UNDERSTANDING DATA RACES AND ACTORS IN SWIFT 5.5

Antoine van der Lee / @twannl

SwiftLeeds October 7th 2021, Leeds, UK

IT'S BEEN A WHILE!

```
func transfer(amount: Double, to other: BankAccountUnsynchronized) throws {
    if amount > balance {
        throw BankError.insufficientFunds
    }

    balanceHistory.append(amount)
    hasBalanceHistory = !balanceHistory.isEmpty

    print("Transfered \(amount) new balance is \(balance)")

    balance = balance - amount
    other.deposit(amount: amount)
}
```

Thread 10: EXC_BAD_ACCESS (code=1, address=0x1c2039cd4230)

```
func transfer(amount: Double, to other: BankAccountUnsynchronized) throws {
    if amount > balance {
        throw BankError.insufficientFunds
    }

    balanceHistory.append(amount)
    hasBalanceHistory = !balanceHistory.isEmpty

    print("Transfered \(amount) new balance is \(balance)")

    balance = balance - amount
    other.deposit(amount: amount)
}
```

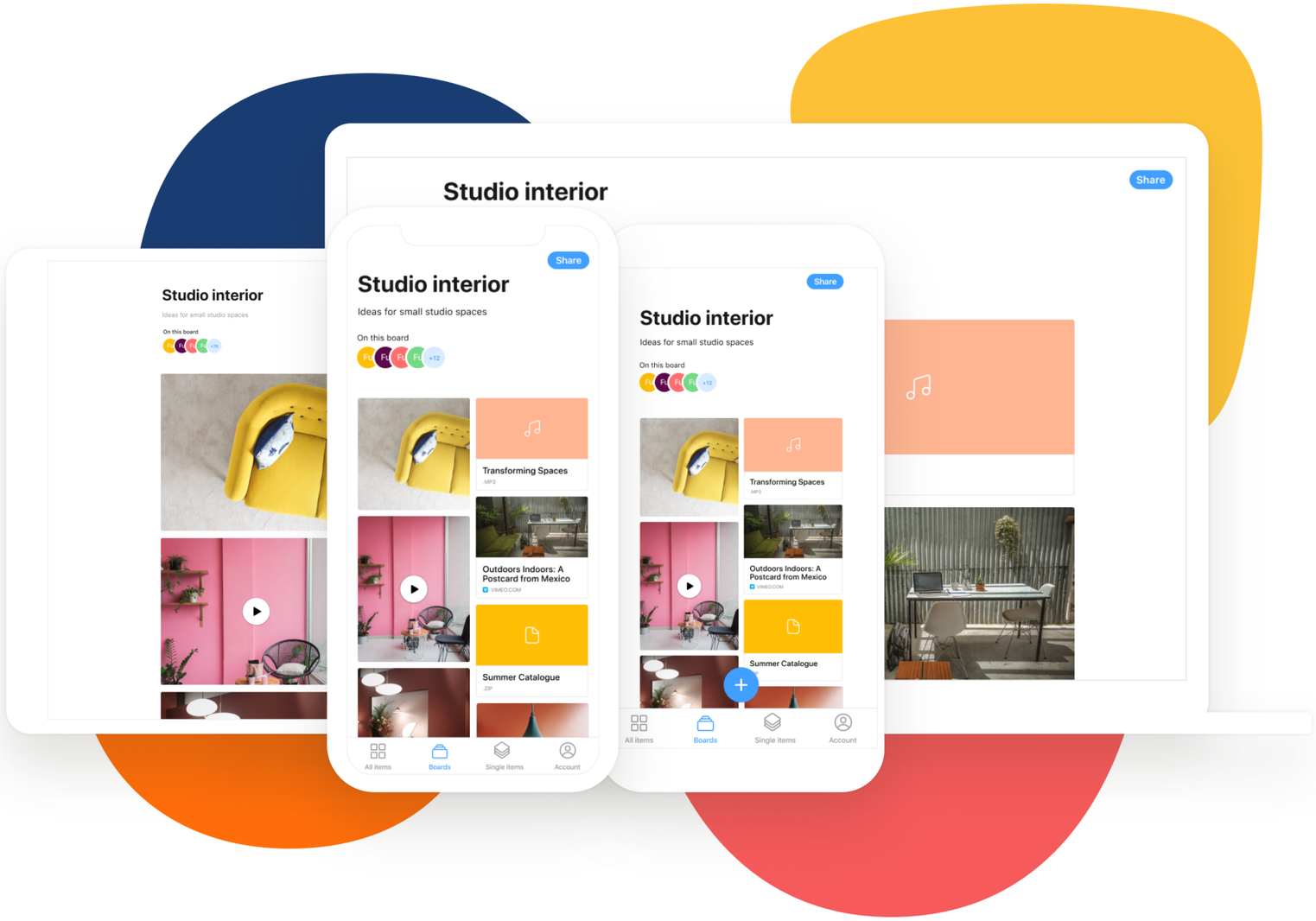
Thread 2: signal SIGABRT

I PROMISE YOU

YOU ARE NOT ALONE

Save everything that inspires your ideas

Collect by WeTransfer



Event summary | 6.0.5 (16686) | iOS 14.8.0 | iPhone SE (2nd generation) | Sep 27, 2021, 9:04:56 AM

Stack trace | Keys | Logs | Data

TXT

Crashed: NSOperationQueue 0x103a33430 (QOS: USER_INITIATED)
EXC_BAD_ACCESS KERN_INVALID_ADDRESS 0x0000000000000010

0	libobjc.A.dylib	objc_msgSend + 32
1	CFNetwork	_CFNetworkHTTPConnectionCacheSetLimit + 141840
2	**Collect**	**<compiler-generated> - Line 4312435188** NetworkProvider.session.getter + 4312435188
3	Collect	**Downloader.swift - Line 67** specialized static Downloader.download(_:using:to:) + 67
4	Collect	**<compiler-generated> - Line 4312418364** static Downloader.download(_:using:to:) + 4312418364
5	Collect	**ContentDownloadOperation.swift - Line 73** ContentDownloadOperation.execute(_:) + 73
6	Collect	**ChainedAsynchronousResultOperation.swift - Line 73** ChainedAsynchronousResultOperation.execute() + 73

WELCOME TO

DATA RACES

What are Data Races?

Data Races occur when

- Multiple threads concurrently access the same data
- At least one of them is a write

SwiftLee Bank

Bank transfer example

```
class BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) {
        guard balance > amount else { return } // Read
        balance -= amount // Write
        other.balance += amount // Write
    }
}

let bankAccountOne = BankAccount(initialDeposit: 100)
let bankAccountTwo = BankAccount(initialDeposit: 100)



bankAccountOne.transfer(amount: 50, to: bankAccountTwo)


bankAccountOne.transfer(amount: 70, to: bankAccountTwo)

print(bankAccountOne.balance) // Prints 50
```

Bank transfer example

```
class BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) {
        guard balance > amount else { return } // Read
        balance -= amount // Write
        other.balance += amount // Write
    }
}

let bankAccountOne = BankAccount(initialDeposit: 100)
let bankAccountTwo = BankAccount(initialDeposit: 100)



bankAccountOne.transfer(amount: 50, to: bankAccountTwo)


bankAccountOne.transfer(amount: 70, to: bankAccountTwo)

print(bankAccountOne.balance) // Prints 50
```

Bank transfer example

```
class BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) {
        guard balance > amount else { return } // Read
        balance -= amount // Write
        other.balance += amount // Write
    }
}

let bankAccountOne = BankAccount(initialDeposit: 100)
let bankAccountTwo = BankAccount(initialDeposit: 100)



bankAccountOne.transfer(amount: 50, to: bankAccountTwo)


bankAccountOne.transfer(amount: 70, to: bankAccountTwo)

print(bankAccountOne.balance) // Prints 50
```

Bank transfer example

```
class BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) {
        guard balance > amount else { return } // Read
        balance -= amount // Write
        other.balance += amount // Write
    }
}

let bankAccountOne = BankAccount(initialDeposit: 100)
let bankAccountTwo = BankAccount(initialDeposit: 100)
```

```
bankAccountOne.transfer(amount: 50, to: bankAccountTwo)


bankAccountOne.transfer(amount: 70, to: bankAccountTwo)

print(bankAccountOne.balance) // Prints 50
```

Bank transfer example

```
class BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) {
        guard balance > amount else { return } // Read
        balance -= amount // Write
        other.balance += amount // Write
    }
}

let bankAccountOne = BankAccount(initialDeposit: 100)
let bankAccountTwo = BankAccount(initialDeposit: 100)
```

```
DispatchQueue.global(qos: .background).async {
    bankAccountOne.transfer(amount: 50, to: bankAccountTwo)
}
DispatchQueue.global(qos: .background).async {
    bankAccountOne.transfer(amount: 70, to: bankAccountTwo)
}
print(bankAccountOne.balance) // Prints -20
```

Bank transfer example

```
func transfer(amount: Double, to other: BankAccount) {
    // Thread 1 & 2 check this at the same time due to multi-threading
    // Thread 1: 100 > 50 = true
    // Thread 2: 100 > 70 = true
    guard balance > amount else { return } // Read
    // Thread 1: 100 - 50 = 50
    // Thread 2: 50 - 70 = -20
    balance -= amount // Write
    other.balance += amount // Write
}
```

Bank transfer example

```
func transfer(amount: Double, to other: BankAccount) {

    // Thread 1 & 2 check this at the same time due to multi-threading
    // Thread 1: 100 > 50 = true
    // Thread 2: 100 > 70 = true
    guard balance > amount else { return } // Read

    // Thread 1: 100 - 50 = 50
    // Thread 2: 50 - 70 = -20
    balance -= amount // Write
    other.balance += amount // Write
}
```

Solving Data Races

Using a lock queue

```
let serialLockQueue = DispatchQueue(label: "banking.lock.queue")

func transfer(amount: Double, to other: BankAccount) {
    serialLockQueue.sync {
        // Only one thread at a time
        // Thread 1: 100 > 50 = true
        // Thread 2: 50 > 70 = false
        guard balance > amount else { return } // Read

        // Thread 1: 100 - 50 = 50
        balance -= amount // Write
        other.balance += amount // Write
    }
}
```

Using a lock queue

```
let serialLockQueue = DispatchQueue(label: "banking.lock.queue")

func transfer(amount: Double, to other: BankAccount) {
    serialLockQueue.sync {
        // Only one thread at a time
        // Thread 1: 100 > 50 = true
        // Thread 2: 50 > 70 = false
        guard balance > amount else { return } // Read

        // Thread 1: 100 - 50 = 50
        balance -= amount // Write
        other.balance += amount // Write
    }
}
```

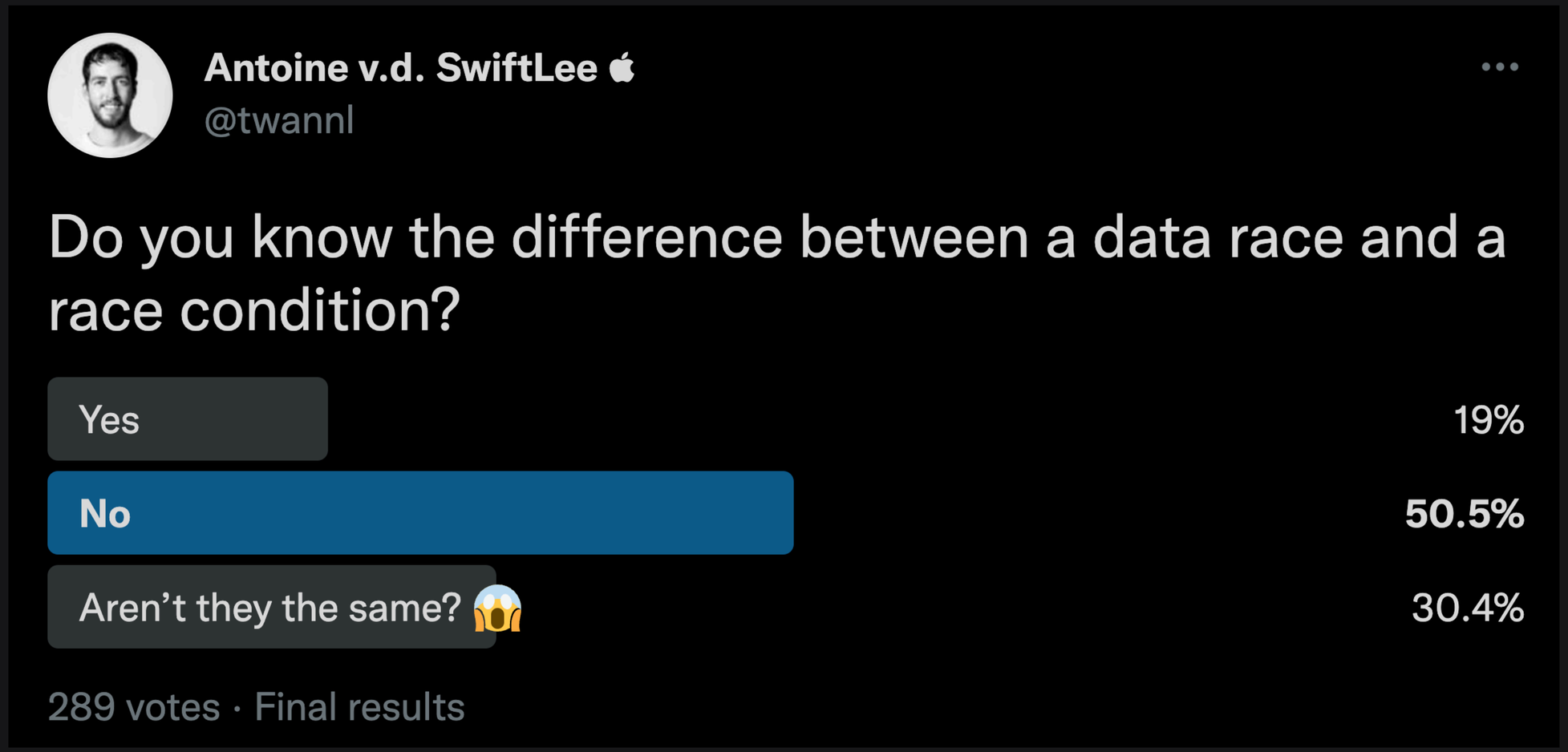
Using a lock queue

```
let serialLockQueue = DispatchQueue(label: "banking.lock.queue")

func transfer(amount: Double, to other: BankAccount) {
    serialLockQueue.sync {
        // Only one thread at a time
        // Thread 1: 100 > 50 = true
        // Thread 2: 50 > 70 = false
        guard balance > amount else { return } // Read

        // Thread 1: 100 - 50 = 50
        balance -= amount // Write
        other.balance += amount // Write
    }
}
```

DATA RACE != RACE CONDITION


Antoine v.d. SwiftLee
@twannl
Do you know the difference between a data race and a race condition?
Yes
19%
No
50.5%
Aren’t they the same? 😱
30.4%
289 votes · Final results

Race condition effect - Thread 1 wins

```
let serialLockQueue = DispatchQueue(label: "banking.lock.queue")

func transfer(amount: Double, to other: BankAccount) {
    serialLockQueue.sync {
        // Only one thread at a time
        // Thread 1: 100 > 50 = true
        // Thread 2: 50 > 70 = false
        guard balance > amount else { return } // Read

        // Thread 1: 100 - 50 = 50
        balance -= amount // Write
        other.balance += amount // Write
    }
}
```

Race condition effect - Thread 2 wins

```
let serialLockQueue = DispatchQueue(label: "banking.lock.queue")

func transfer(amount: Double, to other: BankAccount) {
    serialLockQueue.sync {
        // Only one thread at a time
        // Thread 2: 100 > 70 = true
        // Thread 1: 30 > 50 = false
        guard balance > amount else { return } // Read

        // Thread 1: 100 - 70 = 30
        balance -= amount // Write
        other.balance += amount // Write
    }
}
```

What about actors?

Swift 5.5 Released!

SEPTEMBER 20, 2021  Ted Kremenek

> Ted Kremenek is a member of the Swift Core Team and manages the Languages and Runtimes group at Apple.

Swift 5.5 is now officially released! Swift 5.5 is a massive release, which includes newly introduced language capabilities for concurrency, including `async/await`, structured concurrency, and Actors. My heartfelt thanks to the entire Swift community for all the active discussion, review, and iteration on the concurrency (and other additions) that make up the release. Thank you!

How to learn more

An updated version of The Swift Programming Language for Swift 5.5 is now available on Swift.org. It is also available for free on the Apple Books store.

You can try out some of the new features in this playground put together by Paul Hudson!

SOLVING DATA RACES USING

ACTORS

Actors

- Provide synchronization for shared mutable state
- Ensures mutually-exclusive access to its state

Converting to Actor

```
class BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) {
        guard balance > amount else { return }
        balance -= amount
        other.balance += amount
    }
}
```

Converting to Actor

```
actor BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) async {
        guard balance > amount else { return }
        balance -= amount
        other.balance += amount
    }
}
```

Converting to Actor

```
actor BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) async {
        guard balance > amount else { return }
        balance -= amount
        other.balance += amount
    }
}
```

⛔ Actor-isolated property ‘balance’ can’t be mutated on a non-isolated actor instance

Actor isolation

```
actor BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) async {
        guard balance > amount else { return }
        balance -= amount
        await toAccount.deposit(amount: amount)
    }

    func deposit(amount: Double) {
        balance = balance + amount
    }
}
```

Actor isolation

```
actor BankAccount {
    isolated var balance: Double

    isolated func transfer(amount: Double, to other: BankAccount) async {
        guard balance > amount else { return }
        balance -= amount
        await toAccount.deposit(amount: amount)
    }

    isolated func deposit(amount: Double) {
        balance = balance + amount
    }
}
```

Actor isolation

```
actor BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) async {
        guard balance > amount else { return }
        balance -= amount
        await toAccount.deposit(amount: amount)
    }

    func deposit(amount: Double) {
        balance = balance + amount
    }
}
```

Actor isolation

```
actor BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: isolated BankAccount) async {
        guard balance > amount else { return }
        balance -= amount
        toAccount.balance = balance + amount
    }
}
```

Actor protocol conformances

```
actor BankAccount {
    var balance: Double

    // …
}

extension BankAccountActor: CustomStringConvertible {
    var description: String {
        "\(balance)"
    }
}
```

⛔ Actor-isolated property 'description' cannot be used to satisfy a protocol requirement

Actor protocol conformances

```
actor BankAccount {
    var balance: Double

    // …
}

extension BankAccountActor: CustomStringConvertible {
    nonisolated var description: String {
        "\(balance)"
    }
}
```

⛔ Actor-isolated property 'balance' can not be referenced from a non-isolated context

Actor protocol conformances

```
actor BankAccount {
    var balance: Double
    let accountHolder: String // Immutable

    // …
}

extension BankAccountActor: CustomStringConvertible {
    nonisolated var description: String {
        "\(accountHolder)"
    }
}
```

Global Actors

```
@MainActor
final class BankAccountsViewModel {
    // ..
}

@MainActor func thisExecutesOnTheMainThread() { … }

@MainActor var images: [UIImage] = □

func updateData(completion: @MainActor @escaping () → ()) {
```

Detecting Data Races

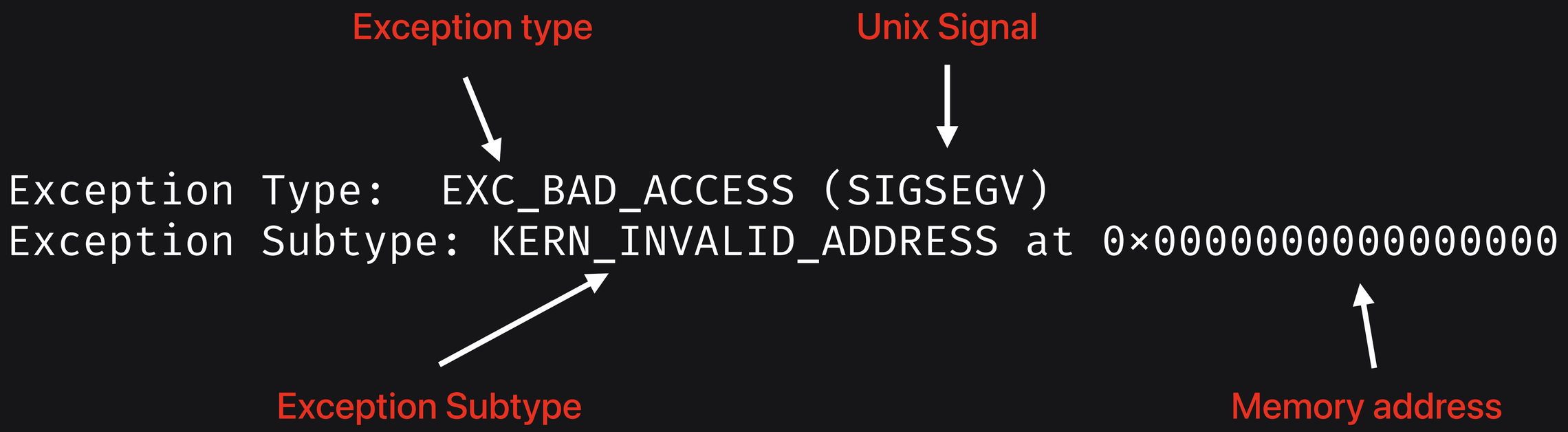
Recognising exceptions in crashes

Crashed: NSOperationQueue 0x103a33430 (QOS: USER_INITIATED)

EXC_BAD_ACCESS KERN_INVALID_ADDRESS 0x0000000000000010

0	libobjc.A.dylib	objc_msgSend + 32
1	CFNetwork	_CFNetworkHTTPConnectionCacheSetLimit + 141840
2	**Collect**	**<compiler-generated> - Line 4312435188** NetworkProvider.session.getter + 4312435188
3	Collect	**Downloader.swift - Line 67** specialized static Downloader.download(_:using:to:) + 67
4	Collect	**<compiler-generated> - Line 4312418364** static Downloader.download(_:using:to:) + 4312418364
5	Collect	**ContentDownloadOperation.swift - Line 73**

Recognising exceptions in crashes



EXC_BAD_ACCESS

Unix Signal, e.g. SIGSEGV, SIGBUS, SEGV_MAPERR

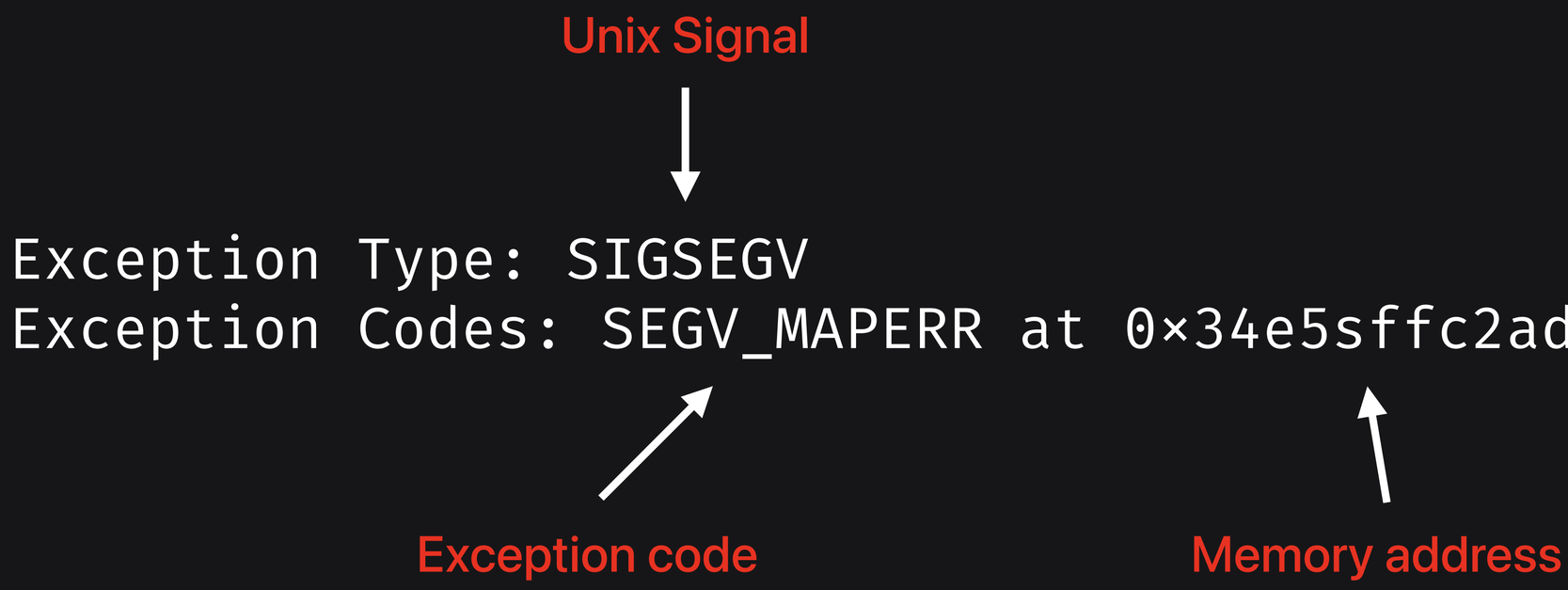
- KERN_INVALID_ADDRESS
  The crashed thread accessed unmapped memory
- KERN_PROTECTION_FAILURE
  The crashed thread tried to use a valid memory address that's protected.
- KERN_MEMORY_ERROR
  The crashed thread tried to access memory that couldn't return data
- EXC_ARM_DA_ALIGN
  The crashed thread tried to access memory that isn't appropriately aligned

Unix Signal

- SIGSEGV
  Segmentation Fault, Invalid memory reference.
- SIGBUS
  Accessing an undefined memory object portion.
- SEGV_MAPERR
  Address not mapped with object.

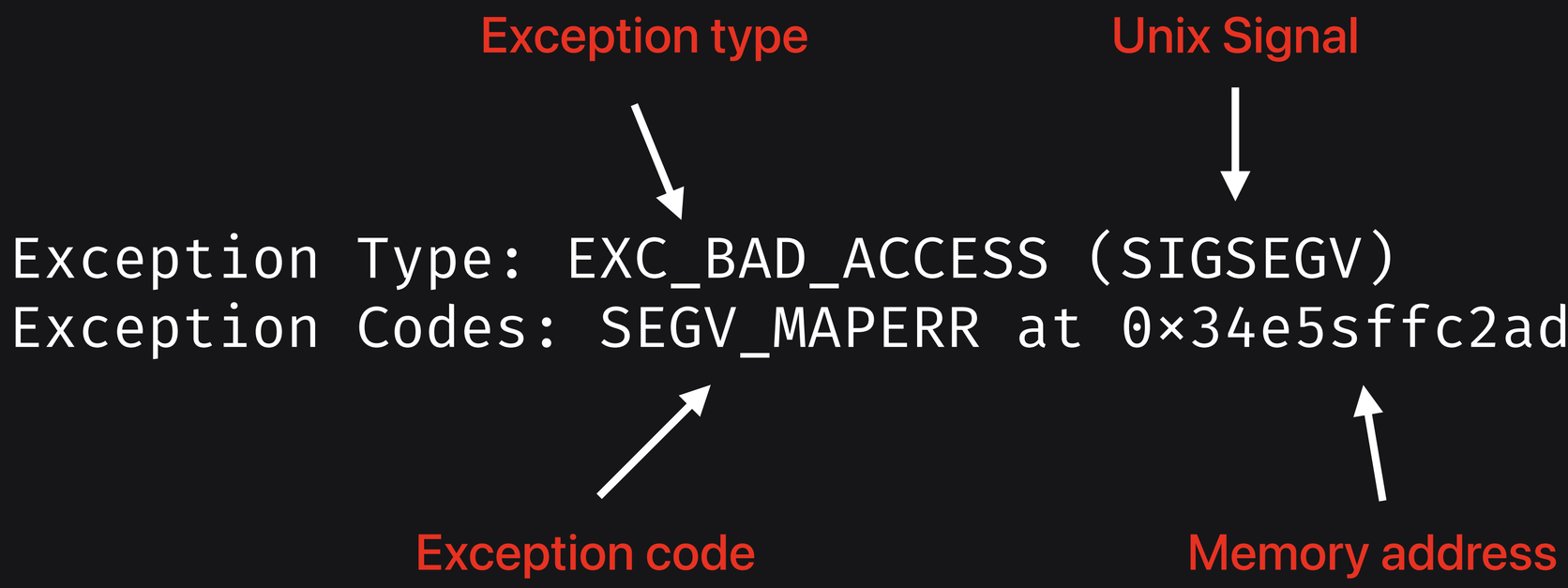
Recognising exceptions in crashes

macOS, occasionally

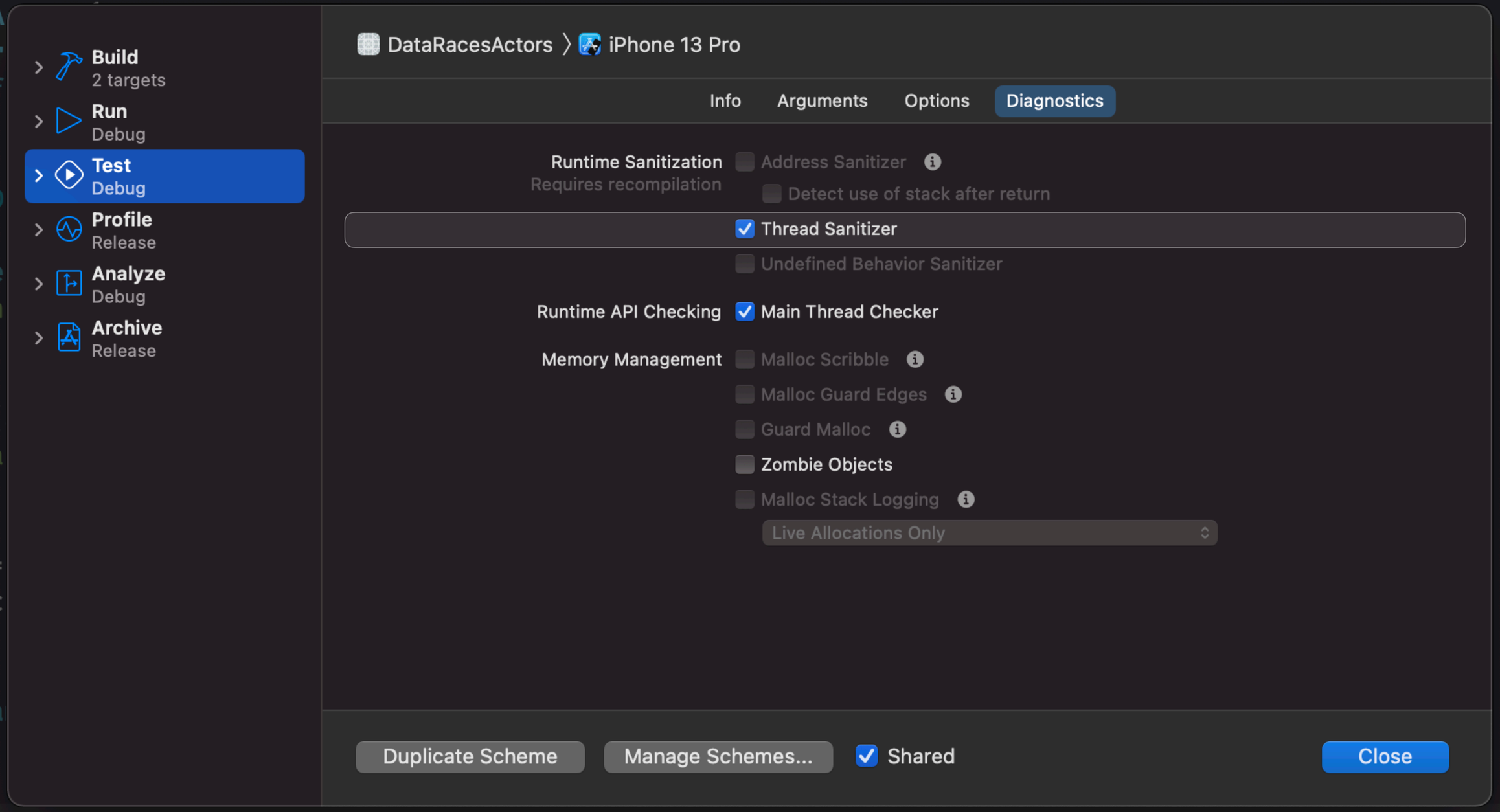


Recognising exceptions in crashes

macOS, occasionally



~~DETECTING~~ PREVENTING DATA RACES


Build
2 targets
Run
Debug
Test
Debug
Profile
Release
Analyze
Debug
Archive
Release
DataRacesActors
iPhone 13 Pro
Info
Arguments
Options
Diagnostics
Runtime Sanitization
Requires recompilation
Address Sanitizer
Detect use of stack after return
Thread Sanitizer
Undefined Behavior Sanitizer
Runtime API Checking
Main Thread Checker
Memory Management
Malloc Scribble
Malloc Guard Edges
Guard Malloc
Zombie Objects
Malloc Stack Logging
Live Allocations Only
Duplicate Scheme
Manage Schemes...
Shared
Close

Thread Sanitizer Enabled

```
class BankAccount {
    var balance: Double

    func transfer(amount: Double, to other: BankAccount) {
        recordAndCheckWrite(balance) // Added by the compiler
        guard balance > amount else { return }
        recordAndCheckWrite(balance) // Added by the compiler
        balance -= amount
        recordAndCheckWrite(balance) // Added by the compiler
        other.balance += amount
    }
}
```

Demo time!

Tie back

Tie back

- Prevent data races using the Thread Sanitizer
- Use Actors or locks for synchronized access
- Data Race != Race condition

SwiftLee Jobs

swiftleejobs.com

SwiftLee

avanderlee.com

SwiftLee Weekly

avanderlee.com/swiftlee-weekly

RocketSim giveaway

SwiftLee

THANKS

@TWANNL

AVANDERLEE.COM

SwiftLeeds