CS 326
Generics

Stephen Freund

Abstraction

- Over computation: procedures (methods)
  - `func distanceFromOrigin(x:Double,y:Double)->Double { return sqrt(x*x + y*y) }`
- Over data: ADTs (classes, structs, protocols, ...)
  - `let p = Point(x:3, y:4)`
- Over types: polymorphism (generics)
  - `let a = Set<Int>()`
    `let b = Set<Double>()`

Related Abstractions: Never Do This...

```
class StringStack {

 var data = [String]()

 func push(_ elem : String) {
  data.append(elem)
 }

 func pop() -> String? {
  return data.popLast()
 }
}
```

```
class IntStack {

 var data = [Int]()

 func push(_ elem : Int) {
  data.append(elem)
 }

 func pop() -> Int? {
  return data.popLast()
 }
}
```

Generic Stack

```
class Stack<Element> {

  var data = [Element]()

  func push(_ elem : Element) {
    data.append(elem)
  }

  func pop() -> Element? {
    return data.popLast()
  }
}
```

`Numeric` is a protocol that `Int`, `Double`, etc. all conform to.

Swift doesn't actually allow us to use `Stack<Numeric>` because of a pesky problem related to using protocols as type parameters, but let's assume it's okay for now... More on this at the end of lecture.

Instantiations

`Stack<Int>`
`Stack<Numeric>`
`Stack<String>`
`Stack<Stack<Int>>`

Type Variables Are Types

```
class Stack<Element> {

  var data = [Element]()

func push(_ elem : Element) {
    data.append(elem)
  }

  func pop() -> Element? {
    return data.popLast()
  }
}
```

Declaration

Use

Use

Use

Bounded Polymorphism

```
class SummableStack<Element : Numeric> {

  var data = [Element]()

  var sum : Element { return data.reduce(0, { $0+$1 }) }

  func push(_ elem : Element) { ... }
  func pop() -> Element? { ... }
}

let st = SummableStack<Int>()
st.push(3)
st.push(4)
assert(st.sum == 7)
```

Upper Bound

Can apply any op defined on upper bound

Instantiations

`SummableStack<Int>`	✓
`SummableStack<Double>`	✓
`SummableStack<String>`	✗

Bounded Polymorphism

```
class SummableStack<Element : Numeric> {

  var data = [Element]()

  var sum : Element { return data.reduce(0, +) }

  func push(_ elem : Element) { ... }
  func pop() -> Element? { ... }
}

let st = SummableStack<Int>()
st.push(3)
st.push(4)
assert(st.sum == 7)
```

Upper Bound

Can apply any op defined on upper bound

Instantiations

`SummableStack<Int>`	✓
`SummableStack<Double>`	✓
`SummableStack<String>`	✗

Extending A Generic Class

```
class SummableStack<Element : Numeric> : Stack<Element> {

  var sum : Element { return data.reduce(0, +) }

}
```

- Benefits:
  - **`SummableStack`** inherits implementation of **`Stack`**
  - **`SummableStack<E>`** is subtype of **`Stack<E>`**

Protocols As Bounds

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

class EqStack<Element: Equatable>: Stack<Element>, Equatable {

  static func == (lhs: EqStack<Element>,
                  rhs: EqStack<Element>) -> Bool {
    return lhs.data == rhs.data
  }
}
```

Instantiations

```
EqStack<Int>                    ✓
EqStack<Double>                 ✓
EqStack<EqStack<String>>        ✓
```

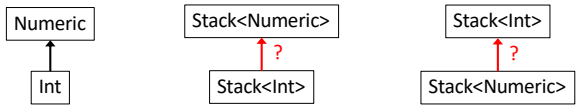
Protocols As Bounds

```
class PrintableSummableStack<Element:
                             Numeric & CustomStringConvertible> :
        SummableStack<Element>, CustomStringConvertible {

  var description: String{ return ... }

}
```

Instantiations

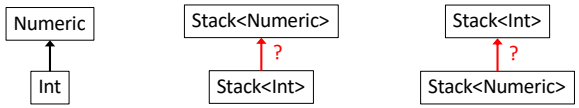
```
PrintableSummableStack<Int>     ✓
PrintableSummableStack<Double>  ✓
```

Generics and Subtyping




```
class Stack<Element> {
  func push(_ elem : Element)
  func pop() -> Element?
}

var s = Stack<Numeric>()  Stack<Int>()
...
...
```

Generics and Subtyping




```
class Stack<Element> {
  func push(_ elem : Element)
  func pop() -> Element?
}

var s = Stack<Int>()  Stack<Numeric>()
...
...
```

Generic Stack

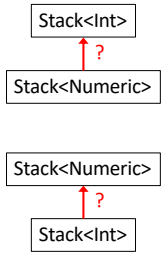
```
class Stack<Element> {

  var data = [Element]()
  func push(_ elem : Element) {
    data.append(elem)
  }

  func pop() -> Element? {
    return data.popLast()
  }
}
```

Instantiations

```
Stack<Int>
Stack<Numeric>
Stack<String>
Stack<Stack<Int>>
```

Generics and Subtyping

```
class Stack<Numeric> {
  func push(_ elem : Numeric)
  func pop() -> Numeric?
}

class Stack<Int> {
  func push(_ elem : Int)
  func pop() -> Int?
}
```



True Subtypes For Classes

```
class A { ... }
class B : A { ... }
```

- If a B object is used in place of an A object, then the result should be consistent with having just used an A object.
- B can only override a method with one having a stronger (or equal) spec
  - Promise More (Stronger Postcondition)
  - Require Less (Weaker Precondition)

Stronger Method Specifications

- Promise More:

```
class Array {

  /// -Returns: index of
  /// key in items
  func index(of key: Int) -> Int
}

class StrongerArray : Array {

  /// -Returns: index of
  /// first occurrence of in
  /// key items
  func index(of key: Int) -> Int
}
```

Stronger Method Specifications

- Require Less:

```
class Array {

  /// -Requires: self is sorted
  func index(of key: Int) -> Int
}

class StrongerArray : Array {

  /// -Returns: true
  func index(of key: Int) -> Int
}
```

Stronger Method Specifications

- Promise More:

```
class A{

  func get() -> Numeric { ... }

}

class B : A {

  func get() -> Int { ... }

}
```

- Covariant Return Types: Subtype returns more specific type

Stronger Method Specifications

- Require Less:

```
class A{

  func set(x : Int) { ... }

}

class B : A {

  func set(x : Numeric) { ... }

}
```

- Contravariant Param Types: Subtype's arg is more general type

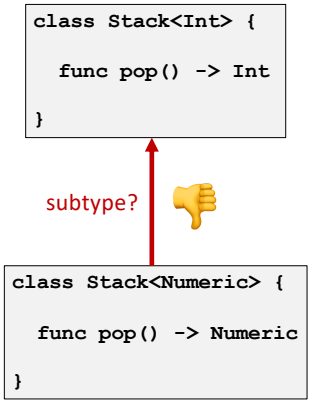
"Read-Only" Stacks

```
class Stack<Numeric> {

  func pop() -> Numeric

}
```

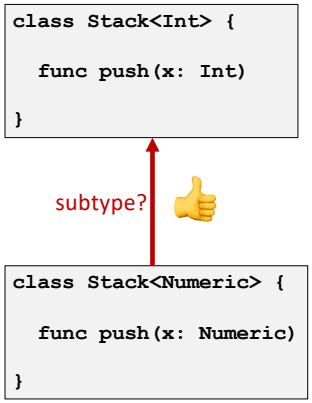
subtype? 

```
class Stack<Int> {

  func pop() -> Int

}
```

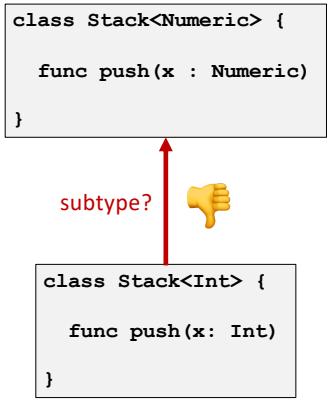
"Read-Only" Stacks



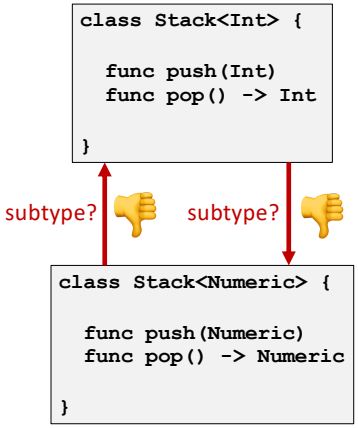
"Write-Only" Stacks



"Write-Only" Stacks

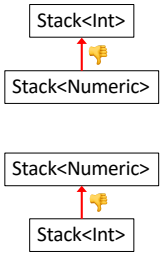


"Read-And-Write" Stacks: Invariant



Generics and Subtyping

```
class Stack<Numeric> {
  func push(_ elem : Numeric)
  func pop() -> Numeric?
}

class Stack<Int> {
  func push(_ elem : Int)
  func pop() -> Int?
}
```



- Subtype must have stronger method specs
  - covariant return types:
    "promise more" (more specific type)
  - contravariant parameter types:
    "require less" (more general type)

Special Cases in Swift

- History: Java array covariance
  - `ColorPoint[] <: Point[]`
- Arrays

```
let cpts = [ColorPoint]()
var pts : [Point] = cpts
pts.append(Point())
let cpts2 = pts as! [ColorPoint]  // unsafe
```

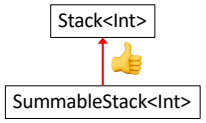
- Optionals

```
let cptOption : ColorPoint? = ColorPoint()
var ptOption : Point? = cptOption
```

- Maps, other built-ins, ...

Extending Generic Class

```
class Stack<Element> {
  func push(_ elem : Element)
  func pop() -> Element?
}

class SummableStack<Element: Numeric> : Stack<Element> {

  var sum : Element { return data.reduce(0, { $0+$1 }) }

}
```



Generic Methods

```
func choose(elems : [Any]) -> Any {
  let i = random(0, elems.count)
  return elems[i]
}

let a = [1,2,3,4]
let v : ??? = choose(a)
```

```
func choose<T>(elems : [T]) -> T{
  let i = random(0, elems.count)
  return elems[i]
}

let a = [1,2,3,4]
let v : Int = choose(a)
```

Generic Methods

```
func flip<S,T>(pair : (S,T)) -> (T,S) {
  let (a,b) = pair
  return (b,a)
}

let t = flip(pair: (3,"cow"))
// t is ("cow, 3)

let t2 = flip(pair:(6.4, CGPoint(x:2,y:43)))
// t is (CGPoint(x:2,y:43), 6.4)
```

Generic Methods With Bounds

```
func index<T: Equatable>(of value: T,
                         in array: [T]) -> Int? {
  for (index, elem) in array.enumerated() {
    if (value == elem) {
      return index
    }
  }
  return nil
}

let i = index(of: "cow", in: ["moo", "cow"])
```

Associated Types

```
protocol Container {
  associatedtype Item
  func append(_ item: Item)
  var count: Int { get }
  subscript(i: Int) -> Item { get }
}
```

Associated Types

```
class IntStack : Container {
  var data = [Int]()
  func push(_ elem : Int) { ... }
  func pop() -> Int? { ... }

  typealias Item = Int

  func append(_ item: Item) { self.push(item) }

  var count : Int { return data.count }

  subscript(i: Int) -> Int { return data[i] }
}
```

Associated Types

```
class IntStack : Container {
  var data = [Int]()
  func push(_ elem : Int) { ... }
  func pop() -> Int? { ... }

  typealias Item = Int // Inferred

  func append(_ item: Item) { self.push(item) }

  var count : Int { return data.count }

  subscript(i: Int) -> Int { return data[i] }
}
```

Associated Types

```
class Stack<Element> : Container {
  var data = [Element]()
  func push(_ elem : Element) { ... }

  func pop() -> Element? { ... }

  //  typealias Item = Element

  func append(_ item: Element) { push(item) }
  var count: Int { return data.count }
  subscript(i: Int) -> Element { return data[i] }
}
```

Associated Types

```
class Stack<Item> : Container {
  var data = [Item]()
  func push(_ elem : Item) { ... }

  func pop() -> Item? { ... }

  //  typealias Item = Item

  func append(_ item: Item) { push(item) }
  var count: Int { return data.count }
  subscript(i: Int) -> Item { return data[i] }
}
```

Bounds on Associated Types

```
protocol EqContainer : Equatable {
  associatedtype Item : Equatable
  func append(_ item: Item)
  var count: Int { get }
  subscript(i: Int) -> Item { get }
}

class EqStack<Item: Equatable> : EqContainer {
  var data = [Item]()
  func push(_ elem : Item) { ... }
  func pop() -> Item? { ... }
  func append(_ item: Item) { self.push(item) }
  var count: Int { return data.count }
  subscript(i: Int) -> Item{ return data[i] }

  static func == (lhs: EquatableStack<Item>,
                  rhs: EquatableStack<Item>) -> Bool {
    return lhs.data == rhs.data
  }
}
```

Extensions With Where Clauses

```
extension Stack where Item: Equatable {
  func topIs(_ elem) -> Bool {
    return count > 0 &&
           self[count-1] == elem
  }
}

extension Container where Item == Double {
  func average() -> Double {
    var sum = 0.0
    for index in 0..<count {
      sum += self[index]
    }
    return sum / Double(count)
  }
}
let st = Stack<Double>()
...
print(st.average())
```

```
protocol EqContainer: Container
where Item : Equatable { }
```

max on Arrays

```
let ints = [1,2,3,4]
print(ints.max()!)
let strings = [ "D", "B", "A", "C"]
print(strings.max()!)

let pts = [ CGPoint(x:1,y:1), CGPoint(x:1,y:1) ]
print(pts.max()) ← Error


struct Array<T> { ... }

extension Array where T : Comparable {
  func max() -> T? { ... }
}
```

Pesky Limitation

```
class Stack<Element> : Sequence {

  var data = [Element]()

  typealias Iterator = Array<Element>.Iterator

  func makeIterator() -> Iterator {
    return data.makeIterator()
  }
}


class MyADT {
  private var stack = Stack<Int>()

  public var sum : Int {
    return stack.reduce(0, +)
  }

  public var elements : Sequence {
    return stack
  }
}
```

Protocol 'Sequence' can only be used as a generic constraint because it has Self or associated type requirements

```
let adt = MyADT()
...
for x in adt.elements {
 // don't know what
 // Sequence.Element is...
}
```

Pesky Limitation

```
class Stack<Element> : Sequence {

  var data = [Element]()

  typealias Iterator = Array<Element>.Iterator

  func makeIterator() -> Iterator {
    return data.makeIterator()
  }
}


class MyADT {
  private var stack = Stack<Int>()

  public var sum : Int {
    return stack.reduce(0, +)
  }

  public var elements : [Int] {
    return Array(stack)
  }
}
```

Fix #1: convert Sequence into an array

```
let adt = MyADT()
...
for x in adt.elements {
  // x is an int!
}
```

Pesky Limitation

```
class Stack<Element> : Sequence {

  var data = [Element]()

  typealias Iterator = Array<Element>.Iterator

  func makeIterator() -> Iterator {
    return data.makeIterator()
  }
}


class MyADT {
  private var stack = Stack<Int>()

  public var sum : Int {
    return stack.reduce(0, +)
  }

  public var elements : AnySequence<Int> {
    return AnySequence<Int>(stack)
  }
}
```

Fix #2: convert Sequence into an AnySequence<T> object

```
let adt = MyADT()
...
for x in adt.elements {
  // x is an int!
}
```

Tips For Writing Generics

- Sometimes teasing out genericity is tricky…
- Start by writing a concrete instantiation
  - Get it correct (testing, reasoning, etc.)
  - Consider writing a second concrete version
- Generalize it by adding type parameters
  - Think about which types are the same or different
  - Are there bounds? Hashable, Equatable, Comparable?
  - The compiler will help you find errors

Generic Methods With Bounds

```
func sum<T: Numeric>(array: [T]) -> T {
  return array.reduce(0, { $0+$1} )
}

let i = sum([1,2,3,4])
let d = sum([1.1,2.2,3.3,4.4])
```