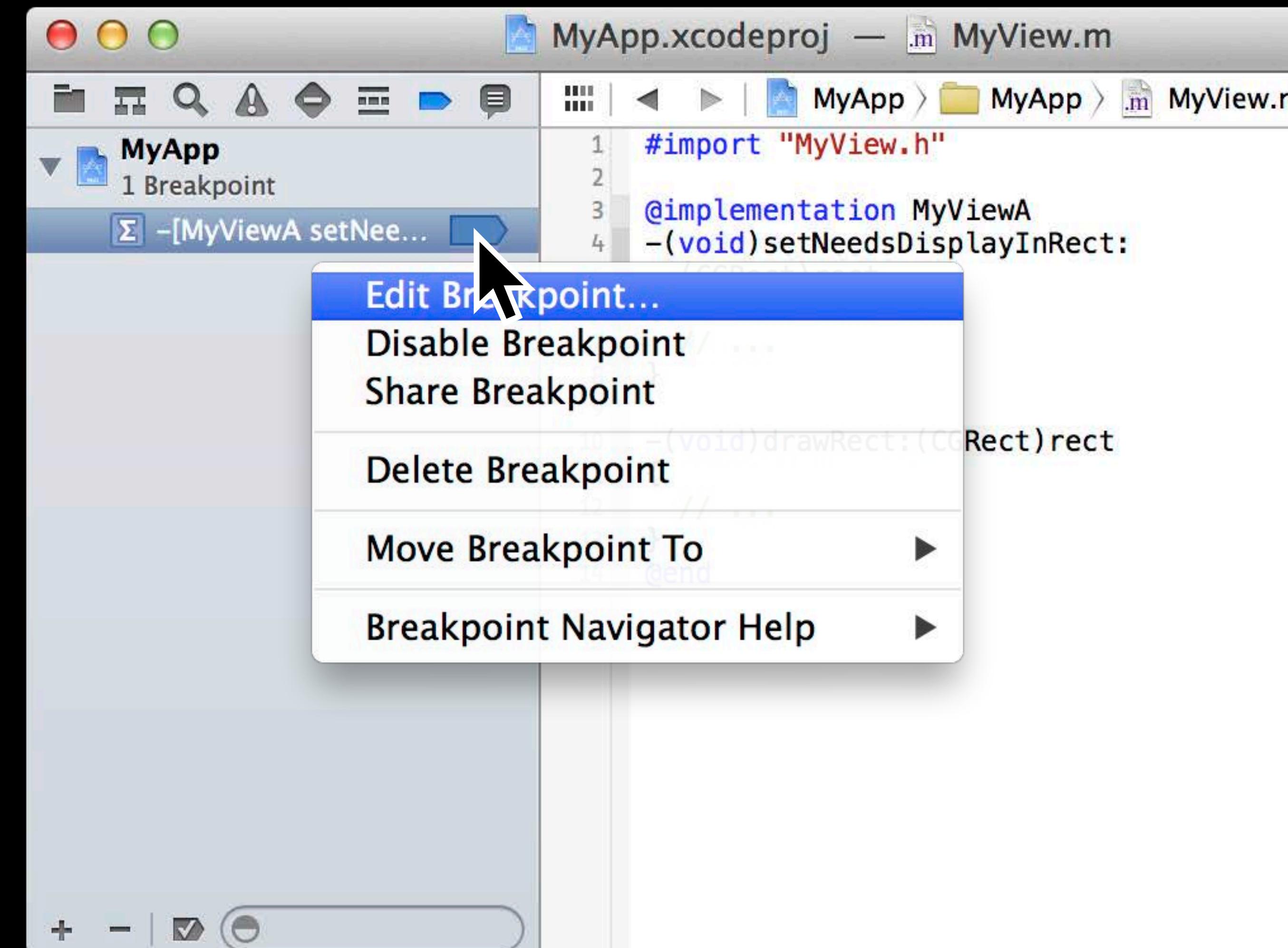
-(void)drawRect:(CGRect)rect
{
    // ...
}
@end
```

A mouse cursor is hovering over the breakpoint in the sidebar.

Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit
 - b “-[MyViewA
setNeedsDisplayInRect:]”

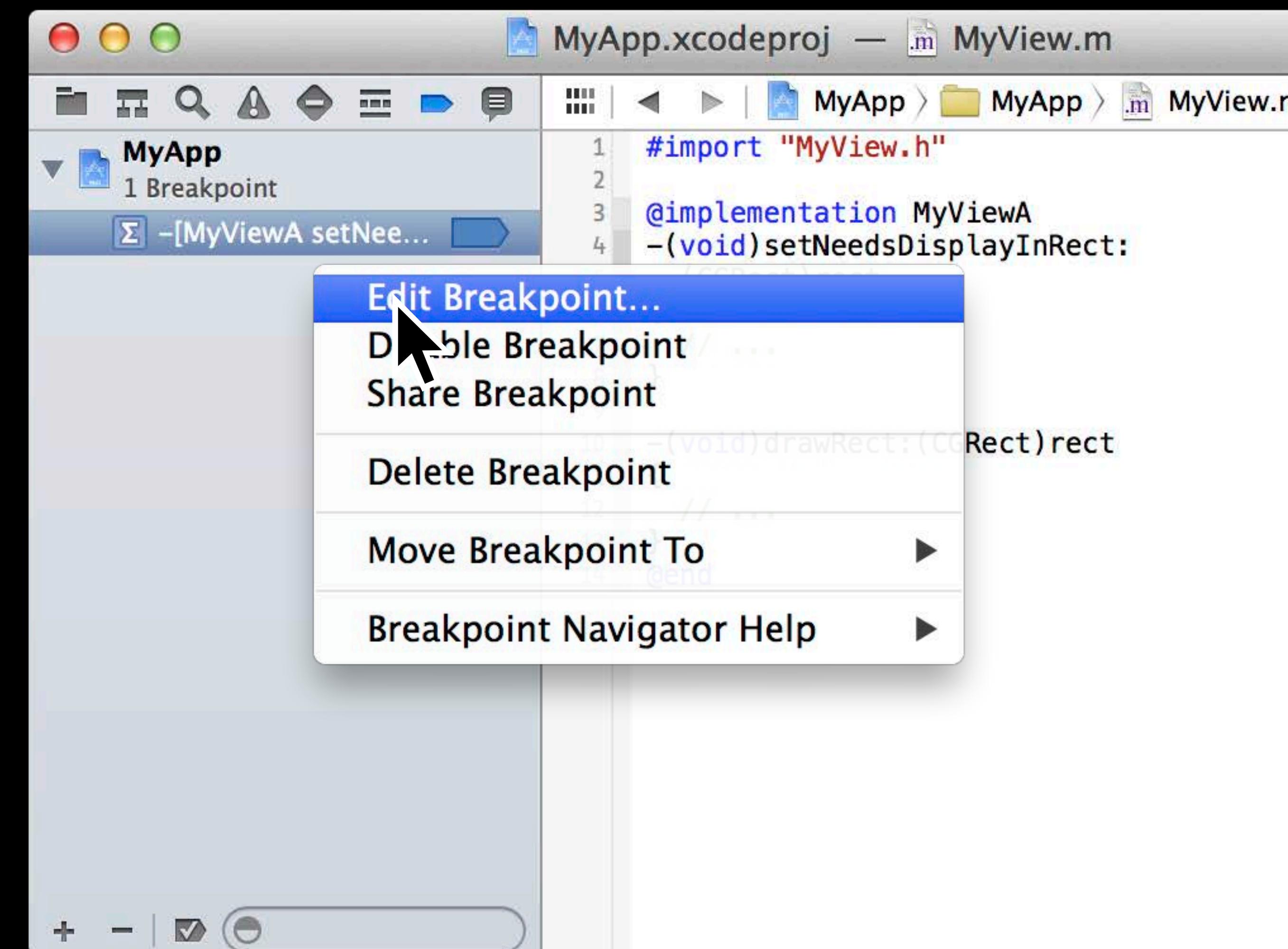
```
br co a    breakpoint command add
> p rect expression rect
> bt      thread backtrace
> c      process continue
> DONE
```



Commands Save Time

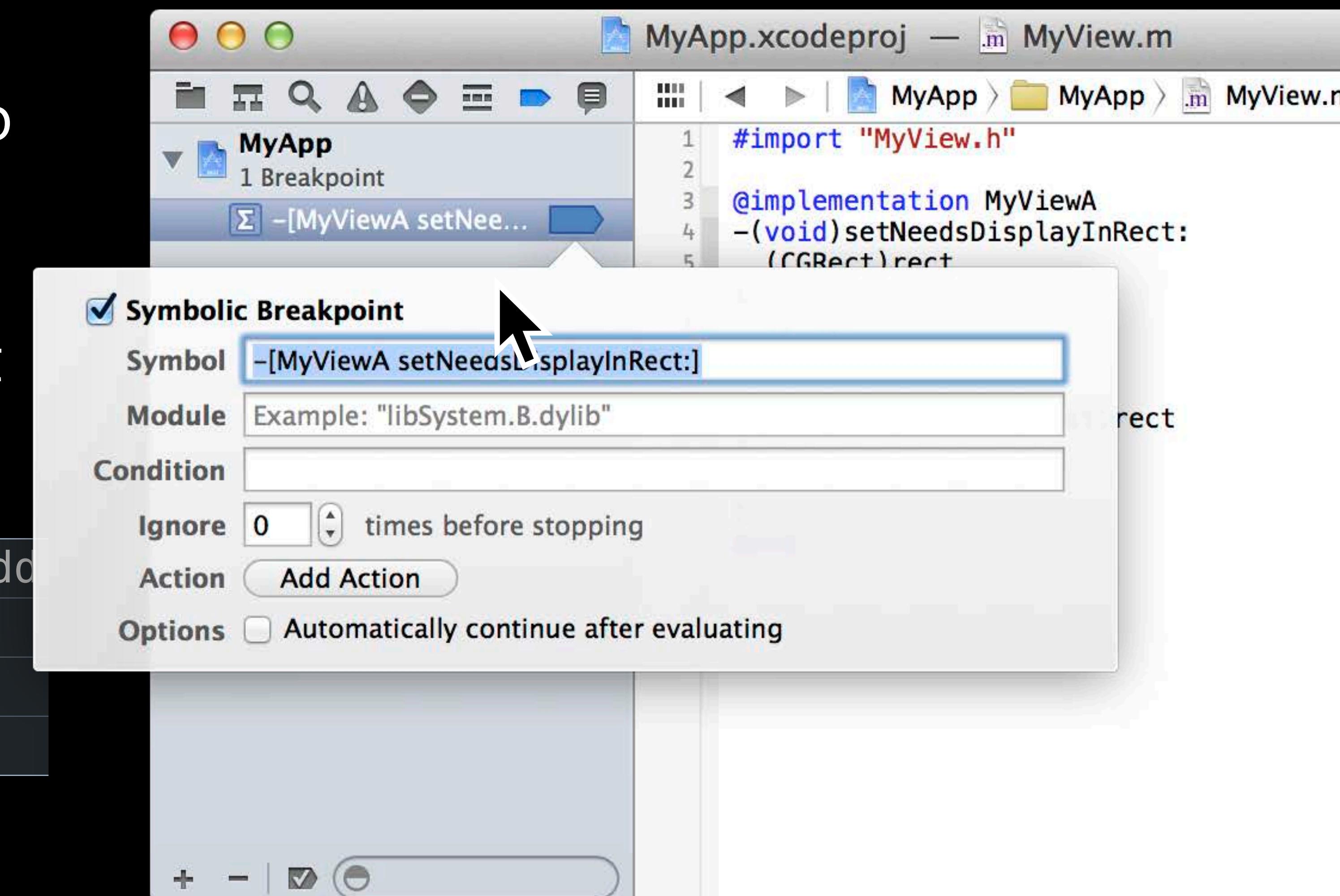
- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit
 - b “-[MyViewA setNeedsDisplayInRect:]”

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



Commands Save Time

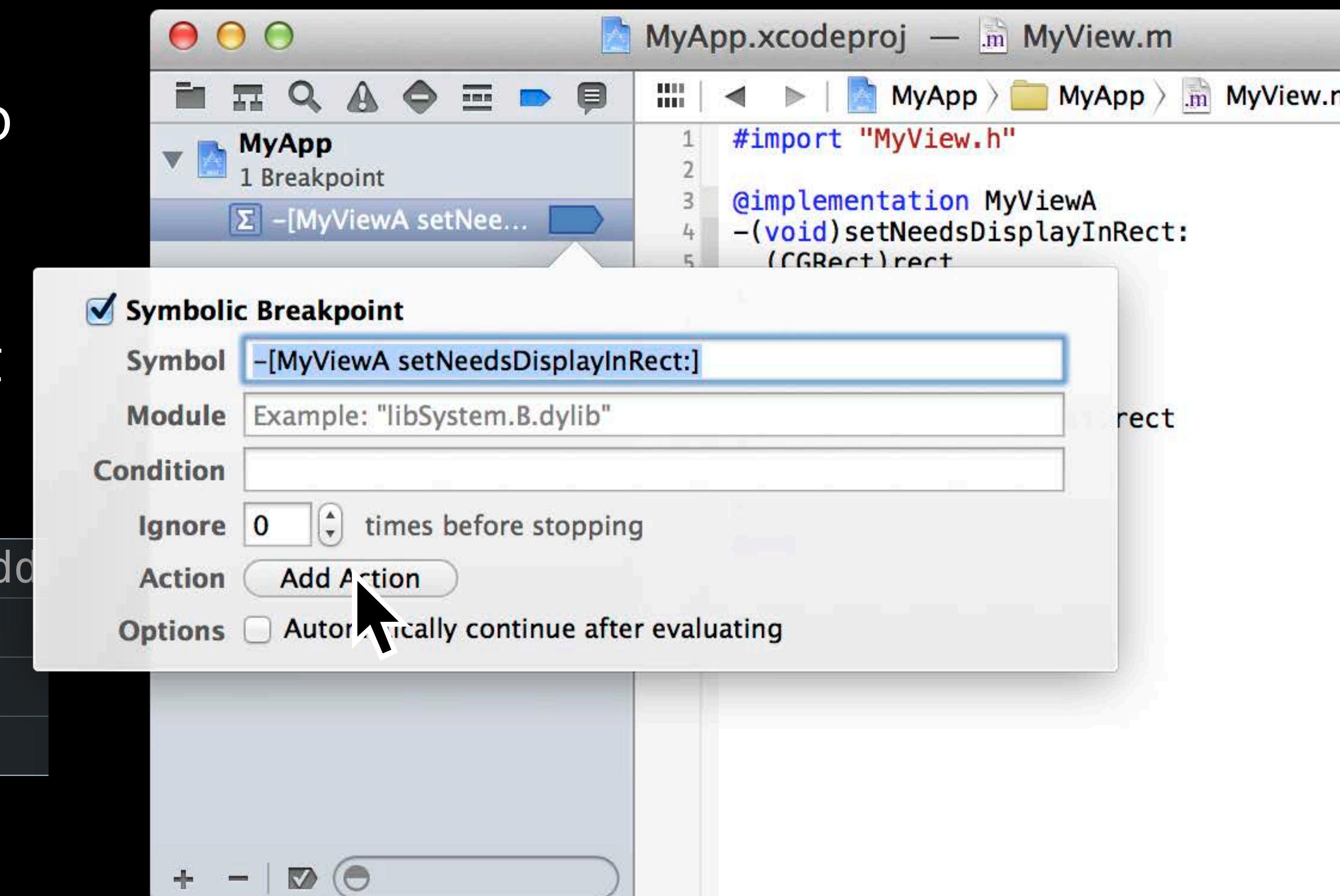
- Switching between your app and Xcode is tedious
 - Breakpoint commands run each time a breakpoint is hit
 - b “-[MyViewA setNeedsDisplayInRect:]”
- ```
br co a breakpoint command add
> p rect expression rect
> bt thread backtrace
> c process continue
> DONE
```



# Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit
  - b “-[MyViewA setNeedsDisplayInRect:]”

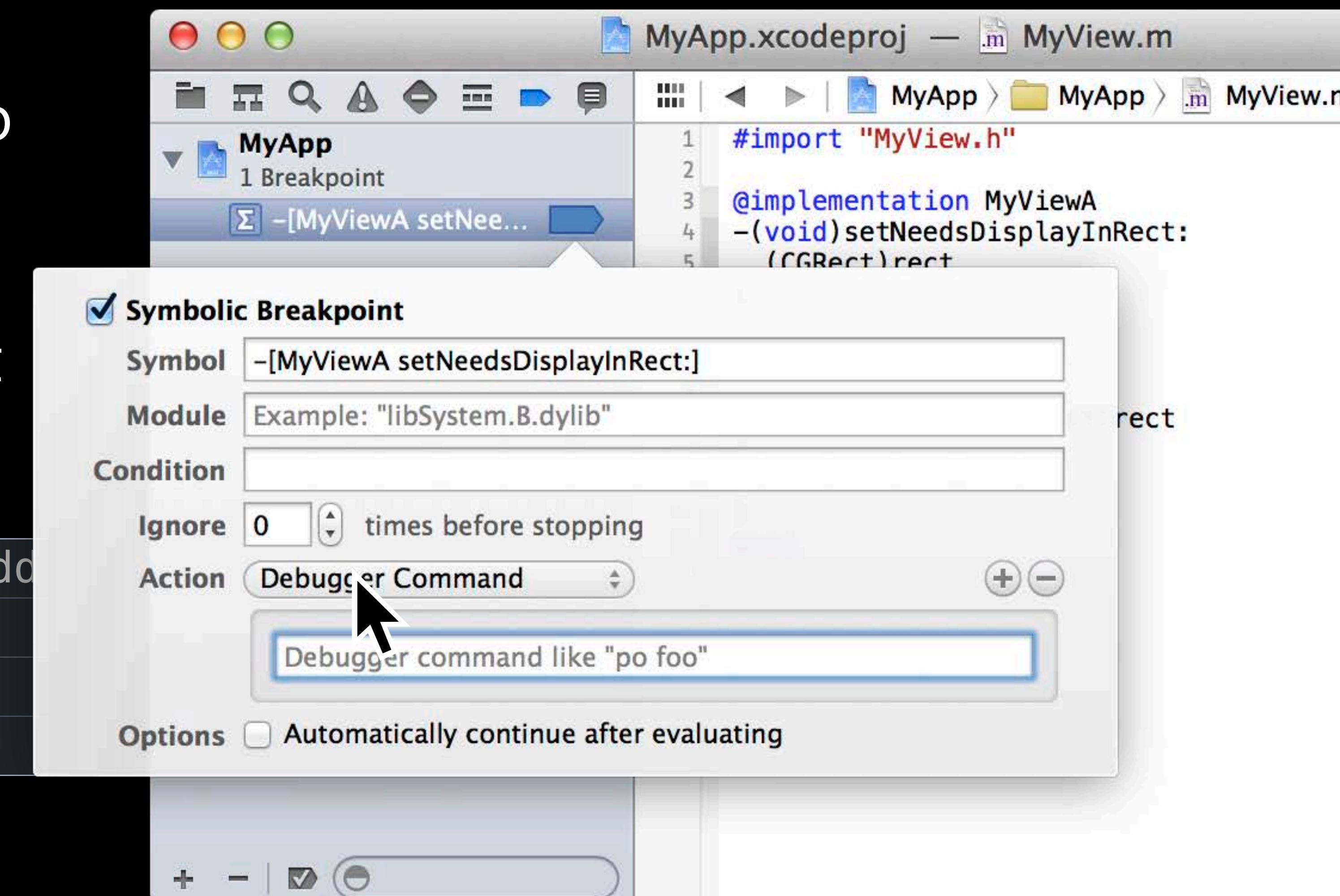
```
br co a breakpoint command add
> p rect expression rect
> bt thread backtrace
> c process continue
> DONE
```



# 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
```

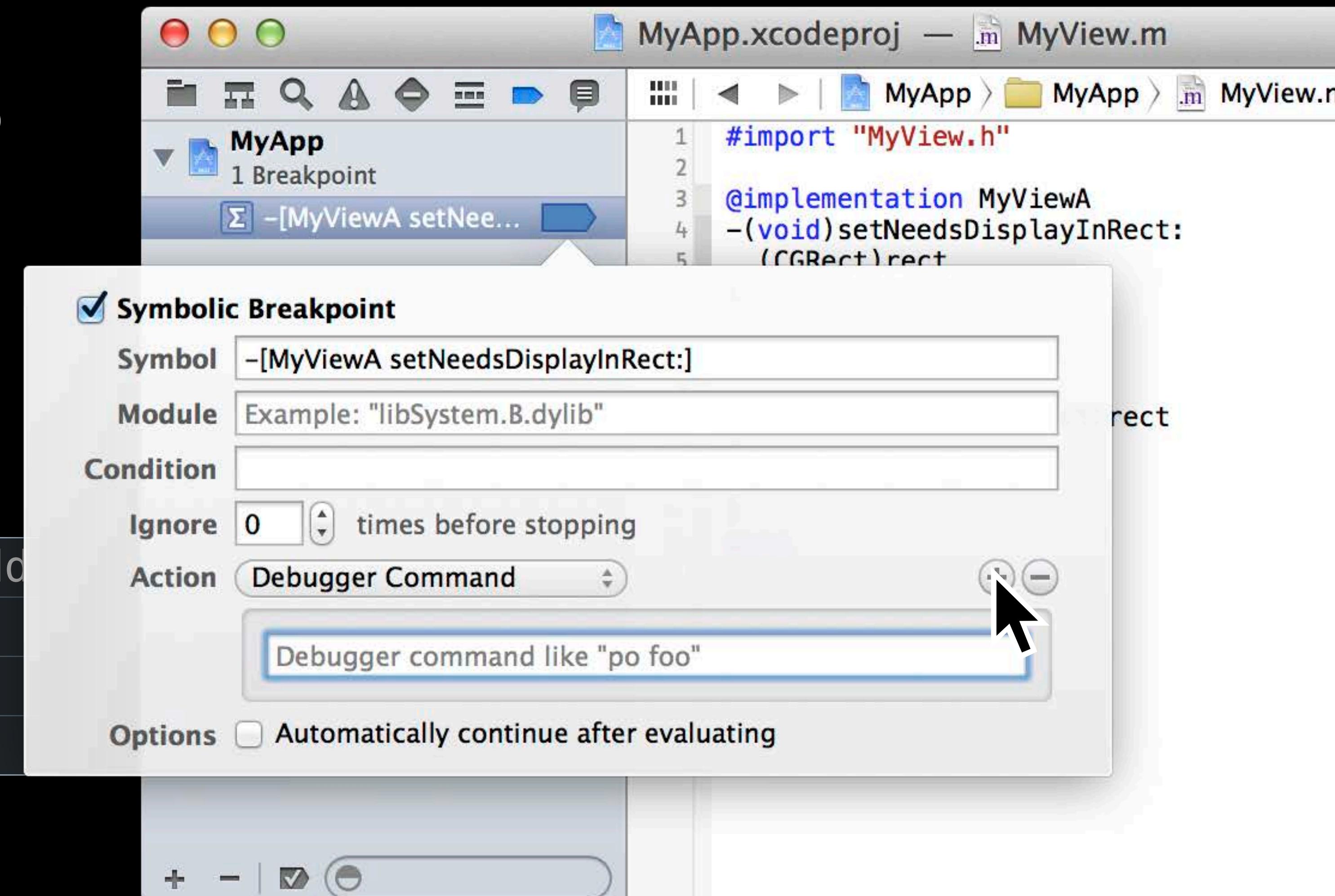


# Commands Save Time

- Switching between your app and Xcode is tedious
  - Breakpoint commands run each time a breakpoint is hit
    - b “-[MyViewA setNeedsDisplayInRect:]”

---

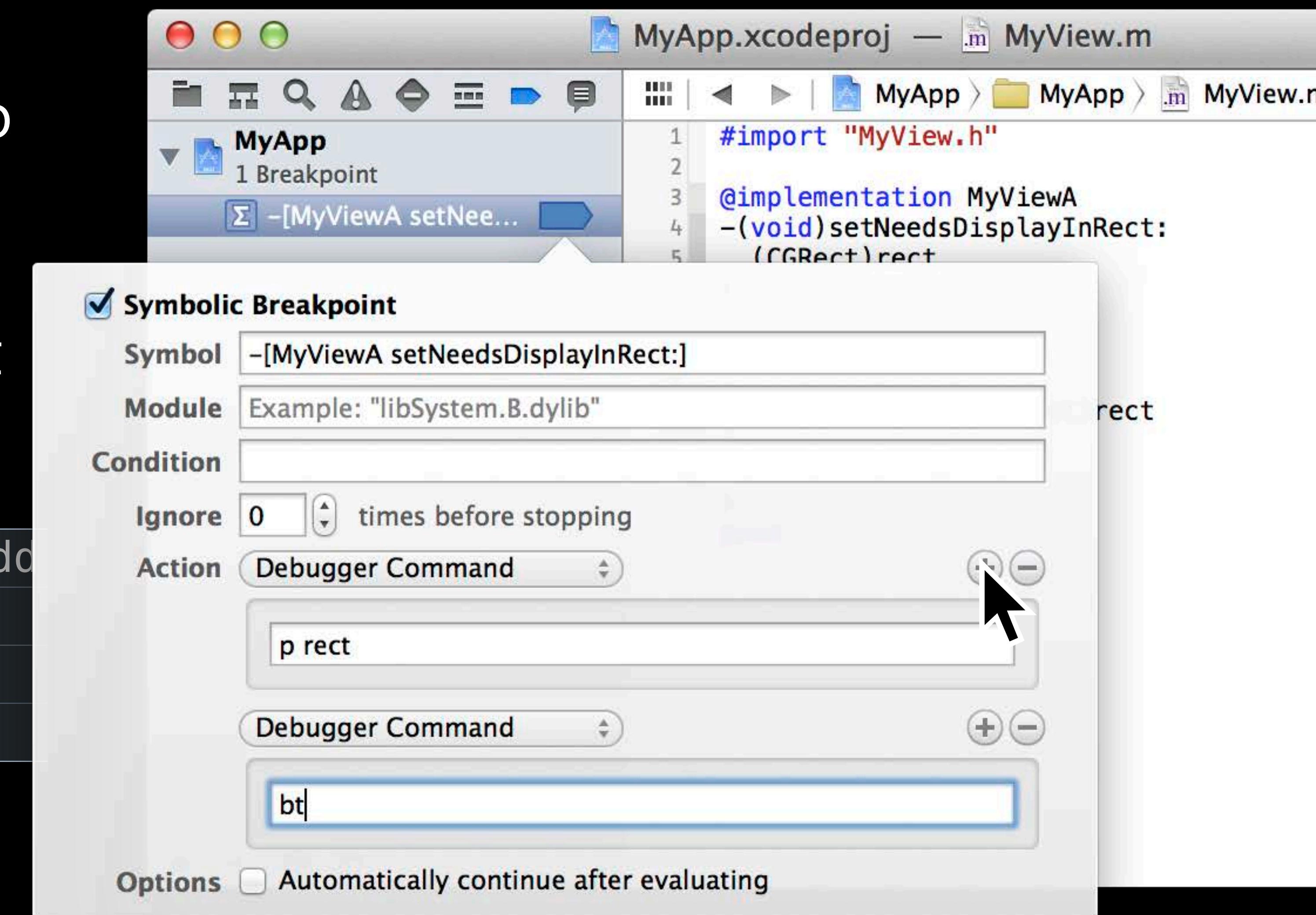
|        |      |   |            |           |    |
|--------|------|---|------------|-----------|----|
| br     | co   | a | breakpoint | command   | ad |
| > p    | rect |   | expression | rect      |    |
| > bt   |      |   | thread     | backtrace |    |
| > c    |      |   | process    | continue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



# Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit
  - b “-[MyViewA setNeedsDisplayInRect:]”

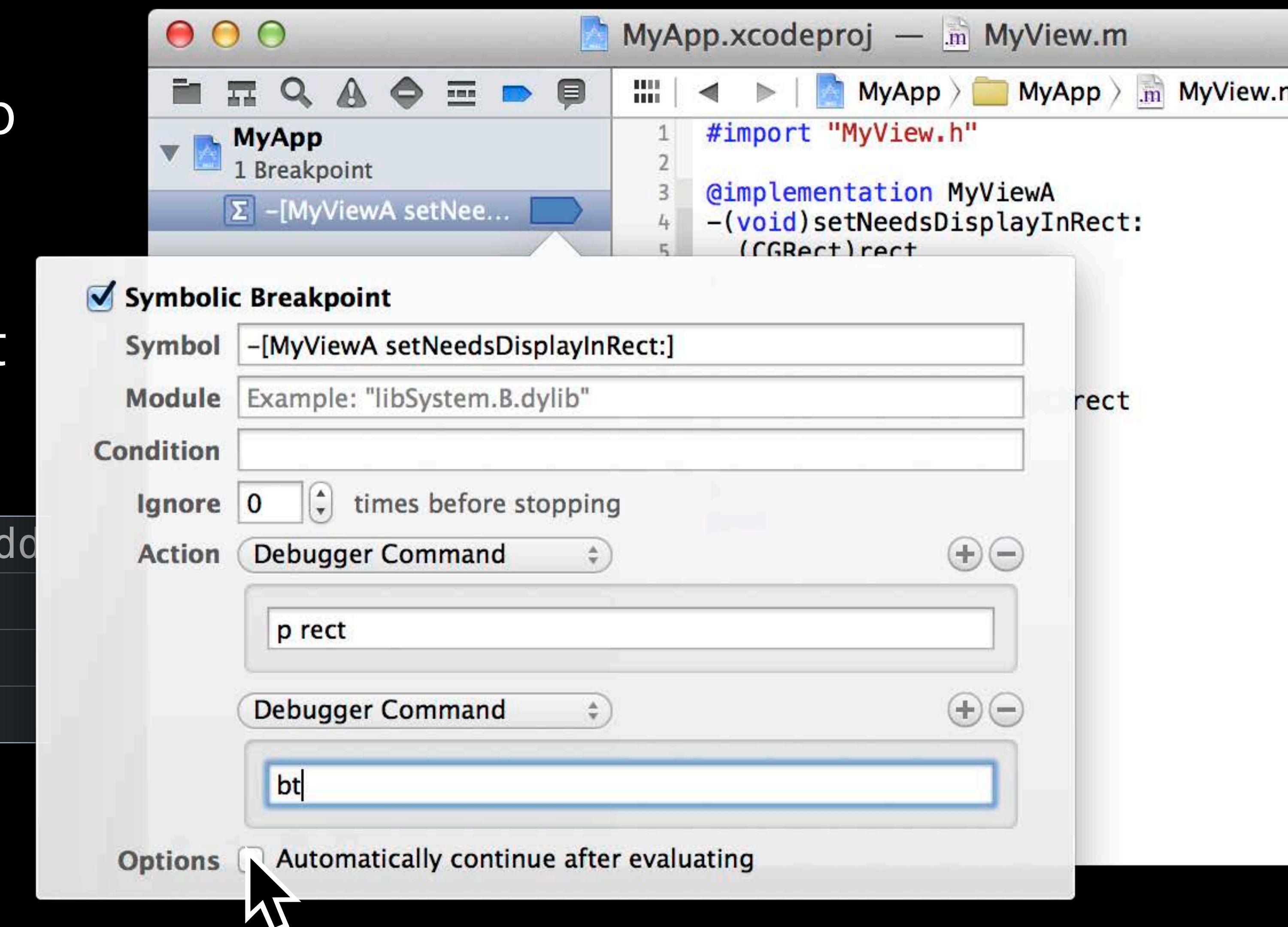
```
br co a breakpoint command add
> p rect expression rect
> bt thread backtrace
> c process continue
> DONE
```



# Commands Save Time

- Switching between your app and Xcode is tedious
- Breakpoint commands run each time a breakpoint is hit
  - b “-[MyViewA setNeedsDisplayInRect:]”

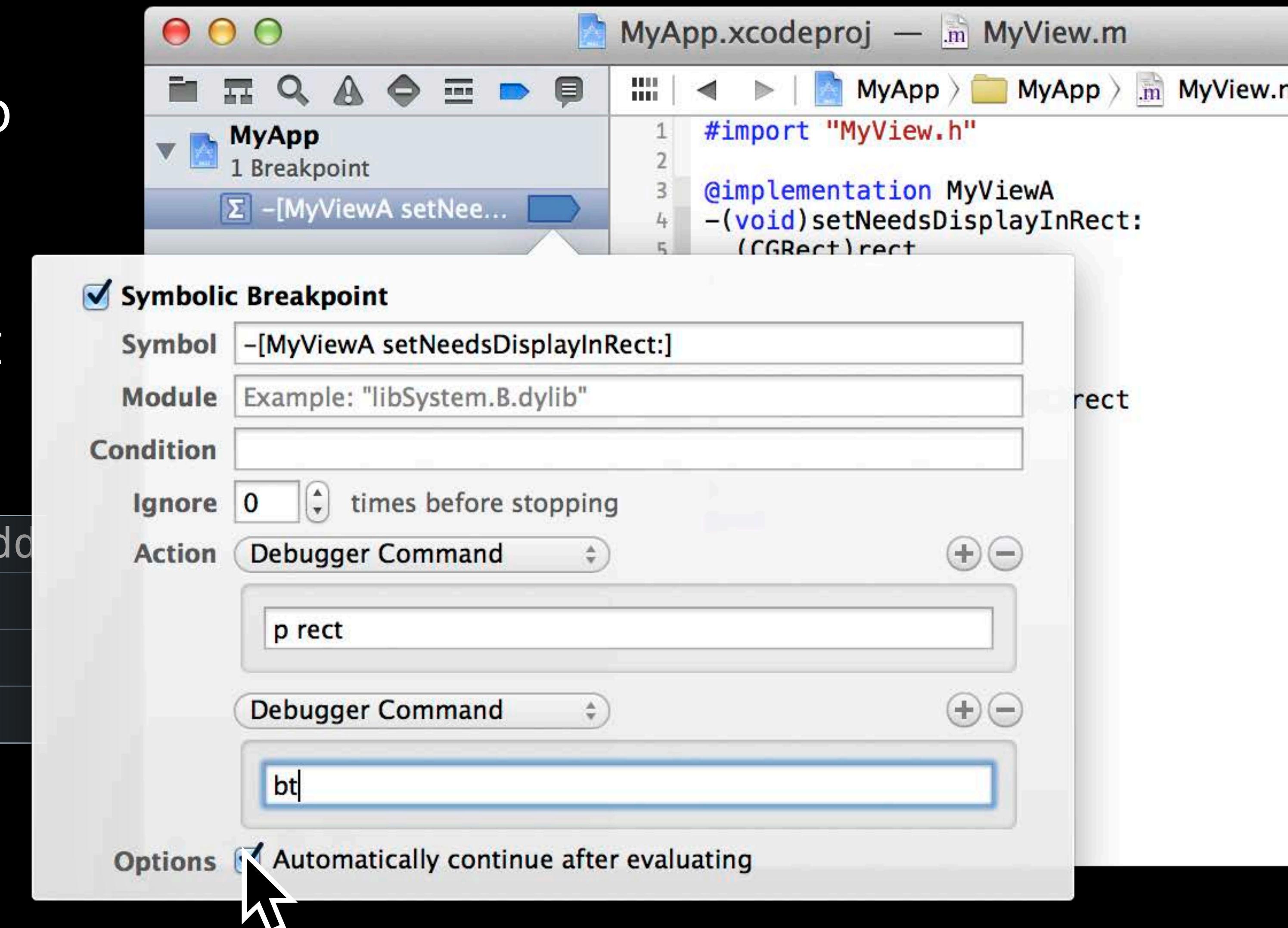
```
br co a breakpoint command add
> p rect expression rect
> bt thread backtrace
> c process continue
> DONE
```



# 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”
```

The screenshot shows the Xcode interface during a debug session. The title bar indicates "MyApp.xcodeproj — MyModel.m". The left sidebar shows the project structure with "MyApp" selected. The main area displays the code for "MyModel.m". A breakpoint is set at line 7, which contains the code "return self;". The condition for this breakpoint is "(lldb) p id \$myModel = self". The "Breakpoint Conditions" section of the Xcode interface shows the condition "(lldb) p id \$myModel = self". The bottom status bar shows "All Output".

```
#import "MyModel.h"

@implementation MyModel
-(id)init
{
 self = [super init];
 return self;
}
-(void)dealloc
{
}
@end
```

```
(lldb) p id $myModel = self
(lldb)
```

# 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
```

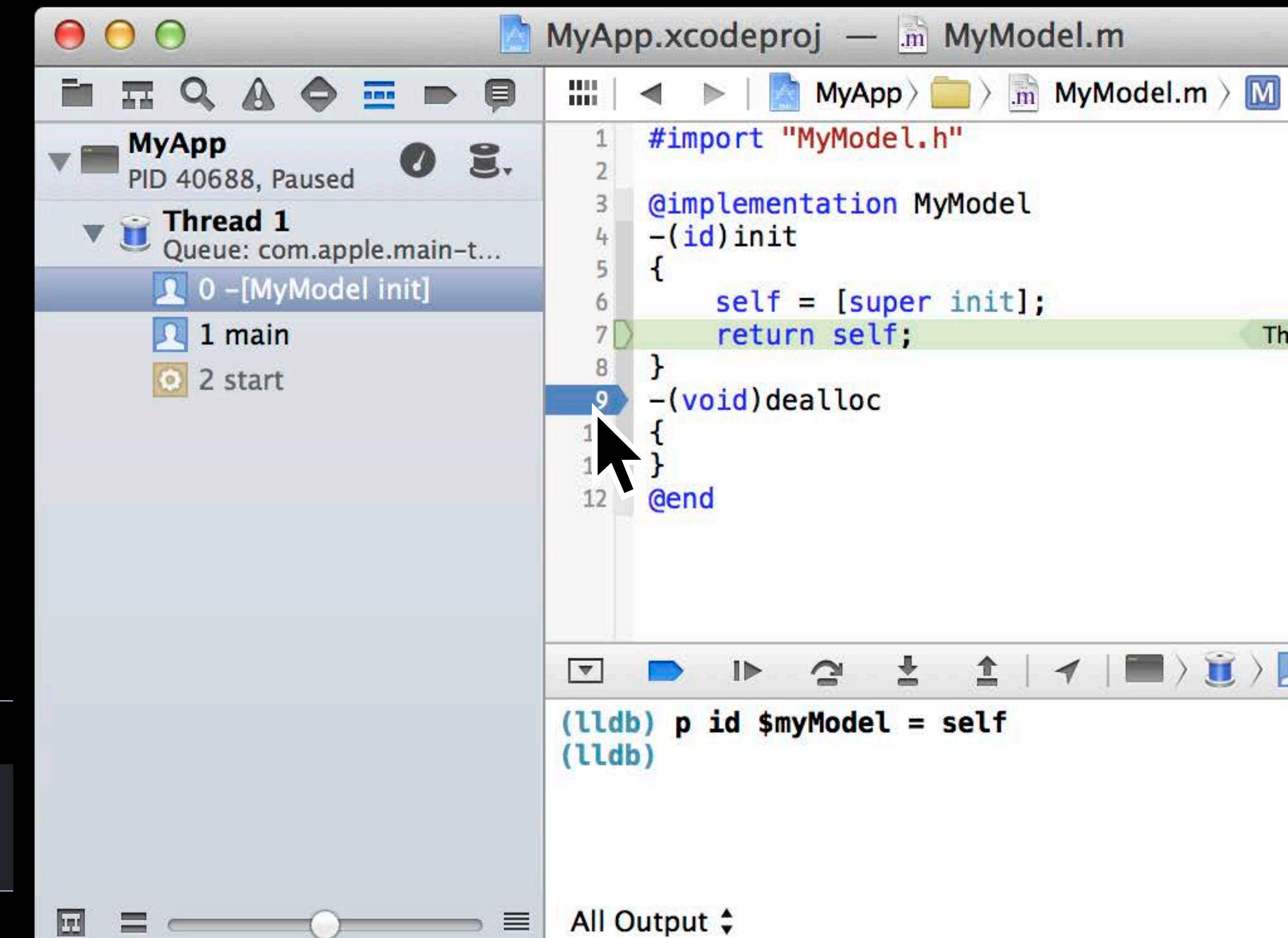
Creates a persistent variable of type id

```
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
```

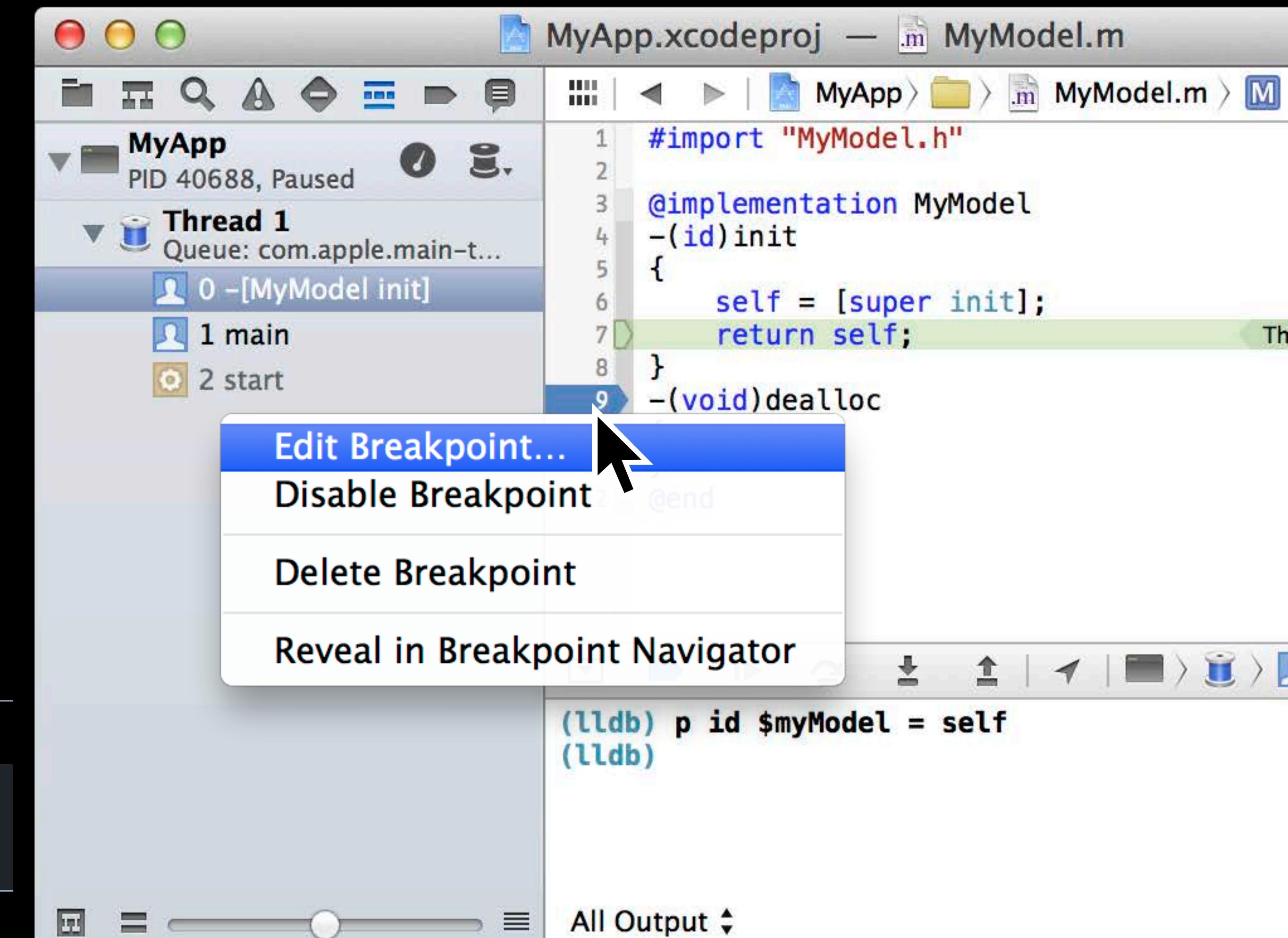
Creates a persistent variable of type id

```
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
```

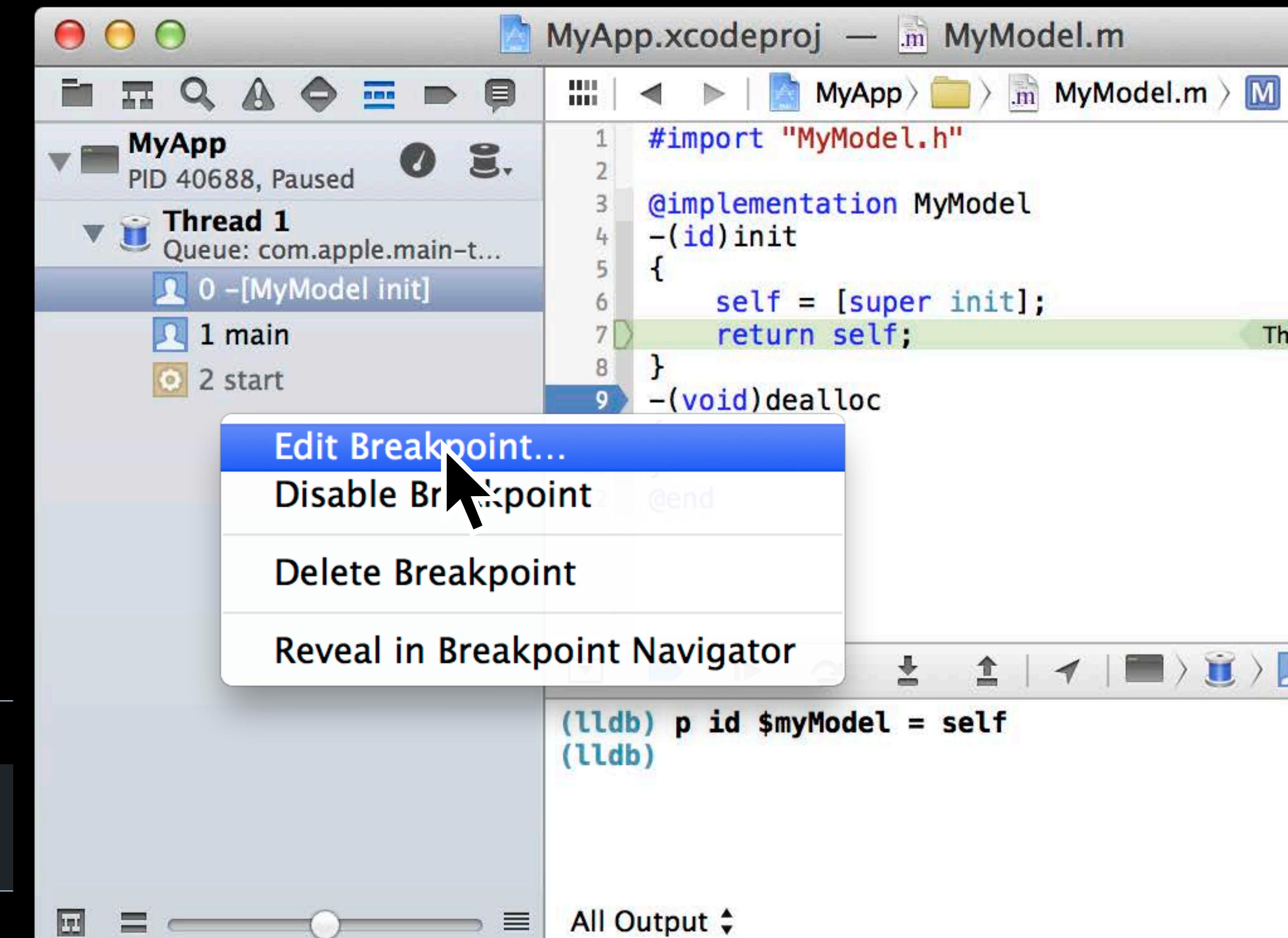
Creates a persistent variable of type id

```
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
```

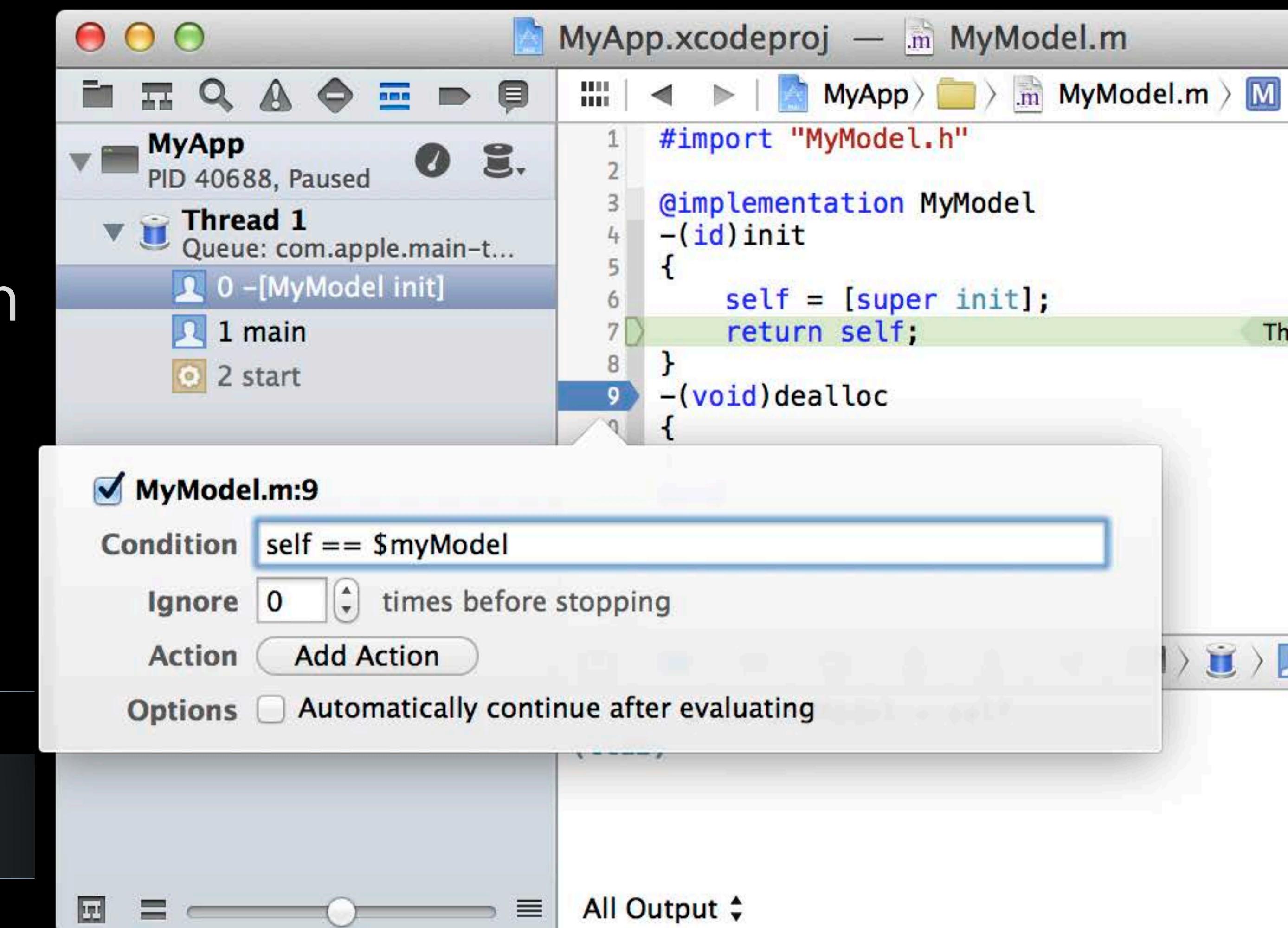
Creates a persistent variable of type id

```
expression id $myModel = self
```

```
b “[MyModel dealloc]”
```

```
br m -c “self == $myModel”
```

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



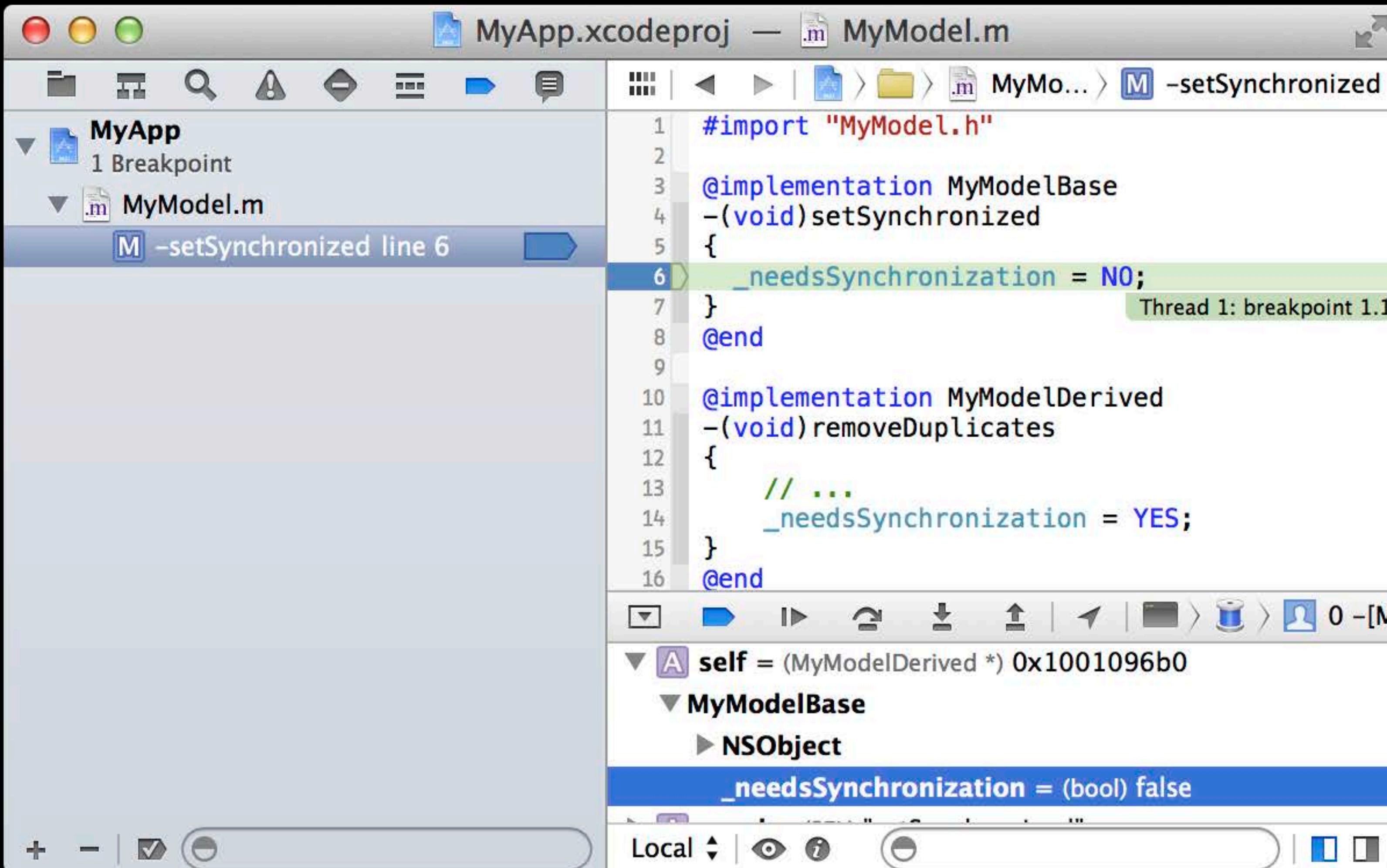
# Focus on Memory with Watchpoints

- 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

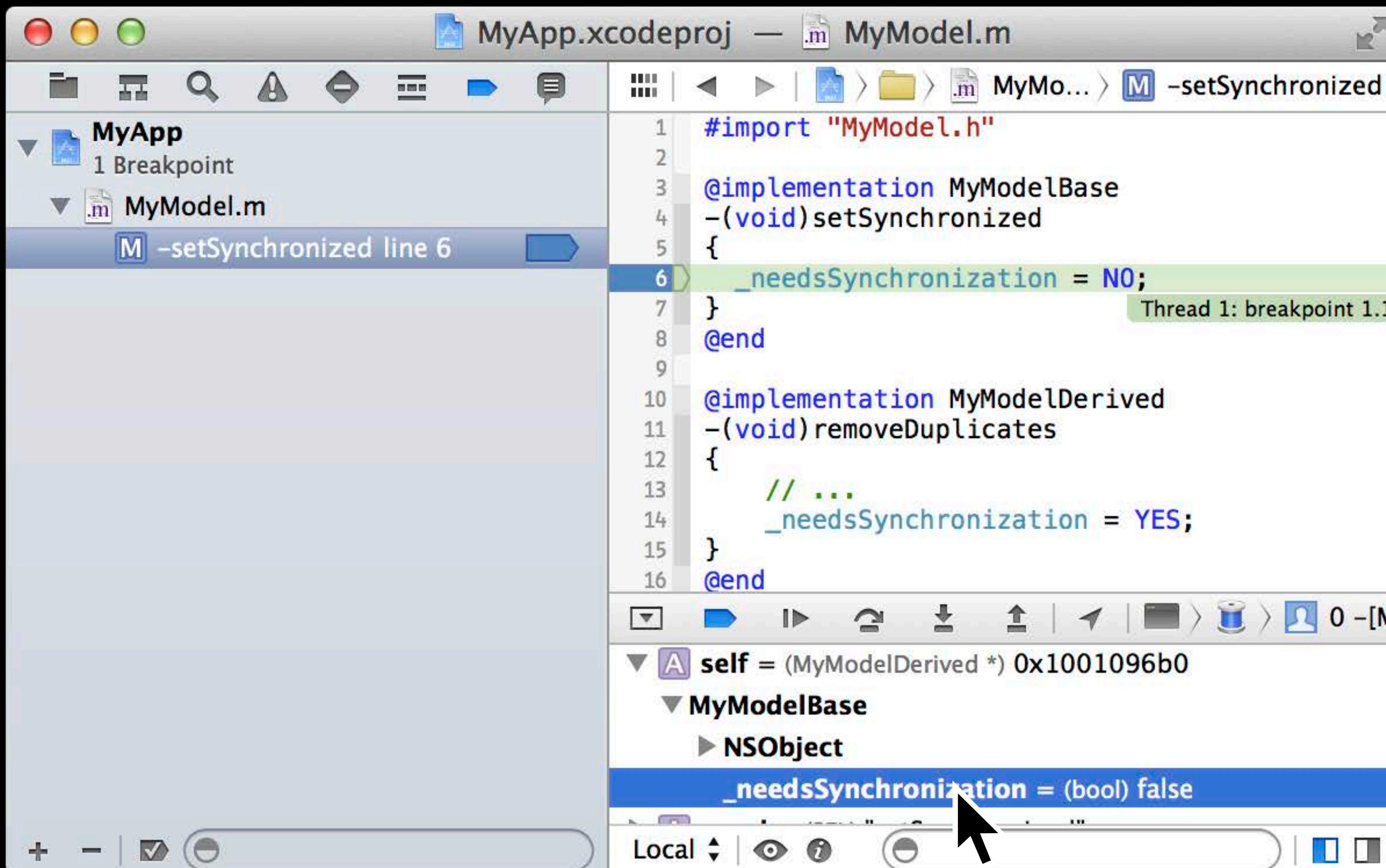


The screenshot shows the Xcode IDE interface. The top bar displays "MyApp.xcodeproj" and "MyModel.m". The left sidebar shows a project structure with "MyApp" and "MyModel.m". A breakpoint is set at line 6 of "MyModel.m", which is currently open in the main editor. The code is as follows:

```
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
```

The line `_needsSynchronization = NO;` is highlighted in green, and a tooltip "Thread 1: breakpoint 1.1" appears over it. The bottom right corner of the code editor has a status bar with "0 -[M]". The bottom pane shows the variable inspector with a list of variables. The variable `self` is expanded, showing its class hierarchy: `MyModelDerived` and `NSObject`. The value of `_needsSynchronization` is shown as `(bool) false`.

# Focus on Memory with Watchpoints

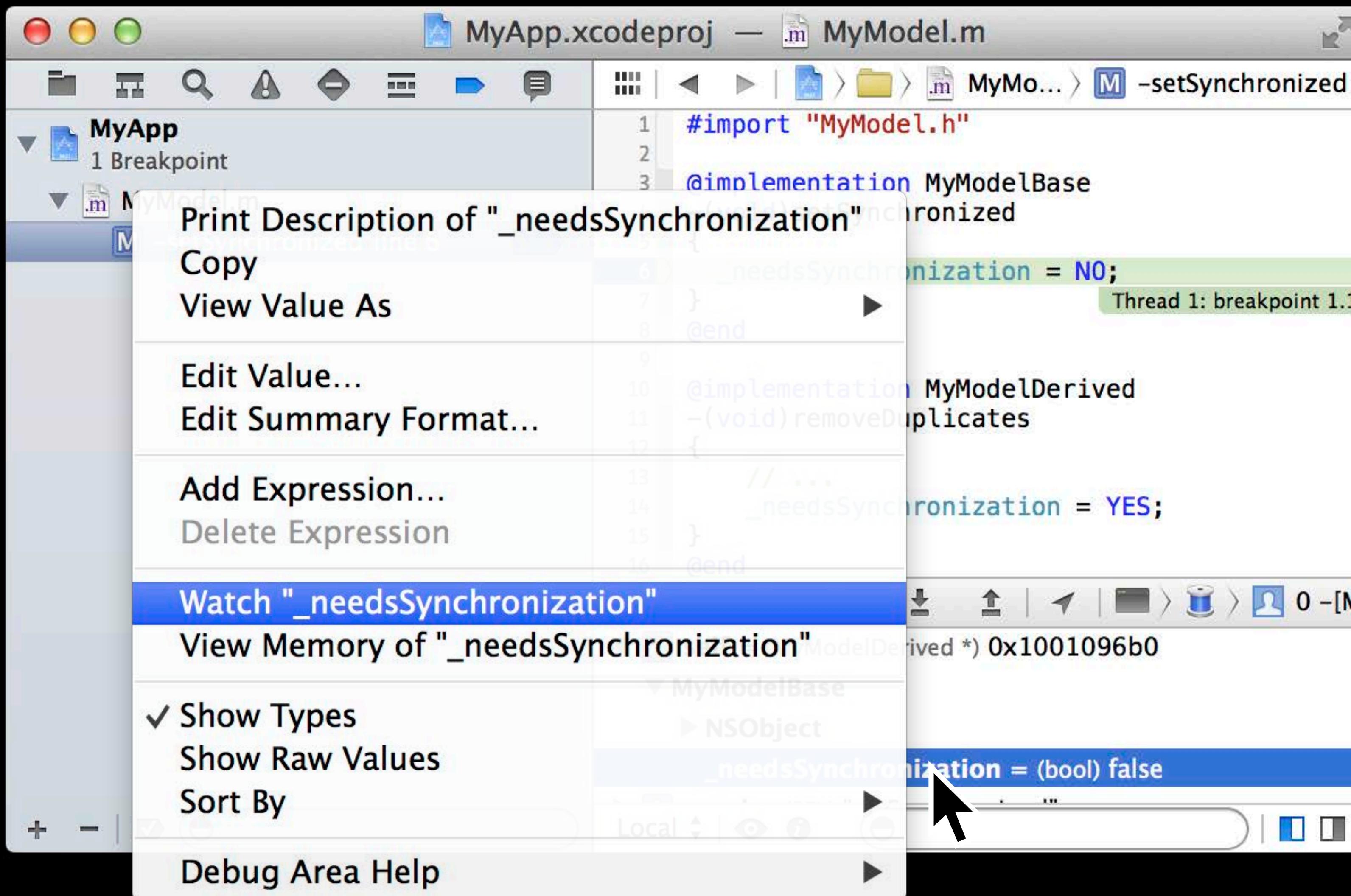


The screenshot shows the Xcode IDE interface with the following details:

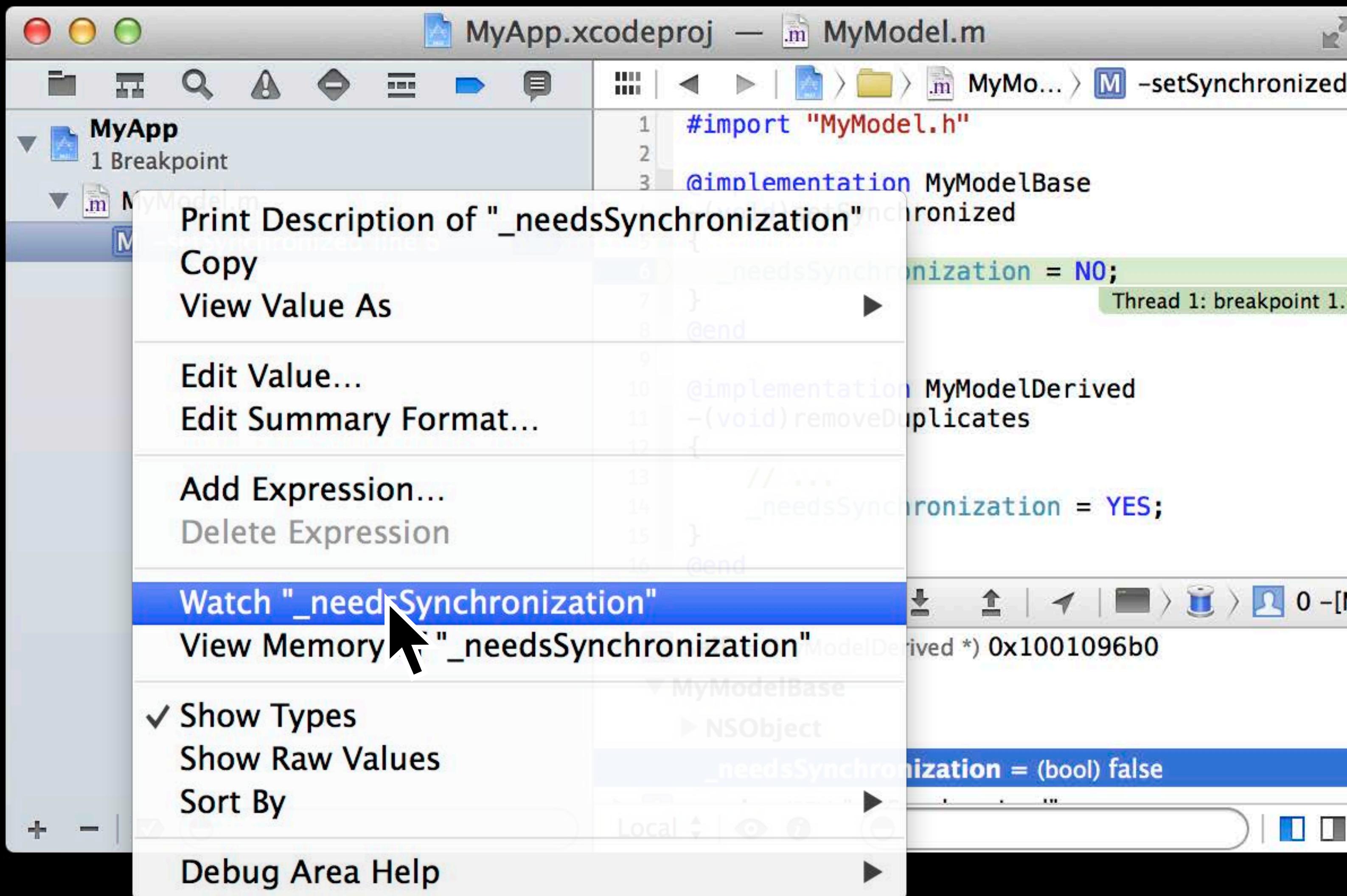
- Project:** MyApp.xcodeproj
- File:** MyModel.m
- Breakpoint:** -setSynchronized line 6
- Code:**

```
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
```
- Watchpoint:** A memory watchpoint is set on the variable `_needsSynchronization` at line 6. The variable is highlighted in green, and a tooltip indicates "Thread 1: breakpoint 1.1".
- Variables:** The variable `self` is shown in the sidebar, pointing to `MyModelDerived`, which in turn points to `NSObject`. The value of `_needsSynchronization` is listed as `(bool) false`.
- Toolbar:** The toolbar at the bottom includes icons for zooming, search, and other development tools.

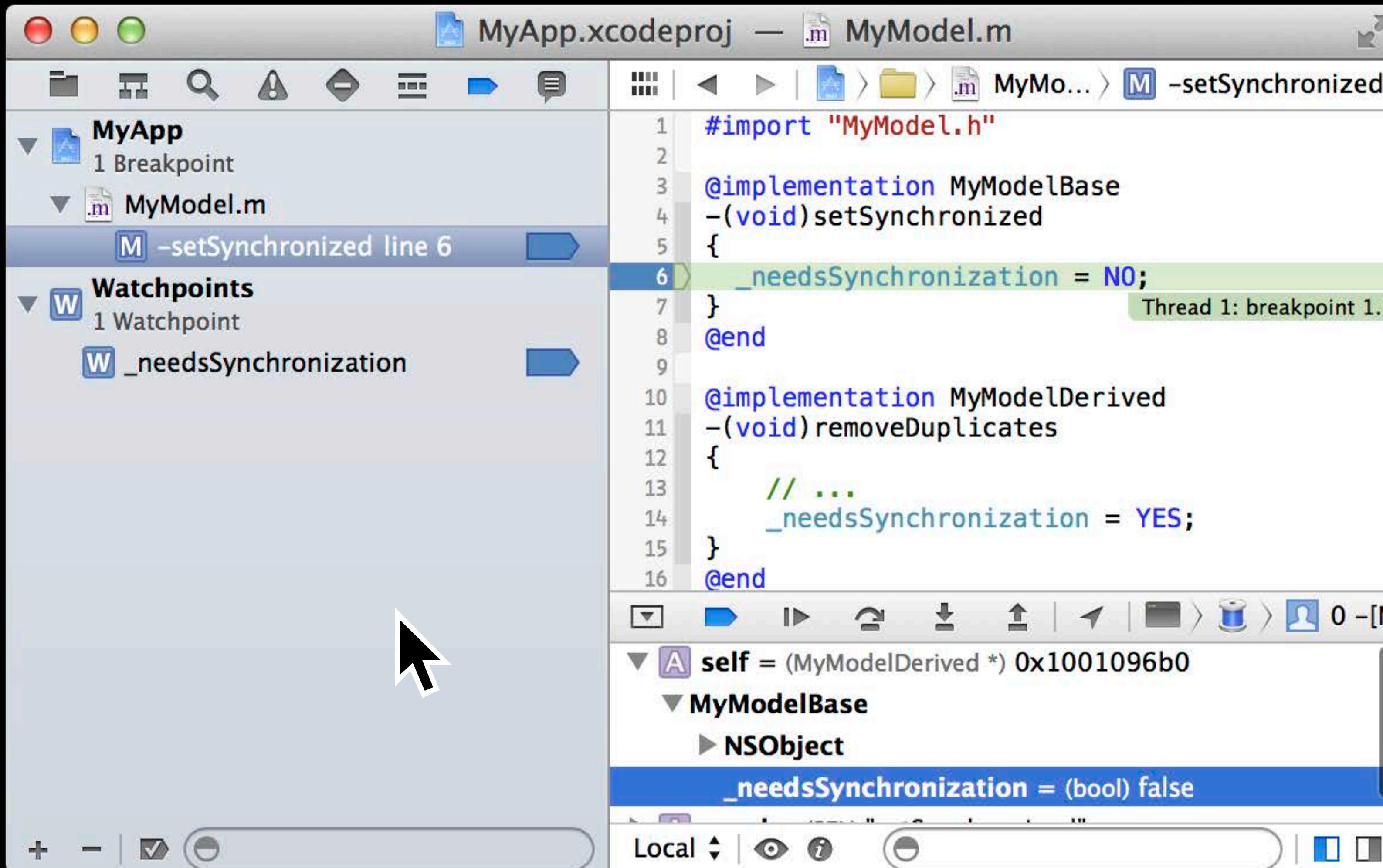
# Focus on Memory with Watchpoints



# Focus on Memory with Watchpoints



# Focus on Memory with Watchpoints



The screenshot shows the Xcode IDE interface with the following details:

- Project:** MyApp.xcodeproj
- File:** MyModel.m
- Breakpoint:** A blue arrow icon is shown next to the line number 6 in the code editor.
- Watchpoints:** A list on the left sidebar shows one watchpoint named "\_needsSynchronization".
- Code Editor:** The code for MyModel.m is displayed:

```
#import "MyModel.h"

@implementation MyModelBase
-(void)setSynchronized
{
 _needsSynchronization = NO;
}
@end

@implementation MyModelDerived
-(void)removeDuplicates
{
 // ...
 _needsSynchronization = YES;
}
@end
```
- Callout:** A callout box labeled "Thread 1: breakpoint 1.1" points to the line where the watchpoint was set.
- Variables View:** The bottom right shows the variable self with its type (MyModelDerived \*) and memory address (0x1001096b0). It also shows the value of the variable \_needsSynchronization, which is highlighted in blue and set to false.

# Focus on Memory with Watchpoints

The screenshot shows the Xcode IDE interface. The top bar displays the project name "MyApp.xcodeproj" and the file "MyModel.m". The left sidebar shows the project structure with "MyApp" and "MyModel.m" selected. Under "MyModel.m", there is a breakpoint at line 6 labeled "-setSynchronized line 6" and a watchpoint for the variable "\_needsSynchronization". The main editor area shows the implementation of the MyModelBase class with a breakpoint set on line 6. The variable "\_needsSynchronization" is highlighted in green. A tooltip indicates "Thread 1: breakpoint 1.1". The bottom right corner of the editor shows the memory address of the self pointer: "self = (MyModelDerived \*) 0x1001096b0". The bottom navigation bar includes buttons for Local, Global, and Stack, along with other debugger controls.

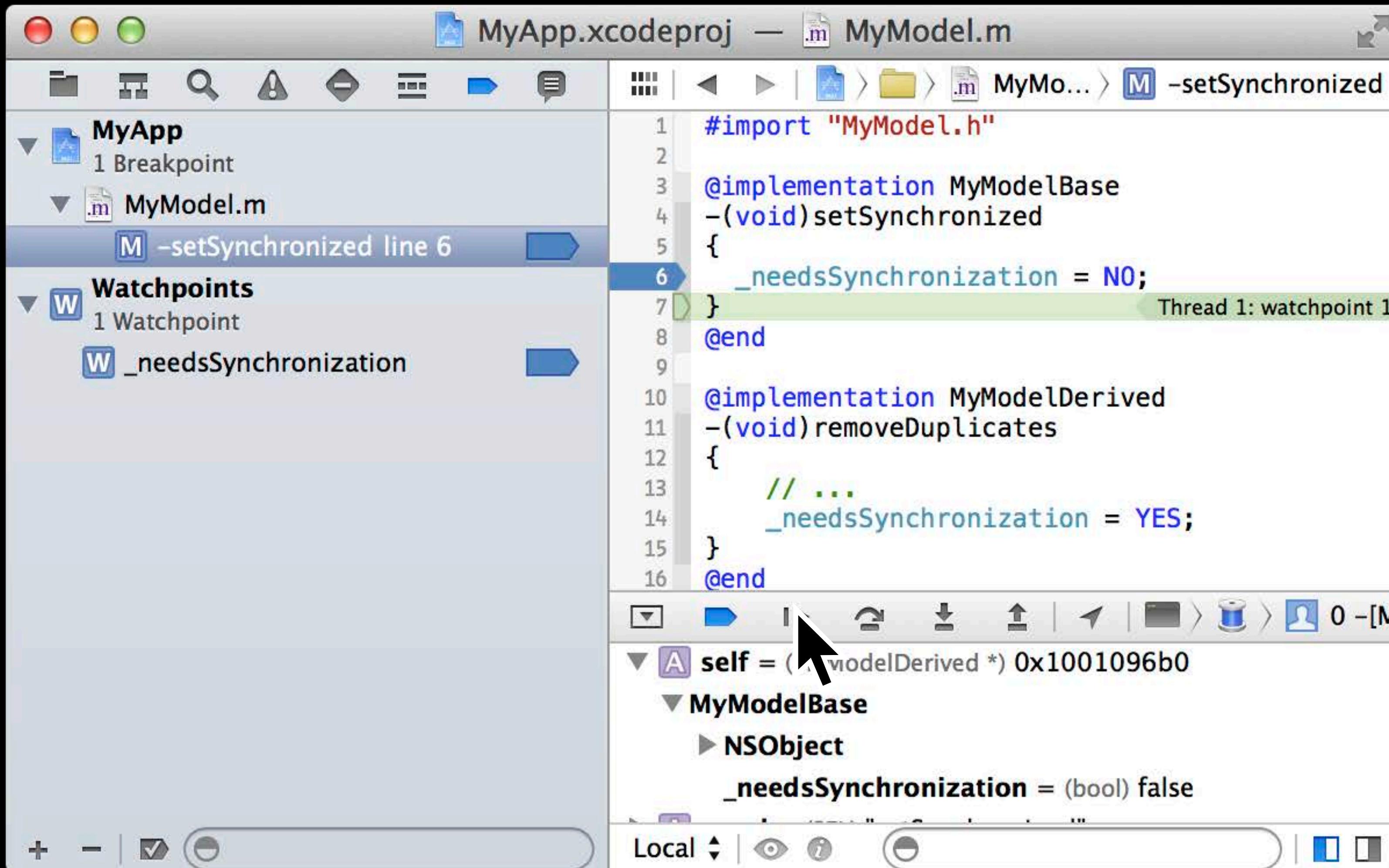
```
#import "MyModel.h"

@implementation MyModelBase
-(void)setSynchronized
{
 _needsSynchronization = NO;
}
@end

@implementation MyModelDerived
-(void)removeDuplicates
{
 // ...
 _needsSynchronization = YES;
}
@end
```

A mouse cursor is hovering over the memory address of the self pointer in the bottom right corner of the editor.

# Focus on Memory with Watchpoints



The screenshot shows the Xcode IDE interface during a debug session. The top bar displays the project name "MyApp.xcodeproj" and the file "MyModel.m". The left sidebar shows the project structure with "MyApp" and "MyModel.m" selected. Under "MyModel.m", there is a breakpoint at line 6 labeled "-setSynchronized line 6" and a watchpoint for the variable "\_needsSynchronization". The main editor area shows the implementation of the MyModelBase and MyModelDerived classes. Line 6 contains the assignment `_needsSynchronization = NO;`. A green highlight covers the code from line 7 to the end of the method. A tooltip "Thread 1: watchpoint 1" is visible near the cursor. The bottom right corner of the editor shows the status "0 -[M]". The bottom navigation bar includes buttons for "Local", "Breakpoint", and "Run". The bottom status bar shows memory usage information: "self = (MyModelDerived \*) 0x1001096b0".

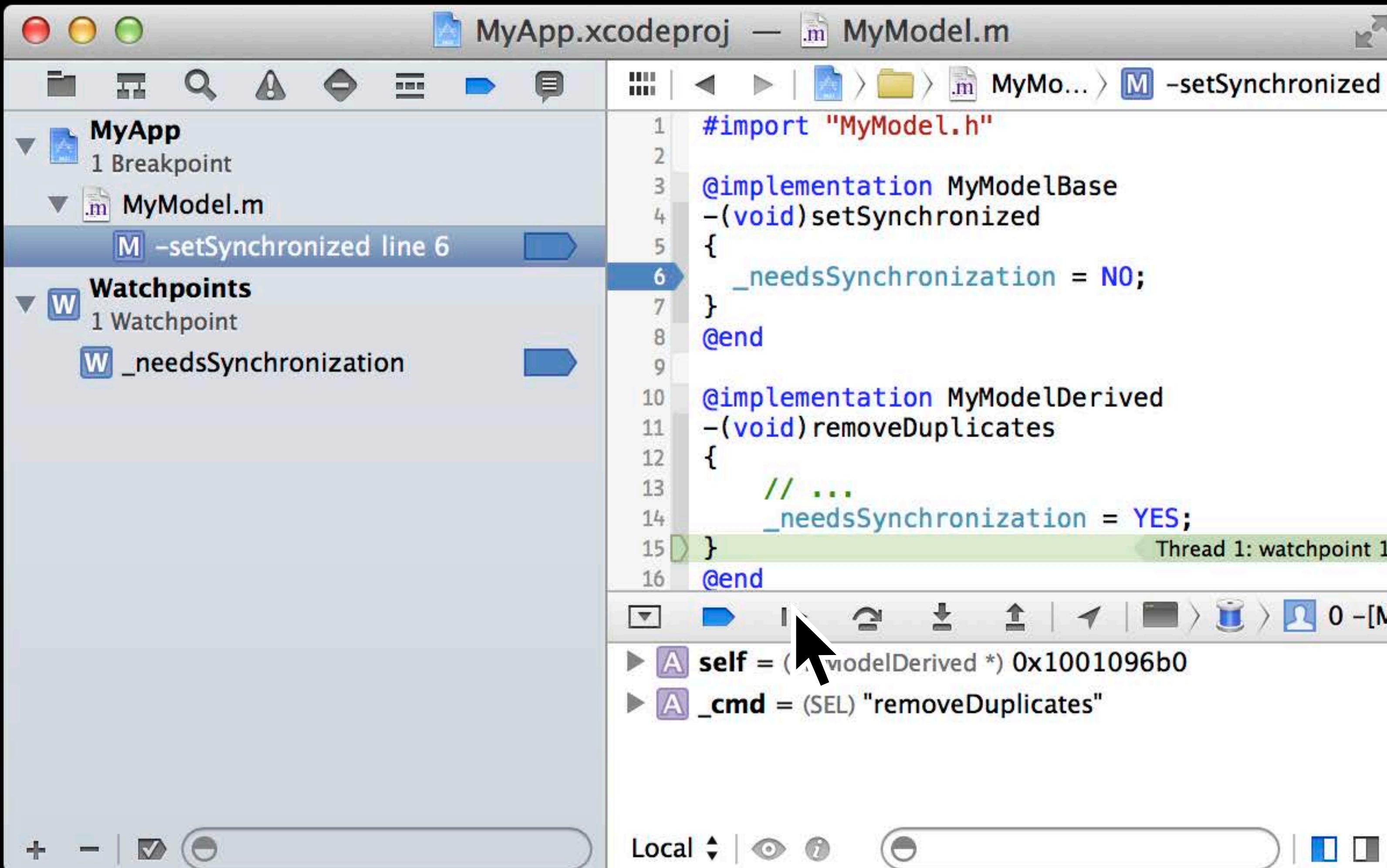
```
#import "MyModel.h"

@implementation MyModelBase
-(void)setSynchronized
{
 _needsSynchronization = NO;
}
@end

@implementation MyModelDerived
-(void)removeDuplicates
{
 // ...
 _needsSynchronization = YES;
}
@end
```

A mouse cursor is hovering over the "Run" button in the bottom navigation bar.

# Focus on Memory with Watchpoints



The screenshot shows the Xcode IDE interface with the following details:

- Project:** MyApp.xcodeproj
- File:** MyModel.m
- Breakpoint:** -setSynchronized line 6
- Watchpoints:** \_needsSynchronization
- Code:**

```
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
```
- Call Stack:**
  - A self = (MyModelDerived \*) 0x1001096b0
  - A \_cmd = (SEL) "removeDuplicates"
- Thread:** Thread 1: watchpoint 1

# 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

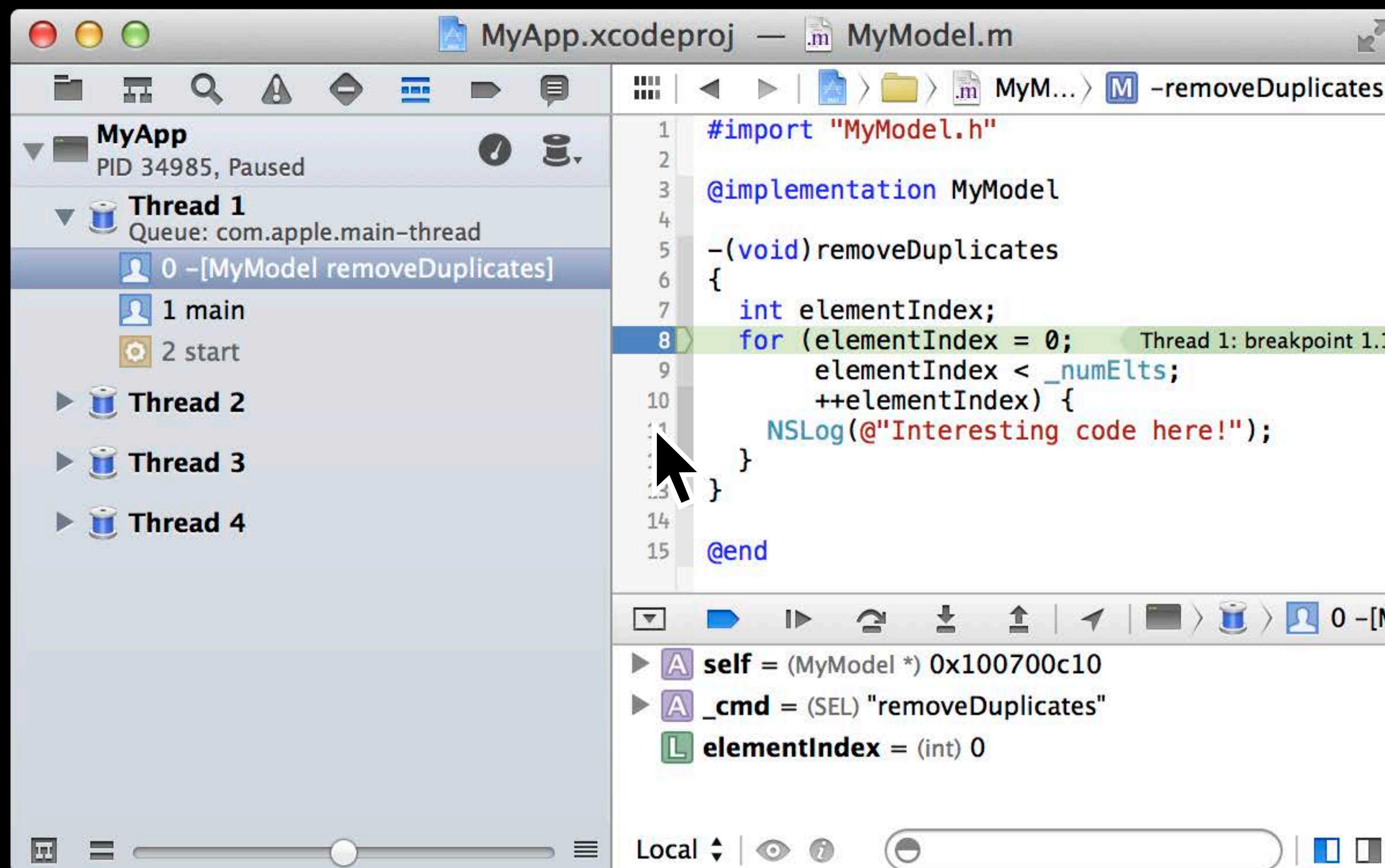
The screenshot shows the Xcode interface during a debugging session. The title bar indicates the project is "MyApp.xcodeproj" and the file is "MyModel.m". The left sidebar shows the application "MyApp" is running with PID 34985 and is Paused. Under "Thread 1", the current stack frame is highlighted: "0 -[MyModel removeDuplicates]". Below it, the call stack includes "1 main" and "2 start". Other threads are listed as "Thread 2", "Thread 3", and "Thread 4". The right pane displays the code for the `-removeDuplicates` method:

```
#import "MyModel.h"
@implementation MyModel
-(void)removeDuplicates
{
 int elementIndex;
 for (elementIndex = 0; elementIndex < _numElts;
 ++elementIndex) {
 NSLog(@"Interesting code here!");
 }
}
@end
```

The line `NSLog(@"Interesting code here!");` is currently being executed, as indicated by the green highlighting of the code area. A tooltip "Thread 1: breakpoint 1.1" appears over the line number 8. The bottom of the screen shows the Local variable inspector with the following values:

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

# Avoiding Repeated Steps

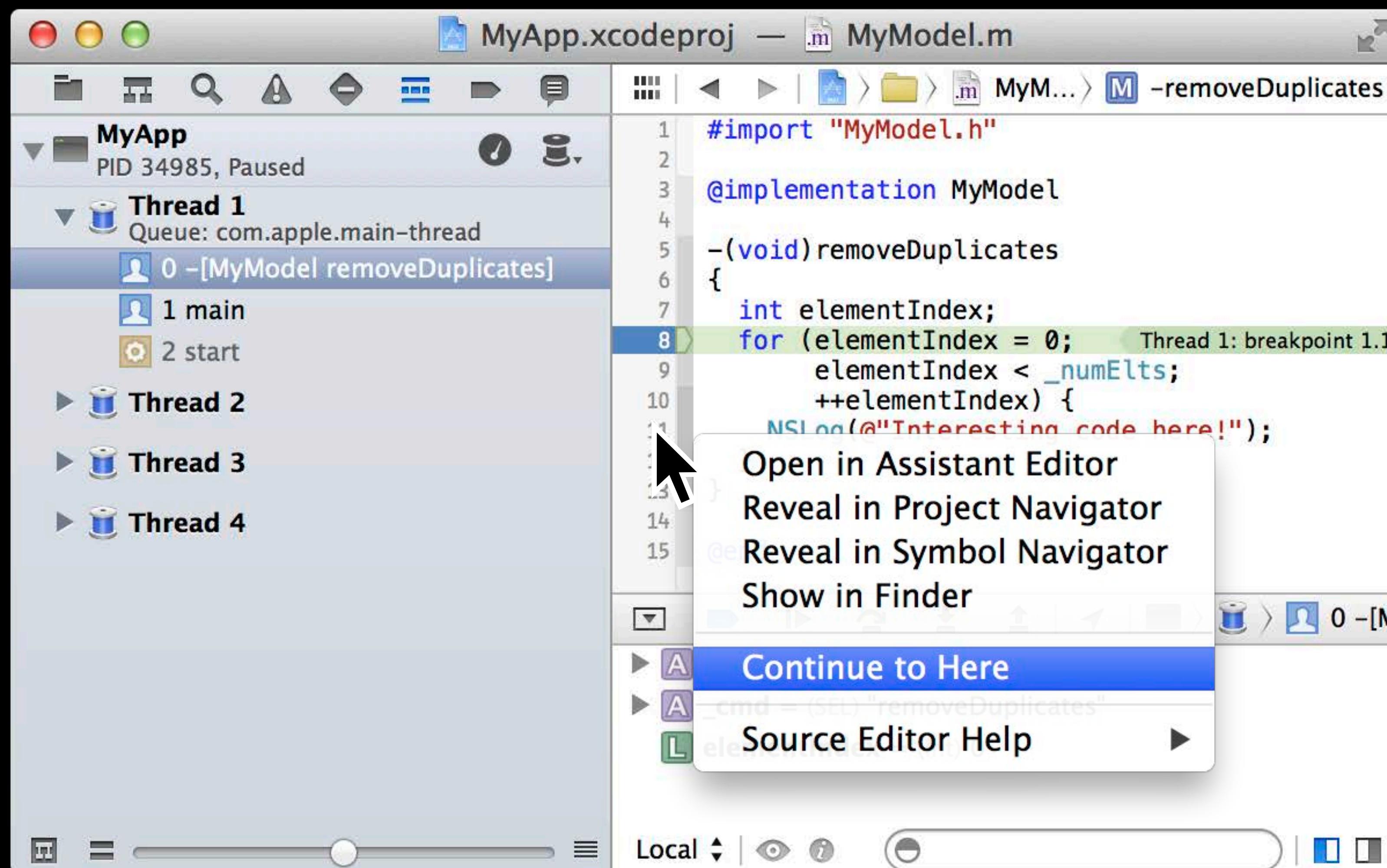


The screenshot shows the Xcode interface during a debugging session. The title bar indicates the project is "MyApp.xcodeproj" and the file is "MyModel.m". The left sidebar shows the application "MyApp" is running with PID 34985 and is Paused. Under "Thread 1", the current stack frame is "0 -[MyModel removeDuplicates]". The code editor displays the implementation of the `-removeDuplicates` method:

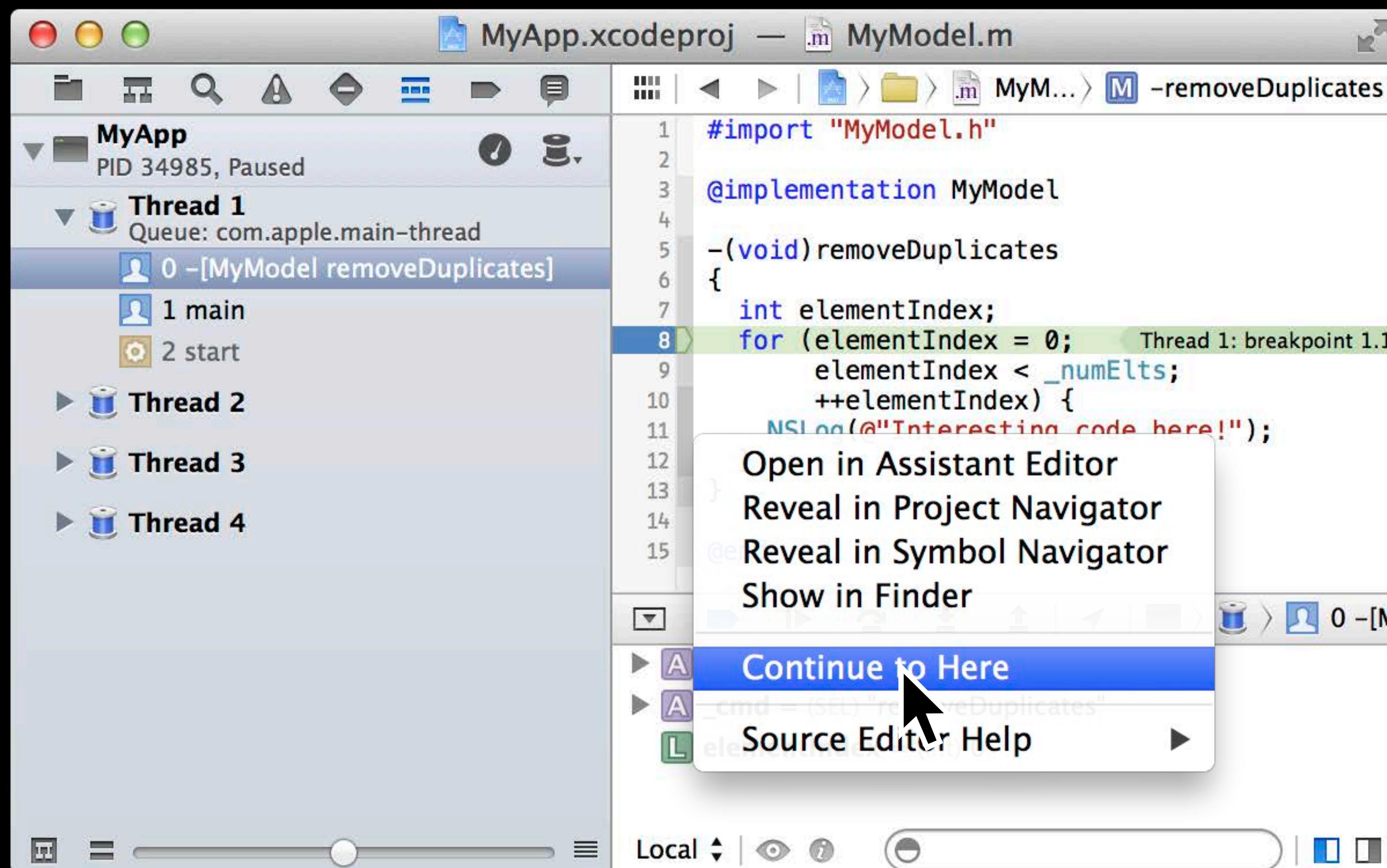
```
#import "MyModel.h"
@implementation MyModel
-(void)removeDuplicates
{
 int elementIndex;
 for (elementIndex = 0; elementIndex < _numElts;
 ++elementIndex) {
 NSLog(@"Interesting code here!");
 }
}
@end
```

A mouse cursor is hovering over the line `NSLog(@"Interesting code here!");`. A tooltip "Thread 1: breakpoint 1.1" appears near the cursor. The bottom-left corner of the Xcode window shows the status bar with "Local".

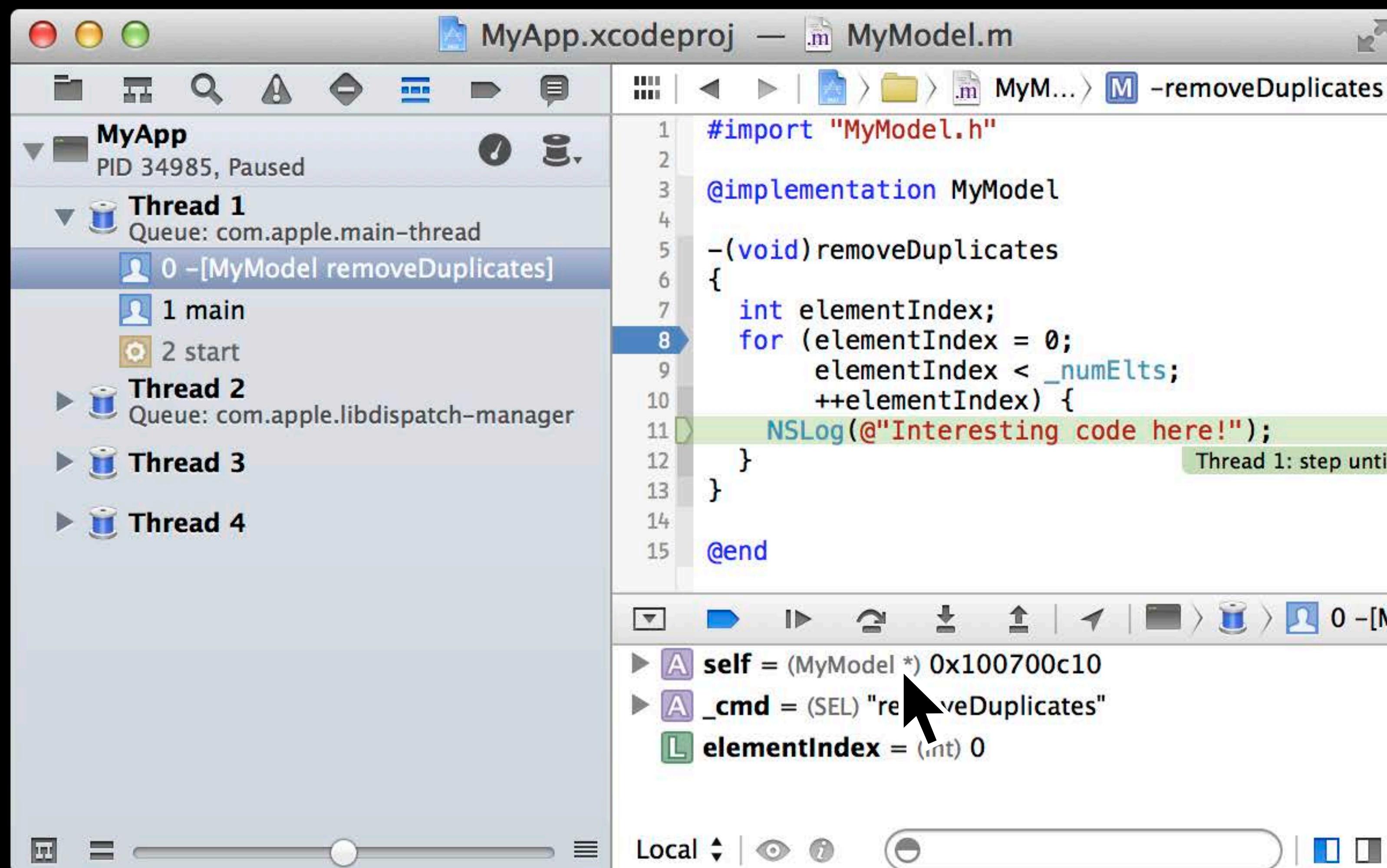
# Avoiding Repeated Steps



# Avoiding Repeated Steps



# Avoiding Repeated Steps



# Hitting Breakpoints While Stepping

The screenshot shows the Xcode IDE interface during a debugging session. The title bar indicates the project is "MyApp.xcodeproj" and the file is "main.m".

The left sidebar shows the project structure with "MyApp" selected, and the "Thread 1" list, which includes "0 main" and "1 start".

The main area displays the "main.m" code:

```
#import <Foundation/Foundation.h>
#import "MyModel.h"

int main(int argc, const char * argv[])
{
 @autoreleasepool {
 MyModel *model = [MyModel new];
 [model removeDuplicates];
 NSLog(@"%@", @"Code I care about");
 }
 return 0;
}
```

The line "[model removeDuplicates];" is highlighted in green, indicating it is the current line of execution. A tooltip "Thread 1: breakpoint 2.1" appears over this line.

The bottom right corner of the code editor has a small black arrow pointing towards the Local variable list.

The Local variable list shows the following variables:

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

# Hitting Breakpoints While Stepping

The screenshot shows the Xcode IDE interface during a debugging session. The title bar indicates the project is "MyApp.xcodeproj" and the file is "main.m".

The left sidebar shows the project structure with "MyApp" selected, and the "Thread 1" tab is active, showing the thread is paused at PID 39887. The current stack frame is "0 main".

The right pane displays the code in "main.m". Line 9 contains the instruction `[model removeDuplicates];`. A green highlight covers the entire line, and a tooltip above it says "Thread 1: breakpoint 2.1".

The bottom right corner of the code editor has a small black arrow icon pointing upwards.

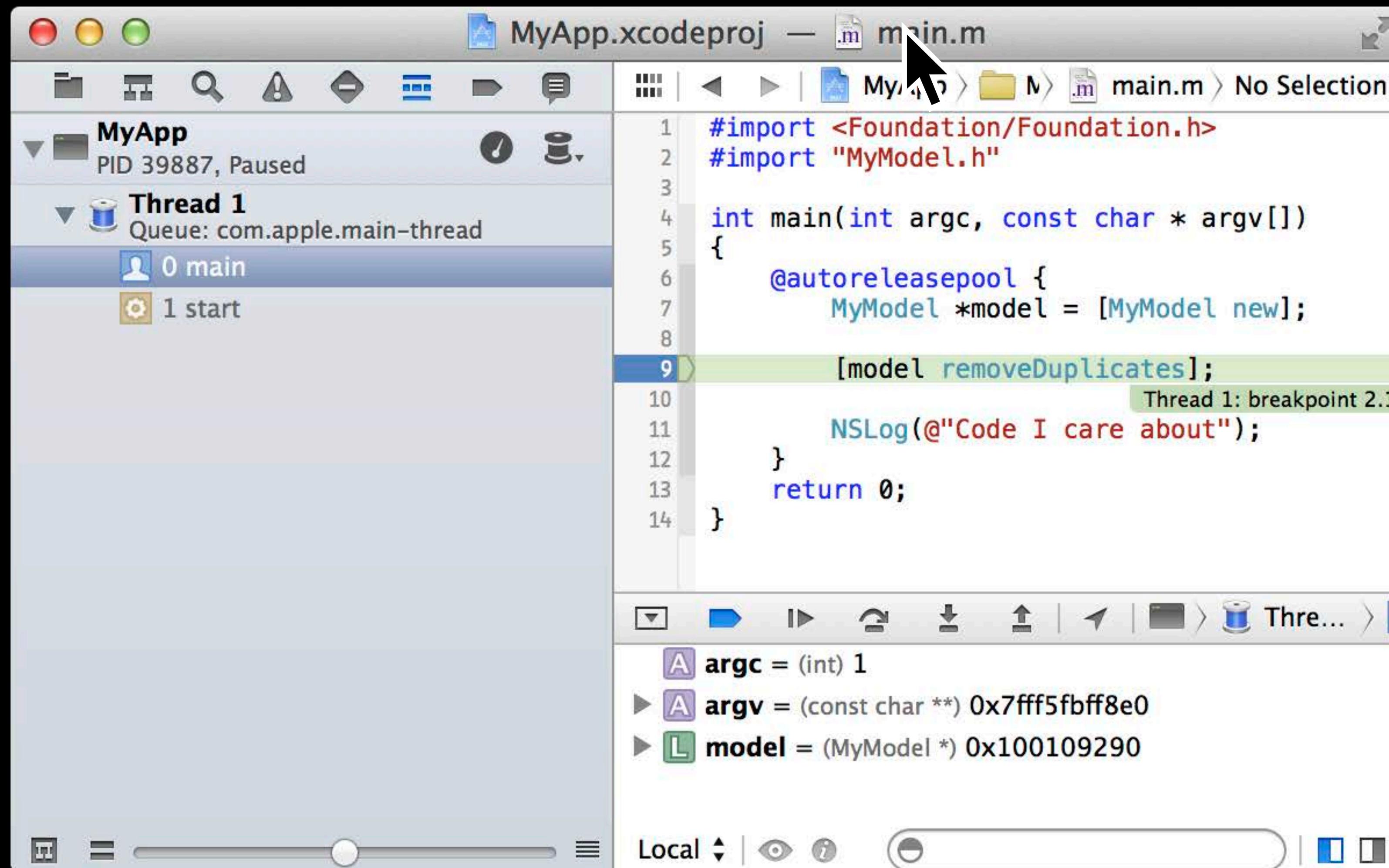
The bottom status bar shows local variables: `argc = (int) 1`, `argv = (const char **) 0x7fff5fbff8e0`, and `model = (MyModel *) 0x100109290`.

At the bottom center, there is a "Local" dropdown menu and a toolbar with various debugging icons.

```
#import <Foundation/Foundation.h>
#import "MyModel.h"

int main(int argc, const char * argv[])
{
 @autoreleasepool {
 MyModel *model = [MyModel new];
 [model removeDuplicates];
 NSLog(@"%@", @"Code I care about");
 }
 return 0;
}
```

# Hitting Breakpoints While Stepping



The screenshot shows the Xcode IDE interface during a debugging session. The title bar indicates the project is "MyApp.xcodeproj" and the file is "main.m".

The left sidebar shows the project structure with "MyApp" selected, and the "Thread 1" tab is active, showing "Queue: com.apple.main-thread" and the current thread is "0 main".

The right pane displays the code in "main.m":

```
#import <Foundation/Foundation.h>
#import "MyModel.h"

int main(int argc, const char * argv[])
{
 @autoreleasepool {
 MyModel *model = [MyModel new];
 [model removeDuplicates];
 NSLog(@"%@", @"Code I care about");
 }
 return 0;
}
```

The line "[model removeDuplicates];" is highlighted in green, indicating it is the current instruction being executed. A tooltip "Thread 1: breakpoint 2.1" appears over this line.

The bottom pane shows the "Local" variable inspector with the following values:

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

# Hitting Breakpoints While Stepping

The screenshot shows the Xcode IDE interface during a debugging session. The main window displays the file `main.m` with the following code:

```
#import <Foundation/Foundation.h>
#import "MyModel.h"

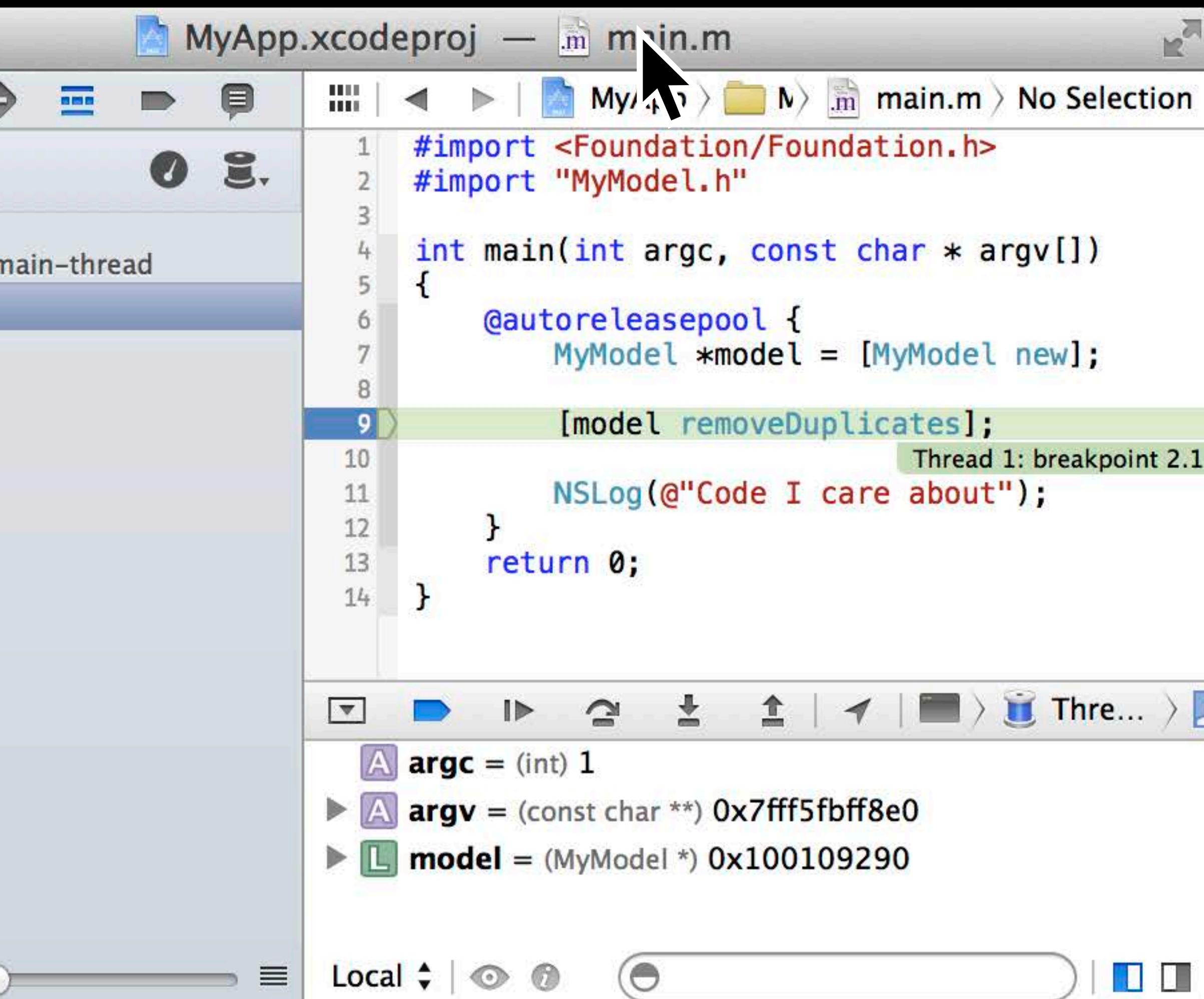
int main(int argc, const char * argv[])
{
 @autoreleasepool {
 MyModel *model = [MyModel new];

 [model removeDuplicates];
 NSLog(@"Code I care about");
 }
 return 0;
}
```

A cursor arrow points to the line `[model removeDuplicates];`. This line is highlighted in green, indicating it is the current instruction being executed. A tooltip above the line reads `Thread 1: breakpoint 2.1`. The bottom-left corner of the Xcode interface shows the current thread is `main-thread`.

In the bottom right corner of the Xcode interface, there is a small icon consisting of two overlapping squares, one blue and one grey.

# Hitting Breakpoints While Stepping



The screenshot shows the Xcode IDE interface during a debugging session. The main window displays the file `main.m` with the following code:

```
#import <Foundation/Foundation.h>
#import "MyModel.h"

int main(int argc, const char * argv[])
{
 @autoreleasepool {
 MyModel *model = [MyModel new];
 [model removeDuplicates];
 NSLog(@"Code I care about");
 }
 return 0;
}
```

A cursor is positioned over the line `[model removeDuplicates];`. The line is highlighted with a green background, and a tooltip `Thread 1: breakpoint 2.1` is visible above it. The code editor has a sidebar labeled `main-thread`. Below the editor is the debugger's control bar with various step and run buttons. The variable inspector shows the following local variables:

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

The bottom of the screen shows the standard Xcode navigation and search bars.

- 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 interface. The top bar displays the project name "MyApp.xcodeproj" and the file "main.m". The main window shows the code for "main.m":

```
#import <Foundation/Foundation.h>
#import "MyModel.h"

int main(int argc, const char * argv[])
{
 @autoreleasepool {
 MyModel *model = [MyModel new];
 [model removeDuplicates];
 NSLog(@"Code I care about");
 }
 return 0;
}
```

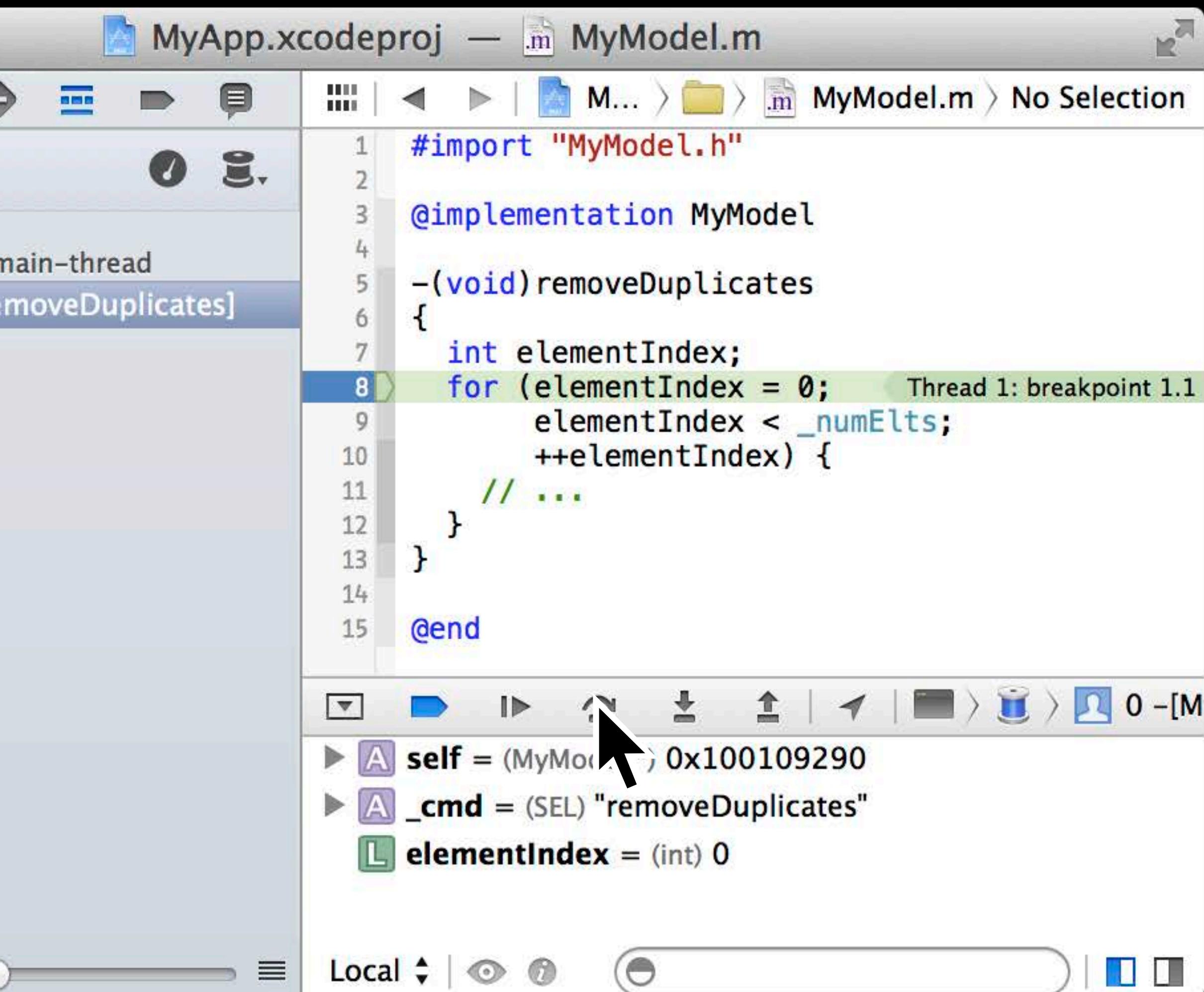
A cursor is positioned over the line "[model removeDuplicates];". A tooltip "Thread 1: breakpoint 2.1" appears above the line. The bottom part of the interface shows the "Variables" and "Registers" panes. The "Variables" pane lists local variables:

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

An arrow points to the step-up button in the toolbar.

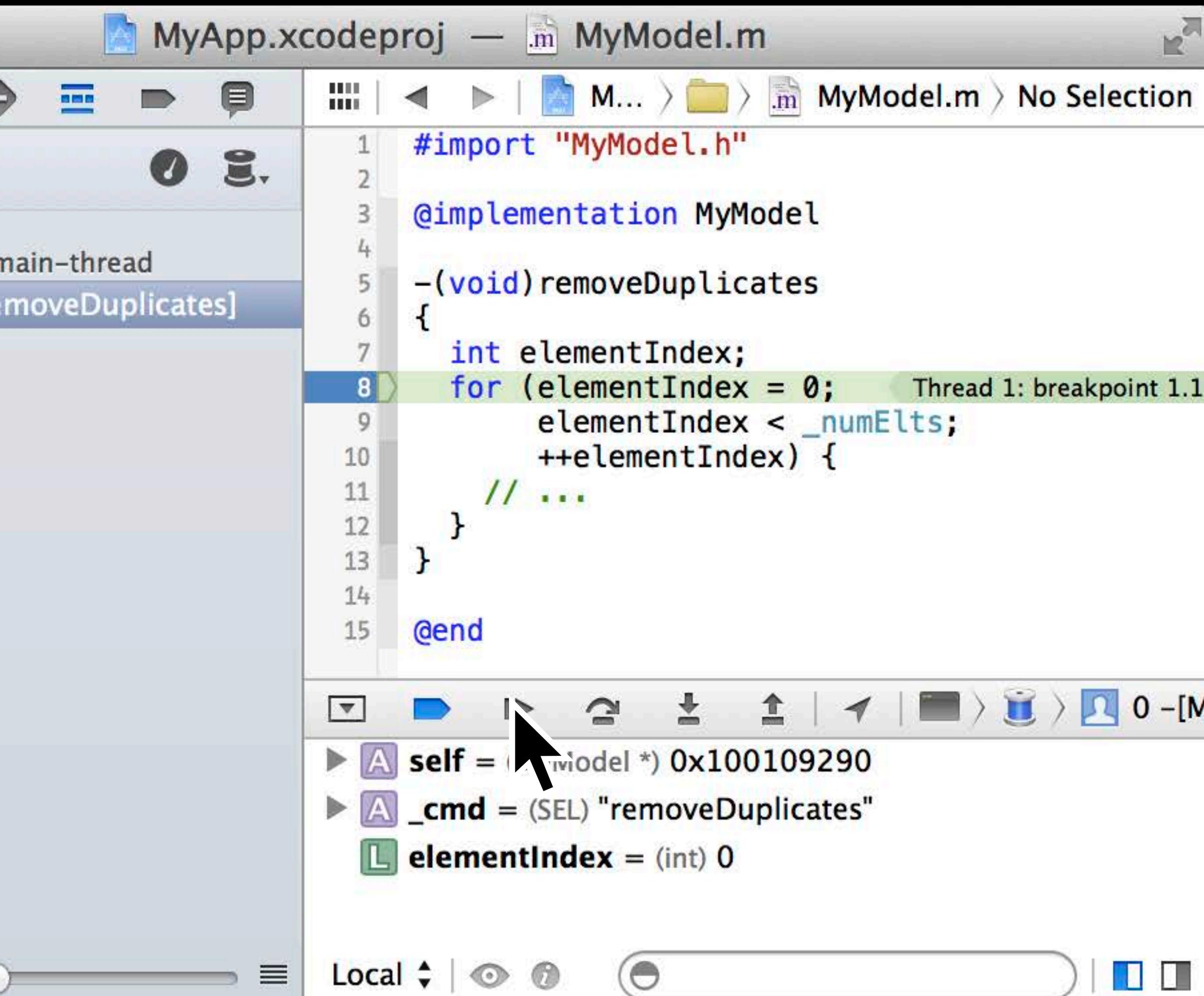
- Stepping can hit breakpoints
- LLDB maintains a stack of things you are doing
  - When you step, LLDB puts it on the stack

# 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
    - If you hit a breakpoint, LLDB remembers 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
    - If you hit a breakpoint, LLDB remembers the stack...

# Hitting Breakpoints While Stepping

The screenshot shows the Xcode IDE interface. The top bar displays 'MyApp.xcodeproj' and 'main.m'. The main window shows the code for 'main.m':

```
#import <Foundation/Foundation.h>
#import "MyModel.h"

int main(int argc, const char * argv[])
{
 @autoreleasepool {
 MyModel *model = [MyModel new];
 [model removeDuplicates];
 NSLog(@"%@", @"Code I care about");
 }
 return 0;
}
```

The line 'NSLog(@"%@", @"Code I care about");' is highlighted in green, indicating it is the current line of execution. A blue arrow points to the line number 9, which is also highlighted in blue. The status bar at the bottom right says 'Thread 1: step over'. The bottom panel shows the Local variable inspector with variables 'argc', 'argv', and 'model' listed.

- 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

frame variable

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

Show all my locals

expression (x + 35)

```
(int) $5 = 36
```

Execute arbitrary code

## Command / Output

## When to Use

frame variable

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

Show all my locals

expression (x + 35)

```
(int) $5 = 36
```

Execute arbitrary code

p @"Hello"

```
(NSString *) $6 = @"Hello"
```

Compact syntax for expression  
Allows GDB-style format (p/x)

## Command / Output

## When to Use

**frame variable**

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

Show all my locals

**expression (x + 35)**

```
(int) $5 = 36
```

Execute arbitrary code

**p @"Hello"**

```
(NSString *) $6 = @"Hello"
```

Compact syntax for **expression**  
Allows GDB-style format (**p/x**)

**po @"Hello"**

```
Hello
```

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

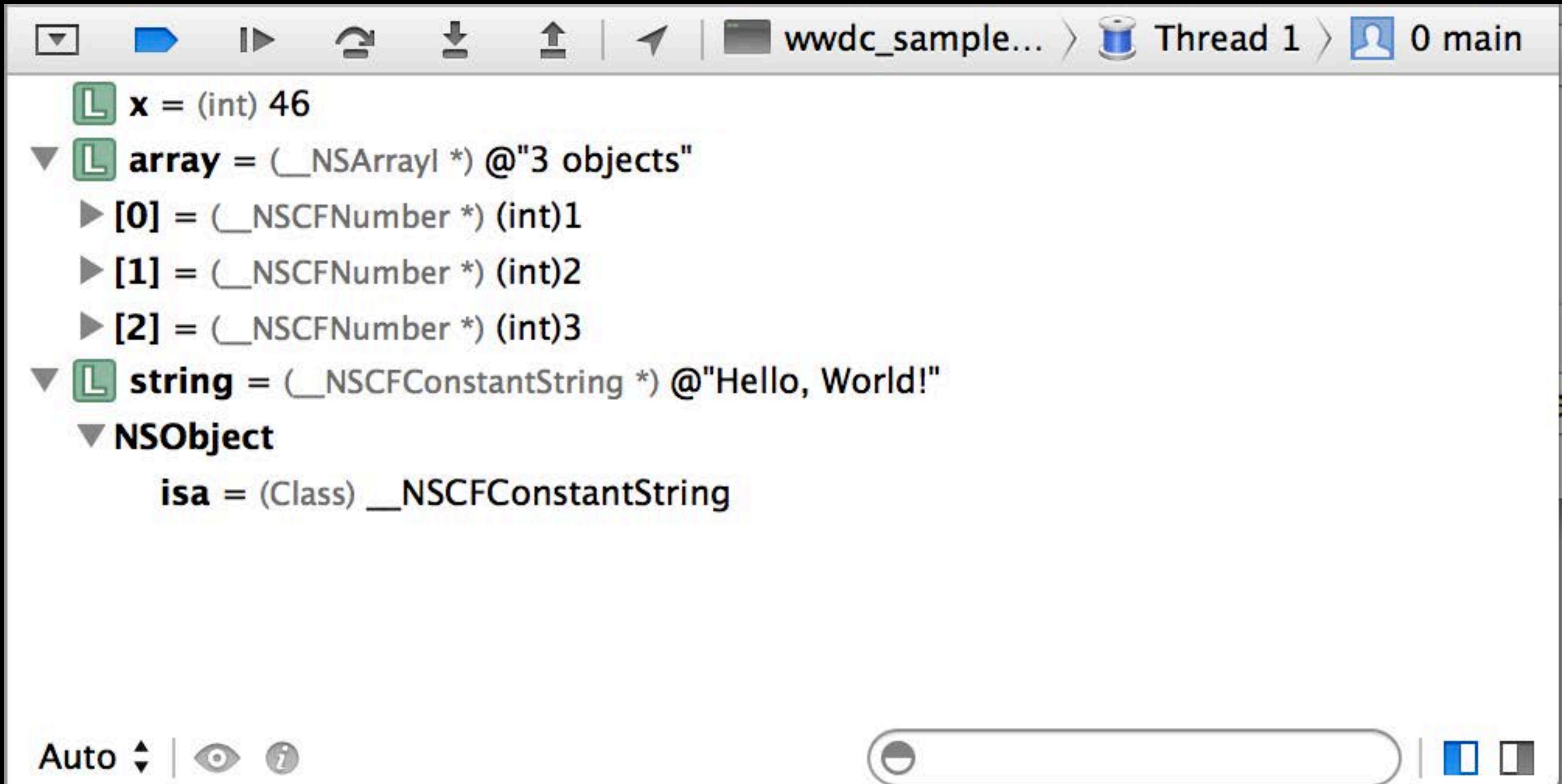
The screenshot shows the Xcode interface during a debugging session. The top bar indicates the project is "wwdc\_sample...", the thread is "Thread 1", and the current stack frame is "0 main". The navigation bar includes standard icons for navigating through memory.

The main content area displays a hierarchical breakdown of variables:

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

# “Data”

## Life with formatters



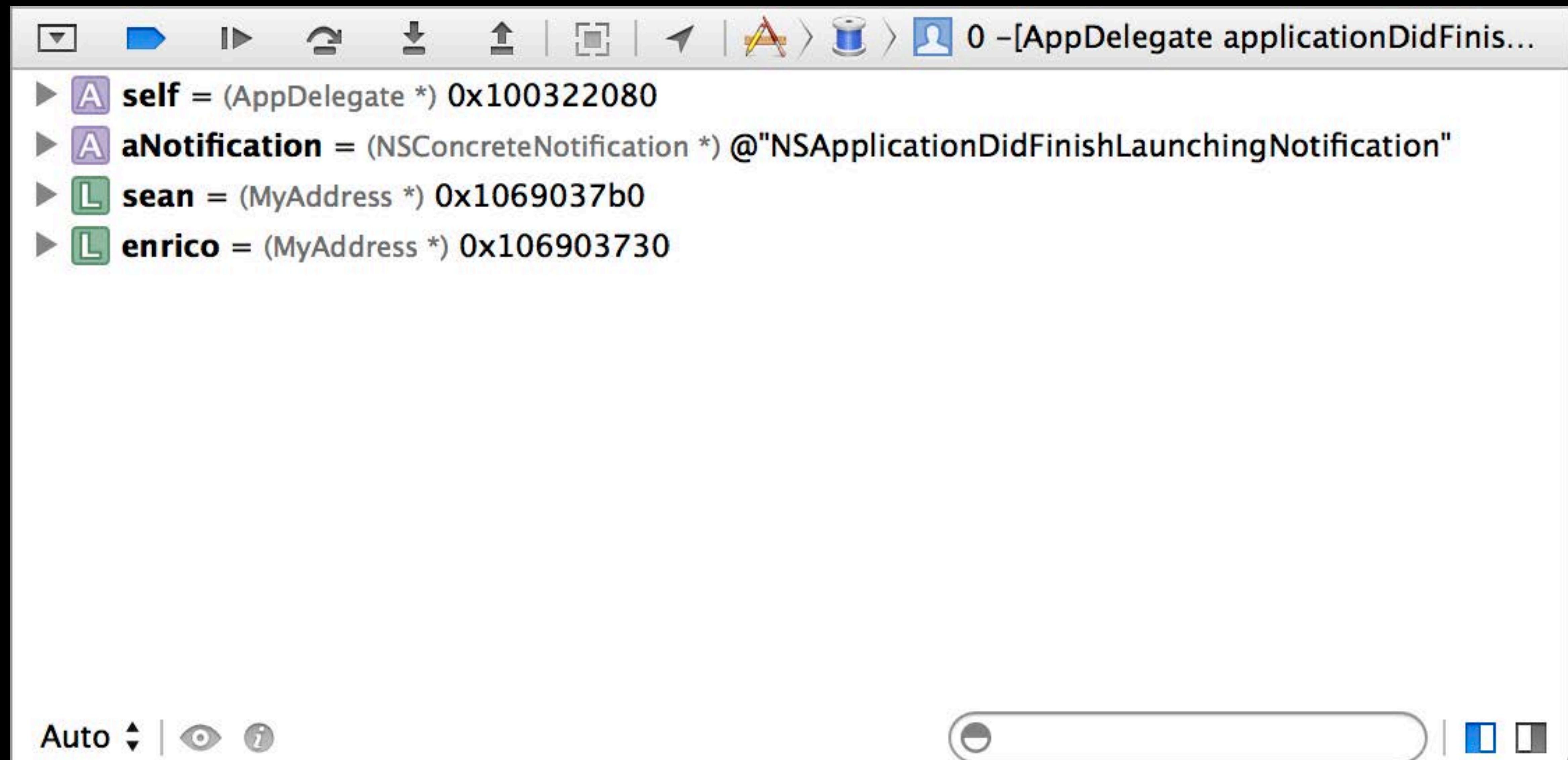
The screenshot shows the Xcode debugger interface with the following variable hierarchy:

- 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 are buttons for "Auto" (with dropdown), "Breakpoint", "Step Over", "Step Into", "Step Out", and "Run".

# “Raw Data”

## Life without formatters



The screenshot shows the Xcode debugger interface with the following details:

- Toolbar:** Includes standard navigation icons (back, forward, search, etc.) and a target selector.
- Text Area:** Displays variable declarations:
  - A self = (AppDelegate \*) 0x100322080**
  - A aNotification = (NSConcreteNotification \*) @"NSApplicationDidFinishLaunchingNotification"**
  - L sean = (MyAddress \*) 0x1069037b0**
  - L enrico = (MyAddress \*) 0x106903730**
- Bottom Bar:** Shows "Auto" mode, a search field, and other interface controls.

# “Raw Data”

## Life without formatters

The screenshot shows the Xcode debugger interface with the following details:

- Toolbar:** Includes icons for back, forward, search, and various debugger functions.
- Text Area:** Displays the memory dump of an object. The title bar says "0 -[AppDelegate applicationDidFinis...".
- Object Summary:** Shows 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
- Bottom Bar:** Includes "Auto" dropdown, visibility controls, and other debugger settings.

# “Data”

## Life with formatters

The screenshot shows the Xcode Data Inspector interface. The top bar has various icons for navigating between files and symbols, and the title "0 -[AppDelegate applicationDidFinis...]".

The main area displays four variables:

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

At the bottom, there are buttons for "Auto" and "Custom" modes, and a search bar.

# 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

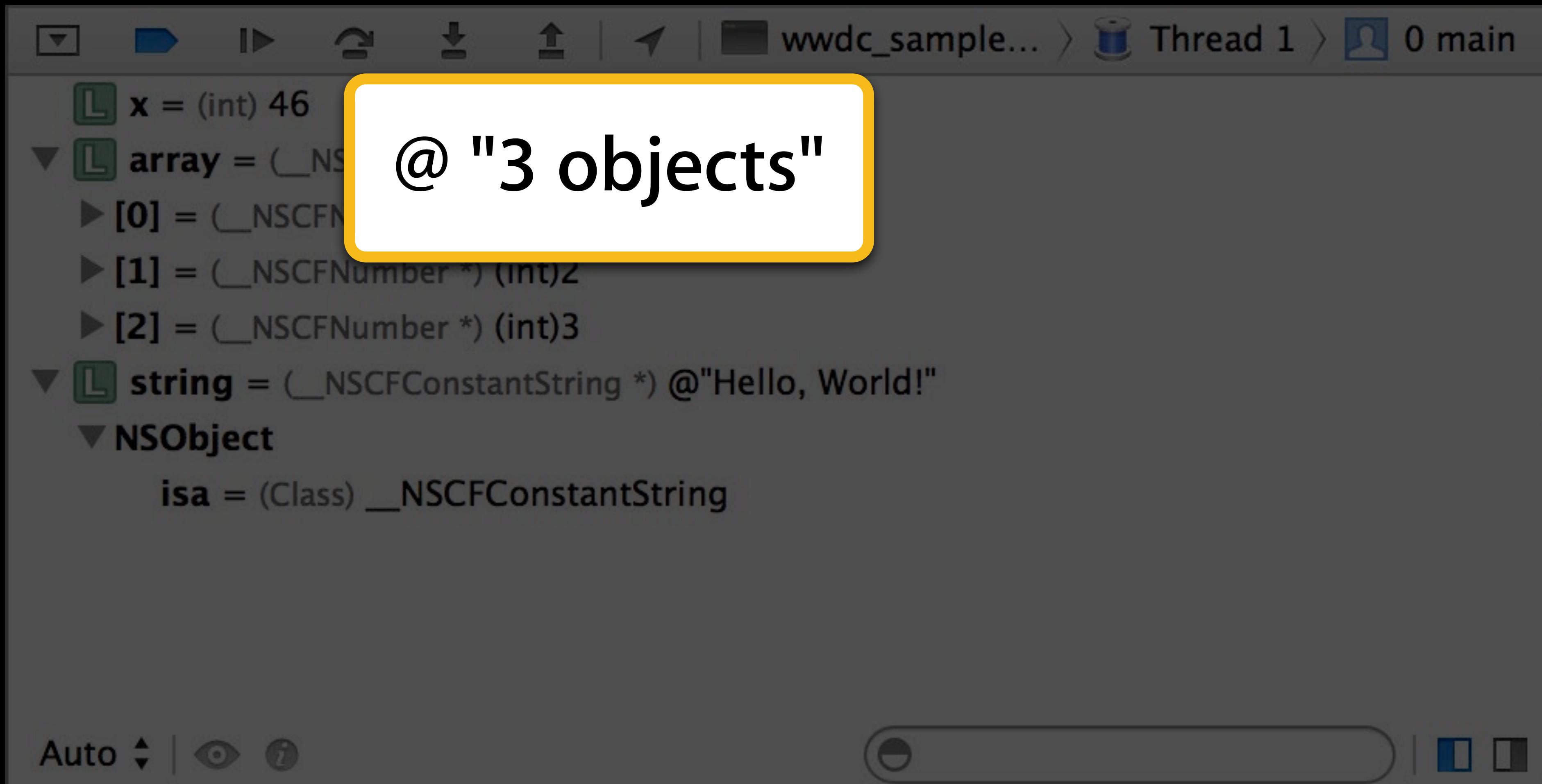
# Summaries

The screenshot shows the Xcode Debug Navigator with the following variable summaries:

- 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 are buttons for "Auto", "Reset", and "Copy".

# Summaries



```
x = (int) 46
array = (@NSCFNumber *) 0x0000000100000000
 [0] = (@NSCFNumber *) 0x0000000100000001
 [1] = (@NSCFNumber *) 0x0000000100000002
 [2] = (@NSCFNumber *) 0x0000000100000003
string = (@NSCFConstantString *) @"Hello, World!"
isa = (Class) __NSCFConstantString
```

@ "3 objects"

# Synthetic Children

The screenshot shows the Xcode interface with the Debug Navigator open. The title bar indicates the project is "wwdc\_sample..." and the thread is "Thread 1". The main pane displays the memory dump for variable "array".

**x = (int) 46**

**array = (`__NSArrayI` \*) @"3 objects"**

- [0] = (`__NSCFNumber` \*) (int)1**
- [1] = (`__NSCFNumber` \*) (int)2**
- [2] = (`__NSCFNumber` \*) (int)3**

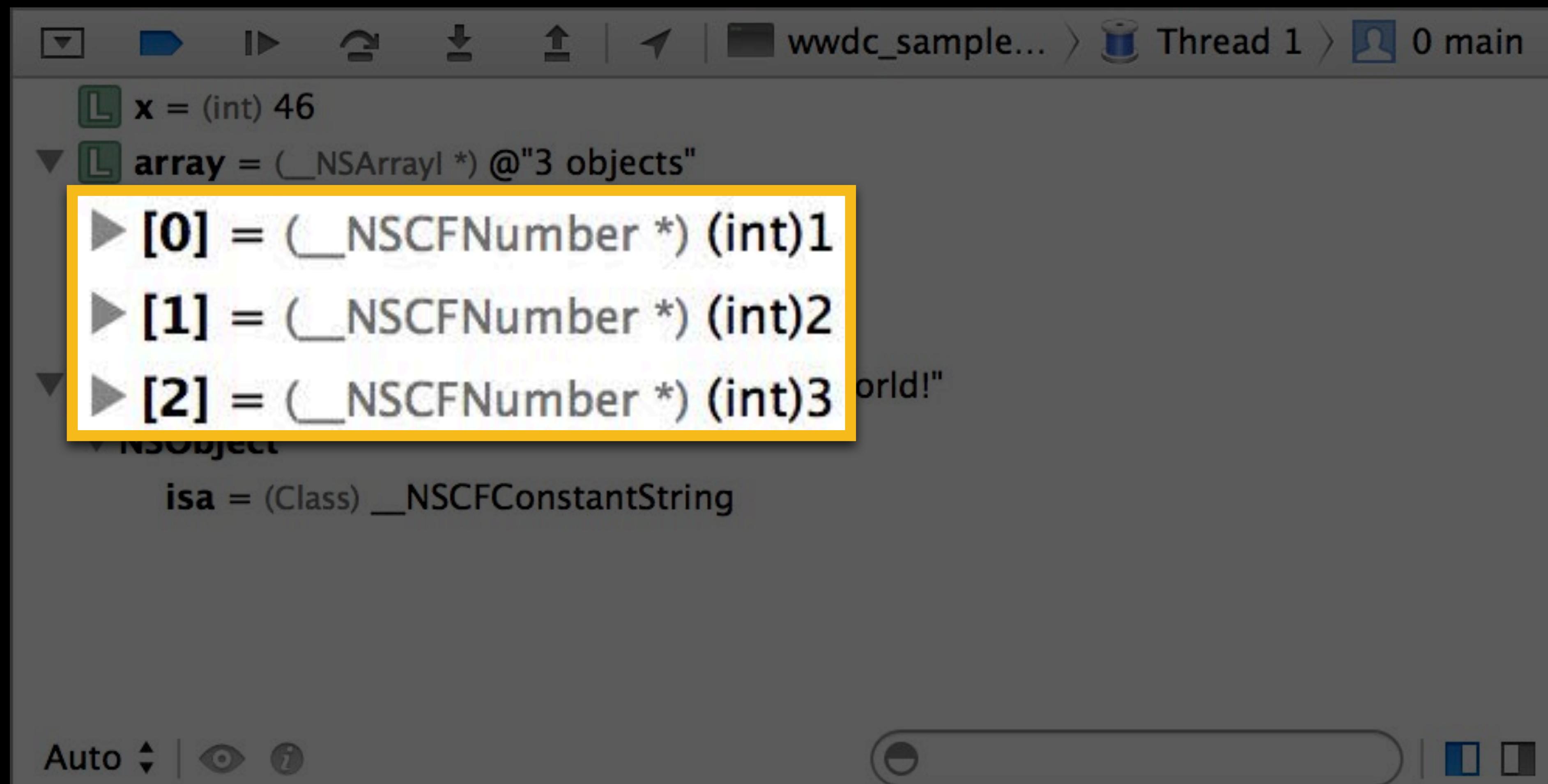
**string = (`__NSCFConstantString` \*) @"Hello, World!"**

**NSObject**

- isa = (Class) `__NSCFConstantString`**

At the bottom left, there are buttons for "Auto" (with a dropdown arrow), "View" (eye icon), and "Info" (info icon). At the bottom right, there are buttons for "Stop" (red circle), "Step Into" (blue square), and "Step Out" (white square).

# Synthetic Children



The screenshot shows the Xcode debugger interface with the title bar "wwdc\_sample..." and "Thread 1". The main pane displays the variable "array" as an NSArray containing three objects. The elements [0], [1], and [2] are highlighted with a yellow box.

```
x = (int) 46
array = (@"3 objects"
 ▶ [0] = (_NSCFNumber *) (int)1
 ▶ [1] = (_NSCFNumber *) (int)2
 ▶ [2] = (_NSCFNumber *) (int)3
isa = (Class) __NSCFConstantString
```

# 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.llvm.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 with the title bar "wwdc\_sample..." and "Thread 1". The main pane displays the variable hierarchy:

- 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 are buttons for "Auto" (with dropdown arrow), "Stop" (eye icon), "Info" (i icon), and "Run" (blue square icon). There is also a search bar.

# SBValue

The screenshot shows the Xcode debugger interface with the title bar "name" highlighted by a blue callout. The title bar also displays the project name "wwdc\_sample...", the thread "Thread 1", and the current stack frame "0 main".

The variable hierarchy is as follows:

- 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 are buttons for "Auto" mode, breakpoints, and other debugger controls.

# SBValue

The screenshot shows the Xcode debugger interface with the title bar "wwdc\_sample..." and "Thread 1". The main pane displays the following SBValue hierarchy:

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

At the bottom, there are buttons for "Auto" (with a dropdown arrow), "Breakpoint" (eye icon), "Info" (info icon), and "Run" (play/pause icon).

# SBValue

The screenshot shows the Xcode debugger interface with the title bar "wwdc\_sample..." and "Thread 1". The main pane displays the following SBValue hierarchy:

- 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 speech bubble icon with the word "summary" is positioned near the "string" entry.

At the bottom, there are buttons for "Auto" (with dropdown arrow), "Breakpoint" (eye icon), "Info" (i icon), and "Run" (play icon). There is also a search bar and a zoom control.

# SBValue

The screenshot shows the Xcode debugger interface with the title bar "wwdc\_sample..." > "Thread 1" > "0 main". The main pane displays the following SBValue hierarchy:

- 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 points from the text "children" to the array's children list.

At the bottom, there are buttons for "Auto" (with a dropdown arrow), "Stop" (eye icon), "Info" (i icon), and a search bar.

# SBValue

The screenshot shows the Xcode debugger interface with the title bar "value" highlighted by a blue callout. The title bar also displays the project name "wwdc\_sample...", the thread "Thread 1", and the current stack frame "0 main".

The variable hierarchy is as follows:

- 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 are buttons for "Auto" (with a dropdown arrow), "Stop" (eye icon), "Info" (i icon), and "Break" (circle icon). There are also buttons for "Next" (right arrow), "Step Into" (blue right arrow), and "Step Out" (double right arrow).

# Example

## Summarizing an Address

# Example

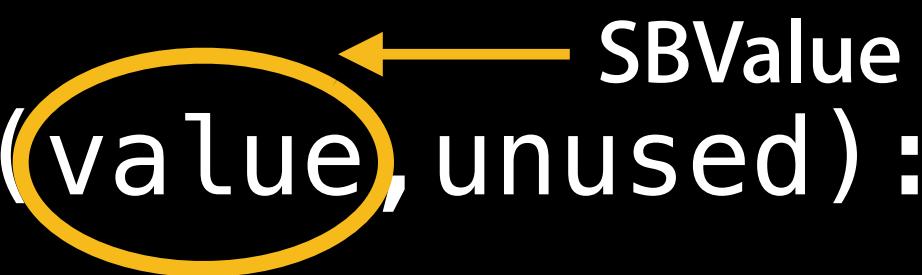
## Summarizing an Address

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

# Example

## Summarizing an Address

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



# Example

## Summarizing an Address

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

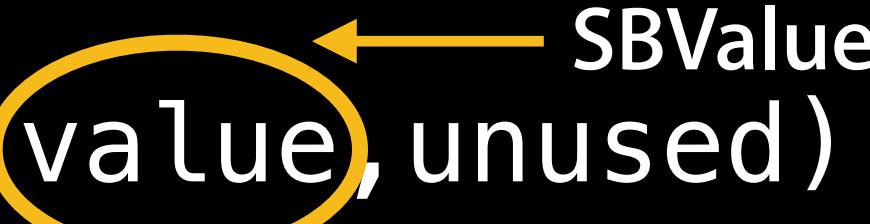


The code snippet demonstrates how to summarize an address by extracting the first name and last name from a given value. The variable 'value' is highlighted with a yellow oval and an annotation 'SBValue' with an arrow pointing to it, indicating its type or context in the code.

# Example

## Summarizing an Address

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



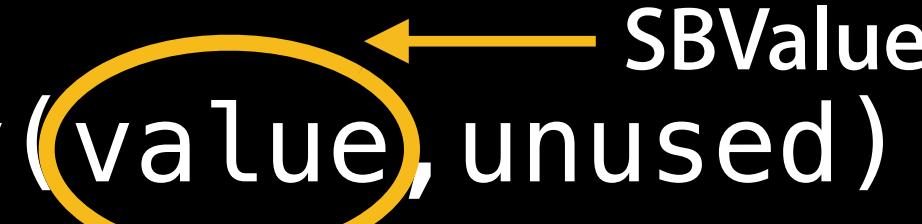
The code snippet demonstrates how to summarize an address by extracting first and last names. It uses the `GetChildMemberWithName` method to find the child members for 'firstName' and 'lastName', and then calls `GetSummary` on each to get a summary of the name.

# 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

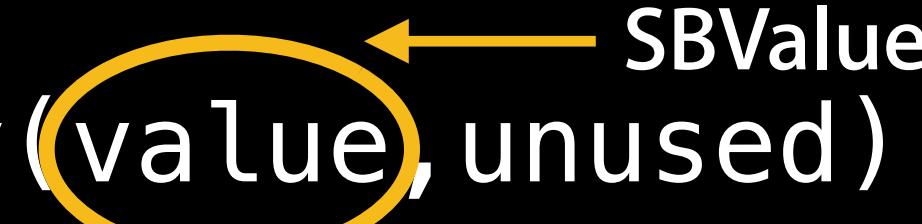
# Example

## Summarizing an Address

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

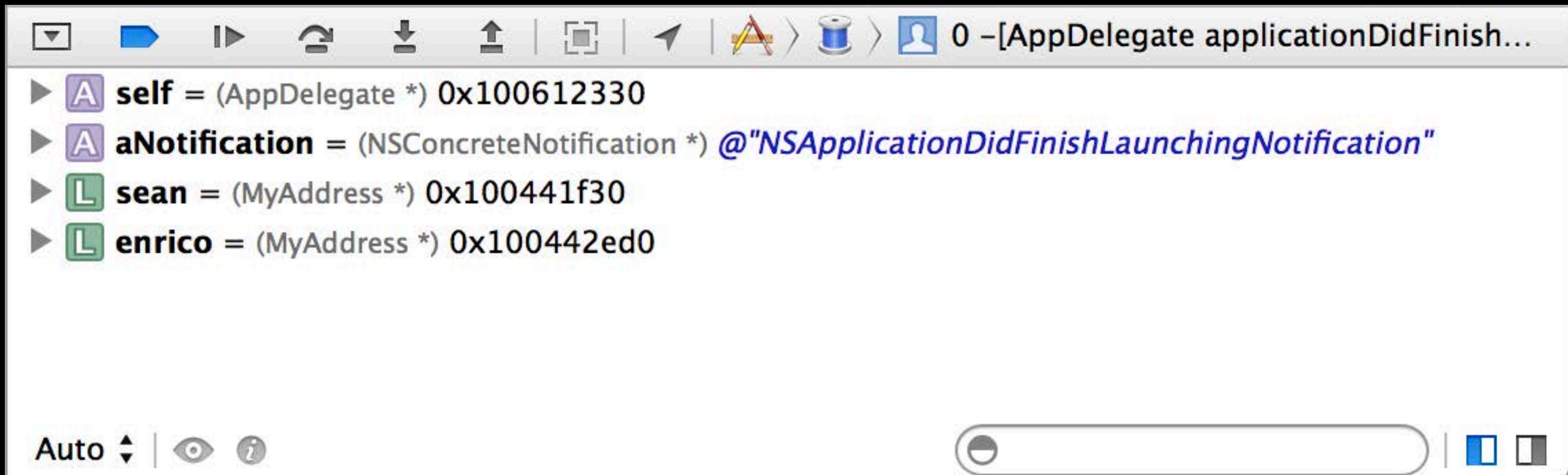
 # process the data as you wish

 return firstNameSummary + " " + lastNameSummary
```



# Example

## Summarizing an Address

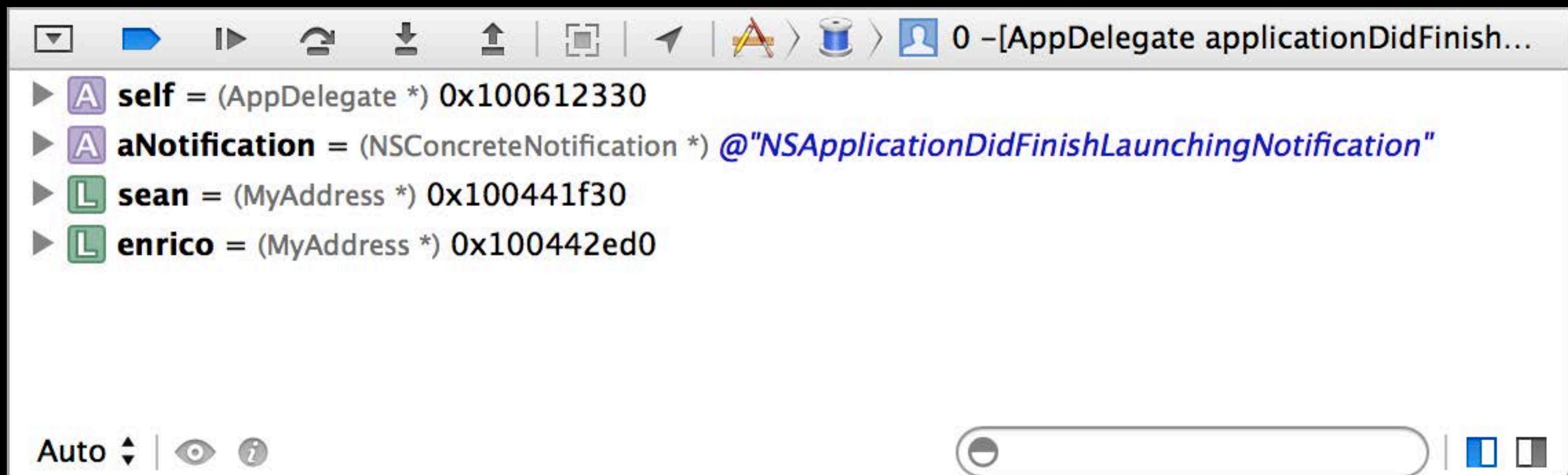


The screenshot shows the Xcode debugger interface with the following details:

- Toolbar:** Includes standard navigation icons (back, forward, search, etc.) and a target selector.
- Text Area:** Displays variable summaries:
  - self** = (AppDelegate \*) 0x100612330
  - aNotification** = (NSConcreteNotification \*) @"NSApplicationDidFinishLaunchingNotification"
  - sean** = (MyAddress \*) 0x100441f30
  - enrico** = (MyAddress \*) 0x100442ed0
- Bottom Controls:** Includes "Auto" dropdown, visibility toggles, and other debugger controls.

# Example

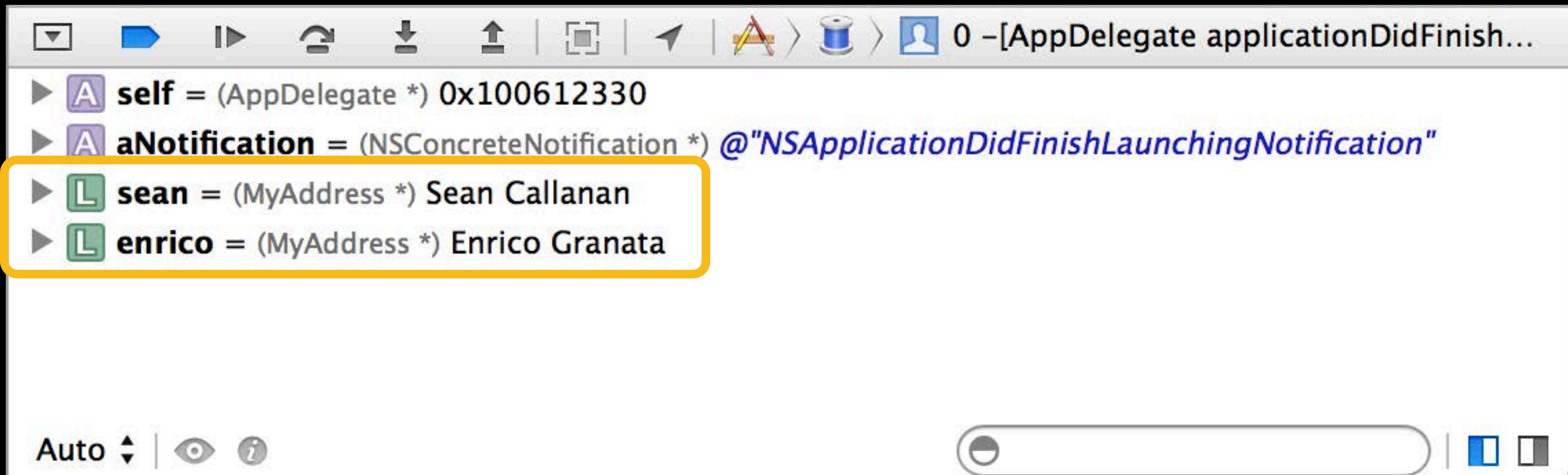
## Summarizing an Address



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

# Example

## Summarizing an Address



The screenshot shows the Xcode debugger interface with the following details:

- Toolbar:** Includes standard navigation icons (back, forward, search, etc.) and a target selector.
- Text Area:** Displays variable summaries:
  - self = (AppDelegate \*) 0x100612330** (purple A icon)
  - aNotification = (NSConcreteNotification \*) @"NSApplicationDidFinishLaunchingNotification"** (purple A icon)
  - sean = (MyAddress \*) Sean Callanan** (green L icon)
  - enrico = (MyAddress \*) Enrico Granata** (green L icon)
- Bottom Controls:** Includes "Auto" dropdown, visibility toggles, and other debugger controls.

A yellow box highlights the two variable summaries for **sean** and **enrico**.

```
0 -[AppDelegate applicationDidFinish...]
▶ A self = (AppDelegate *) 0x100612330
▶ A aNotification = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
▶ L sean = (MyAddress *) Sean Callanan
▶ L enrico = (MyAddress *) Enrico Granata
```

# 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

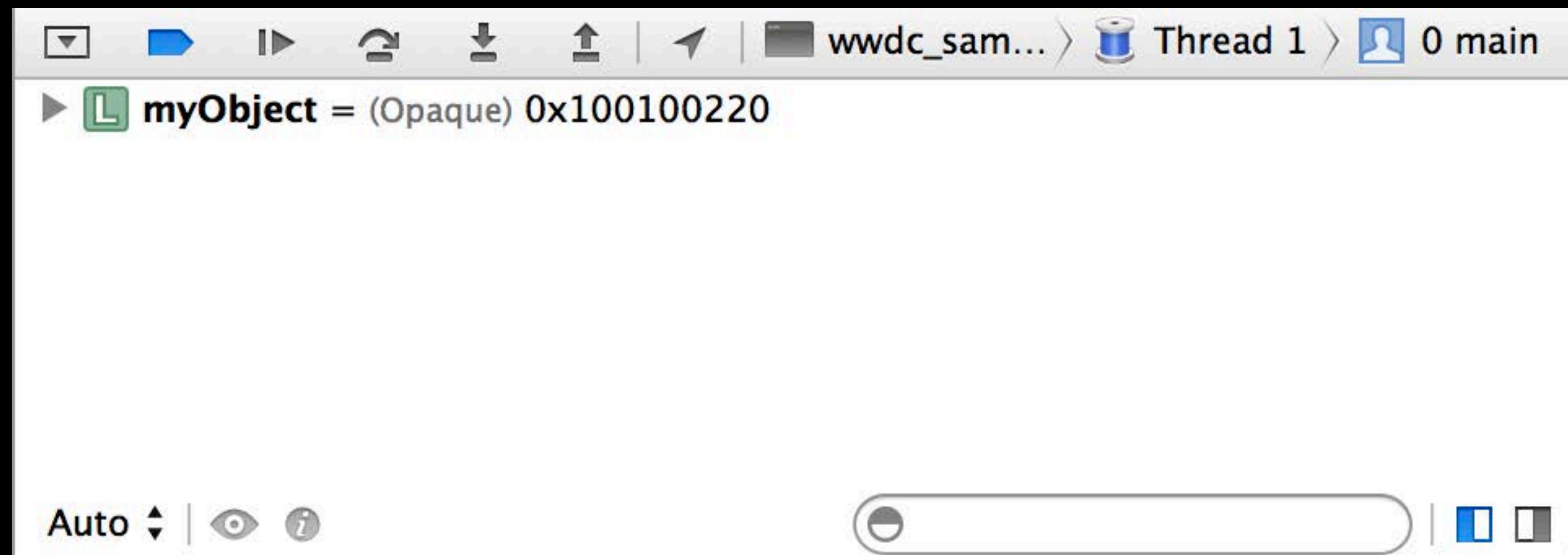
# expression for Data Analysis

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

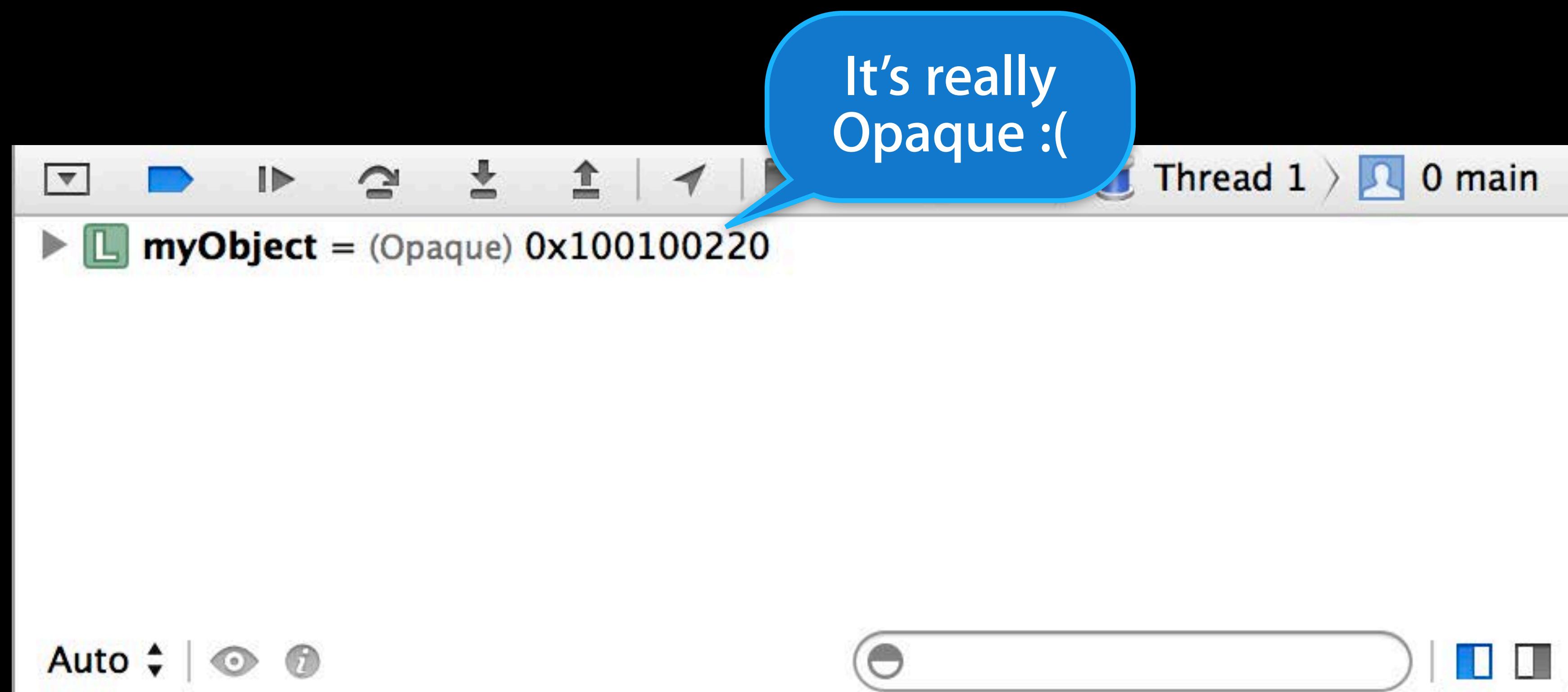
1 struct ImplOpaque {
2 int aThing;
3 float anotherThing;
4 char* oneMoreThing;
5 };
```

Opaque.cpp

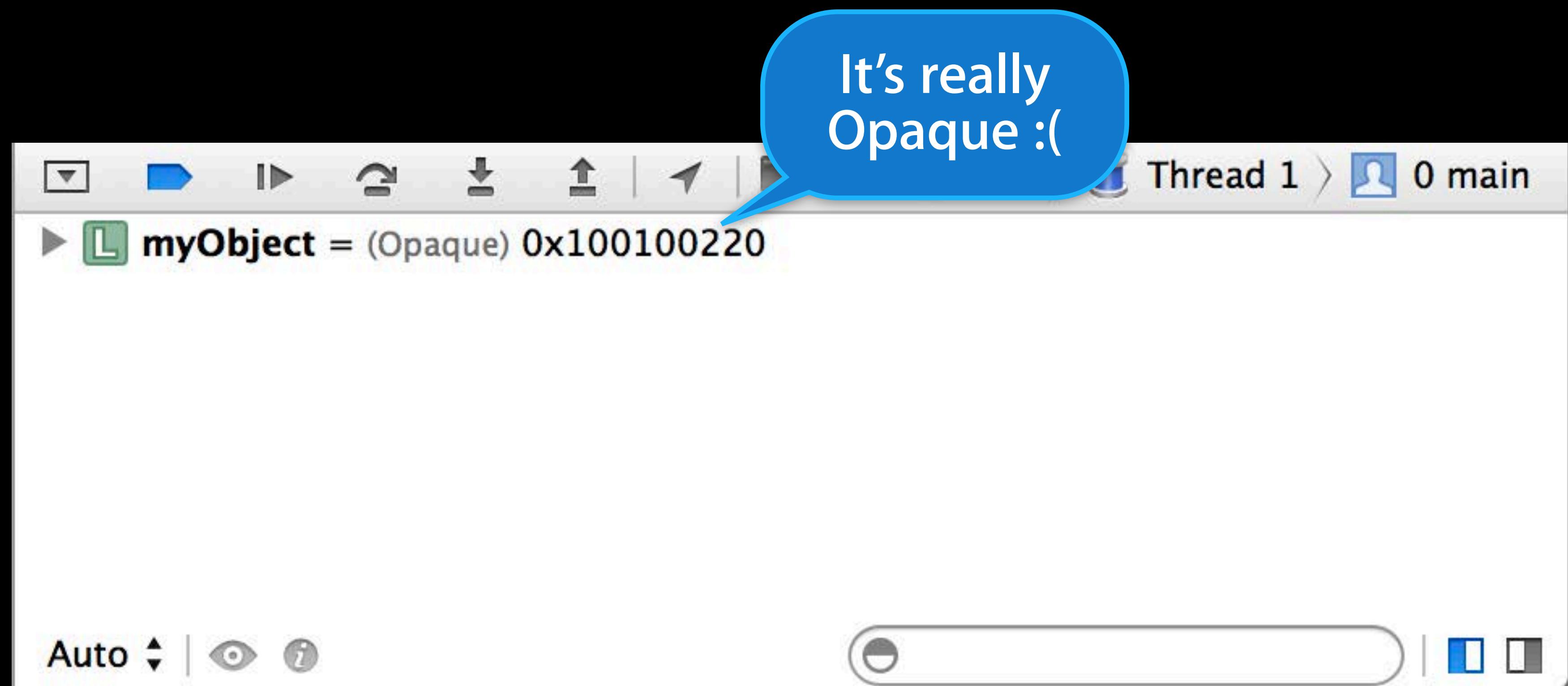
# expression for Data Analysis



# expression for Data Analysis



# expression for Data Analysis



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

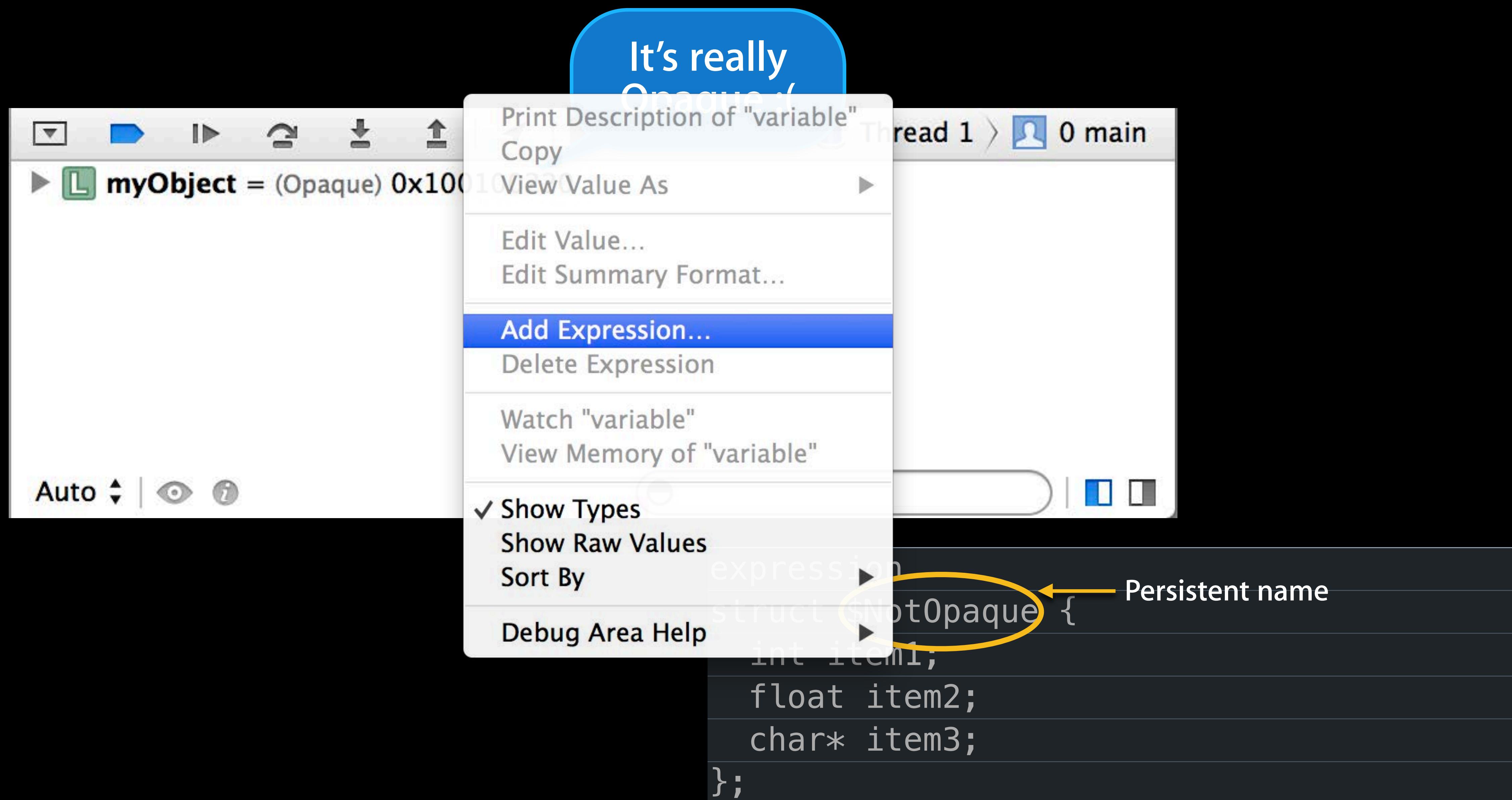
# expression for Data Analysis

The screenshot shows the Xcode debugger interface. In the top-left corner of the main window, there is a blue speech bubble containing the text "It's really Opaque :(". Below this, the debugger's status bar shows "Thread 1" and "0 main". The main pane displays the expression `myObject = (Opaque) 0x100100220`. At the bottom of the screen, a code editor window is open, showing the following C-like code:

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

A yellow oval highlights the identifier `$NotOpaque`, and a yellow arrow points from the word "Persistent name" to this highlighted identifier.

# expression for Data Analysis



# expression for Data Analysis

The screenshot shows the Xcode debugger interface. At the top, there is a toolbar with various icons. Below it, the thread list shows "Thread 1 > 0 main". In the main pane, a variable is listed: `myObject = (Opaque) 0x100100220`. A blue speech bubble points to this line with the text "It's really Opaque :(

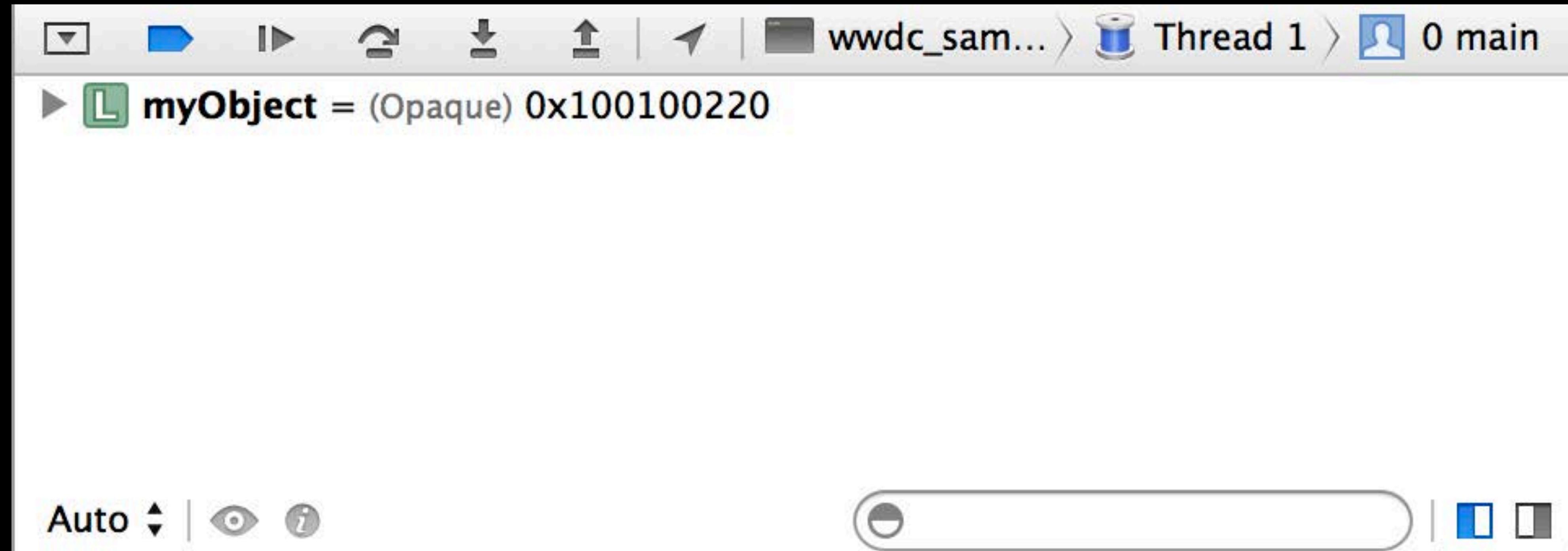
A yellow box highlights the expression input field, which contains `($NotOpaque*)myObject`. There is also a checkbox labeled "Show in All Stack Frames" and a "Done" button.

Below the debugger, a code editor window displays the following C-like code:

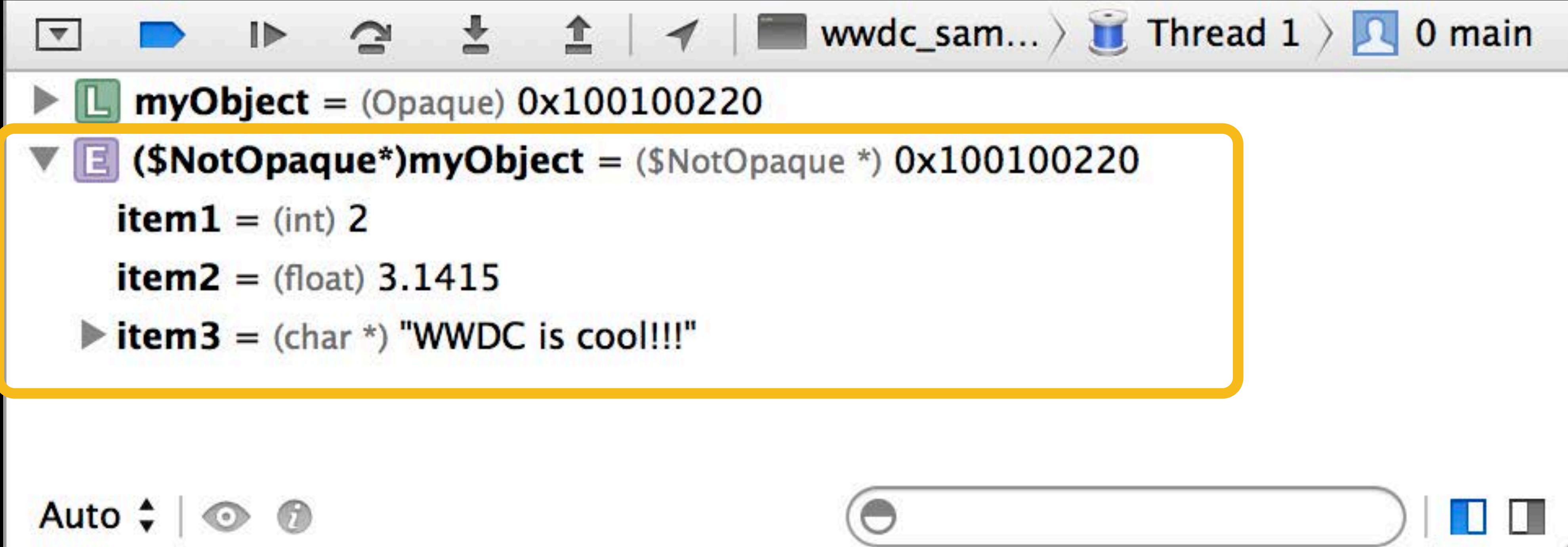
```
expression
struct $NotOpaque {
 int item1;
 float item2;
 char* item3;
};
```

The identifier `$NotOpaque` is circled in yellow, and an arrow points from the text "Persistent name" to this circle.

# expression for Data Analysis



# expression for Data Analysis



The screenshot shows the Xcode interface with the "Debug Navigator" open. The title bar indicates the project is "wwdc\_sam..." and the thread is "Thread 1". The main area displays a memory dump for the variable `myObject`. A yellow box highlights the dump for `myObject`, which is of type `(Opaque)` and has the value `0x100100220`. Below this, the dump for `($NotOpaque*)myObject` is shown, also with the value `0x100100220`. This dump reveals three pointers: `item1` (int) pointing to `2`, `item2` (float) pointing to `3.1415`, and `item3` (char \*) pointing to the string `"WWDC is cool!!!"`. The bottom of the window shows standard Xcode navigation and search controls.

```
myObject = (Opaque) 0x100100220
($NotOpaque*)myObject = ($NotOpaque *) 0x100100220
 item1 = (int) 2
 item2 = (float) 3.1415
 item3 = (char *) "WWDC is cool!!!"
```

# 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

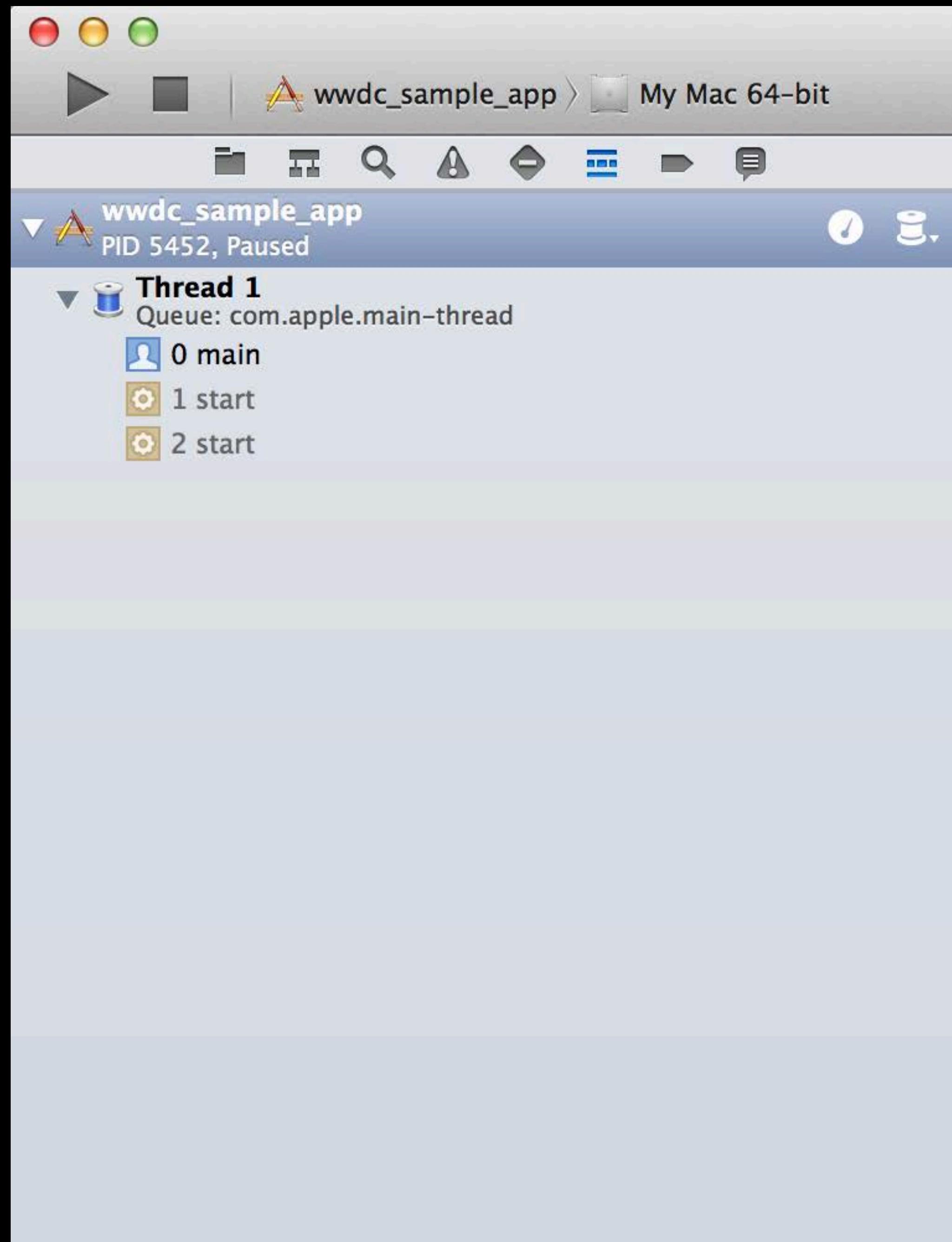
# The LLDB Object Model

- 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

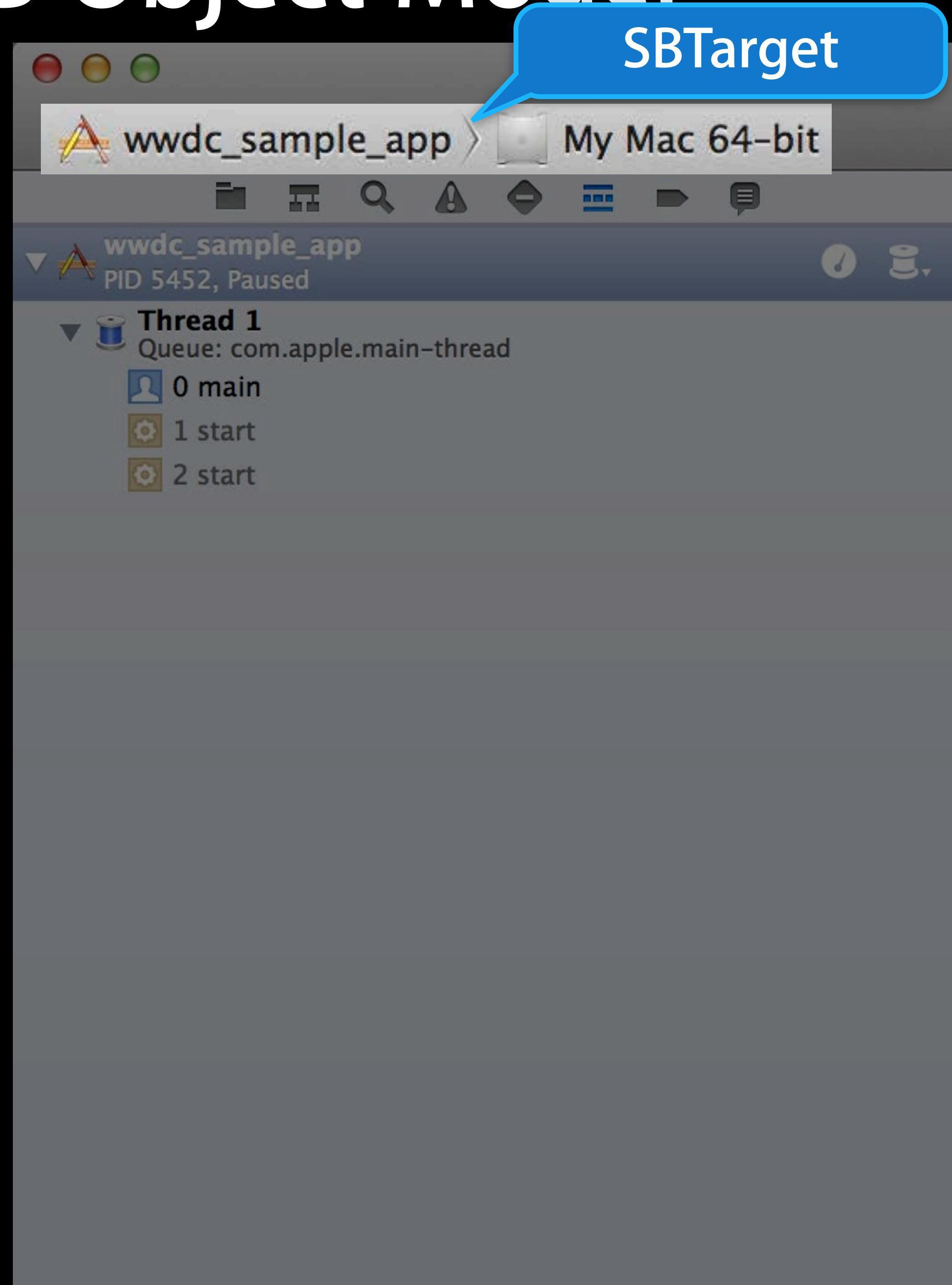
# The LLDB Object Model



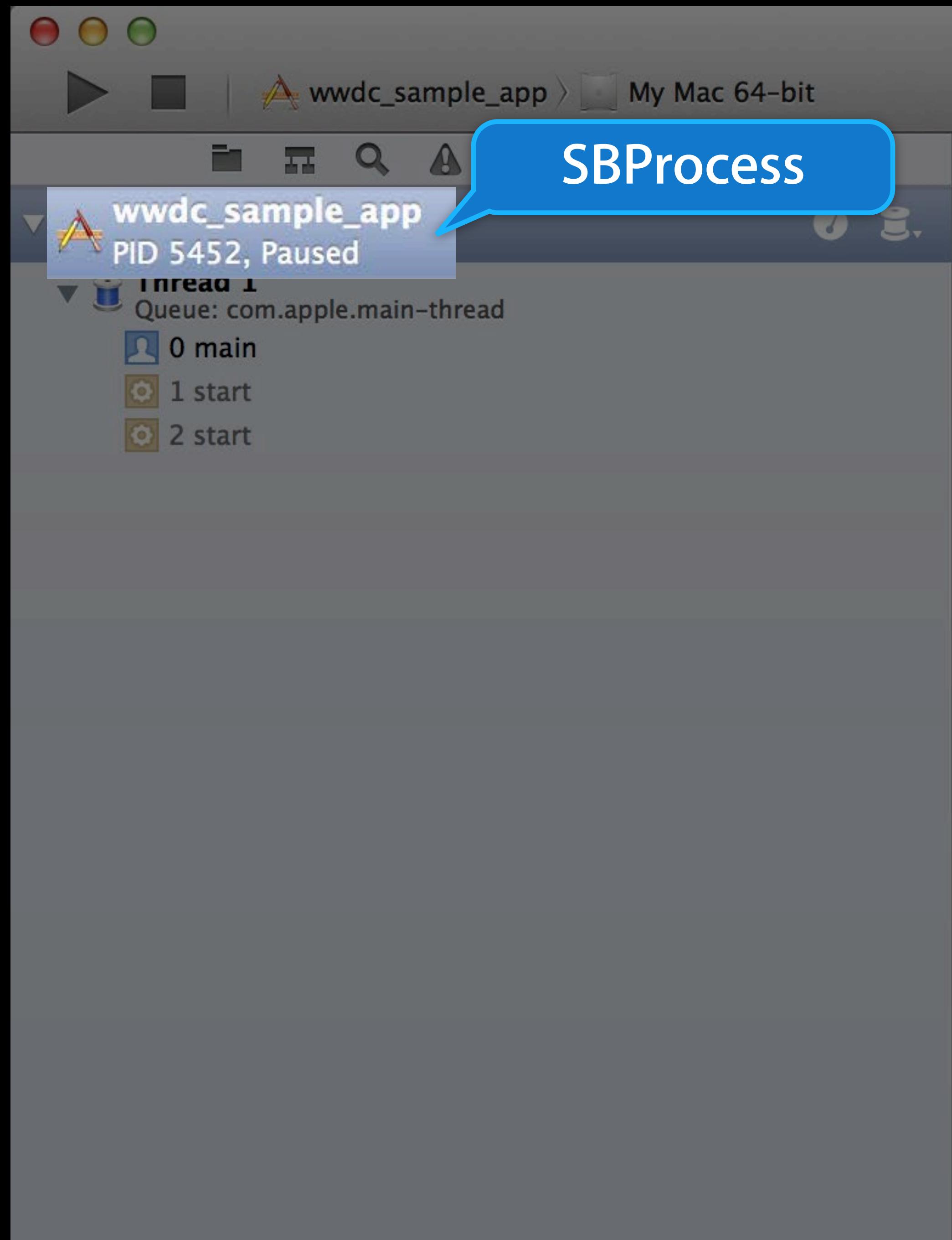
The screenshot shows the Xcode debugger interface with the LLDB Object Model open. The left pane displays the target "wwdc\_sample\_app" (PID 5452, Paused) and its threads, including "Thread 1" (Queue: com.apple.main-thread) which has frames for "main", "start", and "start". The right pane shows the source code of the main.m file, which contains the following code:

```
1 // main.m
2 // wwdc_sample_app
3 // Create
4 // Copyri
5 //
6 //
7 //
8 //
9 #import <
10
11 int main()
12 {
13 return 0;
}
```

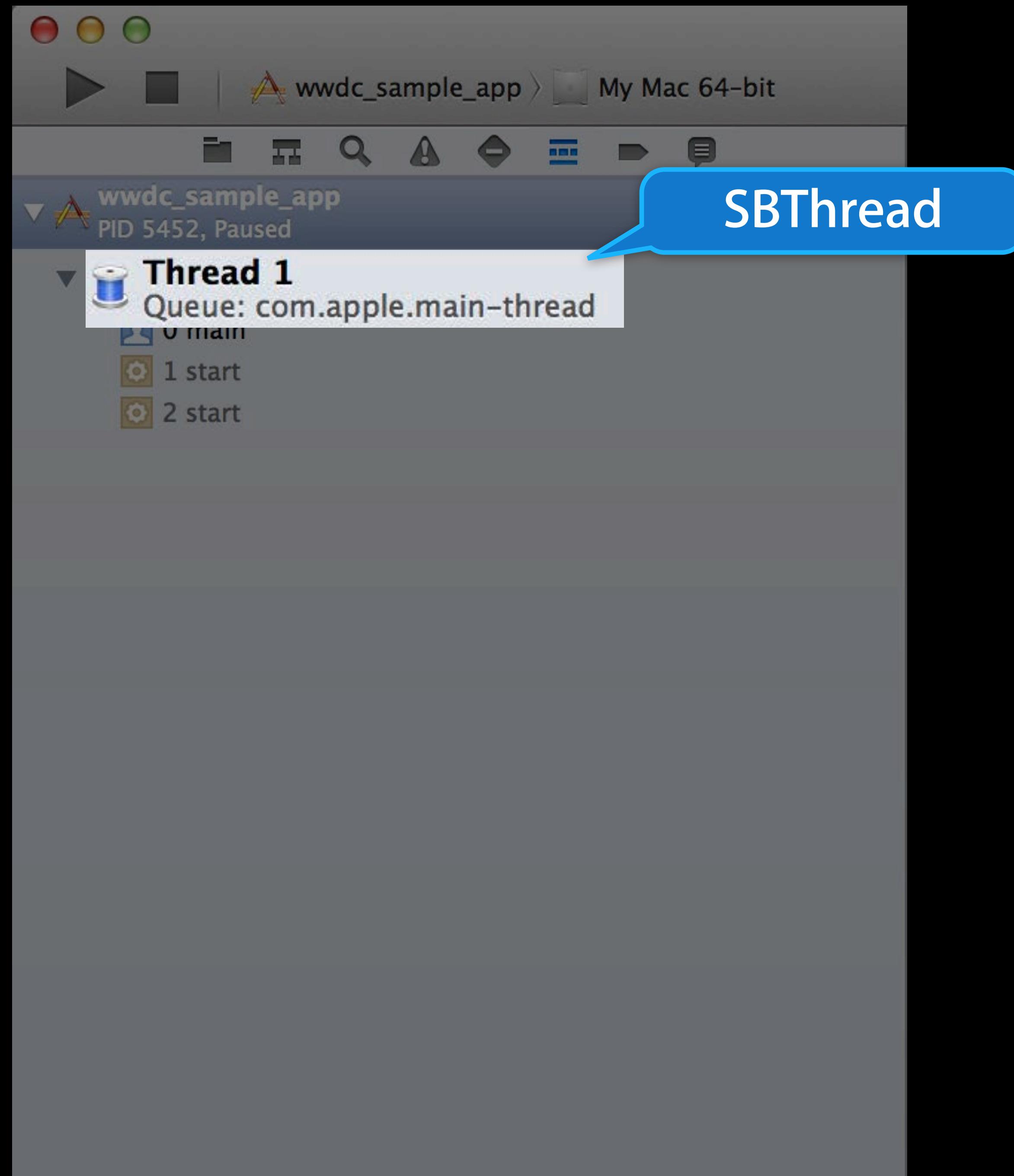
# The LLDB Object Model



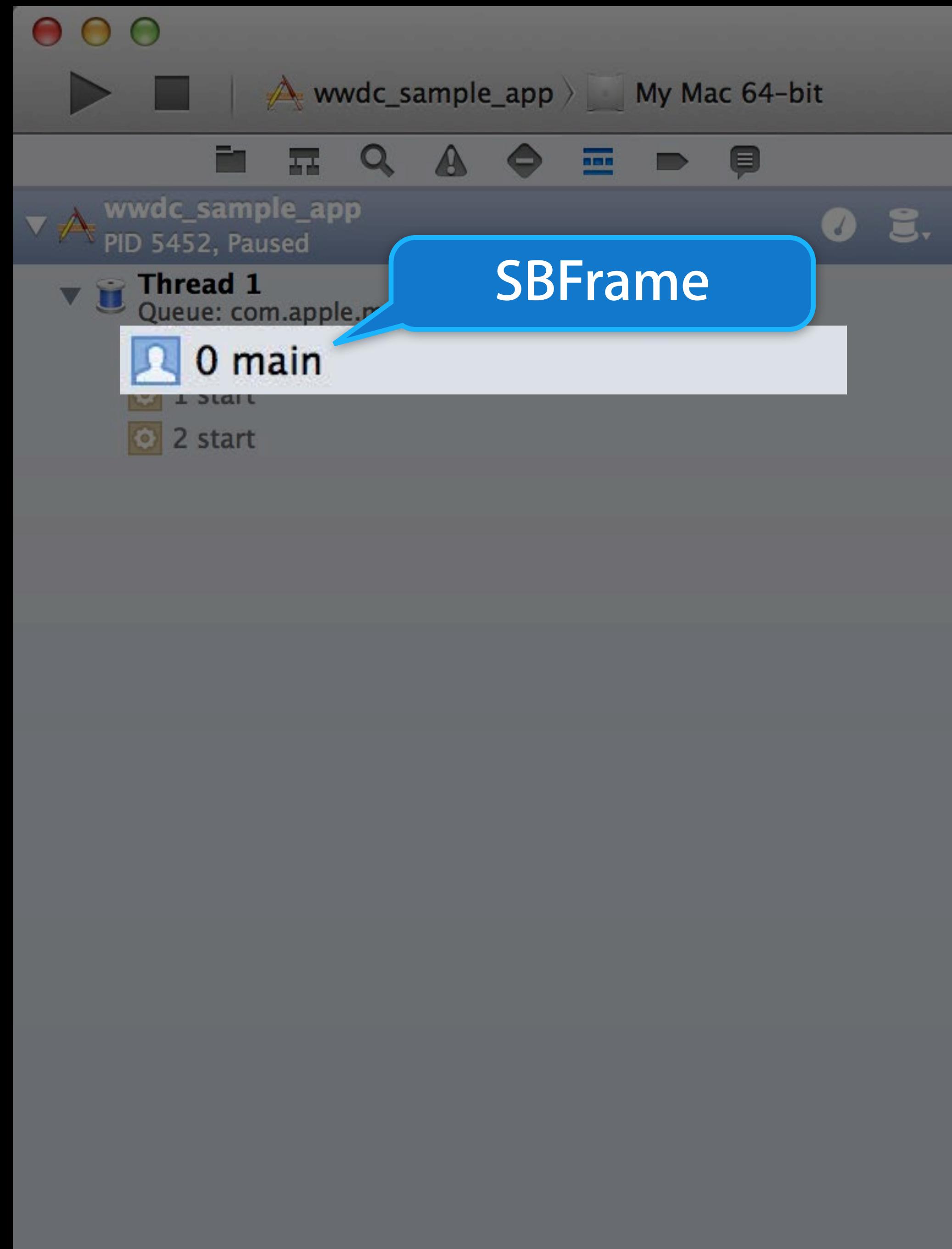
# The LLDB Object Model



# 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

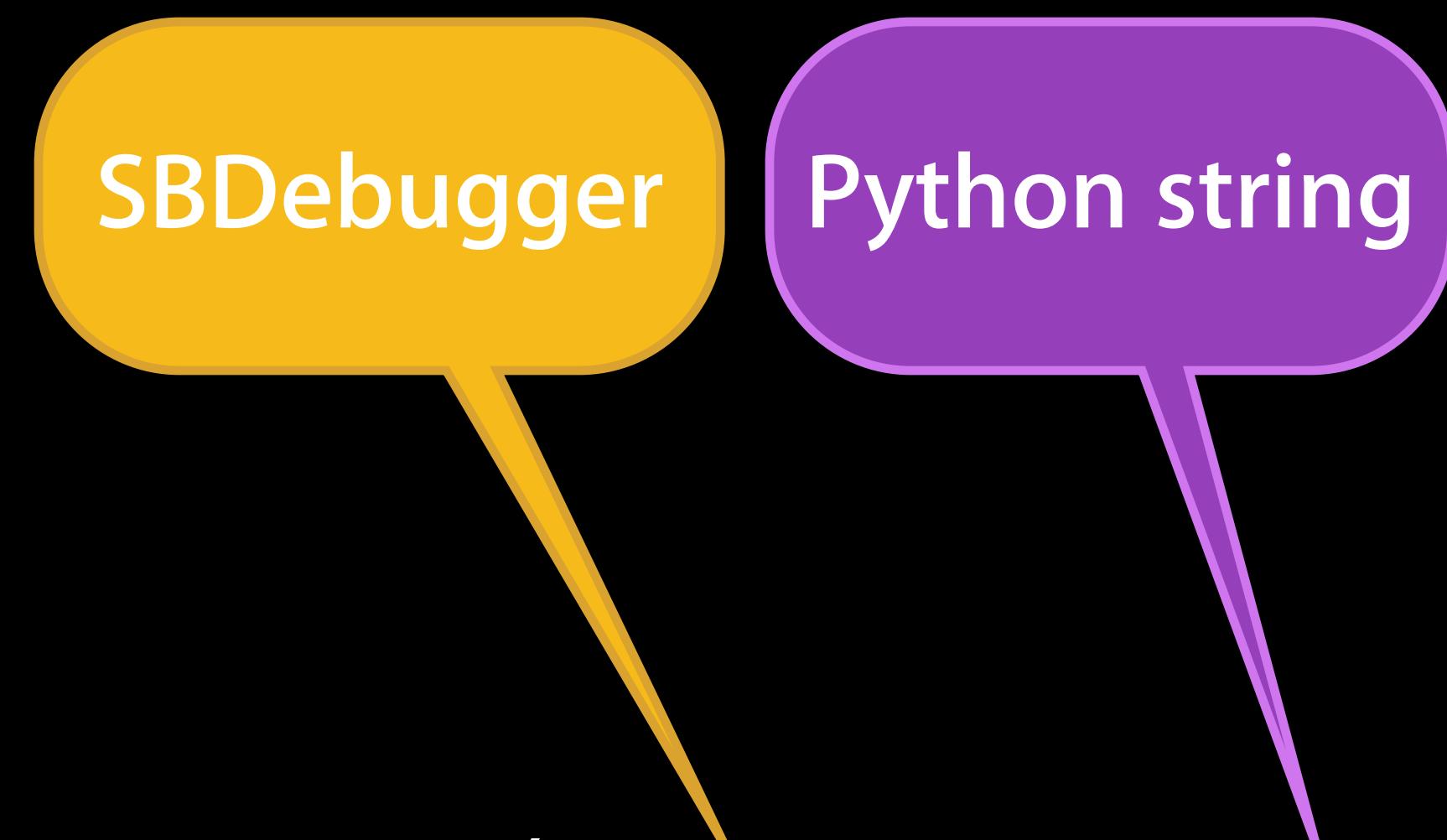


SBDebugger

```
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



```
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

The diagram illustrates the components of a Python command definition. Three colored callouts point to specific parts of the code:

- A yellow callout labeled "SBDebugger" points to the variable `debugger` in the command implementation.
- A purple callout labeled "Python string" points to the variable `user_input`.
- A teal callout labeled "SBCommandReturnObject" points to the variable `result`.

```
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

The diagram illustrates the components of a Python command definition. Three colored callouts point to specific parts of the code:

- A yellow callout labeled "SBDebugger" points to the variable `debugger`.
- A purple callout labeled "Python string" points to the variable `user_input`.
- A teal callout labeled "SBCommandReturnObject" points to the variable `result`.

```
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() \\\n .GetProcess().GetSelectedThread()
```

Loop over  
all frames

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

Check for  
recursion

Display counter

# Example

## Calculate depth of a recursion

Utilize LLDB  
Object Model

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

Loop over  
all frames

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

Check for  
recursion

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

Display counter

# Example

## Calculate depth of a recursion

Utilize LLDB  
Object Model

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

Loop over  
all frames

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

Check for  
recursion

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

*Display counter*

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

# Example

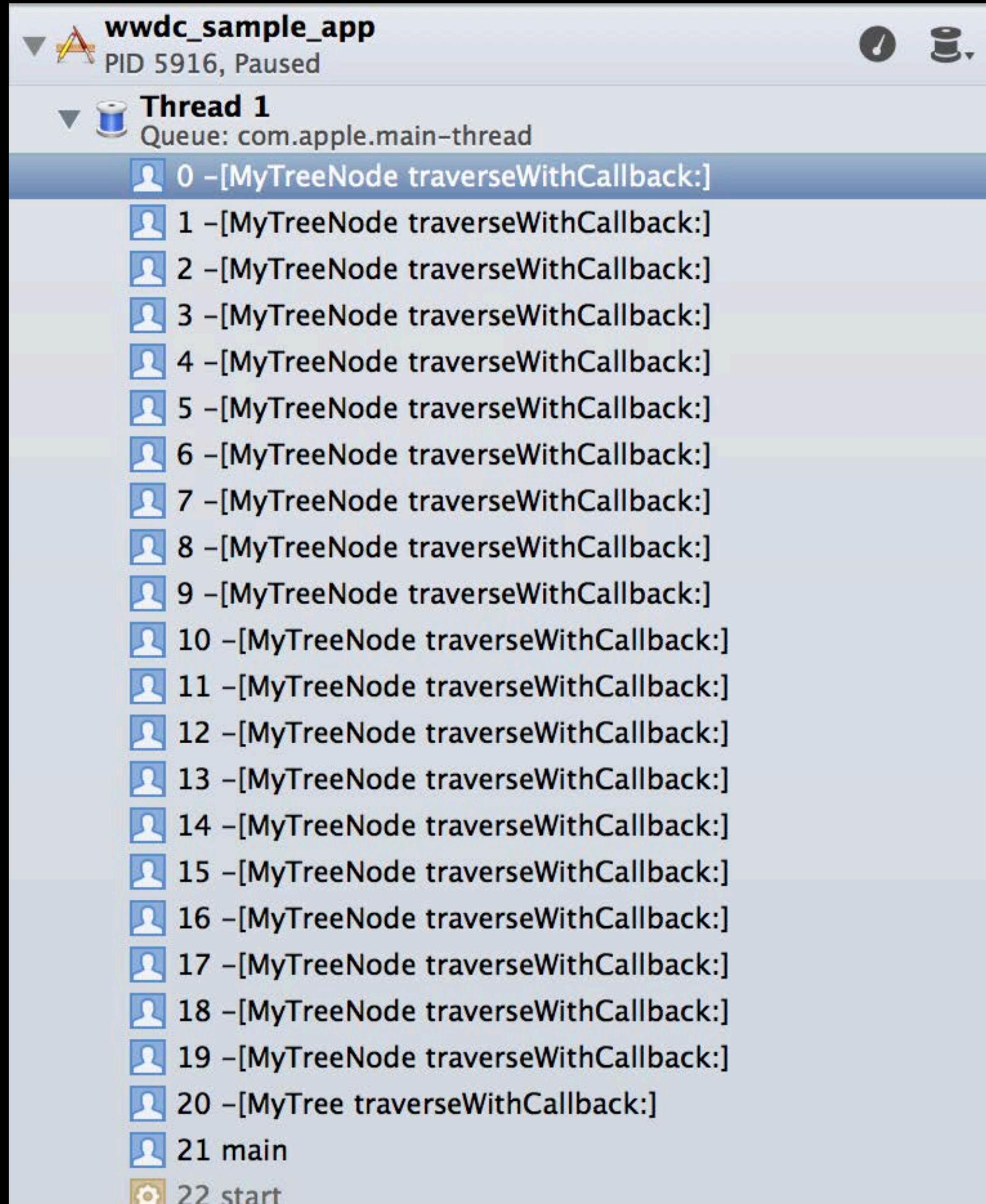
## Calculate depth of a recursion

```
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

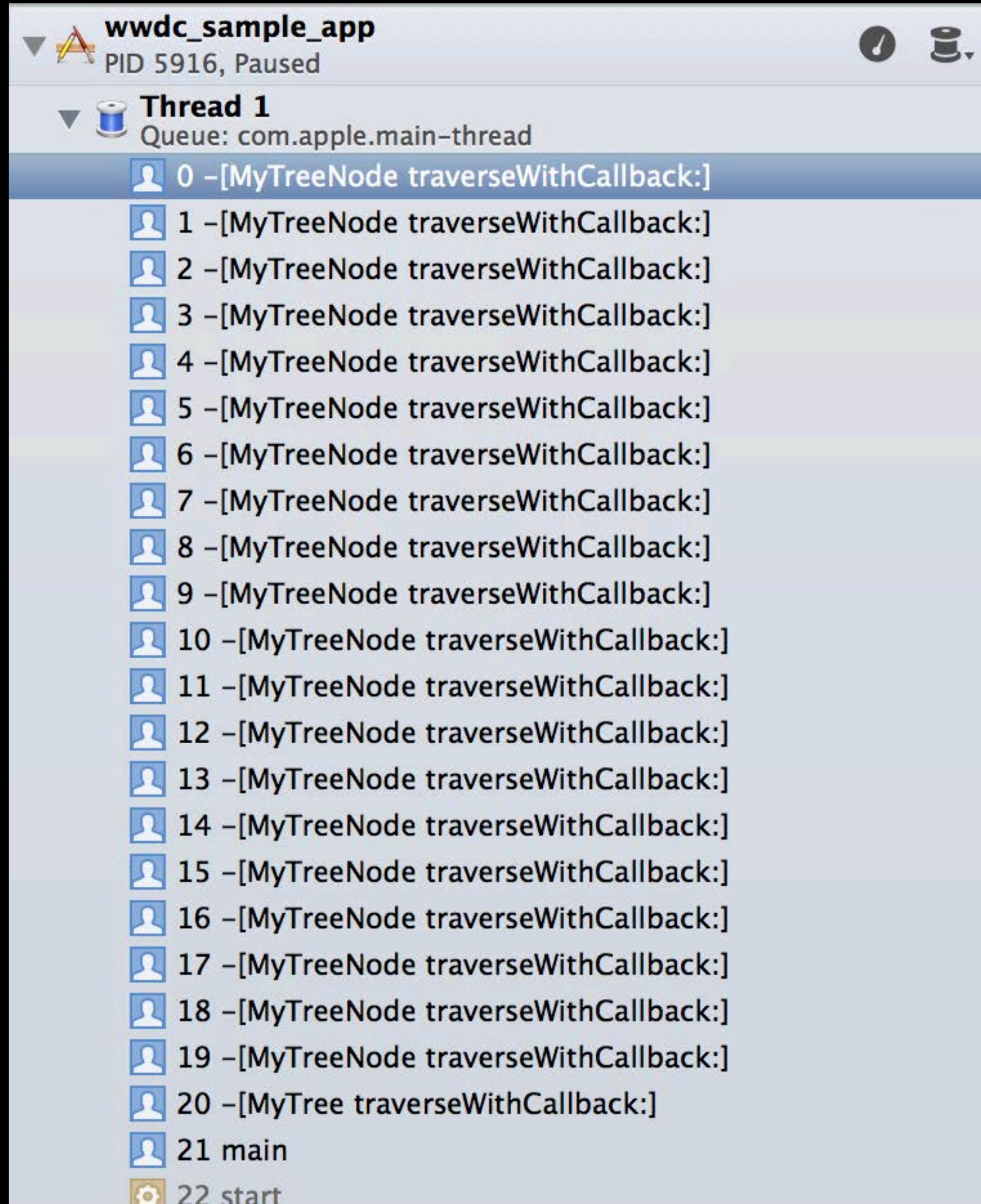


(lldb)

All Output ▾

# Example

## Calculate depth of a recursion



(lldb) **depth**

All Output ▾

# Example

## Calculate depth of a recursion

The screenshot shows the Xcode interface during a debug session. On the left, the Debug Navigator displays a call stack for Thread 1 on the main queue. The stack consists of 22 entries, all of which are calls to the same method, `-[MyTreeNode traverseWithCallback:]`. The entries are numbered from 0 to 21, with entry 0 at the top and entry 21 at the bottom. The entry for entry 21 is highlighted with a blue bar. On the right, the LLDB console window shows the command `(lldb) depth` followed by the output `depth: 20`, which is highlighted with an orange rectangle. Below the LLDB window, the text "All Output" is visible.

```
(lldb) depth
depth: 20
(lldb)
```

All Output ▾

# 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

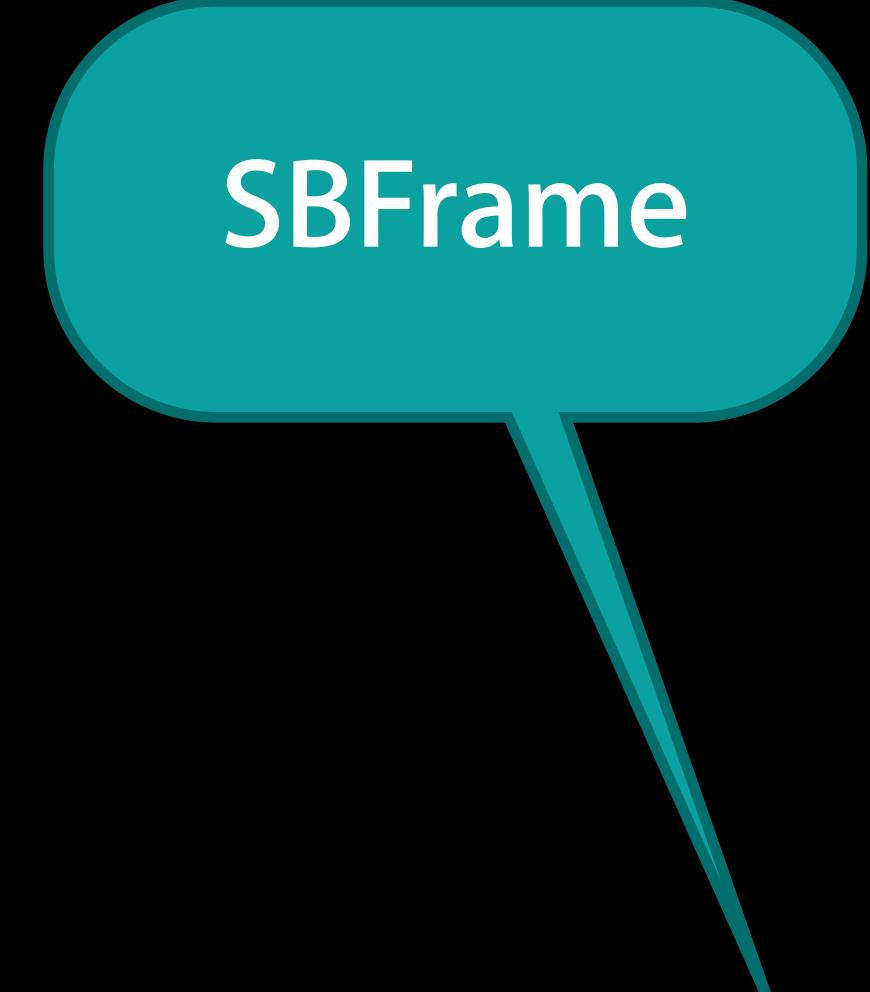
# How Breakpoint Actions Work

- 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

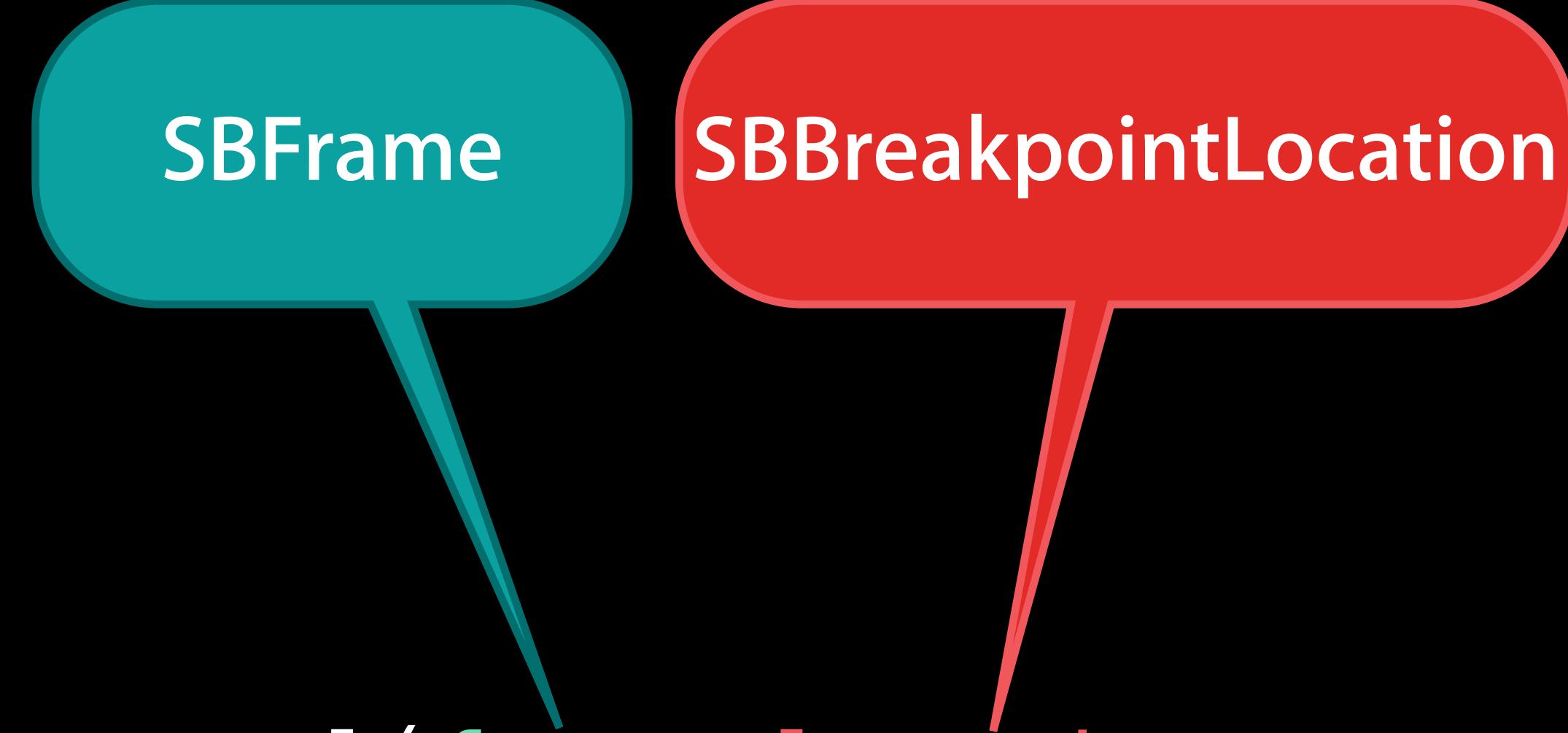


SBFrame

```
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 >= threshold*

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

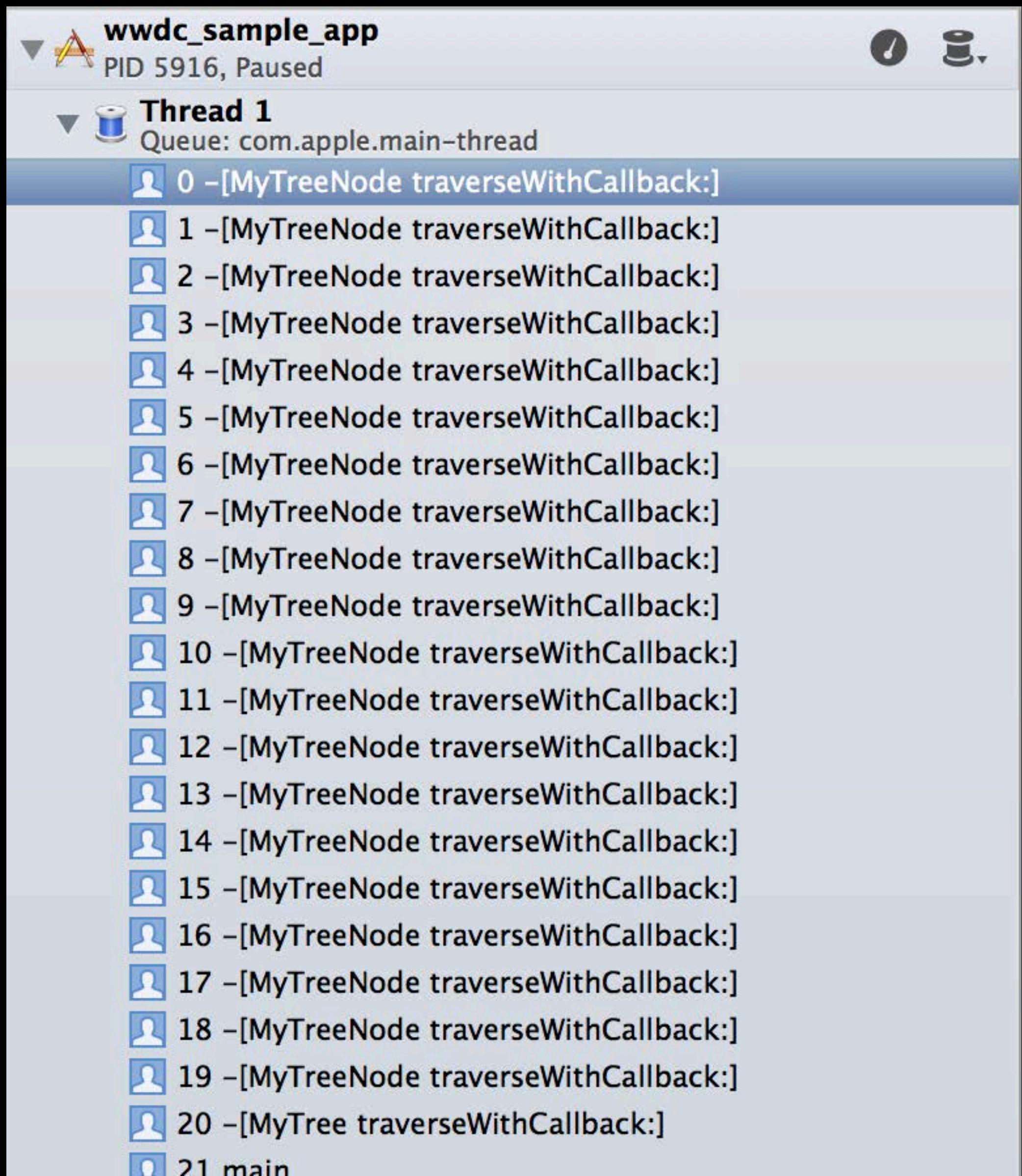
# Example

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
```

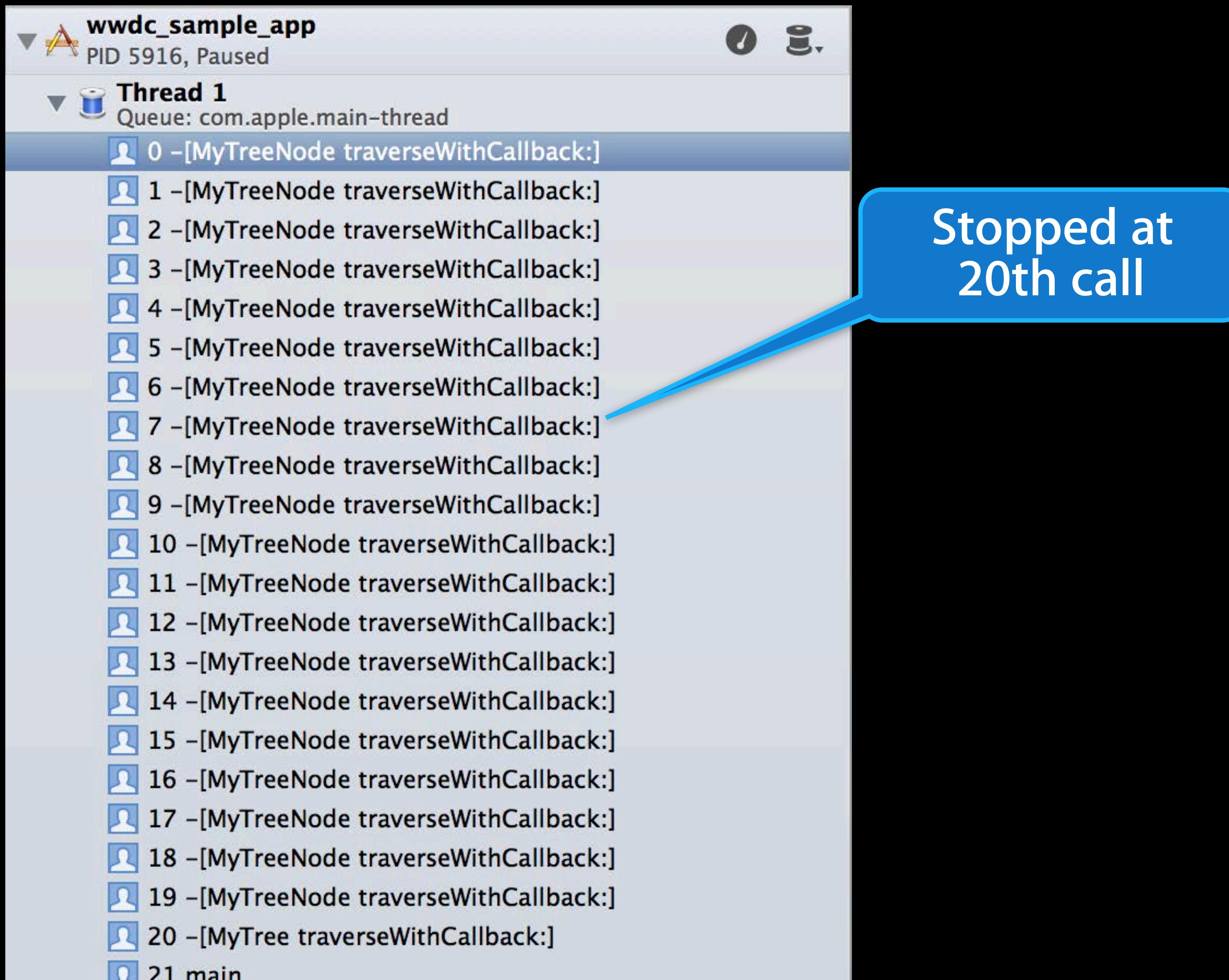
# Example

Stop if a recursion is more than  $n$  levels deep



# Example

Stop if a recursion is more than  $n$  levels deep



# 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](mailto: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

