CS 326
DispatchQueues, Equality, and Hashing

Stephen Freund

Swift Access Levels

- **public**
  - reusable components
  - only operations for clients
- **private**
  - internal representation
  - helper methods
- **internal (default)**
  - module's internal classes/code
  - eg: Controllers specialized for single app
  - internal rep that we want to access in tests...

Multithreading

- DispatchQueues
  - global: general work queues
  - main: serialized UI event queue
- Put code to run on queue:

```
queue.async { body }
```

- General pattern:

```
DispatchQueue.global().async {
  let data = ...long or blocking operation...
  DispatchQueue.main.async {
    update model and UI elements using data
  }
}
```

Animation Until Some Condition

In WindowController:

```
func oneStep() {
  DispatchQueue.global().async { [weak self] in
    usleep(microseconds) or other time consuming work
    DispatchQueue.main.async {
      if (!timeToEnd) {
        update model and UI elements
        self?.oneStep()
      }
    }
  }
}
```

Equivalence Relation

- **Reflexive**: $a = a$
- **Symmetric**: $a = b$ iff $b = a$
- **Transitive**: if $a = b$ and $b = c$, then $a = c$

Reference Equality: ===

```
public class Duration {
  // Rep Inv: min >= 0 && 0 <= sec < 60

  public let min : Int
  public let sec : Int

  public init(_ min : Int, _ sec : Int) {
    assert(min >= 0 && 0 <= sec && sec < 60)
    self.min = min
    self.sec = sec
  }
}

let d1 = Duration(3, 10)
let d2 = Duration(3, 10)
let d3 = d1
```

d1 === d2?
d1 === d3?
d2 === d3?

Equatable Protocol

```
protocol Equatable {
  static func == (lhs: Self, rhs: Self) -> Bool
}
```

From Specification:

Equality implies substitutability—any two instances that compare equally can be used interchangeably in any code that depends on their values. To maintain substitutability, the == operator should take into account all visible aspects of an Equatable type. …

Since equality between instances of Equatable types is an equivalence relation, any of your custom types that conform to Equatable must satisfy three conditions, for any values a, b, and c:

- *a == a is always true (Reflexivity)*
- *a == b implies b == a (Symmetry)*
- *a == b and b == c implies a == c (Transitivity)*

Equatable Durations

```
public class Duration : Equatable {

  // Rep Inv: min >= 0 && 0 <= sec < 60
  public let min : Int
  public let sec : Int

  ...

  static public func ==(_ d1: Duration, _ d2: Duration) -> Bool {
    return d1.min == d2.min && d1.sec == d2.sec
  }
}

let d1 = Duration(3, 10)
let d2 = Duration(3, 10)
let d3 = d1
```

d1 == d2?
d1 == d3?
d2 == d3?

Equatable Durations (2)

```
public class Duration : Equatable {

  // Rep Inv: min >= 0 && 0 <= sec < 60
  public let min : Int
  public let sec : Int

  ...

  static public func ==(_ d1: Duration, _ d2: Duration) -> Bool {
    return d1.min * 60 + d1.sec == d2.min * 60 + d2.sec
  }
}

let d1 = Duration(3, 10)
let d2 = Duration(3, 10)
let d3 = d1
```

d1 == d2?
d1 == d3?
d2 == d3?

Equatable Durations (3)

```
public class Duration : Equatable {

  // Rep Inv: min >= 0 && 0 <= sec < 60
  public let min : Int
  public let sec : Int

  ...

  static public func ==(_ d1: Duration, _ d2: Duration) -> Bool {
    return d1.min == d2.min && d1.sec == d2.sec
  }
}

let d1 = Duration(3, 10)
let d2 = Duration(3, 10)
let d3 = d1
```

d1 == d2?
d1 == d3?
d2 == d3?

Equatable Durations (4)

```
public class Duration : Equatable {

  // Rep Inv: min >= 0 && 0 <= sec < 60
  public let min : Int
  public let sec : Int

  ...

  static public func ==(_ d1: Duration, _ d2: Duration) -> Bool {
    return d1.min * 60 + d1.sec == d2.min * 60 + d2.sec
  }
}

let d1 = Duration(3, 10)
let d2 = Duration(3, 10)
let d3 = d1
```

d1 == d2?
d1 == d3?
d2 == d3?

Equatable Durations (5)

```
public class Duration : Equatable {

  // Rep Inv: min >= 0 && 0 <= sec < 60
  public let min : Int
  public let sec : Int

  ...

  static public func ==(_ d1: Duration, _ d2: Duration) -> Bool {
    return d1.min == d2.min
  }
}

let d1 = Duration(3, 10)
let d2 = Duration(3, 10)
let d3 = d1
```

d1 == d2?
d1 == d3?
d2 == d3?

Swift Details

- You get == for free with:
  - enums
  - structs containing Equatables
  - tuples containing Equatables
- == is a static method
  - Compile-time resolution of operand types.
  - Not like .equals in Java
    - leads to tricky run-time resolution of method calls
    - more in 334…

Subclassing and ==

```
public class LabelledDuration : Duration{

  public let label : String
}

let d1 = LabelledDuration(1,50,"A")
let d2 = LabelledDuration(2,50,"B")
let d3 = LabelledDuration(2,50,"C")
```

d1 == d2?
d1 == d3?
d2 == d3?

Subclassing and ==

```
public class ComparableDuration : Duration, Comparable {

  public static func <(_ d1: ComparableDuration,
                       _ d2: ComparableDuration) -> Bool {
    return d1.min < d2.min
           || (d1.min == d2.min && d1.sec < d2.sec)
  }
}

let d1 = ComparableDuration(1,50)
let d2 = ComparableDuration(2,50)
let d3 = ComparableDuration(2,50)

assert(d1 < d2)
assert(d2 > d1)
assert(d2 >= d3)
```

d1 == d2?
d1 == d3?
d2 == d3?

Subclassing and ==

```
public class NanoDuration : Duration {

  public let nano : Int

  public init(_ min : Int, _ sec : Int, _ nano : Int) {
    self.nano = nano
    super.init(min, sec)
  }
}

let n1 = NanoDuration(0,50,100)
let n2 = NanoDuration(0,50,200)
let n3 = NanoDuration(0,50,200)
```

n1 == n2?
n1 == n3?
n2 == n3?

Subclassing and ==

```
public class NanoDuration : Duration {
  public let nano : Int
  public init(_ min : Int, _ sec : Int, _ nano : Int) {
    self.nano = nano
    super.init(min, sec)
  }
  public static func ==(_ d1: NanoDuration,
                        _ d2: NanoDuration) -> Bool {
    return d1.min == d2.min && d1.sec == d2.sec
           && d1.nano == d2.nano
  }
}

let n1 = NanoDuration(0,50,100)
let n2 = NanoDuration(0,50,200)
let n3 = NanoDuration(0,50,200)
```

n1 == n2?
n1 == n3?
n2 == n3?

But...

```
let d1 = Duration(0,50)
let d2 = NanoDuration(0,50,100)
let d3 = NanoDuration(0,50,200)
```

d1 == d2?
d1 == d3?
d2 == d3?

Composition (Delegation)

```
public class NanoDuration {
  public let duration : Duration
  public let nano : Int

  public static func ==(_ d1: NanoDuration,
                        _ d2: NanoDuration) -> Bool {
    return d1.duration == d2.duration
           && d1.nano == d2.nano
  }
}
```

- Tradeoffs?

Hashable Protocol

- I want a set of Durations:

```
var x = Set<Duration>()
```

error: type 'Duration' does not conform to protocol 'Hashable'

- Protocol definition:

```
protocol Hashable : Equatable {
  var hashValue : Int
}
```

Hashable Protocol Spec

- *You can use any type that conforms to the Hashable protocol in a set or as a dictionary key... A hash value, provided by a type's hashValue property, is an integer that is the same for any two instances that compare equally. That is, for two instances a and b of the same type, if a == b, then a.hashValue == b.hashValue. The reverse is not true: Two instances with equal hash values are not necessarily equal to each other.*
- So:
  - Consistent with equality:
    - a == b ⇒ a.hashValue == b.hashValue
  - Self-consistent:
    - a == a, so a.hashValue = a.hashValue
    - ...so long as a doesn't change between reads

Implementing hashValue

```
public class Duration : Equatable, Hashable {
  public static func ==(_ d1: Duration,
                        _ d2: Duration) -> Bool {
    return d1.min == d2.min && d1.sec == d2.sec
  }

  public var hashValue: Int {
        return 1
        return min
        return min ^ sec
        return 31 &* min &+ sec
        return 60 &* min &+ sec
  }

}
```

Implementing hashValue (2)

```
public class Duration : Equatable, Hashable {
  public static func ==(_ d1: Duration,
                        _ d2: Duration) -> Bool {
    return d1.min * 60 + d1.sec == d2.min * 60 + d2.sec
  }

  public var hashValue: Int {
        return 1
        return min
        return min ^ sec
        return 31 &* min &+ sec
        return 60 &* min &+ sec
  }

}
```

Why Overflowing Operations?

- Integer operations may crash due to overflow/underflow.

```
public class Pair : Hashable {
  let s1 : String
  let s2 : String

  var hashValue : Int {
    return s1.hashValue + s2.hashValue
  }
```

Sum could be greater than Int.max or less than Int.min!

- To avoid:
  - use overflow/underflow-permitting math operators:
    - `s1.hashValue &+ s2.hashValue`
    - `31 &* value1 &+ value2`
  - Better: xor if the values are uniformly distributed
    - `s1.hashValue ^ s2.hashValue`

Equality and Time

- If two immutable objects are equal now, will they always be equal?

```
let d1 = Duration(10,50)
let d2 = Duration(10,50)
assert(d1==d2)
...
assert(d1==d2)
```

- Yes! Abstract value never changes
  - Equality is forever, even if rep changes (benevolent side effect)
- **Behavioural Equivalence**: if a == b, then no sequence of operations applied to a and b can distinguish them.

Equality and Time

- If two mutable objects are equal now, will they always be equal?

```
let d1 = MutableDuration(10,50)
let d2 = MutableDuration(10,50)
assert(d1==d2)
a1.min = 5
assert(d1==d2)
```

- No! Abstract value may change
  - Dangerous! Document precisely. Better: don't do it.
- **Observational Equivalence**: if a == b, then no sequence of *observer* operations can distinguish them.

Observational Equivalence Pitfalls

```
let set = Set<MutableDuration>()
let d1 = Duration(10,50)
let d2 = Duration(20,20)
set.add(d1)
set.add(d2)
d1.min = 20
for d in set {
  print(d)
}
```

- Set's Rep Invariant is broken by mutation after insertion
  - How could we avoid this? Why don't we do that?

Notions of Equality

- **a == b iff:**
  - **Reference**: a and b are the same object
  - **Behavioral**: a and b cannot be distinguished by any sequence of operations
  - **Observational**: a and b cannot be distinguished by any sequence of observer operations