TODAY

- Memory safety
- Generics
CONFLICTING ACCESSES

• **Conflicting** accesses:
  • same location
  • overlapping duration
  • at least one is a write

• Swift accesses can be
  • synchronous (most)
  • longterm (problematic)
    • in-out parameters
    • value-based data

```swift
// A write access to the memory where one is stored.
var one = 1

// A read access from the memory where one is stored.
print("We’re number \(one)!")
```

EXAMPLE: EFFECT OF LONG-TERM ACCESSES

• **Items** and **Total** consistent before and after, but not during.

```
<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>During</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eggs $3</td>
<td>TV $298</td>
<td>Eggs $3</td>
</tr>
<tr>
<td></td>
<td>Cookies $2</td>
<td>T-shirt $17</td>
<td>Cookies $2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TV $298</td>
</tr>
<tr>
<td></td>
<td>Total $5</td>
<td></td>
<td>T-shirt $17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total $320</td>
</tr>
</tbody>
</table>
```

• Not clear what the “right” total would be during update.

• With swift, these issues occur in **single-threaded** execution!
IN-OUT PARAMETERS

• in-out parameters

```swift
var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
}

increment(&stepSize)
// Error: conflicting accesses to stepSize
```

• solve through explicit copies

```swift
// Make an explicit copy.
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// Update the original.
stepSize = copyOfStepSize
// stepSize is now 2
```

• same argument for multiple in-out parameters

```swift
func balance(_ x: inout Int, _ y: inout Int) {
    let sum = x + y
    x = sum / 2
    y = sum - x
}

var playerOneScore = 42
var playerTwoScore = 30
balance(&playerOneScore, &playerTwoScore) // OK
balance(&playerOneScore, &playerOneScore) // Error: conflicting accesses to playerOneScore
```
CONFLICTING ACCESS TO SELF

- conflicting access to self

```swift
import UIKit

struct Player {
    var name: String
    var health: Int
    var energy: Int
    static let maxHealth = 10
    mutating func restoreHealth() {
        health = Player.maxHealth
    }
    
    func balance(_ x: inout Int, _ y: inout Int) {
        let sum = x + y
        x = sum / 2
        y = sum - x
    }
    
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}

var playerInformation = (health: 10, energy: 20)
balance(&playerInformation.health, &playerInformation.energy)
// Error: conflicting access to properties of playerInformation

var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Marie", health: 5, energy: 10)

oscar.shareHealth(with: &maria) // OK
```

CONFLICTING ACCESS TO PROPERTIES

The issue:

- structs, tuples, and enumerations are value-based
- mutating any piece of the value mutates the whole value

- `.health` and `.energy` are the same tuple, so **conflicting writes**
CONFlicting Access To STRUCTS

- `.health` and `.energy` are the same tuple, so conflicting writes

```swift
var holly = Player(name: "Holly", health: 10, energy: 10)
balance(&holly.health, &holly.energy) // Error
```

- Part of the above problem is because `holly` global,
  - use a local variable to constrain compiler proofs:

```swift
func someFunction() {
    var oscar = Player(name: "Oscar", health: 10, energy: 10)
    balance(&oscar.health, &oscar.energy) // OK
}
```

COMPILER SAFETY PROOFS

- Compiler will only allow access that it can prove safe
- “Memory safety” is goal, “exclusive access” stronger than needed
- Compiler can prove “exclusive access” for access iff:
  - accessing stored properties only
  - local variable
  - not captured by closures
    (or only by non-escaping closures)

- Recall:
  - non-escaping closures are:
    - passed to a function
    - executed by function before it returns
    - not stored
GENERICS

- var numbers = [1,2,3]
- var numbers: [Int] = [1,2,3]
- var numbers: Array<Int> = [1,2,3]  // full generic annotation

- Generics in structs, classes, enumerations, functions, initializers

- Generic protocols using associated types

GENERIC FUNCTIONS

```swift
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```

- Have to build another function for strings, etc...
GENERIC FUNCTIONS

```swift
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let tempA = a
    a = b
    b = tempA
}
```

- “T” is a placeholder (also a "type parameter")
  - doesn't specify a type
  - does say a, b must be same type
- Naming type parameters
  - use descriptive names, such as `Element`, or `Item`
  - or T, U, V

GENERIC FUNCTIONS

```swift
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

- `Element` placeholder used:
  - to create property "items"
  - to specify arg to `push()`
  - return value of `pop()`

```swift
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
```
EXTENDING GENERIC TYPES

- Let’s say we want to extend a generic parameter.
  - type parameter name () must be same

```swift
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

TYPE CONSTRAINTS

- Stack defined previous slide can work w/ any type
- Sometimes useful to enforce type constraints
- For example, Dictionary requires keys to be hashable
  - otherwise, none could be found, or inserted

```swift
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

- Type parameter `T` has a type constraint that requires it to be a subclass of SomeClass.
- `U` is required to conform to the protocol SomeProtocol.
TYPE CONSTRAINTS

• Maybe want to search for a string in an array....
  // Non-generic func to find index of string in array.
  func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
      if value == valueToFind {
        return index
      }
    }
    return nil
  }
• but might be useful for non-strings as well, so....
  // Might want a generic function to find values in arrays, and write:
  func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
      if value == valueToFind {
        return index
      }
    }
    return nil
  }
• Doesn’t work, needs T to be constrained to Equatable

Adding constraints to an Associated Type

TYPE ASSOCIATIONS

• Protocols use specific types, what if we want them abstract?
  • “type associations”

  protocol Container {
    // placeholder type
    associatedtype Item: Equatable
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
  }
  • Container requires three capabilities:
    • append() Item func
    • count() Items func
    • retrieval of Item through subscripting
  • Doesn't specify:
    • how the items are stored
    • their type
TYPE ASSOCIATIONS

struct IntStack: Container {
    // original IntStack implementation
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias Item = Int
    mutating func append(_ item: Int) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}

Not actually needed because of type inference from func()s.

GENERIC WHERE CLAUSES

• A Generic Where Clause can require:
  • definition of associated type to conform for protocol conformance
  • certain type parameters and associated type must be same
• Write it right after opening curly brace of type or func
func allItemsMatch<C1: Container, C2: Container>(_ someContainer: C1, _ anotherContainer: C2) -> Bool
where C1.Item == C2.Item, C1.Item: Equatable {
    // Check that both containers contain the same number of items.
    if someContainer.count != anotherContainer.count {
        return false
    }
    // Check each pair of items to see if they're equivalent.
    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }
    // All items match, so return true.
    return true
}

- allItemsMatch() checks to see if instances contain same items in same order
  - they don't have to be the same type of container
  - they do have to be the same type of item

- The type constraints and generic where clause require:
  1. C1 must conform to the Container protocol
  2. C2 must also conform to the Container protocol
  3. The Item for C1 must be the same as the Item for C2
  4. The Item for C1 must conform to the Equatable protocol
  5. (3) and (4) mean that C2 must also conform to Equatable
func allItemsMatch<C1: Container, C2: Container>(_ someContainer: C1, _ anotherContainer: C2) -> Bool
where C1.Item == C2.Item, C1.Item: Equatable {
  // Check that both containers contain the same number of items.
  if someContainer.count != anotherContainer.count {
    return false
  }
  // Check each pair of items to see if they’re equivalent.
  for i in 0..<someContainer.count {
    if someContainer[i] != anotherContainer[i] {
      return false
    }
  }
  // All items match, so return true.
  return true
}

// Generic where clauses can even be used in defining a protocol’s own type constraints:

protocol SuffixableContainer: Container {
  associatedtype Suffix: SuffixableContainer where Suffix.Item == Item
}

extension Stack: SuffixableContainer {
  func suffix(_ size: Int) -> Suffix {
    var result = Stack()
    for index in (count-size)...count {
      result.append(self[index])
    }
    return result
  }
  // Inferred that Suffix is Stack.
}

var stackOfInts = Stack<Int>(){
  stackOfInts.append(10)
  stackOfInts.append(20)
  stackOfInts.append(30)
}

let suffix = stackOfInts.suffix(2)
// suffix contains 20 and 30
EXTENSION W/ GENERIC WHERE

- Extensions can be made to only work for conforming base types
  - extension defined w/ a Where clause
  - funcs in extension only defined for base types that conform

```swift
extension Stack where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false
        }
        return topItem == item
    }
}
```