WWDC2013

Advanced Debugging with LLDB

Session 413

Kate Stone
Software Behavioralist

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 -O -- 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 -O -- foo`
▪ Alias	`po foo`

- We will use this notation:

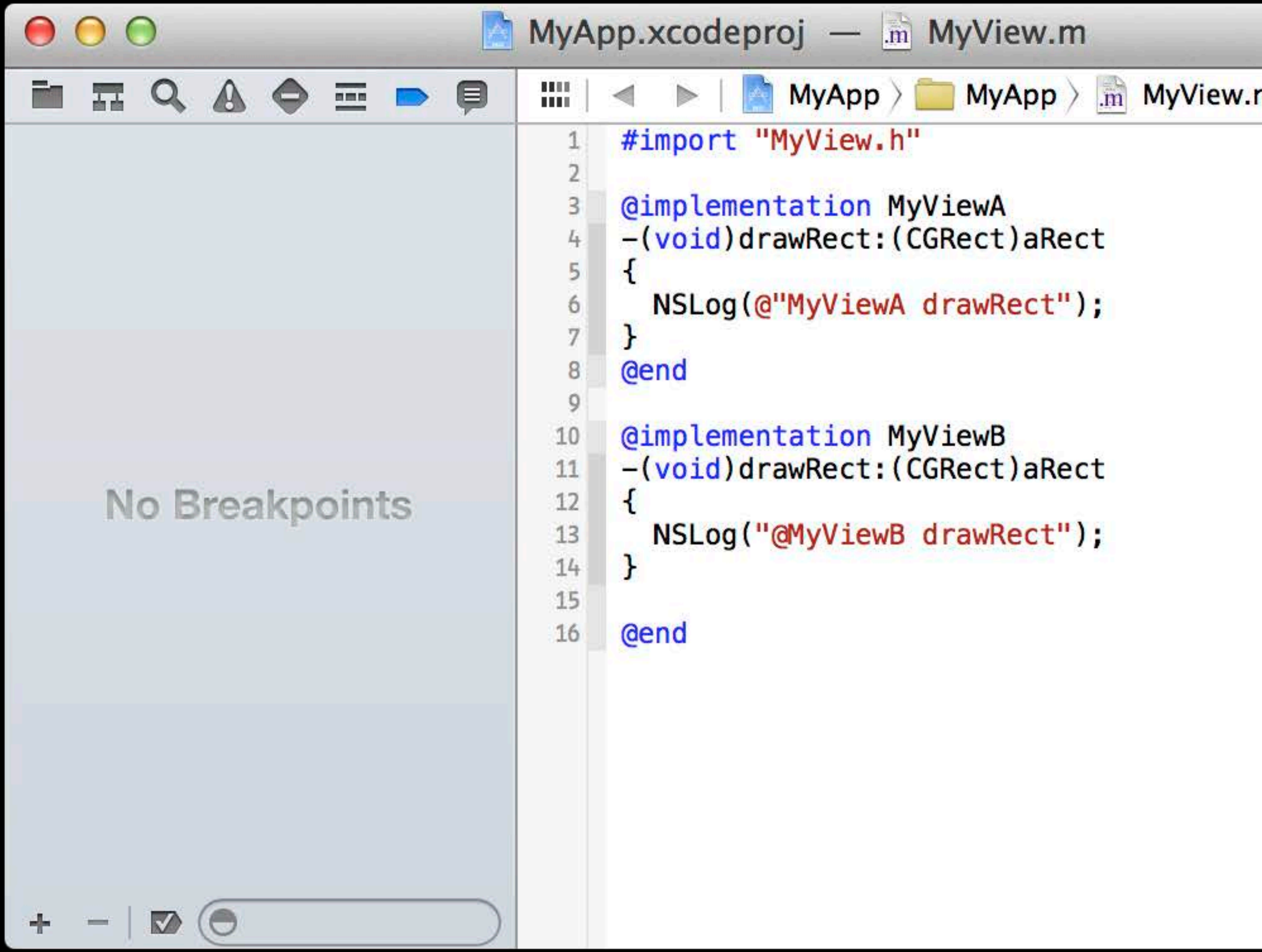
```
po foo
```

← Shortest possible form

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

← Discoverable form

Common Breakpoint Scenarios



No Breakpoints

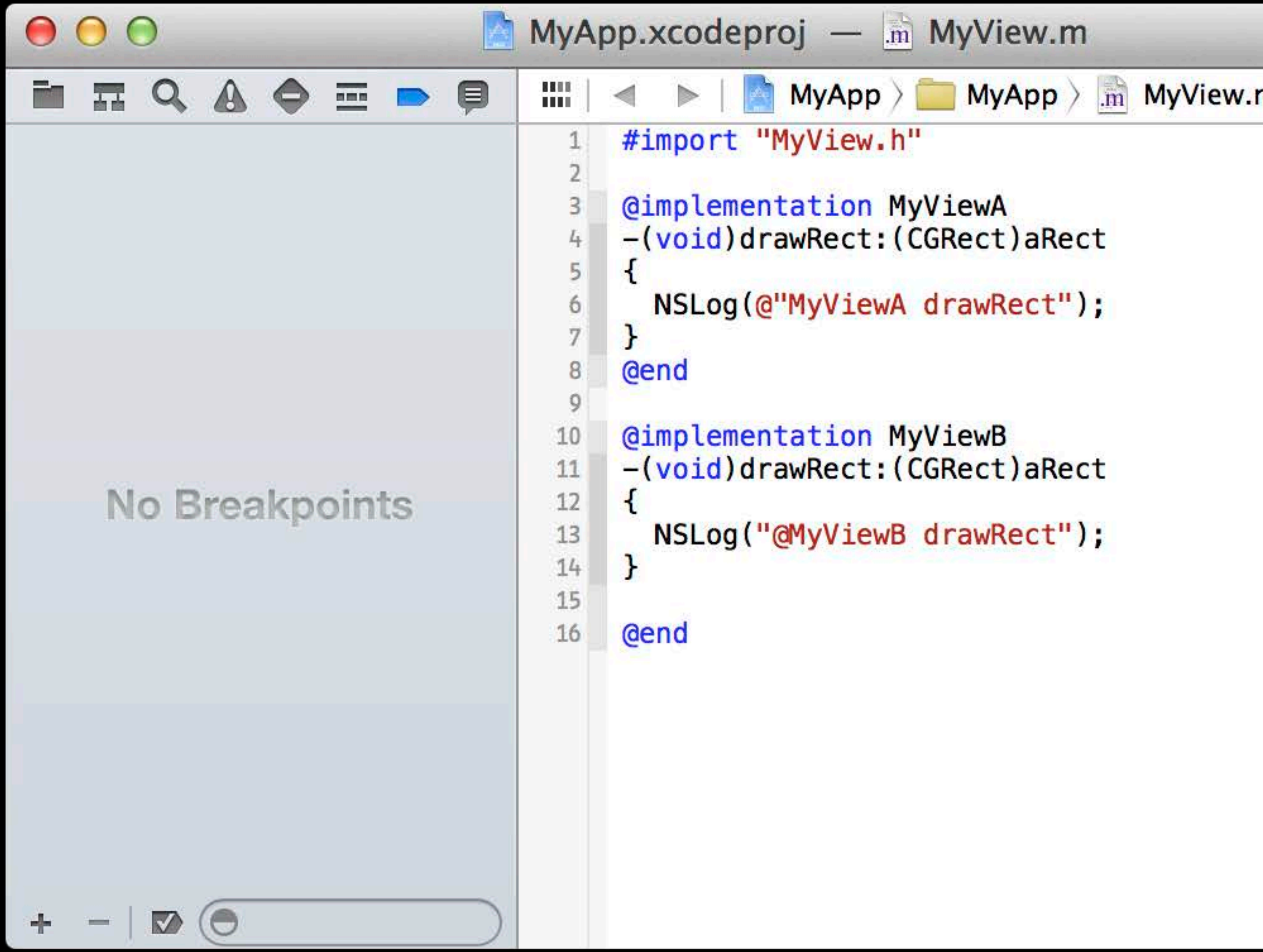
```
#import "MyView.h"

@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
```



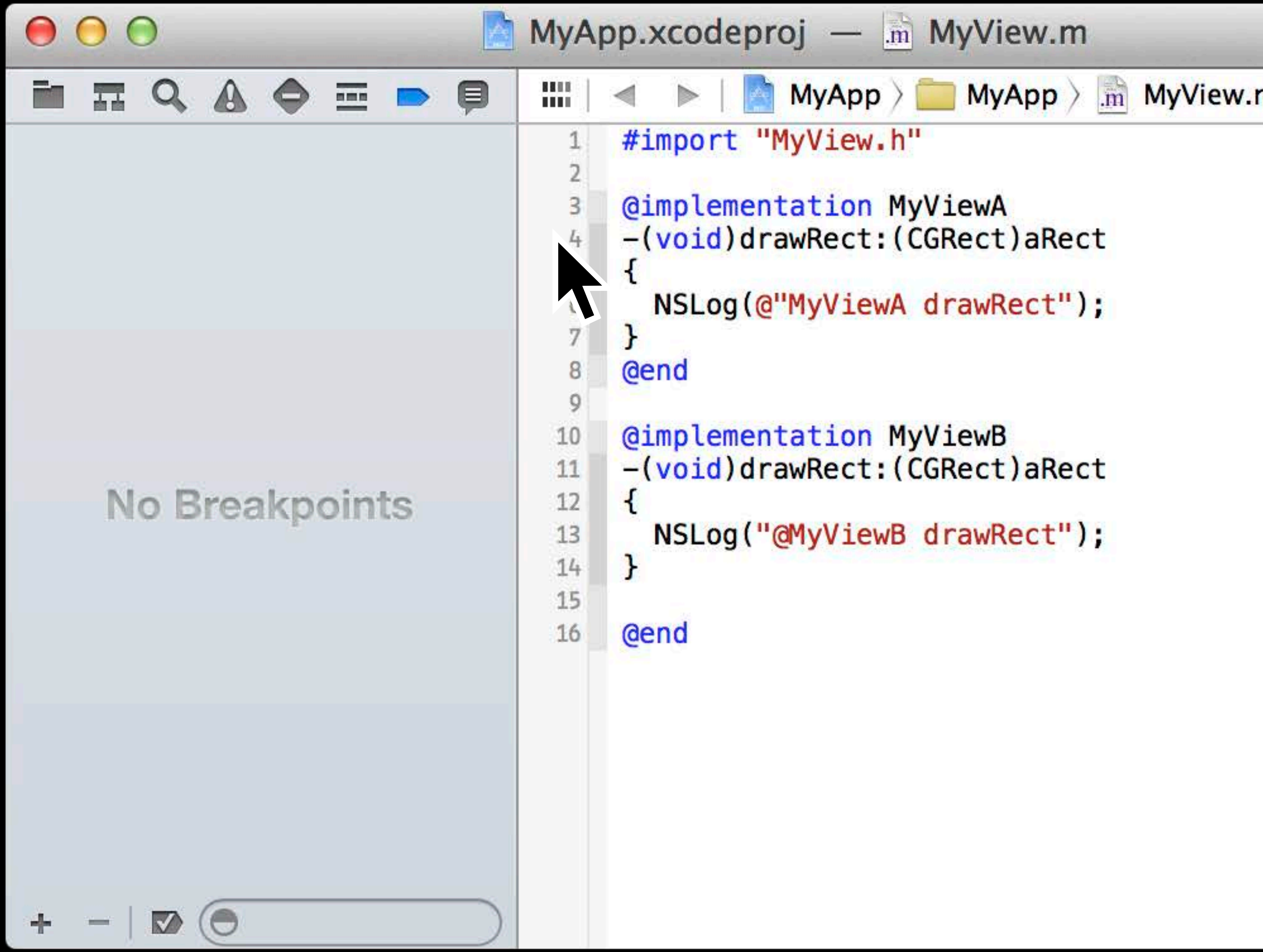
```
 1  #import "MyView.h"
 2
 3  @implementation MyViewA
 4  -(void)drawRect:(CGRect)aRect
 5  {
 6    NSLog(@"MyViewA drawRect");
 7  }
 8  @end
 9
10  @implementation MyViewB
11  -(void)drawRect:(CGRect)aRect
12  {
13    NSLog("@MyViewB drawRect");
14  }
15
16  @end
```

Common Breakpoint Scenarios

- Stop at a source line:

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

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



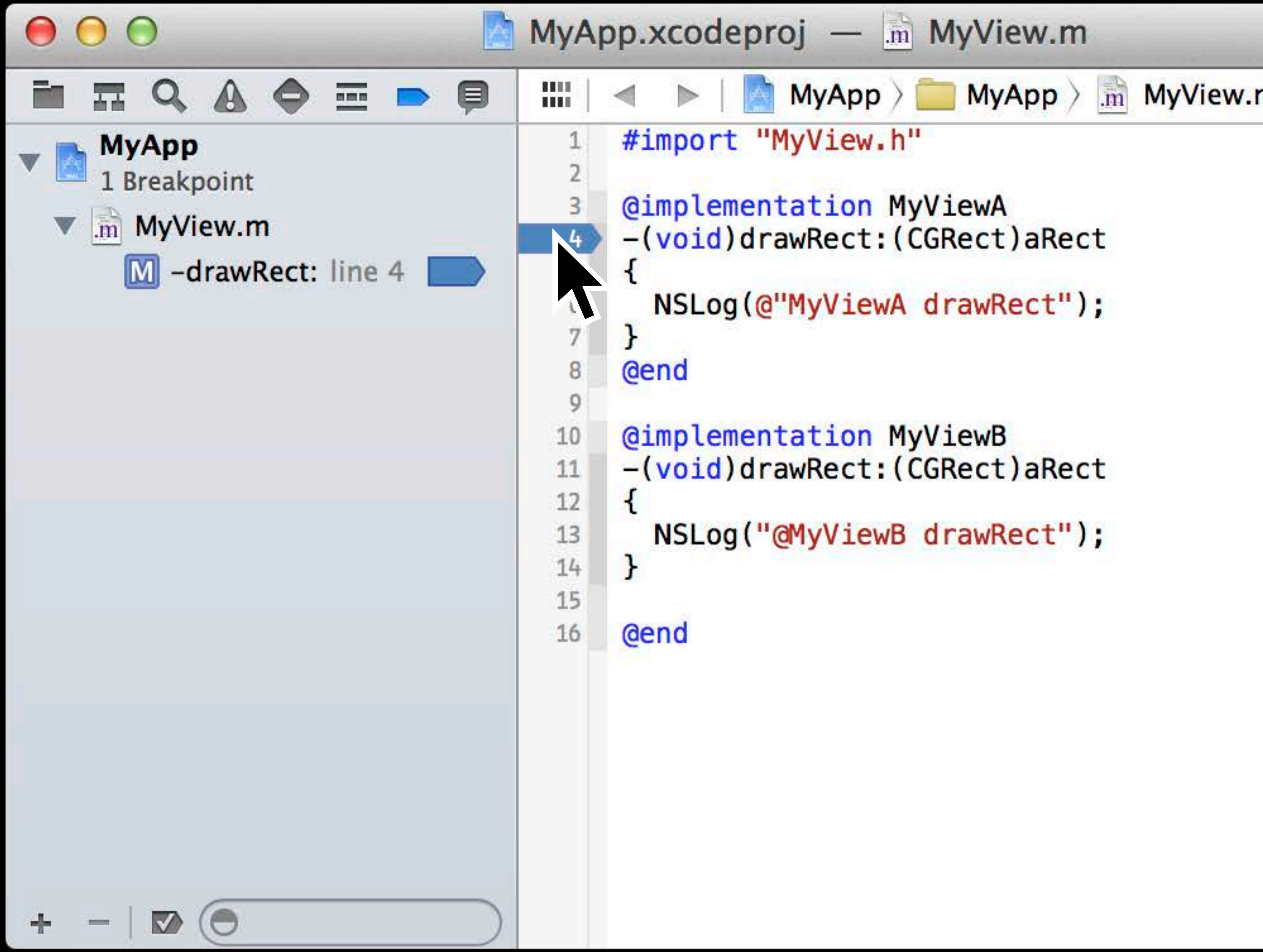
```
1   #import "MyView.h"
2
3   @implementation MyViewA
4   -(void)drawRect:(CGRect)aRect
5   {
6     NSLog(@"MyViewA drawRect");
7   }
8   @end
9
10  @implementation MyViewB
11  -(void)drawRect:(CGRect)aRect
12  {
13    NSLog("@MyViewB drawRect");
14  }
15
16  @end
```

Common Breakpoint Scenarios

- Stop at a source line:

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

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



```
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:

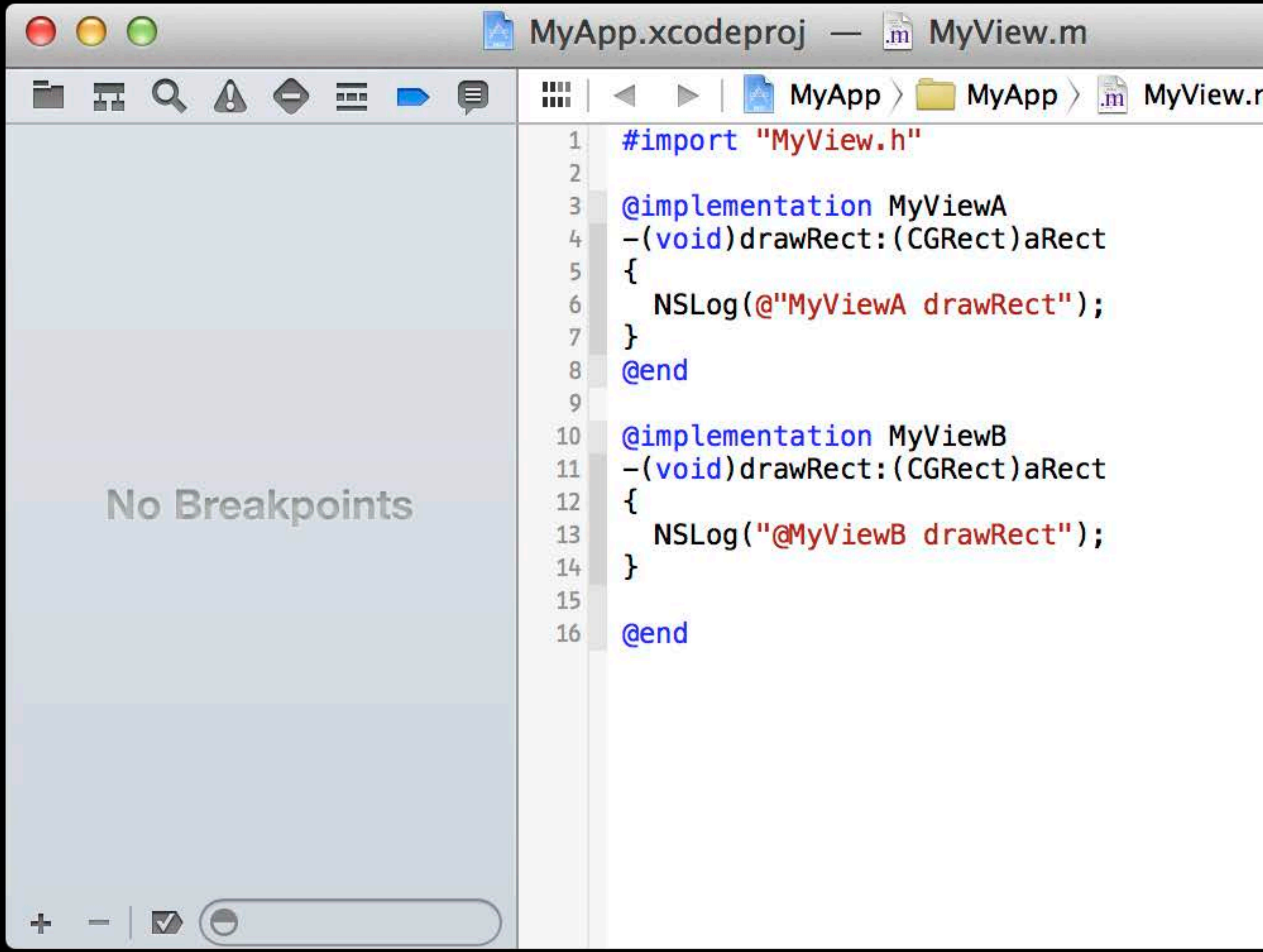
```
b MyView.m:4
```

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

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

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



```
 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:

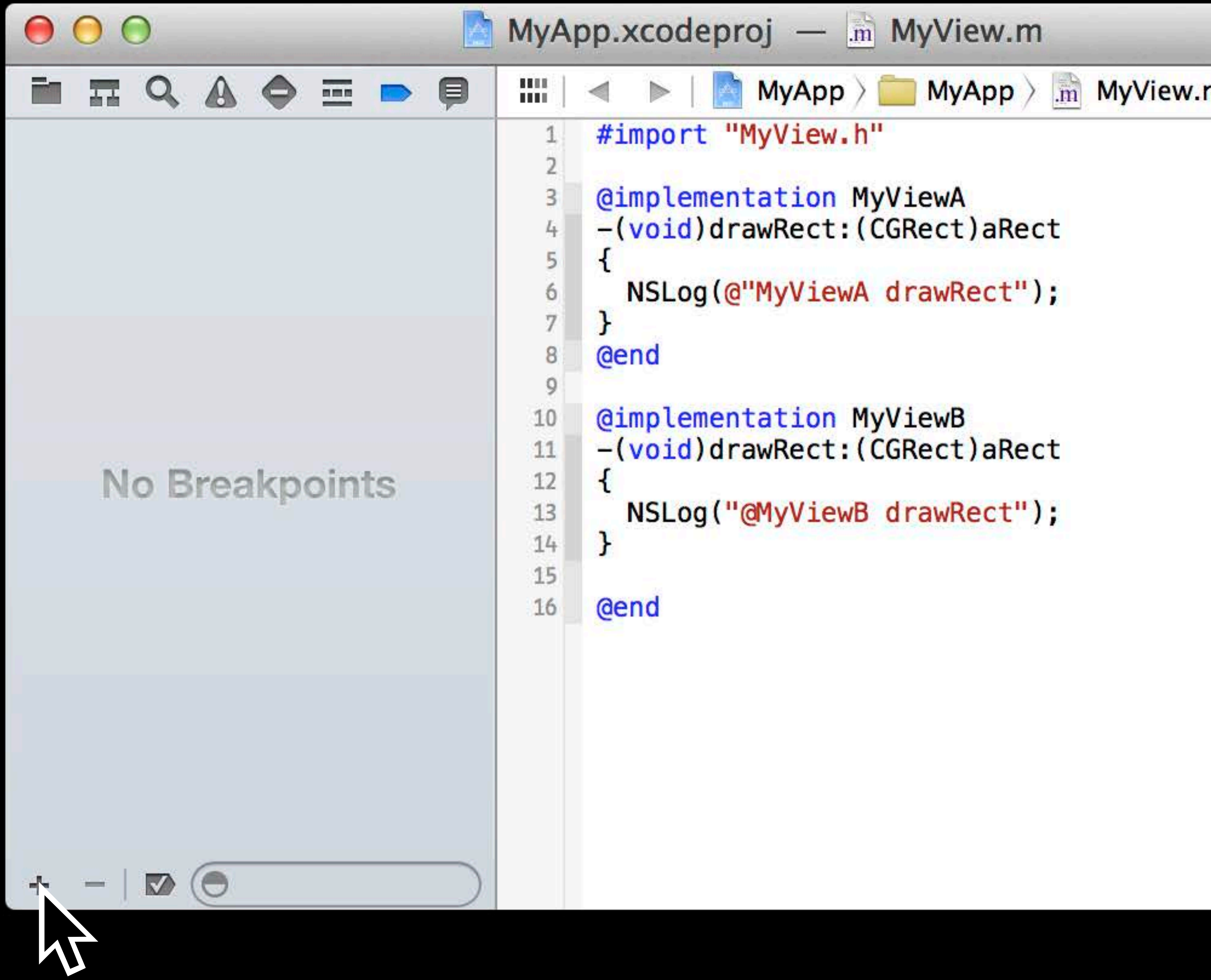
```
b MyView.m:4
```

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

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

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



```
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:

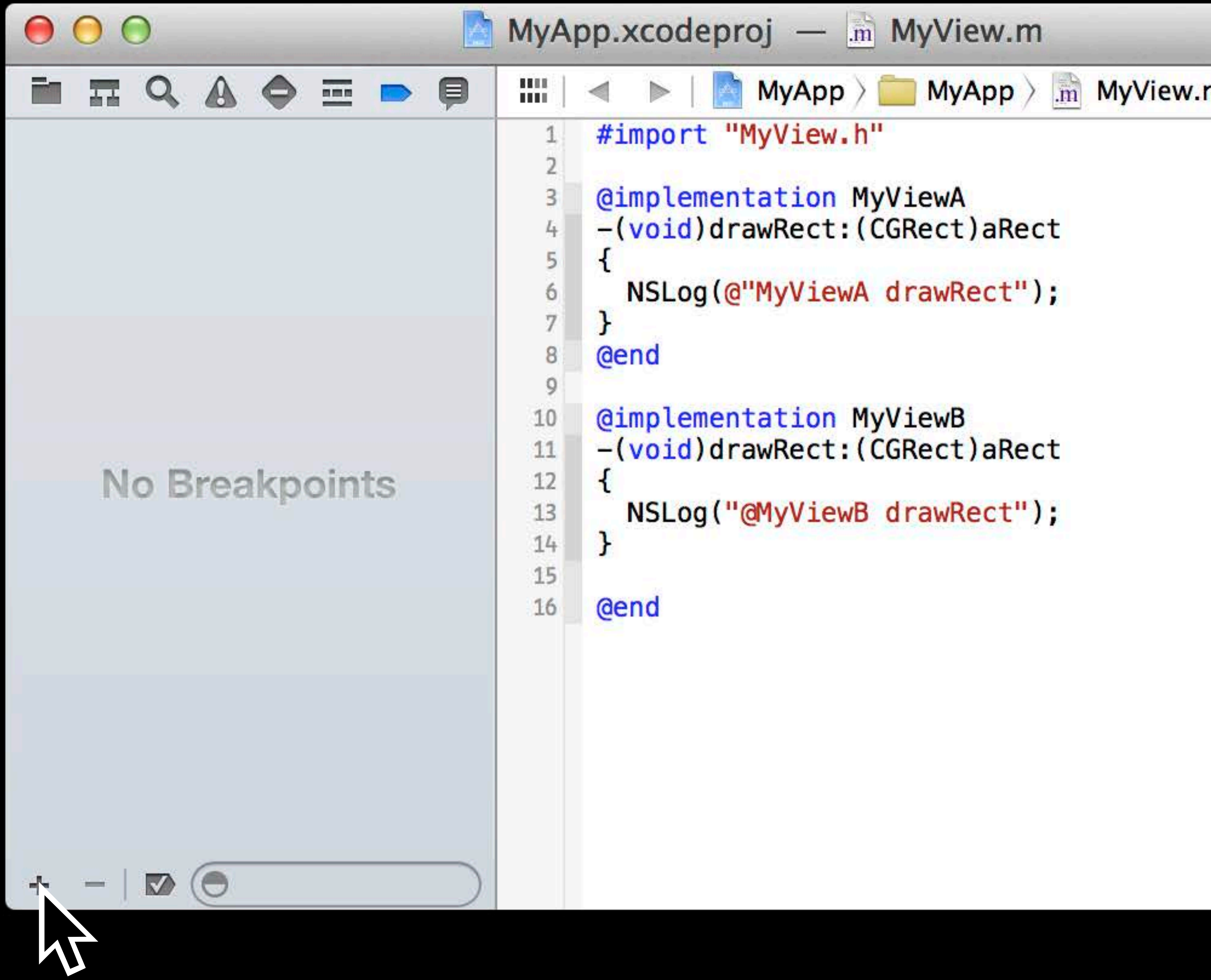
```
b MyView.m:4
```

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

- Stop at a method:

```
b “–[MyViewA drawRect:]”
```

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



```
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:

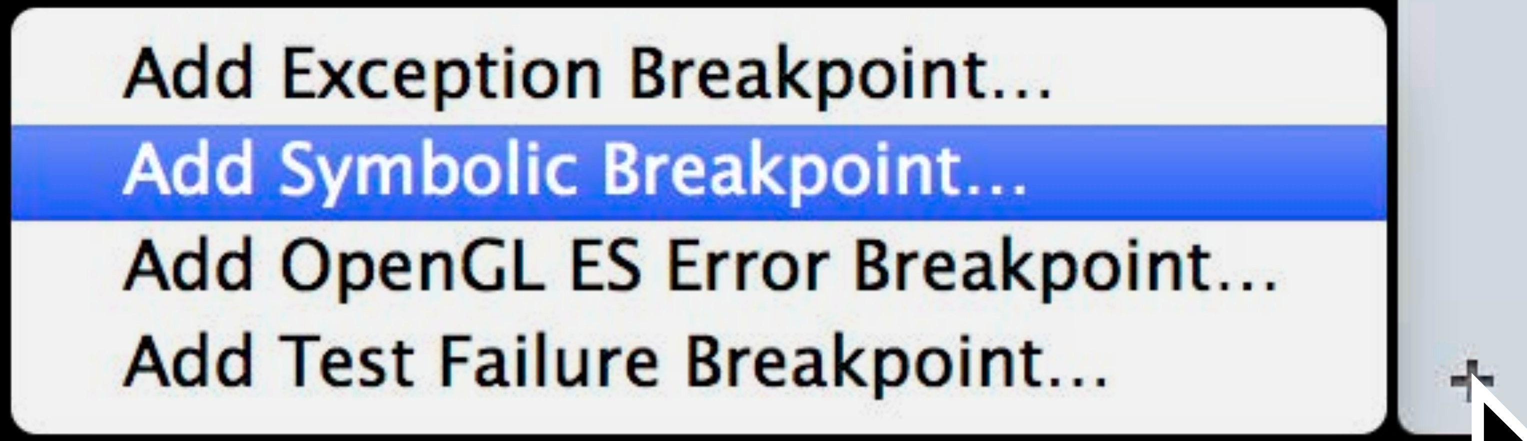
```
b MyView.m:4
```

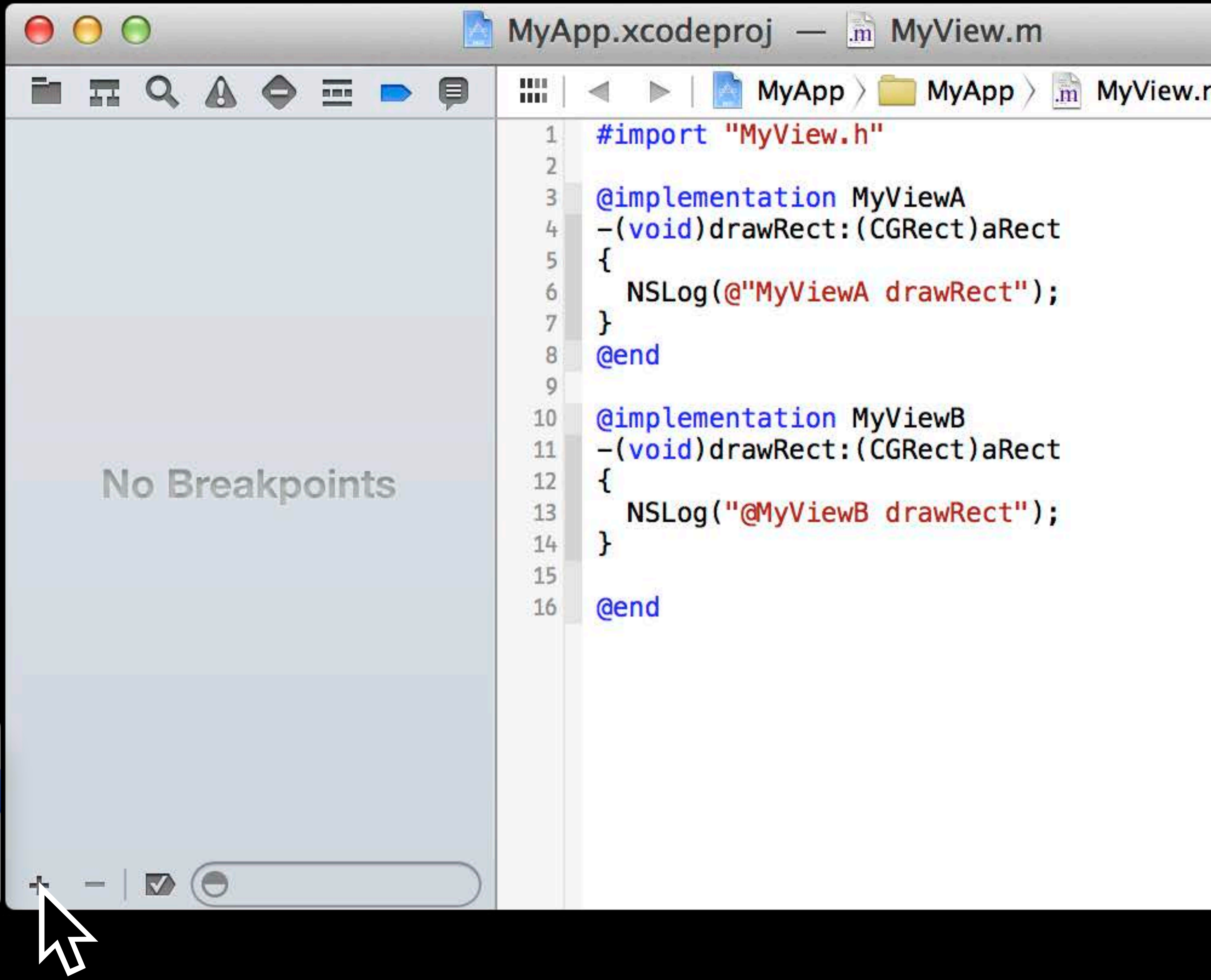
```
breakpoint set
  --file MyView.m --line 4
```

- 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...



```
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:

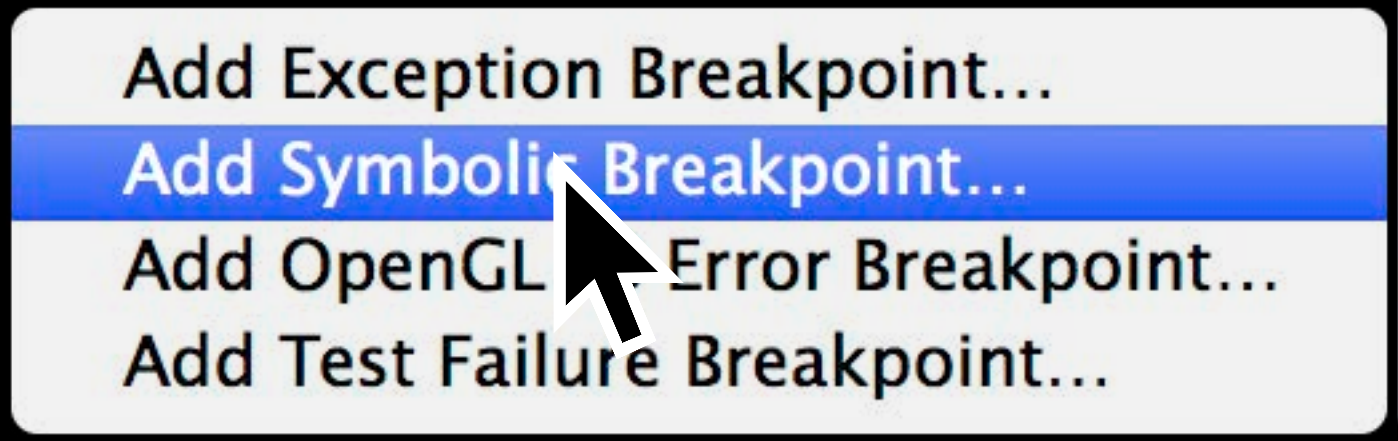
```
b MyView.m:4
```

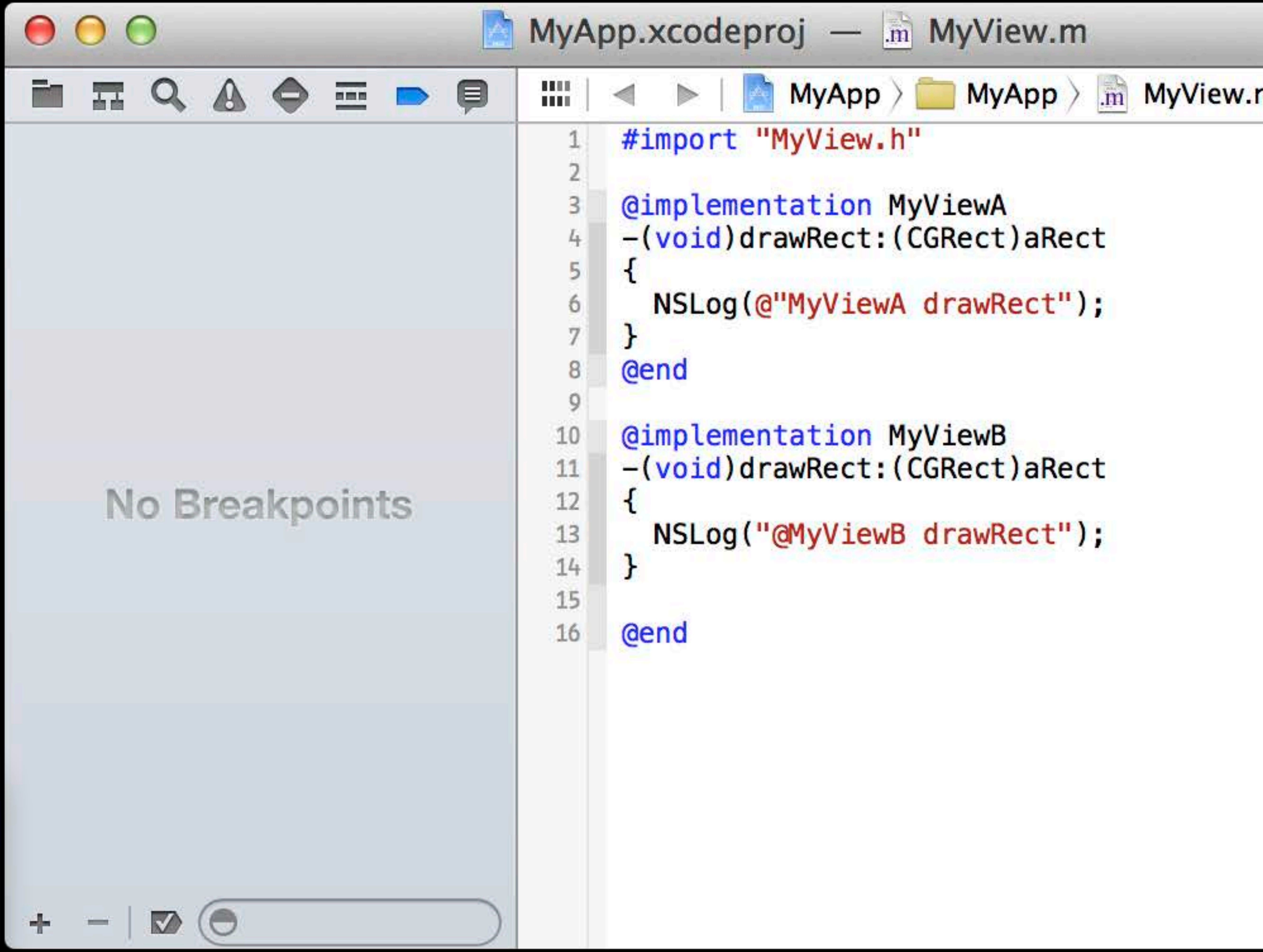
```
breakpoint set
  --file MyView.m --line 4
```

- 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...



```
#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:

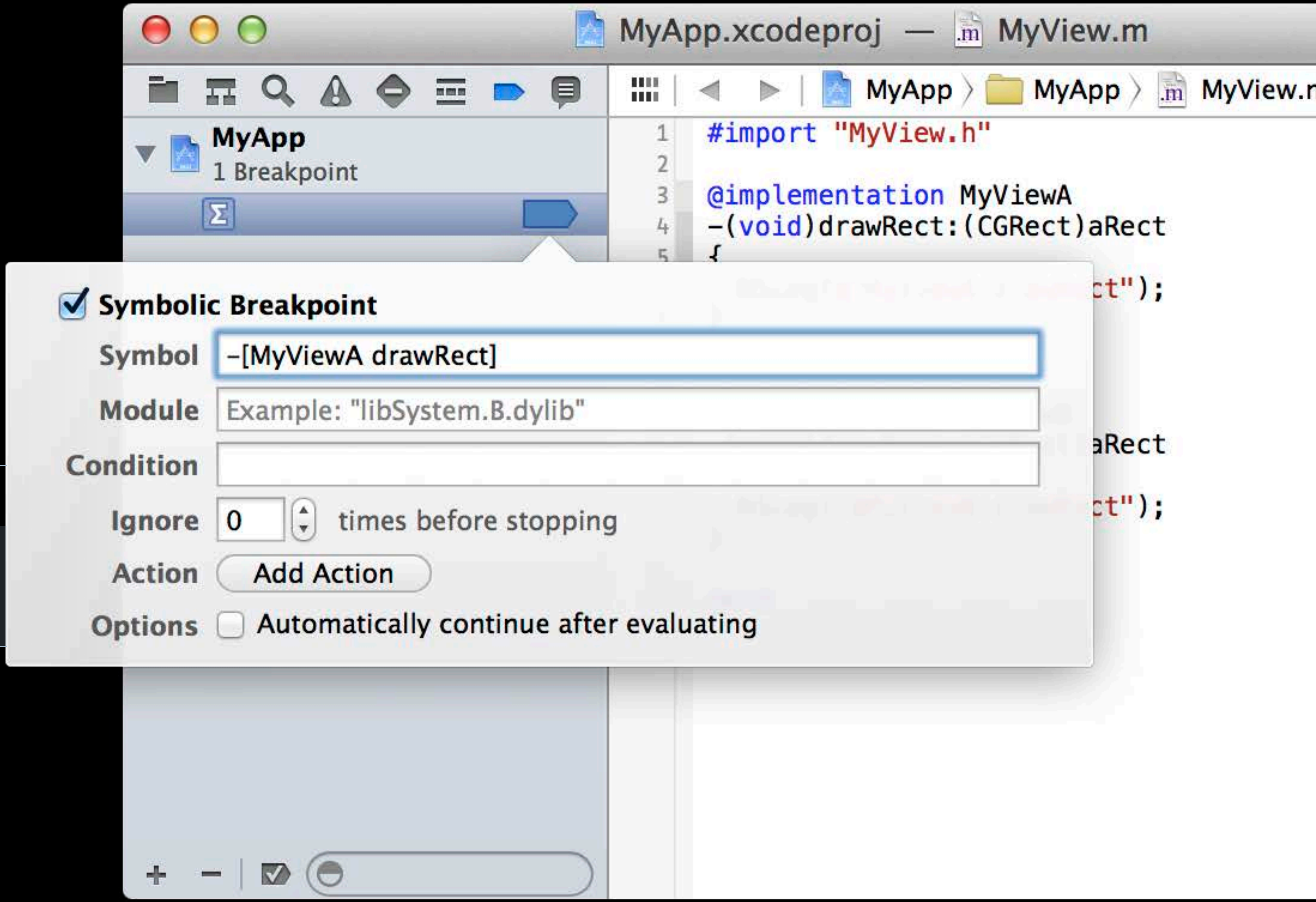
```
b MyView.m:4
```

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

- Stop at a method:

```
b “–[MyViewA drawRect:]”
```

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



```
#import "MyView.h"

@implementation MyViewA
-(void)drawRect:(CGRect)aRect
{
                                        ct");
                                      aRect
                                        ct");
```

☑ Symbolic Breakpoint

Symbol –[MyViewA drawRect]

Module Example: "libSystem.B.dylib"

Condition

Ignore 0 times before stopping

Action Add Action

Options □ Automatically continue after evaluating

Common Breakpoint Scenarios

- Stop at a source line:

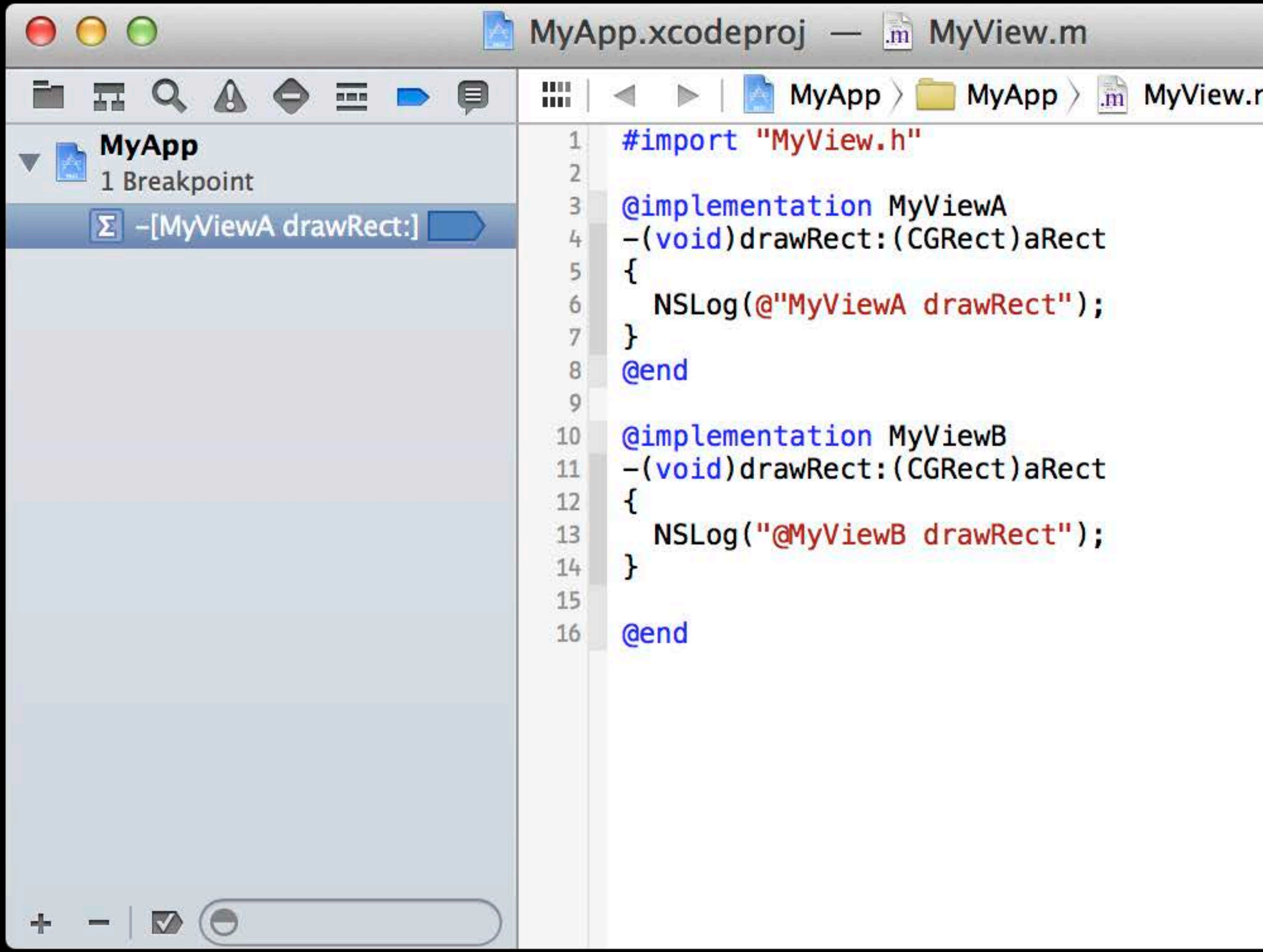
```
b MyView.m:4
```

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

- Stop at a method:

```
b “-[MyViewA drawRect:]”
```

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



```
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
```

- Stop at a method:

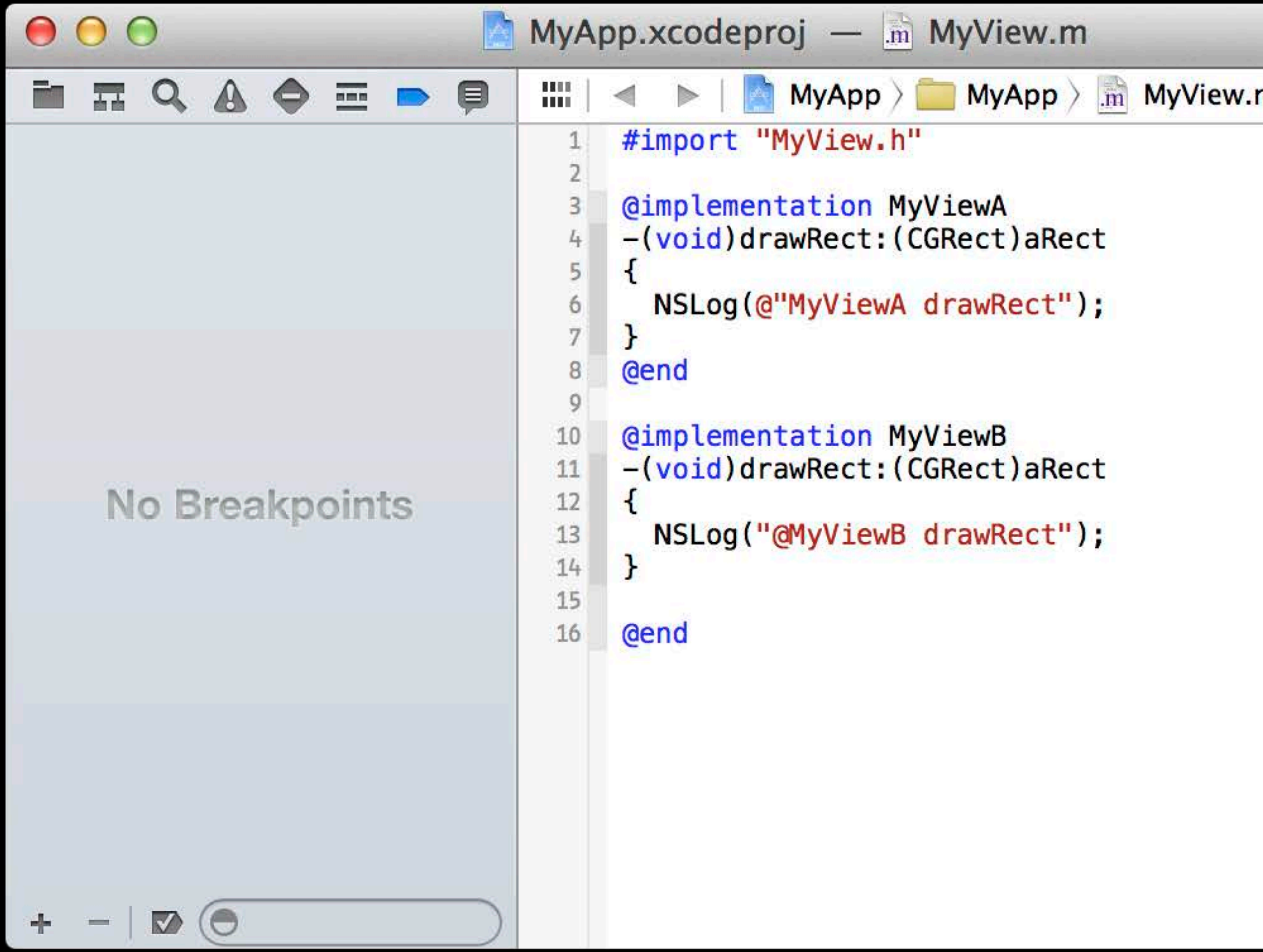
```
b “-[MyViewA drawRect:]”
```

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

- Stop whenever any object receives a selector:

```
b drawRect:
```

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

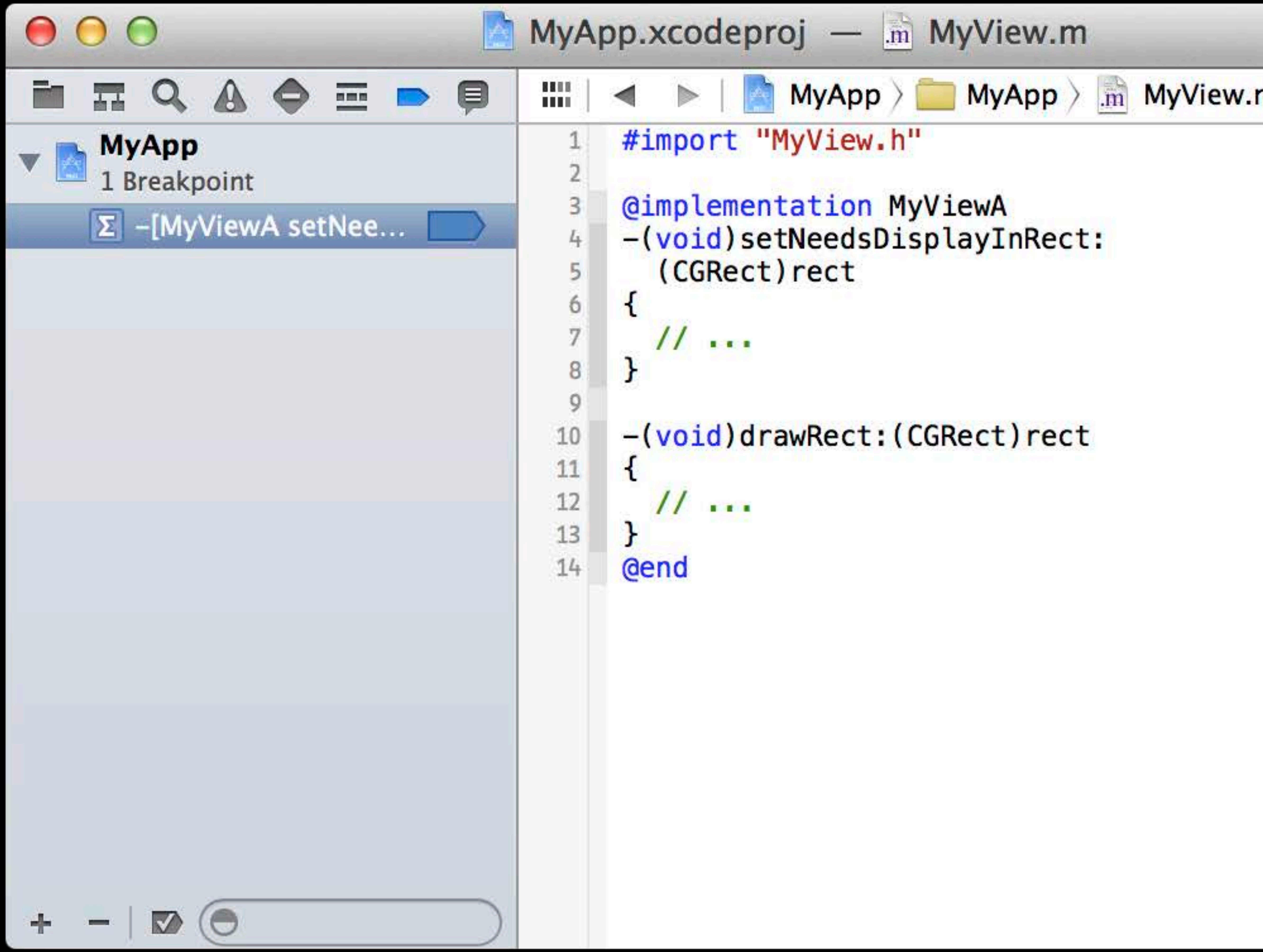


```
1  #import "MyView.h"
2
3  @implementation MyViewA
4  -(void)drawRect:(CGRect)aRect
5  {
6    NSLog(@"MyViewA drawRect");
7  }
8  @end
9
10 @implementation MyViewB
11 -(void)drawRect:(CGRect)aRect
12 {
13   NSLog("@MyViewB drawRect");
14 }
15
16 @end
```

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
```

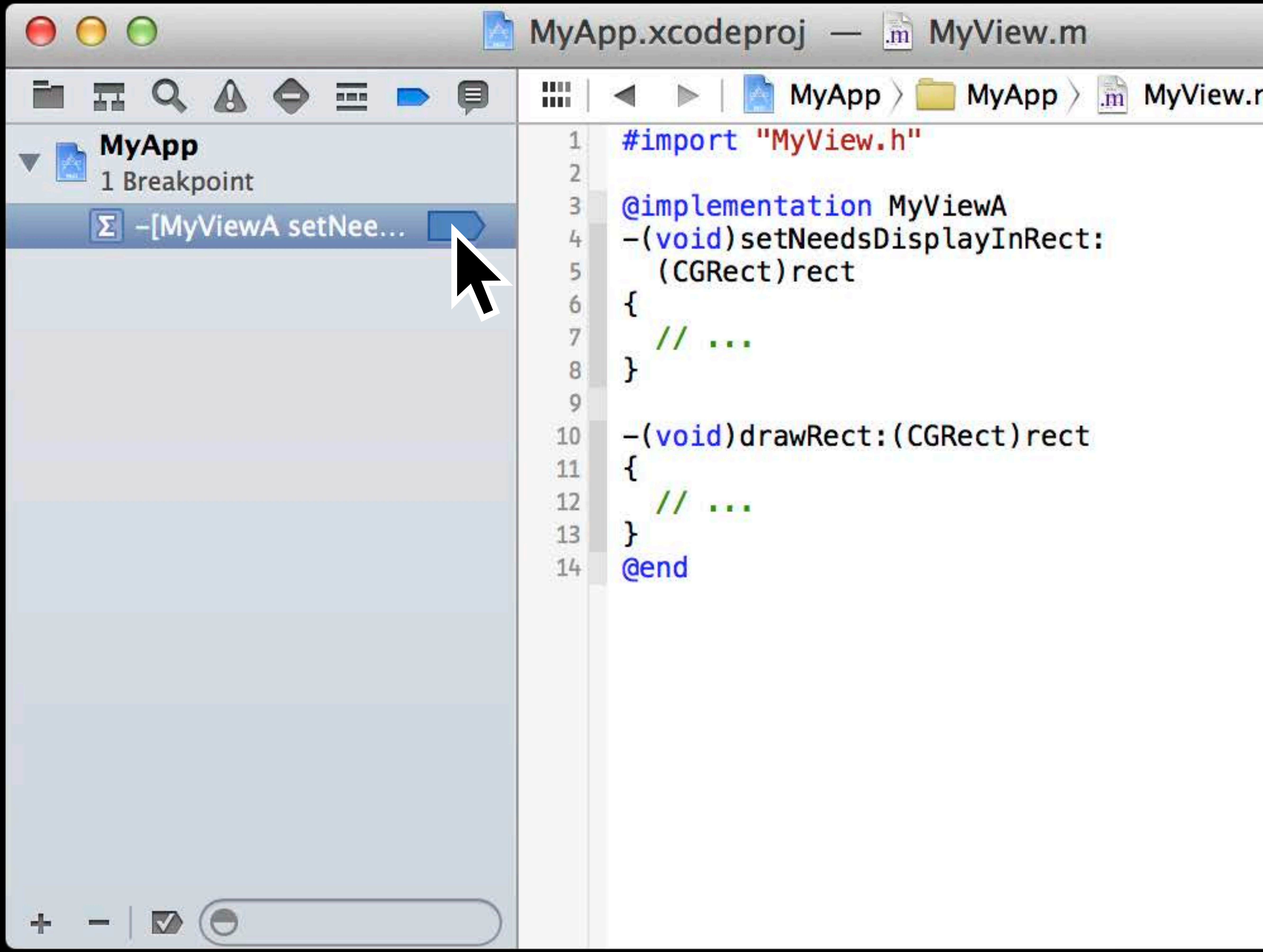


```
#import "MyView.h"

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

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

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
```



```
#import "MyView.h"

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

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

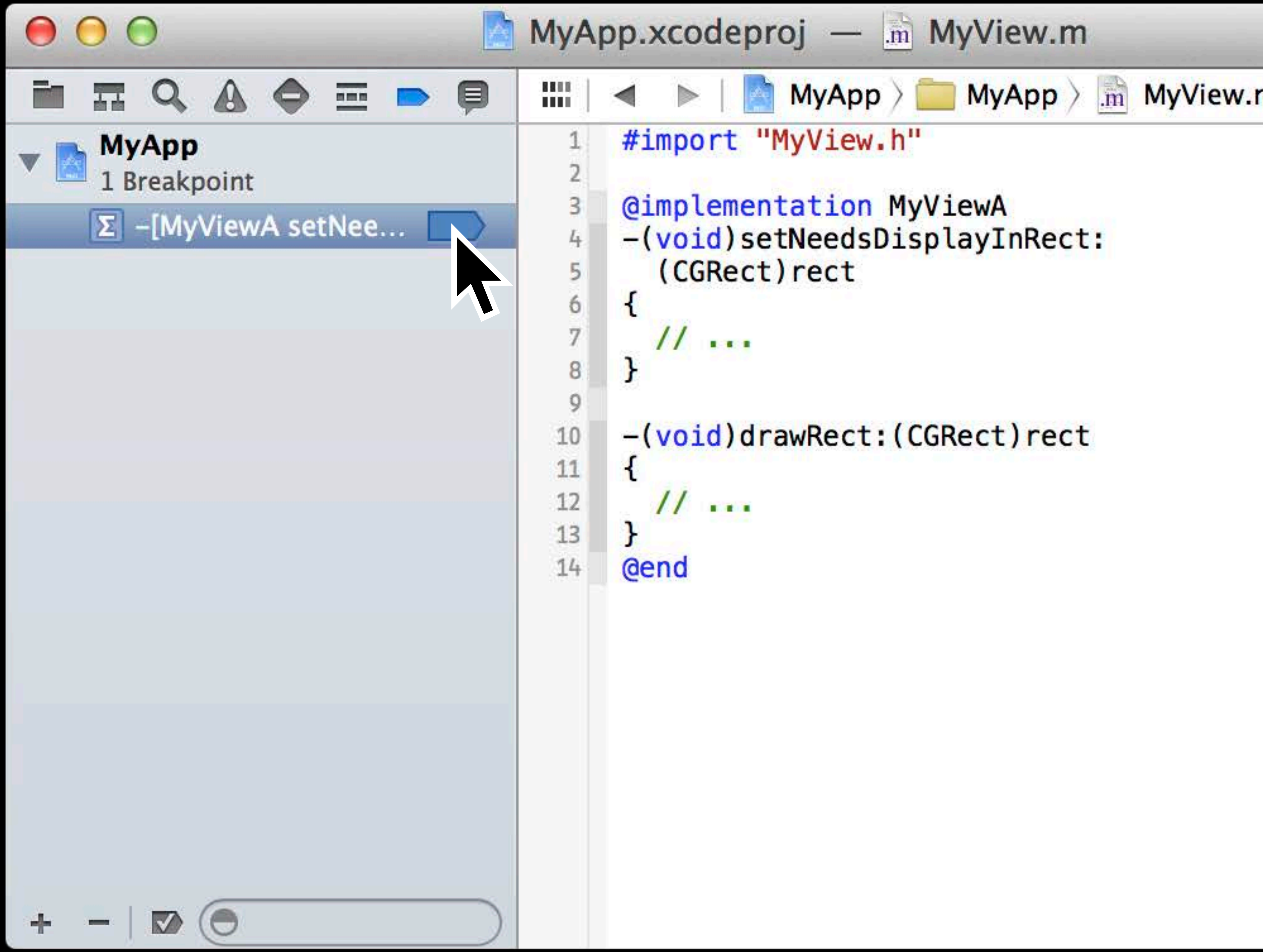
Commands Save Time

- 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
```

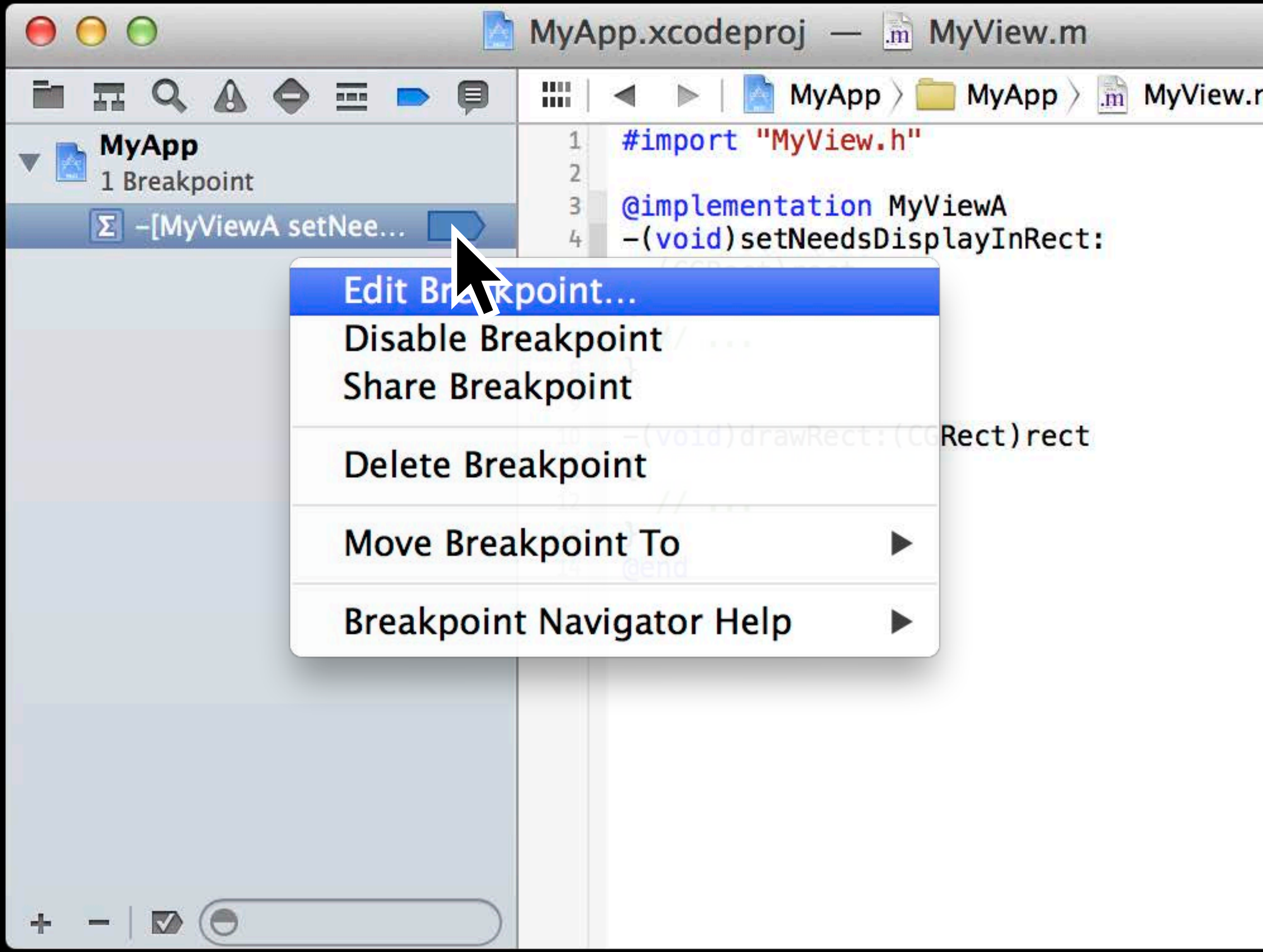


```
#import "MyView.h"

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

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

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
```



```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
```

```
Rect)rect
```

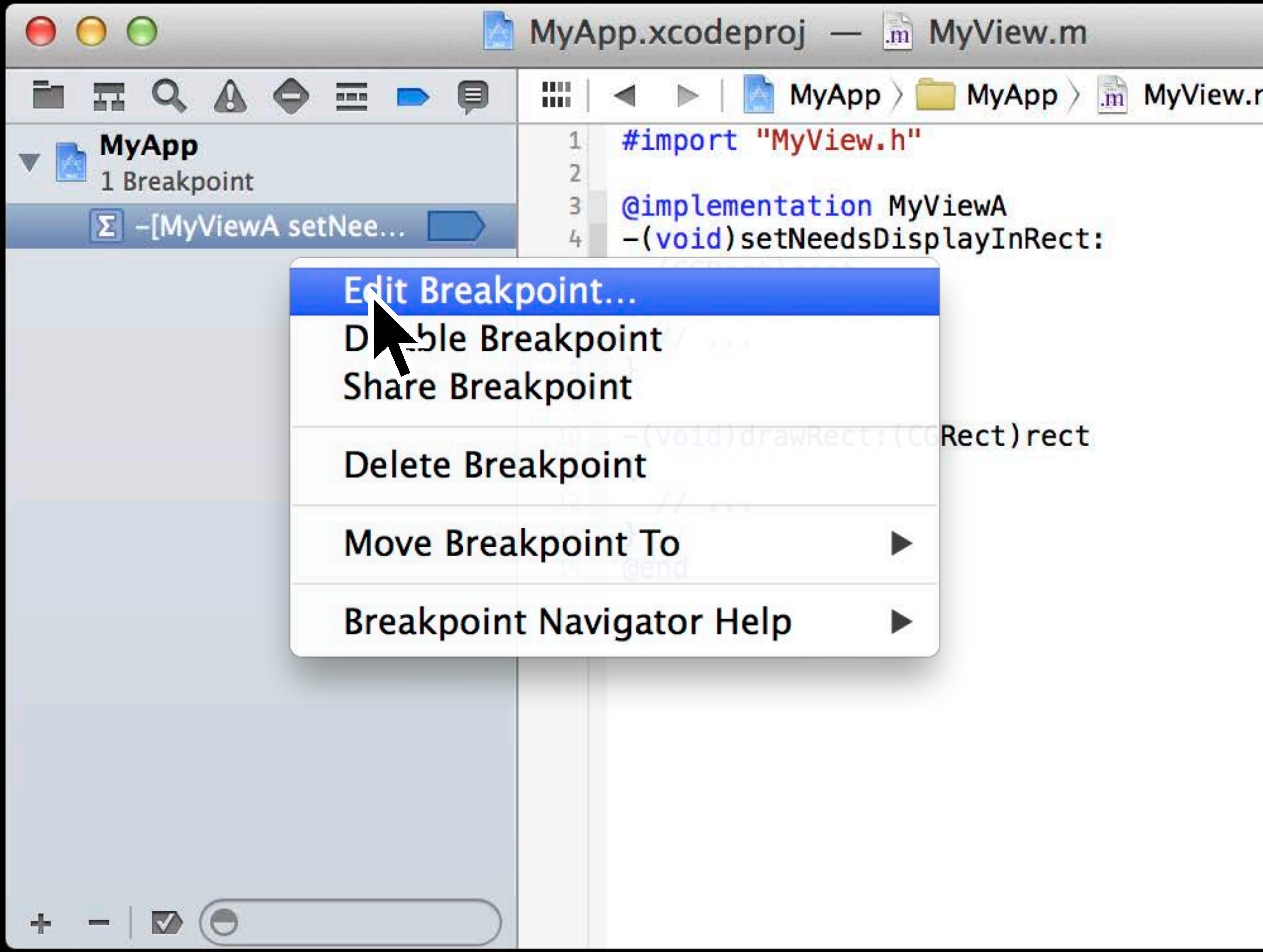
Commands Save Time

- 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
```

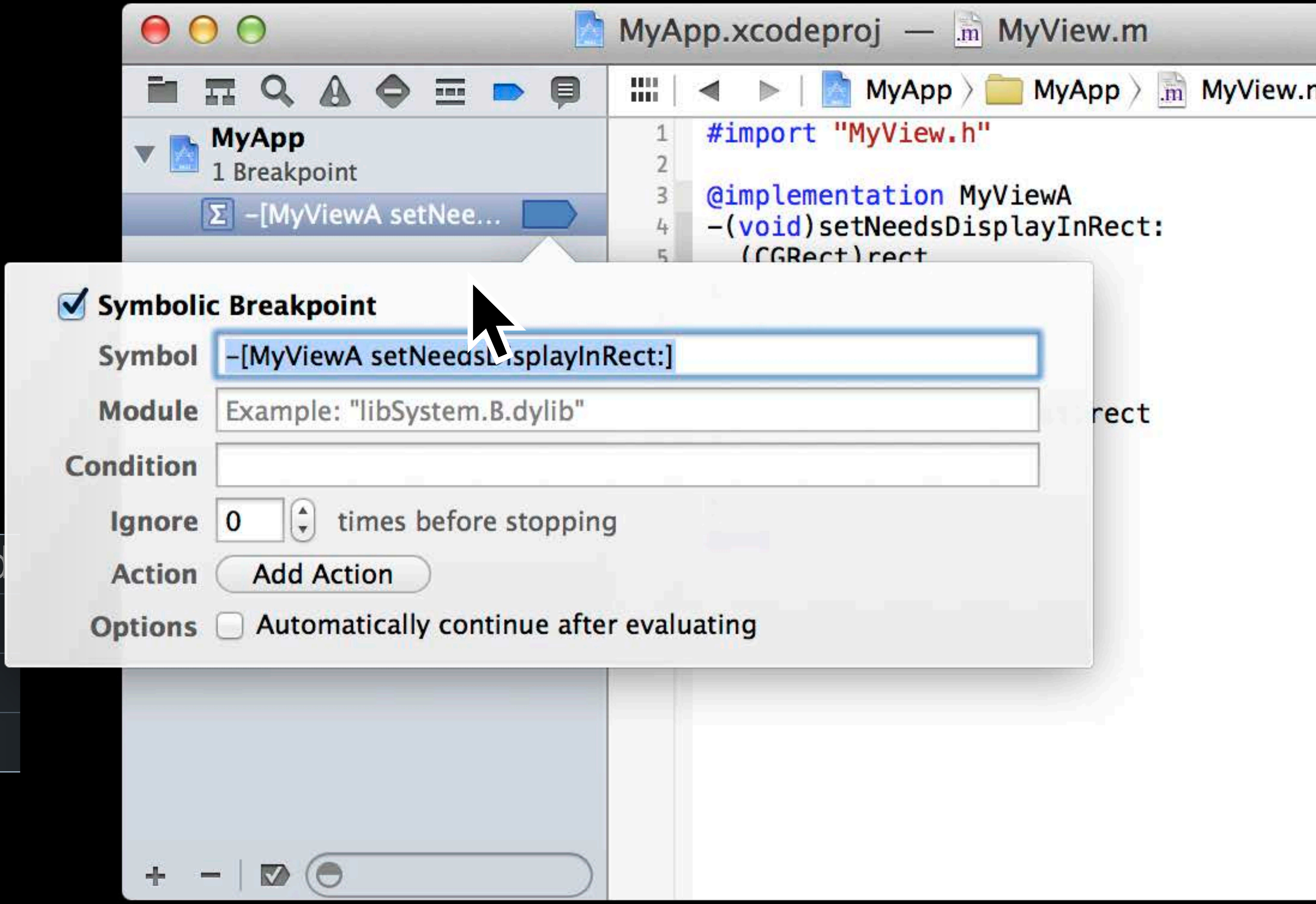


```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
Rect)rect
```

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
```



```
#import "MyView.h"

@implementation MyViewA
–(void)setNeedsDisplayInRect:
  (CGRect)rect
```

☑ Symbolic Breakpoint

Symbol: –[MyViewA setNeedsDisplayInRect:]

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

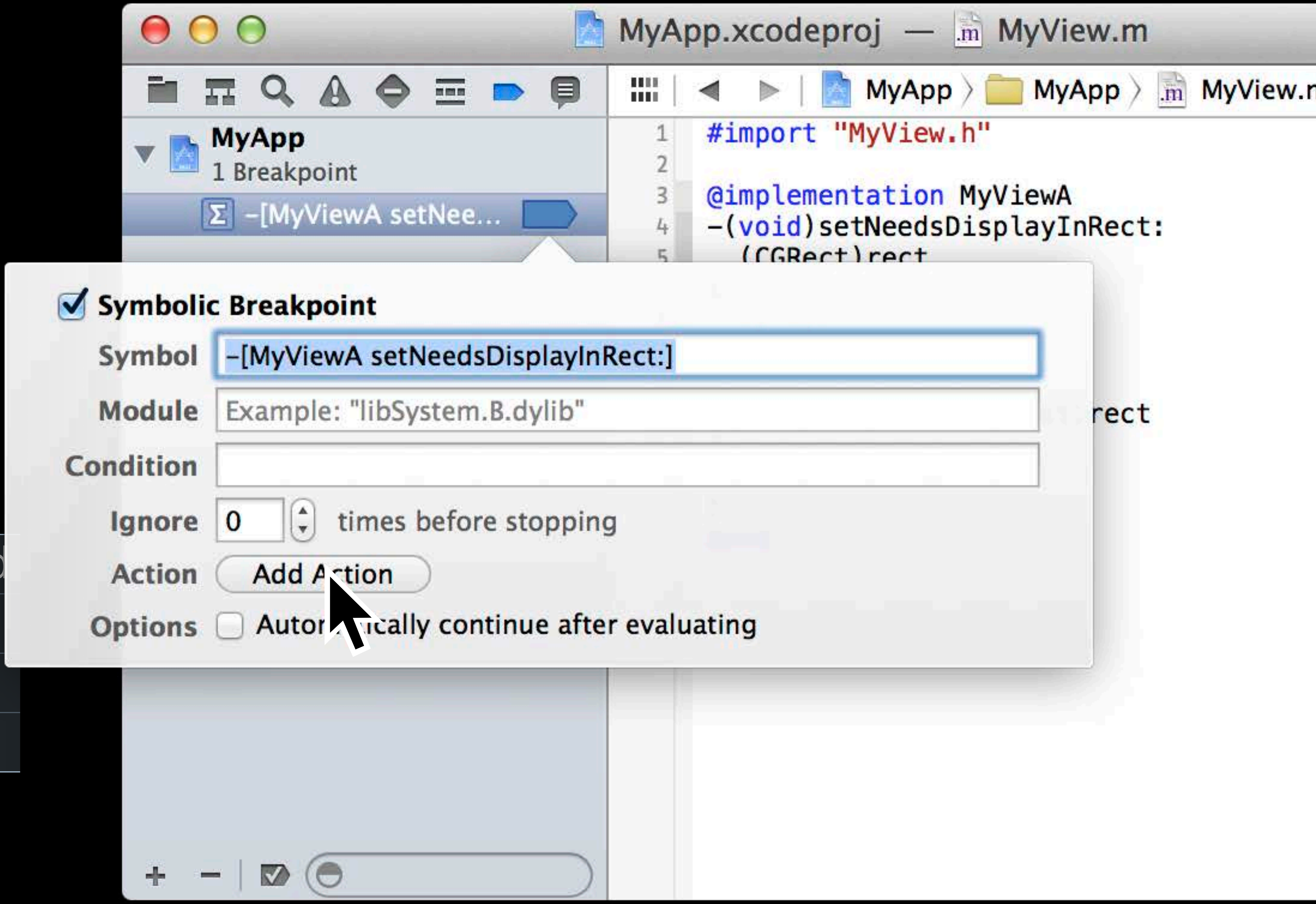
Action: Add Action

Options: ☐ Automatically continue after evaluating

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
```



MyApp.xcodeproj — MyView.m

MyApp
1 Breakpoint
–[MyViewA setNee...

```
#import "MyView.h"

@implementation MyViewA
–(void)setNeedsDisplayInRect:
  (CGRect)rect
```

☑ Symbolic Breakpoint

Symbol: –[MyViewA setNeedsDisplayInRect:]

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

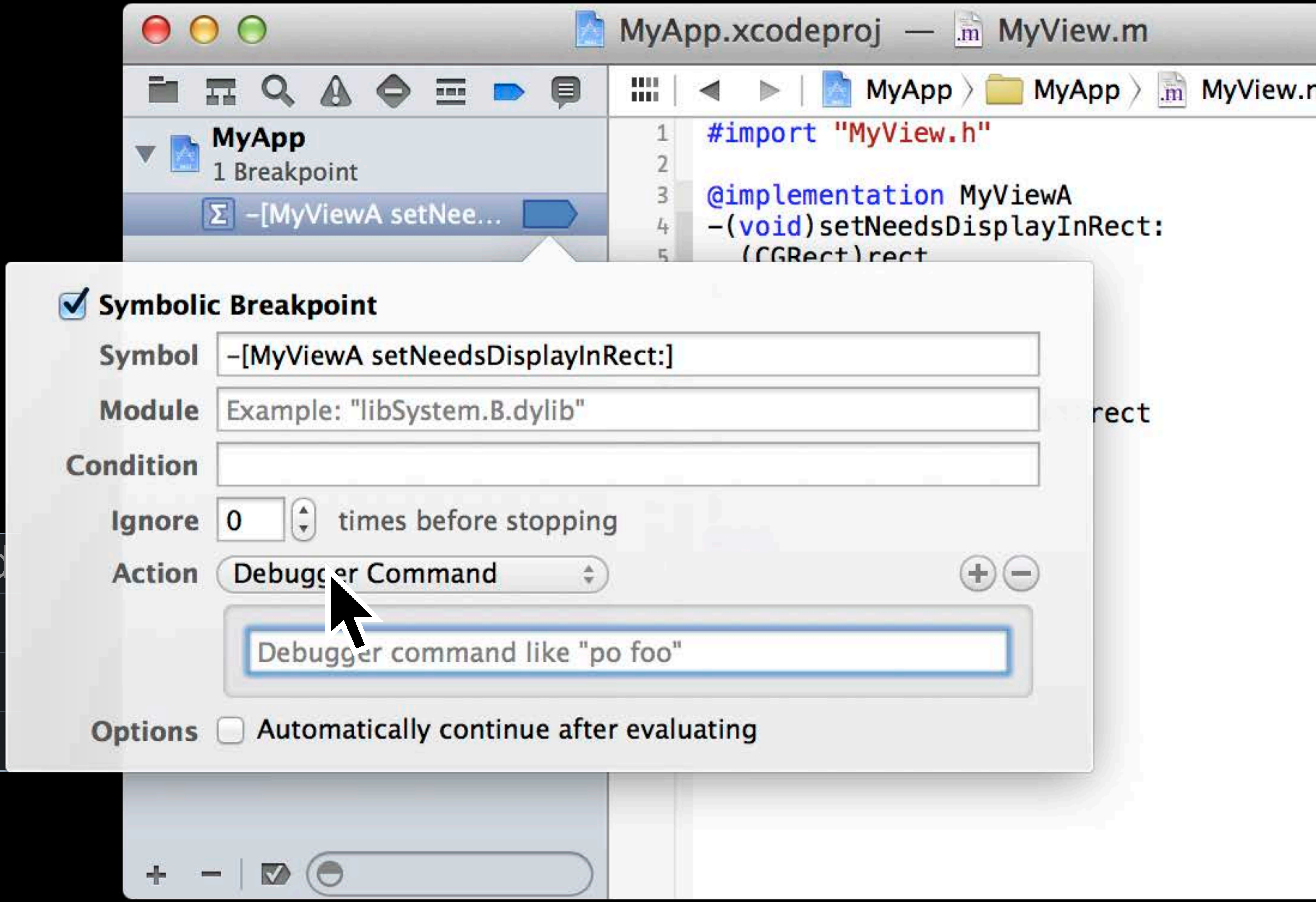
Action: Add Action

Options: □ Automatically continue after evaluating

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
```



```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
  (CGRect)rect
```

☑ Symbolic Breakpoint

Symbol: –[MyViewA setNeedsDisplayInRect:]

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

Action: Debugger Command

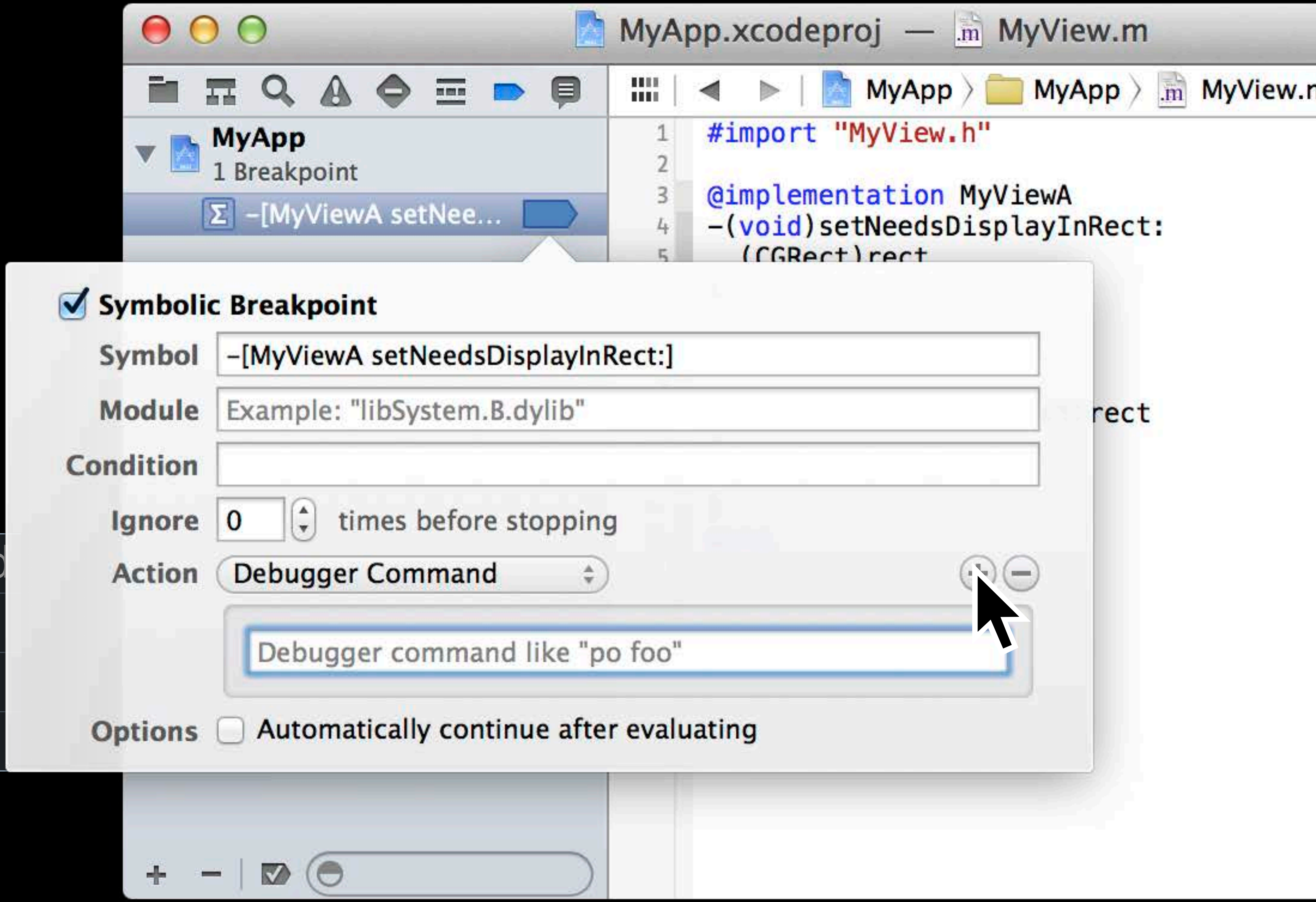
Debugger command like "po foo"

Options: □ Automatically continue after evaluating

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
```



```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
  (CGRect)rect
```

☑ Symbolic Breakpoint

Symbol: –[MyViewA setNeedsDisplayInRect:]

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

Action: Debugger Command

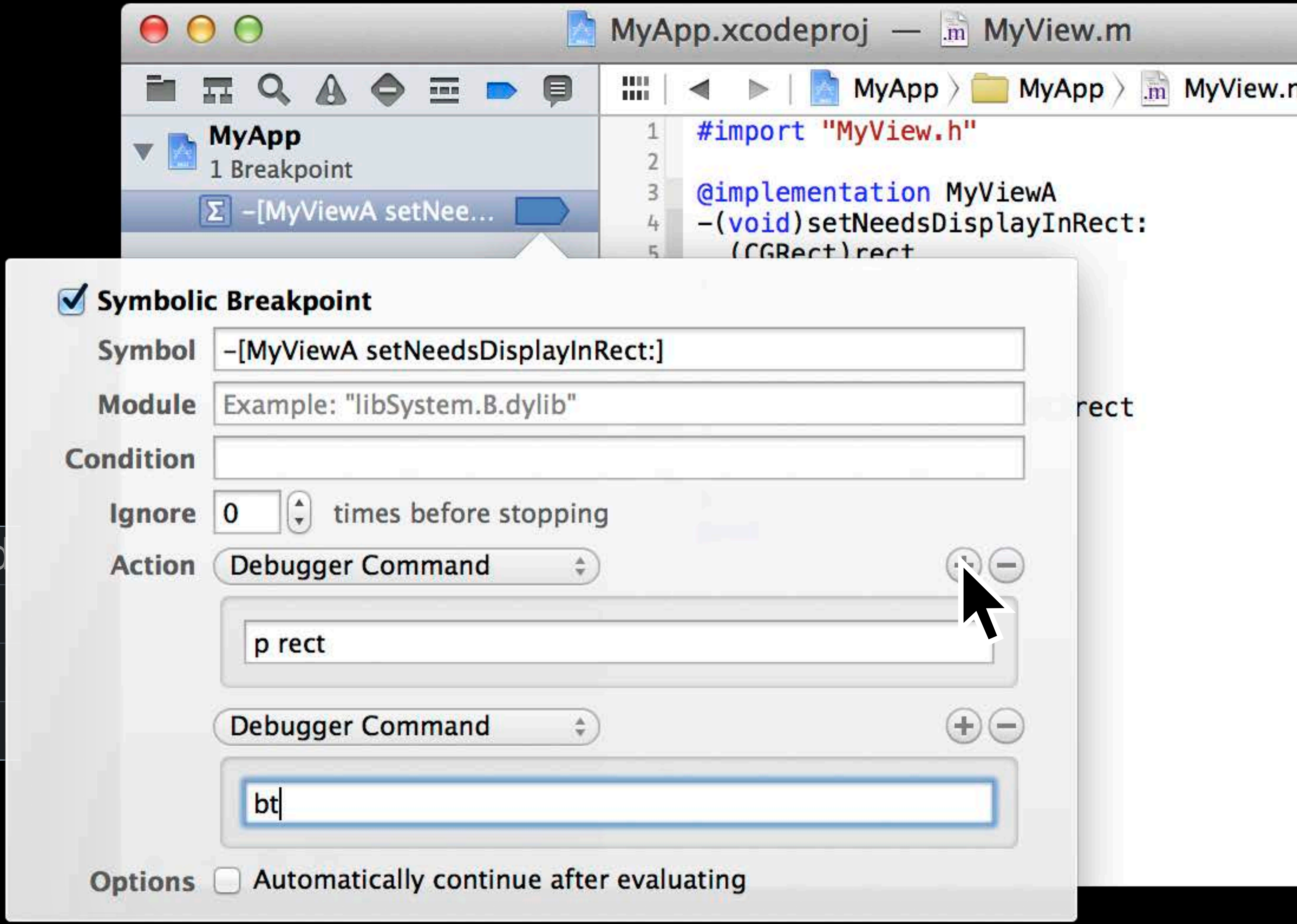
Debugger command like "po foo"

Options: □ Automatically continue after evaluating

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
```



MyApp.xcodeproj — MyView.m

MyApp 〉 MyApp 〉 MyView.m

MyApp
1 Breakpoint
–[MyViewA setNee...

```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
   (CGRect)rect
```

☑ Symbolic Breakpoint

Symbol: –[MyViewA setNeedsDisplayInRect:]

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

Action: Debugger Command

p rect

Debugger Command

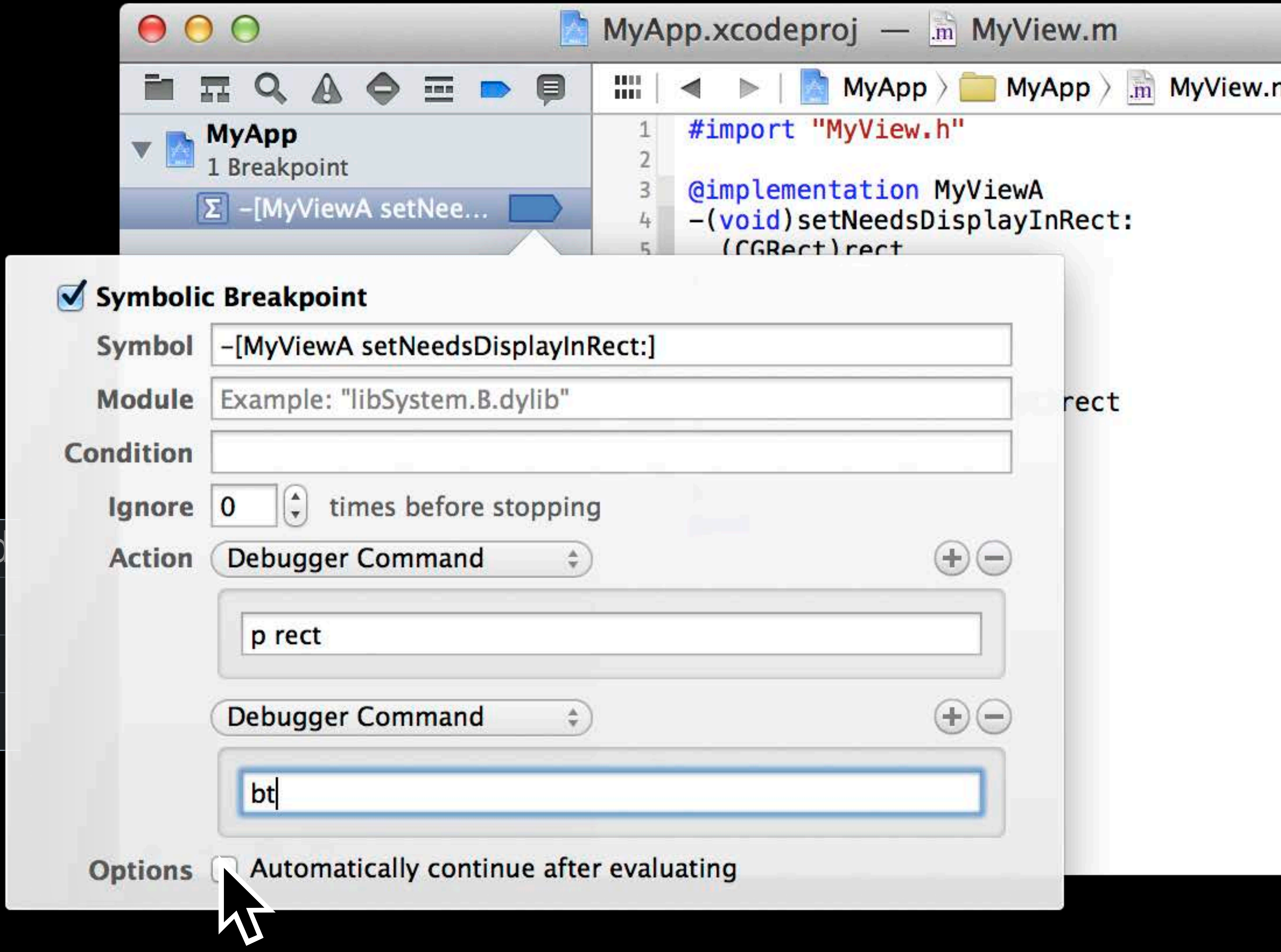
bt

Options: □ Automatically continue after evaluating

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
```



```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
  (CGRect)rect
```

☑ Symbolic Breakpoint

Symbol: –[MyViewA setNeedsDisplayInRect:]

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

Action: Debugger Command

p rect

Debugger Command

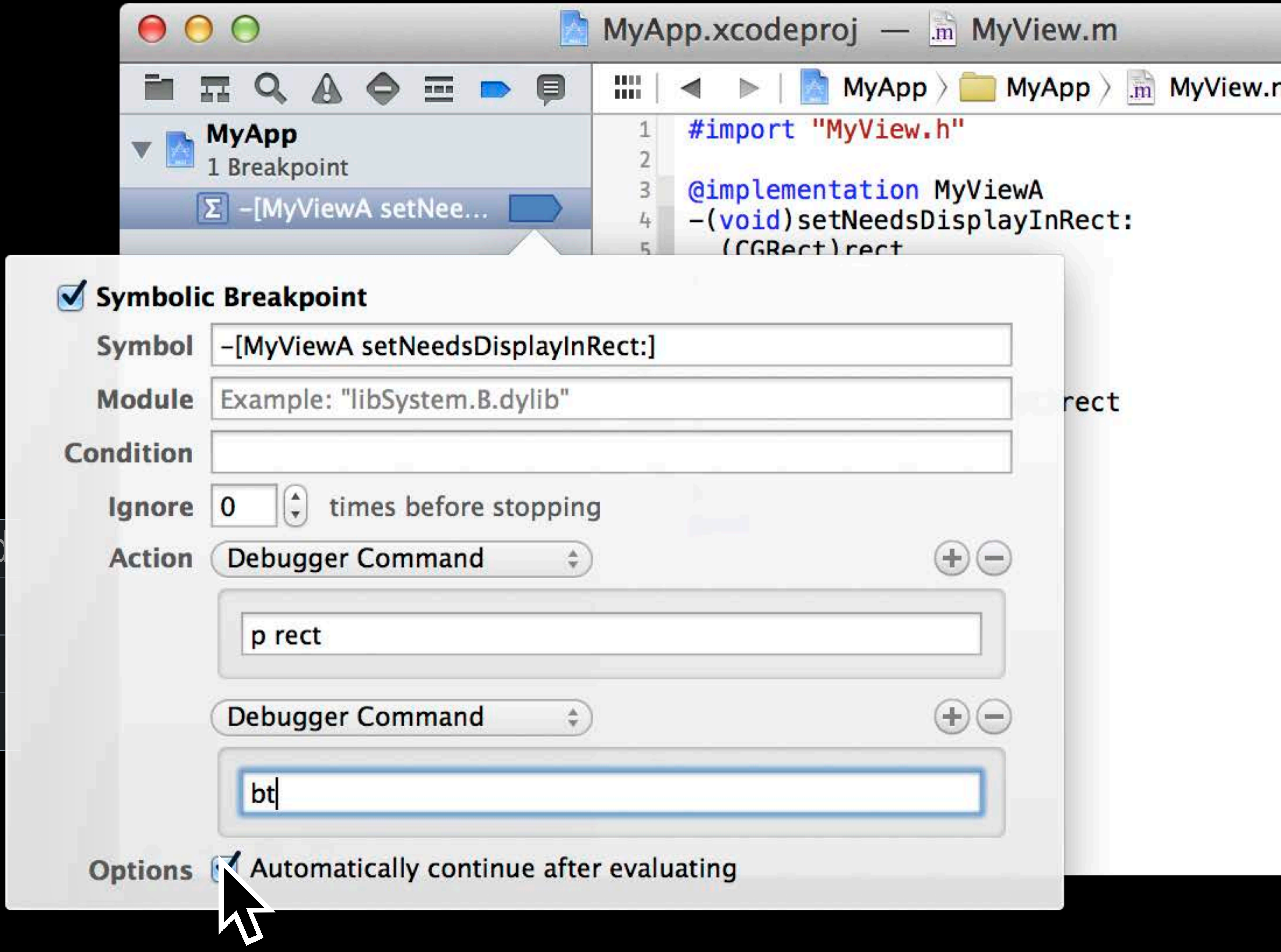
bt

Options: □ Automatically continue after evaluating

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
```



```
#import "MyView.h"

@implementation MyViewA
-(void)setNeedsDisplayInRect:
  (CGRect)rect
```

☑ Symbolic Breakpoint

Symbol: –[MyViewA setNeedsDisplayInRect:]

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

Action: Debugger Command

p rect

Debugger Command

bt

Options: ☑ Automatically continue after evaluating

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
```

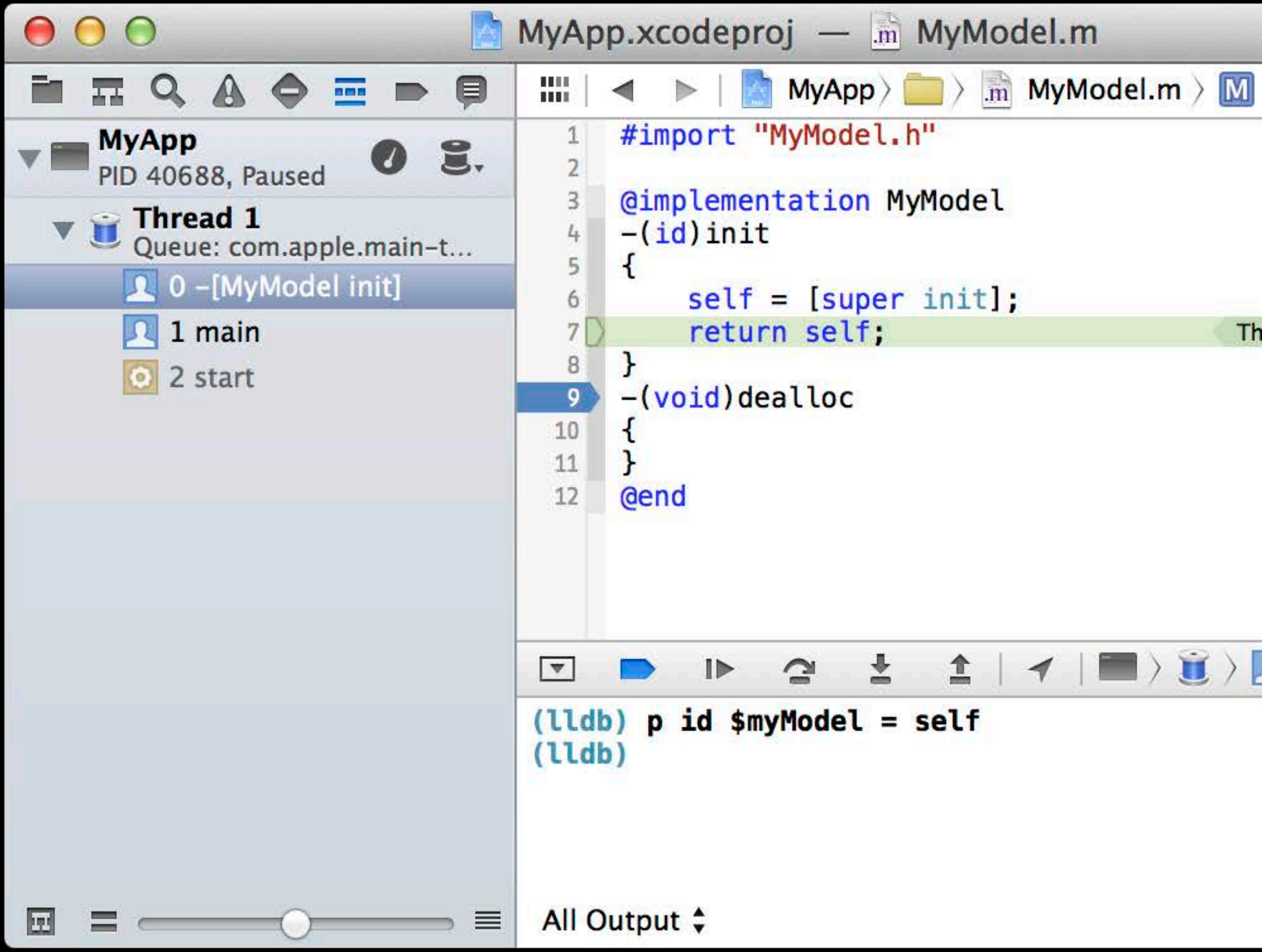
Creates a persistent variable of type id

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

```
b “–[MyModel dealloc]”
```

```
br m –c “self == $myModel”
```

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



```
#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
```

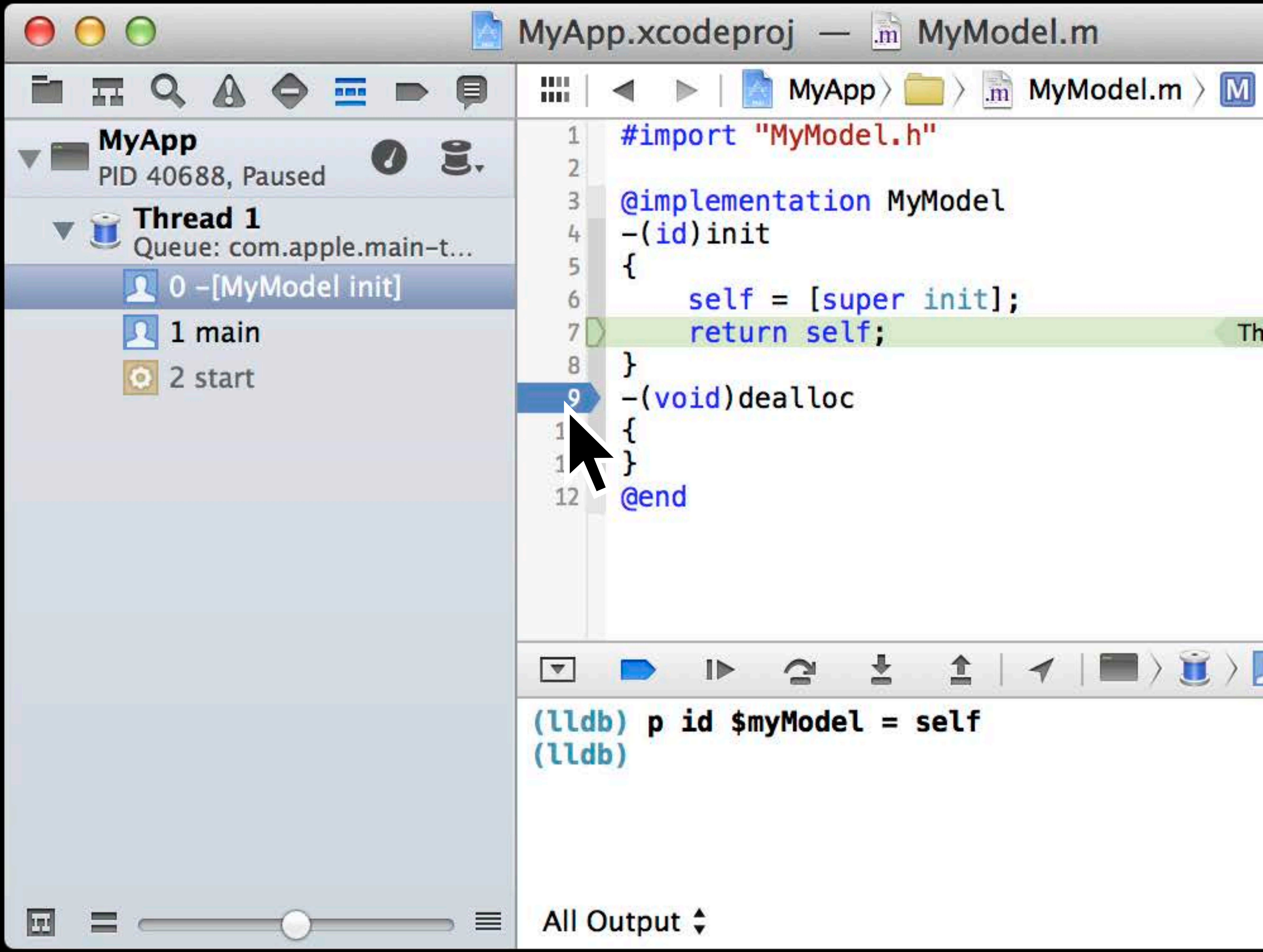
Creates a persistent variable of type id

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

```
b “-[MyModel dealloc]”
```

```
br m -c “self == $myModel”
```

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



```
#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
```

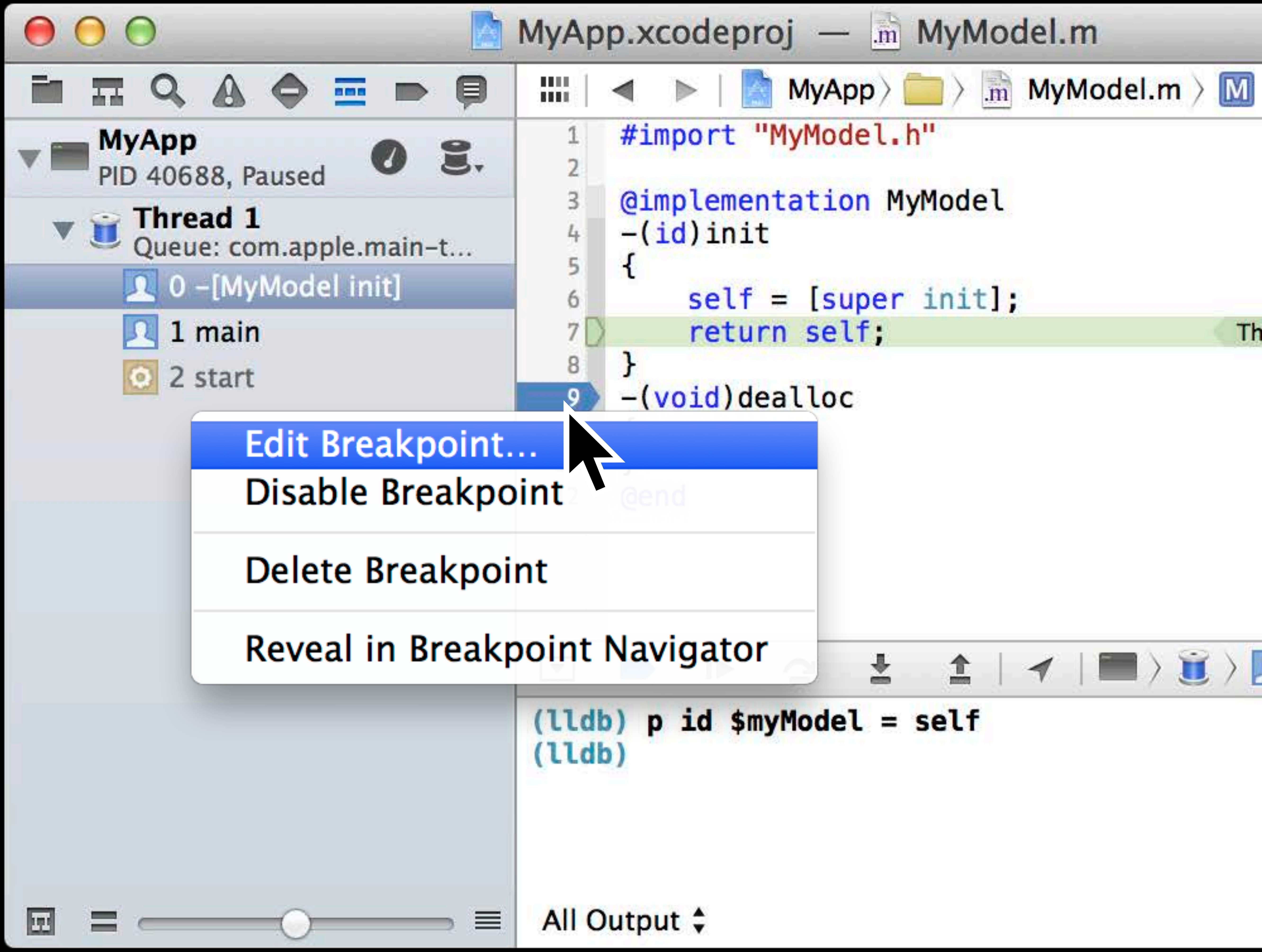
Creates a persistent variable of type id

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

```
b “-[MyModel dealloc]”
```

```
br m -c “self == $myModel”
```

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



```
#import "MyModel.h"

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

```
(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
```

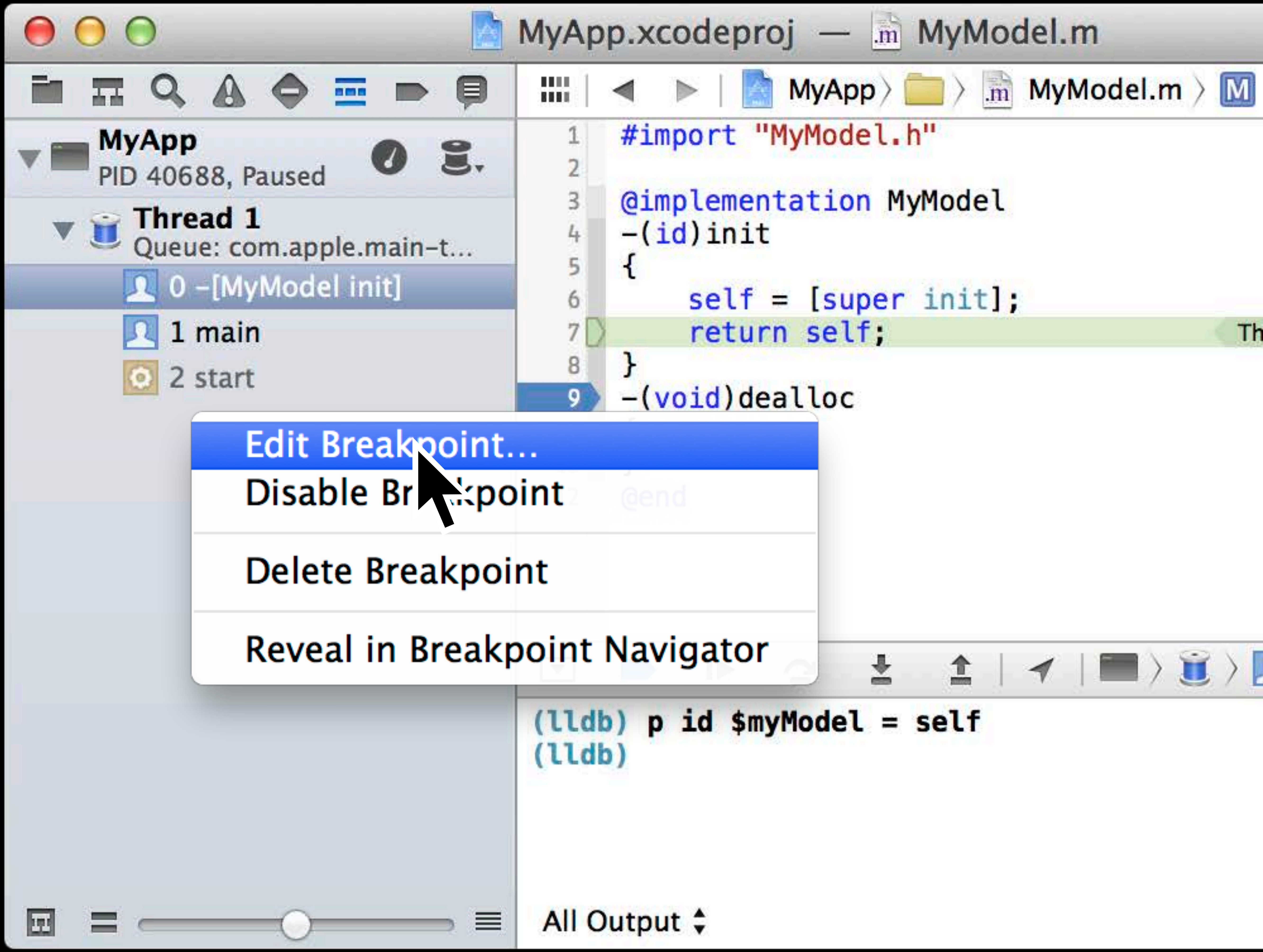
Creates a persistent variable of type id

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

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

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

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



```
#import "MyModel.h"

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

```
(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
```

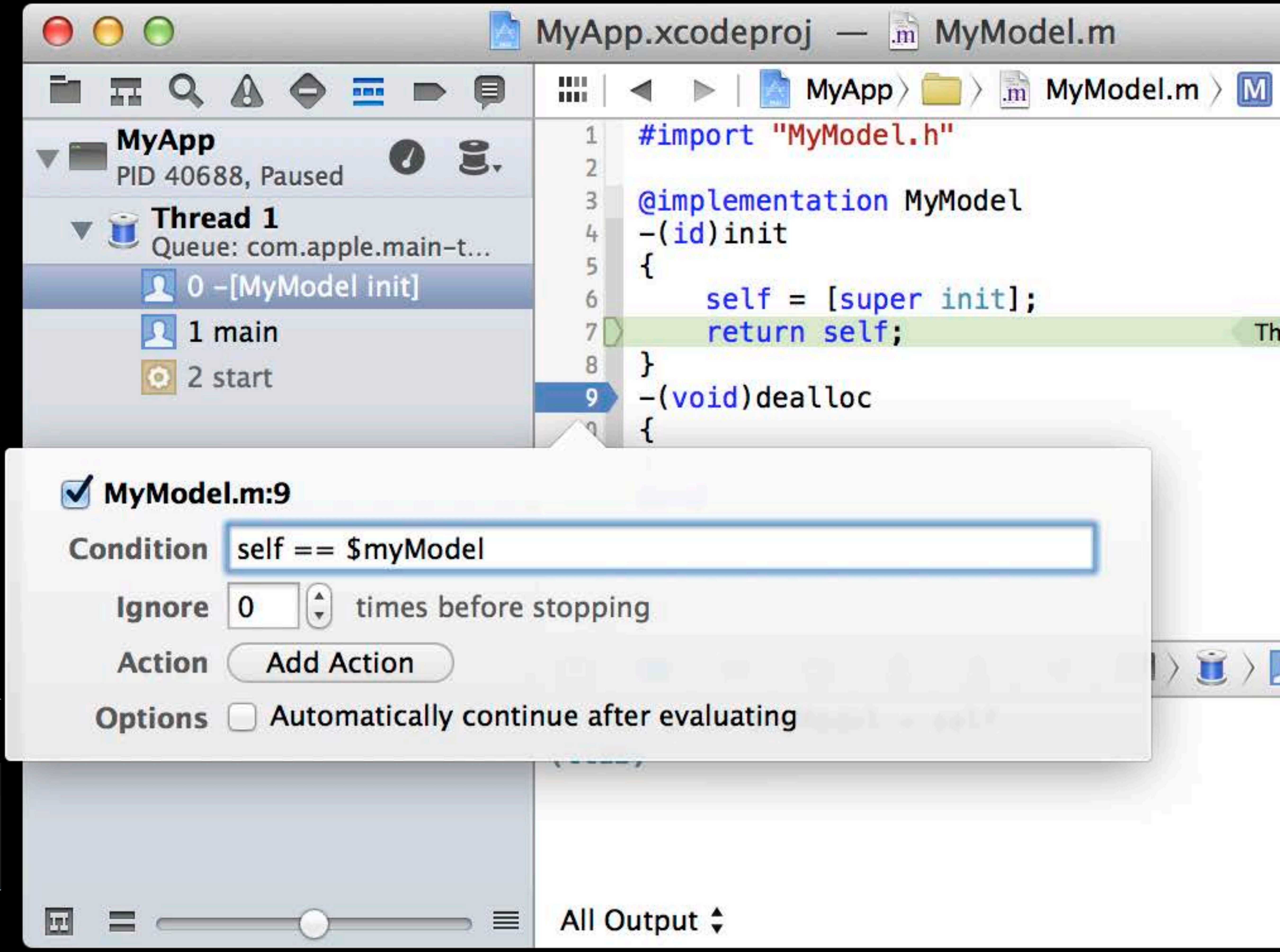
Creates a persistent variable of type id

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

```
b “–[MyModel dealloc]”
```

```
br m –c “self == $myModel”
```

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



```
#import "MyModel.h"

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

☑ MyModel.m:9

Condition: self == $myModel

Ignore: 0 times before stopping

Action: Add Action

Options: □ Automatically continue after evaluating

Focus on Memory with Watchpoints

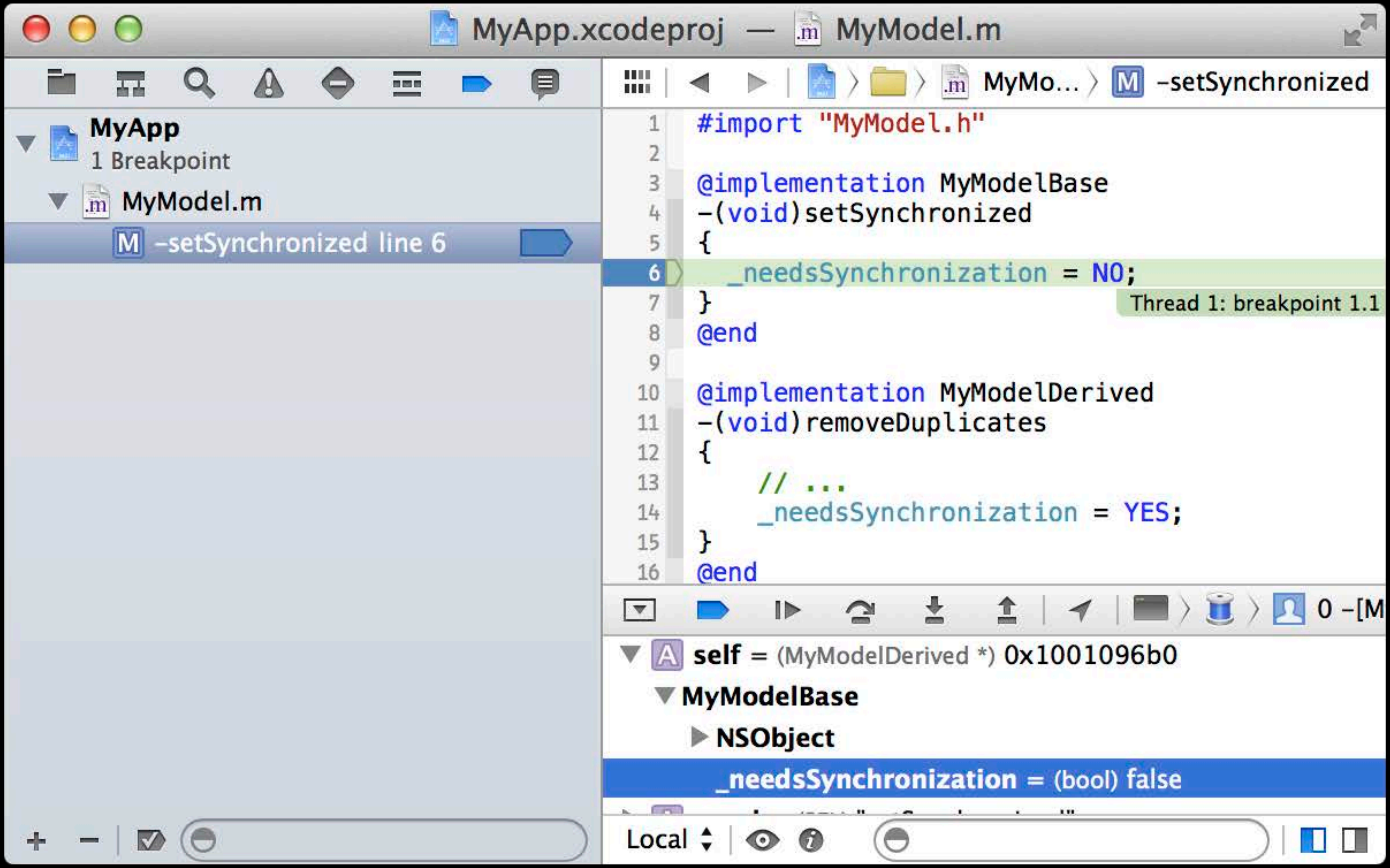
- Someone is changing a value, but all you know is its location
- Watchpoints pause the program if the value is accessed

```
w s v self->_needsSynchronization
```

```
watchpoint set variable
  self->_needsSynchronization
```

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

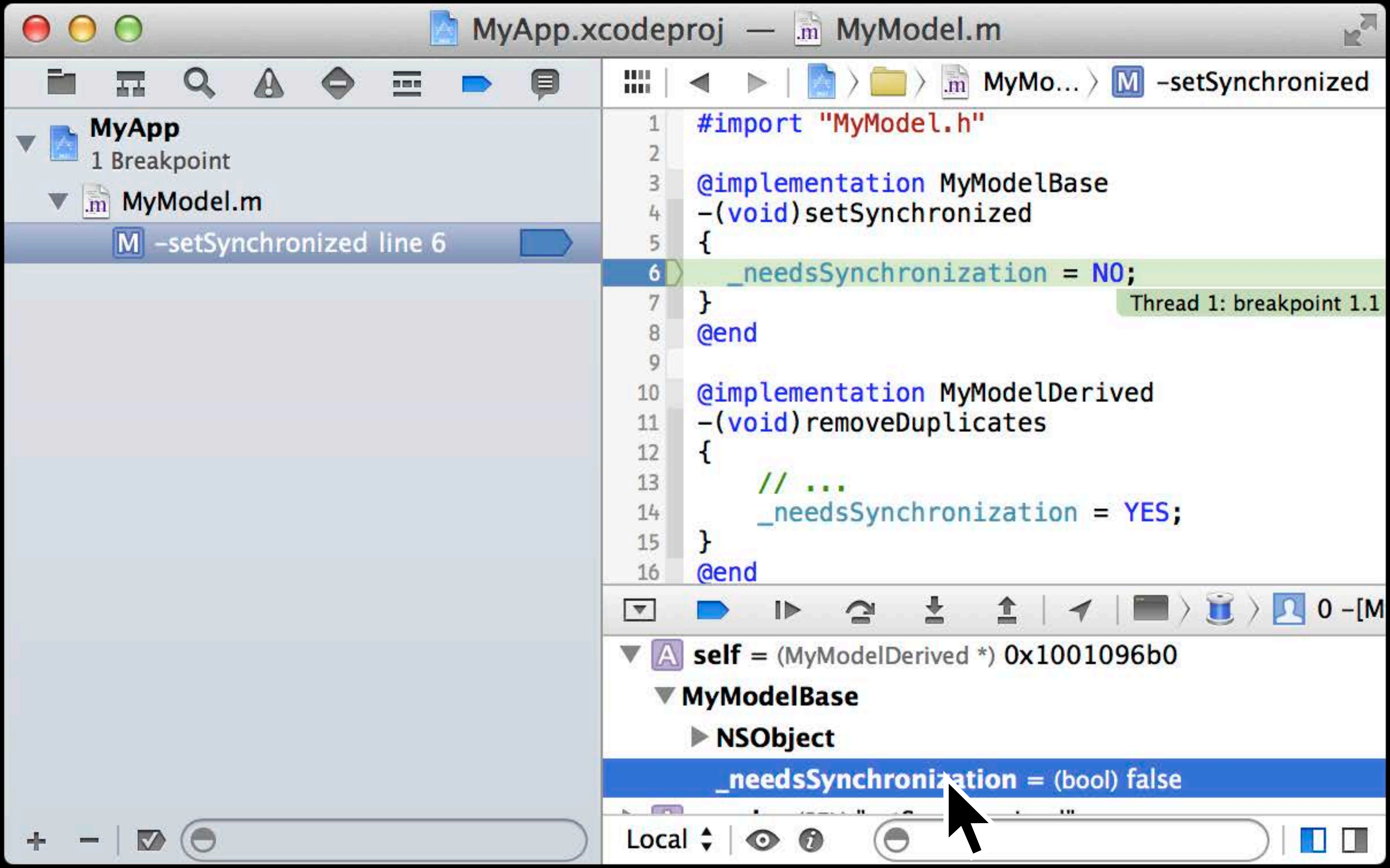
Focus on Memory with Watchpoints



```
#import "MyModel.h"

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

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

```
▼ self = (MyModelDerived *) 0x1001096b0
  ▼ MyModelBase
    ▶ NSObject
      _needsSynchronization = (bool) false
```

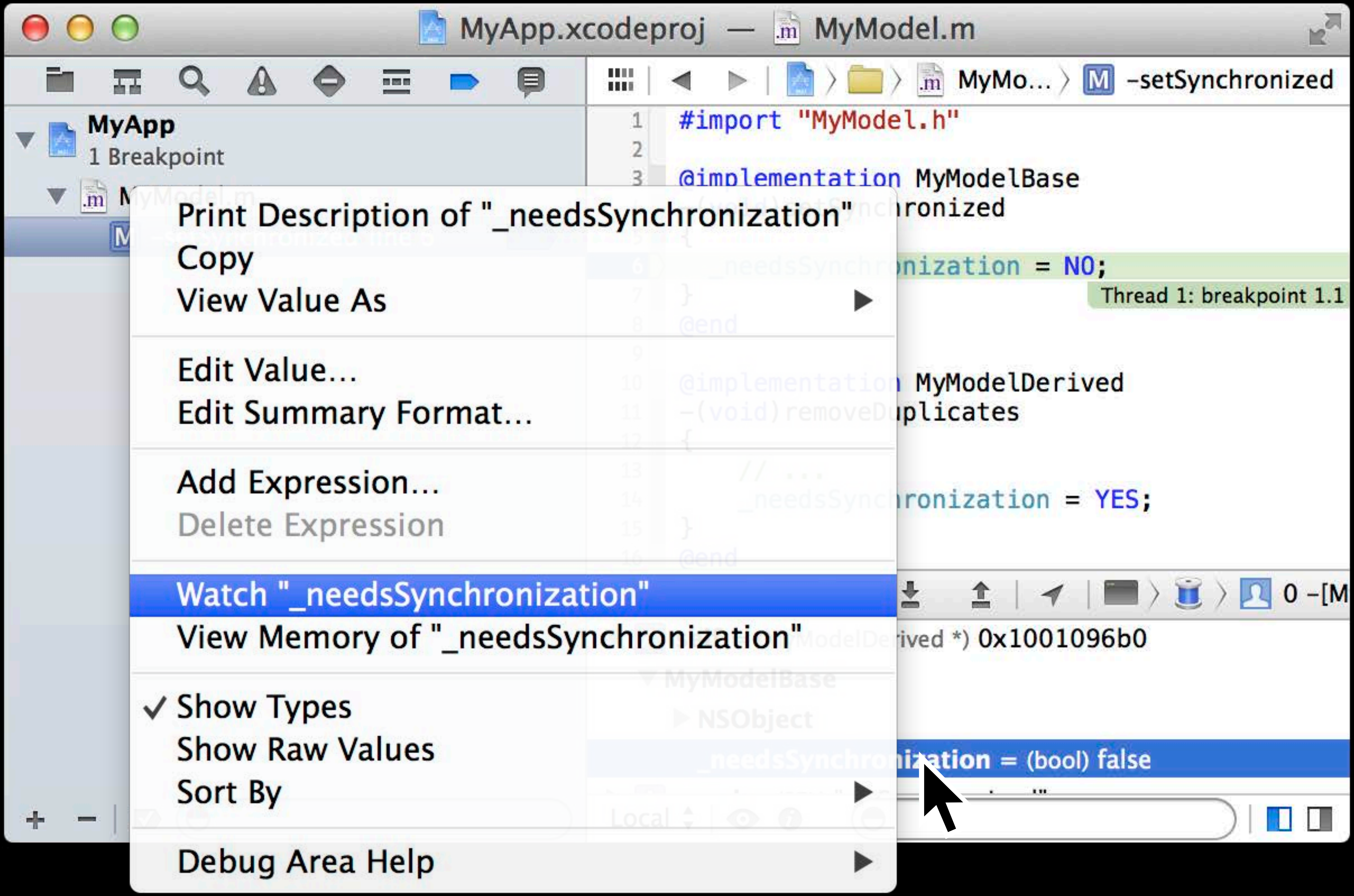
Focus on Memory with Watchpoints



```
#import "MyModel.h"

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

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

```
self = (MyModelDerived *) 0x1001096b0
  MyModelBase
    NSObject
    _needsSynchronization = (bool) false
```

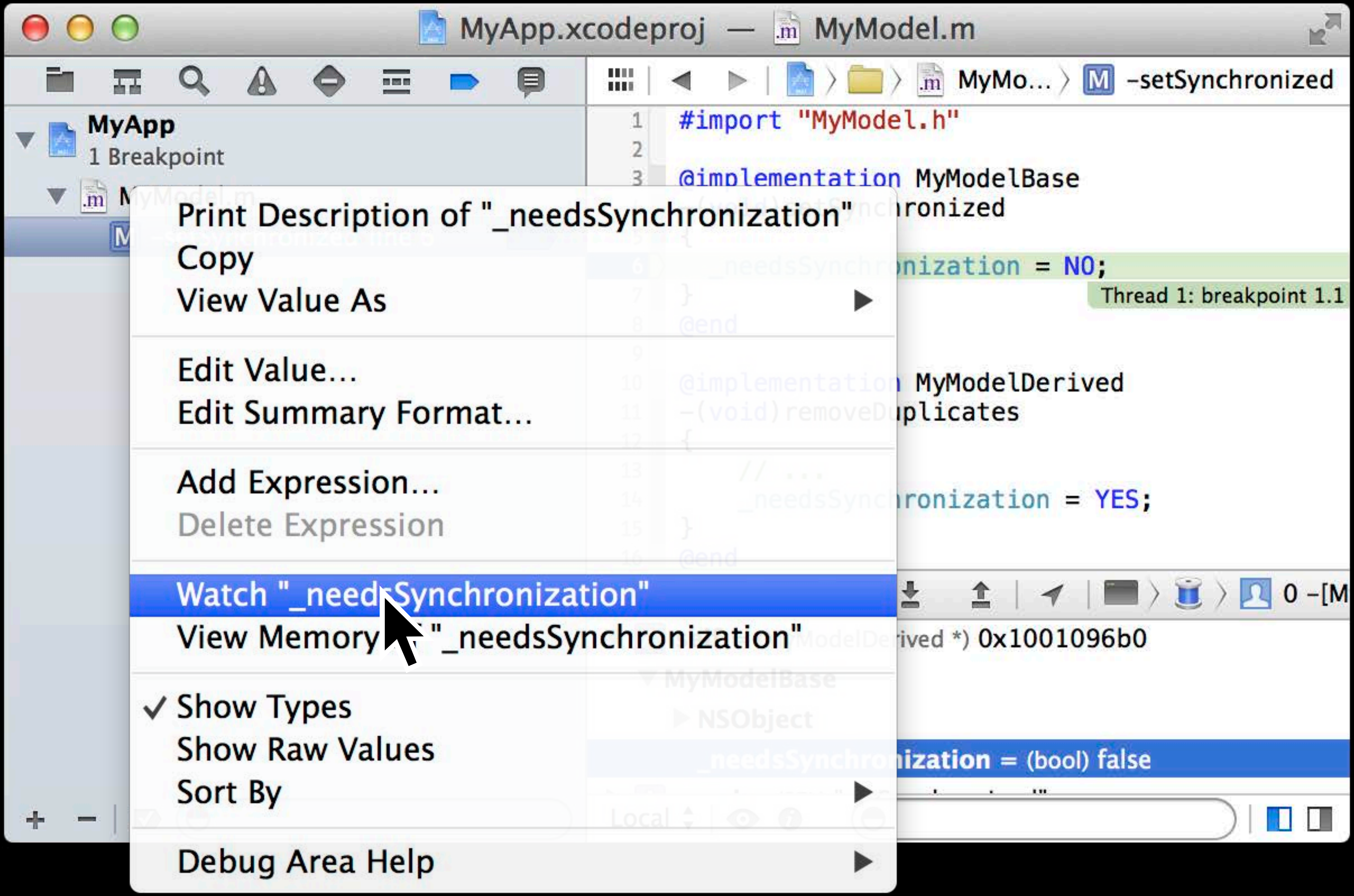
Focus on Memory with Watchpoints



```
#import "MyModel.h"

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

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

```
(MyModelDerived *) 0x1001096b0
▼ MyModelBase
  ▶ NSObject
    _needsSynchronization = (bool) false
```

Focus on Memory with Watchpoints



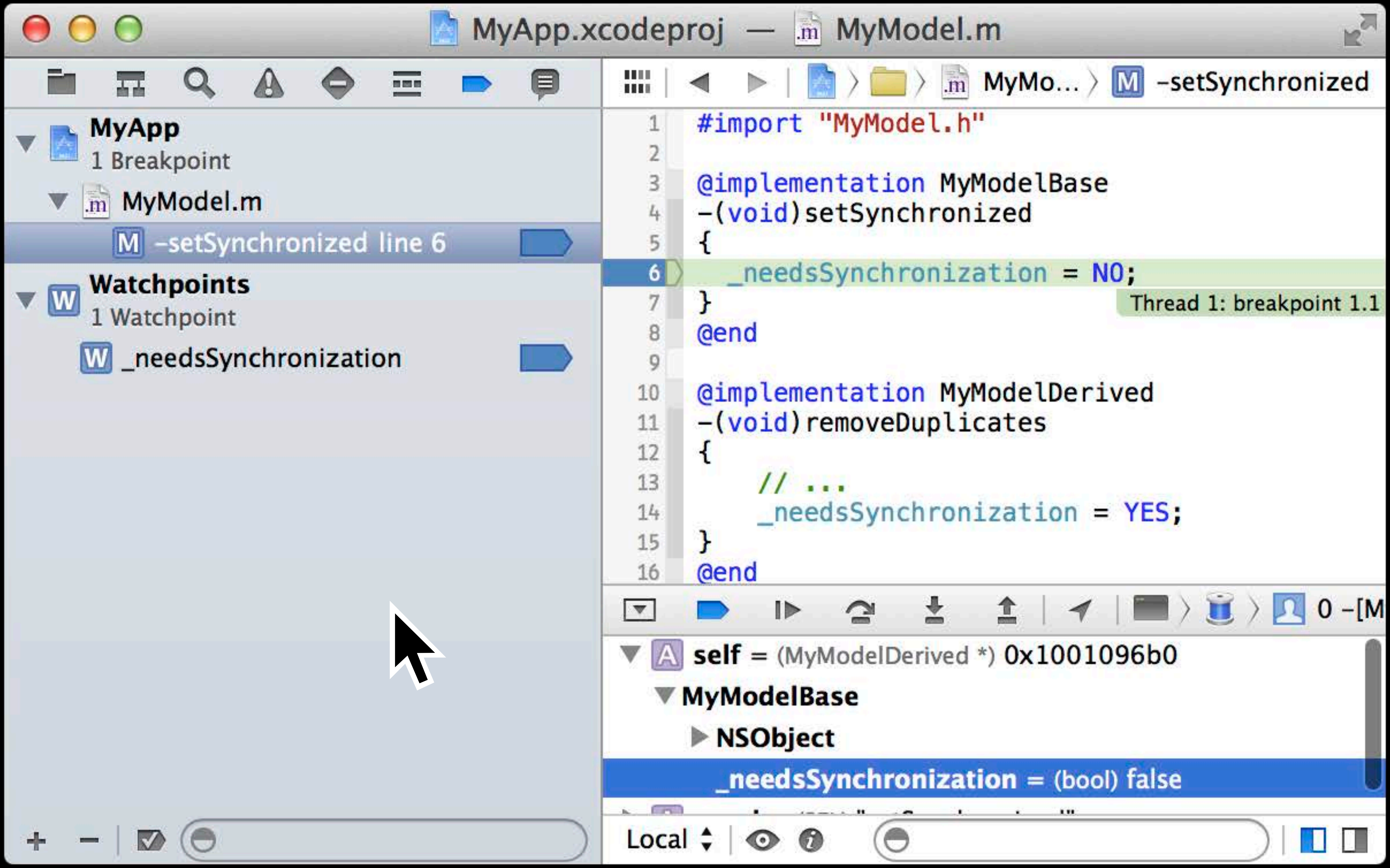
```
#import "MyModel.h"

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

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

Thread 1: breakpoint 1.1

(MyModelDerived *) 0x1001096b0

_needsSynchronization = (bool) false

Focus on Memory with Watchpoints



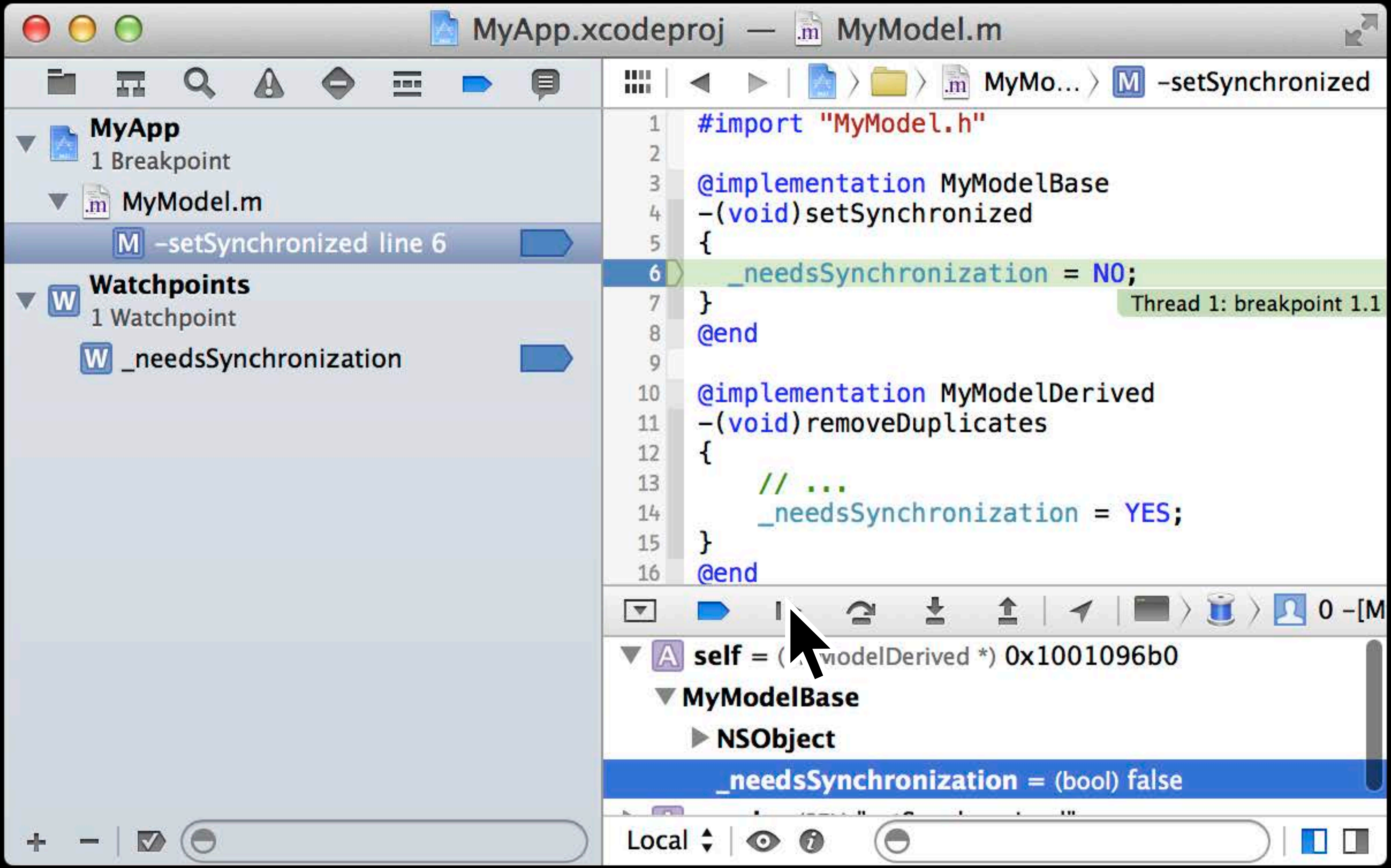
MyApp.xcodeproj — MyModel.m

```
 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
```

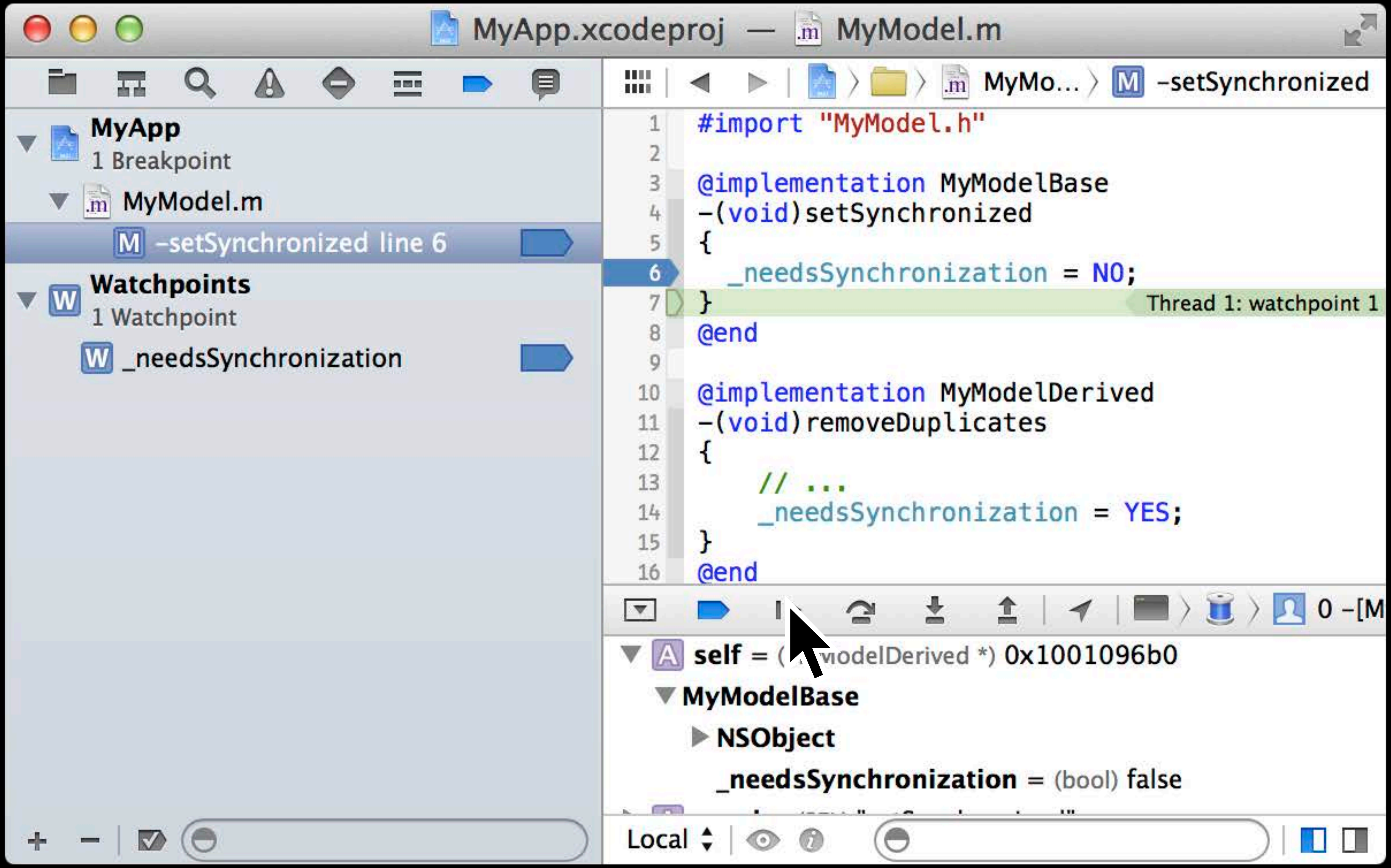
Focus on Memory with Watchpoints



```
#import "MyModel.h"

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

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

```
▼ self = (MyModelDerived *) 0x1001096b0
  ▼ MyModelBase
    ▶ NSObject
      _needsSynchronization = (bool) false
```

Focus on Memory with Watchpoints



MyApp.xcodeproj — MyModel.m

MyApp
1 Breakpoint
MyModel.m
-setSynchronized line 6

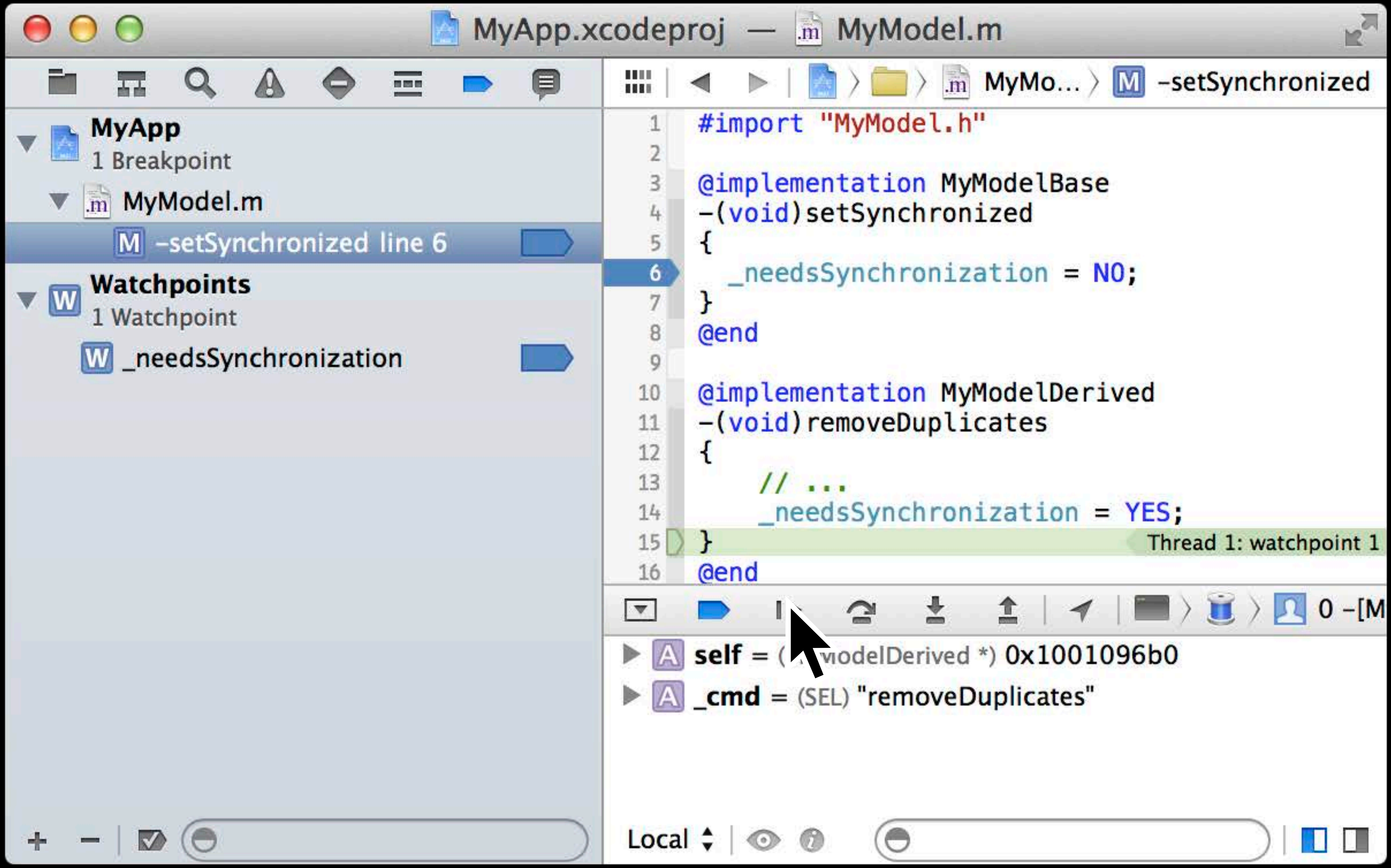
Watchpoints
1 Watchpoint
_needsSynchronization

```
#import "MyModel.h"

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

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

Thread 1: watchpoint 1

```
self = (MyModelDerived *) 0x1001096b0
 MyModelBase
  NSObject
  _needsSynchronization = (bool) false
```

Focus on Memory with Watchpoints



```
#import "MyModel.h"

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

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

```
self = (MyModelDerived *) 0x1001096b0
_cmd = (SEL) "removeDuplicates"
```

Stepping Through Problems

Execution control without surprises

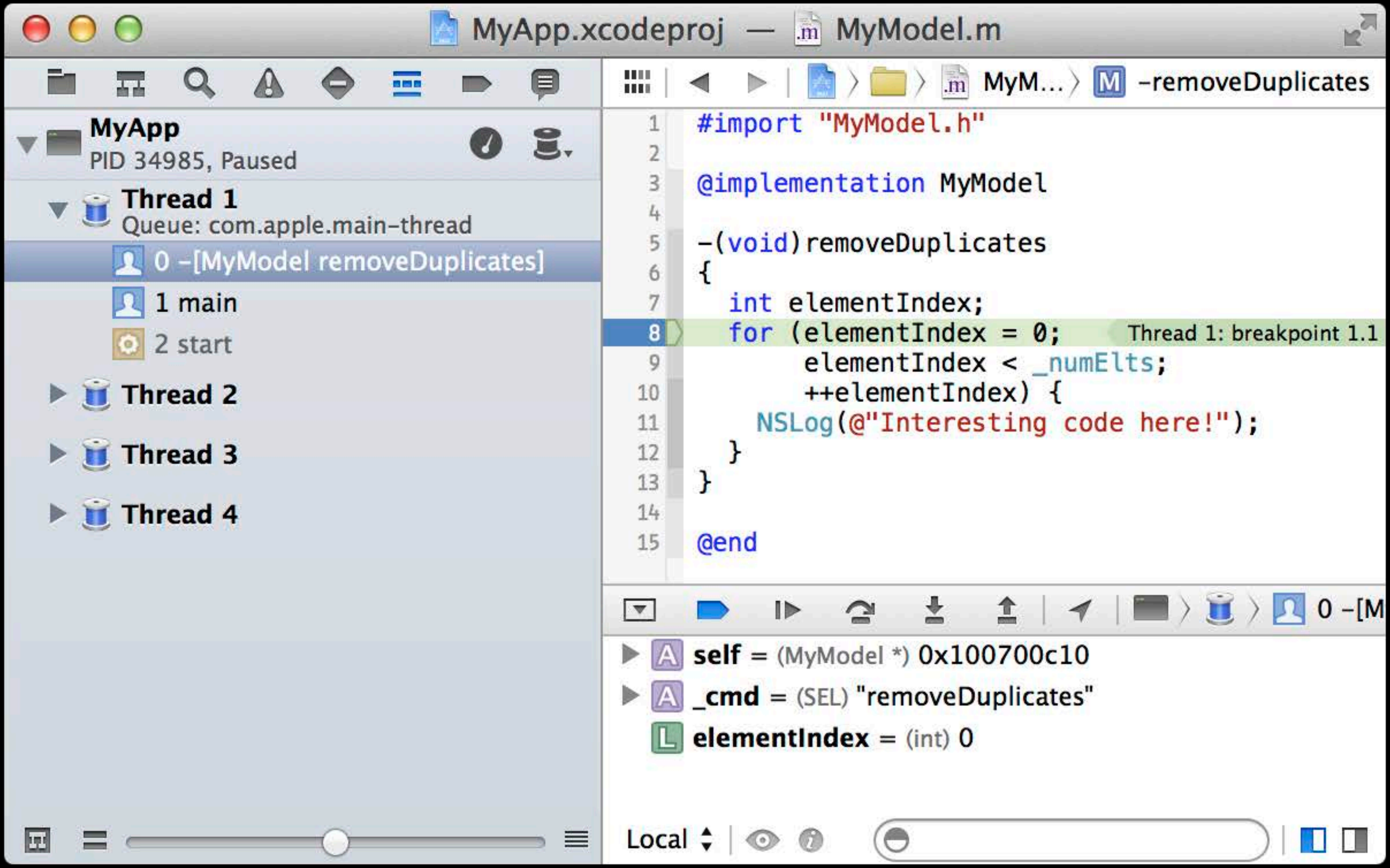
Avoiding Repeated Steps

- 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

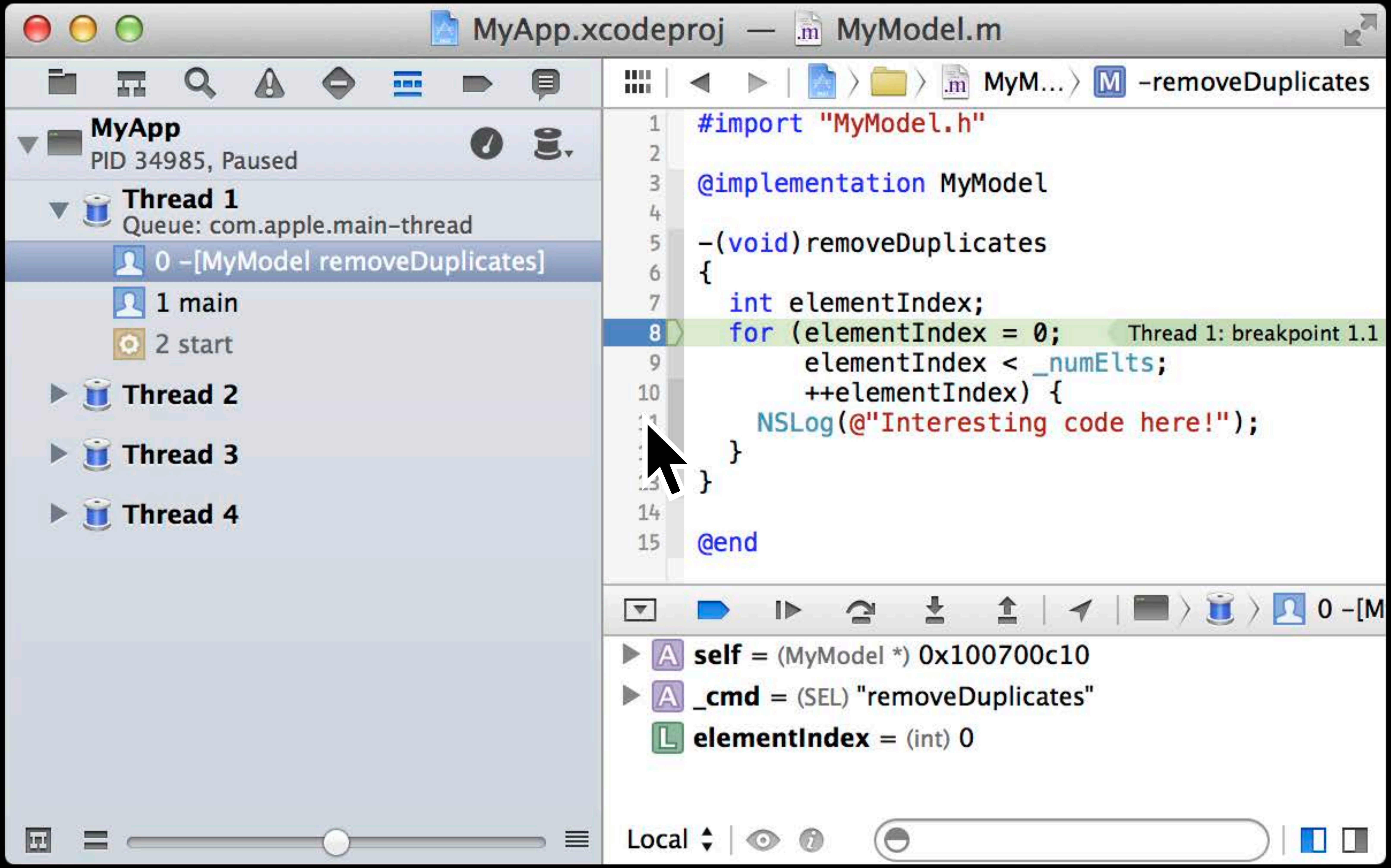
Avoiding Repeated Steps



```
#import "MyModel.h"

@implementation MyModel

-(void)removeDuplicates
{
  int elementIndex;
  for (elementIndex = 0;
         elementIndex < _numElts;
         ++elementIndex) {
    NSLog(@"Interesting code here!");
  }
}

@end
```

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

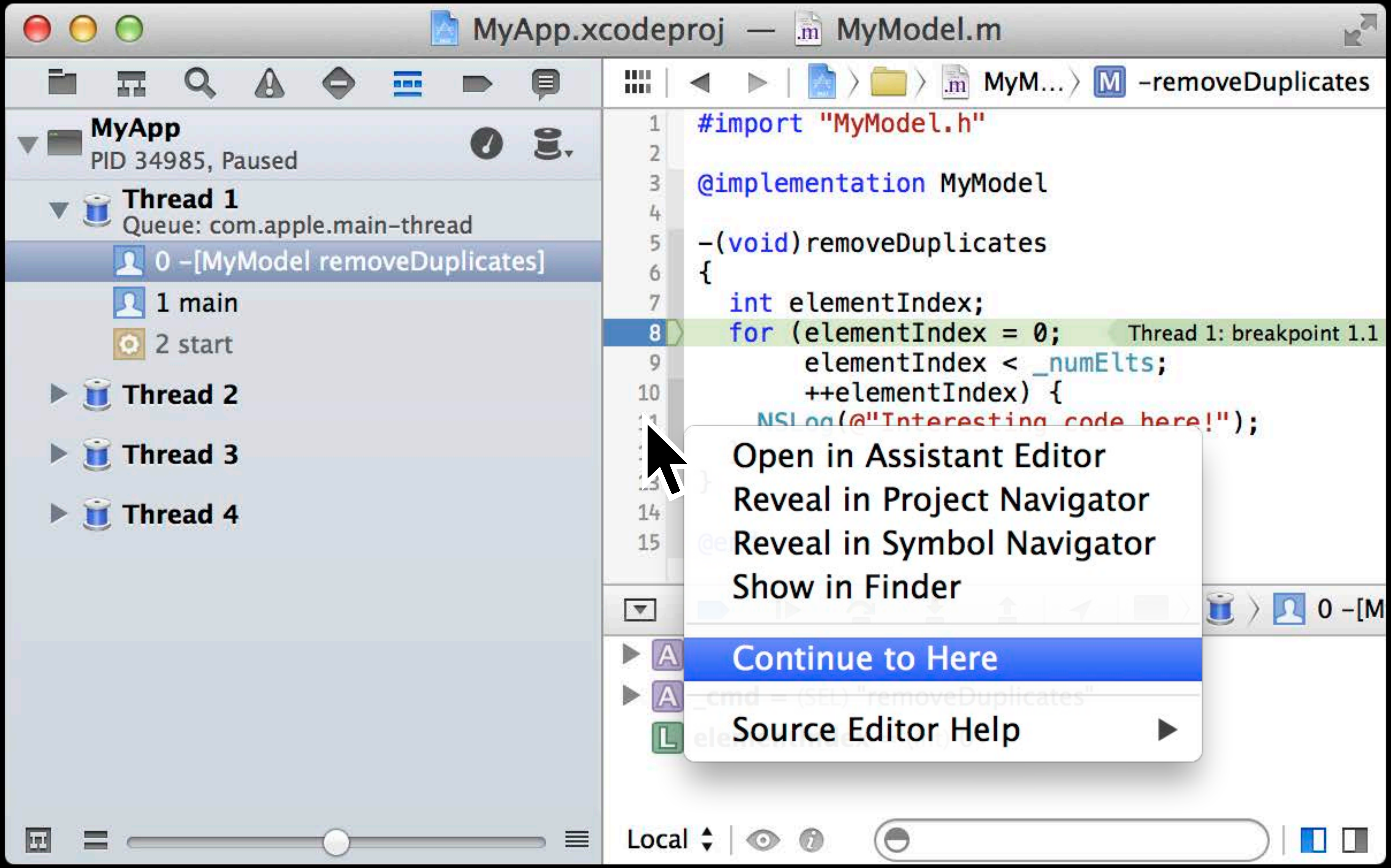
Avoiding Repeated Steps



```
#import "MyModel.h"

@implementation MyModel

-(void)removeDuplicates
{
  int elementIndex;
  for (elementIndex = 0;
       elementIndex < _numElts;
       ++elementIndex) {
    NSLog(@"Interesting code here!");
  }
}

@end
```

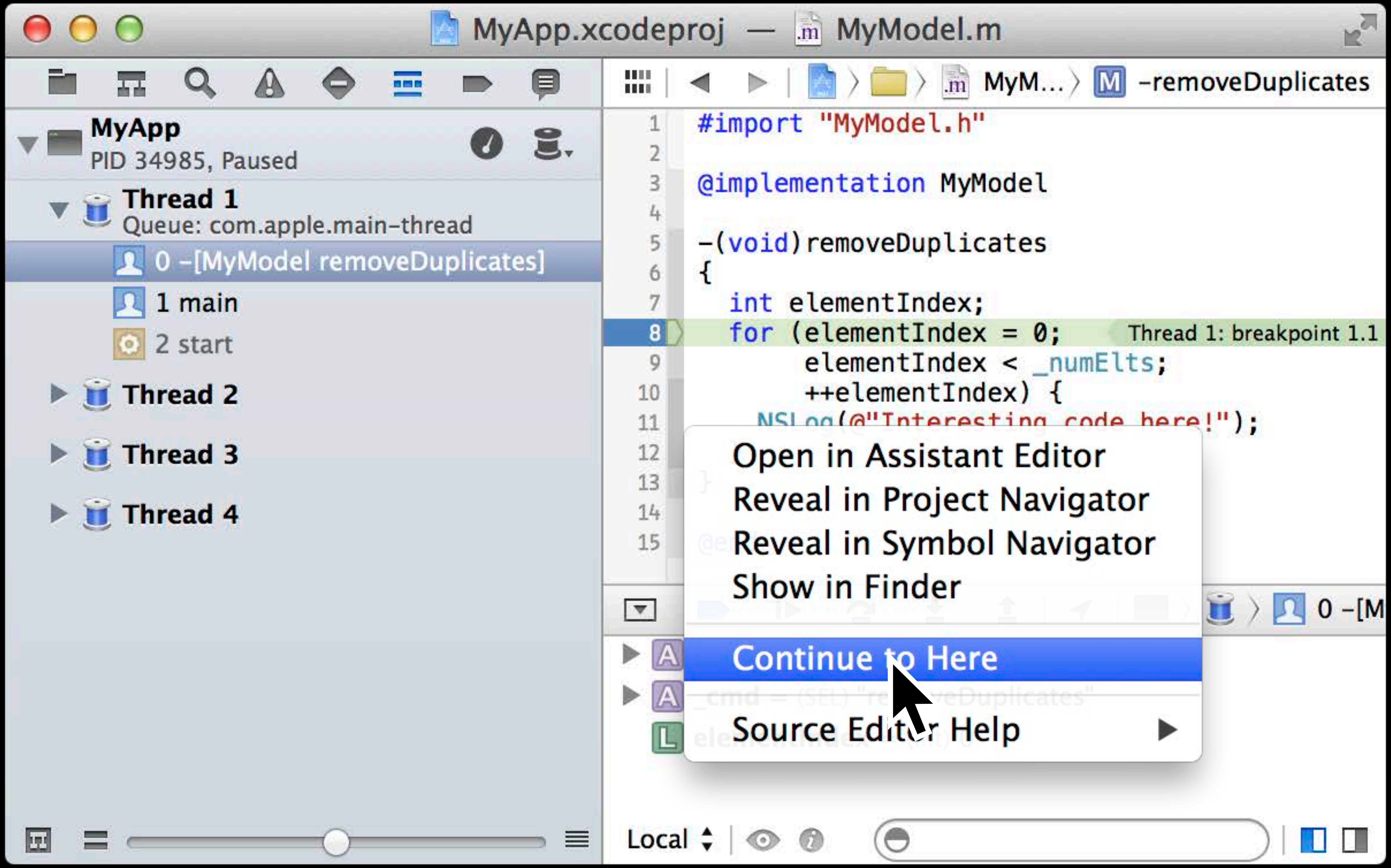
```
self = (MyModel *) 0x100700c10
_cmd = (SEL) "removeDuplicates"
elementIndex = (int) 0
```

Avoiding Repeated Steps



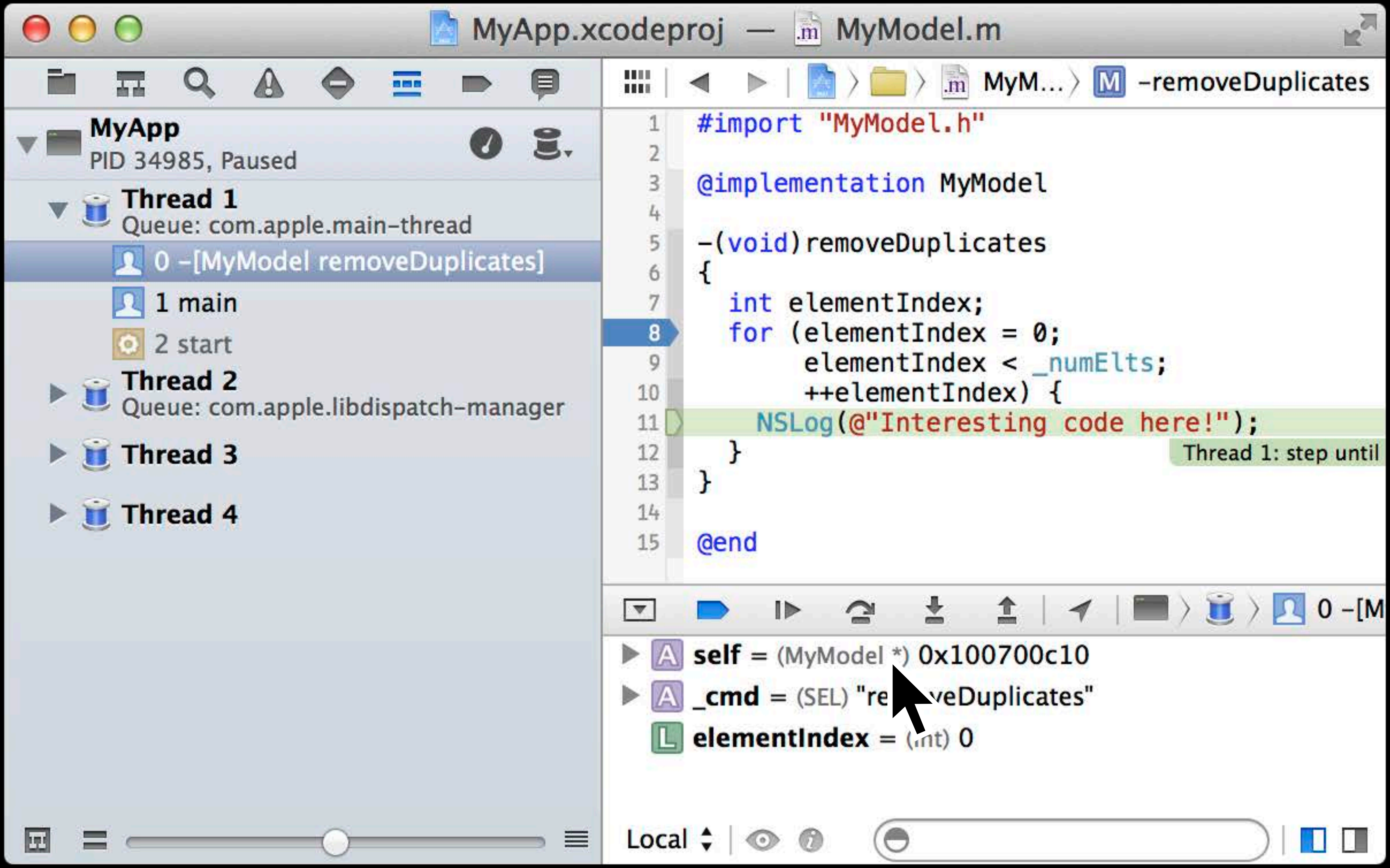
```
#import "MyModel.h"

@implementation MyModel

-(void)removeDuplicates
{
  int elementIndex;
  for (elementIndex = 0;
        elementIndex < _numElts;
        ++elementIndex) {
    NSLog(@"Interesting code here!");
```

Avoiding Repeated Steps



```
#import "MyModel.h"

@implementation MyModel

-(void)removeDuplicates
{
  int elementIndex;
  for (elementIndex = 0;
       elementIndex < _numElts;
       ++elementIndex) {
    NSLog(@"Interesting code here!");
```

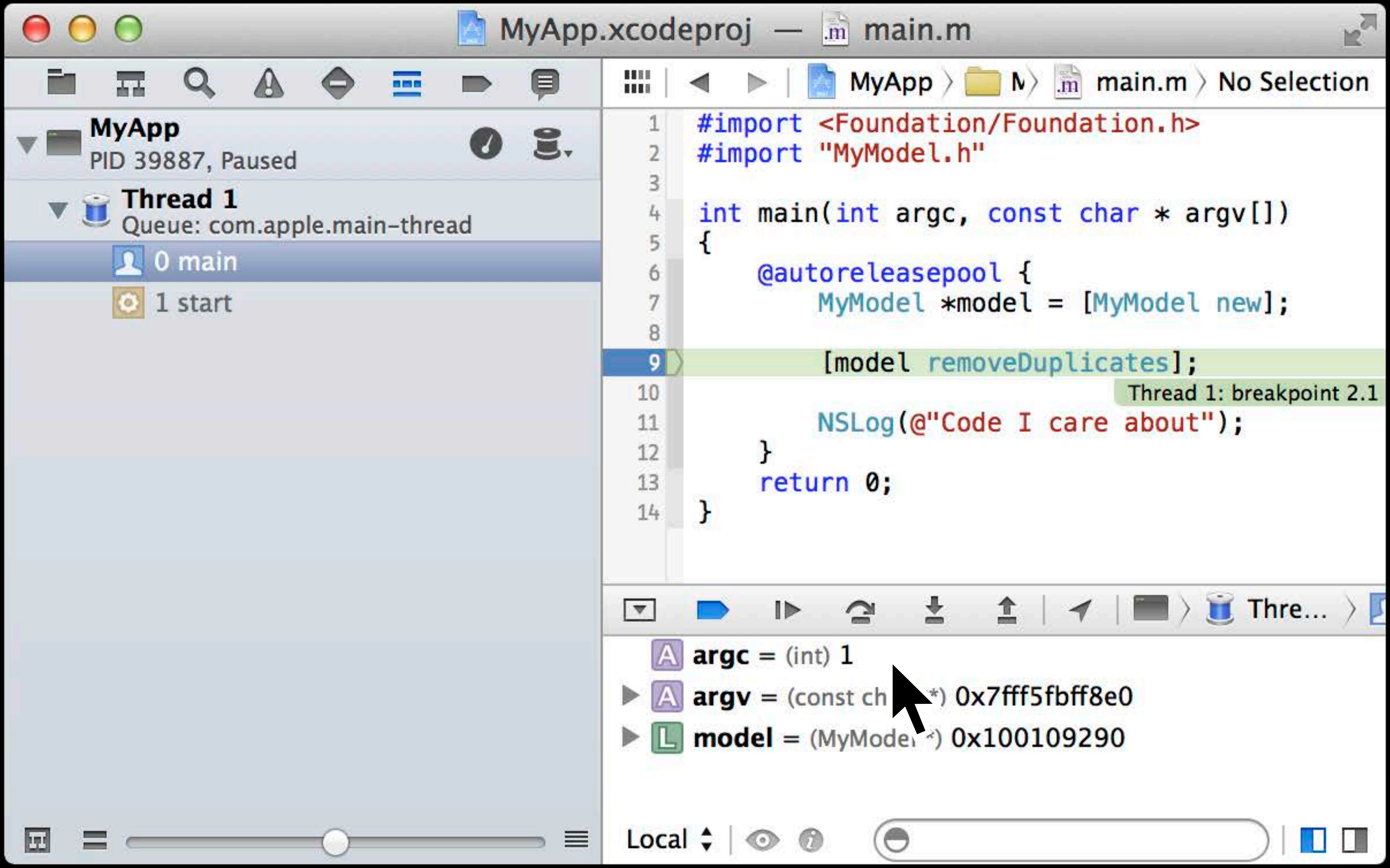
Avoiding Repeated Steps



```
#import "MyModel.h"

@implementation MyModel

-(void)removeDuplicates
{
  int elementIndex;
  for (elementIndex = 0;
       elementIndex < _numElts;
       ++elementIndex) {
    NSLog(@"Interesting code here!");
  }
}

@end
```

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

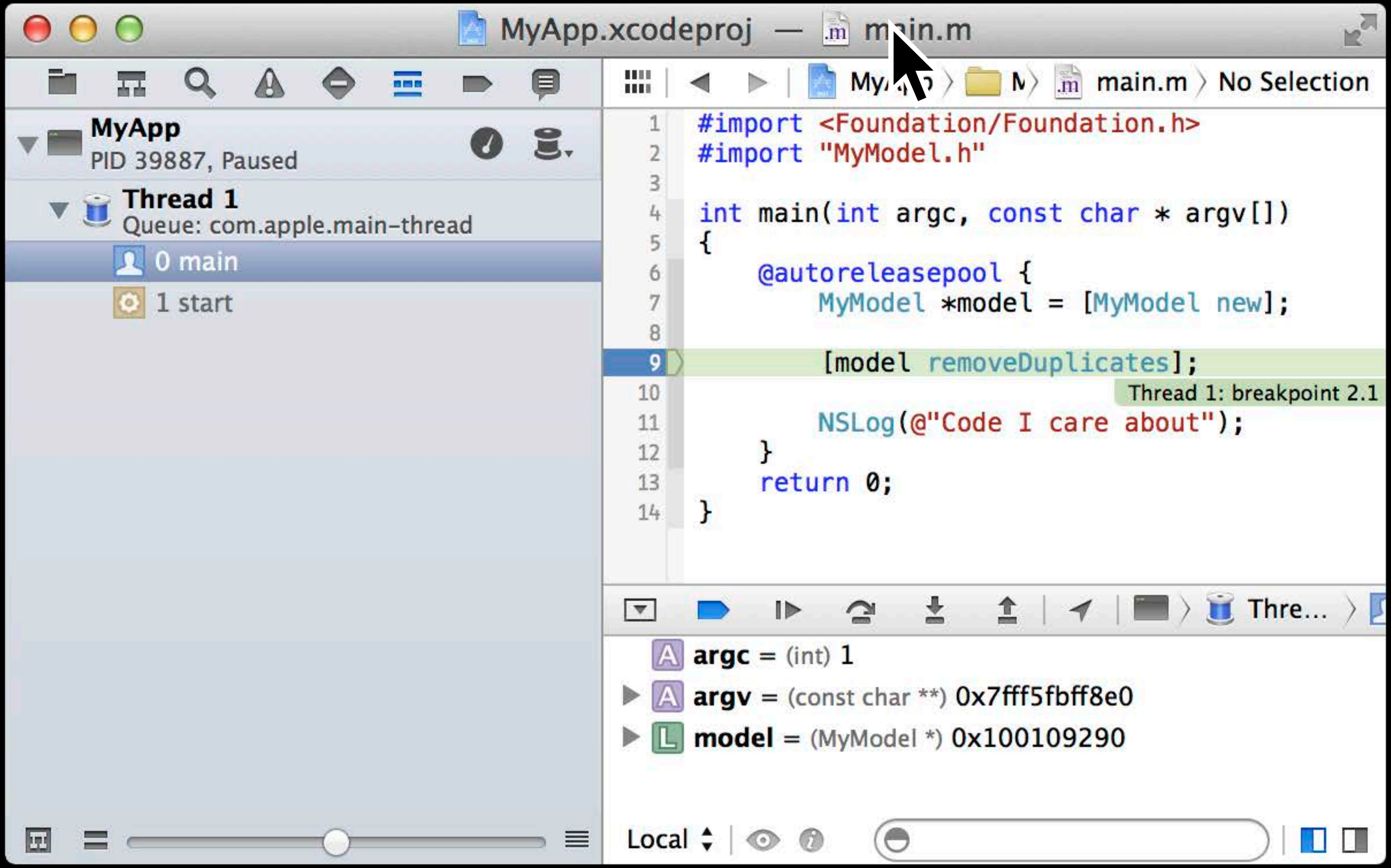
Hitting Breakpoints While Stepping



```
#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;
}
```

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

Hitting Breakpoints While Stepping



MyApp.xcodeproj — main.m

MyApp
PID 39887, Paused

Thread 1
Queue: com.apple.main-thread

0 main

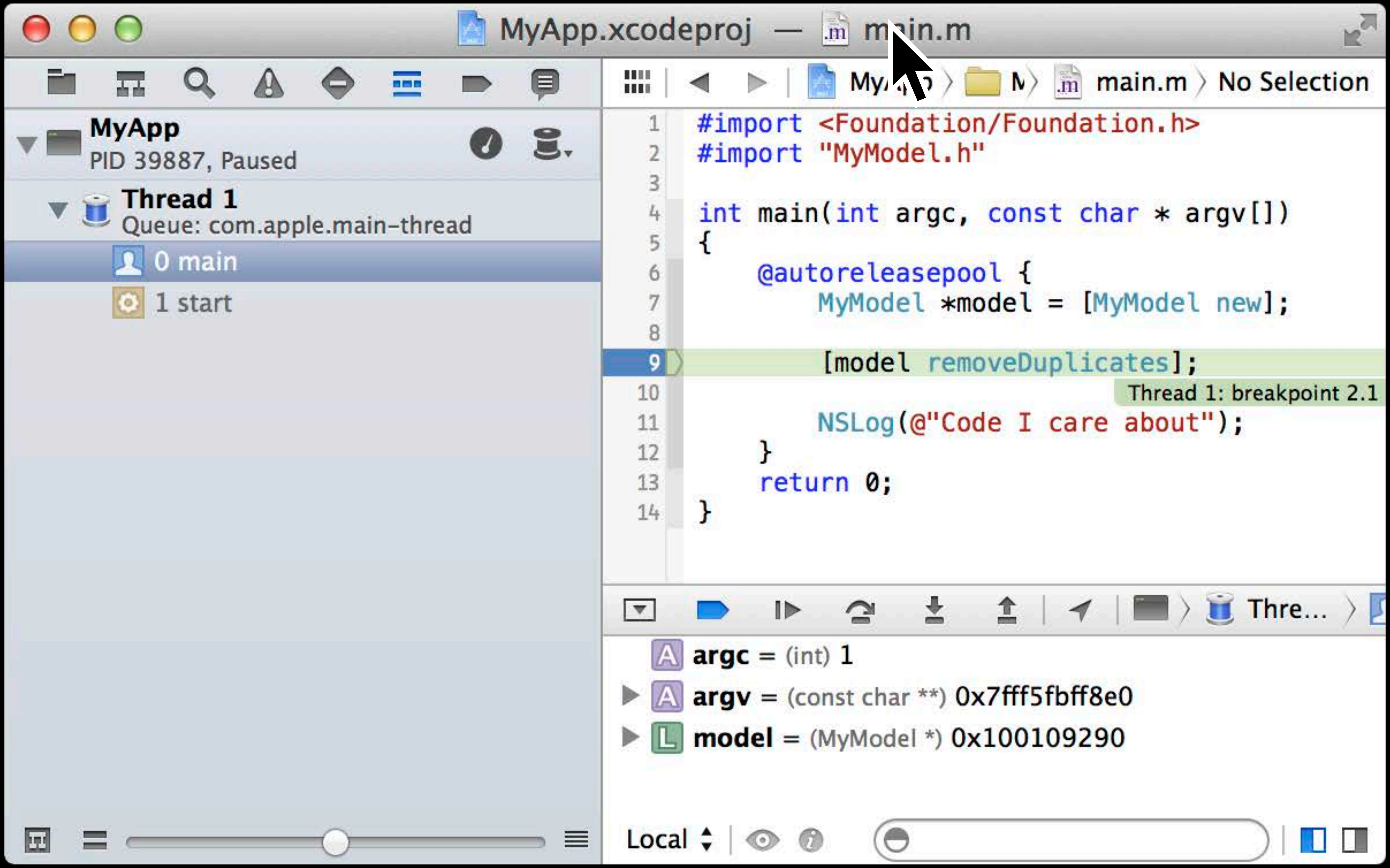
1 start

```
#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;
}
```

Thread 1: breakpoint 2.1

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

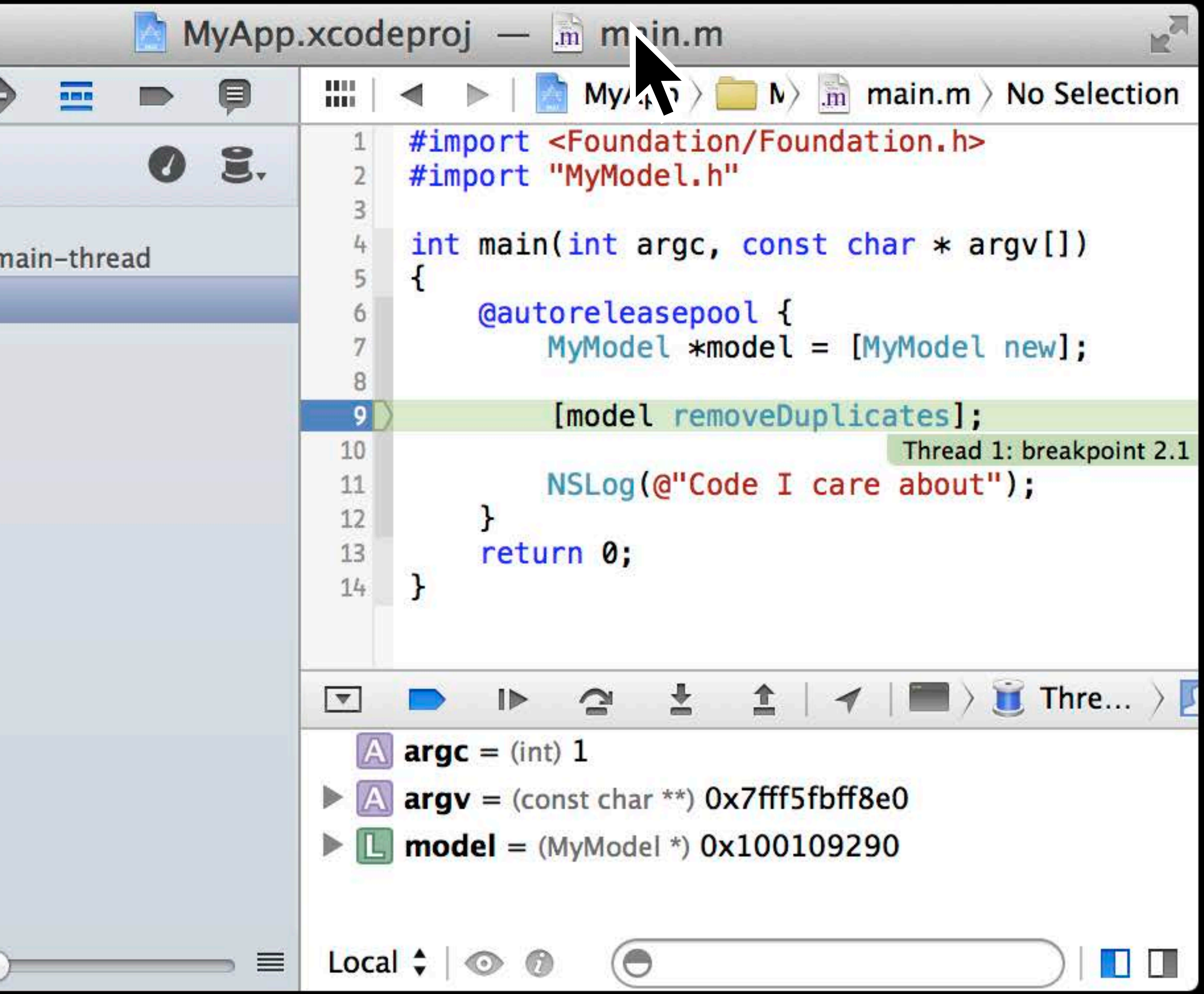
Hitting Breakpoints While Stepping



```
#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;
}
```

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

Hitting Breakpoints While Stepping

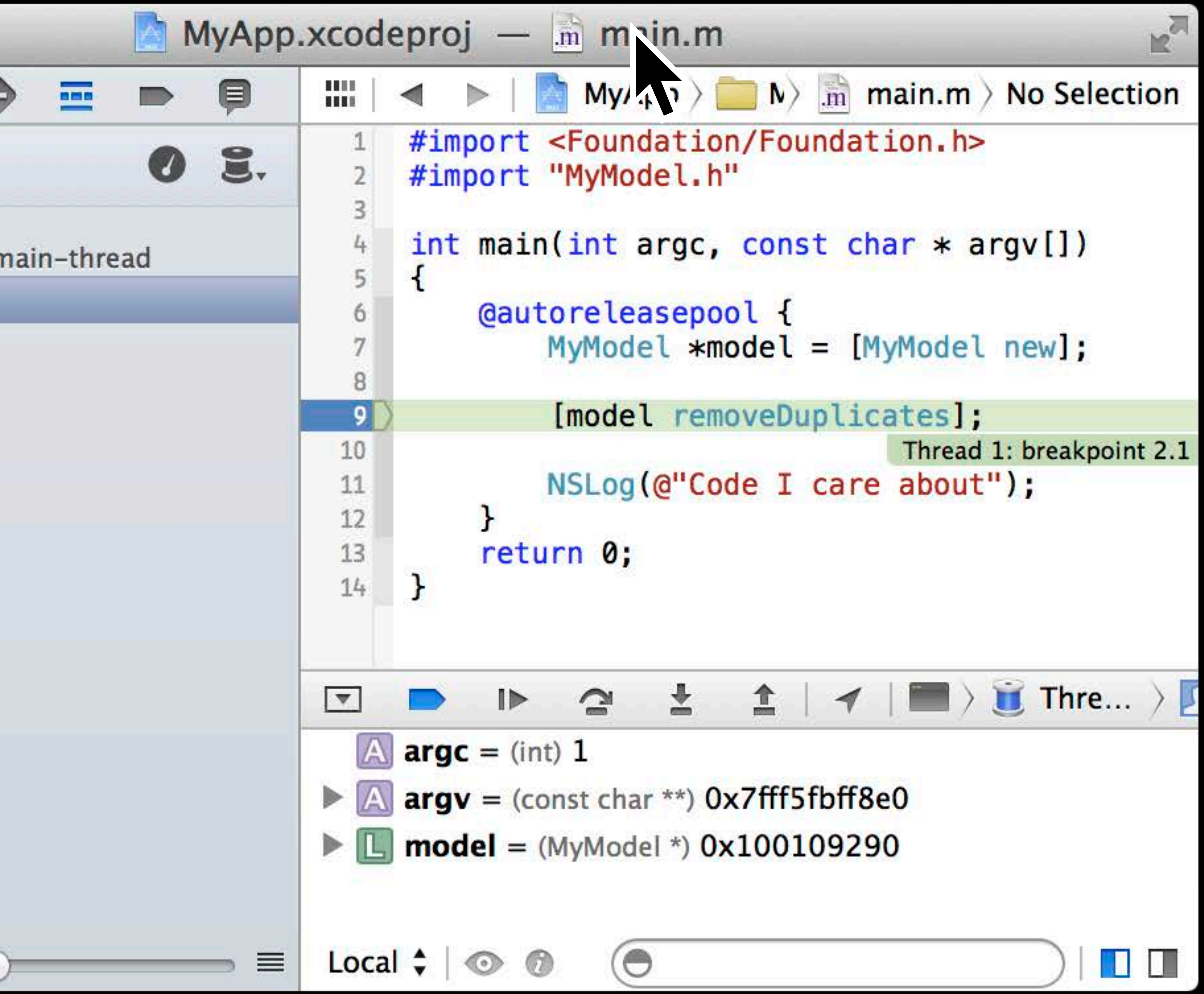


```
#import <Foundation/Foundation.h>
#import "MyModel.h"

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

        [model removeDuplicates];

        NSLog(@"Code I care about");
    }
    return 0;
}
```

Thread 1: breakpoint 2.1

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

Hitting Breakpoints While Stepping

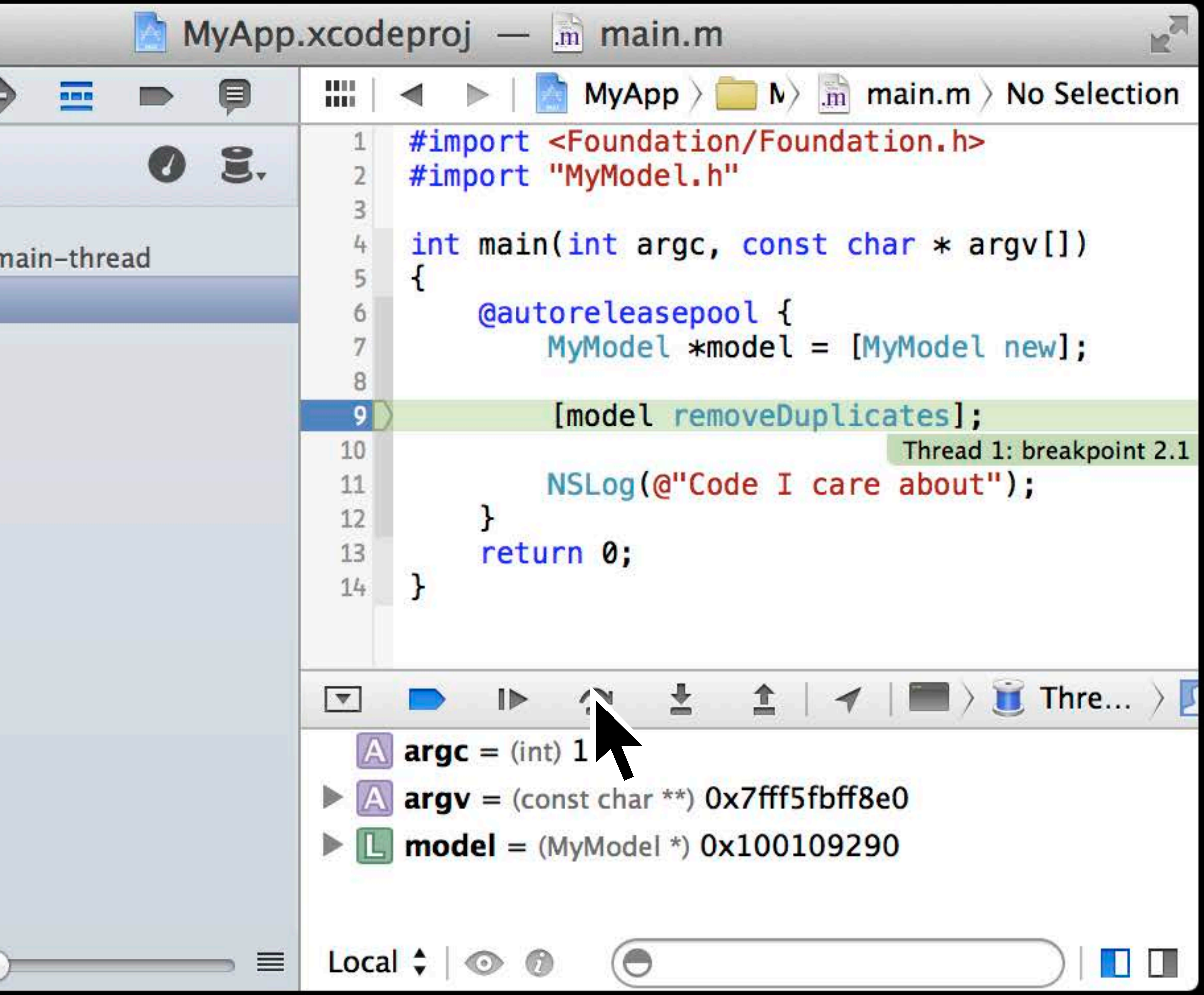


```
#import <Foundation/Foundation.h>
#import "MyModel.h"

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

        [model removeDuplicates];

        NSLog(@"Code I care about");
    }
    return 0;
}
```

```
argc = (int) 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

Hitting Breakpoints While Stepping

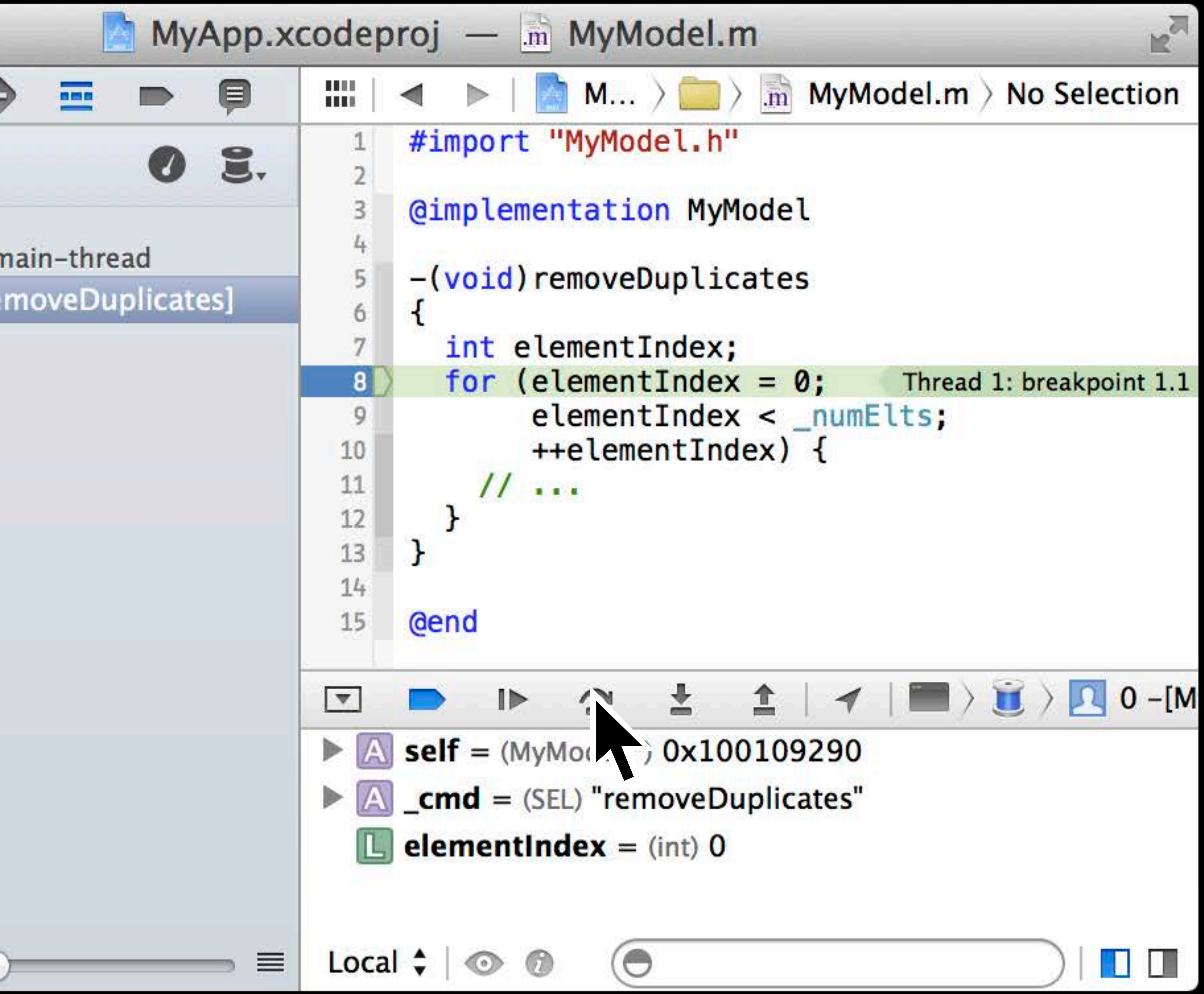


```
#import <Foundation/Foundation.h>
#import "MyModel.h"

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

        [model removeDuplicates];

        NSLog(@"Code I care about");
    }
    return 0;
}
```

```
argc = (int) 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

Hitting Breakpoints While Stepping

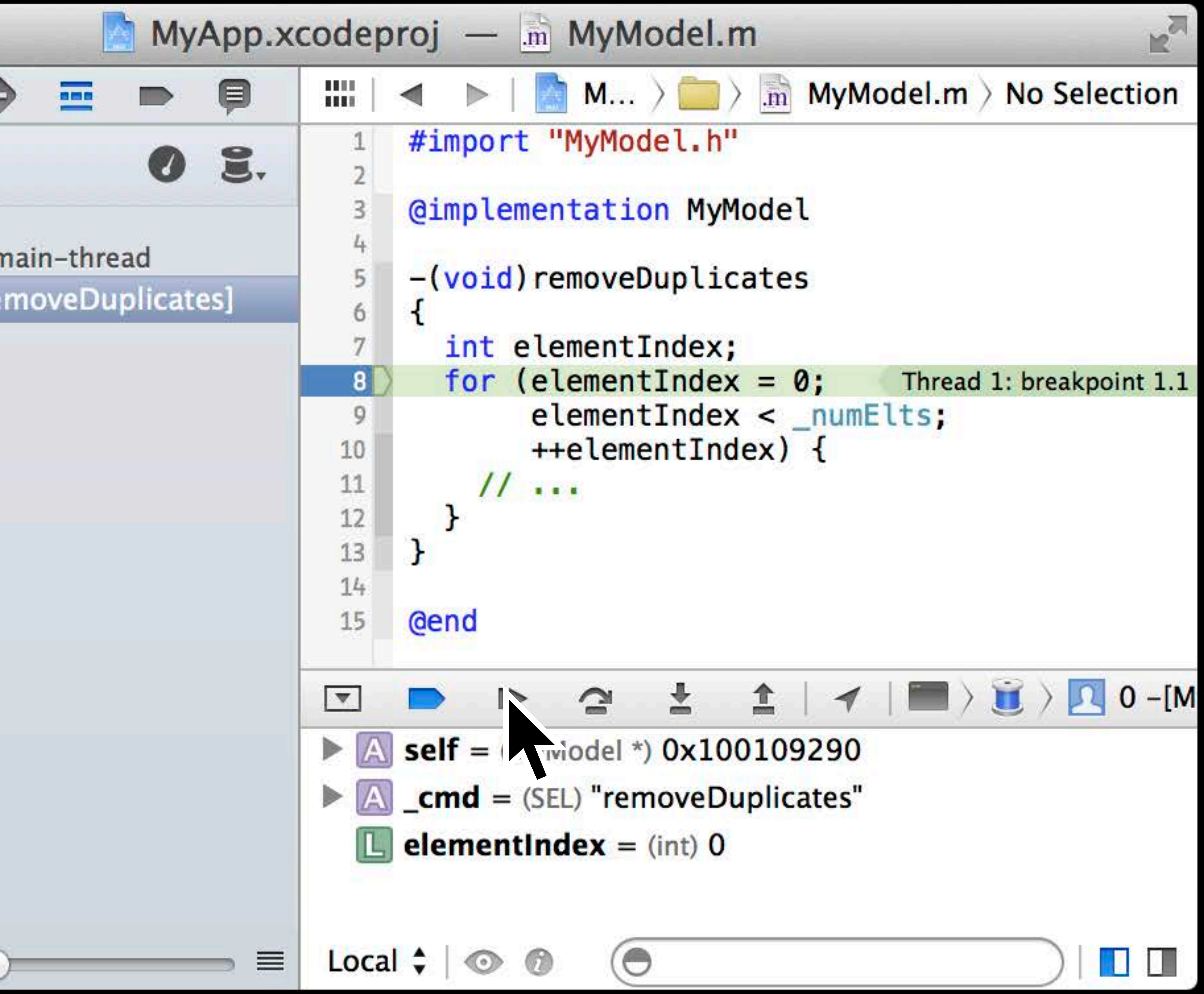


```
#import "MyModel.h"

@implementation MyModel

-(void)removeDuplicates
{
  int elementIndex;
  for (elementIndex = 0;
       elementIndex < _numElts;
       ++elementIndex) {
    // ...
  }
}

@end
```

```
▶ A self = (MyMo…) 0x100109290
▶ A _cmd = (SEL) "removeDuplicates"
  L 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

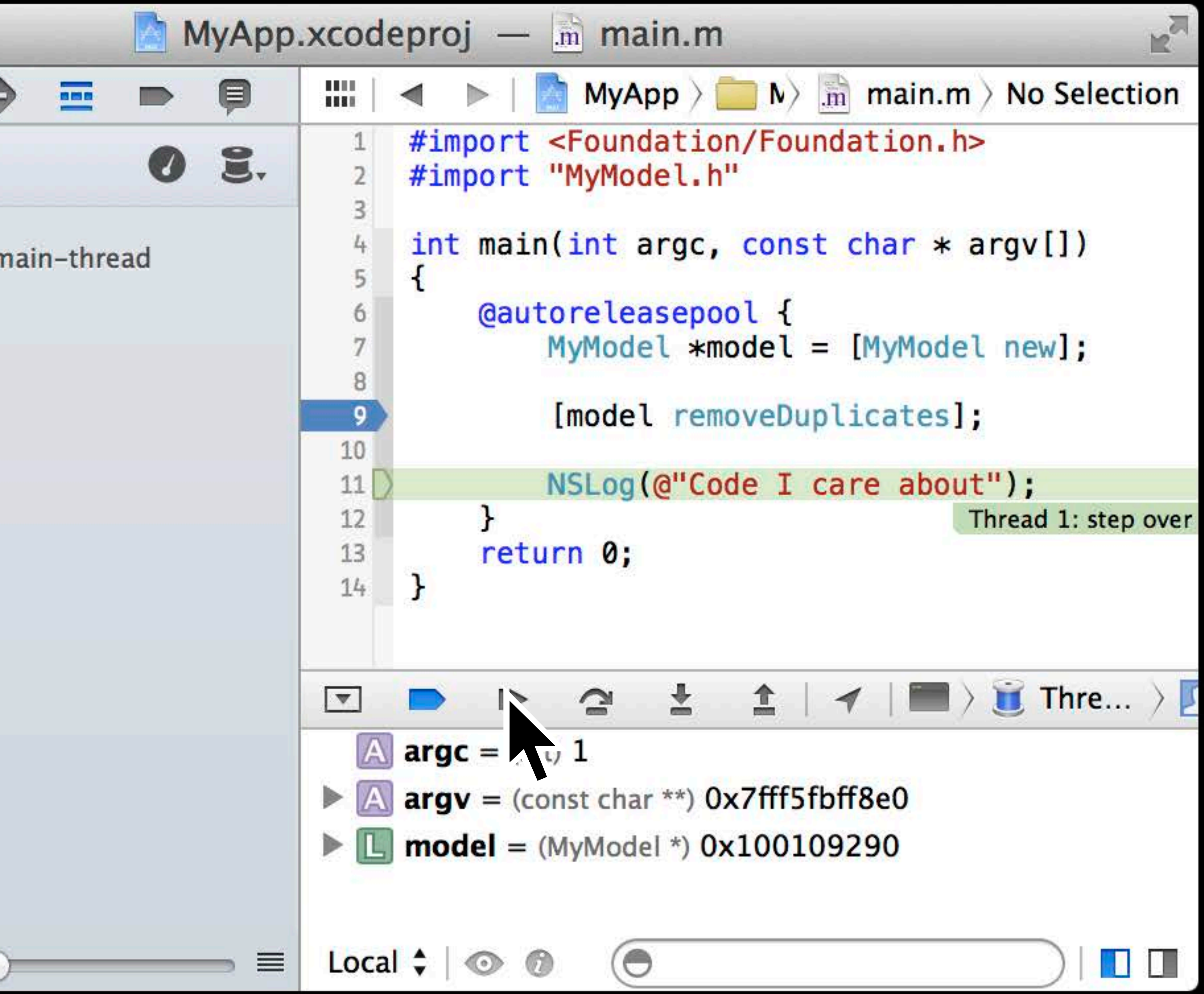


```
#import "MyModel.h"

@implementation MyModel

-(void)removeDuplicates
{
  int elementIndex;
  for (elementIndex = 0;
       elementIndex < _numElts;
       ++elementIndex) {
    // ...
  }
}

@end
```

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



```
#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;
}
```

```
argc = (int) 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]
Process 31109 stopped
* thread #1:
  -[ModelDerived removeDuplicates]
```

Don’t ignore breakpoints!
LLDB does by default

- 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”

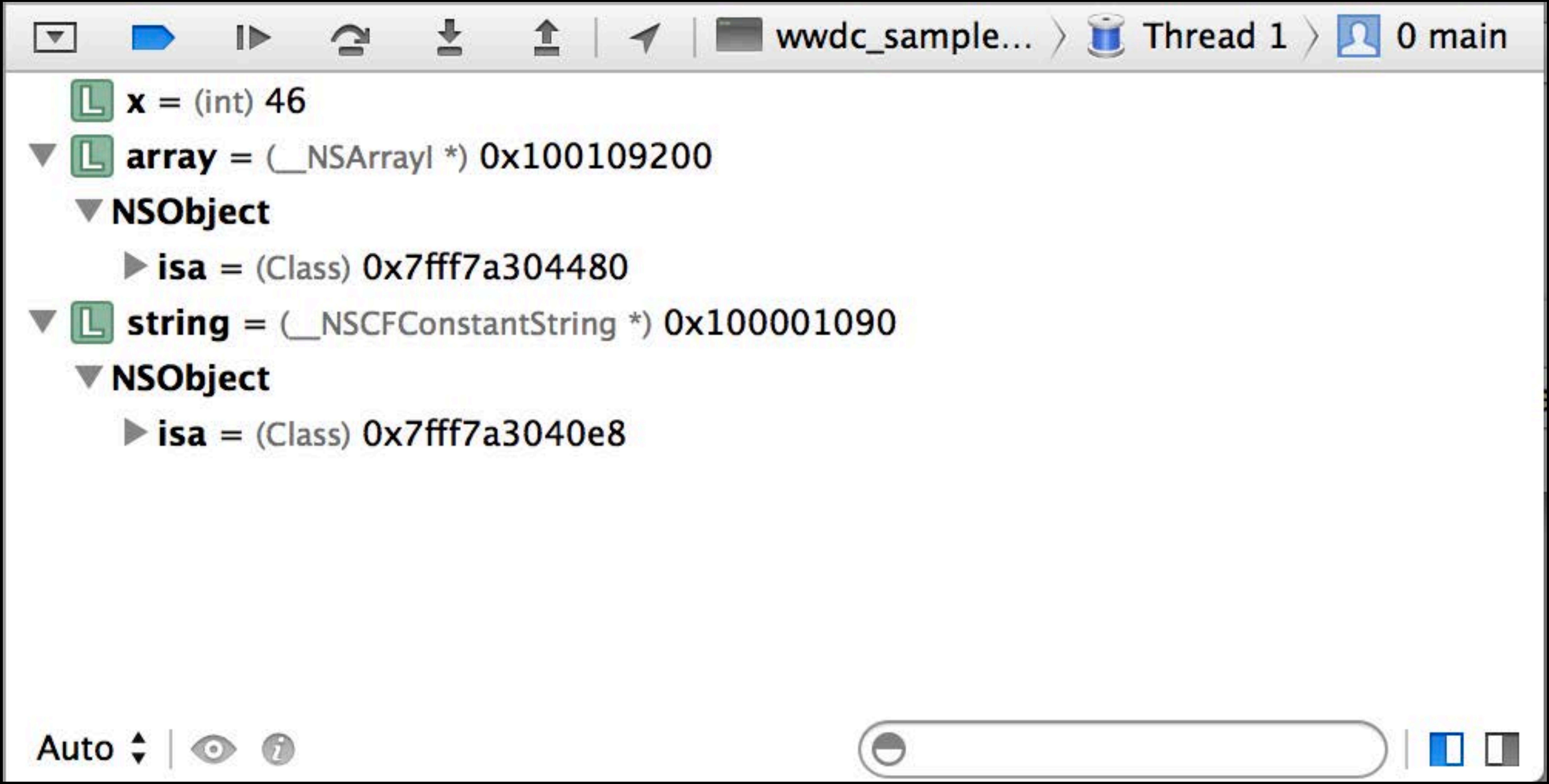
- Raw data is not always easy to decipher
  - Too complex
  - Not your types

“Raw Data” vs. “Data”

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

“Raw Data”

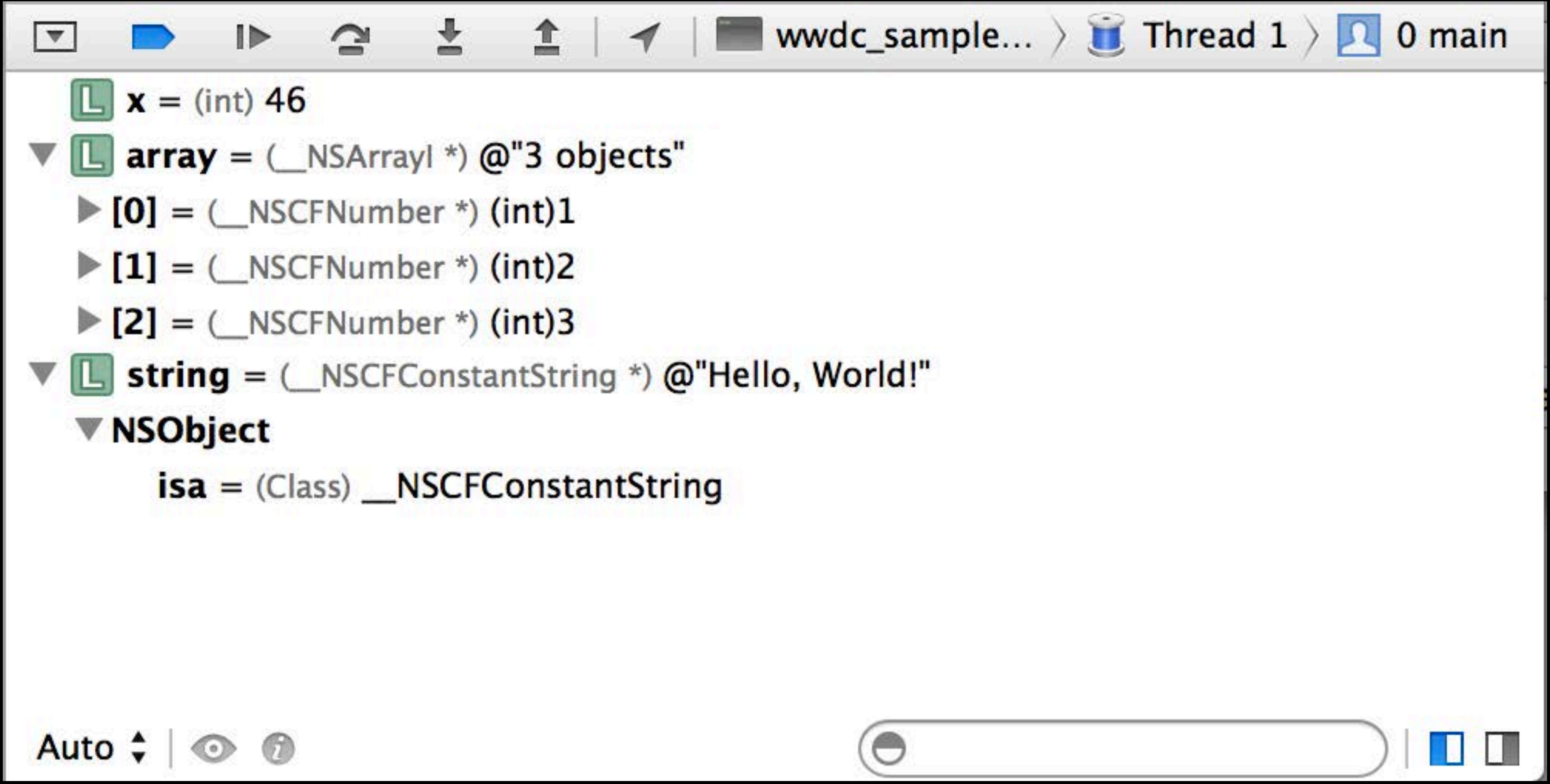
Life without formatters



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

“Data”

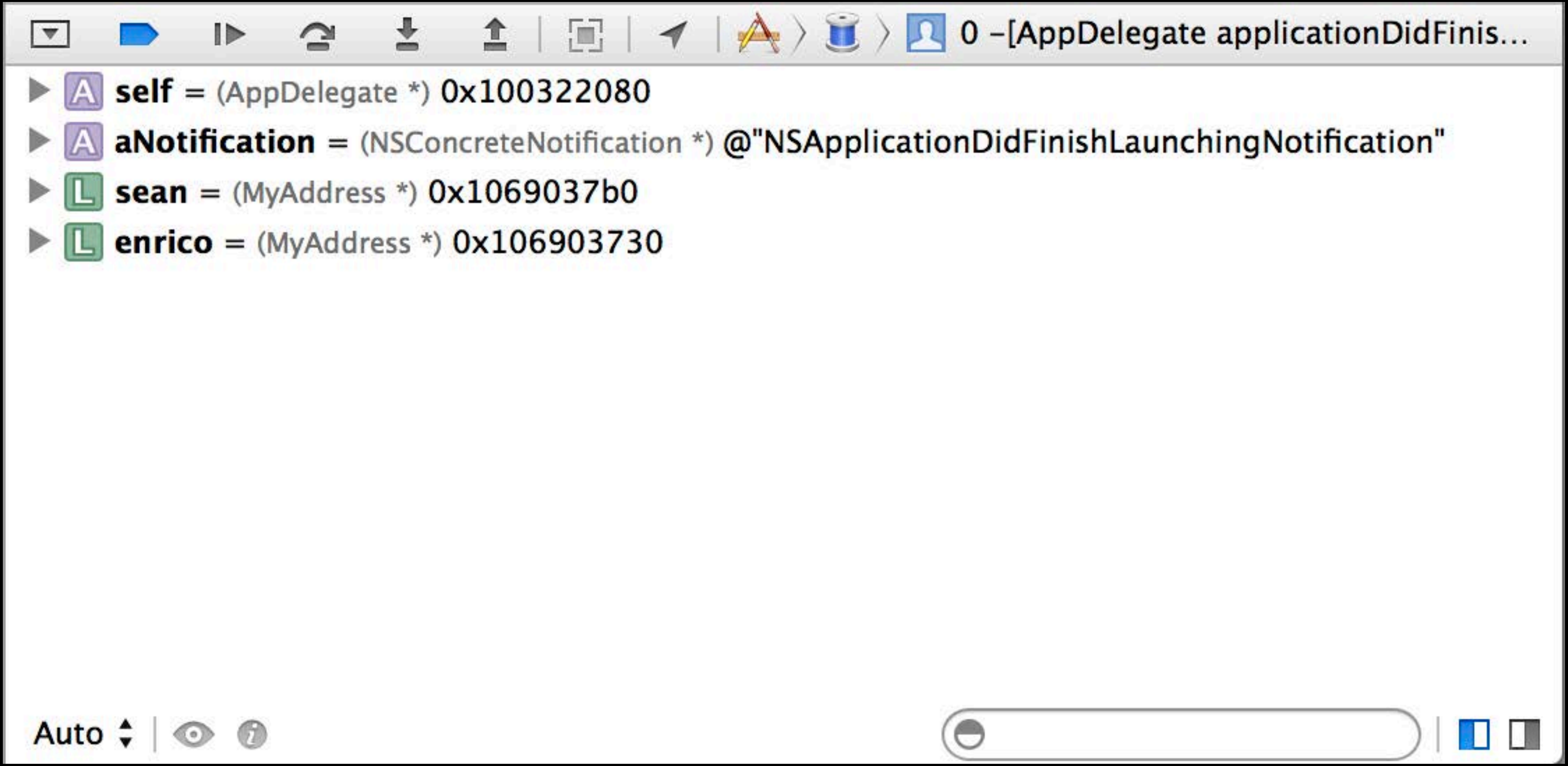
Life with formatters



```
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
```

“Raw Data”

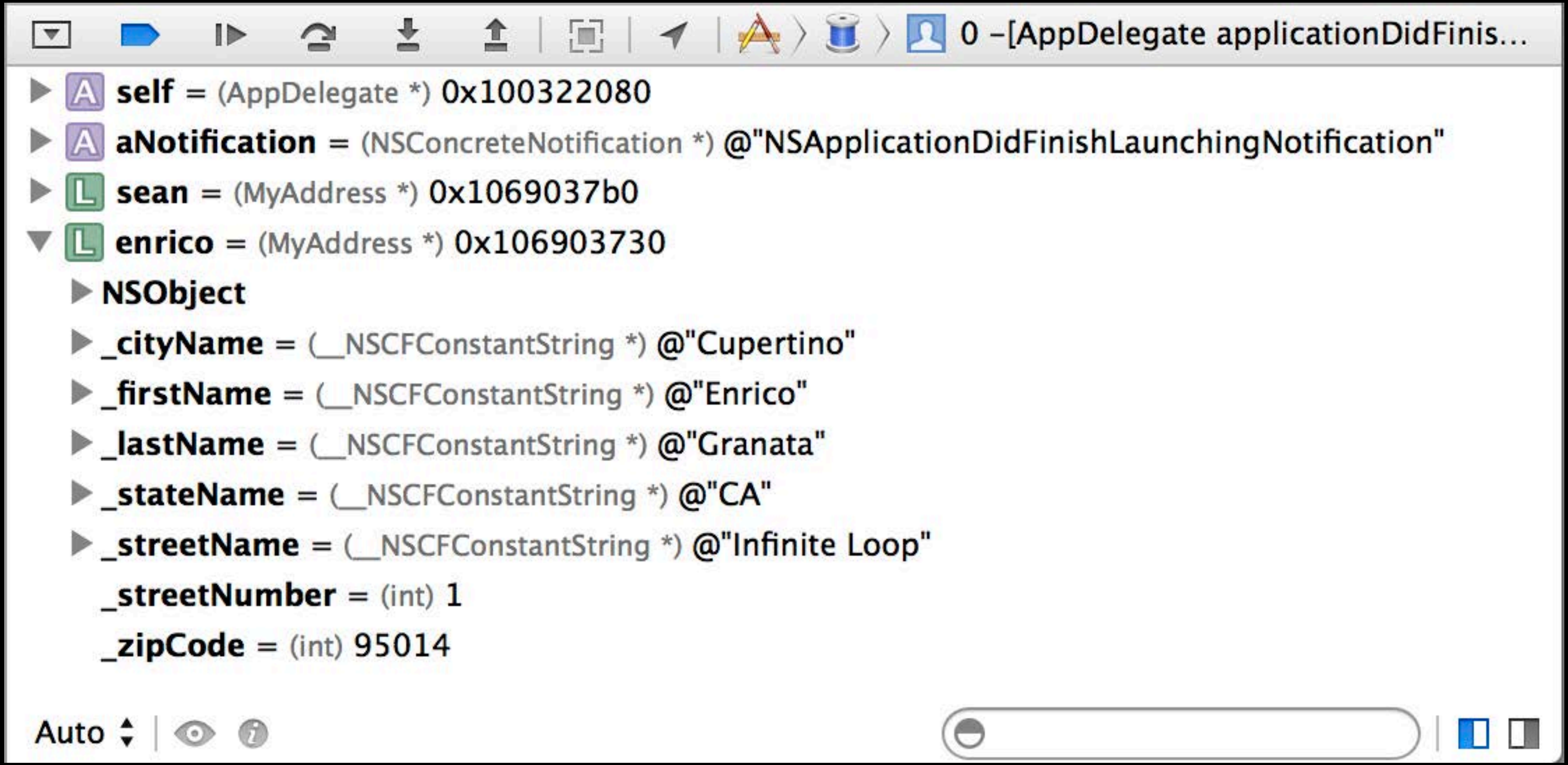
Life without formatters



```
self = (AppDelegate *) 0x100322080
aNotification = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
sean = (MyAddress *) 0x1069037b0
enrico = (MyAddress *) 0x106903730
```

“Raw Data”

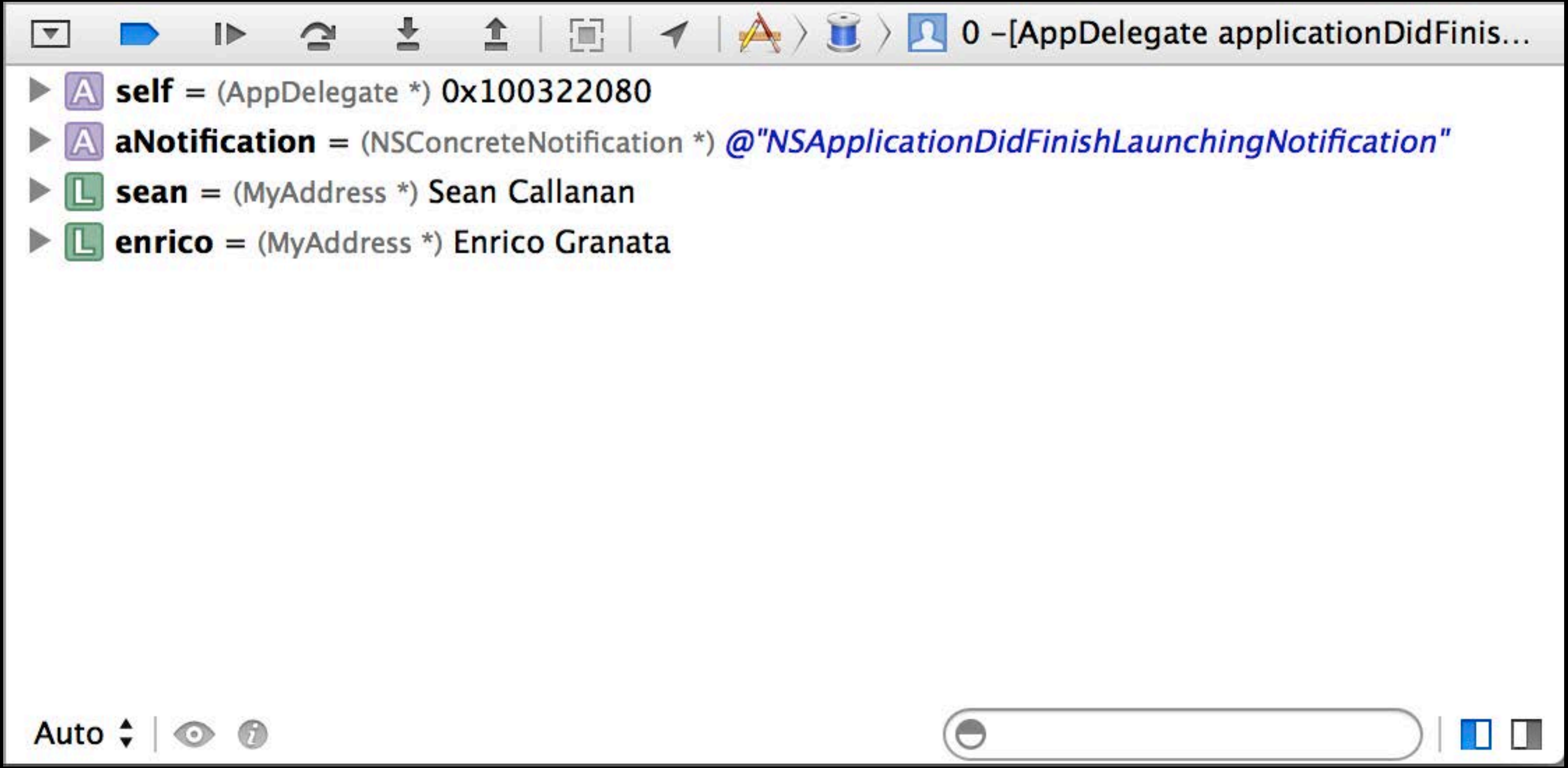
Life without formatters



```
▶ A self = (AppDelegate *) 0x100322080
▶ A aNotification = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
▶ L sean = (MyAddress *) 0x1069037b0
▼ L 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
```

“Data”

Life with formatters



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

Data Formatters

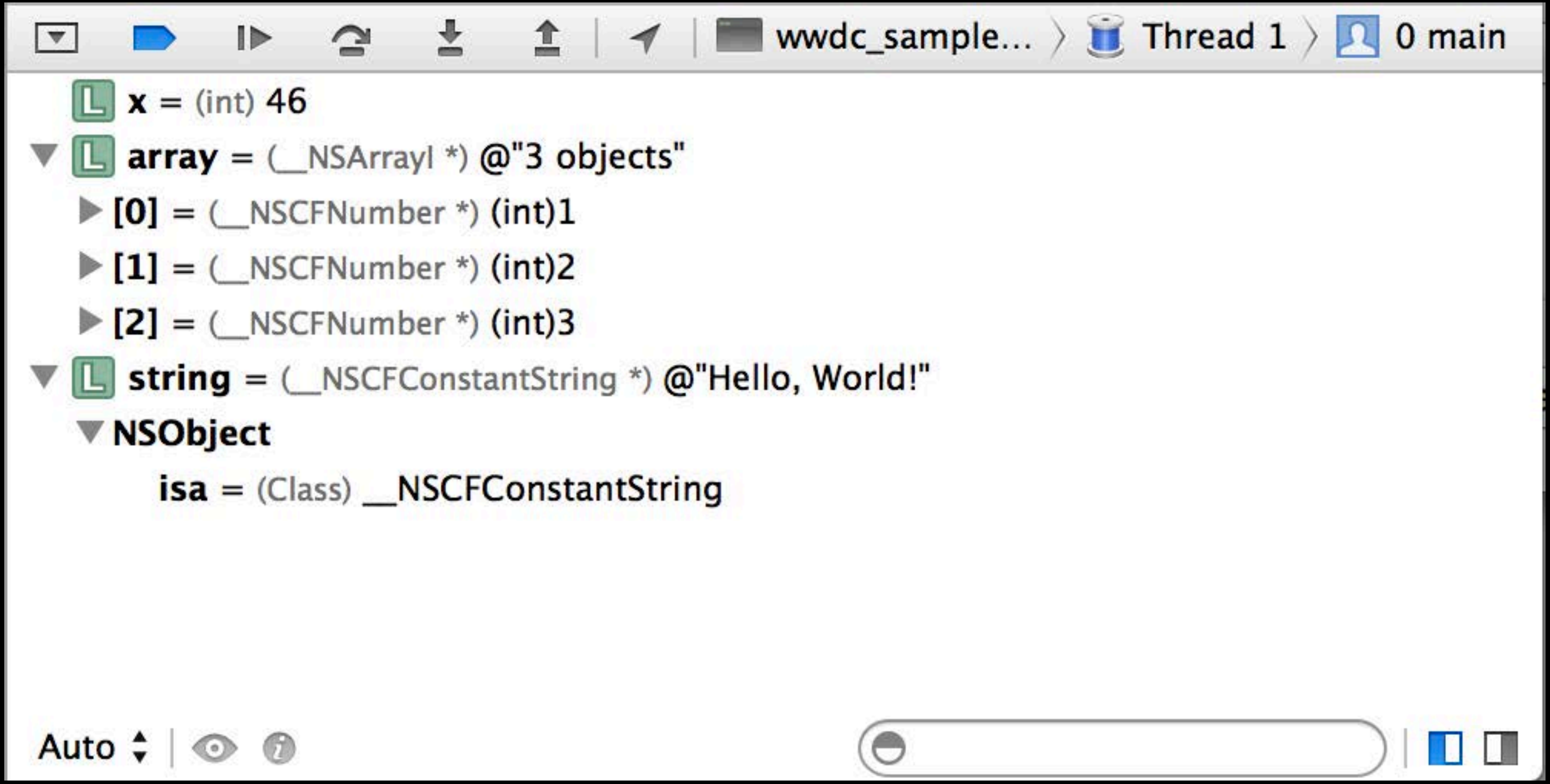
Data Formatters

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

Data Formatters

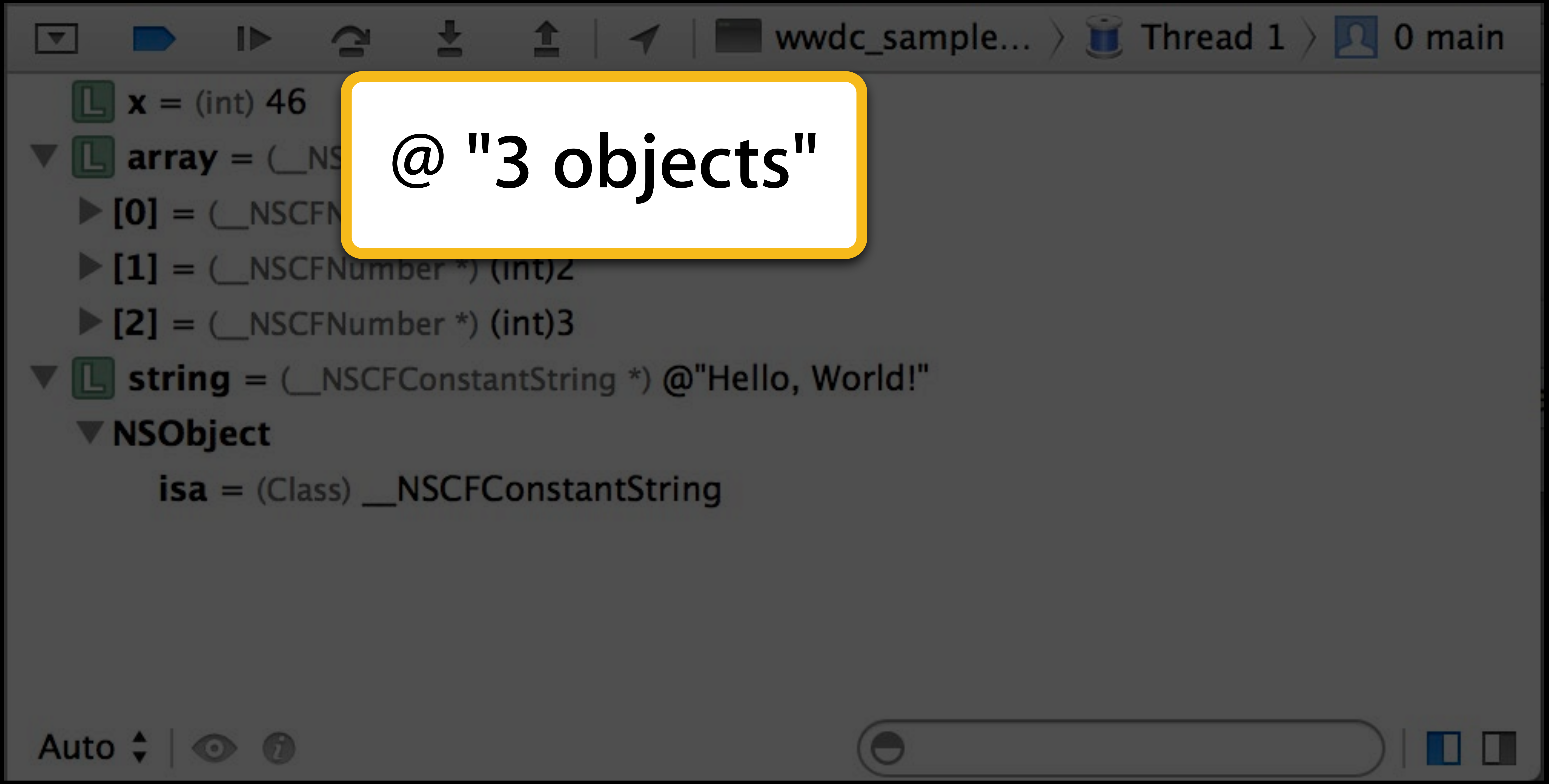
- Built-in formatters for system libraries
    - STL
    - CoreFoundation
    - Foundation
- What we do… you can do too

Summaries



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

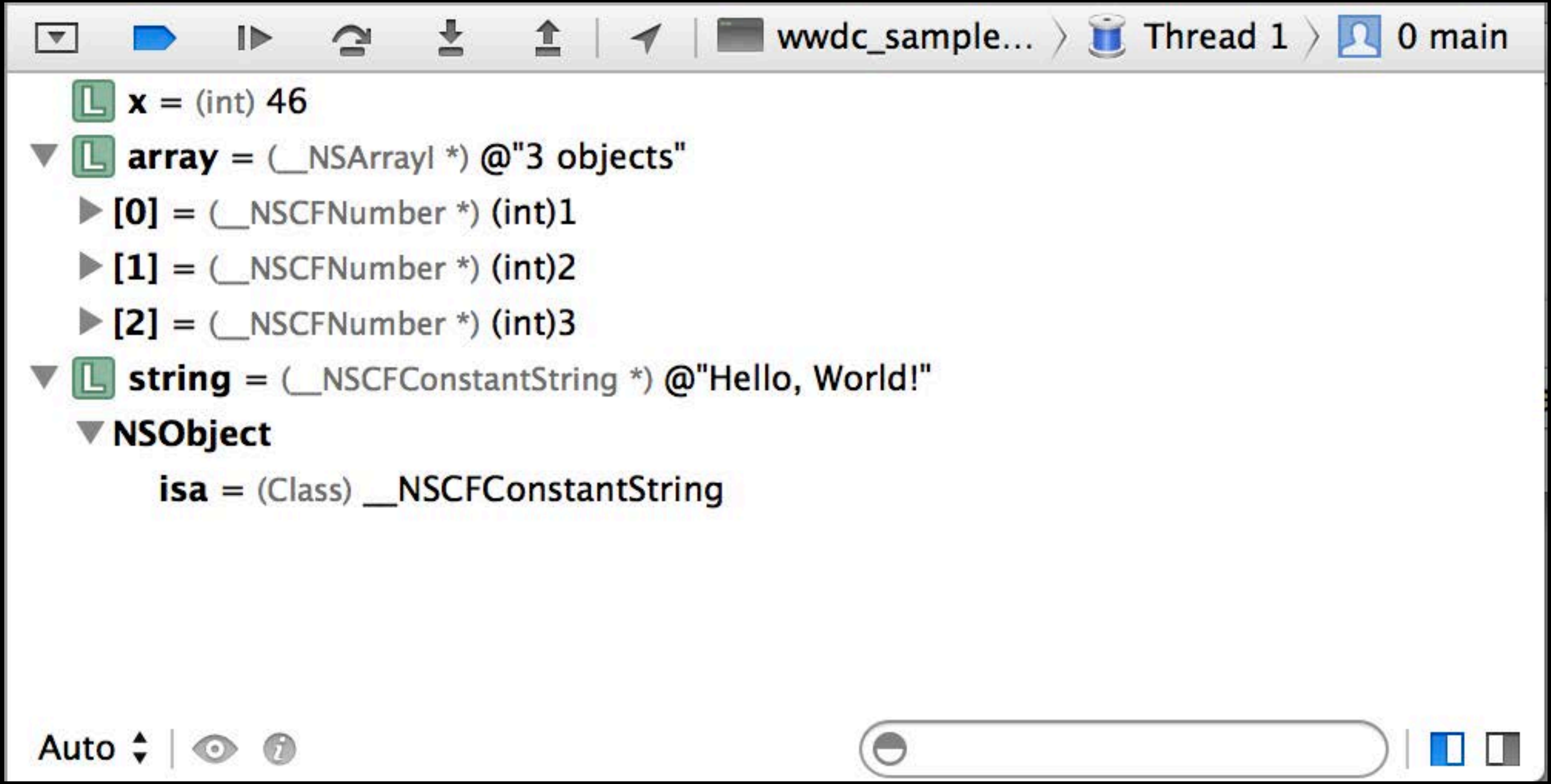
Summaries



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

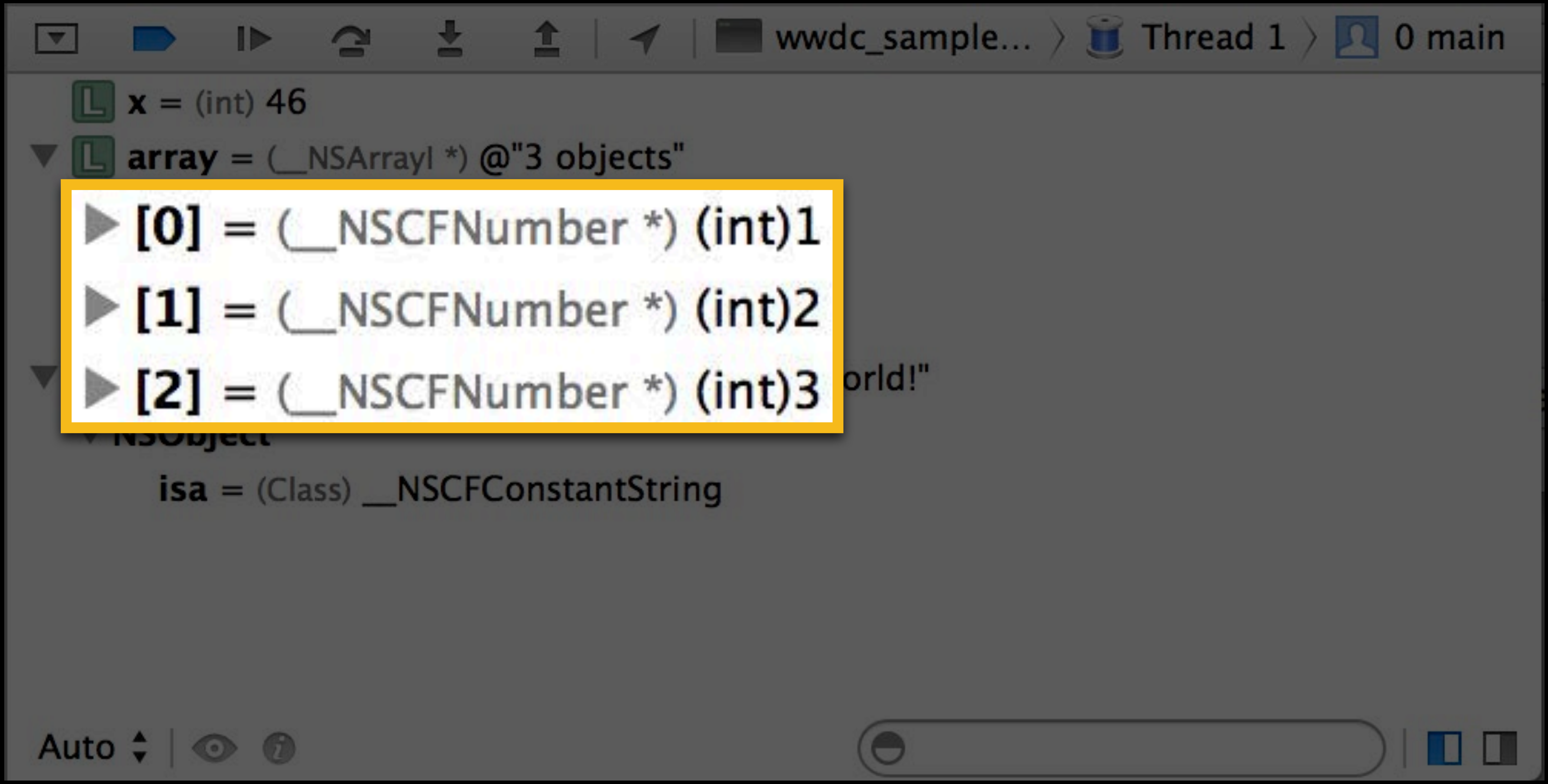
@ "3 objects"

Synthetic Children



```
  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
```

Synthetic Children



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

How Python Summaries Work

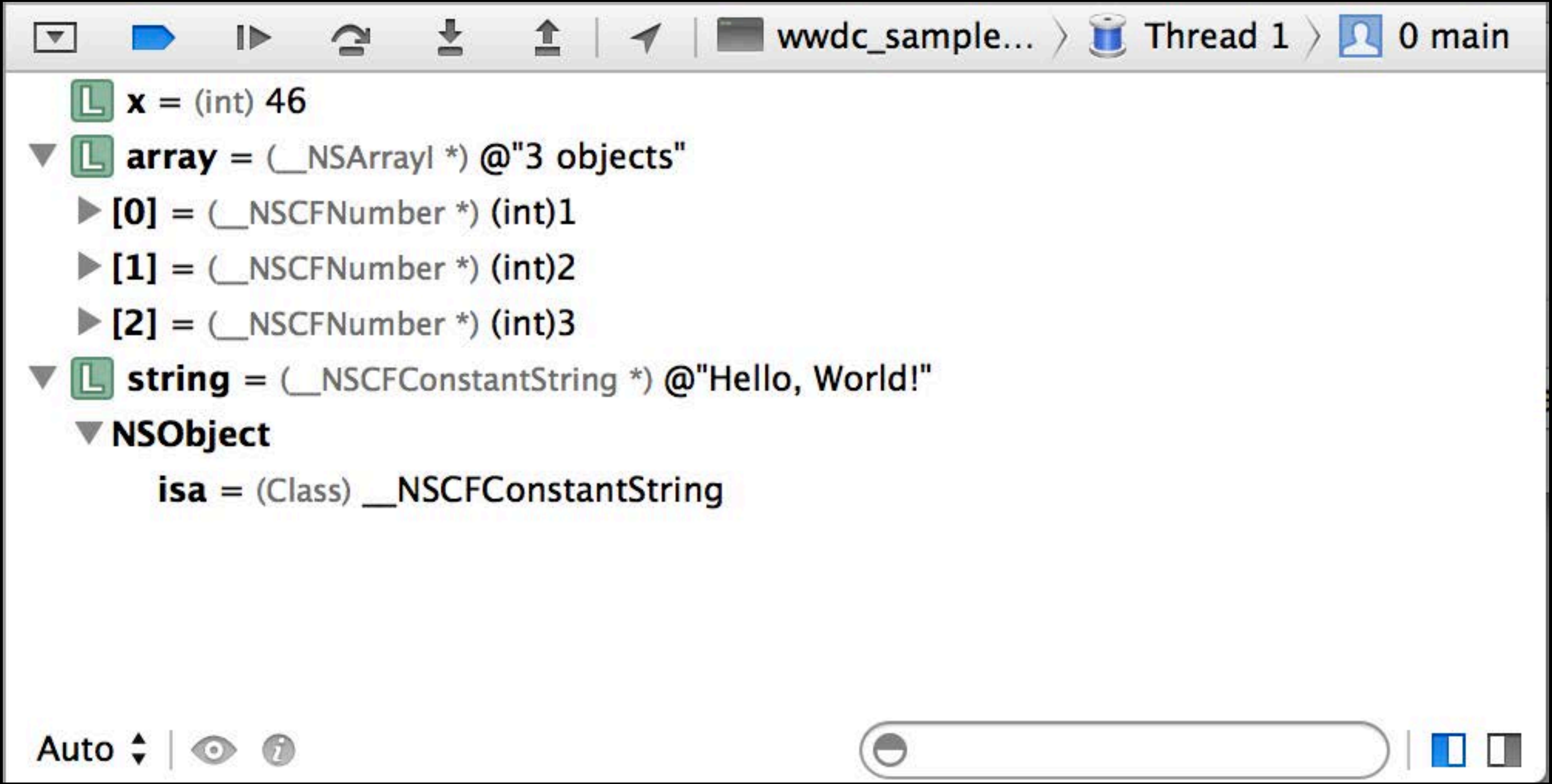
How Python Summaries Work

- 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

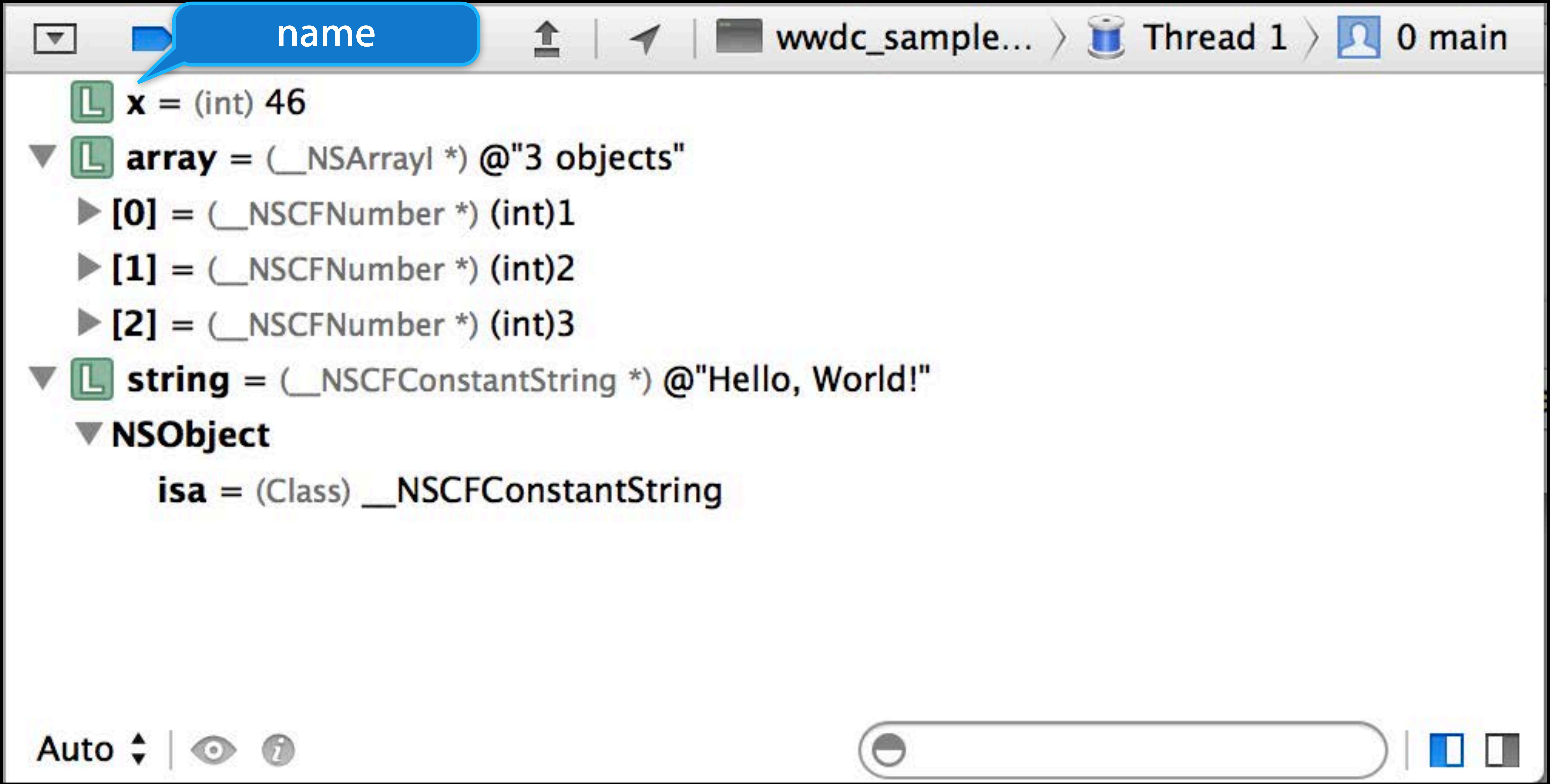
- Summaries match a type to a Python function
  - Base matching is by type name
  - Refer to LLDB web site for other rules
    - http://lldb.llvm.org/varformats.html
- The function is called whenever a value is displayed
  - LLDB passes an SBValue to it
    - Part of the LLDB Object Model
  - The function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

SBValue



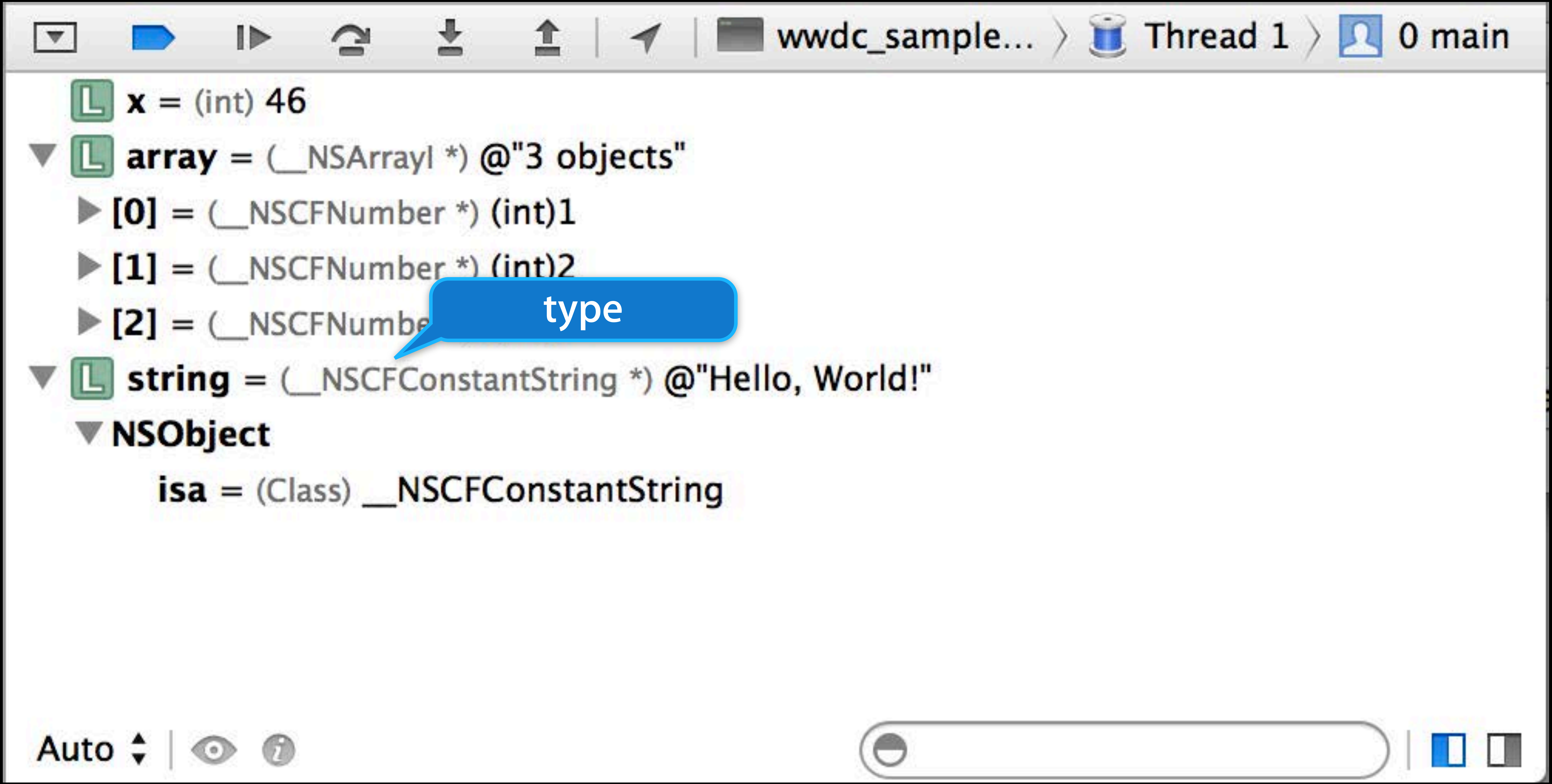
```
    x = (int) 46
▼ array = (__NSArrayI *) @"3 objects"
  ▶ [0] = (__NSCFNumber *) (int)1
  ▶ [1] = (__NSCFNumber *) (int)2
  ▶ [2] = (__NSCFNumber *) (int)3
▼ string = (__NSCFConstantString *) @"Hello, World!"
  ▼ NSObject
      isa = (Class) __NSCFConstantString
```

SBValue



```
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
```

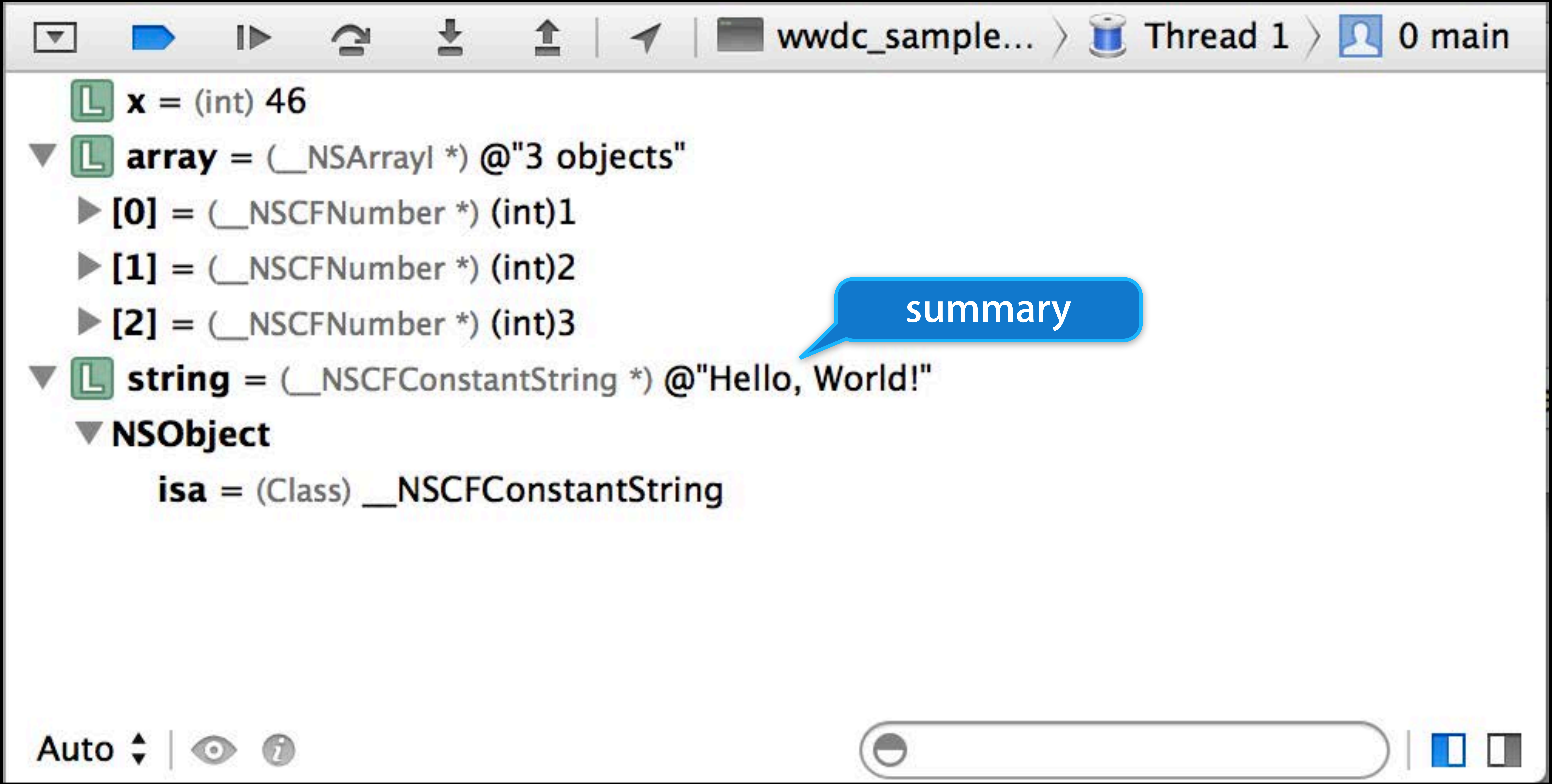
SBValue



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

type

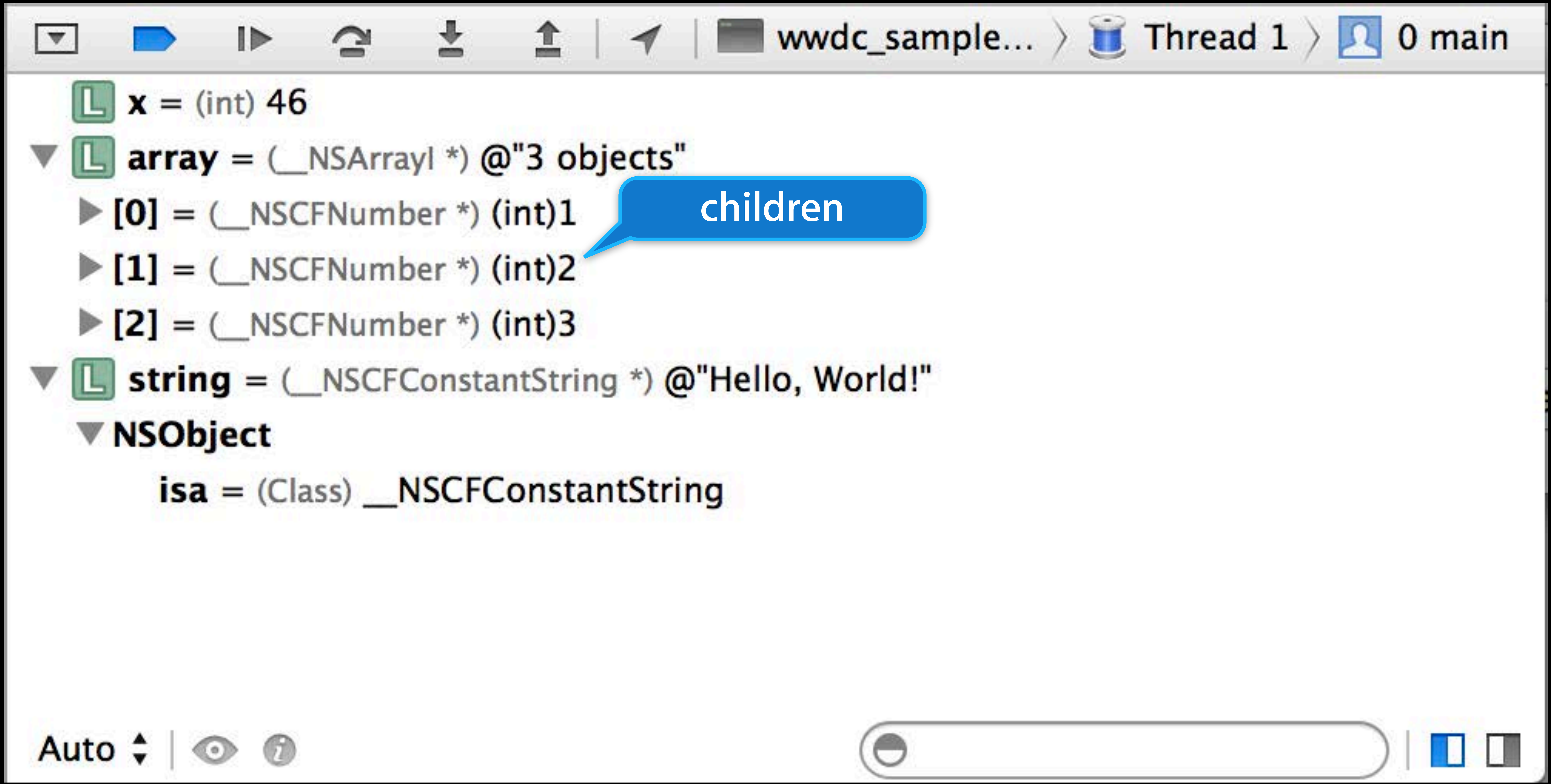
SBValue



```
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
```

summary

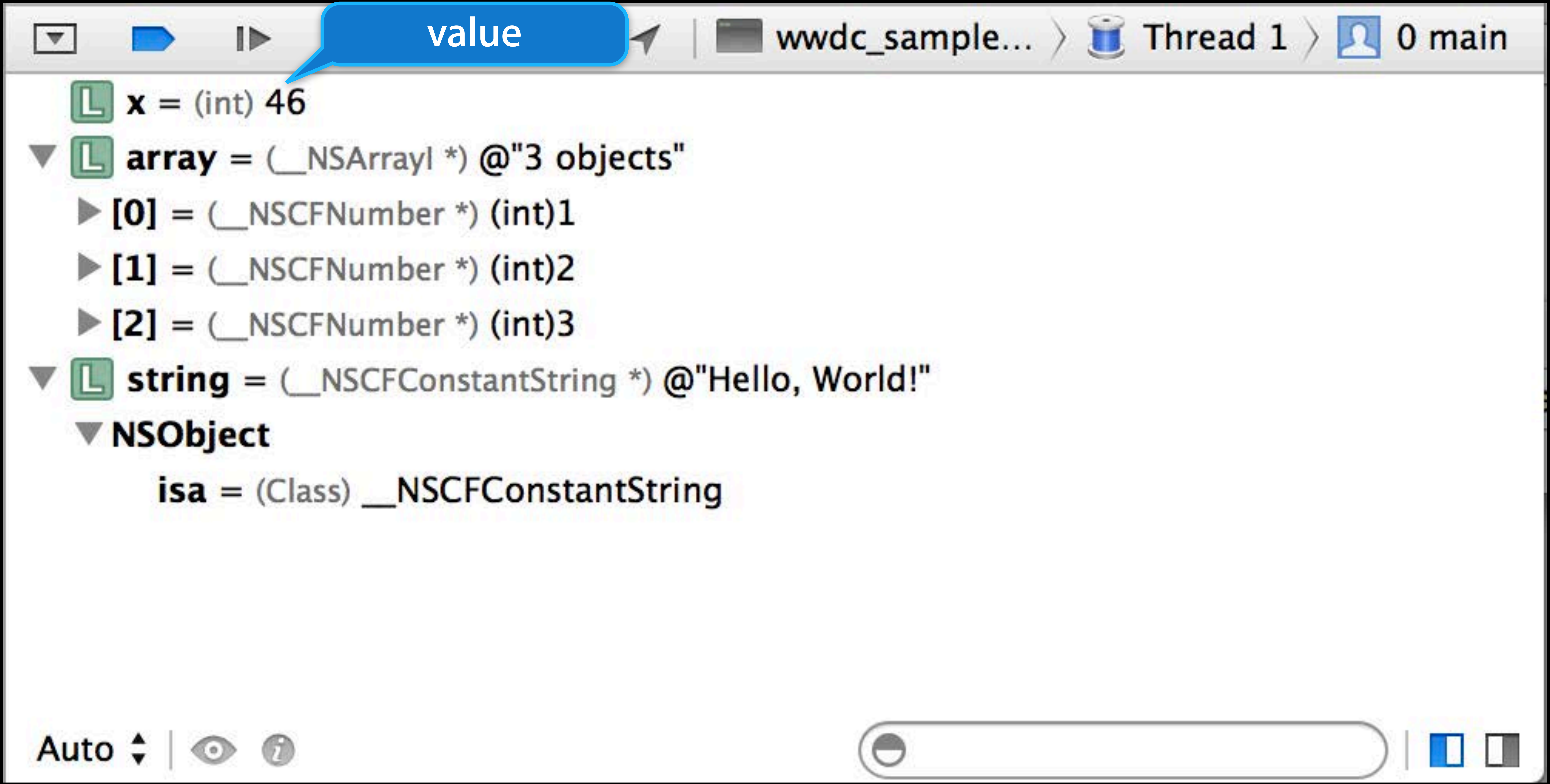
SBValue



```
  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
```

children

SBValue



```
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
```

Example

Summarizing an Address

Example

Summarizing an Address

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

Example

Summarizing an Address

SBValue

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

Example

Summarizing an Address

SBValue

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

Example

Summarizing an Address

SBValue

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

Example

Summarizing an Address

SBValue

```
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
```

Example

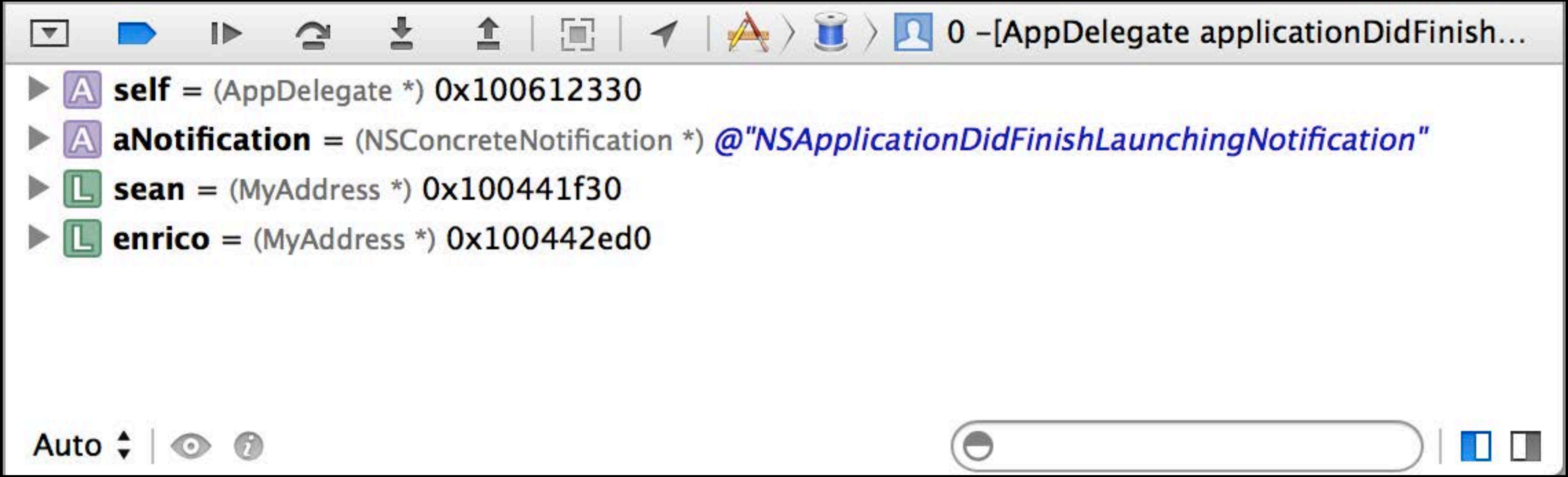
Summarizing an Address

SBValue

```
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

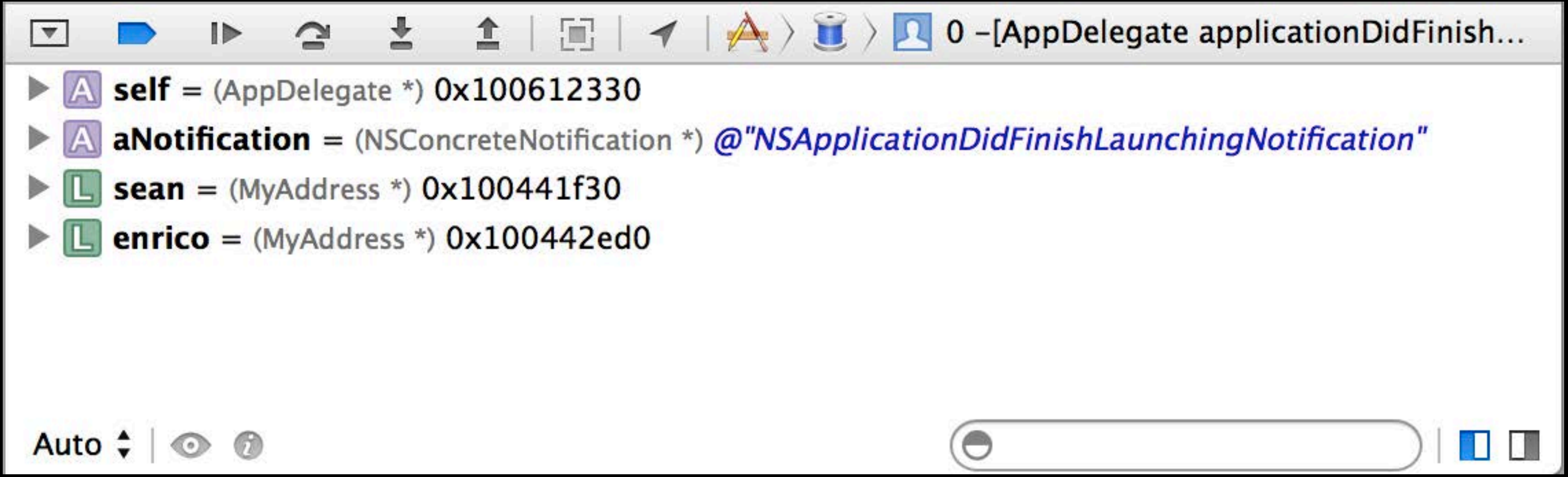


0 –[AppDelegate applicationDidFinish…

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

Example

Summarizing an Address

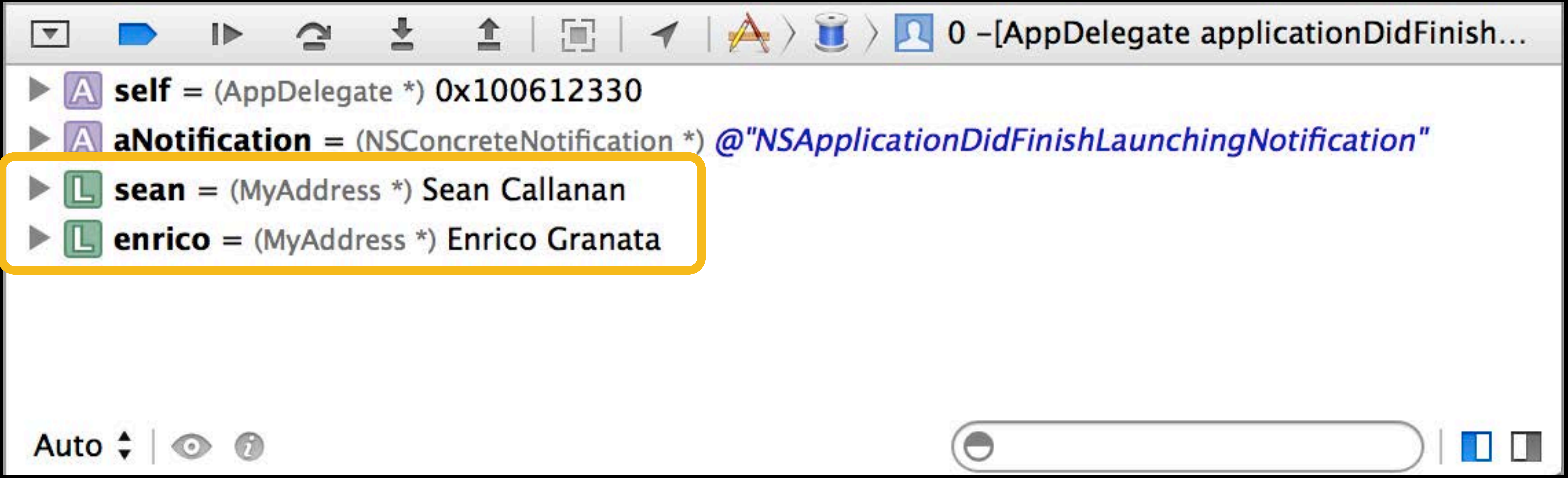


```
self = (AppDelegate *) 0x100612330
aNotification = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
sean = (MyAddress *) 0x100441f30
enrico = (MyAddress *) 0x100442ed0
```

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

Example

Summarizing an Address



```
self = (AppDelegate *) 0x100612330
aNotification = (NSConcreteNotification *) @"NSApplicationDidFinishLaunchingNotification"
sean = (MyAddress *) Sean Callanan
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

```c
typedef void* Opaque;

Opaque makeOpaque();
int useOpaque(Opaque);
void freeOpaque(Opaque);
```

Opaque.h

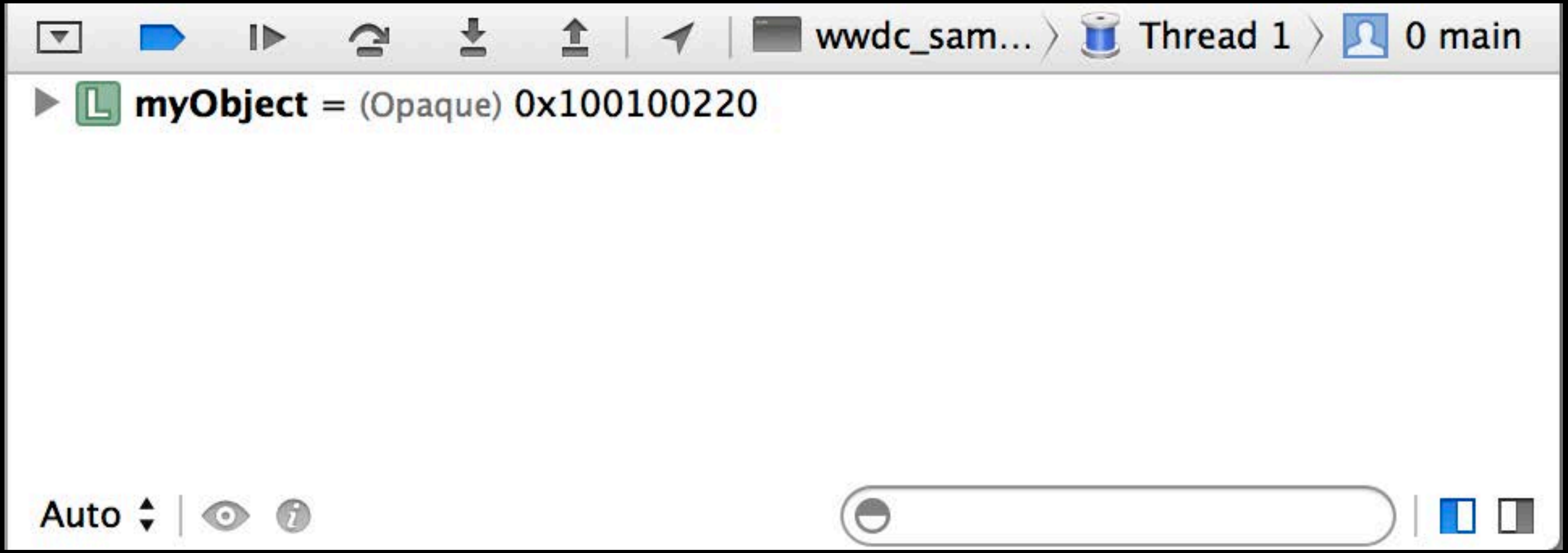
expression for Data Analysis

```
typedef void* Opaque;

Opaque makeOpaque();
int useOpa
void freeOp
```

```
struct ImplOpaque {
  int aThing;
  float anotherThing;
  char* oneMoreThing;
};
```

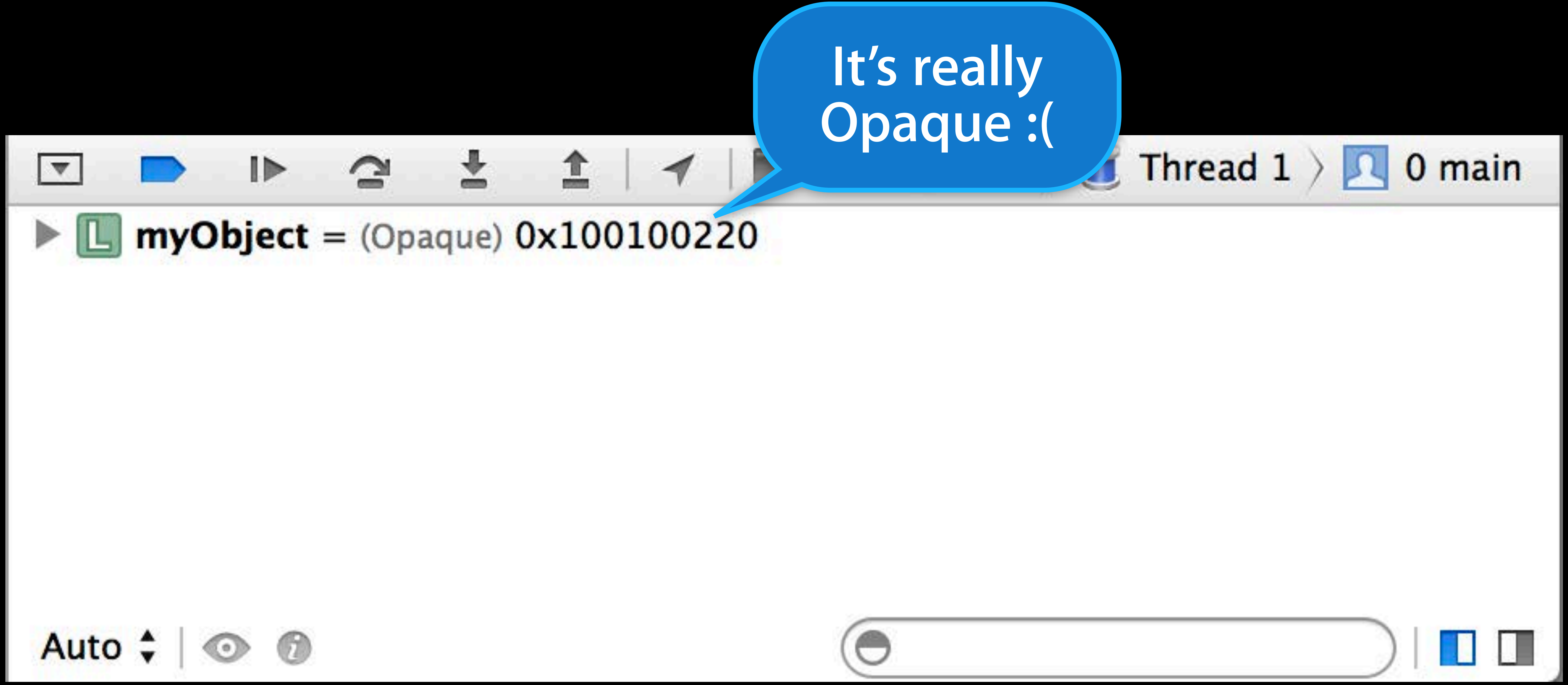
Opaque.cpp

expression for Data Analysis



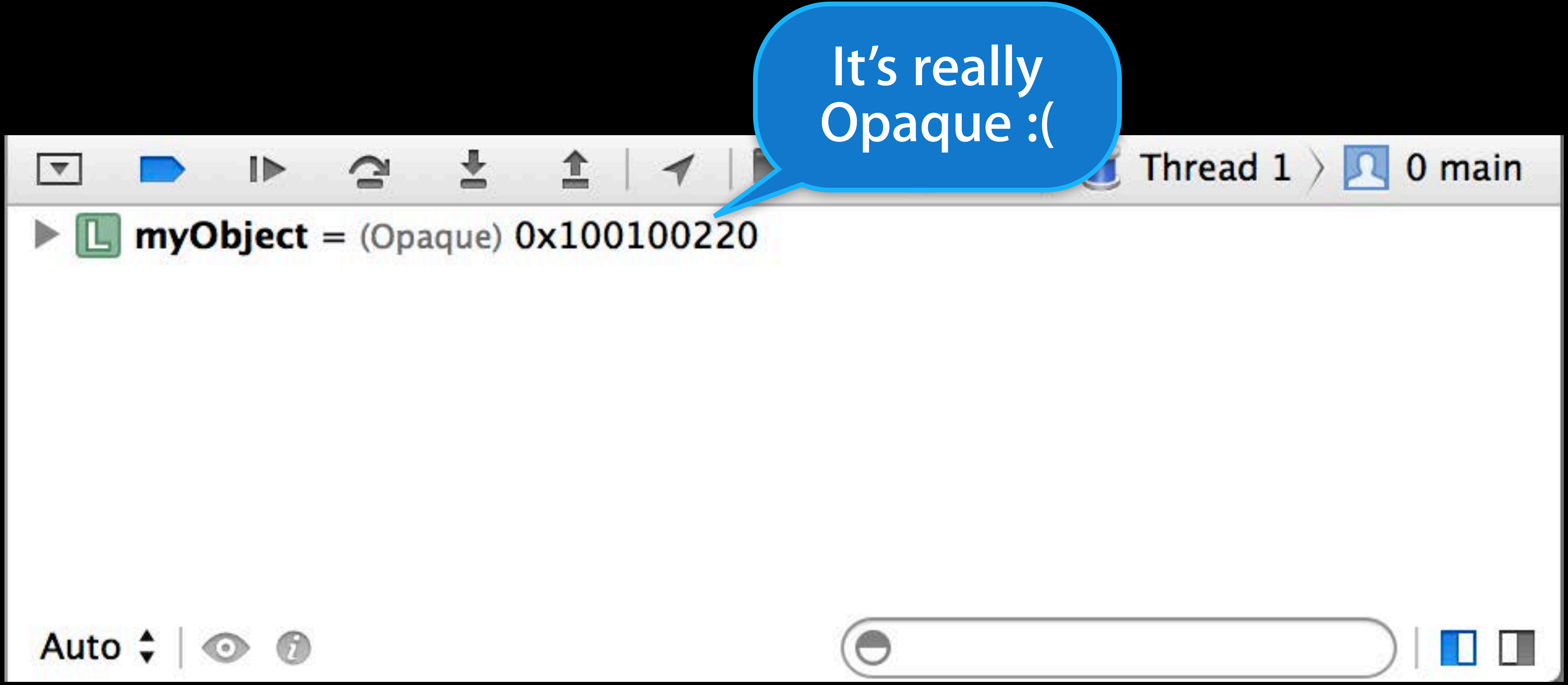
```
▶ L myObject = (Opaque) 0x100100220
```

expression for Data Analysis



myObject = (Opaque) 0x100100220

expression for Data Analysis

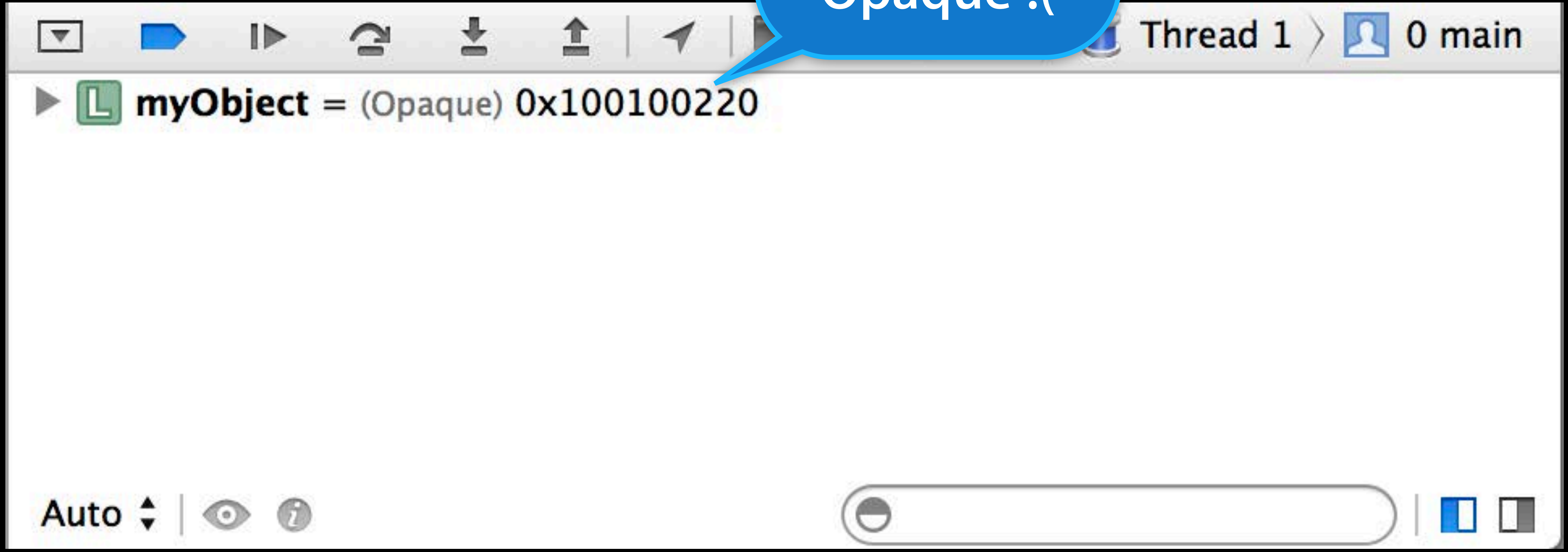


myObject = (Opaque) 0x100100220

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

expression for Data Analysis

It’s really Opaque :(

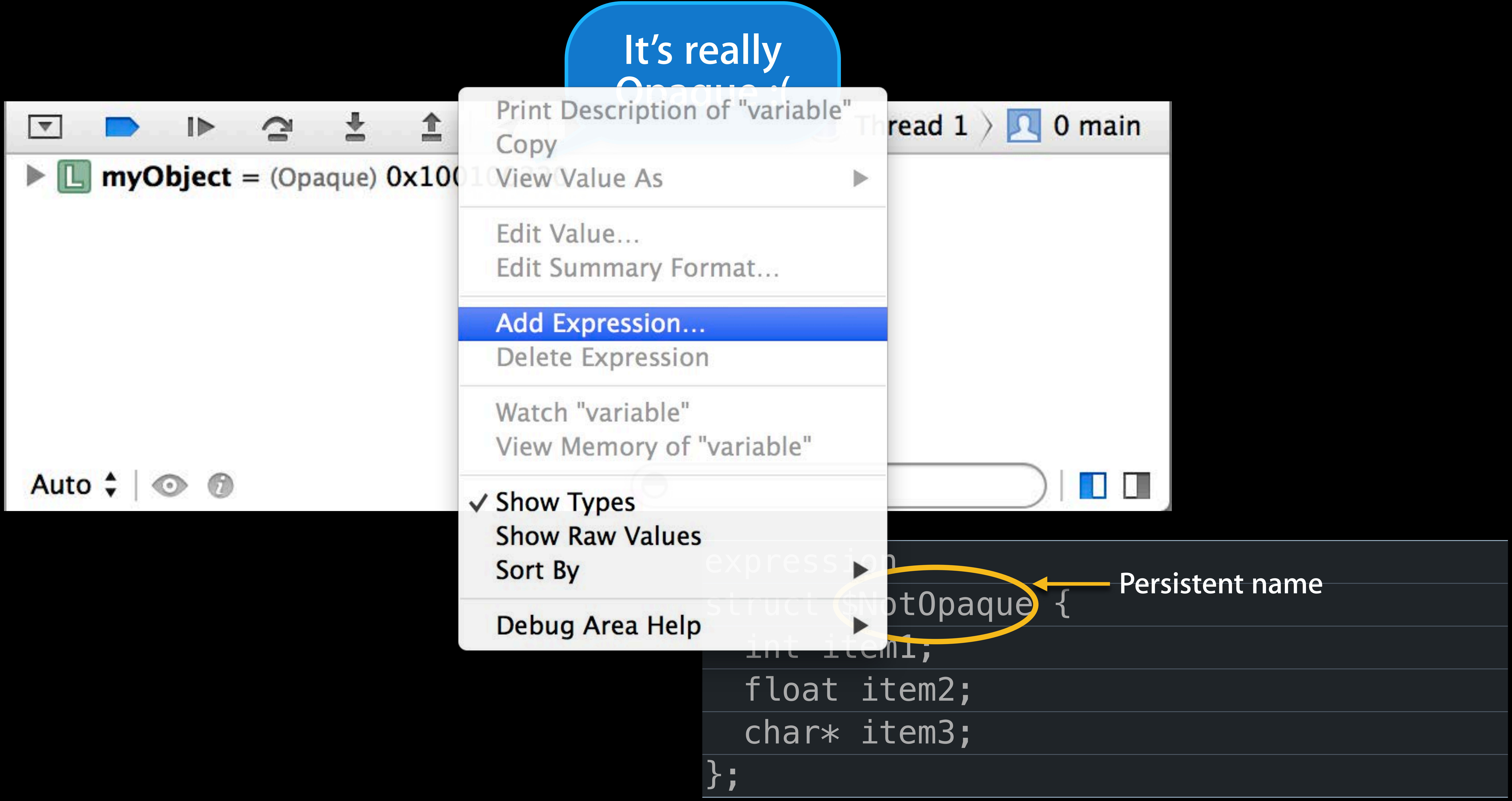


myObject = (Opaque) 0x100100220

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

Persistent name

expression for Data Analysis



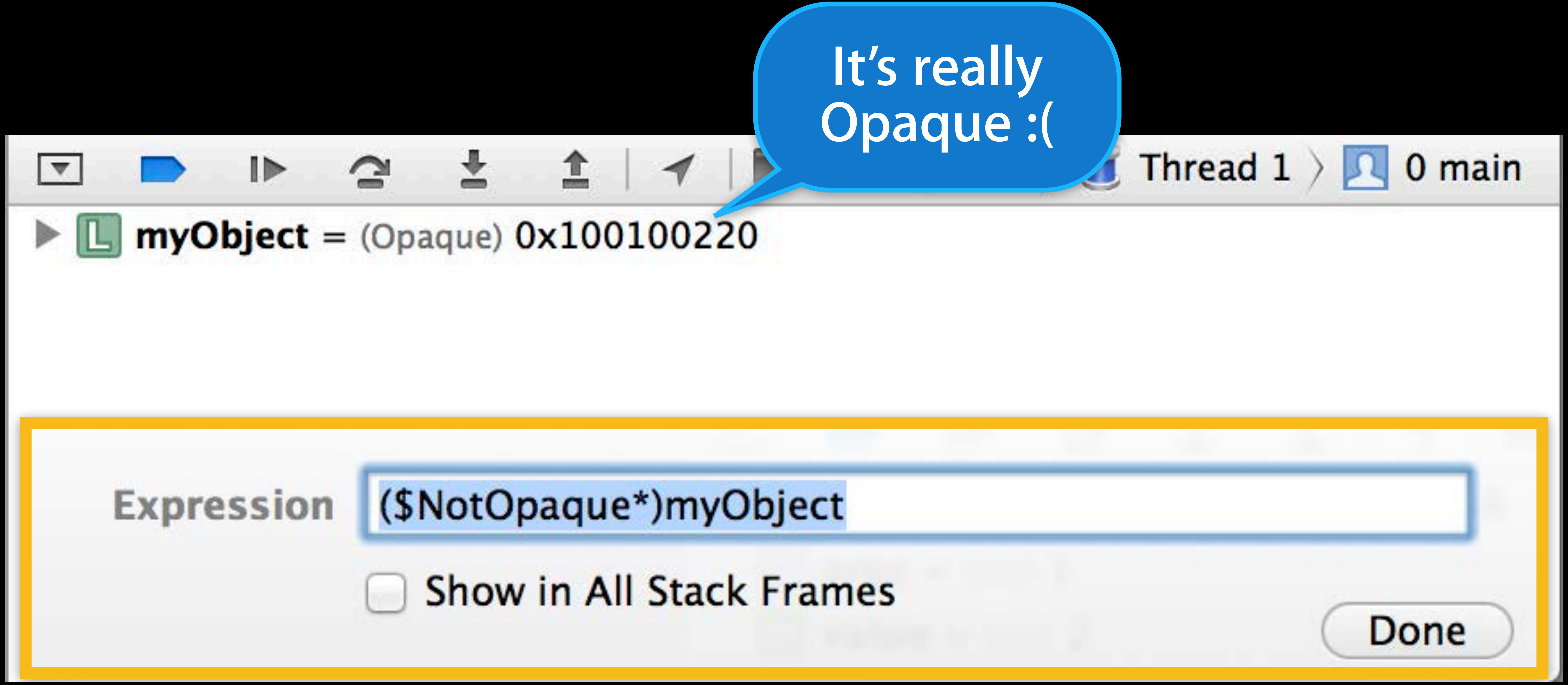
It’s really Opaque :(

myObject = (Opaque) 0x10010

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

Persistent name

expression for Data Analysis



It’s really Opaque :(

▶ myObject = (Opaque) 0x100100220

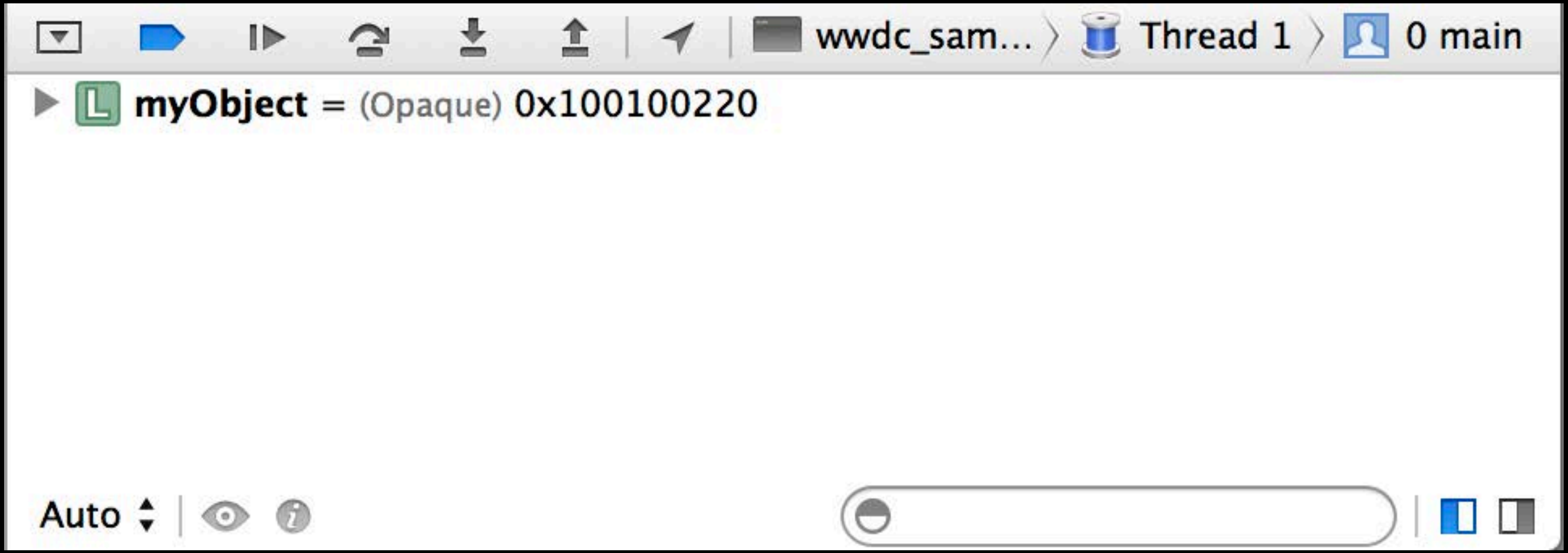
Expression ($NotOpaque*)myObject

☐ Show in All Stack Frames

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

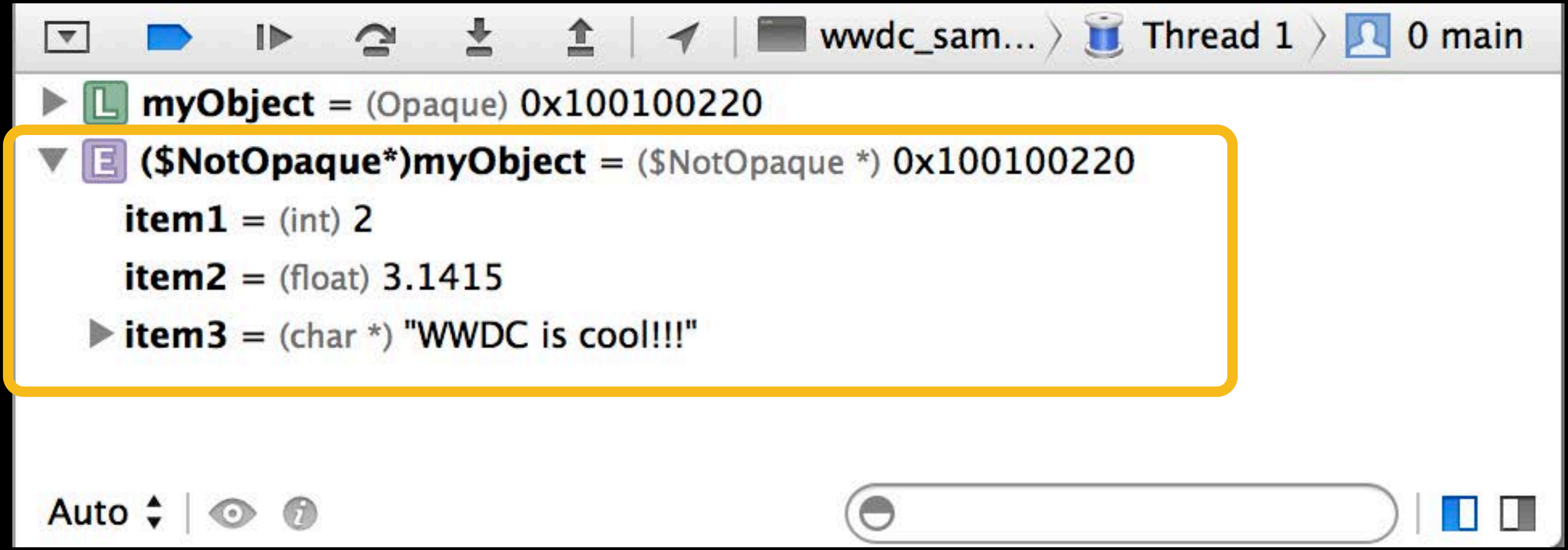
Persistent name

expression for Data Analysis



myObject = (Opaque) 0x100100220

expression for Data Analysis



```
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

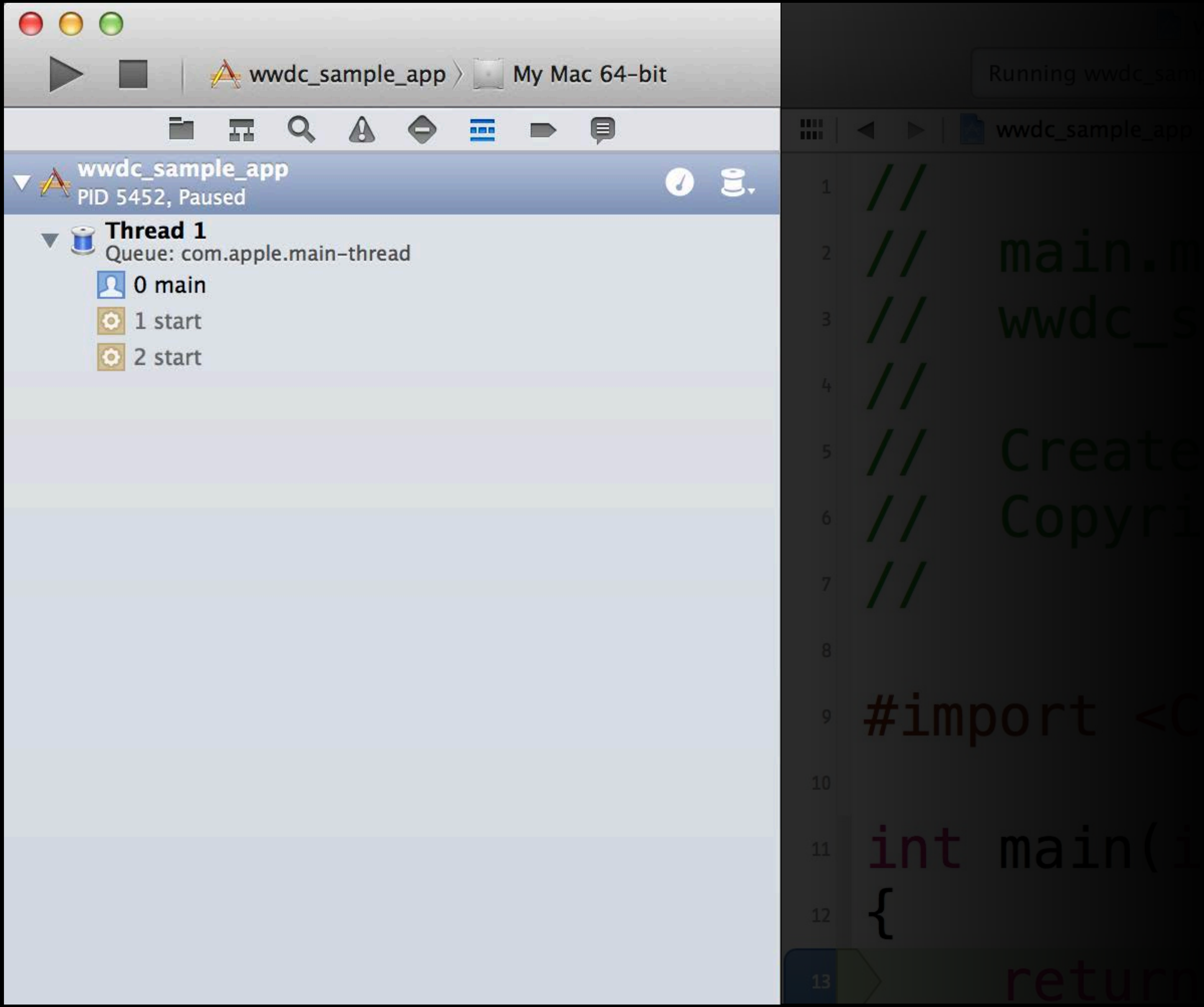
The LLDB Object Model

- 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

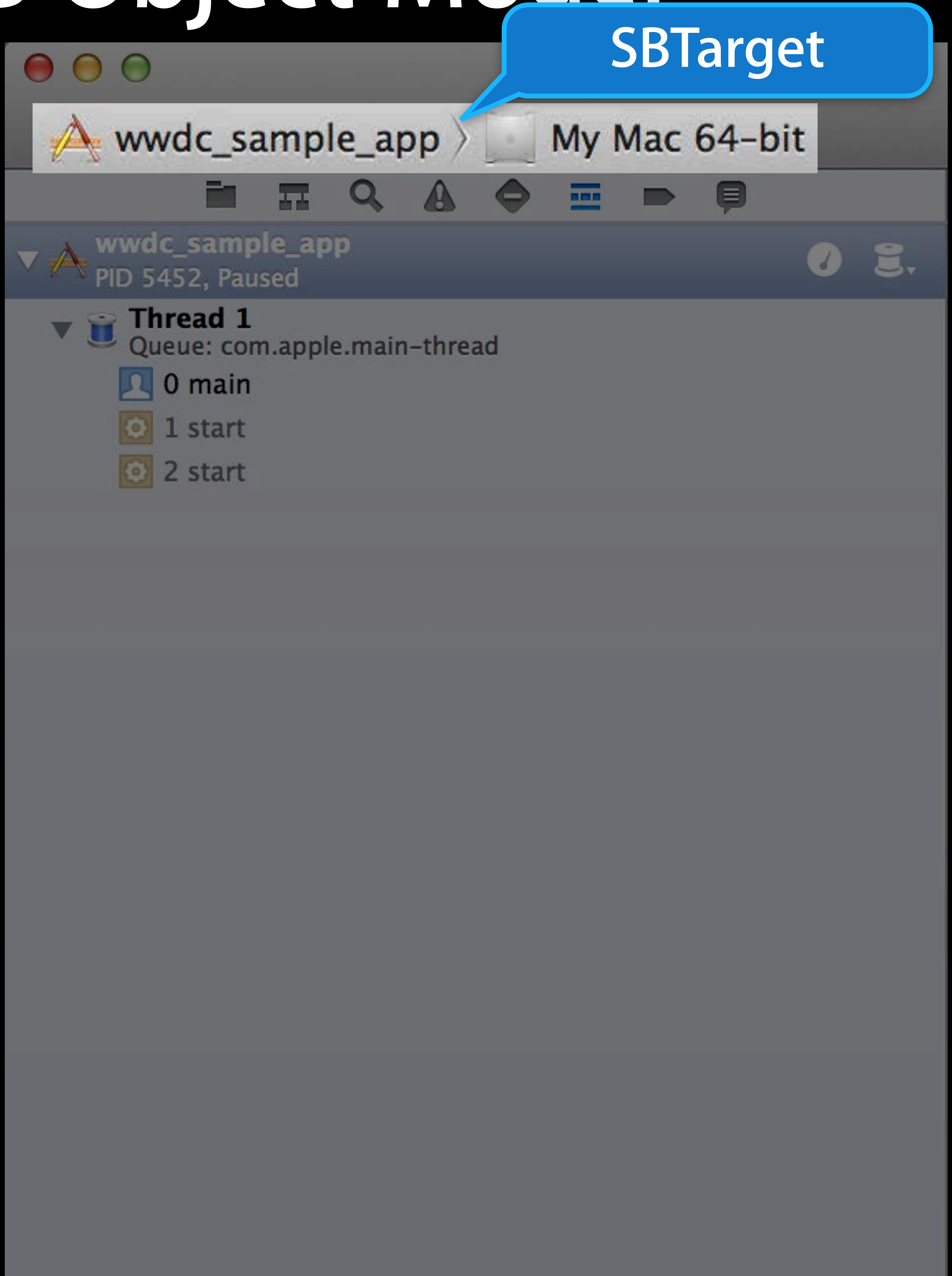
The LLDB Object Model



```
wwdc_sample_app
PID 5452, Paused
  Thread 1
  Queue: com.apple.main-thread
    0 main
    1 start
    2 start
```

```
//
//  main.m
//  wwdc_s
//
//  Create
//  Copyri
//

#import <C

int main(i
{
    return
```

The LLDB Object Model



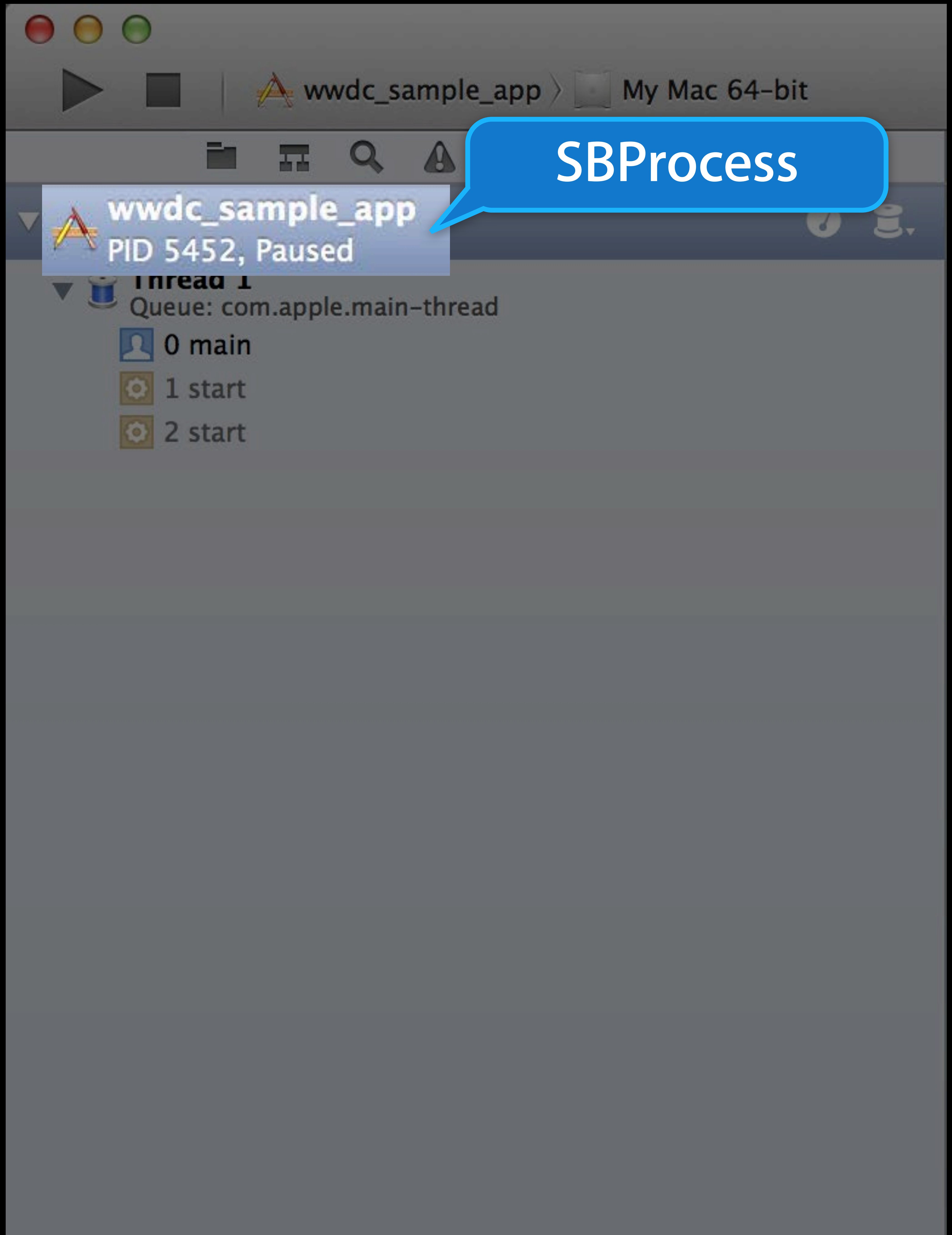
SBTarget

wwdc_sample_app
PID 5452, Paused

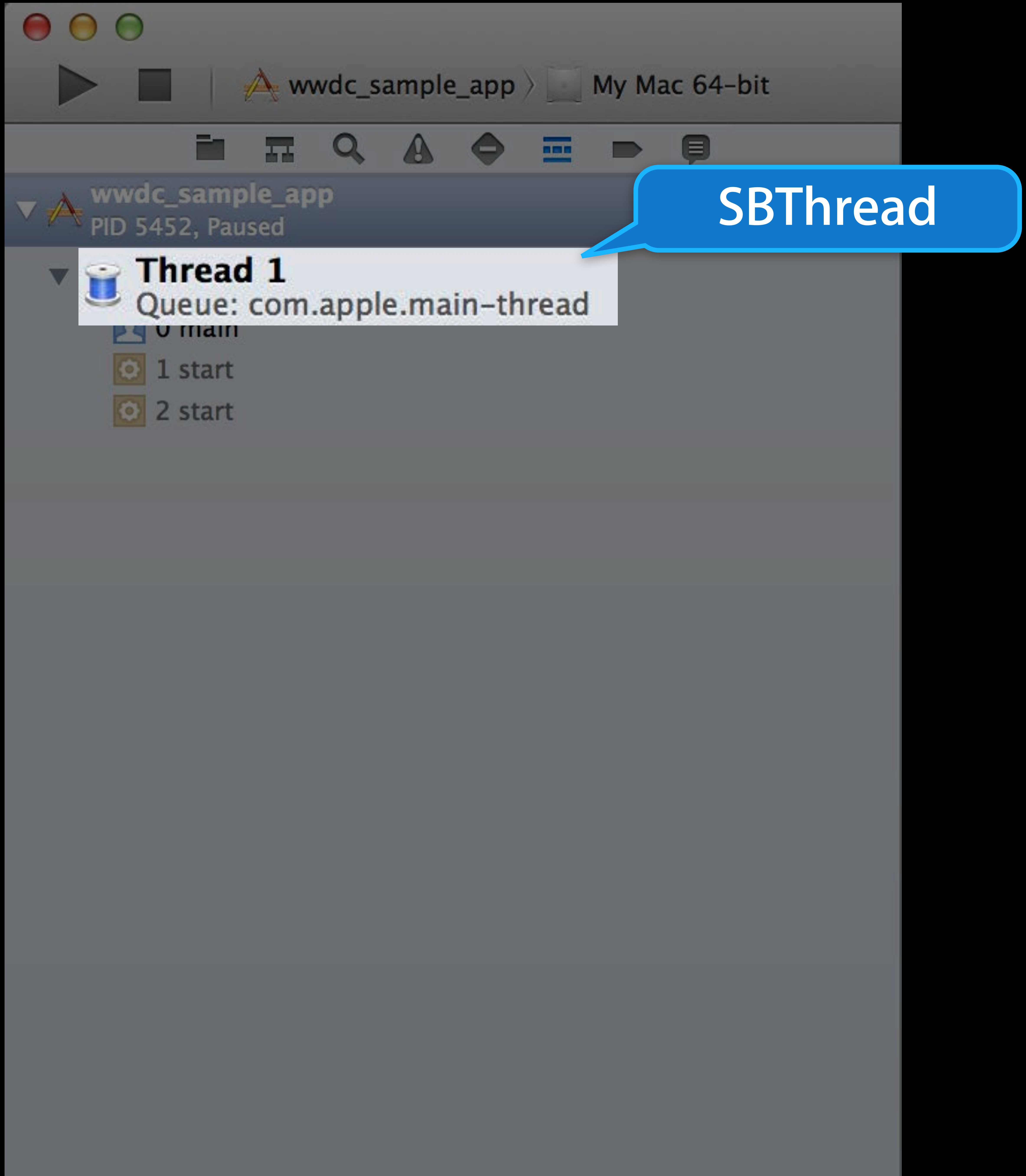
Thread 1
Queue: com.apple.main-thread

0 main
1 start
2 start

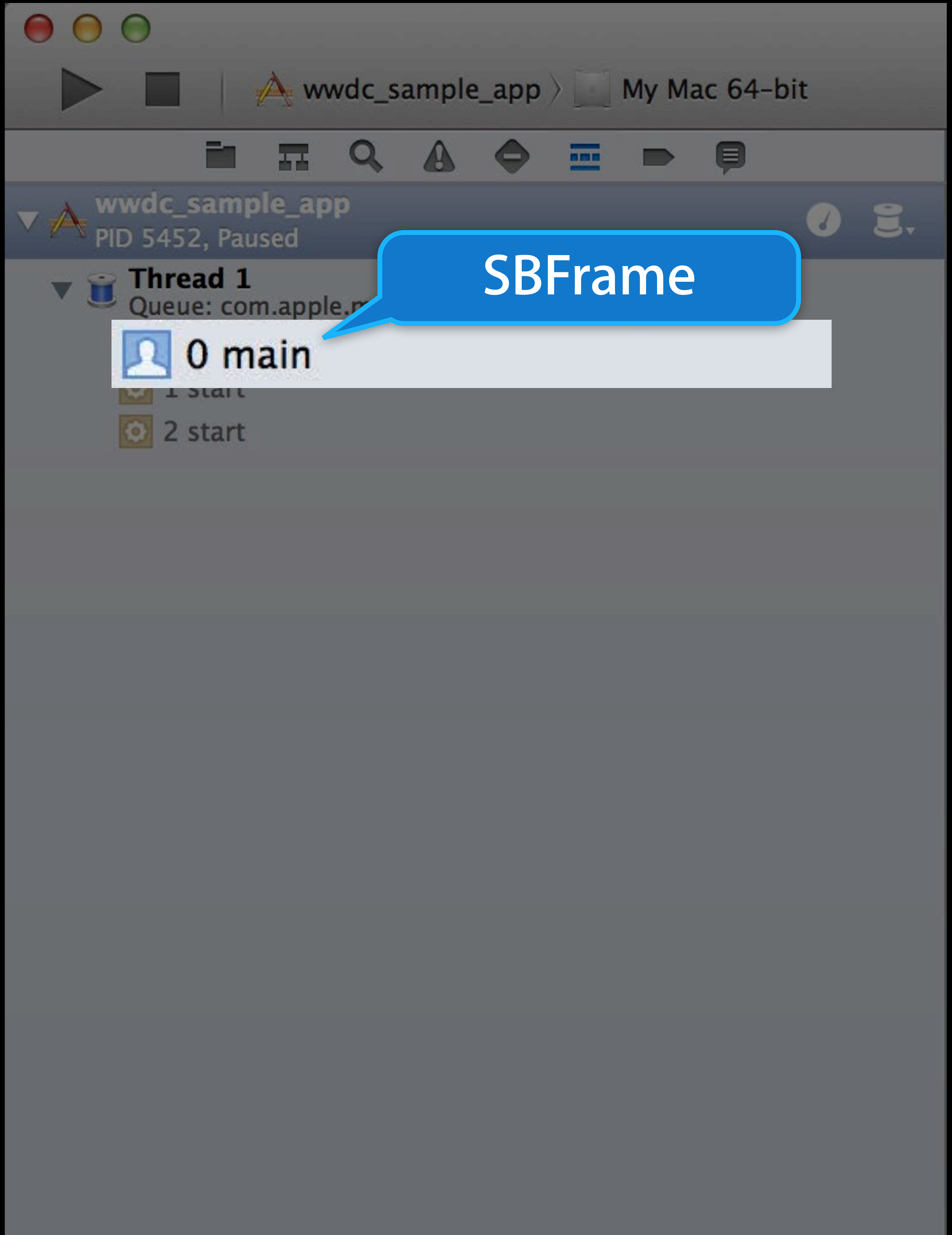
The LLDB Object Model



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

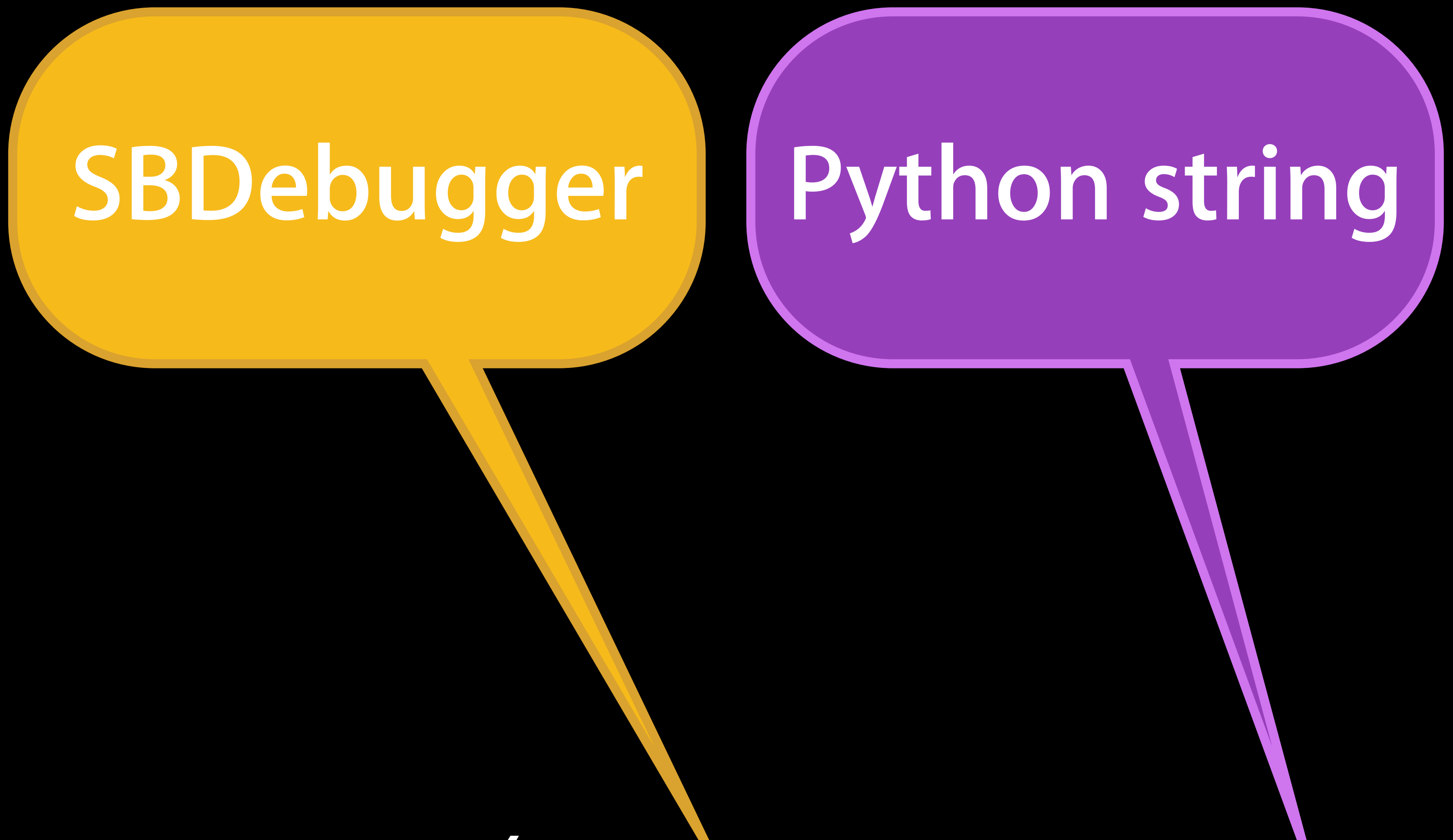
- Commands associate a name with a Python function
    - The function is invoked whenever the command is typed

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



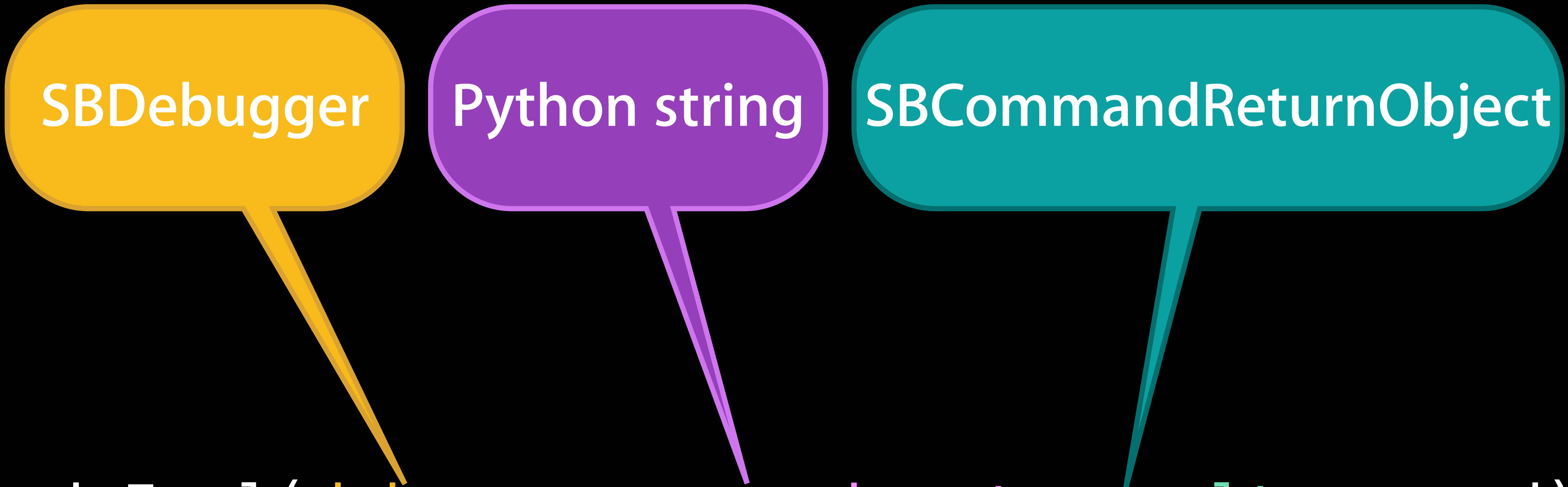
SBDebugger

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

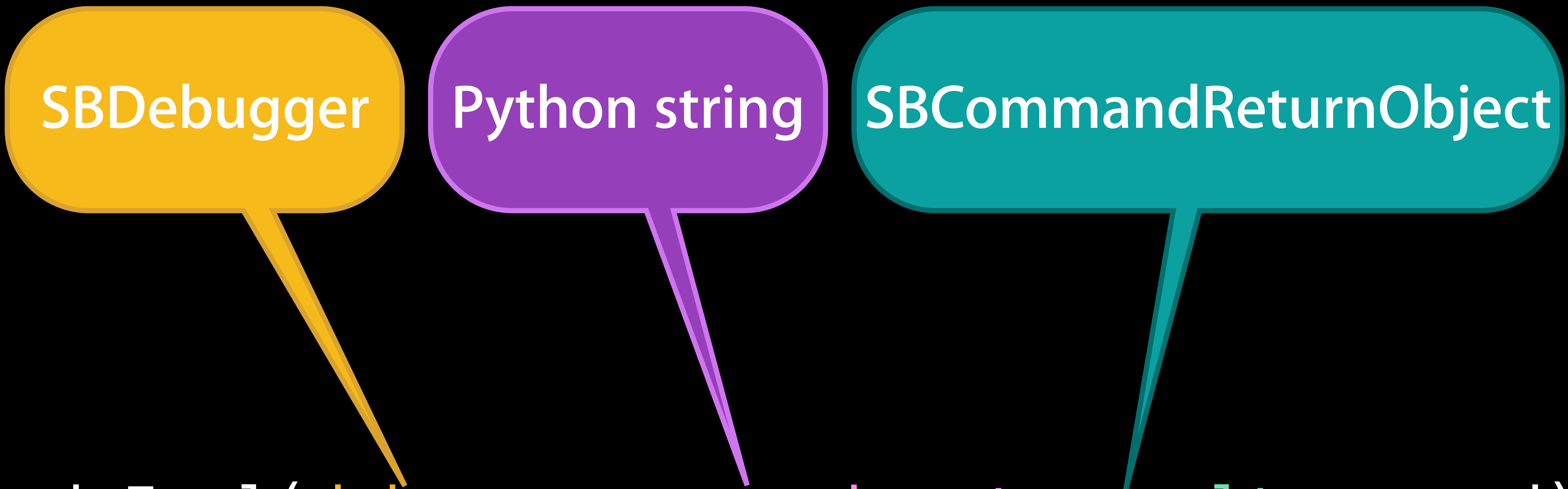
Python string

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

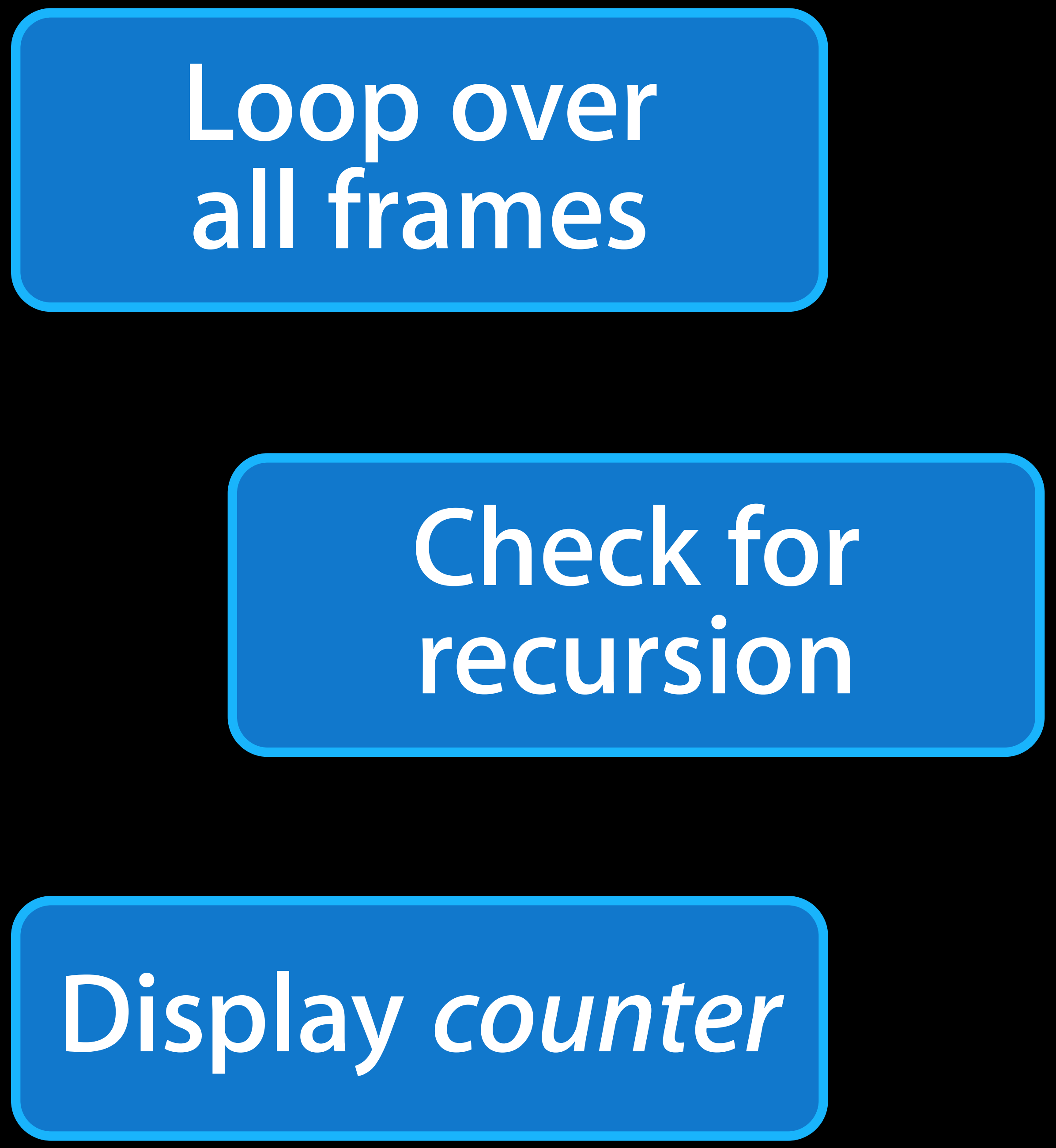
SBCommandReturnObject

```
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)
```

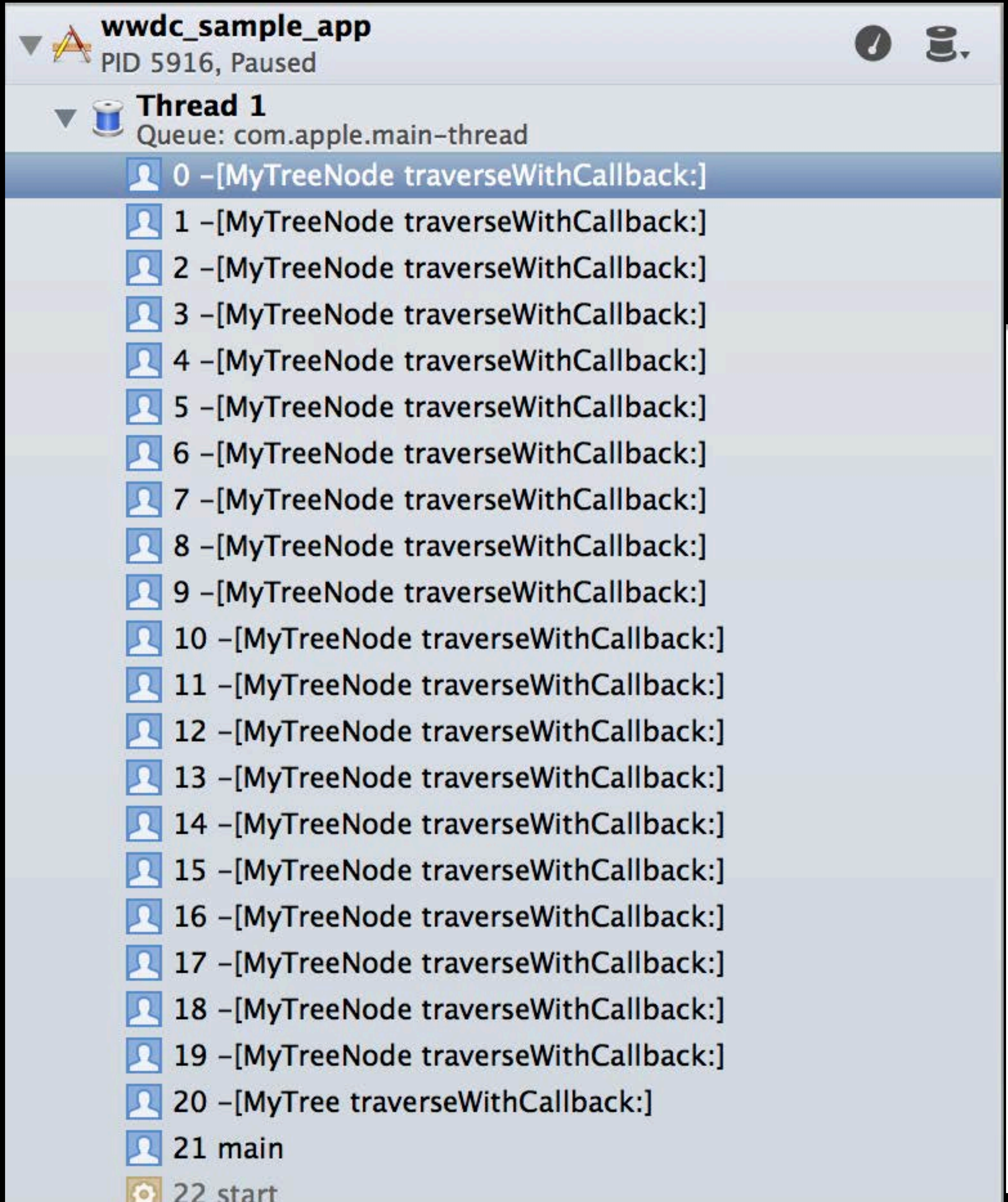
Example

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

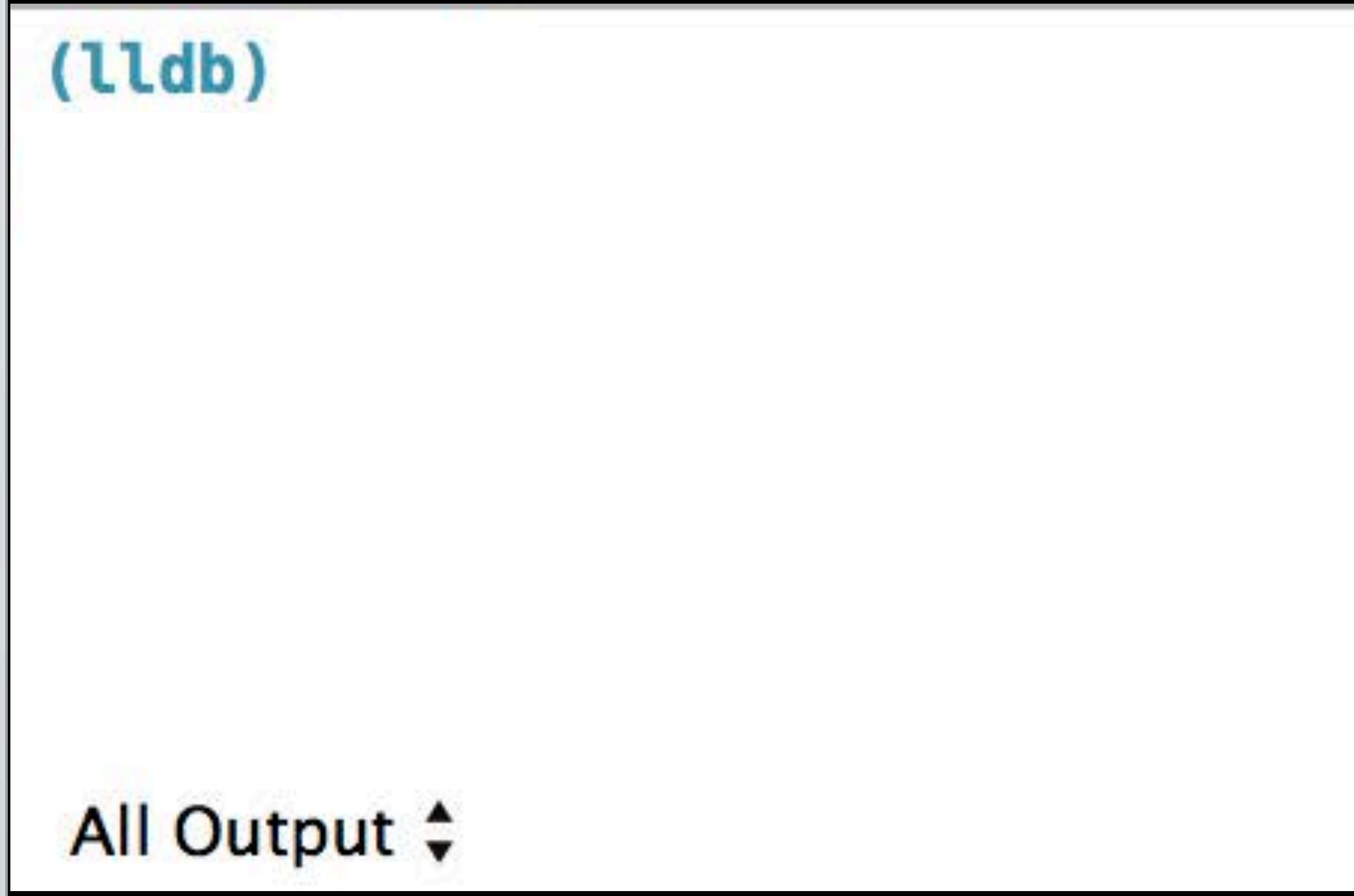
Calculate depth of a recursion



wwdc_sample_app
PID 5916, Paused

Thread 1
Queue: com.apple.main-thread

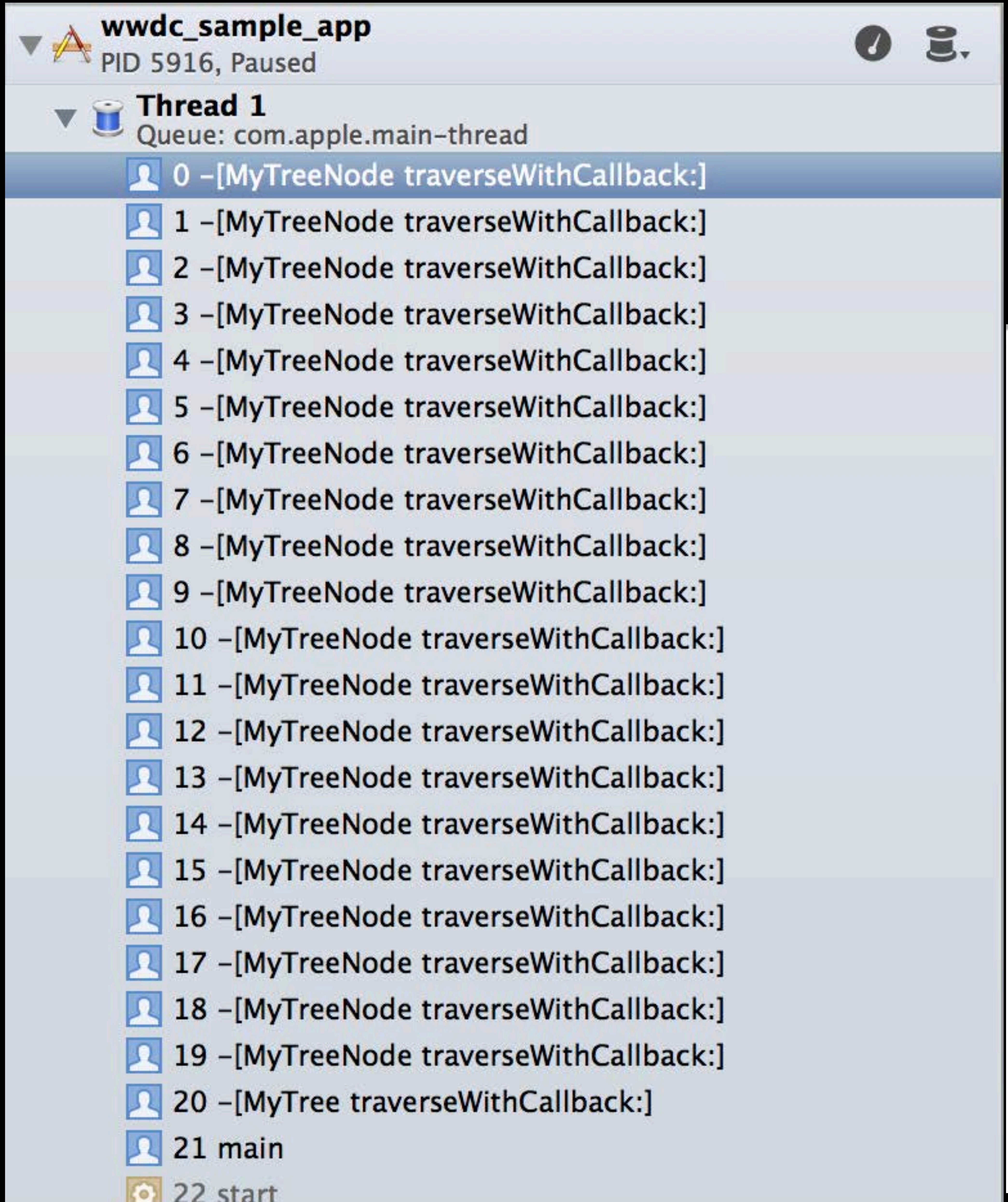
```
0 -[MyTreeNode traverseWithCallback:]
1 -[MyTreeNode traverseWithCallback:]
2 -[MyTreeNode traverseWithCallback:]
3 -[MyTreeNode traverseWithCallback:]
4 -[MyTreeNode traverseWithCallback:]
5 -[MyTreeNode traverseWithCallback:]
6 -[MyTreeNode traverseWithCallback:]
7 -[MyTreeNode traverseWithCallback:]
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
```



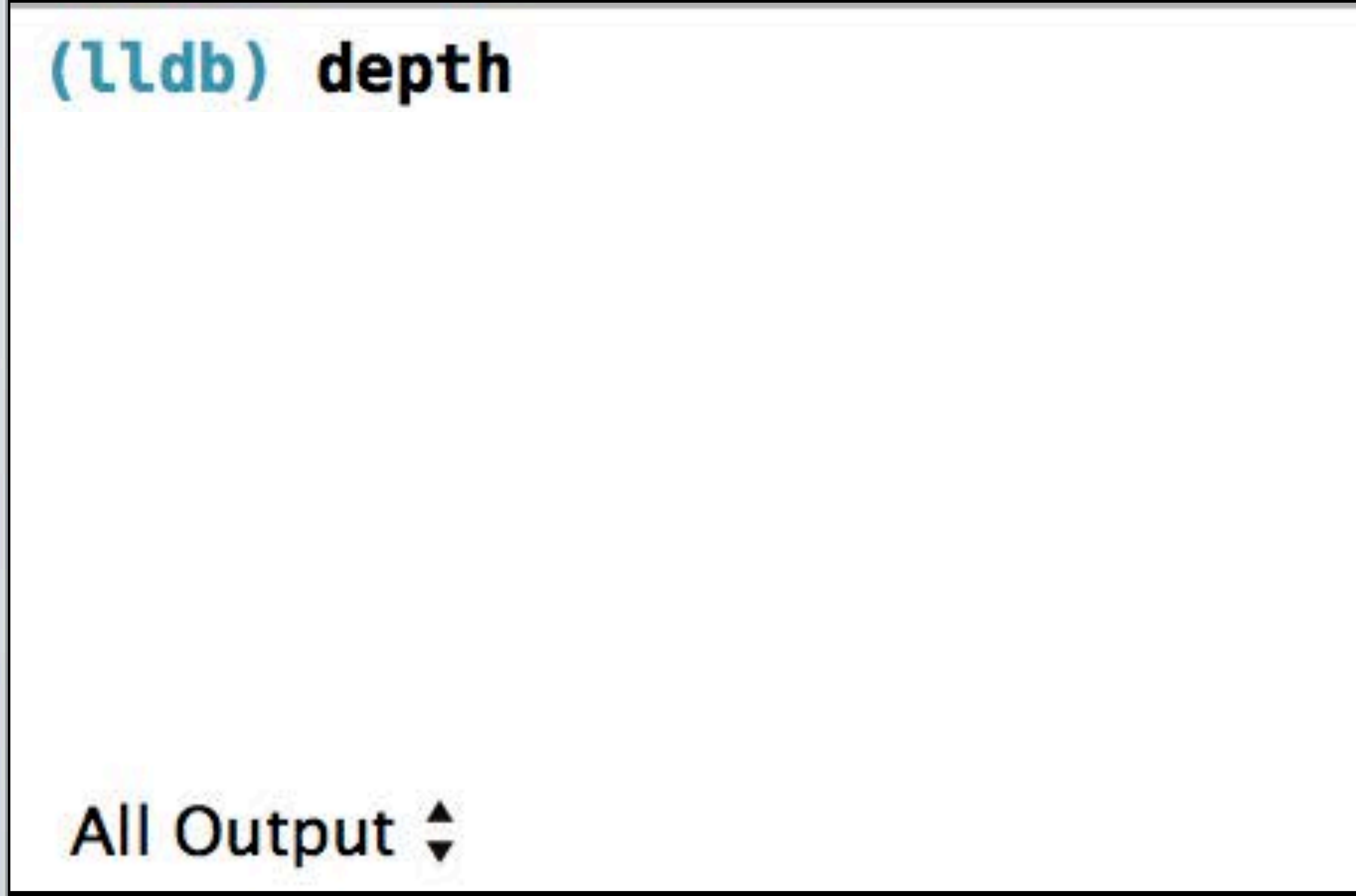
```
(lldb)
```

Example

Calculate depth of a recursion



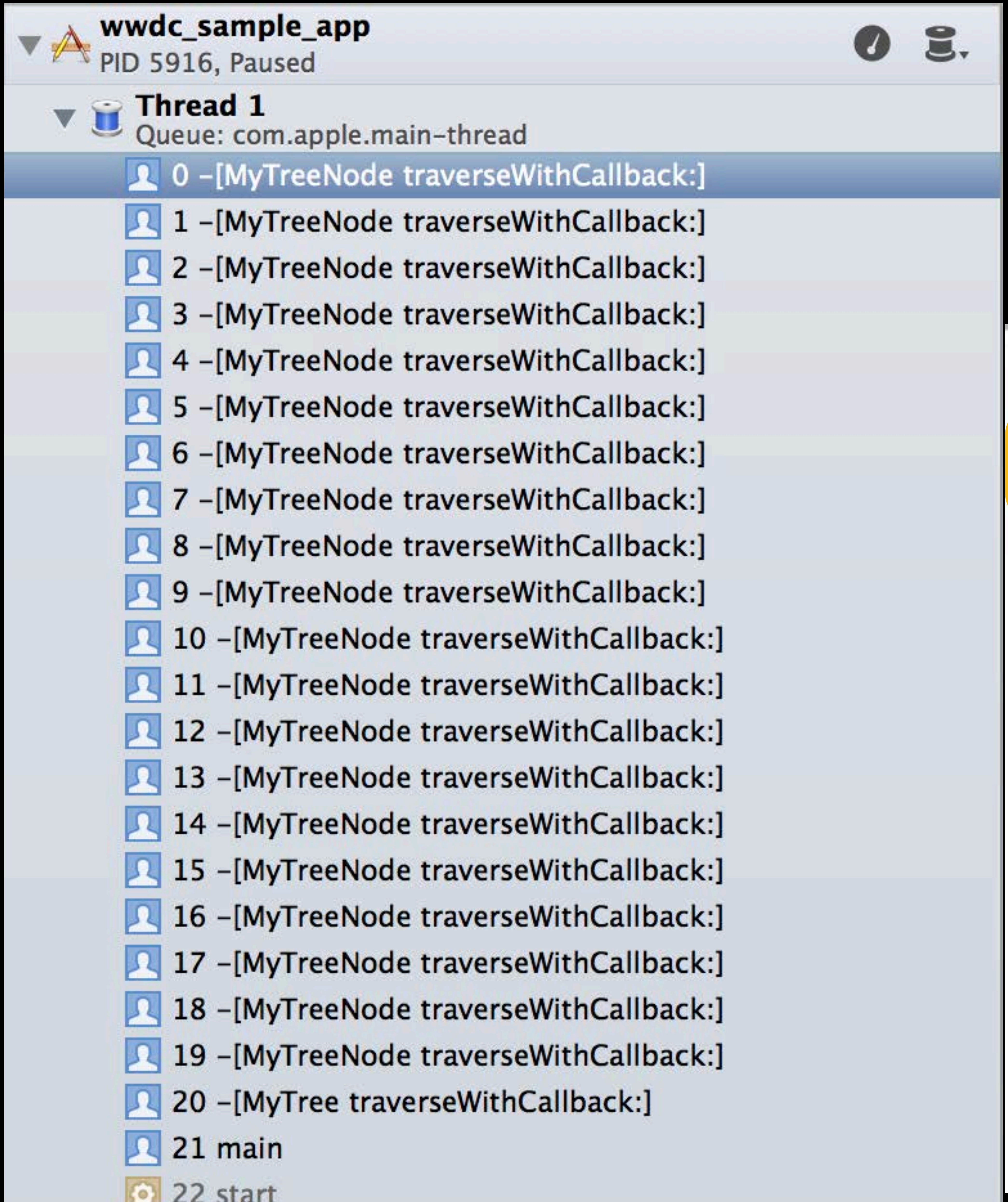
```
wwdc_sample_app
PID 5916, Paused
  Thread 1
  Queue: com.apple.main-thread
    0 -[MyTreeNode traverseWithCallback:]
    1 -[MyTreeNode traverseWithCallback:]
    2 -[MyTreeNode traverseWithCallback:]
    3 -[MyTreeNode traverseWithCallback:]
    4 -[MyTreeNode traverseWithCallback:]
    5 -[MyTreeNode 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) depth
```

Example

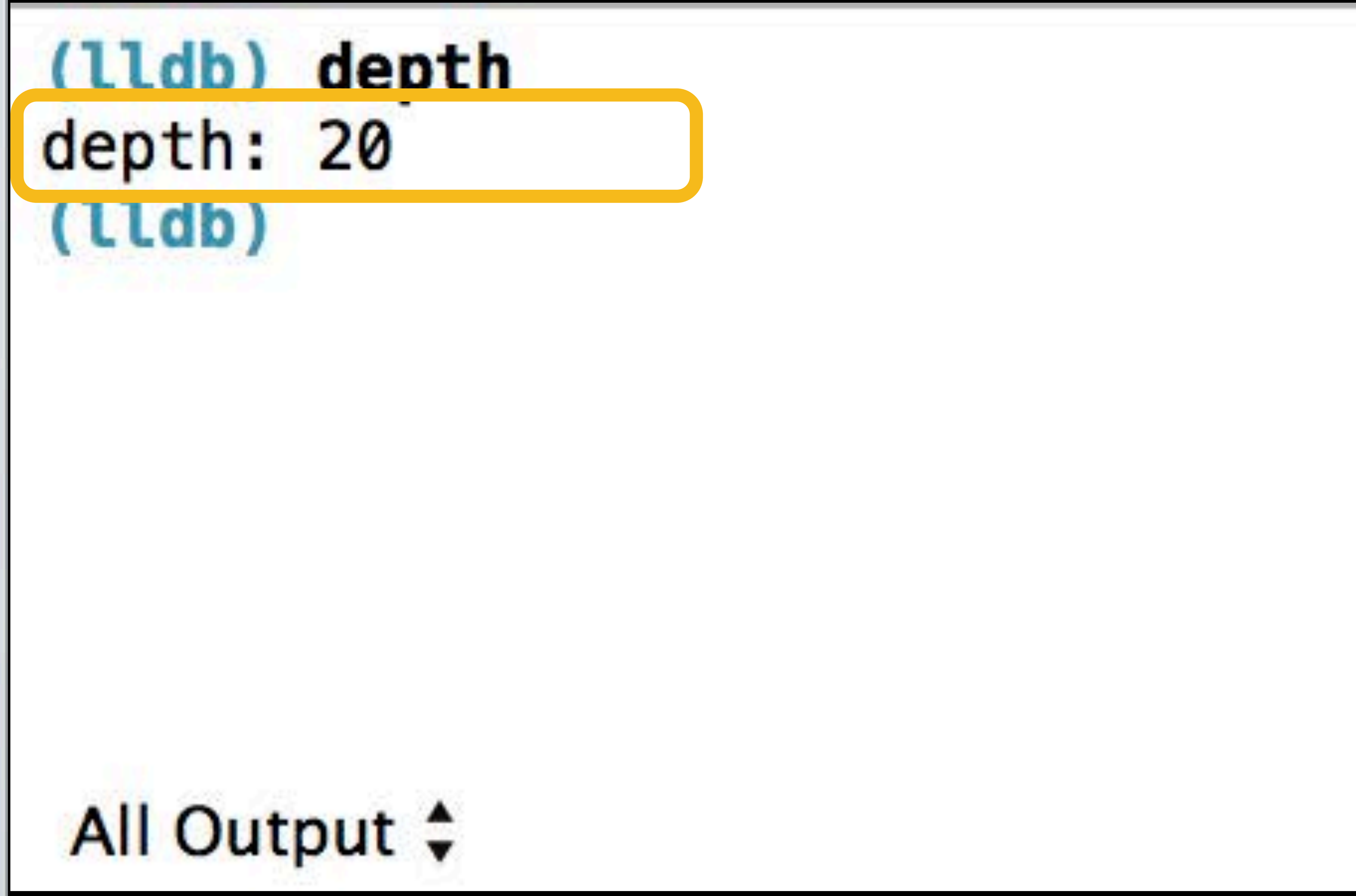
Calculate depth of a recursion



wwdc_sample_app
PID 5916, Paused

Thread 1
Queue: com.apple.main-thread

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

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

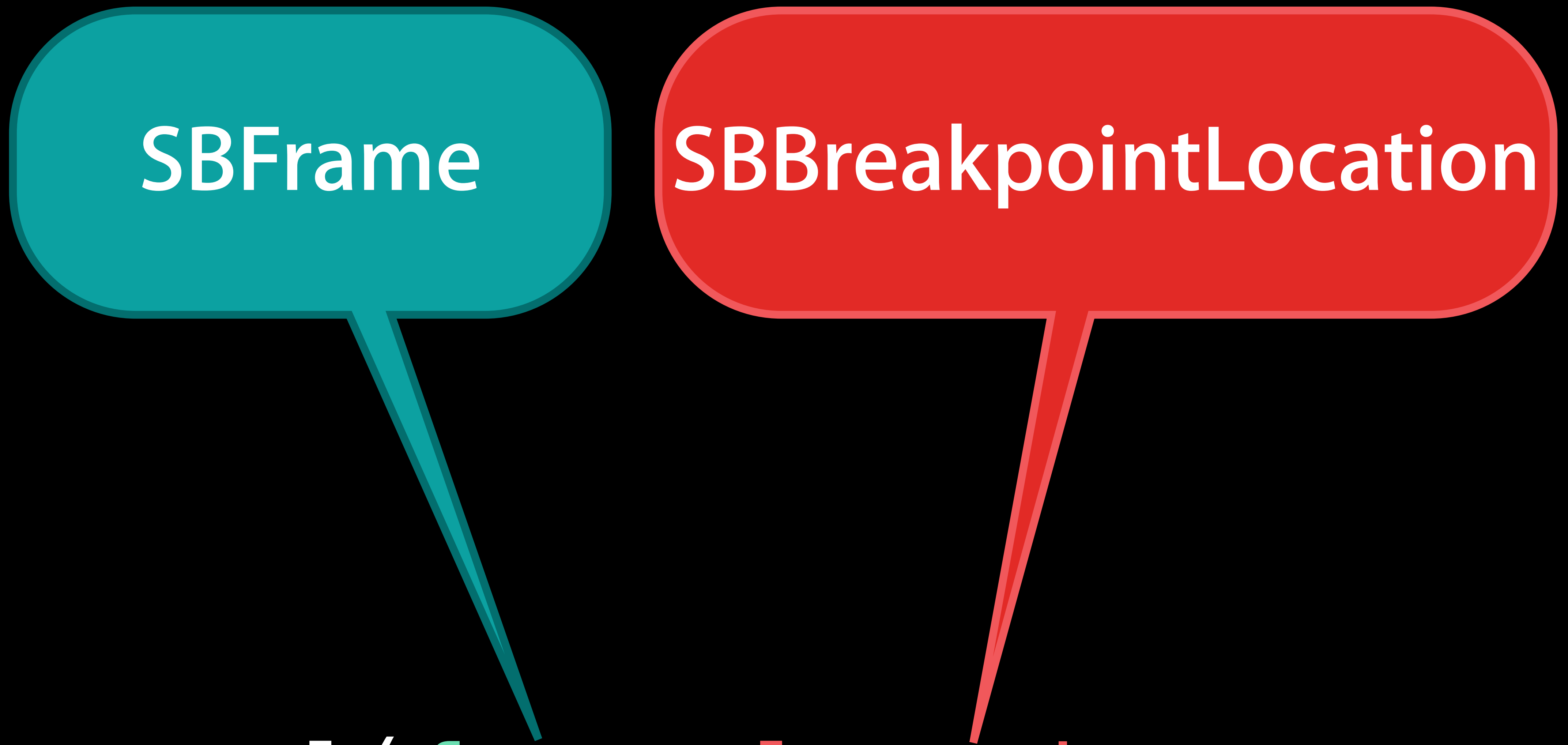
- Breakpoint actions associate a breakpoint with a Python function
    - The function is invoked whenever the breakpoint is hit
    - The function can return False to tell LLDB to continue your program

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



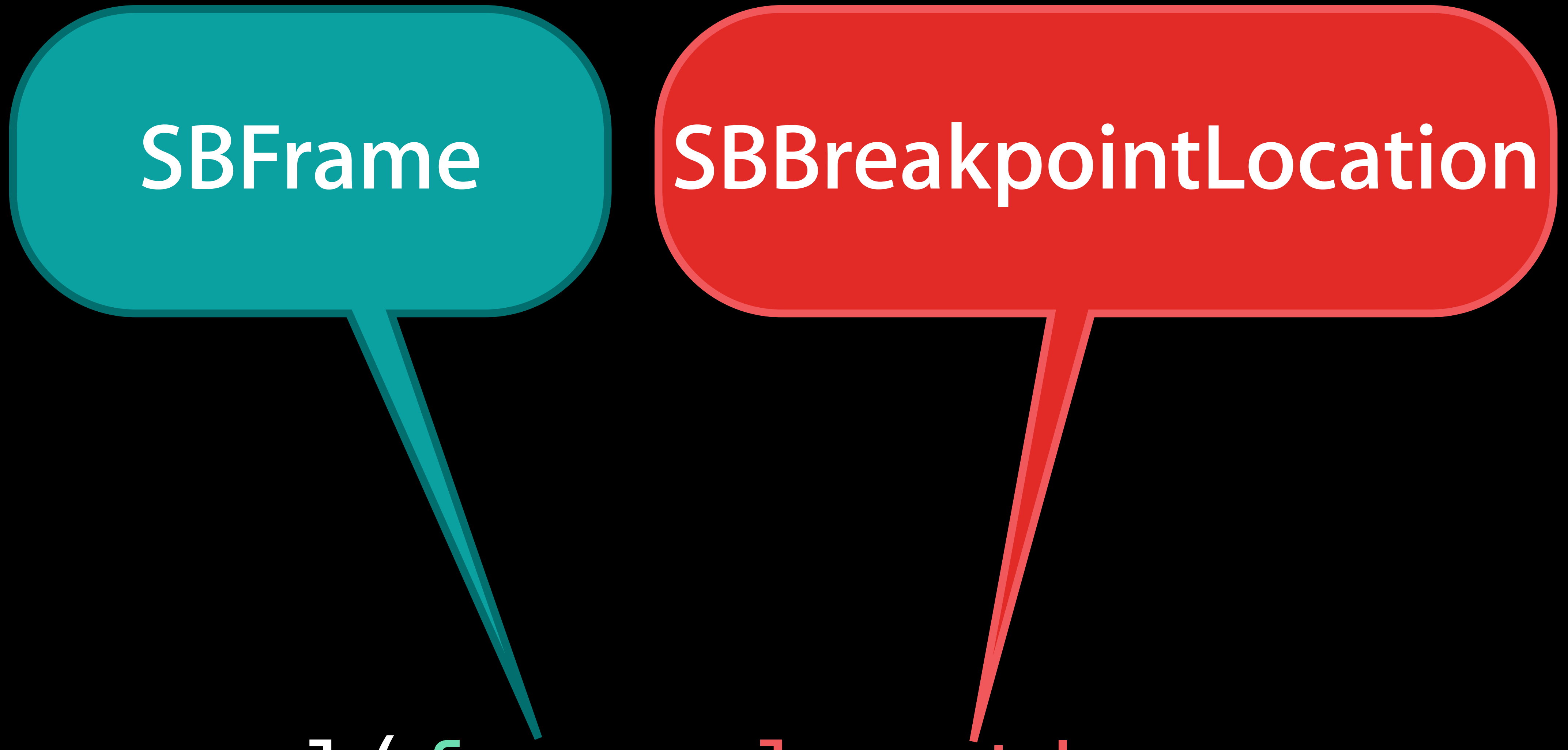
SBFrame

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!

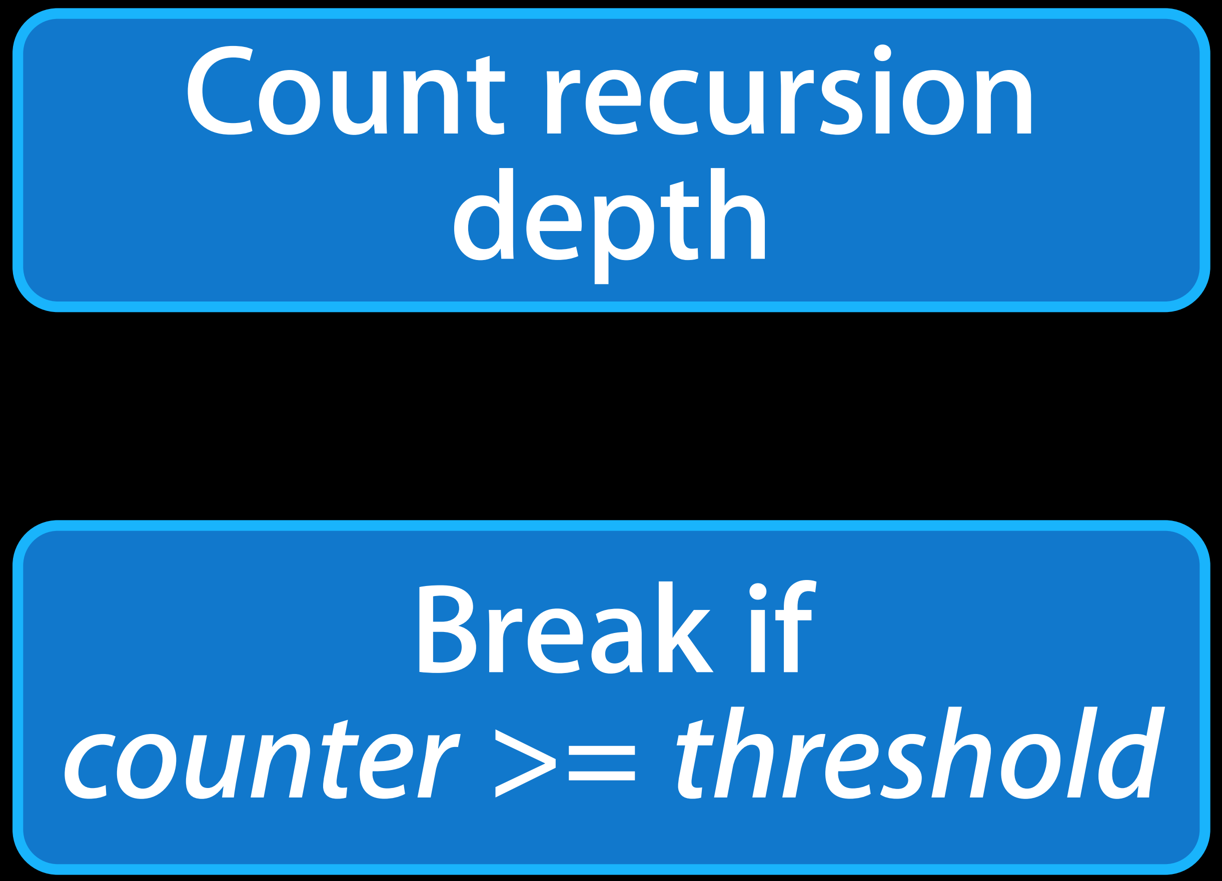
Example

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 >= threshold*

Example

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

Count recursion depth ✓

Break if *counter >= 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
```

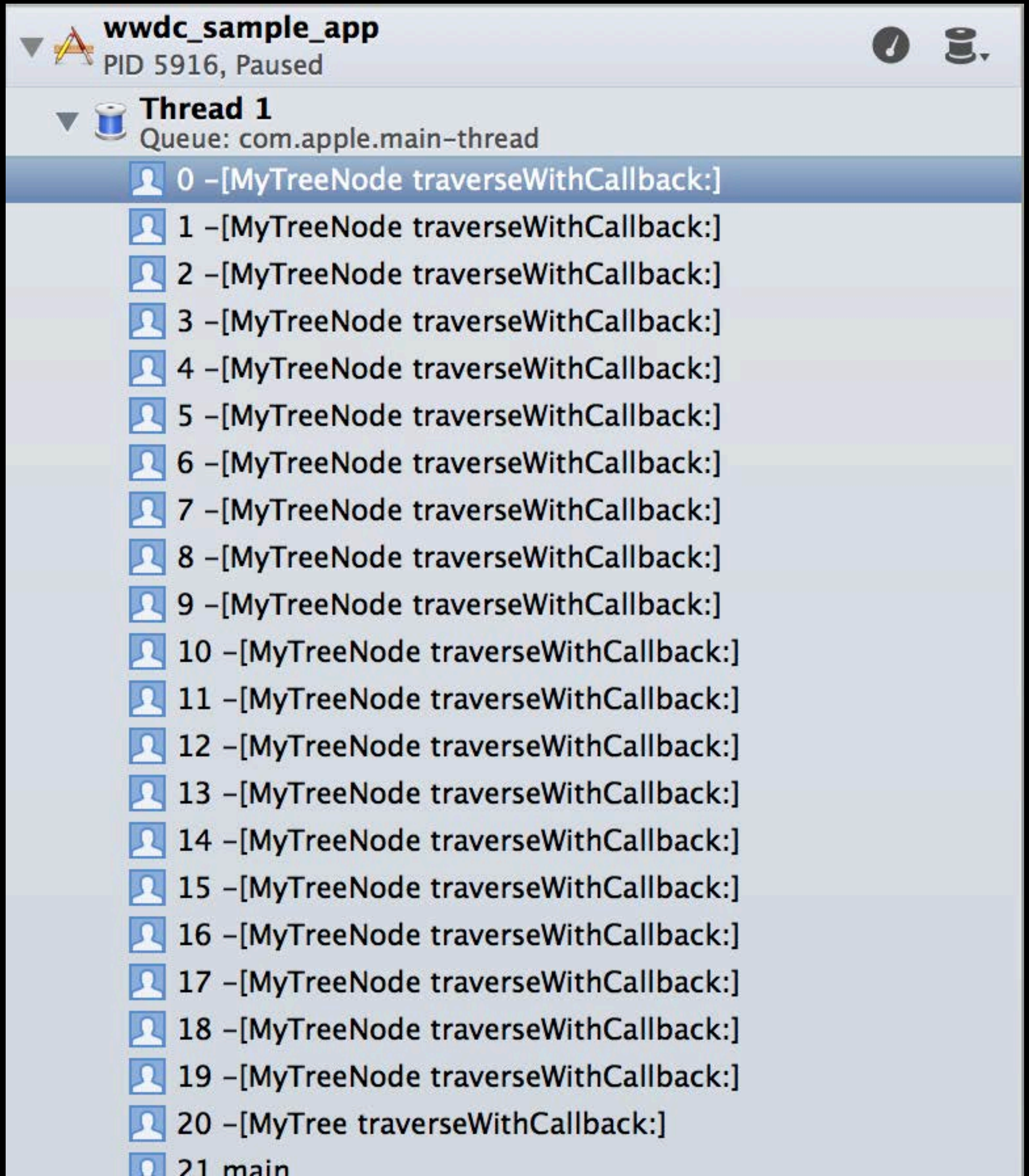
Example

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

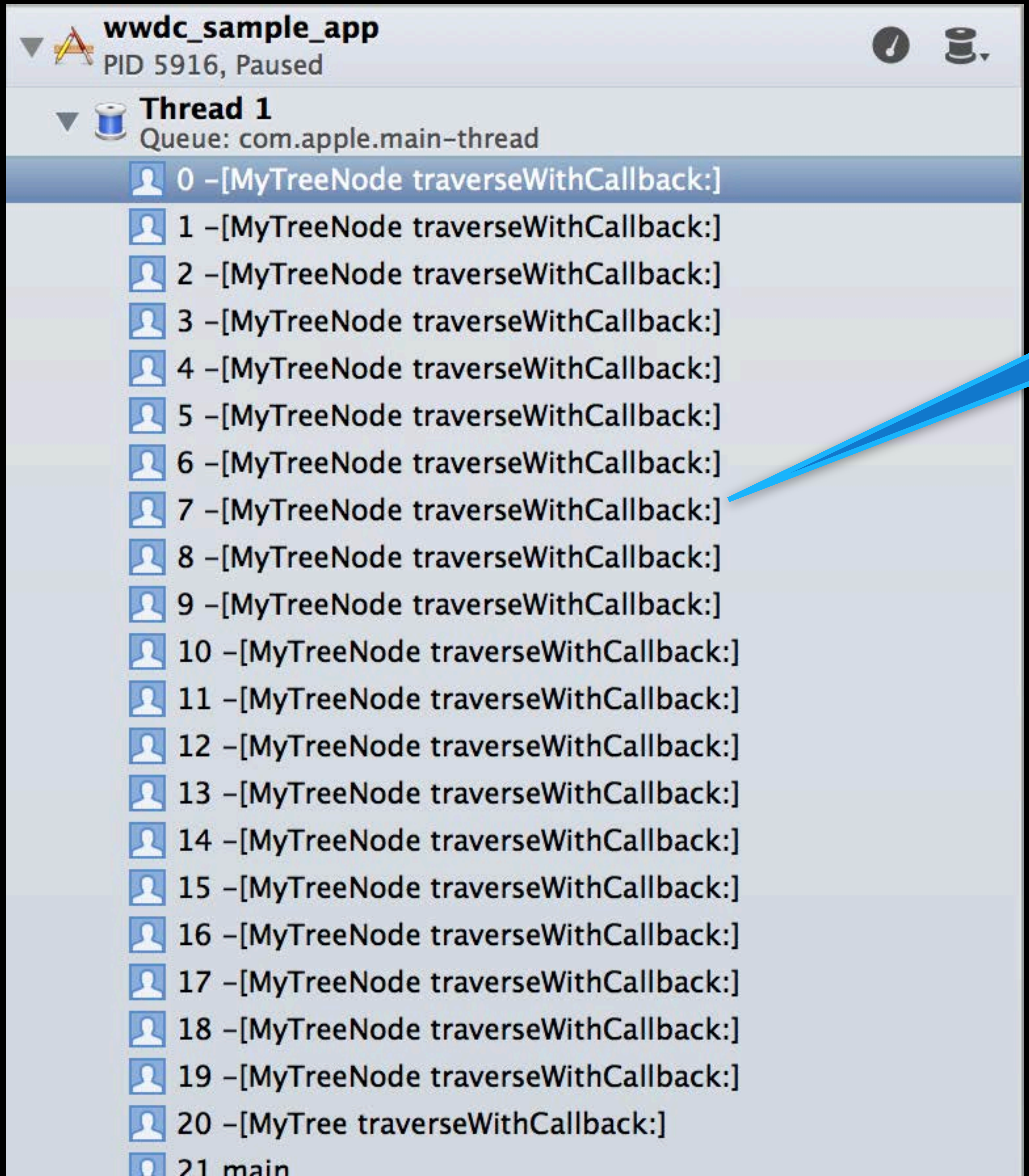
Stop if a recursion is more than *n* levels deep



```
wwdc_sample_app
PID 5916, Paused
Thread 1
Queue: com.apple.main-thread
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
```

Example

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



```
wwdc_sample_app
PID 5916, Paused
Thread 1
Queue: com.apple.main-thread
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
```

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