Tools

#WWDC14

Advanced Swift Debugging in LLDB

Debugging in a Swift world

Session 410
Enrico Granata
Debugger Engineer

Introduction

Introduction

Swift is the modern language of Cocoa

Introduction

Swift is the modern language of Cocoa

Your existing tools, improved

Introduction

Swift is the modern language of Cocoa

Your existing tools, improved

The debugger can help you explore in the context of your app

Introduction

Swift is the modern language of Cocoa

Your existing tools, improved

The debugger can help you explore in the context of your app

- And be productive

Introduction

Swift is the modern language of Cocoa

Your existing tools, improved

The debugger can help you explore in the context of your app

- And be productive

Swift feels awesome in LLDB

What You Will Learn

What You Will Learn

Swift types in LLDB

What You Will Learn

Swift types in LLDB

- Optional types

What You Will Learn

Swift types in LLDB

- Optional types
- Protocols

What You Will Learn

Swift types in LLDB

- Optional types
- Protocols
- Generics

What You Will Learn

Swift types in LLDB

- Optional types
- Protocols
- Generics

Debugging combined Swift and Objective-C

What You Will Learn

Swift types in LLDB

- Optional types
- Protocols
- Generics

Debugging combined Swift and Objective-C

Stepping

What You Will Learn

Swift types in LLDB

- Optional types
- Protocols
- Generics

Debugging combined Swift and Objective-C

Stepping

Data formatters for Swift objects

What You Will Learn

Swift types in LLDB

- Optional types
- Protocols
- Generics

Debugging combined Swift and Objective-C

Stepping

Data formatters for Swift objects

Name uniqueness in Swift

Optional Types

Optional Types

Optionals introduce indirection

Optional Types

Optionals introduce indirection

- Is it there? Is it not?

Optional Types

Optionals introduce indirection

- Is it there? Is it not?

LLDB implicitly unwraps whenever possible

Optional Types

Optionals introduce indirection

- Is it there? Is it not?

LLDB implicitly unwraps whenever possible

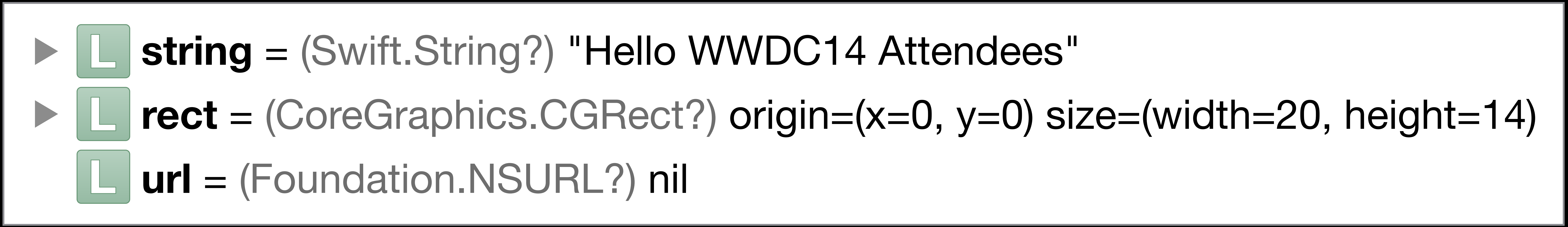
`nil` used consistently for the no-value situation

Optional Types

```
var string: String? = "Hello WWDC14 Attendees"
var rect: NSRect? = NSMakeRect(0, 0, 20, 14)
var url: NSURL? = nil
```

Optional Types

```swift
var string: String? = "Hello WWDC14 Attendees"
var rect: NSRect? = NSMakeRect(0, 0, 20, 14)
var url: NSURL? = nil
```



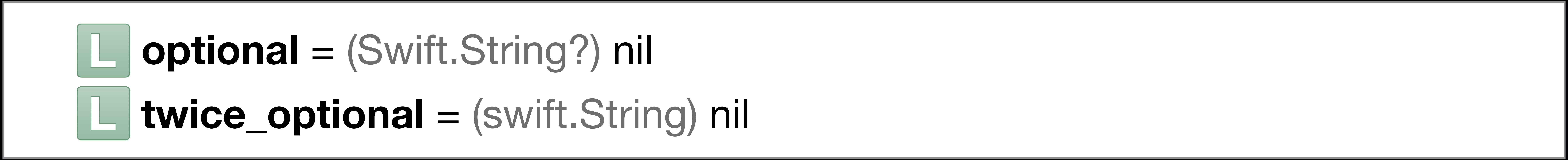
```
▶ L string = (Swift.String?) "Hello WWDC14 Attendees"
▶ L rect = (CoreGraphics.CGRect?) origin=(x=0, y=0) size=(width=20, height=14)
  L url = (Foundation.NSURL?) nil
```

Double Optional

```swift
var optional: String? = nil
var twice_optional: String?? = Optional.Some(nil)
```

Double Optional

```swift
var optional: String? = nil
var twice_optional: String?? = Optional.Some(nil)
```



```
optional = (Swift.String?) nil
twice_optional = (swift.String) nil
```

By default, propagate `nil` upwards

Raw Display

```
(lldb) fr v -R twice_optional
```

Raw Display

```
(lldb) fr v -R twice_optional
```

Raw Display

```
(lldb) fr v -R twice_optional
(Swift.String??) twice_optional = Some {
```

Raw Display

```
(lldb) fr v -R twice_optional
(Swift.String??) twice_optional = Some {
  Some = None {
```

Raw Display

```
(lldb) fr v -R twice_optional
(Swift.String??) twice_optional = Some {
  Some = None {
    Some = {
      core = {
        _baseAddress = {
          value = 0x0000000000000000
        }
        _countAndFlags = {
          value = 0
        }
        _owner = None {
          Some = {
            instance_type = 0x0000000000000000
          }
        }
      }
    }
  }
}
```

Types

Types

What is a type?

Types

What is a type?

Lots of answers

Types

What is a type?

Lots of answers

a classification that determines a set of valid values and operations for data

Types

What is a type?

Lots of answers

a classification that determines a set of valid values and operations for data

Data can have multiple types

Static/Dynamic Types

Static/Dynamic Types

```
var url: AnyObject = NSURL(string: “http://www.apple.com”)
```

Variables have a declared (aka static) type

Static/Dynamic Types

```
var url: AnyObject = NSURL(string: “http://www.apple.com”)
```

Variables have a declared (aka static) type

Static/Dynamic Types

```
var url: AnyObject = NSURL(string: “http://www.apple.com”)
```

Variables have a declared (aka static) type

```
url.hash
```

Static/Dynamic Types

```
var url: AnyObject = NSURL(string: “http://www.apple.com”)
```

Variables have a declared (aka static) type

```
url.hash
```

Which “hash” gets called?

Static/Dynamic Types

```
var url: AnyObject = NSURL(string: “http://www.apple.com”)
```

Variables have a declared (aka static) type

```
url.hash
```

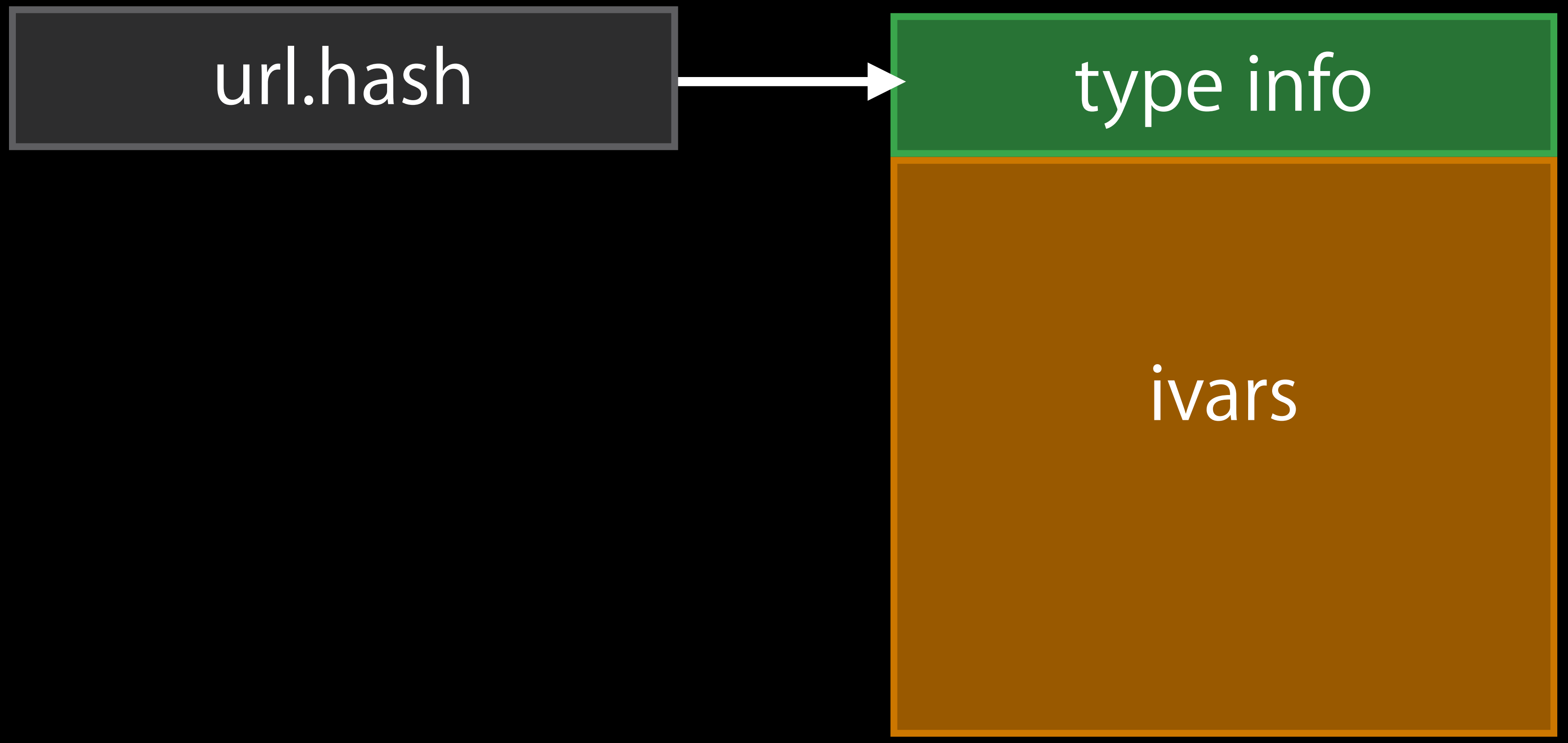
Which “hash” gets called?

The one that matches the runtime (aka *dynamic*) type

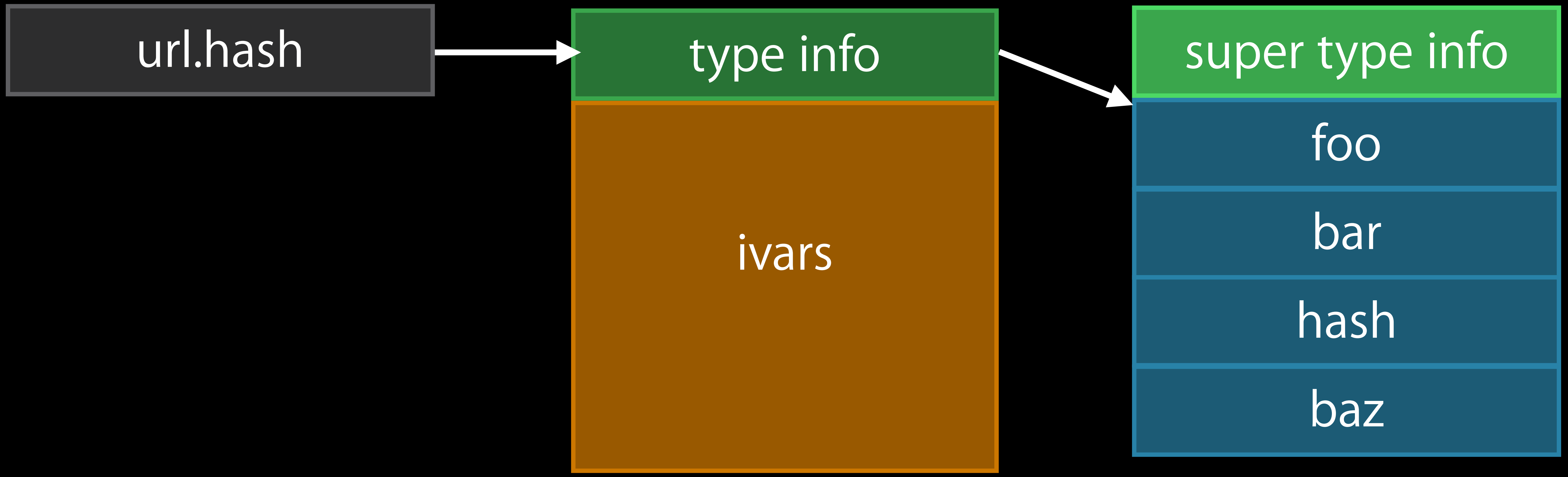
Dynamic Dispatch

url.hash

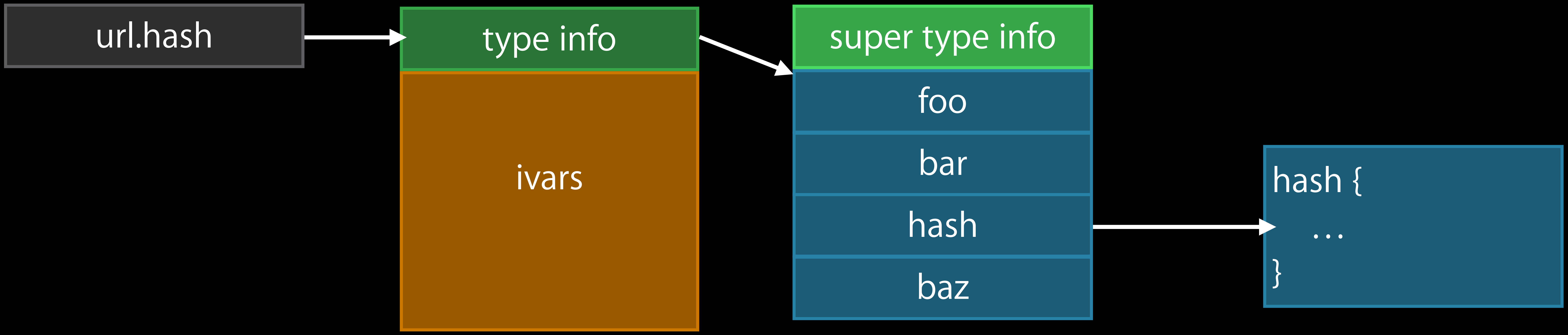
Dynamic Dispatch



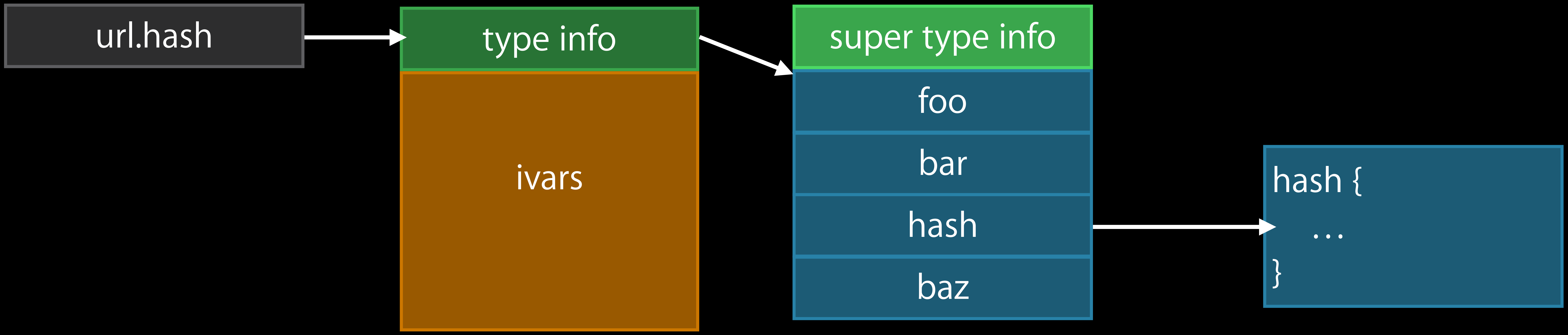
Dynamic Dispatch



Dynamic Dispatch

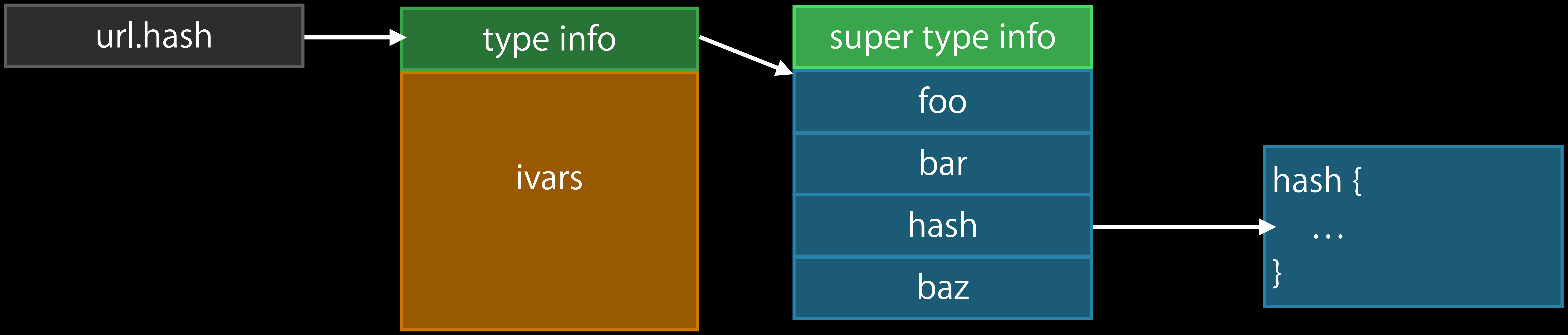


Dynamic Dispatch



What if the runtime can’t find an implementation?

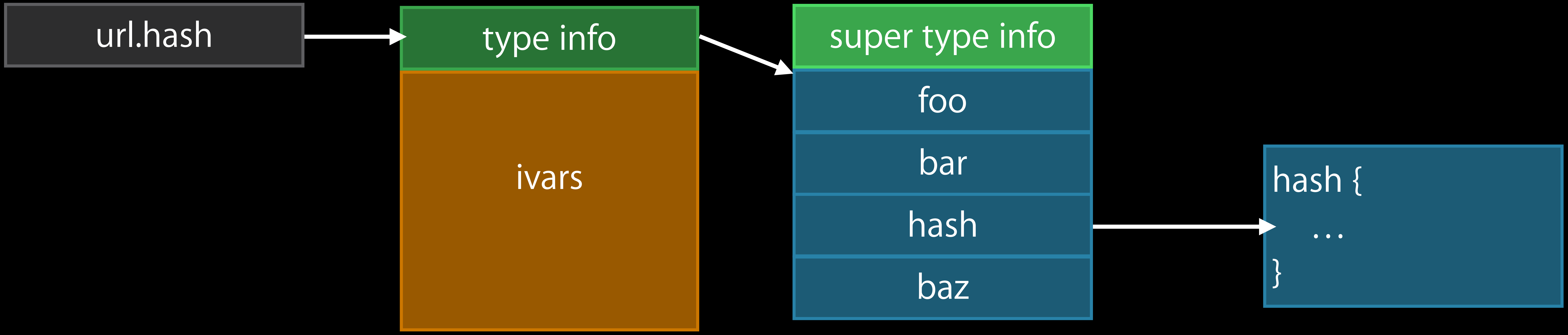
Dynamic Dispatch



What if the runtime can’t find an implementation?

- Ask the superclass

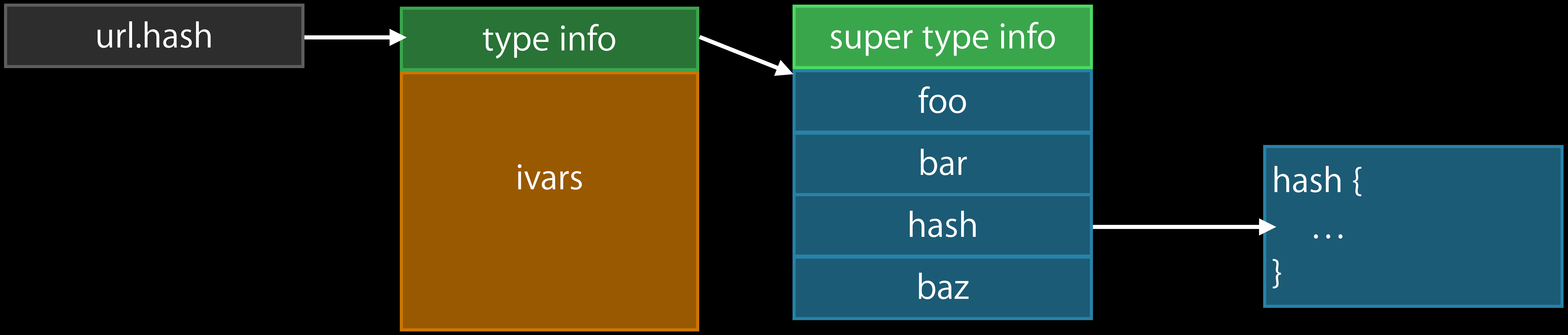
Dynamic Dispatch



What if the runtime can't find an implementation?

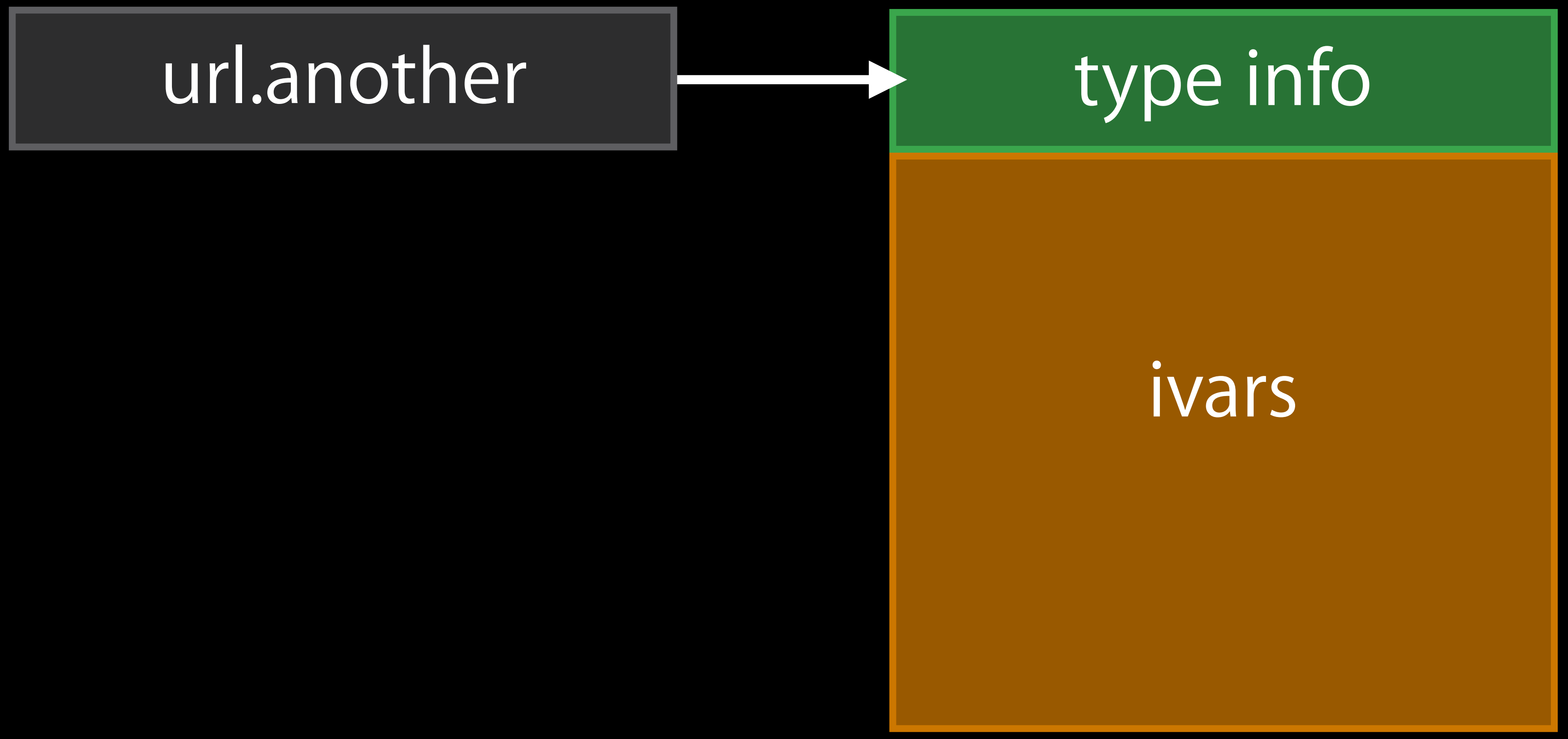
Ask the superclass

Dynamic Dispatch

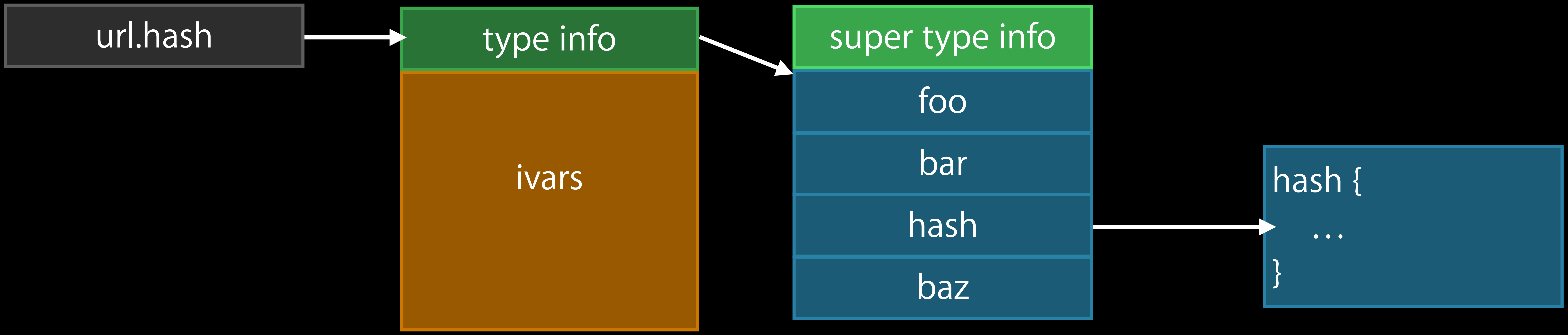


What if the runtime can't find an implementation?

Ask the superclass

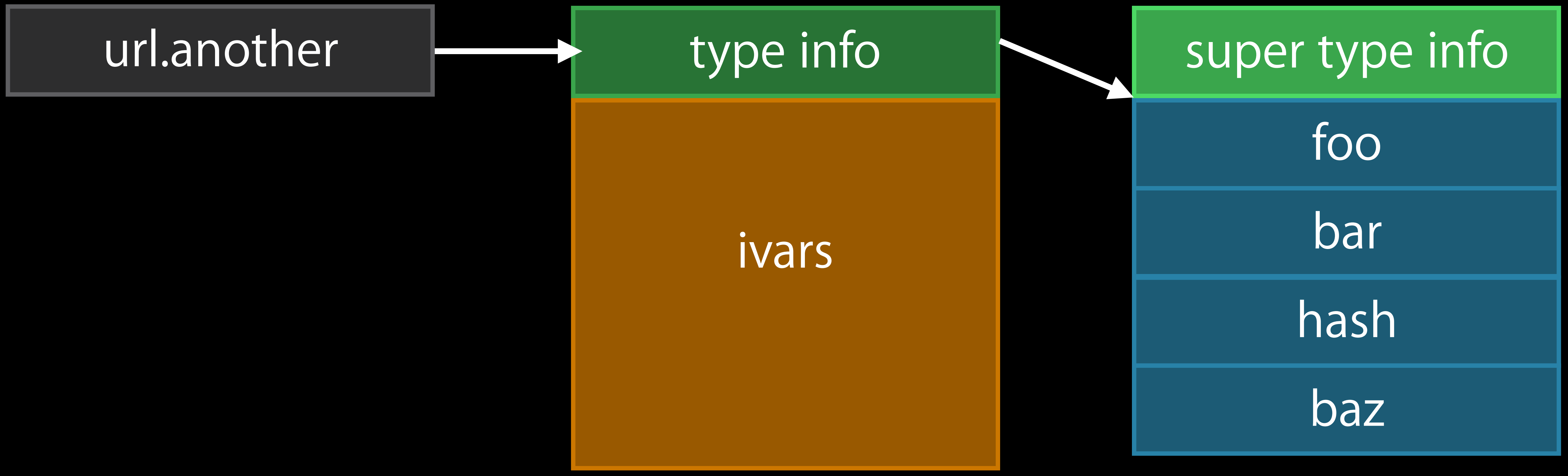


Dynamic Dispatch

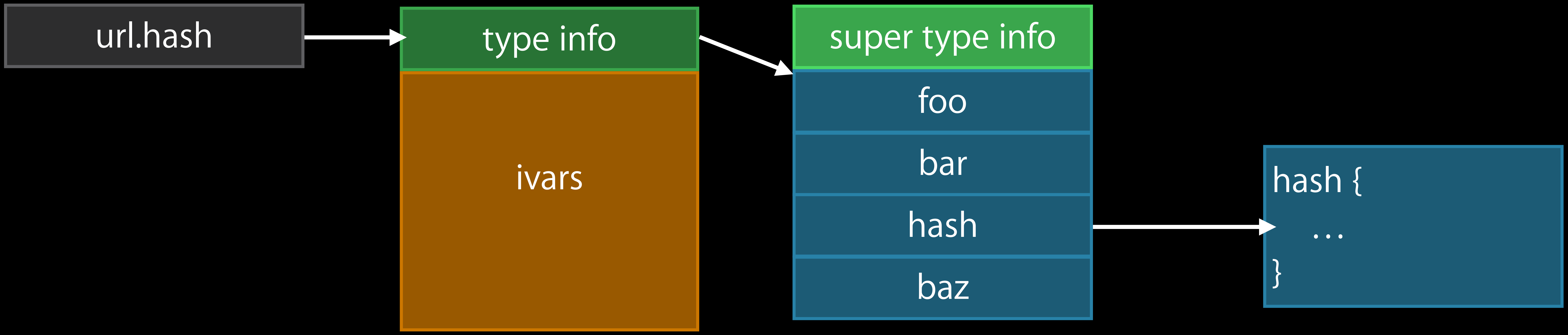


What if the runtime can't find an implementation?

Ask the superclass

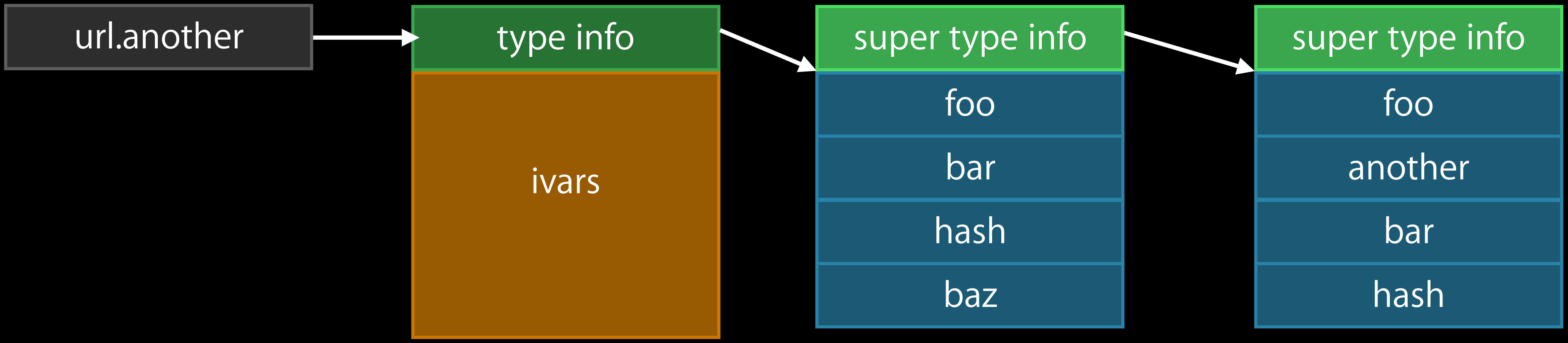


Dynamic Dispatch

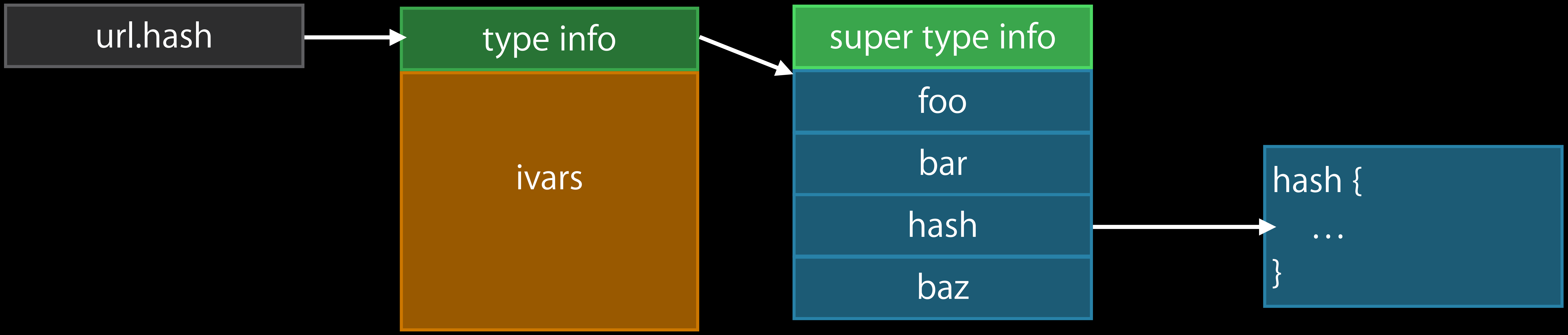


What if the runtime can't find an implementation?

Ask the superclass

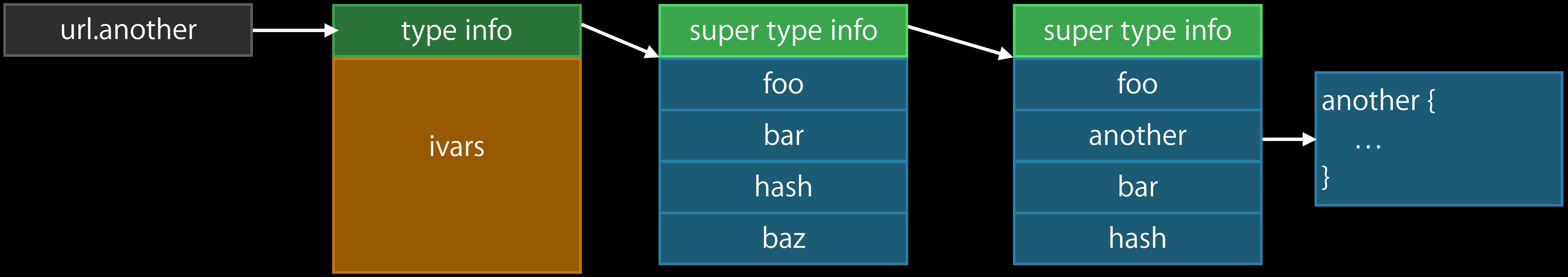


Dynamic Dispatch

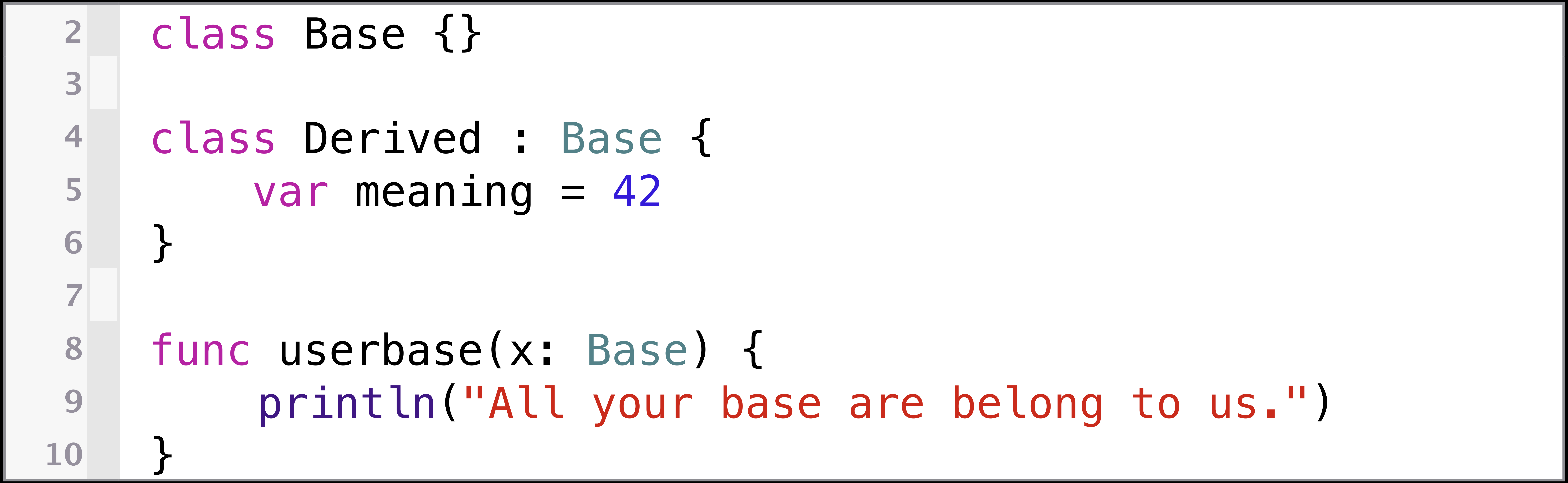


What if the runtime can't find an implementation?

Ask the superclass



Dynamic Types in LLDB



```swift
class Base {}

class Derived : Base {
    var meaning = 42
}

func userbase(x: Base) {
    println("All your base are belong to us.")
}
```

Dynamic Types in LLDB

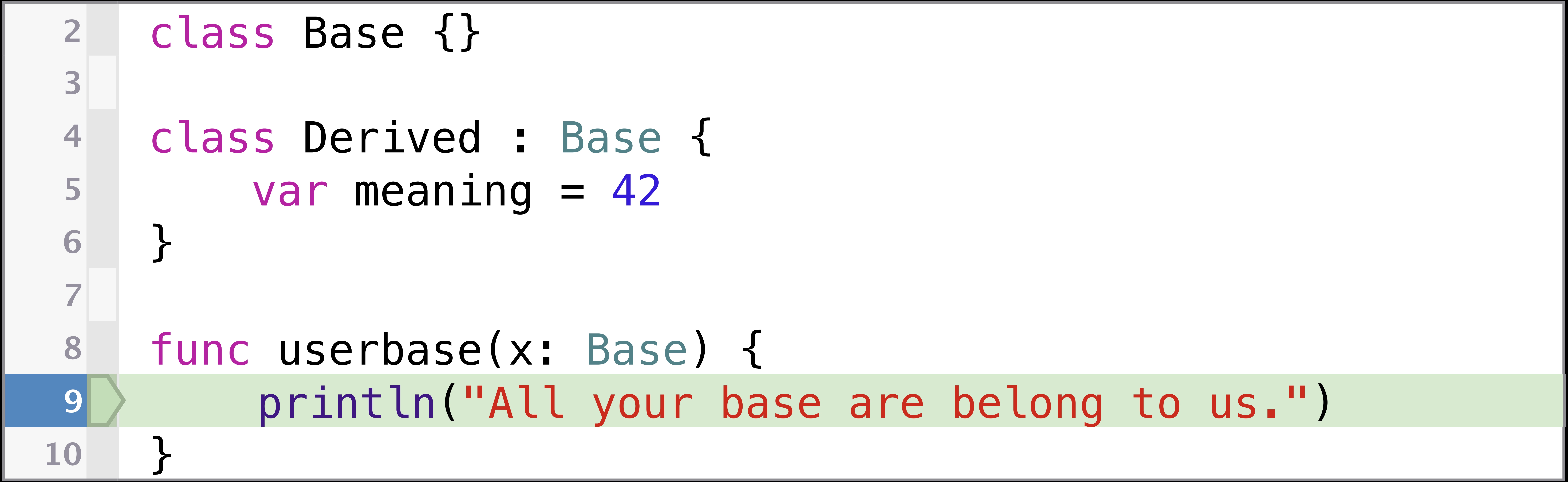
```
class Base {}

class Derived : Base {
    var meaning = 42
}

func userbase(x: Base) {
    println("All your base are belong to us.")
}
```

```
useBase (aDerived)
```

Dynamic Types in LLDB

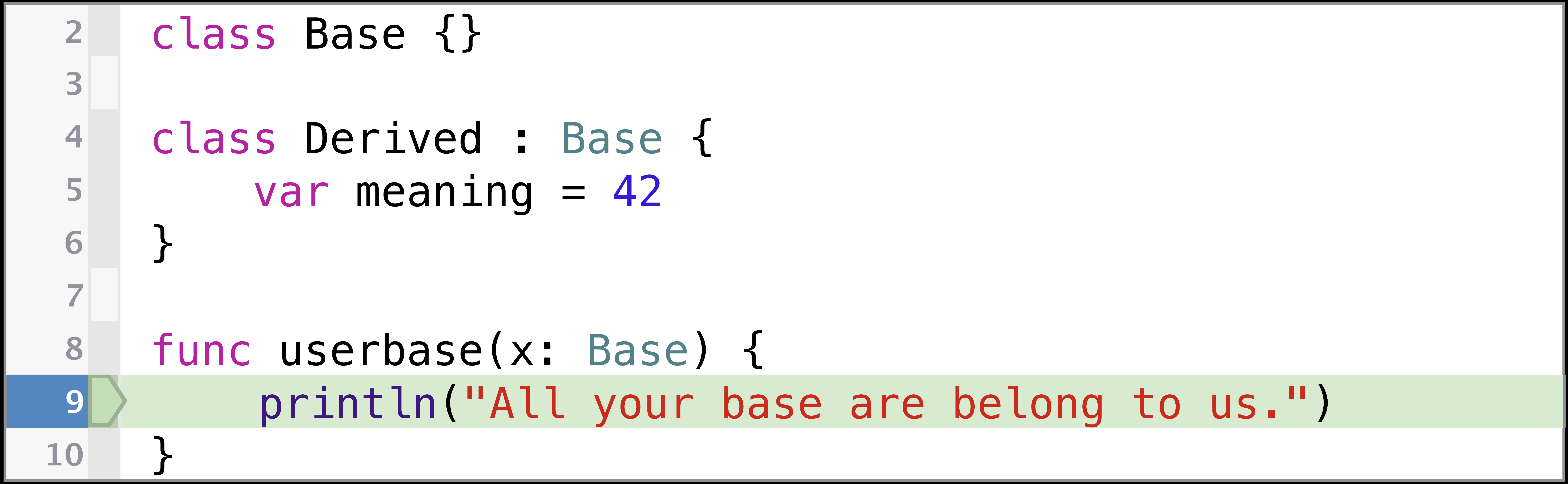
```swift
class Base {}

class Derived : Base {
    var meaning = 42
}

func userbase(x: Base) {
    println("All your base are belong to us.")
}
```

Dynamic Types in LLDB

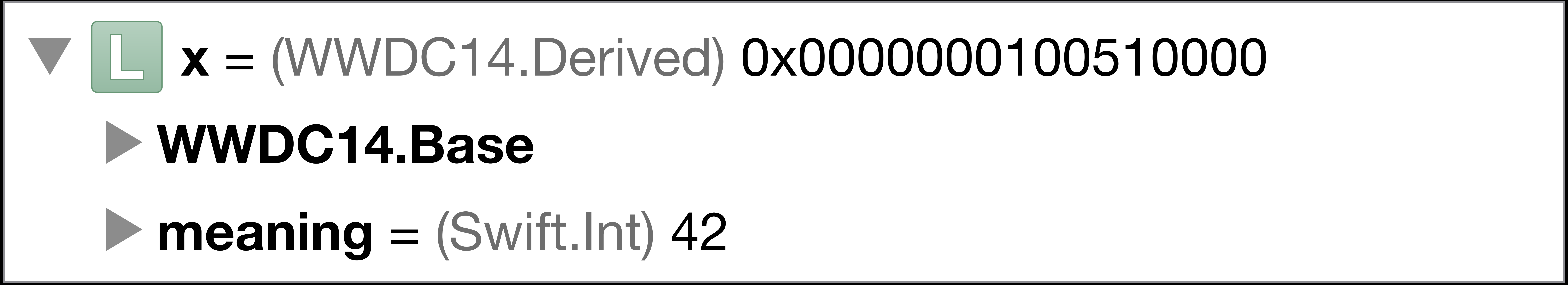


```
class Base {}

class Derived : Base {
    var meaning = 42
}

func userbase(x: Base) {
    println("All your base are belong to us.")
}
```

Dynamic Types in LLDB



```swift
class Base {}

class Derived : Base {
    var meaning = 42
}

func userbase(x: Base) {
    println("All your base are belong to us.")
}
```



```
▼ L x = (WWDC14.Derived) 0x0000000100510000
    ▶ WWDC14.Base
    ▶ meaning = (Swift.Int) 42
```

Protocols

Protocols

Protocols are types in Swift

Protocols

Protocols are types in Swift

- Variables can have protocol types

Protocols

Protocols are types in Swift

- Variables can have protocol types
- Protocols can be included in function signatures

Protocols

Protocols are types in Swift

- Variables can have protocol types
- Protocols can be included in function signatures

Variables of protocol type are limited

Protocols

Protocols are types in Swift

- Variables can have protocol types
- Protocols can be included in function signatures

Variables of protocol type are limited

- To only reveal what the protocol allows

Protocols

Protocols are types in Swift

- Variables can have protocol types
- Protocols can be included in function signatures

Variables of protocol type are limited

- To only reveal what the protocol allows

LLDB opens the curtain for you

Protocols

Protocols are types in Swift

- Variables can have protocol types
- Protocols can be included in function signatures

Variables of protocol type are limited

- To only reveal what the protocol allows

LLDB opens the curtain for you

- And it shows you the full value

Protocols

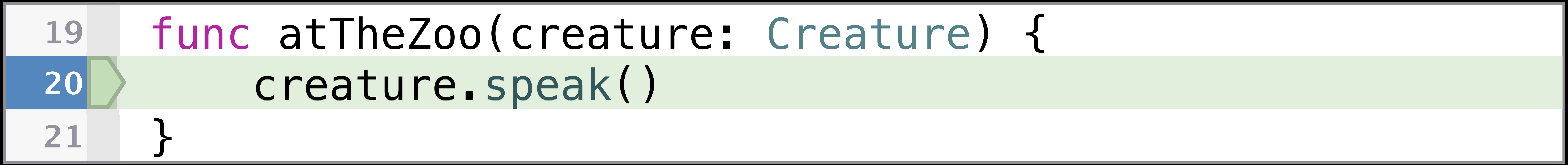
```swift
protocol Creature {
     func speak()
}

class Cat: Creature {
     func speak() {
         println("Meow. Purr")
     }
}
class Dog: Creature {
     func speak() {
         println(“Woof!”)
     }
}
```

```swift
func atTheZoo(creature: Creature) {
    creature.speak()
}
```

Protocols

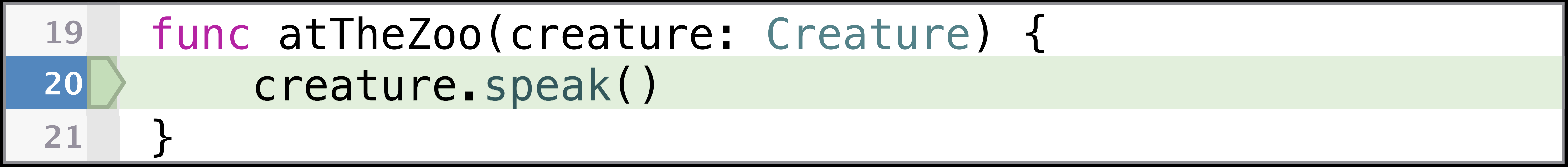
```swift
func atTheZoo(creature: Creature) {
    creature.speak()
}
```

Protocols

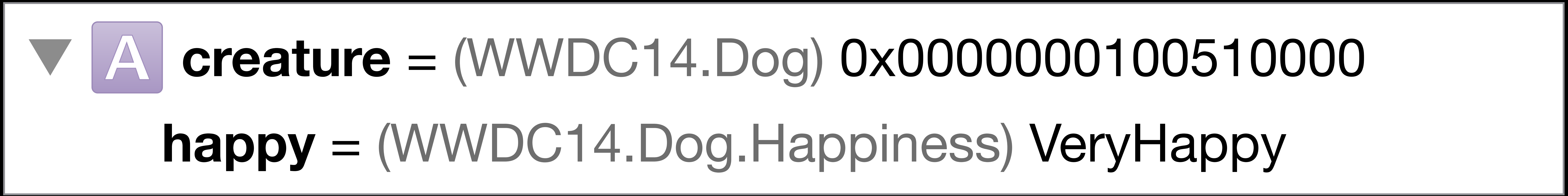


```
func atTheZoo(creature: Creature) {
    creature.speak()
}
```

Protocols



```
func atTheZoo(creature: Creature) {
    creature.speak()
}
```



```
▼ A creature = (WWDC14.Dog) 0x0000000100510000
      happy = (WWDC14.Dog.Happiness) VeryHappy
```

Protocols

Protocols

Console users beware

Protocols

Console users beware

```
(lldb) fr v creature
(Creature) creature = {
  payload_data_0 = 0x0000000100510000 -> 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
  payload_data_1 = 0x00007fff5fbffa10
  payload_data_2 = 0x0000000100002b52 WWDC14`WWDC14.play_with_hierarchy () -> () + 66 at hierarchy.swift:20
  instance_type = 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
  protocol_witness_0 = 0x0000000100007510 protocol witness table for WWDC14.Dog : WWDC14.Creature
}
```

Protocols

Console users beware

```
(lldb) fr v creature
(Creature) creature = {
  payload_data_0 = 0x0000000100510000 -> 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
  payload_data_1 = 0x00007fff5fbffa10
  payload_data_2 = 0x0000000100002b52 WWDC14`WWDC14.play_with_hierarchy () -> () + 66 at hierarchy.swift:20
  instance_type = 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
  protocol_witness_0 = 0x0000000100007510 protocol witness table for WWDC14.Dog : WWDC14.Creature
}
```

```
(lldb) fr v -d r creature
```

Protocols

Console users beware

```
(lldb) fr v creature
(Creature) creature = {
   payload_data_0 = 0x0000000100510000 -> 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
   payload_data_1 = 0x00007fff5fbffa10
   payload_data_2 = 0x0000000100002b52 WWDC14`WWDC14.play_with_hierarchy () -> () + 66 at hierarchy.swift:20
   instance_type = 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
   protocol_witness_0 = 0x0000000100007510 protocol witness table for WWDC14.Dog : WWDC14.Creature
}
```

```
(lldb) fr v -d r creature
```

Protocols

Console users beware

```
(lldb) fr v creature
(Creature) creature = {
   payload_data_0 = 0x0000000100510000 -> 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
   payload_data_1 = 0x00007fff5fbffa10
   payload_data_2 = 0x0000000100002b52 WWDC14`WWDC14.play_with_hierarchy () -> () + 66 at hierarchy.swift:20
   instance_type = 0x00000001000081c0 direct type metadata for WWDC14.Dog + 16
   protocol_witness_0 = 0x0000000100007510 protocol witness table for WWDC14.Dog : WWDC14.Creature
}
```

```
(lldb) fr v -d r creature
(WWDC14.Dog) creature = 0x0000000100510000 {
  happy = VeryHappy
}
```

Generics

Generics

Swift has native support for generics

Generics

Swift has native support for generics

Type information passed to functions

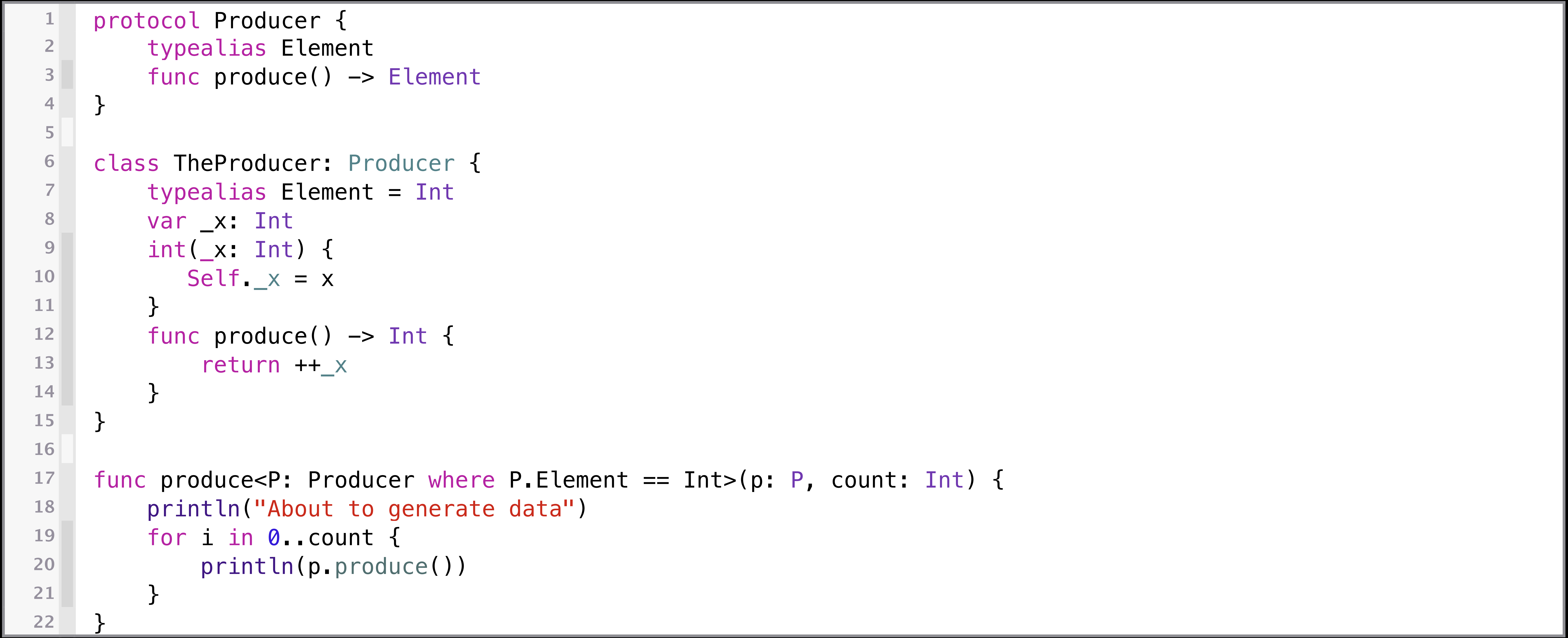
Generics

Swift has native support for generics

Type information passed to functions

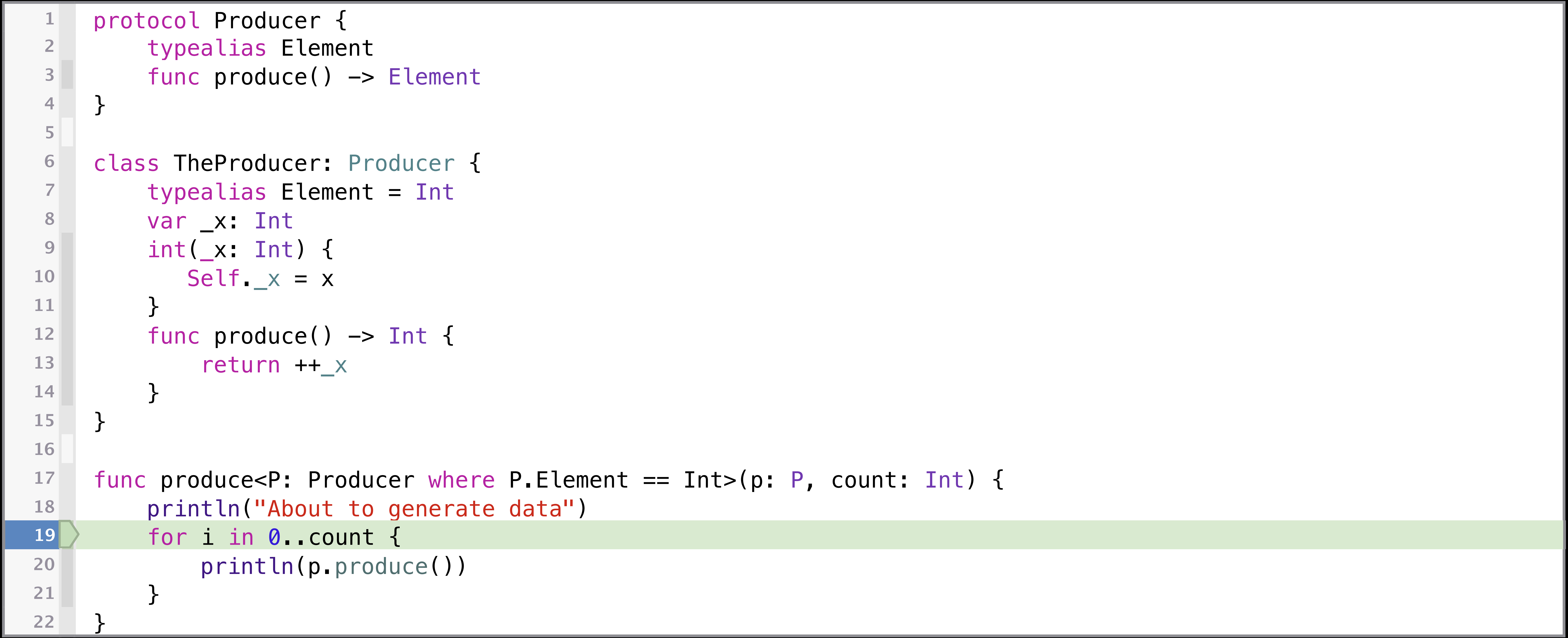
- LLDB uses it to reconstruct code’s meaning

Generics



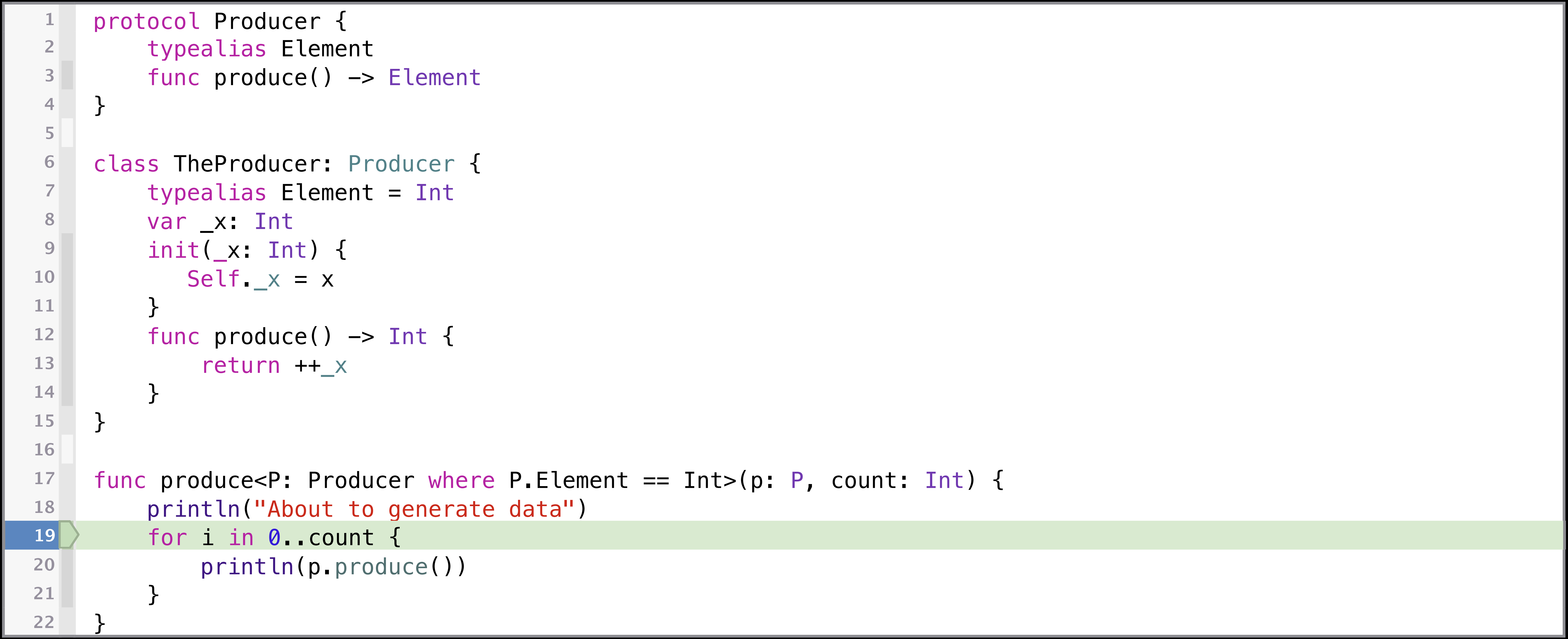
```swift
protocol Producer {
    typealias Element
    func produce() -> Element
}

class TheProducer: Producer {
    typealias Element = Int
    var _x: Int
    int(_x: Int) {
       Self._x = x
    }
    func produce() -> Int {
        return ++_x
    }
}

func produce<P: Producer where P.Element == Int>(p: P, count: Int) {
    println("About to generate data")
    for i in 0..count {
        println(p.produce())
    }
}
```

Generics

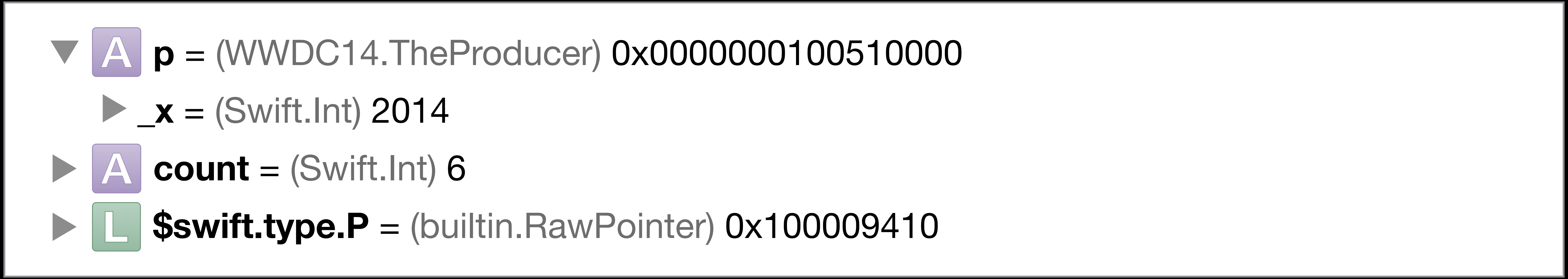


```
protocol Producer {
    typealias Element
    func produce() -> Element
}

class TheProducer: Producer {
    typealias Element = Int
    var _x: Int
    int(_x: Int) {
       Self._x = x
    }
    func produce() -> Int {
        return ++_x
    }
}

func produce<P: Producer where P.Element == Int>(p: P, count: Int) {
    println("About to generate data")
    for i in 0..count {
        println(p.produce())
    }
}
```

Generics

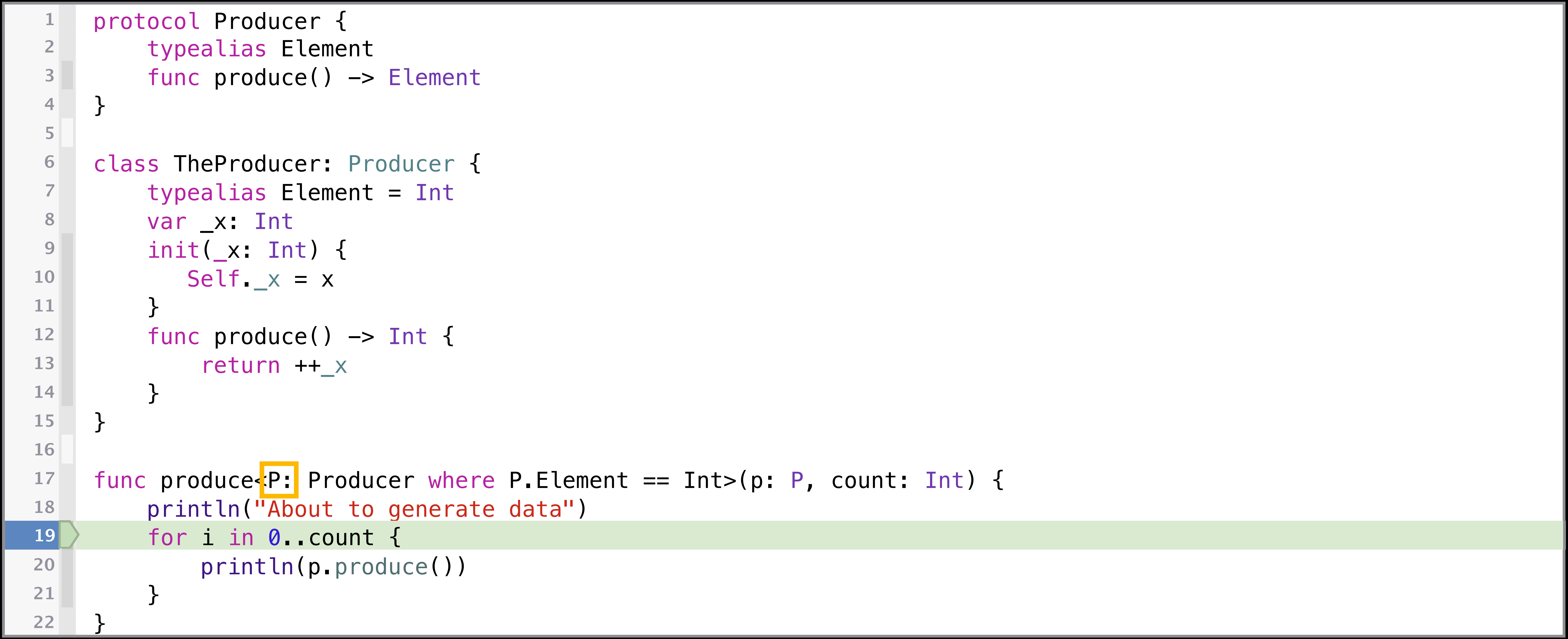


```
protocol Producer {
    typealias Element
    func produce() -> Element
}

class TheProducer: Producer {
    typealias Element = Int
    var _x: Int
    init(_x: Int) {
       Self._x = x
    }
    func produce() -> Int {
        return ++_x
    }
}

func produce<P: Producer where P.Element == Int>(p: P, count: Int) {
    println("About to generate data")
    for i in 0..count {
        println(p.produce())
    }
}
```

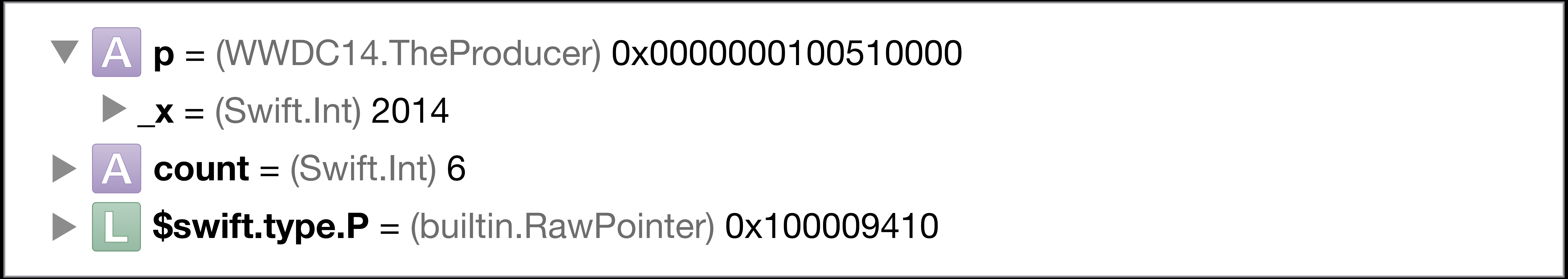


```
▼ A p = (WWDC14.TheProducer) 0x0000000100510000
    ▶ _x = (Swift.Int) 2014
▶ A count = (Swift.Int) 6
▶ L $swift.type.P = (builtin.RawPointer) 0x100009410
```

Generics

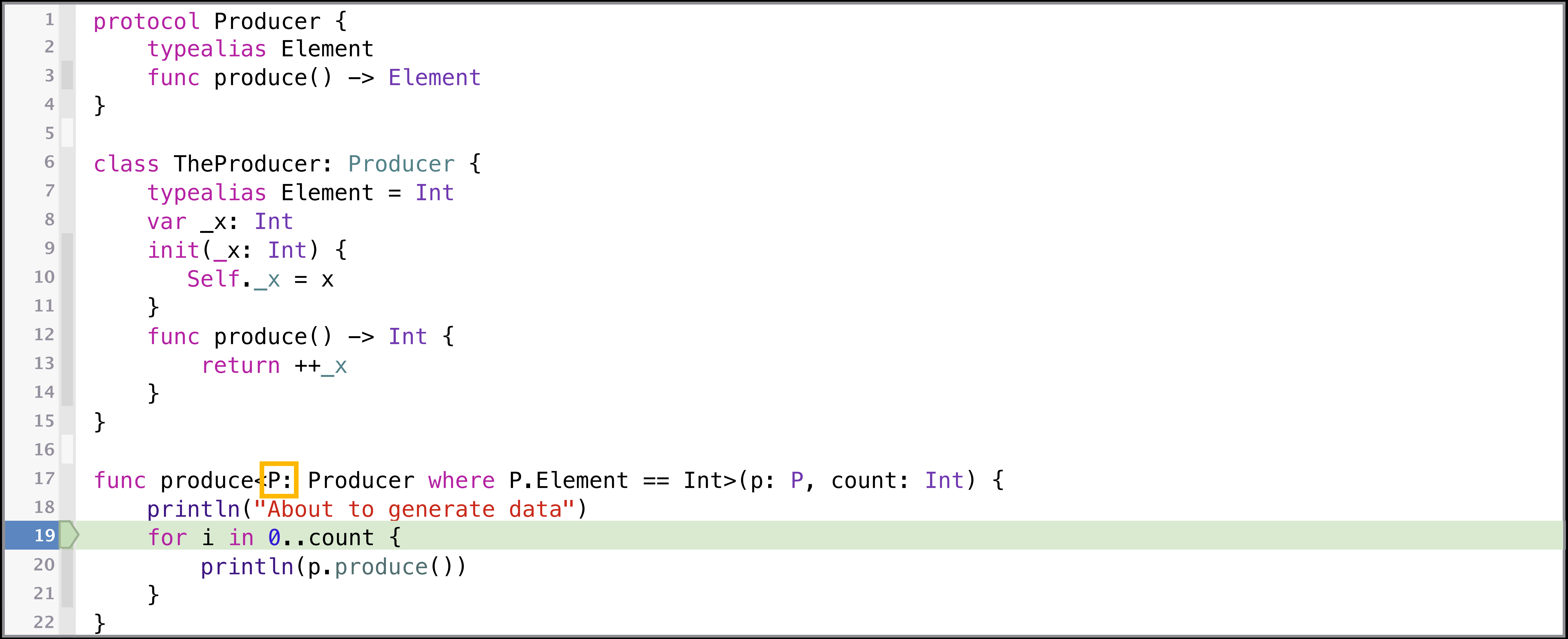


```
protocol Producer {
    typealias Element
    func produce() -> Element
}

class TheProducer: Producer {
    typealias Element = Int
    var _x: Int
    init(_x: Int) {
       Self._x = x
    }
    func produce() -> Int {
        return ++_x
    }
}

func produce<P: Producer where P.Element == Int>(p: P, count: Int) {
    println("About to generate data")
    for i in 0..count {
        println(p.produce())
    }
}
```

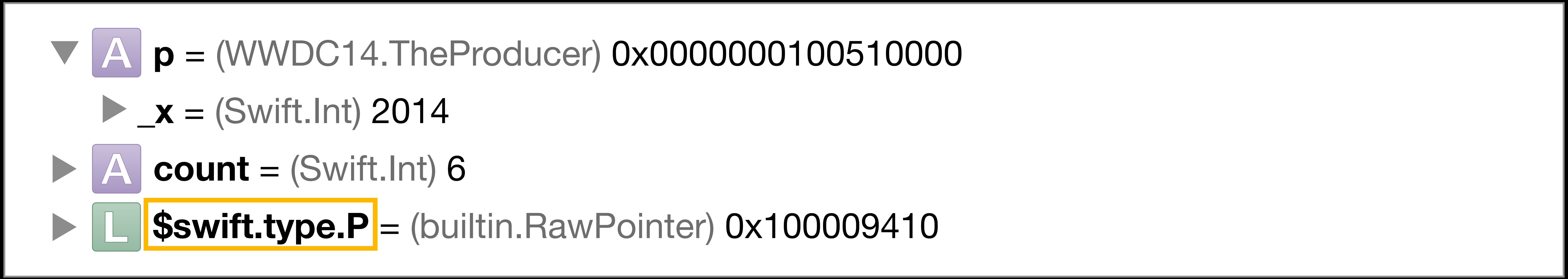


```
▼ A p = (WWDC14.TheProducer) 0x0000000100510000
    ▶ _x = (Swift.Int) 2014
▶ A count = (Swift.Int) 6
▶ L $swift.type.P = (builtin.RawPointer) 0x100009410
```

Generics

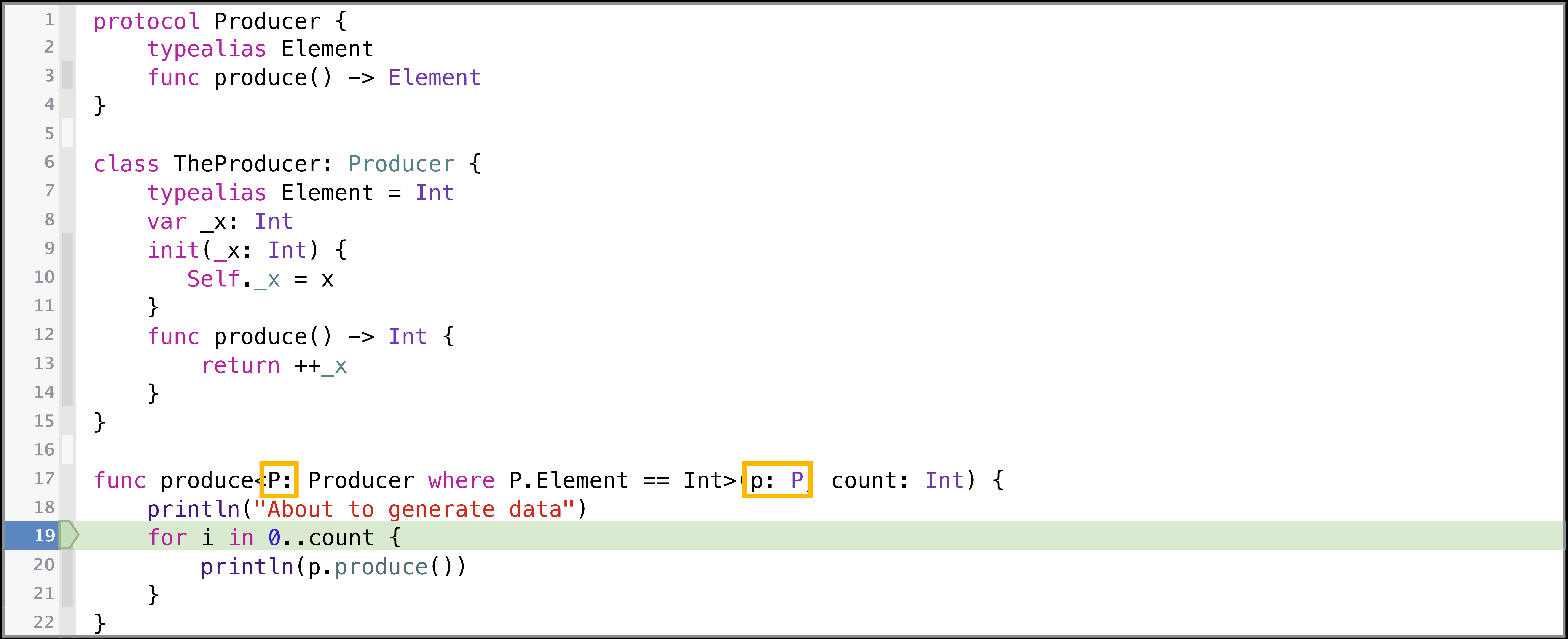


```
protocol Producer {
    typealias Element
    func produce() -> Element
}

class TheProducer: Producer {
    typealias Element = Int
    var _x: Int
    init(_x: Int) {
       Self._x = x
    }
    func produce() -> Int {
        return ++_x
    }
}

func produce<P: Producer where P.Element == Int>(p: P, count: Int) {
    println("About to generate data")
    for i in 0..count {
        println(p.produce())
    }
}
```

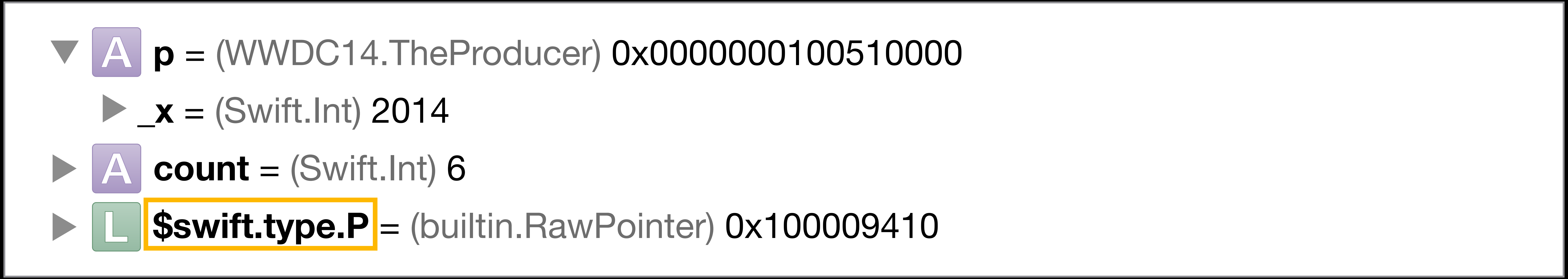


```
▼ A p = (WWDC14.TheProducer) 0x0000000100510000
    ▶ _x = (Swift.Int) 2014
▶ A count = (Swift.Int) 6
▶ L $swift.type.P = (builtin.RawPointer) 0x100009410
```

Generics

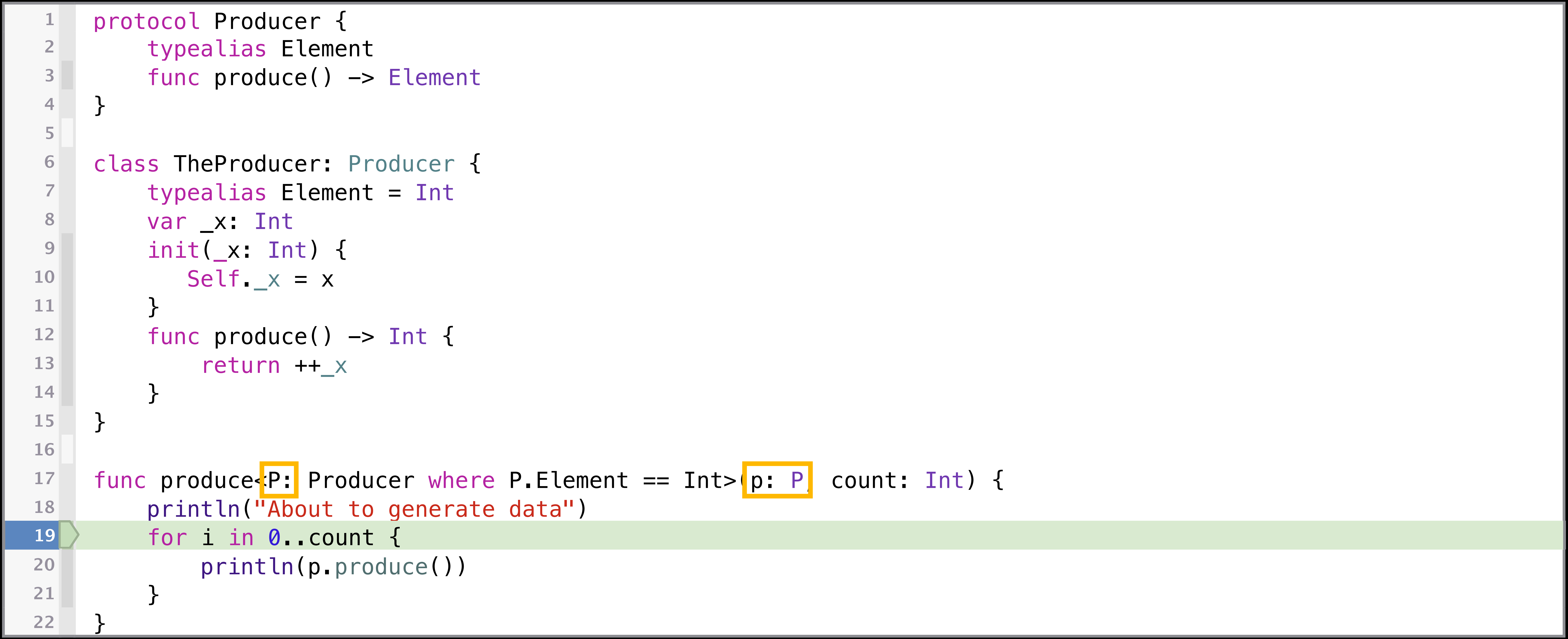


```
protocol Producer {
    typealias Element
    func produce() -> Element
}

class TheProducer: Producer {
    typealias Element = Int
    var _x: Int
    init(_x: Int) {
       Self._x = x
    }
    func produce() -> Int {
        return ++_x
    }
}

func produce<P: Producer where P.Element == Int>(p: P, count: Int) {
    println("About to generate data")
    for i in 0..count {
        println(p.produce())
    }
}
```

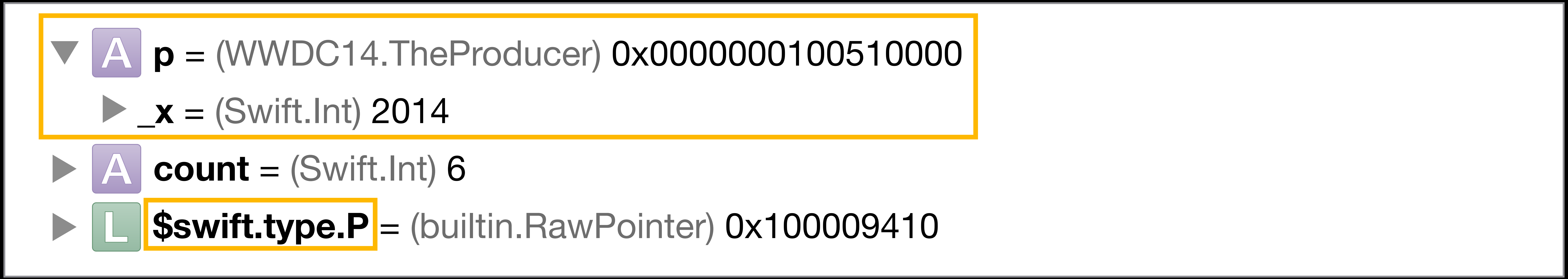


```
▼ A p = (WWDC14.TheProducer) 0x0000000100510000
    ▶ _x = (Swift.Int) 2014
▶ A count = (Swift.Int) 6
▶ L $swift.type.P = (builtin.RawPointer) 0x100009410
```

Generics



```swift
protocol Producer {
    typealias Element
    func produce() -> Element
}

class TheProducer: Producer {
    typealias Element = Int
    var _x: Int
    init(_x: Int) {
       Self._x = x
    }
    func produce() -> Int {
        return ++_x
    }
}

func produce<P: Producer where P.Element == Int>(p: P, count: Int) {
    println("About to generate data")
    for i in 0..count {
        println(p.produce())
    }
}
```



```
▼ A p = (WWDC14.TheProducer) 0x0000000100510000
    ▶ _x = (Swift.Int) 2014
▶ A count = (Swift.Int) 6
▶ L $swift.type.P = (builtin.RawPointer) 0x100009410
```

Debugging Optimized Swift Code

Debugging Optimized Swift Code

Debug builds: Literal translation of your code

Debugging Optimized Swift Code

Debug builds: Literal translation of your code

Optimized builds: Enhanced toward a goal (speed, memory…)

Debugging Optimized Swift Code

Debug builds: Literal translation of your code

Optimized builds: Enhanced toward a goal (speed, memory…)

The first rule of debugging optimized code

- Don’t!

Debugging Optimized Swift Code

Debug builds: Literal translation of your code

Optimized builds: Enhanced toward a goal (speed, memory…)

The first rule of debugging optimized code

- Don’t!

Generics may be optimized for specific types

Protocols may be devirtualized

Objective-C Interop

Objective-C Interop

Two main cases:

- ObjC frameworks in Swift apps
- Apps with ObjC and Swift source code

Objective-C Interop

Two main cases:

- ObjC frameworks in Swift apps
- Apps with ObjC and Swift source code

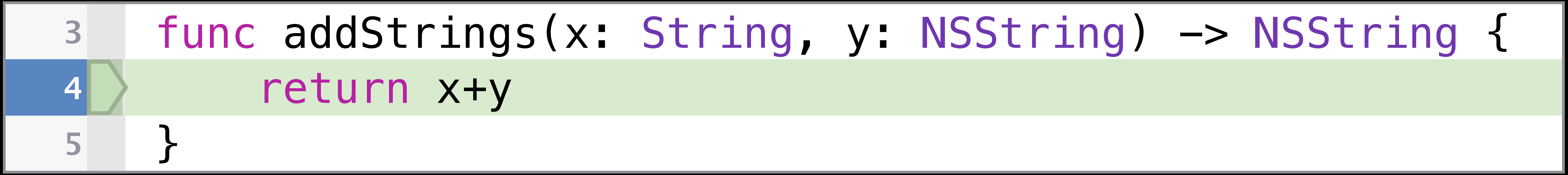
What to expect?

- Variables view
- Expression evaluation
- po

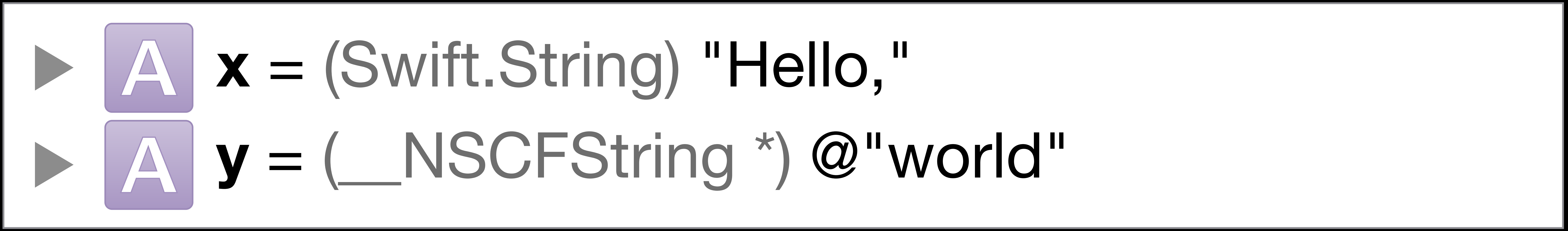
Objective-C Interop: Variables View

Most native experience

- Data shown in the type's language of origin
- Formatters apply in all cases



```
func addStrings(x: String, y: NSString) -> NSString {
    return x+y
}
```



```
x = (Swift.String) "Hello,"
y = (__NSCFString *) @"world"
```

Objective-C Interop: Expressions

Objective-C Interop: Expressions

Expressions see two separate worlds

Objective-C Interop: Expressions

Expressions see two separate worlds

- Objects in Swift frames only for Swift expressions

Objective-C Interop: Expressions

Expressions see two separate worlds

- Objects in Swift frames only for Swift expressions
- Two namespaces for your results

Objective-C Interop: Expressions

Expressions see two separate worlds

- Objects in Swift frames only for Swift expressions
- Two namespaces for your results
    - $0, $1, … for Objective-C
    - $R0, $R1, … for Swift

Objective-C Interop: Expressions

Objective-C Interop: Expressions

```
(lldb) f
frame #0: 0x0000000100005de0 WWDC14`-[CocoaClass description](self=0x000000010040e940,
_cmd=0x00007fff8c7eaf49) + 16 at CocoaClass.m:19
```

Objective-C Interop: Expressions

```
(lldb) f
frame #0: 0x0000000100005de0 WWDC14`-[CocoaClass description](self=0x000000010040e940,
_cmd=0x00007fff8c7eaf49) + 16 at CocoaClass.m:19
```

Objective-C Interop: Expressions

```
(lldb) f
frame #0: 0x0000000100005de0 WWDC14`-[CocoaClass description](self=0x000000010040e940,
_cmd=0x00007fff8c7eaf49) + 16 at CocoaClass.m:19
```

```
(lldb) p self
CocoaClass *) $0 = 0x000000010040e940
(lldb) |
```

Objective-C Interop: Expressions

```
(lldb) f
frame #0: 0x0000000100005de0 WWDC14`-[CocoaClass description](self=0x000000010040e940,
_cmd=0x00007fff8c7eaf49) + 16 at CocoaClass.m:19
```

```
(lldb) p self
CocoaClass *) $0 = 0x000000010040e940
(lldb) |
```

```
(lldb) f
frame #0: 0x0000000100004a1f WWDC14`WWDC14.play_with_usecocoa () -> () + 95 at
usecocoa.swift:6
```

Objective-C Interop: Expressions

```
(lldb) f
frame #0: 0x0000000100005de0 WWDC14`-[CocoaClass description](self=0x000000010040e940,
_cmd=0x00007fff8c7eaf49) + 16 at CocoaClass.m:19
```

```
(lldb) p self
CocoaClass *) $0 = 0x000000010040e940
(lldb) |
```

```
(lldb) f
frame #0: 0x0000000100004a1f WWDC14`WWDC14.play_with_usecocoa () -> () + 95 at
usecocoa.swift:6
```

Objective-C Interop: Expressions

```
(lldb) f
frame #0: 0x0000000100005de0 WWDC14`-[CocoaClass description](self=0x000000010040e940,
_cmd=0x00007fff8c7eaf49) + 16 at CocoaClass.m:19
```

```
(lldb) p self
CocoaClass *) $0 = 0x000000010040e940
(lldb) |
```

```
(lldb) f
frame #0: 0x0000000100004a1f WWDC14`WWDC14.play_with_usecocoa () -> () + 95 at
usecocoa.swift:6
```

```
(lldb) p (Class) [$0 class]
error: <REPL>:1:9: error: anonymous closure argument not contained in a closure
(Class)[$0 class]
        ^
<REPL>:1:12: error: expected ',' separator
(Class) [$0 class]
            ^
```

Objective-C Interop: Expressions

Objective-C Interop: Expressions

Language can be changed

Objective-C Interop: Expressions

Language can be changed

```
(lldb) expr -l objc++ -- (Class)[$0 class]
(Class) $2 = CocoaClass
```

Objective-C Interop: Expressions

Language can be changed

```
(lldb) expr -l objc++ -- (Class)[$0 class]
(Class) $2 = CocoaClass
```

Objective-C Interop: Expressions

Language can be changed

- But locals will not be available

```
(lldb) expr -l objc++ -- (Class)[$0 class]
(Class) $2 = CocoaClass
```

Objective-C Interop: po

po honors most native experience:

- Swift objects display using formatters
- Objective-C objects use `-description`

Objective-C Interop

```
class MyObject: NSObject {
    var myInt = 1
    override var description: String! {
       return "Hello Swift subclass. myInt = \(myInt)"
    }
}
```

Objective-C Interop

```
class MyObject: NSObject {
    var myInt = 1
    override var description: String! {
       return "Hello Swift subclass. myInt = \(myInt)"
    }
}
```

po uses formatters—ignores description

Objective-C Interop

```
class MyObject: NSObject {
    var myInt = 1
    override var description: String! {
       return "Hello Swift subclass. myInt = \(myInt)"
    }
}
```

po uses formatters—ignores description

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

Objective-C Interop

```
class MyObject: NSObject {
    var myInt = 1
    override var description: String! {
        return "Hello Swift subclass. myInt = \(myInt)"
    }
}
```

po uses formatters—ignores description

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

What if I want to use my description property?

Objective-C Interop

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

```
(lldb) expr
```

Objective-C Interop

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

```
(lldb) expr –l objc++
```

Objective-C Interop

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

```
(lldb) expr -l objc++ -O --
```

Objective-C Interop

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

```
(lldb) expr -l objc++ -O --    0x0000000100700ea0
```

Objective-C Interop

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

```
(lldb) expr -l objc++ -O -- (id)0x0000000100700ea0
```

Objective-C Interop

```
(lldb) po object
0x0000000100700ea0
 (ObjectiveC.NSObject = {}, myInt = 1)
```

```
(lldb) expr -l objc++ -O -- (id)0x0000000100700ea0
       Hello Swift subclass. myInt = 1
```

Stepping

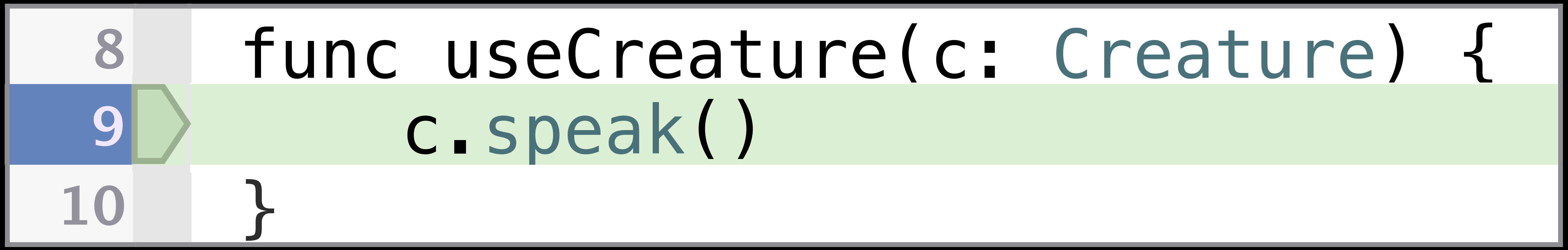
Stepping around Swift code

- Protocols
- Closures

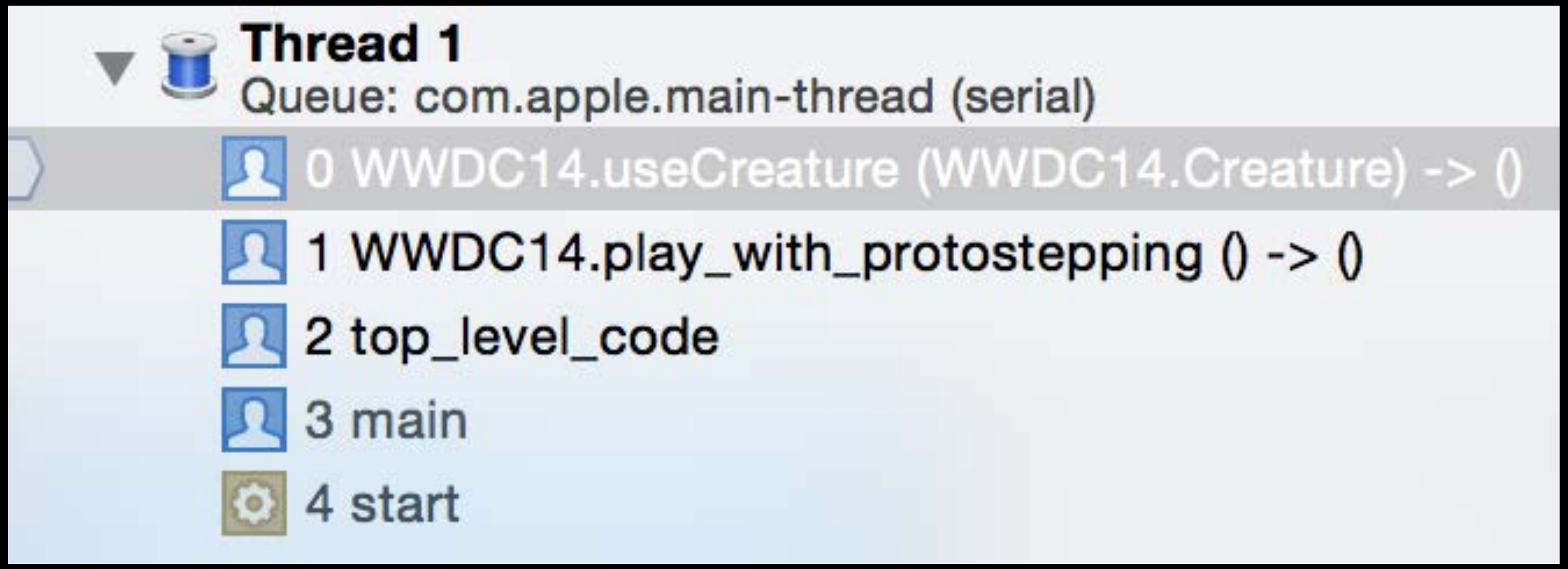
Stepping: Protocols

```swift
func useCreature(c: Creature) {
    c.speak()
}
```

Stepping: Protocols

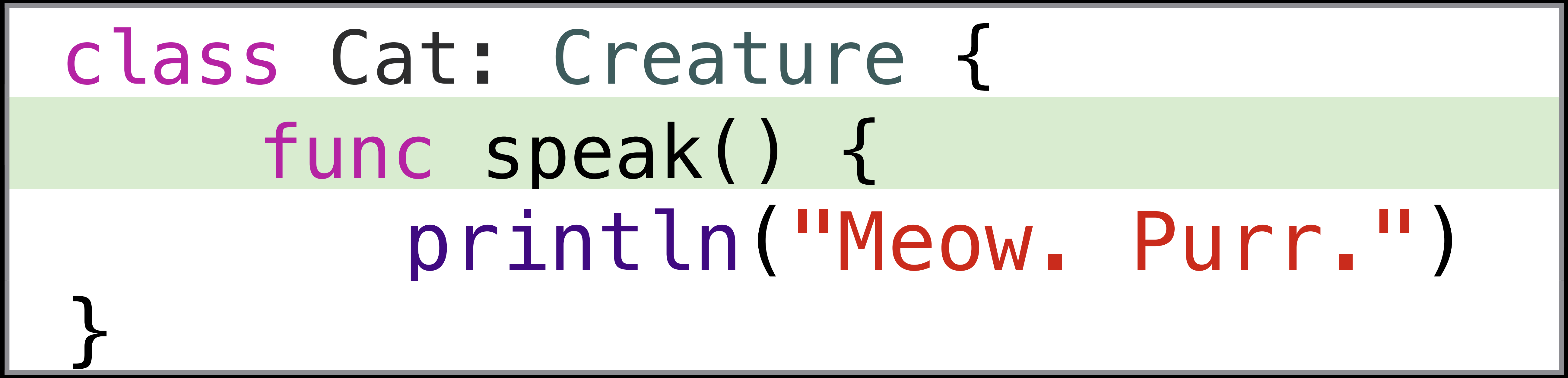


```
func useCreature(c: Creature) {
    c.speak()
}
```

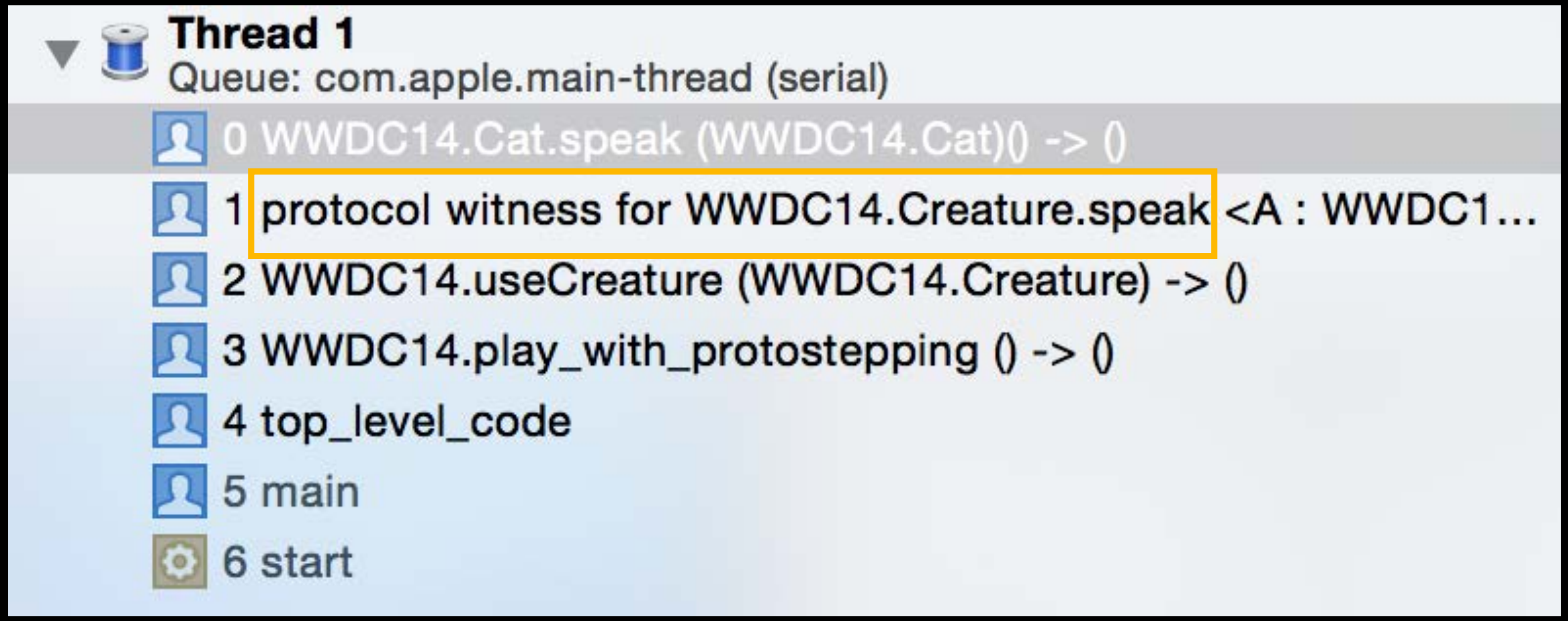


```
Thread 1
Queue: com.apple.main-thread (serial)
0 WWDC14.useCreature (WWDC14.Creature) -> ()
1 WWDC14.play_with_protostepping () -> ()
2 top_level_code
3 main
4 start
```

Stepping: Protocols

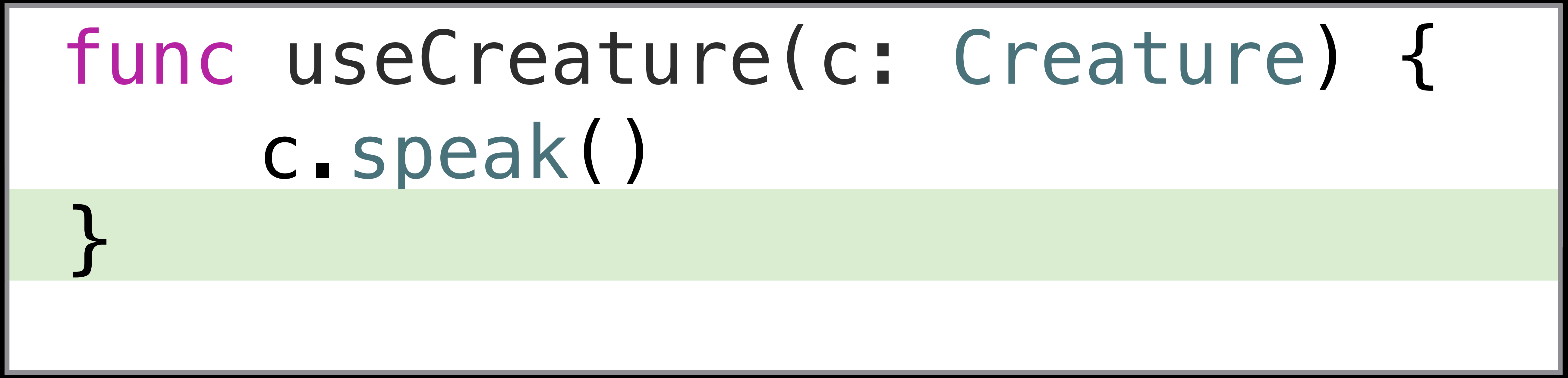


```
class Cat: Creature {
    func speak() {
        println("Meow. Purr.")
}
```

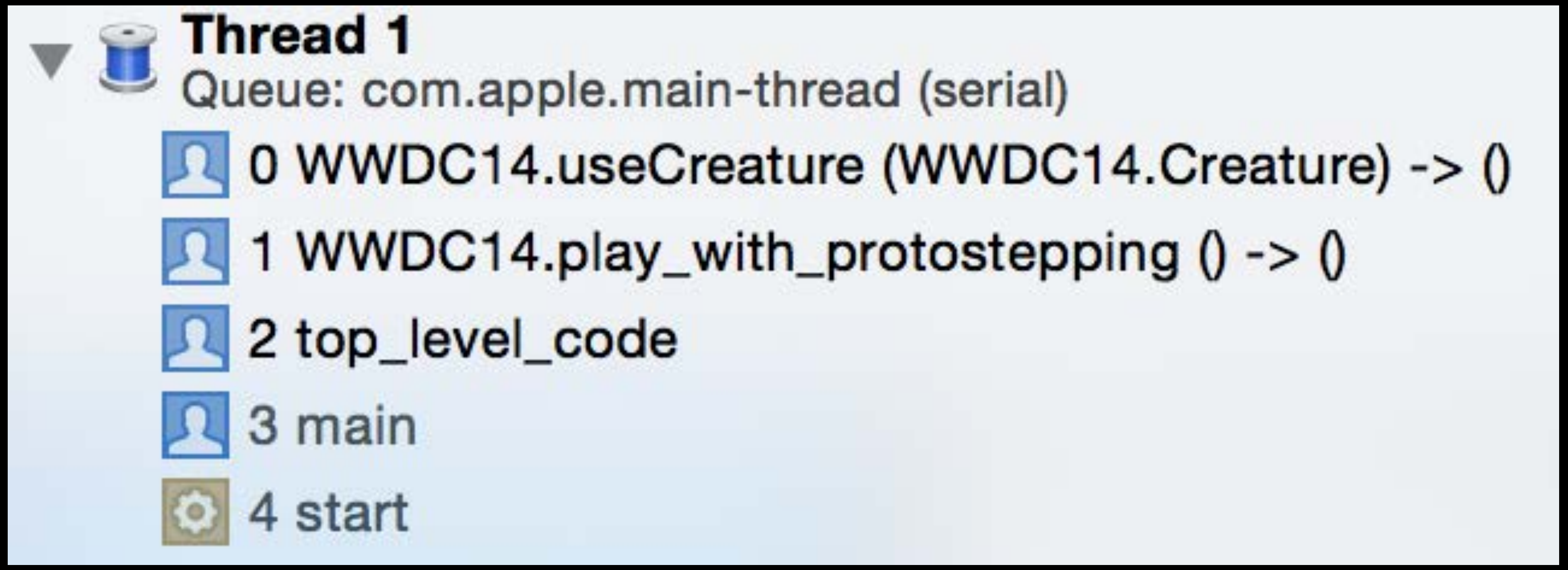


```
Thread 1
Queue: com.apple.main-thread (serial)
  0 WWDC14.Cat.speak (WWDC14.Cat)() -> ()
  1 protocol witness for WWDC14.Creature.speak <A : WWDC1...
  2 WWDC14.useCreature (WWDC14.Creature) -> ()
  3 WWDC14.play_with_protostepping () -> ()
  4 top_level_code
  5 main
  6 start
```

Stepping: Protocols

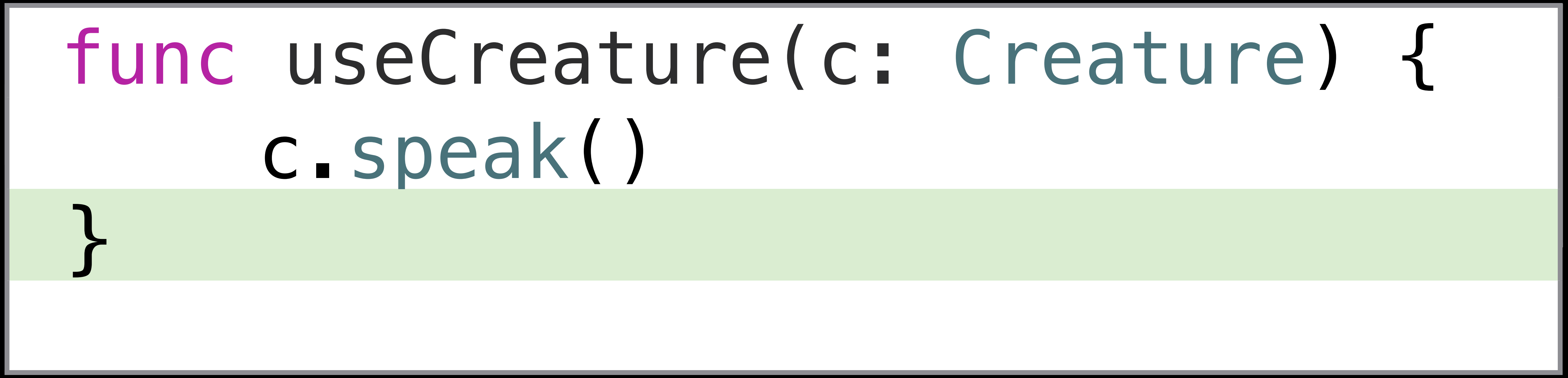


```
func useCreature(c: Creature) {
    c.speak()
}
```

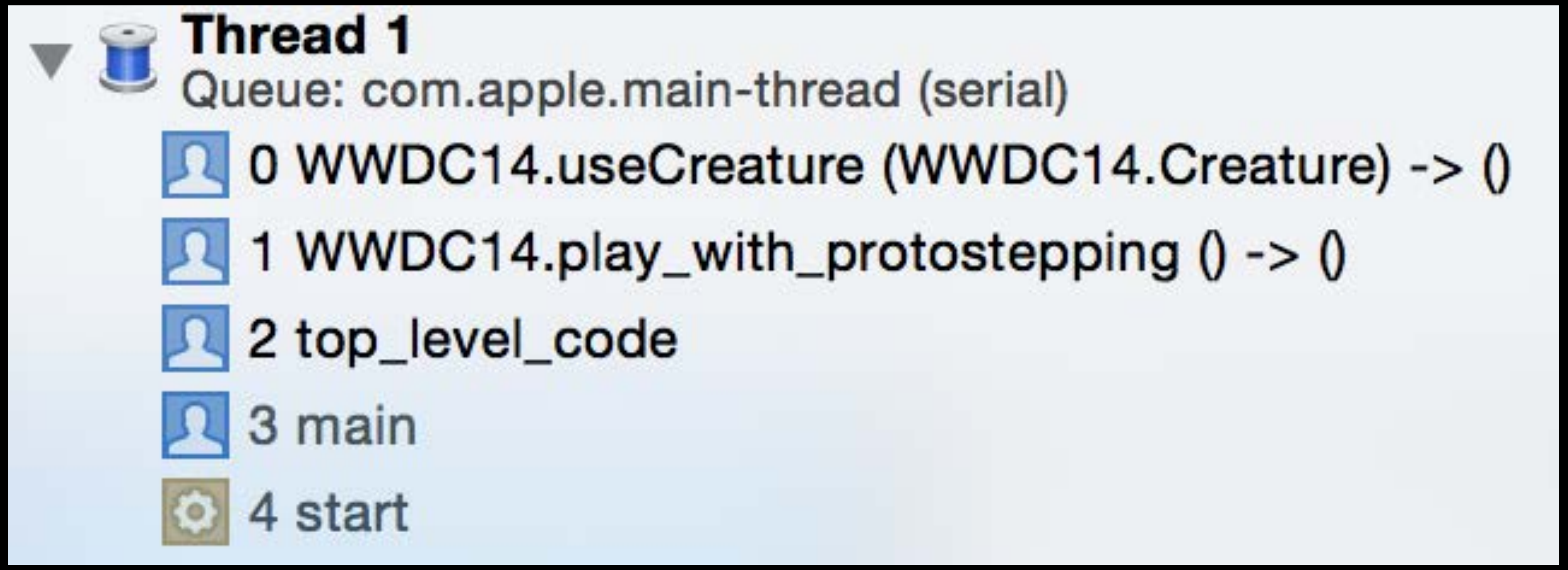


```
Thread 1
Queue: com.apple.main-thread (serial)
    0 WWDC14.useCreature (WWDC14.Creature) -> ()
    1 WWDC14.play_with_protostepping () -> ()
    2 top_level_code
    3 main
    4 start
```

Stepping: Protocols



```
func useCreature(c: Creature) {
    c.speak()
}
```

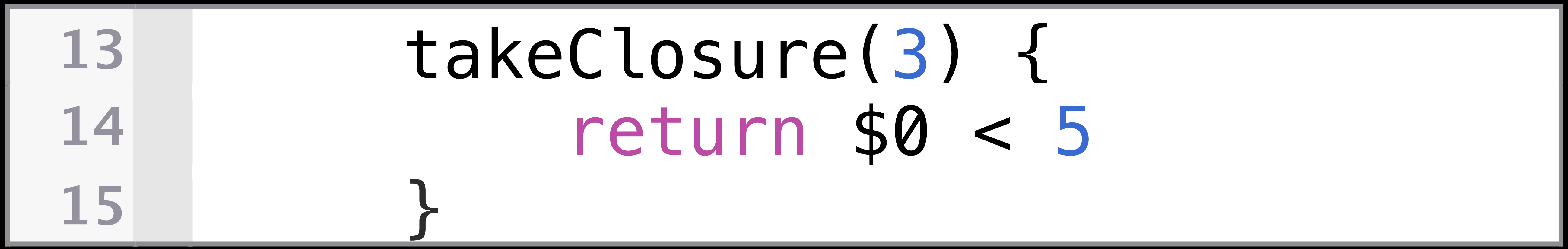


```
Thread 1
Queue: com.apple.main-thread (serial)
  0 WWDC14.useCreature (WWDC14.Creature) -> ()
  1 WWDC14.play_with_protostepping () -> ()
  2 top_level_code
  3 main
  4 start
```

Stepping inside protocol implementations involves a layer of dynamic dispatch ("protocol witness")

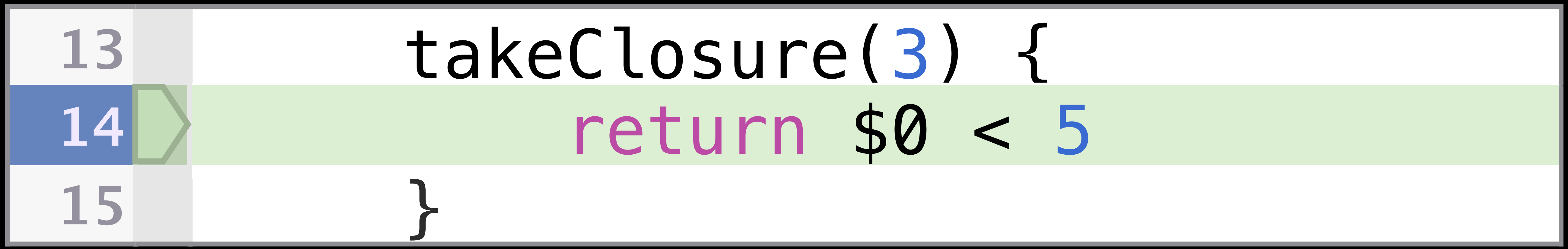
Stepping out of the implementation steps out of the witness

Stepping: Breakpoint In a Closure



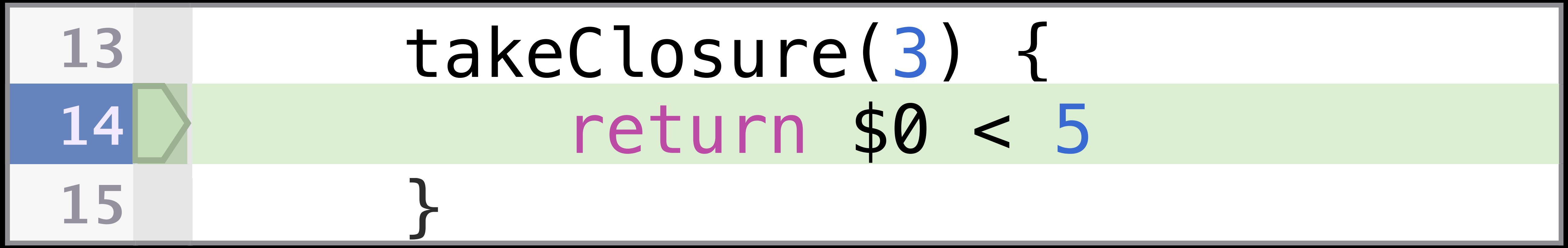
```swift
    takeClosure(3) {
        return $0 < 5
    }
```

Stepping: Breakpoint In a Closure

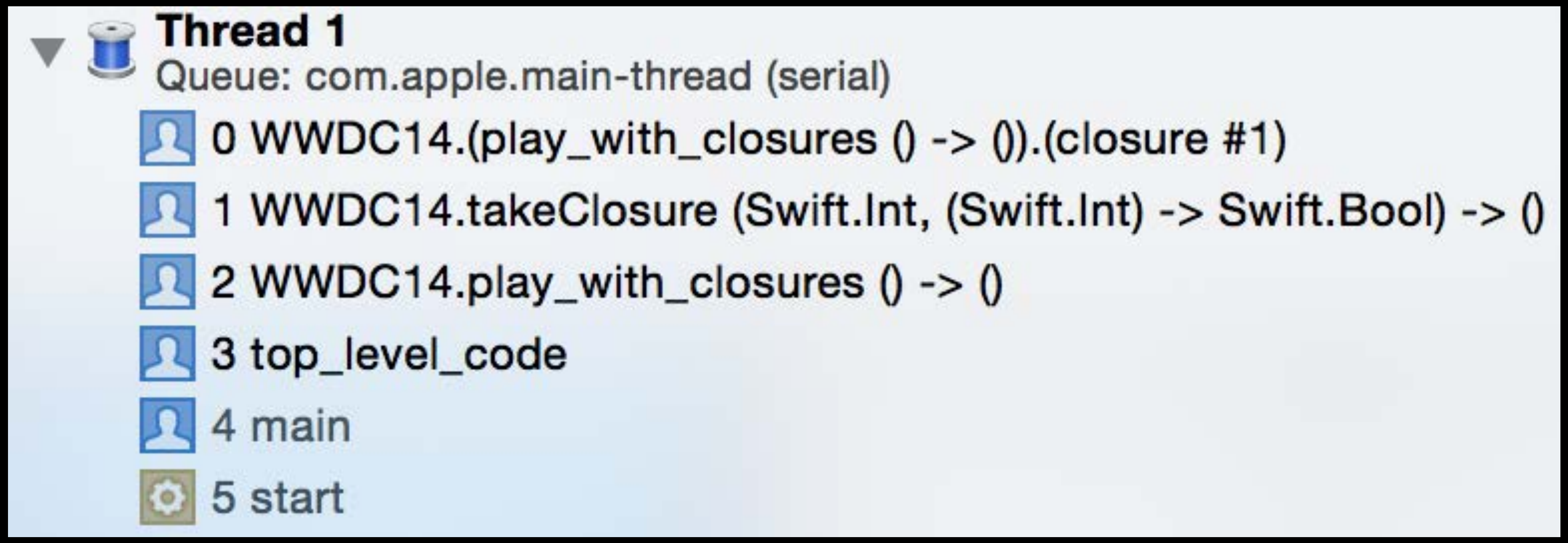


```
        takeClosure(3) {
            return $0 < 5
        }
```

Stepping: Breakpoint In a Closure

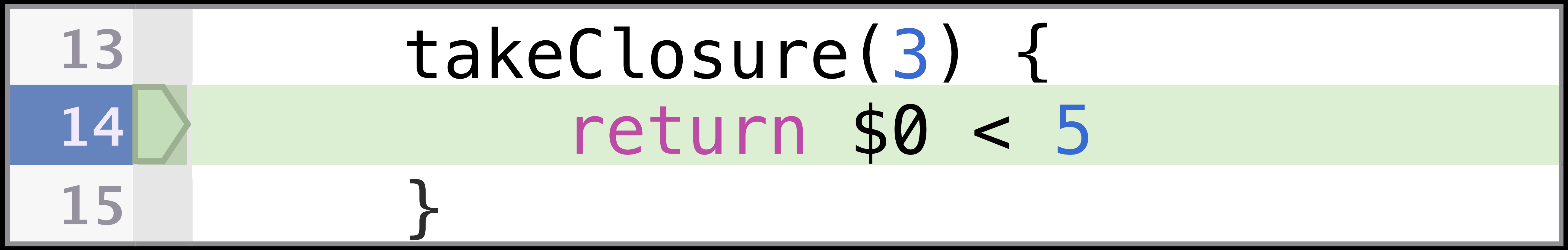


```
        takeClosure(3) {
            return $0 < 5
        }
```

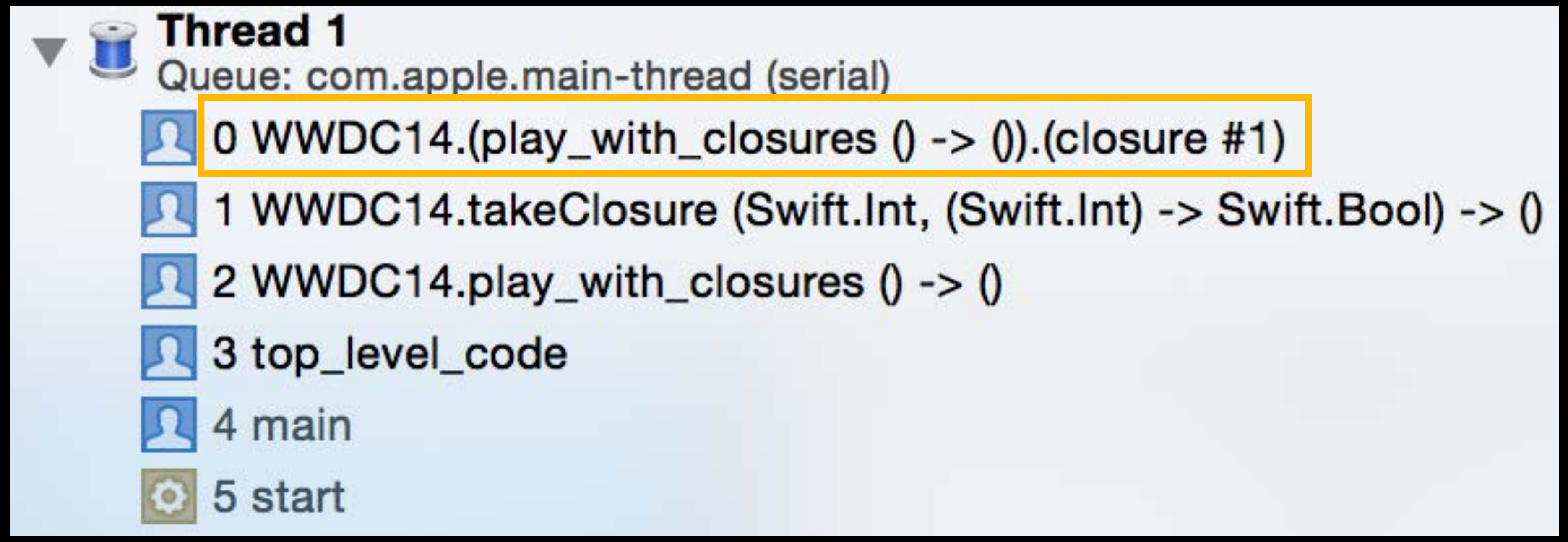


```
Thread 1
Queue: com.apple.main-thread (serial)
0 WWDC14.(play_with_closures () -> ()).(closure #1)
1 WWDC14.takeClosure (Swift.Int, (Swift.Int) -> Swift.Bool) -> ()
2 WWDC14.play_with_closures () -> ()
3 top_level_code
4 main
5 start
```

Stepping: Breakpoint In a Closure

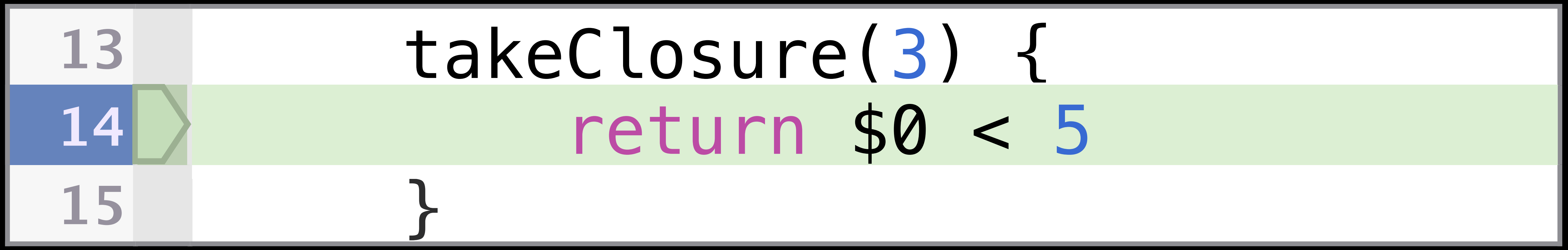


```
        takeClosure(3) {
            return $0 < 5
        }
```

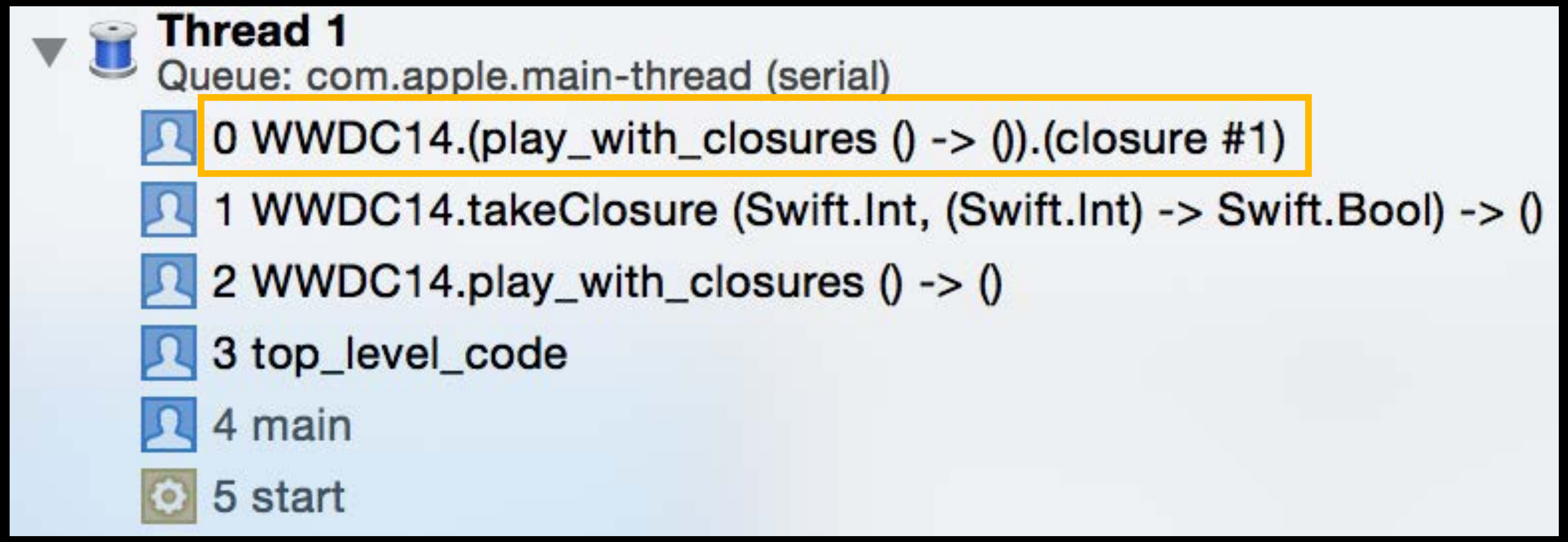


```
Thread 1
Queue: com.apple.main-thread (serial)
0 WWDC14.(play_with_closures () -> ()).(closure #1)
1 WWDC14.takeClosure (Swift.Int, (Swift.Int) -> Swift.Bool) -> ()
2 WWDC14.play_with_closures () -> ()
3 top_level_code
4 main
5 start
```

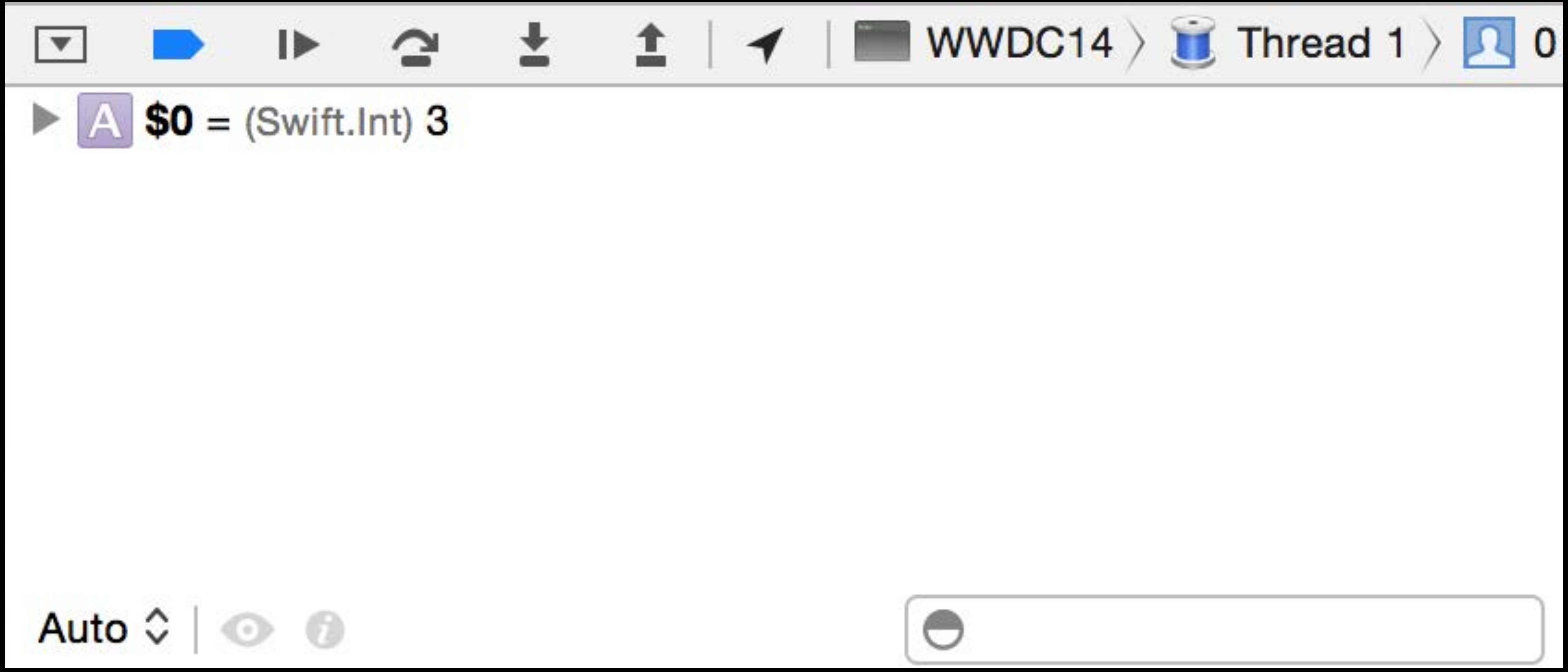
Stepping: Breakpoint In a Closure



```
takeClosure(3) {
    return $0 < 5
}
```



```
Thread 1
Queue: com.apple.main-thread (serial)
  0 WWDC14.(play_with_closures () -> ()).(closure #1)
  1 WWDC14.takeClosure (Swift.Int, (Swift.Int) -> Swift.Bool) -> ()
  2 WWDC14.play_with_closures () -> ()
  3 top_level_code
  4 main
  5 start
```



```
$0 = (Swift.Int) 3
```

Data Formatters for Swift Objects

Data Formatters for Swift Objects

LLDB data formatters improve data display

- Hide implementation details
- Focus on what matters

Data Formatters for Swift Objects

LLDB data formatters improve data display

- Hide implementation details
- Focus on what matters

LLDB formats Swift library types automatically

Data Formatters for Swift Objects

LLDB data formatters improve data display

- Hide implementation details
- Focus on what matters

LLDB formats Swift library types automatically

You can roll your own

- Just like for C++/Objective-C

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

Data Formatters for Swift Objects

```swift
struct Address {
     var name: String
     var city: String
     var zip: Int
     var state: State
}
```

```
(lldb)po enrico
{
Name = "Enrico Granata"
City = "Mountain View"
Zip = 94043
State = California
}
```

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

```
(lldb)po enrico
{
Name = "Enrico Granata"
City = "Mountain View"
Zip = 94043
State = California
}
```

```
(lldb) type summary add -s "${var.name} \n ${var.city} \n ${var.zip}, ${var.state}" WWDC14.Address
```

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

```
(lldb)po enrico
{
Name = "Enrico Granata"
City = "Mountain View"
Zip = 94043
State = California
}
```

```
(lldb) type summary add -s "${var.name} \n ${var.city} \n ${var.zip}, ${var.state}" WWDC14.Address
```

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

```
(lldb)po enrico
{
Name = "Enrico Granata"
City = "Mountain View"
Zip = 94043
State = California
}
```

```
(lldb) type summary add -s "${var.name} \n ${var.city} \n ${var.zip}, ${var.state}" WWDC14.Address
```

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

```
(lldb)po enrico
{
Name = "Enrico Granata"
City = "Mountain View"
Zip = 94043
State = California
}
```

```
(lldb) type summary add -s "${var.name} \n ${var.city} \n ${var.zip}, ${var.state}" WWDC14.Address
```

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

```
(lldb)po enrico
{
Name = "Enrico Granata"
City = "Mountain View"
Zip = 94043
State = California
}
```

```
(lldb) type summary add -s "${var.name} \n ${var.city} \n ${var.zip}, ${var.state}" WWDC14.Address
```

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

```
(lldb)po enrico
{
Name = "Enrico Granata"
City = "Mountain View"
Zip = 94043
State = California
}
```

```
(lldb) type summary add -s "${var.name} \n ${var.city} \n ${var.zip}, ${var.state}" WWDC14.Address
```

Data Formatters for Swift Objects

```swift
struct Address {
    var name: String
    var city: String
    var zip: Int
    var state: State
}
```

```
(lldb)po enrico
"Enrico Granata"
 "Mountain View"
 94043, California
```

Data Formatters for Swift Objects

Data Formatters for Swift Objects

Caveats

Data Formatters for Swift Objects

Caveats

- Type name must be fully qualified (include module)

Data Formatters for Swift Objects

Caveats

- Type name must be fully qualified (include module)
- In Python, use SBValue.GetSummary()

Data Formatters for Swift Objects

Caveats

- Type name must be fully qualified (include module)
- In Python, use SBValue.GetSummary()
  - Except for enums!

Uniqueness

MyApp

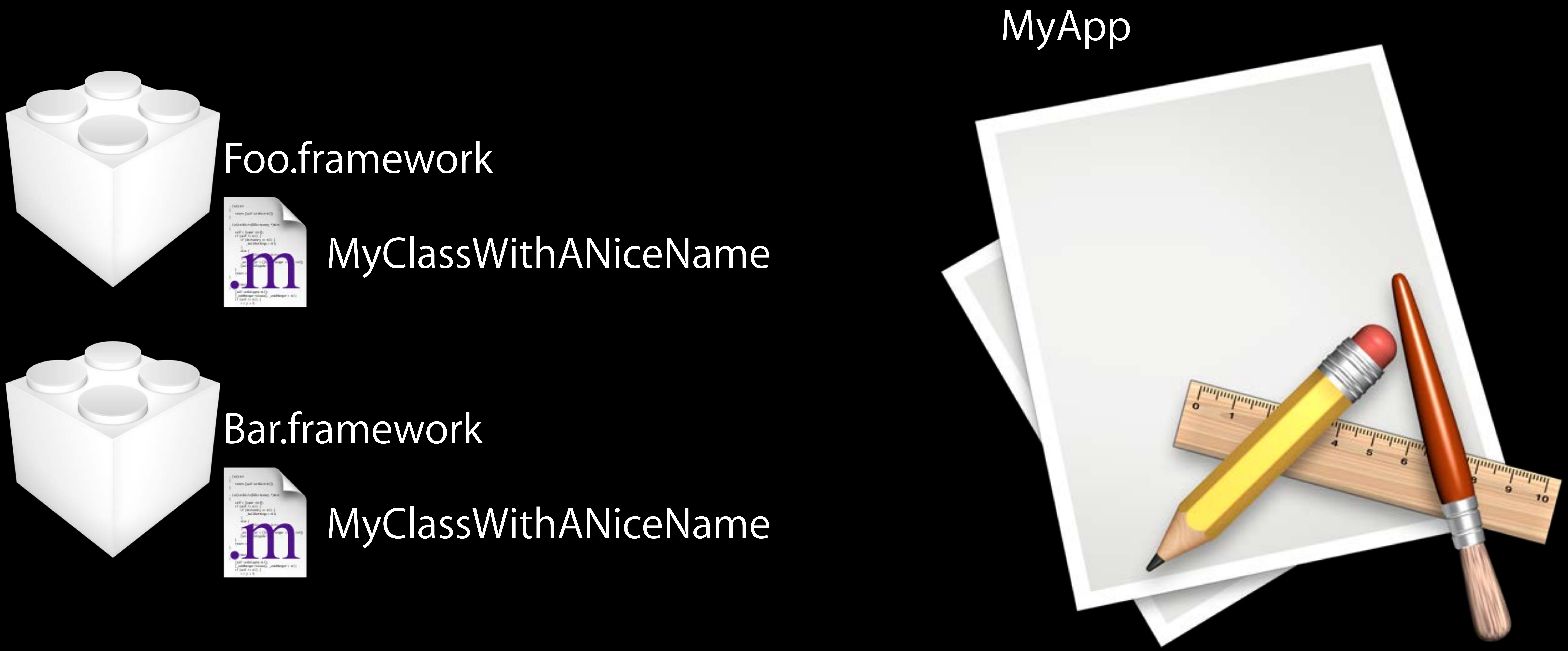
Uniqueness

Foo.framework

MyClassWithANiceName

MyApp

Uniqueness


MyApp
Foo.framework
MyClassWithANiceName
Bar.framework
MyClassWithANiceName

Uniqueness

Uniqueness

Swift provides uniqueness

- Of function overloads
- Of classes in different frameworks

Uniqueness

Swift provides uniqueness

- Of function overloads
- Of classes in different frameworks

Mangled names are the way

Uniqueness

Module1.swift

```
class MyClass {…}
```

Module2.swift

```
class MyClass {…}
```

Uniqueness

Module1.swift

```
class MyClass {…}
```

Module2.swift

```
class MyClass {…}
```

Swift Compiler

Uniqueness

Module1.swift

```
class MyClass {…}
```

Module2.swift

```
class MyClass {…}
```

Swift Compiler

Uniqueness

Module1.swift

```
class MyClass {…}
```

Module2.swift

```
class MyClass {…}
```

Swift Compiler

Module1

_TtC7Module17MyClass

Uniqueness

Module1.swift

```
class MyClass {…}
```

Module2.swift

```
class MyClass {…}
```

Swift Compiler

Module1
_TtC7Module17MyClass

Module2
_TtC7Module27MyClass

Uniqueness

Uniqueness

What if you encounter a mangled name?

Uniqueness

What if you encounter a mangled name?

Enter swift-demangle!

Uniqueness

What if you encounter a mangled name?

Enter swift-demangle!

```
$ xcrun swift-demangle _TF5MyApp6myFuncFTSiSi_TSS_
```

Uniqueness

What if you encounter a mangled name?

Enter swift-demangle!

```
$ xcrun swift-demangle _TF5MyApp6myFuncFTSiSi_TSS_

_TF5MyApp6myFuncFTSiSi_TSS_ ---> MyApp.myFunc (Swift.Int, Swift.Int) ->
(Swift.String)
```

Modules

MyApp.swift

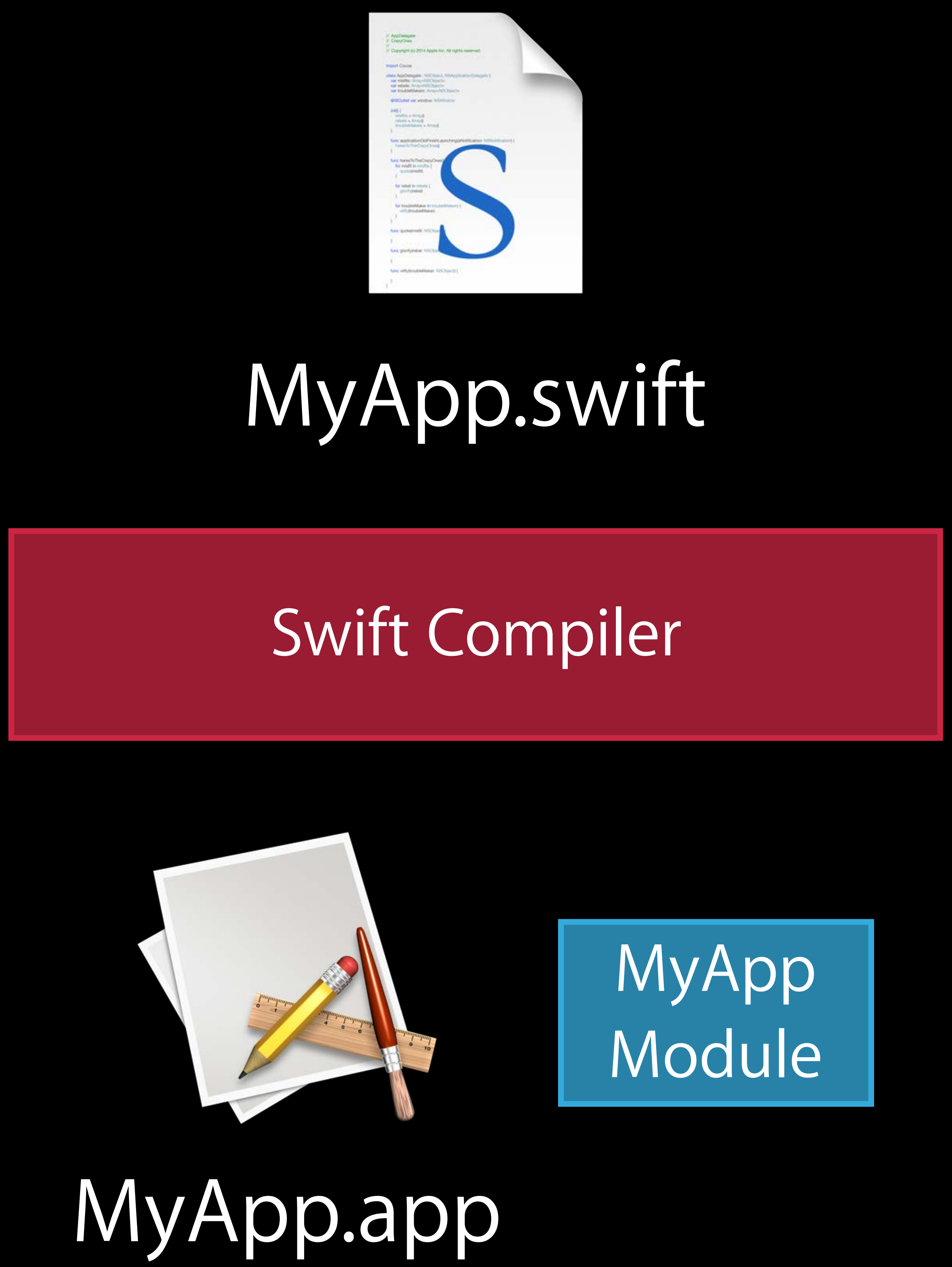
Modules

MyApp.swift
Swift Compiler

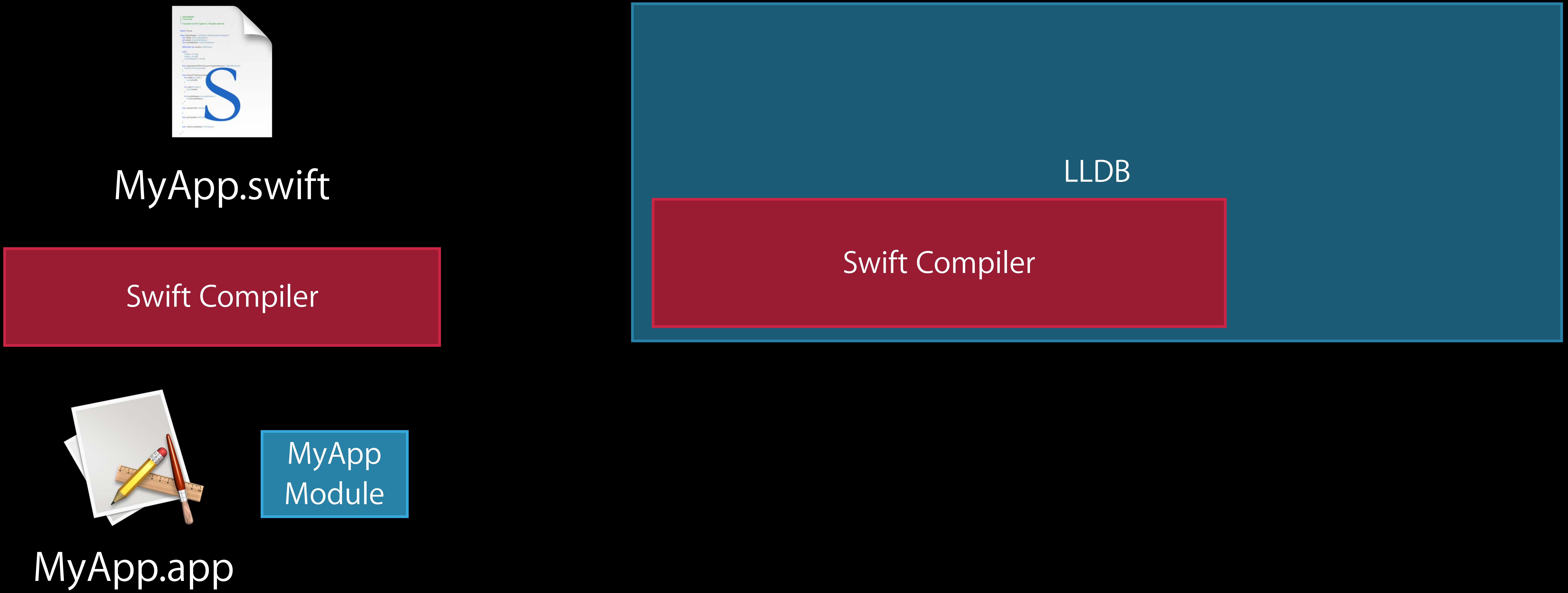
Modules

Swift Compiler
MyApp.swift

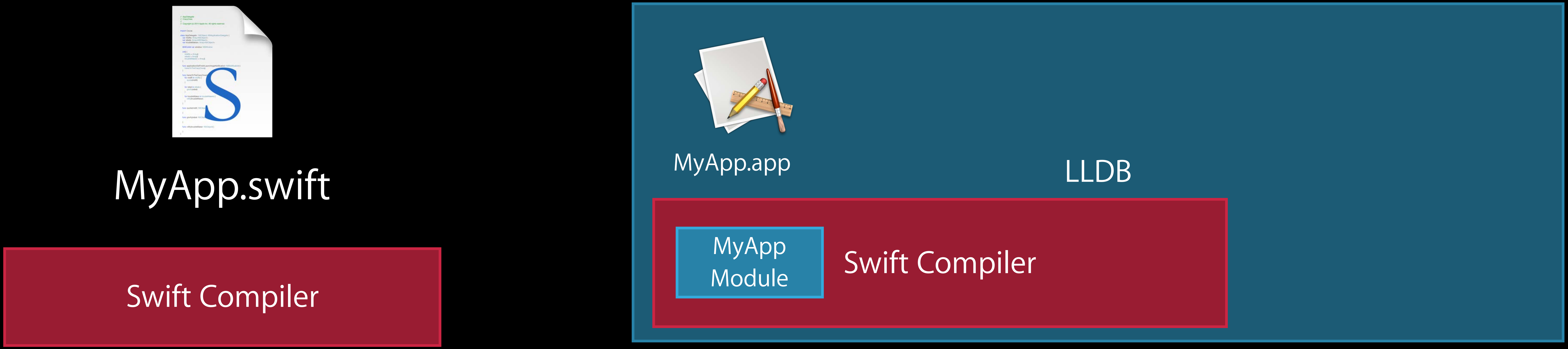
Modules



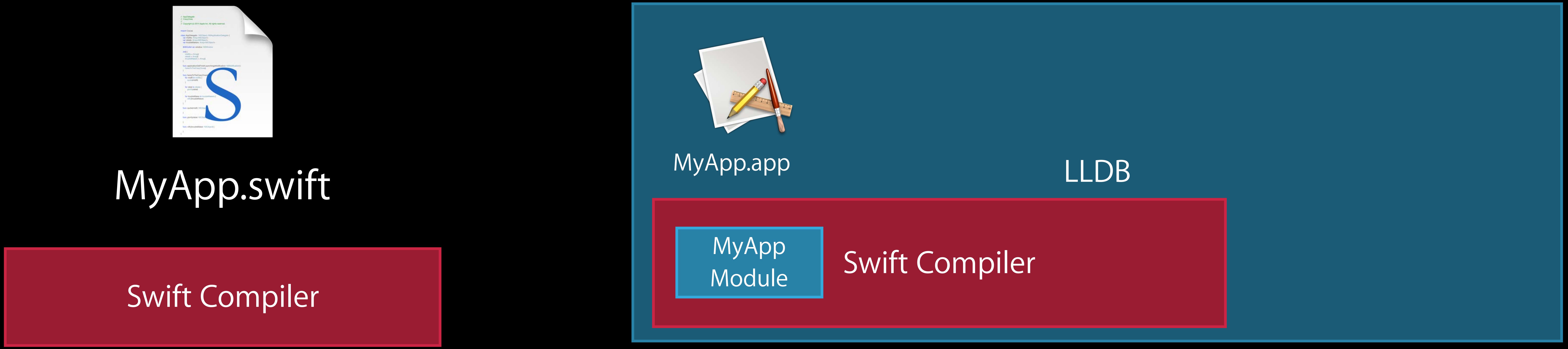
Modules


MyApp.swift
Swift Compiler
MyApp.app
MyApp
Module
LLDB
Swift Compiler

Modules


MyApp.swift
Swift Compiler
MyApp.app
LLDB
MyApp Module
Swift Compiler

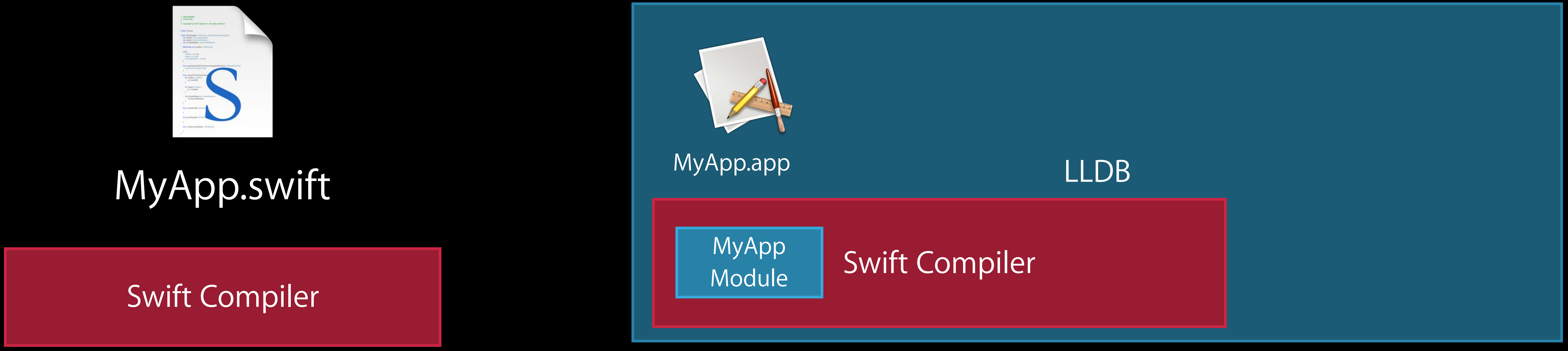
Modules



Modules store the compiler's truth

- No need to reconstruct types from DWARF
- No loss of information

Modules



Modules store the compiler's truth

- No need to reconstruct types from DWARF
- No loss of information

LLDB can see types and functions your program doesn't use

- Yes, generics too!

Summary

Summary

Choose your language

Summary

Choose your language

LLDB provides helpful investigation tools

Summary

Choose your language

LLDB provides helpful investigation tools

We talked about:

- Swift types in LLDB
- Stepping
- Data formatters
- Modules

Summary

Choose your language

LLDB provides helpful investigation tools

We talked about:

- Swift types in LLDB
- Stepping
- Data formatters
- Modules

Your feedback matters!

More Information

Dave DeLong
Developer Tools Evangelist
delong@apple.com

Documentation
Apple Developer Forums
http://devforums.apple.com

LLDB Website
http://lldb.llvm.org

Related Sessions

● Debugging in Xcode 6	Marina	Wednesday 10:15AM
● Introduction to LLDB and the Swift REPL	Mission	Thursday 10:15AM
● Advanced Swift	Presidio	Thursday 11:30AM

Labs

● Swift Lab	Tools Lab A	Friday 9:00AM
● LLDB and Xcode Debugging Lab	Tools Lab B	Friday 9:00AM

WWDC14