Aprendendo a programar para iOS

Entendendo o Código

Tipos primitivos

- int
- float
- double
- char
- long
- bool

Variáveis

- Declaração de variáveis:

```
int a;
float b = 2.0;
bool c, d = true, e = YES, f = NO;
```

Operadores

- ++ --
- * / %
- + -
- == != < > <= >=
- && || !

Entradas e Saídas

```
NSLog(@"Hello world!");

int var;
scanf("%i", &i);

NSLog(@"Hello, World!  %i", var);
NSLog(@"Hello, World!  %@", objeto);
```

Comando condicional

```
if (a == 0) {

} else if (a == 1) {

} else {

}
```

Comando condicional

```
switch(expression){
   case constant-expression  :
      statement(s);
      break; /* optional */
   case constant-expression  :
      statement(s);
      break; /* optional */
    /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

Comando condicional

```
int a = (b > 2 ? 4 : -5);
```

Comando de repetição - while

```
while(condition) {
   statement(s);
}
```

```
#import <Foundation/Foundation.h>
int main () {
   int a = 10;
   while( a < 20 ) {
      NSLog(@"value of a: %d\n", a);
      a++;
   }
   return 0;
}
```

Comando de repetição - for

```
#import <Foundation/Foundation.h>
int main () {
   int a;
   for( a = 10; a < 20; a = a + 1 ) {
      NSLog(@"value of a: %d\n", a);
   }
   return 0;
}
```

Comando de repetição – do while

```
#import <Foundation/Foundation.h>
int main () {
   int a = 10;
   do {
      NSLog(@"value of a: %d\n", a);
      a = a + 1;
   } while( a < 20 );
   return 0;
}
```

Compando de repetição

- **break**: comando que interrompe qualquer repetição, pulando automaticamente para o próximo comando após a mesma.

- **continue**: comando que pula para a próxima iteração da repetição, ignorando todos os comandos posteriores ao continue até o limite da repetição atual.

POO – Programação Orientada a Objetos

- Classes e objetos
- Atributos/Propriedades
- Métodos
- Construtores
- Encapsulamento
- Herança
- Interfaces
- Extensão

Classes e Objetos

- Classes são criadas em dois arquivos separados:
  - .h => Arquivo com a declaração da classe e dos seus métodos e atributos públicos.
  - .m => Arquivo com a implementação da classe, com o corpo de todos métodos (tanto públicos quanto privados).

Classes e Objetos

```
//Arquivo box.h
#import <Foundation/Foundation.h>

@interface Box:NSObject
{
    //Instance variables
    double length;   // Length of a box
    double breadth;  // Breadth of a box
}
@property double height; // Property
- (double) volume; // Method

@end
```

Classes e Objetos

```
//Arquivo box.m
#import "box.h"

@implementation Box
-(id)init { // Construtor
    self = [super init];
    length = 1.0;
    breadth = 1.0;
    return self;
}
-(double) volume {
    return length*breadth*height;
}
@end
```

Atributos

- Atributos de instância são definidas no arquivo .h
- Elas não podem ser acessadas de outras classes ou arquivos, apenas do .m da própria classe. Atributos não são gerenciadas pelo ARC

```
@interface Box:NSObject
{
    double length;   // Length of a box
    double breadth;  // Breadth of a box
}
...

@end
```

Propriedades

- Propriedades são definidas no .h ou no .m
- Elas podem ser acessadas de outras classes se estiverem no .h
- Propriedades são gerenciadas pelo ARC, se desejarmos

```
#import <Foundation/Foundation.h>
@interface Box:NSObject

@property double height;
@property NSString *name;
@property ClassName *variable_name;

@end
```

Propriedades

- Opções podem ser passadas na definição de uma propriedade para mudar a forma como os gets/sets/variável de instância é criada.
- Também é possível dar dicas ao ARC sobre como gerenciar a variável na memória.

```
@property (readonly/readwrite) double height;
@property (atomic/nonatomic) NSString *name;
@property (strong/weak) NSString *name;
```

Métodos

- Métodos privados são criados apenas no arquivo .h
- Métodos públicos são declarados no .h e criados no .m

```
#import <Foundation/Foundation.h>

@interface Box:NSObject

- (int) nome_do_metodo;

@end
```

Métodos

```
//Arquivo box.m
#import "box.h"

@implementation Box

-(int) nome_do_metodo {
    return 10;
}

- (return_type) method_name:(argument_type) argumentName1
joiningArgument2: (argument_type) argumentName2 ...
joiningArgumentn: (argument_type) argumentNamen {
    body of the function
    return value;
}

@end
```

Métodos

```
//Arquivo box.m
-(void) metodoVoid {
    //métodos com tipo de retorno void não retornam nada.
}
-(int) metodoInt {
    //métodos com tipo de retorno diferente de void devem ter
um return.
    return 50;
}

-(double) somaIsso: (double) a comIsso: (double) b {
    return a + b;
}

-(double) somaIsso: (double) a comIsso: (double) b eIsso:
(double) c {
    return a + b + c;
}
```

Usando Métodos

```
Box *a = [[Box alloc] init];
[a metodoVoid];

int b = [a metodoInt];
double c = [a somaIsso: 2 comIsso: 3];
double d = [a somaIsso: 2 comIsso: 3 eIsso: 5];
```

Construtores

- Métodos que retornam o tipo id, precisam chamar o construtor da classe pai explícitamente e precisam terminar com um return self;
- Podem existir vários construtores, se a assinatura do método for diferente.
- É preciso colocar a declaração no arquivo .h

Construtores

```
#import <Foundation/Foundation.h>

@interface Box:NSObject
-(id)initWithLength: (double) l;
@end

#import "box.h”
@implementation Box

-(id)init {
    self = [super init];
    self.length = 0;
    self.breadth = 0;
    return self;
}
-(id)initWithLength: (double) l; {
    self = [super init];
    self.length = l;
    breadth = 1.0;
    return self;
}
@end
```

Exemplo

```
#import <Foundation/Foundation.h>

@interface Box:NSObject

@property double height;
@property double width;
@property double length;

- (double) volume;

@end
```

Exemplo

```
#import "box.h"

@implementation Box

-(id)init { // Construtor
    self = [super init];
    self.length = 1.0;
    self.width = 1.0;
    self.height = 1.0;
    return self;
}
-(double) volume {
    return self.length*self.width*self.height;
}
@end
```

Exemplo

```
#import <Foundation/Foundation.h>
#import "box.h"

int main (int argc, const char * argv[])
{
    NSLog (@"hello world");
    Box *b;
    b = [[Box alloc] init];
    NSLog(@"Volume: %d", [b volume]);
    b.height = 4;
    b.width = 3;
    b.length = 5;
    NSLog(@"Volume: %d", [b volume]);
    return 0;
}
```

Exercícios

Material do slide 2, sobre a linguagem:

http://bit.ly/ios-1-pt2

Pra quem quiser praticar em casa, com uma versão um pouco mais antiga do objective-c:

http://bit.ly/objective-c-online

Exercícios

Crie uma classe chamada ContaCorrente com as seguintes especificações:

- Atributos da classe ContaCorrente:
  numeroConta (inteiro), **correntista** (NSString), **saldo** (double)
- Implementar apenas um construtor recebendo valores para os atributos *numeroConta* e *correntista* da classe *ContaCorrente.*
- Todas as propriedades da classe devem ser readonly.
- Implementar o método **boolean deposita(double valor)** que deposita um valor na conta corrente. O método retorna verdadeiro se o depósito foi realizado com sucesso ou falso em caso contrário. OBS: Verificar se o valor informado é maior que zero.
- Implementar o método **boolean saque(double valor)** que realiza um saque na conta corrente. O método retorna verdadeiro se o saque foi realizado com sucesso ou falso e caso contrário. OBS: Verificar se o valor informado é maior que zero e se há saldo suficiente para realizar o saque.
- Implementar o método boolean transfere(double valor, ContaCorrente c2) que realiza uma transferência de um valor da conta corrente para a conta corrente c2. O método retorna verdadeiro se a transferência foi realizada com sucesso ou falso em caso contrário. OBS: Verificar se o valor informado é maior que zero, se o objeto c2 não é nulo e se há saldo suficiente para realizar a transferência.

Exercícios

No Main, fazer:

- Instanciar dois objetos do tipo ContaCorrente, com dados hardcoded (número e nome do correntista). Em seguida, o programa deve oferecer um menu para o usuário com as seguintes opções:
- Imprimir dados de uma conta. Para esta opção o usuário deverá informar o número da conta;
- Realizar depósito. Para esta opção o usuário deverá informar o número da conta e o valor para depósito;
- Realizar saque. Para esta opção o usuário deverá informar o número da conta e o valor para saque;
- Realizar transferência. Para esta opção o usuário deverá informar o número da conta origem, o número da conta destino e o valor para transferência;
- **OBS:** Para as opções de depósito, saque e transferência, o programa deve imprimir na tela uma mensagem indicando se o depósito, o saque ou a transferência foi realizado com sucesso ou não.

Gabarito

- Projeto solução: https://gist.github.com/crystianwendel/37d6e9784c6bd1bdb174f7f30c377fab

Encapsulamento

- Propriedades criam os métodos get/set automáticamente.
- Mas podemos especificá-los apenas criando um método com o mesmo nome que o objective-c usaria.
- Podemos inibir a criação do get ou do set configurando a propriedade com readonly, writeonly ou readwrite.

Encapsulamento

- Toda propriedade cria automaticamente um atributo de instância com um nome similar, apenas com um _ antes.
- O uso da variável com o _ ao invés do método encapsulado pula o gerenciamento de memória ARC.
- Mas podemos especificá-los apenas criando um método com o mesmo nome que o objective-c usaria.

Encapsulamento

```
@property NSString *firstName;


- (NSString *) firstName {
    return _firstName;
}


- (void) setFirstName: (NSString *) name {
    _firstName = name;
}
```

Herança

- A classe pai de todas as que usamos é a NSObject.
- Herança é definida no arquivo .h, na declaração @interface
- Para acessar a classe pai, quando estiver em um método da classe filha: super

```
#import <Foundation/Foundation.h>

@interface Box:NSObject

@end
```

Interfaces

- Em Objective-c, interfaces são chamadas de **PROTOCOLOS**.
- Uma classe pode implementar quantos protocolos você desejar.
- A declaração que uma classe implementa um protocolo fica no arquivo .h
- A implementação do método que o protocolo exige que a classe crie fica no arquivo .m
- Podem existir métodos opcionais na declaração do protocolo.

Interfaces

```
@protocol XYZPieChartViewDataSource
@required // Essa declaração é opcional
- (NSUInteger)numberOfSegments;
- (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
@optional
- (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
@end

//Classe que implementa o protocolo
@interface MyClass : NSObject <MyProtocol>
//a definição dos métodos deve estar no .m
@end

// Declaração de uma propriedade que é uma classe qualquer, que
implementa o meu protocolo
@property (weak) id <XYZPieChartViewDataSource> dataSource;
```

Extensão

- Em objective-c, podemos adicionar métodos em classes existentes através da criação de **CATEGORY.**
- Toda category tem um nome, e os métodos só podem ser usados quando você referenciar o arquivo .h da category.

```
@interface ClassName (CategoryName)
– (void) novoMetodo;
@end
```

Extensão

```
// Arquivo XYZPerson+XYZPersonNameDisplayAdditions.h
#import "XYZPerson.h”
@interface XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString;
@end

// Arquivo XYZPerson+XYZPersonNameDisplayAdditions.m
#import "XYZPerson+XYZPersonNameDisplayAdditions.h"

@implementation XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString {
    return [NSString stringWithFormat:@"%@, %@",
self.lastName, self.firstName];
}
@end
```