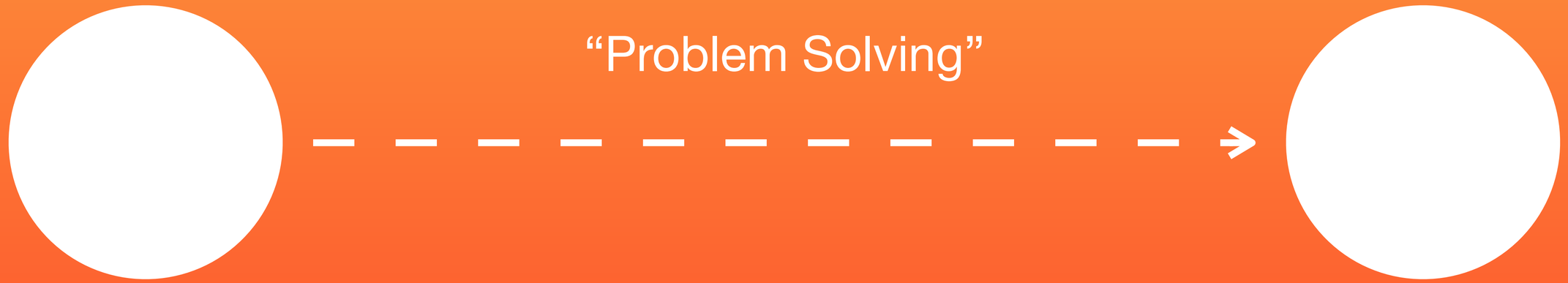
Solving Problems the Swift Way

Ash Furrow
@ashfurrow

1. Better ways to solve familiar problems using Swift
2. Everyone is a beginner again
3. We should share what we learn

Problem-Solving


“Problem Solving”
You are here
You wanna be here

- It would be a shame not to take advantage of these new tools and techniques
- Let’s take a look at some examples

Optionals

- Completely new concept of nil
  - Indicates “missing” value
  - Replaces nil, Nil, NULL, CGRectNull, -1, NSNotFound, NSNull, etc
- Haskell’s “Maybe” monad
- C#’s “Nullable Types”

Optionals

- Works well with Swift's compile-time type safety
  - Which is *awesome*
  - No, seriously, *awesome*
  - Eliminates several classes of bugs
- Don't over-use optional types

Optionals

```
let a = someFunction() //returns Int?
if a != nil {
    // use a!
}
```

Optionals

```
let a = someFunction() //returns Int?
if let b = a {
    // do something with b
}


if let a = a {
    // do something with a
}
```

Tuples

- Tuples are *compound values*
- They are *lightweight*, *temporary* containers for multiple values
- Those values can be named
- Useful for functions with multiple return types

Tuples

```
func calculate() -> (Bool, Int?) {
    // ...
    return (result, errorCode)
}
```

Tuples

```
func calculate() -> (Bool, Int?) {
    // ...
    return (result, errorCode)
}

let calculation = calculate()

if (calculation.0) {
    // …
}
```

Tuples

```
func calculate() -> (Bool, Int?) {
    // ...
    return (result, errorCode)
}

let calculation = calculate()
let (result, _) = calculation

if (result) {
    // …
}
```

Tuples

```
func calculate() -> (result: Bool, errorCode: Int?) {
    // ...
    return (result: result, errorCode: errorCode)
}

let calculation = calculate()
if (calculation.errorCode) {
    // ...
}
```

Tuples

```
for (key, value) in dictionary {
    // ...
}
```

Tuples

- New APIs shouldn’t use out parameters
  - eg: NSError pointers
- Really great for use in *pattern-matching*

Pattern-Matching

- Borrowed from functional programming
  - Really useful in tail-recursive functions
  - Don’t try and apply that technique here
- Like “switch” statements on steroids

Pattern-Matching

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath {
    switch (indexPath.section) {
        case 0:
        {
            switch (indexPath.row) {
                case 0:
                ...
            }
        }
        break;
    }
}
```

Pattern-Matching

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath {
    switch (indexPath.section) {
        case ASHLoginSection:
        {
            switch (indexPath.row) {
                case ASHLoginSectionUserNameRow:
                ...
            }
        }
        break;
    }
}
```

Pattern-Matching

```
override func tableView(tableView: UITableView!,
didSelectRowAtIndexPath indexPath: NSIndexPath!) {
    switch (indexPath.section, indexPath.row) {
    case (0, _):
        ...
    default:
        ...
    }
}
```

Pattern-Matching

```
override func tableView(tableView: UITableView!,
didSelectRowAtIndexPath indexPath: NSIndexPath!) {
    switch (indexPath.section, indexPath.row) {
    case (0, let row):
        ...
    default:
        ...
    }
}
```

Pattern-Matching

```
override func tableView(tableView: UITableView!,
didSelectRowAtIndexPath indexPath: NSIndexPath!) {
    switch (indexPath.section, indexPath.row) {
    case (0, let row) where row > 5:
        ...
    default:
        ...
    }
}
```

Pattern-Matching

```
struct IntList {
    var head: Int = 0
    var tail: IntList?
}

...

switch list {
    case (let head, nil):
        //...
    case (let head, let tail):
        //...
}
```

Generics

- Generics are common in other languages, like C# and C++
- Using a generic type as a placeholder, we can infer the type of variables at compile-time
- A part of Swift's "safe by default" behaviour

Generics

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

Generics

```
var stack = Stack<Int>()

var stack = Stack<String>()

var stack = Stack<Recipe>()
```

Generics

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

func ==<T>(lhs: Stack<T>, rhs: Stack<T>) -> Bool {
    return lhs.items == rhs.items
}
```

Generics

- Use stacks whenever you want to define an abstract data type structure
- Whenever possible, don’t bind new data structures to existing ones
- Use protocols for loose coupling

- Optionals
- Pattern-matching
- Tuples
- Generics

Everyone is a Beginner

Everyone is a Beginner

- No one is an expert in Swift
- This can be kind of stressful
- Relax

Everyone is a Beginner

- The benefits outweighs the cost of learning
  - Depending on your circumstance
- Have your say

Everyone is a Beginner

- The hardest thing is the most important thing
- Start

Everyone is a Beginner

- Don’t be embarrassed to ask questions!
- Try to ask in public so others can benefit from the answer

Everyone is a Beginner

- Let’s borrow ideas

Everyone is a Beginner

- Community-based conventions and guidelines are still being established

We Should Share What We Learn

We Should Share What We Learn

- Conventions and guidelines are still in flux
- There’s an opportunity to significantly alter the future of iOS and OS X programming

We Should Share What We Learn

- The demand for material on Swift is HUGE
- Great opportunity to get known

We Should Share What We Learn

- When you teach, you learn

We Should Share What We Learn

- If we all share what we learn, we all get smarter
- Rising tides lift all boats

We Should Share What We Learn

- Stack Overflow
- Blogs
- Tweets
- Gists
- Open source
- Radars

1. Better ways to solve familiar problems using Swift
2. Everyone is a beginner again
3. We should share what we learn

Let’s Make Better Mistakes Tomorrow

Thank you

@ashfurrow