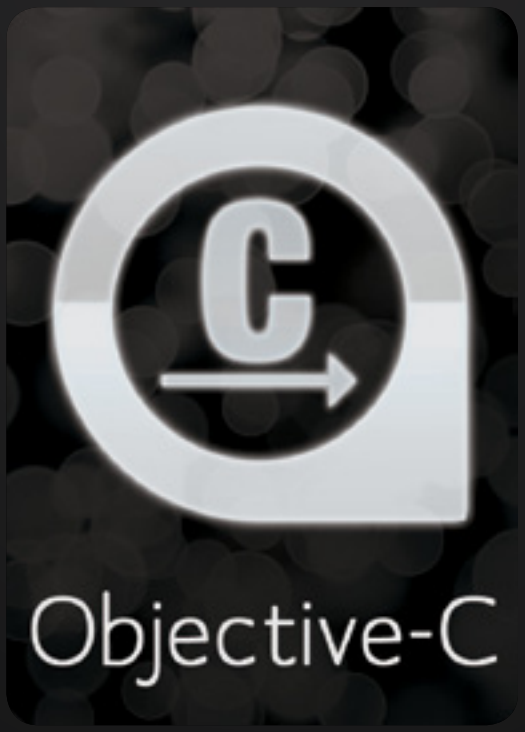
como funciona?

Objective-C



Apostila do Aluno

iai?
INSTITUTO DE ARTES INTERATIVAS

iOS

Elaboração do Conteúdo
André Baldi

Elaboração dos Exercícios
Bruno Eiti

Diagramação
Samuel Sanção de Moura

Edição nº1
Janeiro /2012

iai? Instituto de Artes Interativas
Rua Amauri, 352 - Itaim Bibi
CEP: 01448-000 - São Paulo, SP, Brasil
(11) 3071-4017 | contato@iai.art.br
http://iai.art.br

Sobre o iai?

A história do iai?

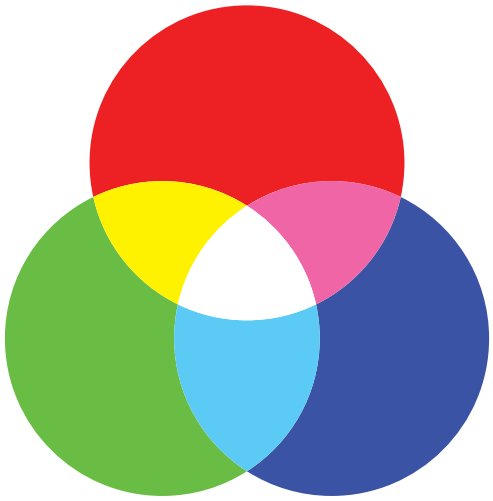
O iai? Instituto de Artes Interativas foi fundada em março de 2009 com cursos de desenvolvimento de aplicativos para iOS contando com uma proposta inovadora e atualizada, nestes últimos 3 anos tivemos mais de 2000 alunos em nossos cursos.

A produtora já trabalha a 2 anos atendendo a demanda do mercado por aplicativos para plataformas móveis como o iOS e Android.

Dessa forma, procuramos uma equipe coesa, criativa, e eficiente para desenvolver soluções e inovações no mundo das artes interativas e educação.

Galeria

Exposição de artes e locação
para eventos corporativos



Escola

Cursos práticos, rápidos
e objetivos

Produtora

Desenvolvimento de
aplicativos móveis

Na área educacional, O Iai? se destaca na oferta cursos com aprendizagem rápida, presencial e a distância, contando com know how em sistemas móveis como iPhone SDK, Android SDK e Windows phone 7, além de Lógica de programação, design, interfaces entre outros. Os cursos do iai? são voltados para profissionais que buscam a capacitação por meio do conhecimento e domínio de novas ferramentas tecnológica com cursos práticos e objetivos aonde aprendem produzindo.

A Produtora atua no mercado no desenvolvimento de projetos em diversos clientes em vários mercados. Aplicativos móveis, peças de marketing e publicidade nesses dispositivos,

Como um espaço de exposição de novas idéias, a Galeria...
Nossa força está nos profissionais qualificados, com professores que também atuam do mercado em diversos projetos, essa é forma de garanti além do conhecimento técnico, dicas, conceitos e informações sobre o mercado.

Como encontrar o iai?

O Iai? conta com várias formas de se comunicar com o seus clientes e amigos, use a forma mais fácil e prática para você.

iai.art.br

instituto de artes interativas

@iaibrasil

iai? instituto de artes interativas

iai.art.br/clube-iai.html

Outros cursos do iai?

IOS SDK
Programação para o sistema operacional da Apple

ANDROID SDK
Programação para o sistema operacional da Google

Lógica de programação
Conceitos Básicos de programação

Orientação a objeto e Objective C
Conceitos de desenvolvimento de sistemas e Linguagem de programação base para o IOS.

Orientação a objeto e Java
Conceitos de desenvolvimento de sistemas e Linguagem de programação base para o Android.

Design
O objetivo do curso é ensinar a montar um projeto de design utilizando como plataforma um aplicativo para iPhone.

Interface
O curso é focado nas guias da Apple e pretende discutir as melhores formas de interface.

Games
Focamos em estratégias e técnicas para desenvolvimento de jogos para o Iphone, demonstrando o funcionamento do Cocos2D.

Windows Phone 7
Venha desenvolver para o mais novo sistema operacional da Micrsoft que promete mexer com Mercado de tecnologia.

Maiores informações acesse: WWW. IAI.ART.BR

Sumário

Sumário

Sumário

Sumário

Capítulo 1

1.1) Meu Primeiro Mac

Para iniciarmos o desenvolvimento para a plataforma iOS é necessário possuir um computador com o sistema operacional Macintosh Operating System (Mac OS). A primeira versão foi lançada em 1984 e até a versão 7.6, era apenas chamado System (ex.: System 4, System 7), da versão 7.6 em diante passou a ser chamado Mac OS. As mais recentes denominam-se Mac OS X.

O Mac OS X é um sistema operacional UNIX baseado no sistema operacional NeXTSTEP que foi desenvolvido pela NeXT Computer. O desenvolvimento do Mac OS X foi possível apenas após a aquisição da NeXT pela Apple em 1996, e o retorno de Steve Jobs ao comando da empresa. O Mac OS X também foi base do sistema operacional iOS usado no iPhone, iPod Touch e iPad.

Em 2006 foi lançado o sistema da Apple para processadores Intel. Apesar de ter sido considerada um "major release", grande parte das alterações do sistema não foram estéticas mas mais funcionais. Com a adoção exclusiva da arquitetura x86 o código legado de suporte a processadores RISC foi excluído da build, e grande parte do sistema (como o Finder) foram reescritos utilizando-se da Cocoa API, gerando um sistema mais ágil e enxuto.

Figura 1.1: Tela inicial do Mac OS X Lion (10.7)

1.2) Finder

O Finder é a aplicação padrão utilizada para gerenciamento de arquivos no Mac OS X. É responsável pela interação do usuário com os arquivos, discos, diretórios, compartilhamentos de rede e aplicações instaladas no sistema. É o primeiro aplicativo aberto após a inicialização do sistema, e o ponto de partida para um usuário Mac.

Menu do Aplicativo

O menu do aplicativo é composto por diversas opções sendo as principais: menu de aplicação, a barra de tarefas, a barra lateral, a janela principal e a barra de informações.

Menu de Aplicação do Finder

Abaixo as principais funcionalidades de cada item do menu:

- Finder: Verificar a versão do Finder, configurar as preferências, esvaziar a lixeira e esconder as janelas ativas.
- Arquivo: Possibilita a criação de arquivos e pastas.
- Editar: Opções de recortar, copiar e colar arquivos, assim como mostrar o que está na área de transferência.
- Visualizar: Podemos configurar o modo de exibição de arquivos e modificar configurações de componentes.
- Ir: Possui atalhos do sistema para acesso rápido.
- Janela: Opções de minimizar, maximizar, esconder.
- Ajuda: Central de ajuda do Finder.

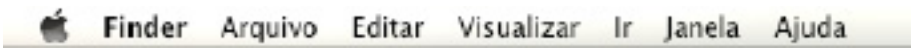


Figura 1.2: Menu do Finder

Todo aplicativo do Mac possui um menu como este, personalizado para suas funções.

Barra de Tarefas

A barra de tarefas padrão possui os controles de navegação (voltar e adiantar), os controles de visualização de Arquivos e diretórios (onde podemos selecionar vê-los por lista, cascata ou coverflow), o botão do quicklook (um visualizador para vários formatos de arquivo),

do controle de ações e a barra de busca do Spotlight.

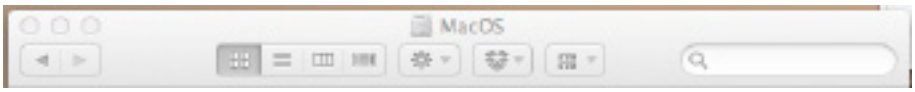


Figura 1.2: Barra de tarefas do Finder

Barra Lateral

A barra lateral é responsável pela exibição dos dispositivos de armazenamento, compartilhamentos de rede, e área do usuário, com suas pastas particulares. Possui ainda uma busca de arquivo acessados por data.

Janela Principal

A janela principal, exibe os arquivos e diretórios presentes no objeto selecionado na barra lateral, existem vários modos de exibição que podemos selecionar na barra de tarefas ou no menu do aplicativo.

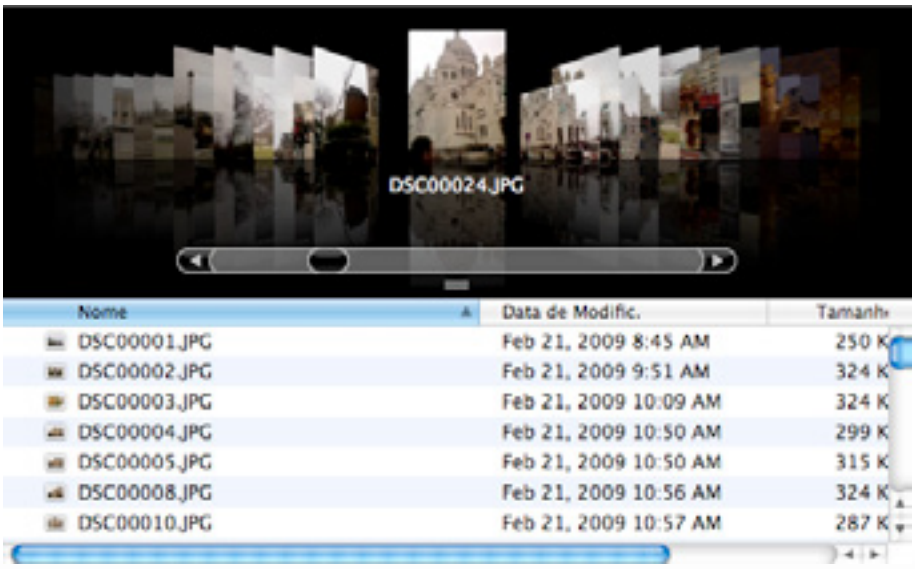


Figura 1.3: Janela principal

Barra de informações

A barra de informações exibe detalhes dos arquivos selecionados assim como também da pasta que os contém.

Figura 1.4: Barra de tarefas

1.3) Shortcuts

Os shortcuts ou atalhos, são combinações de teclas que executam algum comando do programa, agilizando sua utilização. Apesar de alguns serem globais, muitos aplicativos implementam seus próprios atalhos que, na maioria da vezes, podem ser encontrados em seus menus de aplicação. Abaixo segue um guia com a simbologia achada usualmente na documentação dos aplicativos e as teclas correspondentes:

Símbolo	Tecla	Síbolo	Tecla
⌘	command	→,←,↑,↑	setas
⌥	option	⌫	delete
	shift		escape
^	control	↵	return
⇥	tab	⏏	eject

Tabela 1.1: Símbolos mais comuns dos atalhos

Abaixo os atalhos mais comuns nos aplicativos:

Ação	Atalho
Force Quit	⌘+⌥+
Alternar entre programas	⌘+⇥
Copiar	⌘+C
Colar	⌘+V
Recortar	⌘+X
Desfazer	⌘+Z

Fechar programa	⌘+Q
Fechar janela	⌘+W
Minimizar janela	⌘+M
Esconder programa	⌘+H
Salvar	⌘+S
Abrir arquivo	⌘+O
Screenshot de uma região	⌘+⇧+4
Screenshot de um programa	⌘+⇧+4+espaço
Screenshot da tela toda	⌘+⇧+3

Tabela 1.2: Atalhos mais comuns dos programas

1.4) Spotlight

Spotlight é o sistema de buscas do Mac OS X. Ele possui um índice dos arquivos presentes no sistema, e efetua a busca no momento que iniciamos a digitação em sua caixa de texto. Ele está presente tanto na barra de status do sistema como em qualquer janela do Finder. Ele busca arquivos, tanto por nome como por conteúdo, assim como diretórios, conteúdo de e-mail e páginas acessadas do Safari de acordo com as preferências atribuídas a ele.

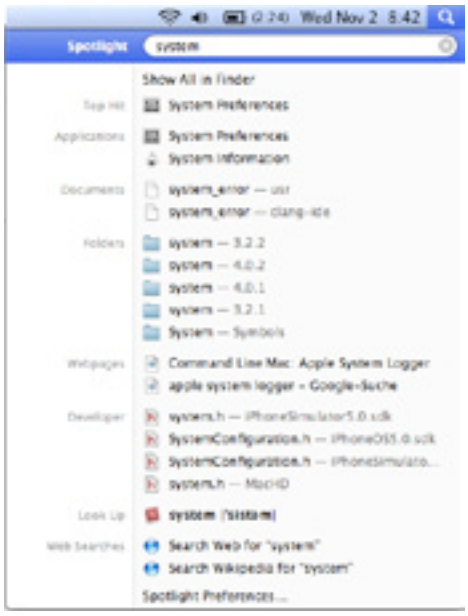


Figura 1.5: Spotlight

1.5) Preferências do Sistema

A janela de preferências do sistema permite-nos configurar vários aspectos do Mac OS X. Nela configuramos nossos dispositivos de entrada e saída (monitores, mouses, teclados), selecionamos o idioma do sistema, configuramos contas de usuários, compartilhamentos de rede, impressoras, além das configurações de aparência do sistema.

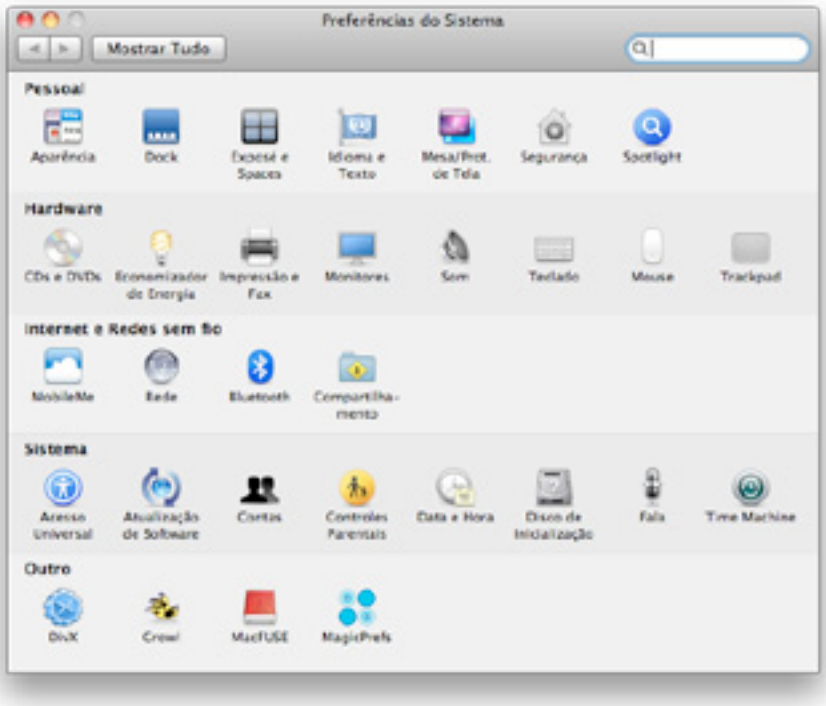


Figura 1.6: Preferências do Sistema

Capítulo 2

2.1) Ambiente de desenvolvimento

O ambiente de desenvolvimento iOS é composto pelos programas disponibilizado pela Apple em seu SDK. Após a instalação, as ferramentas estarão na pasta "Applications" do diretório onde o programa foi instalado (a pasta padrão é /Developer).

A Versão atual do Xcode é a 4.2, e pode ser baixado gratuitamente na Mac App Store (dentro do portal de desenvolvimento, o link de download do Xcode é redirecionado para a Mac App Store).

Dentro do SDK existem ferramentas necessárias para o desenvolvimento, assim como outras que auxiliam e facilitam a correção de possíveis problemas encontrados durante o desenvolvimento.

2.2) Xcode

O Xcode é o ambiente padrão de desenvolvimento (IDE) dos aplicativos iOS. Ela é desenvolvida e mantida pela própria Apple e é o ponto de partida para o desenvolvimentos de aplicativos iOS. Ele possui correção de sintaxe, função de auto-completar trechos de código, marcação de trechos relevantes entre outras funcionalidades.

Ele possui também um editor de interface, utilizado para adicionar componentes a tela do aplicativo, fazer as ligações necessárias com os métodos definidos no código.

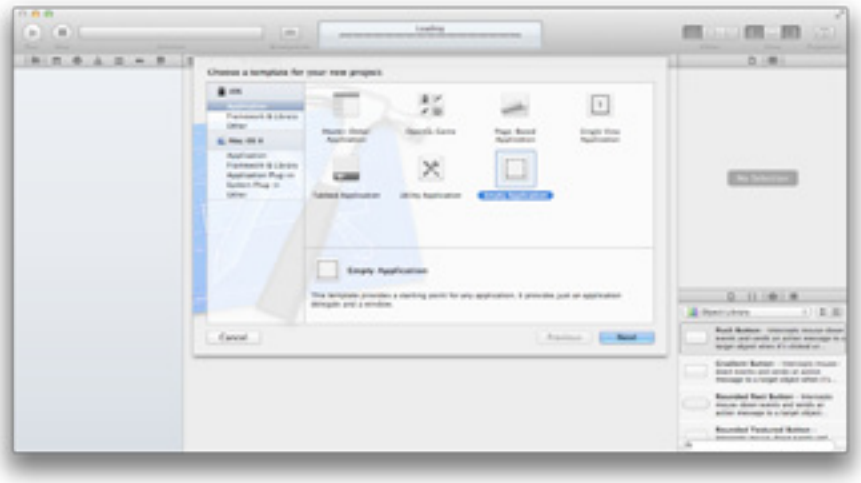


Figura 3.1: Tela de criação de novo projeto do Xcode

Capítulo 3

3.1) Interface do Xcode

Quando fazemos a instalação do Xcode, se não foi alterado seu local de instalação, ele se encontrará na pasta /Developer/Applications, caso contrário estará na pasta /CAMINHO_DA_INSTALACAO/Applications.

Ao abrirmos o Xcode, ele apresentará uma tela de boas vindas, onde encontramos alguns atalhos, como criar um novo projeto, abrir um projeto a partir de um repositório GIT ou SVN e uma lista dos últimos projetos acessados.

Figura 6.1: Tela de boas vindas do Xcode

3.2) Coluna de navegação

A Coluna de navegação mostra hierarquicamente os recursos de nossa aplicação ela é dividida para mostrar sete recursos, indicado por botões em sua barra superior. Sendo elas:

- Project: exibe hierarquicamente os arquivos, itens e produtos resultantes do nosso projeto.
- Symbol: exibe as classes, suas assinaturas de método e variáveis de instância.
- Search: nos permite fazer busca de expressões, arquivos ou palavras no escopo do projeto.
- Issue: exibe warnings e errors de nosso código, normalmente chamados após uma build de nossa aplicação.
- Debug: exibe as threads e as chamadas de sistema correspondentes em uma aplicação pausada, podendo exibir os conteúdos de endereços de memória relevantes.
- Breackpoint: exibe os breakpoints configurados em nossa aplicação.
- Log: exibe o registro dos logs do aplicativo.

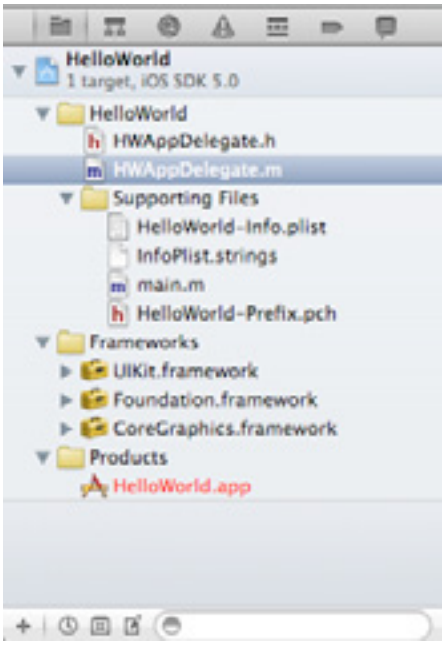


Figura 6.4: Coluna de navegação

3.3) Barra de ferramentas

A barra de ferramentas padrão do Xcode é composta dos seguintes controles:

- Controle de execução rápida: botão para executar e para a execução do aplicativo.
- Controle de seleção de esquema e dispositivos: seleciona em qual dispositivo a aplicação será executada e qual esquema deve ser utilizado na compilação. Podemos selecionar aqui se a aplicação rodará em um dispositivo ou no simulador e em qual versão do iOS será executada.
- Controle de execução de breakpoints: seleciona se os breakpoints configurados do sistema deverão ser acionados durante a execução do aplicativo.
- Display de status da aplicação: exibe o status de andamento das ações da aplicação, além do status da mesma.
- Seletor de editor: seleciona a forma de exibição do editor. Pode ser a padrão, com assistente (divide o editor em 2 partes e seleciona o complemento da classe selecionada), ou o editor de versão, que mostra as mudanças entre versões diferentes do arquivo trabalhado.
- Seletor de views: seleciona quais views laterais deverão ser exibidas além da janela do editor.
- Organizer: abre a janela do Organizer, onde podemos achar as configurações de dispositivos, licenças, perfis de desenvolvimento e repositórios.

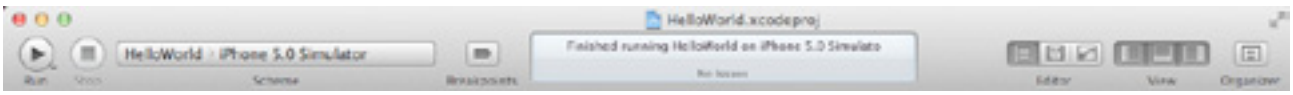


Figura 6.5: Barra de ferramentas

3.4) Coluna de inspectors

Nos exibe os inspetores, responsáveis por nos mostrar as propriedades do objeto selecionado na coluna de navegação. Por padrão, nos mostra o inspetor de arquivos e o inspetor de ajuda rápida, que nos dá informações sobre a classe selecionada. Quando trabalhando no Interface Builder, a coluna mostra também os atributos de componentes visuais, como seus tamanhos, cores, posição, comportamento para interação com o usuário.

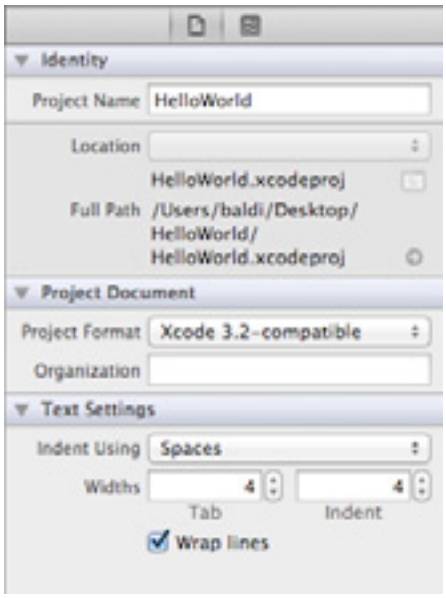


Figura 6.6: Coluna de inspectors

3.5) Área de debug

A área de debug nos mostra informações dos logs gerados pelo aplicativo, além de warnings e errors. Podemos também ver a seção de inspeção de variáveis.

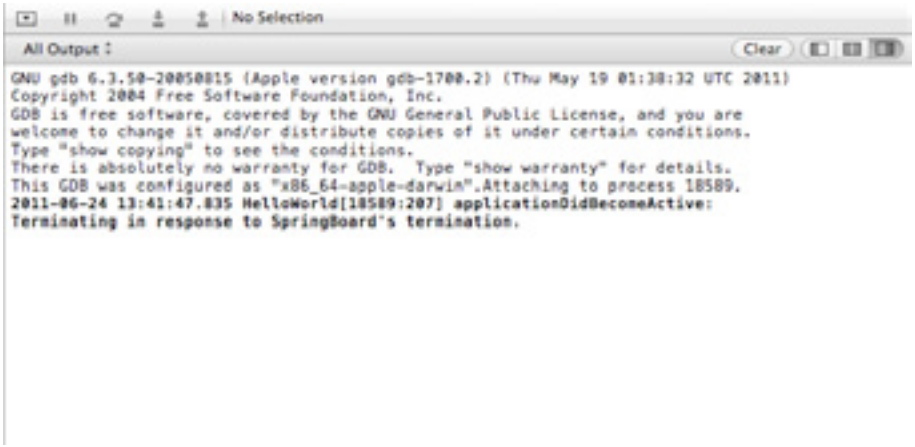


Figura 6.7: Área de debug

3.6) Coluna de bibliotecas

A coluna de biblioteca nos mostra as bibliotecas de objetos do sistema, e possui quatro bibliotecas padrão:

- templates de arquivos: possui templates de arquivos para adicionarmos ao nosso projeto.
- blocos de código: possui blocos de código prontos para utilizarmos em nossas classes.
- objetos: possui objetos visuais para adicionarmos nos arquivos de configuração do layout do aplicativo.
- mídia: possui os arquivos de mídia do projeto.

Figura 6.8: Coluna de bibliotecas

Capítulo 4

4.1) Orientação a Objetos

Orientação a Objetos, ou somente OO, é uma forma de implementarmos um software no qual seus diferentes componentes são definidos em Classes e utilizados através de Objetos.

4.2) Classes

Uma classe é, por definição, o protótipo para um tipo de objeto. Ela define o comportamento que o objeto terá, as mensagens que ele saberá responder seus atributos e seu estado. Por exemplo, uma classe chamada Ventilador terá atributos como "cor", "tamanho", "material", "velocidades", e os métodos "ligar", "desligar" e "trocar de velocidade", além dos estados "ligado" e "desligado". Por convenção, os nomes das classes são iniciados com letra maiúscula.

4.3) Objetos ou Instâncias

As classes guardam somente a definição de um componente, é o "conceito" de Ventilador. Para utilizar esse conceito, precisamos criar objetos ou instâncias dessa classe, o qual vai ter valores para suas propriedades e interagir com os outros objetos do sistema. Portanto, podemos ter um ventilador azul, pequeno, de plástico cuja velocidade que varia de 1 a 3. Ou seja, essa seria uma instância da classe Ventilador. Podemos, com a mesma classe, ter outra instância igual a anterior, mas da cor verde. Reparem que temos duas instâncias diferentes, entretanto a classe é a mesma. Por convenção, os objetos (instâncias) tem seus nomes iniciados por letras minúsculas.

4.4) Herança e Polimorfismo

A classe Ventilador é bastante abrangente e, por isso, podemos criar novas classes a partir dela, como a classe "Ventilador de Teto". Todo ventilador de teto é um ventilador, o que implica que ele tem todos as características que um ventilador possui, com as exceções de que ele deve ser fixado no teto e que ele pode rodar no sentido horário como no sentido anti-horário.

Essa característica é chamada de Herança, no qual temos uma classe Pai (Ventilador) e a classe Filho (Ventilador de Teto), que é uma especialização da classe Pai. Ele recebe, por herança, todas as características e estados da classe Pai (cor, tamanho, velocidades, etc) além de responder às mesmas mensagens.

É possível também que diferentes tipos de ventiladores respondam de forma diferente ao mesmo tipo de ação, por exemplo, aumentar de forma gradual sua velocidade, ao invés de ter apenas velocidades pré-definidas. Ou seja, ambos as classes respondem à mensagem "trocar de velocidade" mas de forma diferente. Esse conceito é chamado de Polimorfismo e é essencial para a OO.

Capítulo 5

5.1) Introdução ao Objective-C

O Objective-C foi criado por Brad Cox e Tom Love no início dos anos 80, implementando funcionalidades da linguagem Smalltalk em um pre processador de C. O objetivo a princípio era adicionar uma reutilização real de código e orientação a objetos na linguagem C, sem perder sua compatibilidade com os códigos já existentes.

Mas a grande popularização da linguagem se deu quando Steve Jobs, que havia fundado a NeXT, licenciou o Objective-C, e lançou o compilador e as bibliotecas que foram a base do sistema NeXTStep que, apesar de não ter tido muita influência como sistema operacional, teve suas ferramentas adotadas por um grande número de companhias, popularizando a linguagem.

Com a aquisição da NeXT, a Apple usou suas bibliotecas e linguagem para a elaboração do seu novo sistema operacional, o Mac OS X.

Em suma, o Objective-C é um superset da linguagem C, adicionando algumas funcionalidades à linguagem. Toda a programação procedural do Objective-C segue a sintaxe do C, o que permite que programas inteiros sejam criados nessa linguagem. Já a implementação da orientação a objeto, segue o padrão Smalltalk de envio de mensagens.

5.2) Objetos

Imaginemos por exemplo um objeto ponto. Um ponto é composto por duas coordenadas inteiras x e y. Um objeto quadrado, que possui quatro objetos ponto, só pode ler os valores de x e y de um dado ponto, através de métodos de acesso (getters), e mudá-los através dos métodos de alteração (setters). Por padrão, os métodos setters são nomeados com "setNome-

DaVariavel" com a primeira letra do nome da variável maiúscula, como todas as variáveis de instância são privadas, os getters têm o mesmo nome da variável. Por exemplo, uma variável de instância x teria como getter um método chamado x, e como setter um método chamado setX.

5.3) Mensagens

Em Objective-C, para que um objeto inicie alguma ação, é enviado a ele uma mensagem para que ele execute um método. As mensagens no Objective-C são definidas com chaves, por exemplo: `[ponto x];`.

No nosso exemplo, ponto é o objeto que se deseja que execute uma ação, e x é o nome do método que ele deverá executar. Quando uma mensagem é enviada a um objeto, o método correspondente é selecionado de seu repertório e executado. A mensagem é seguida pelo delimitador de comando ";" como em toda linha de comando C.

Uma mensagem também pode levar parâmetros. Estes são indicados com o separador ":". Um exemplo de mensagem com parâmetro seria: `[ponto setX:4];`. Essa linha manda uma mensagem ao objeto ponto, para que o método setX seja executado, com o valor inteiro 4. Para métodos que recebem múltiplos argumentos, o exemplo seria da seguinte maneira: `[ponto setX:4 y:10];`. As mensagens trazem ainda, o retorno dos métodos que ela chamam, por exemplo: `int valor = [ponto x];`. Nesse caso, criamos uma variável do tipo primitivo inteiro, que recebe o retorno do método x do objeto ponto.

A partir da versão 2.0, foi implementada uma nova forma de utilizar os métodos de acesso (getters e setters) das variáveis de instância, através do caractere ".". Logo, a expressão

`ponto.x = 10;` é exatamente igual à expressão `[ponto setX:10];`, desde que seus métodos de acesso estejam declarados.

Podemos comparar a sintaxe de Objective-C com a das demais linguagens, como C, Java e Pascal:

C/Java/Pascal/...

Declaração:

```
    void meuMetodo(void) {...}

void meuMetodo(int x) {...}

void potencia(int base, int exp) {...}
```

Chamada:

```
meuMetodo();

meuMetodo(4);

potencia(4,5);
```

Objective-C:

Declaração:

```
-(void)meuMetodo {...}
```

```
-(void)meuMetodo:(int)x {...}

        -(void)potenciaBase:(int)base expoente:(int)exp {...}
```

Chamada:

```
[self meuMetodo];

[self meuMetodo:4];

        [self potenciaBase:4 expoente:5];
```

5.4) Métodos

Os métodos definem quais mensagens nosso objeto é capaz de receber, o que pode ser encarado também como ações efetuadas por ele. Eles consistem de duas partes: a assinatura (ou nome de chamada) e a implementação, cada uma em seu devido lugar na criação das classes.

A assinatura de métodos no Objective-C é dividida em 4 partes, que podem ser visualizadas no exemplo abaixo:

```
-(void)setX:(int)newX y:(int)newY;
```

A primeira parte da assinatura do método é o indicador de tipo, representados pelos sinais + e - eles definem se o método é um método de classe (+) ou de instância (-). A segunda parte é o indicador de tipo de retorno (no exemplo o método retorna "void", mas poderia ser qualquer tipo de objeto). Em seguida temos o nome do método (setX: y:) e finalizando temos

os tipos dos argumentos e seus respectivos nomes ((int)newX e (int)newY). Portanto esse método recebe dois inteiros como parâmetros e não retorna nada.

Normalmente as assinaturas (ou definições) de método, são declaradas no arquivo de cabeçalho (.h) ou dentro de um bloco de interface. Por definição, no Objective-C, não temos implementação de métodos privados. Como dito anteriormente, os métodos podem ser de dois tipos:

- Método de Classe: são chamados diretamente da classe (possuem o indicador + em sua assinatura) e são análogos aos métodos estáticos de outras linguagens de programação. Exemplo: `[Ponto pontoZero];`.
- Método de Instância: são chamados de um objeto da classe (possuem o indicador - em sua assinatura). Exemplo: `[ponto setX:4 y:10];`.

5.5) Classes

As definições de classes no Objective-C são aditivas, ou seja, cada classe que criada é baseada em outra classe, de quem ela herda seus métodos e atributos. A nova classe simplesmente especifica o comportamento da classe pai, adicionando novos métodos e/ou variáveis ou reescrevendo os já existentes. A herança liga todas as classes em uma árvore de hierarquia, com uma única classe como raiz. A criação de uma nova classe é dividida em duas partes:

- Arquivo de Interface (.h): arquivo onde é definido os valores da classe, como sua superclasse, seus métodos e atributos.
- Arquivo de Implementação (.m): arquivo onde é implementado o comportamento da classe

através de seus métodos.

Voltando ao exemplo do ponto, a definição de sua interface estaria no arquivo Ponto.h e sua implementação no arquivo Ponto.m:

Ponto.h

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

@interface Ponto : NSObject

@property (nonatomic) int x;

@property (nonatomic) int y;

+(Ponto *)pontoZero;

-(void)setX:(int)newX y:(int)newY;

      @end
```

Ponto.m

```
#import “Ponto.h”

@implementation Ponto

@synthesize x, y;

+(Ponto *)pontoZero {

    Ponto *p = [[Ponto alloc] init];
```

```
    [p setX:0 y:0];
    return p;
}

-(void)setX:(int)newX y:(int)newY {
    self.x = newX;
    self.y = newY;
}

    @end
```

Capítulo 6

6.1) Foundation Framework

Podemos dizer que um framework é um atalho para os desenvolvedores de software pois ele é um conjunto de classes utilizado para solucionar um determinado problema comum ou então para guiar o desenvolvedor na sua resolução. O Foundation Framework é baseado na segunda hipótese citada acima, pois seu principal objetivo é definir um conjunto básico de classes utilitárias ao Objective-C.

6.2) NSObject

NSObject é a classe raiz da maioria das classes do Objective-C e por isso fornece uma interface básica para as classes que a estendem. Para criarmos um objeto do tipo NSObject primeiros alocamos a memória necessária para em seguida inicializá-lo e para isso utilizamos dois métodos diferentes:

- alloc: cria uma nova instância da classe, mas que ainda não pode ser usada.
- init: inicializa um novo objeto criado pelo método alloc. Subclasses do NSObject devem implementar esse método.

Da mesma forma que alocamos a memória necessária para o objeto, precisamos liberá-la quando não vamos mais utilizá-la e outros objetos a utilizem. Para isso utilizamos o seguinte método:

- release: decrementa o contador de referências do objeto.
- autorelease: retorna um objeto que automaticamente chamará o release quando ele não for

mais utilizado. Recomendado apenas em casos em que não exista outra alternativa.

- dealloc: é chamado automaticamente quando a quantidade de referências do objeto chegar a zero. Ele vai liberar a memória utilizada pelo objeto e nunca deve ser chamado diretamente.

Outros métodos importantes do NSObject:

- class: retorna o objeto do tipo Class.
- description: retorna uma string que descreve o objeto. As subclasses de NSObject podem sobrescrever esse método caso seja necessário imprimir informações relevantes do objeto.

Capítulo 7

7.1) NSString

A classe NSString estende a classe NSObject e define um array de caracteres, ou seja, uma string. Pode parecer um pouco semelhante com as strings de C, mas como ela é um objeto (em C ela classe string é um tipo literal), ela pode receber mensagens. Isso significa que a classe NSString possui métodos, e com eles é possível concatenar, dividir, buscar por substrings, comparar, transformar, entre outros.

A forma mais simples de se criar um objeto do tipo NSString é utilizando o construtor do Objective-C @"", por exemplo `NSString *umaString = @"Primeiro exemplo de string";`. Para enviarmos uma mensagem para o objeto, fazemos da seguinte forma:

```
[umaString length];
```

Como toda instância da classe NSString é um objeto, podemos enviar uma mensagem diretamente para o construtor:

```
[@"Primeiro exemplo de string" length];
```

Os métodos mais comuns para inicializarmos uma objeto do tipo NSString são os seguintes:

- initWithString: retorna uma string com o mesmo conteúdo da string passada como parâmetro.
- initWithFormat: retorna uma string com o conteúdo do formato de string passado como parâmetro.
- string: retorna uma string (autorelase) vazia (tem o mesmo resultado ao inicializarmos com @"").

- stringWithString: retorna uma string (autorelase) com o mesmo conteúdo da string passada como parâmetro.
- stringWithFormat: retorna uma string (autorelase) com o conteúdo do formato de string passado como parâmetro.

Formatos de string são strings que podem receber parâmetros através de diretivas, com o intuito de criamos strings dinâmicas. Existem três tipos mais comuns de diretivas:

- %@: imprime objetos do Objective-C. Implicitamente é chamado o método description do objeto.
- %i: imprime números inteiros.
- %f: imprime números fracionários. Nesse caso é possível definir o número de casas decimais a serem impressas. Por exemplo, se utilizarmos a diretiva alterada para %.3f, vai fazer com que a parte fracionária do número seja arredondada para exibir somente três casas decimais.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSString *s1 = @"";

NSString *s2 = @"Uma string com conteúdo";

NSString *s3 = [[NSString alloc] init];
```

```
NSString *s4 = [[NSString alloc] initWithString:@"Outra string com conteúdo"];

NSString *s5 = [[NSString alloc] initWithString:s2];

NSString *s6 = [[NSString alloc] initWithFormat:@"Conteúdo da string s4: %@", s4];

NSString *s7 = [NSString string];

NSString *s8 = [NSString stringWithString:s5];

      NSString *s9 = [NSString stringWithFormat:@"Primeira string: %@ e segunda
string: %@", s2, s6];
```

Abaixo temos alguns métodos para lidarmos com o conteúdo da string:

- length: retorna o número de caracteres da string.
- stringByAppendingFormat: retorna uma nova string, concatenando a string que recebeu a mensagem com o formato de string passado como parâmetro.
- substringFromIndex: retorna uma nova string com o conteúdo da string que recebeu a mensagem a partir do índice passado como parâmetro.
- substringToIndex: retorna uma nova string com o conteúdo da string que recebeu a mensagem até o índice passado como parâmetro.
- componentsSeparatedByString: retorna um array de substrings da string que recebeu a mensagem separados pela string passada como parâmetro.
- stringByReplacingOccurrencesOfString:withString: retorna uma nova string com substituindo todas as ocorrências da string passada no primeiro parâmetro na string que recebeu a mensagem pela string passada no segundo parâmetro.

- capitalizedString: retorna uma nova string com a primeira letra de cada palavra em maiúsculo e as outras em minúsculo.
- lowercaseString: retorna uma nova string com todas as letras da string que recebeu a mensagem em minúsculo.
- uppercaseString: retorna uma nova string com todas as letras da string que recebeu a mensagem em maiúsculo.
- description: retorna o conteúdo da string.
- intValue: retorna a representação da string em inteiro.
- doubleValue: retorna a representação da string em double (com ponto flutuante).
- boolValue: retorna a representação da string em booleano.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSString *s1 = @"A primeira string";

NSString *s2 = @"A segunda string";

int l = [s1 length];

// 17

NSString *s3 = [s1 stringByAppendingFormat:@" e %@", s2];

// @"A primeira string e A segunda string"

NSString *s4 = [s1 substringFromIndex:5];
```

```
// @”meira string”

NSString *s5 = [s1 substringToIndex:5];

// @”A pri”

NSArray *array = [s1 componentsSeparatedByString:@” “];

// [@”A”, @”primeira”, @”string”]

NSString *s6 = [s1 stringByReplacingOccurrencesOfString:@”r” withString:@”R”];

// @”A pRimeiRa stRing”

NSString *s7 = [s6 capitalizedString];

// @”A Primeira String”

NSString *s8 = [s6 lowercaseString];

// @”a primeira string”

NSString *s9 = [s6 uppercaseString];

// @”A PRIMEIRA STRING”

NSString *s10 = [s9 description];

// @”A PRIMEIRA STRING”

int i = [@”14” intValue];

// 14

double d = [@”3.1415” doubleValue];
```

```
// 3.1415

BOOL b = [@"YES" boolValue];

      // YES
```

7.2) NSLog()

A Apple possui um sistema de logs chamado Apple System Logger (ASL), no qual mensagens são registradas. No Objective-C o método NSLog(), que recebe um formato de string como parâmetro, adiciona um registro a ele. O console da IDE vai exibir os logs registrados pelo aplicativo que está executando:

```
NSLog(@"Exemplo simples de log.");

Exemplo simples de log.
```

7.3) NSMutableString

Os objetos do tipo NSString são imutáveis, ou seja, depois de criados não podem ser alterados, podem somente ser realocados. Para criarmos uma stirng que seja mutável utilizamos a classe NSMutableString. Os métodos da classe NSMutableString são muito parecidos com os da classe NSStirng, com a diferença que o conteúdo gerado pelos métodos são atribuídos à própria string que recebe a mensagem, ao invés de retornar uma nova string.

Os métodos mais comuns são:

- appendFormat: concatena um formato de stirng à string que recebeu a mensagem.
- appendString: concatena uma string à string que recebeu a mensagem.
- insertString:atIndex: insere a string passada como parâmetro na posição determinada na string que recebeu a mensagem.
- setString: altera o valor da string que recebeu a mensagem pela conteúdo da string passada como parâmetro.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSMutableString *ms = [NSMutableString stringWithString:@"Uma string mutável!"];

[ms appendString:@" Que legal!"];

// @"Uma string mutável! Que legal!"

[ms insertString:@" super" atIndex:3];

// @"Uma super string mutável! Que legal!"

[ms setString:@"Mudou tudo agora..."];

      // @"Mudou tudo agora..."
```

Capítulo 8

8.1) NSArray

A classe NSArray guarda uma lista ordenada de objetos, chamada de array. Um objeto dessa classe pode conter qualquer objeto que derive da classe NSObject, seja direta ou indiretamente e não precisa ser de um tipo específico, pode conter qualquer objeto, desde que seja tratado corretamente durante a leitura. Um objeto da classe NSArray é imutável e deve ser construído no momento da sua inicialização.

Para isso, temos os seguintes métodos:

- initWithArray: retorna um array com os mesmos objetos do array passado como parâmetro.
- initWithObjects: retorna um array com os objetos passados como parâmetro, terminado com nil.
- array: retorna um array (autorealease) vazio.
- arrayWithArray: retorna um array (autorealease) com os mesmos objetos do array passado como parâmetro.
- arrayWithObject: retorna um array (autorealease) com o objeto passado como parâmetro.
- arrayWithObjects: retorna um array (autorealease) com os objetos passados como parâmetro, terminado com nil.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSArray *a1 = [[NSArray alloc] initWithObjects:@"objeto 1", @"objeto 2", nil];

NSArray *a2 = [[NSArray alloc] initWithArray:a1];

NSArray *a3 = [NSArray array];

NSArray *a4 = [NSArray arrayWithArray:a2];

NSArray *a5 = [NSArray arrayWithObject:@"objeto"];

    NSArray *a6 = [NSArray arrayWithObjects:@"objeto 1", @"objeto 2", nil];
```

Nos exemplos que criamos os arrays a1 e a6, precisamos terminar a lista de objetos passados como parâmetro com nil para informar quando termina a lista. Importante ressaltar que tipo primitivos (int, float, double, BOOL, etc) não podem ser guardados no NSArray pois eles não são derivados da classe NSObject.

Abaixo temos alguns métodos para lidarmos com o conteúdo do array:

- count: retorna o número de objetos do array.
- containsObject: verifica se o array contém o objeto passado como parâmetro.
- objectAtIndex: retorna o objeto do índice passado como parâmetro.
- lastObject: retorna o último objeto do array.
- indexOfObject: retorna o índice do objeto no array.
- description: retorna uma string com a representação dos objetos do array.

- componentsJoinedByString: agrupa os objetos do array em um string, separados pela string passada como parâmetro.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSArray *a1 = [NSArray arrayWithObjects:@"a", @"b", @"c", @"d", nil];

int c = [a1 count];

// 4

BOOL b = [a1 containsObject:@"e"];

// NO

NSString *s1 = [a1 objectAtIndex:1];

// @"b"

NSString *s2 = [a1 lastObject];

// @"d"

int i = [a1 indexOfObject:@"c"];

      // 3

NSString *s3 = [a1 description];

// @"(a, b, c, d)"

NSString *s4 = [a1 componentsJoinedByString:@" "];

      // @"a b c d"
```

8.2) NSMutableArray

A classe NSArray tamém possui sua versão mutável chamada NSMutableArray. Essa classe permite que, depois de criado o array, seja inserido algum objeto, removido ou que ele seja reordenado. Para isso temos os seguintes métodos:

- addObject: adiciona o objeto passado como parâmetro no final do array.
- insertObject:atIndex: adiciona o objeto passado como parâmetro na posição indicada.
- removeAllObjects: remove todos os objetos do array.
- removeObjectAtIndex: remove o objeto do array na posição passada como parâmetro.
- removeObject: remove o objeto passado como parâmetro do array.
- replaceObjectAtIndex:withObject: troca o objeto da posição indicada pelo objeto passado como parâmetro.
- exchangeObjectAtIndex:withObjectAtIndex: troca os objetos das posições passadas com parâmetro.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSMutableArray *ma = [NSMutableArray arrayWithObjects:@"a", @"b", @"c", @"d",
nil];

[ma addObject:@"e"];

// (@"a", @"b", @"c", @"d", @"e")

[ma insertObject:@"f" atIndex:2];
```

```
// (@"a", @"b", @"f", @"c", @"d", @"e")

[ma removeObjectAtIndex:3];

// (@"a", @"b", @"f", @"d", @"e")

[ma removeObject:@"f"];

// (@"a", @"b", @"d", @"e")

[ma replaceObjectAtIndex:3 withObject:@"c"];

// (@"a", @"b", @"d", @"c")

[ma exchangeObjectAtIndex:2 withObjectAtIndex:3];

// (@"a", @"b", @"c", @"d")

[ma removeAllObjects];

      // ()
```

Capítulo 9

9.1) NSDictionary

Um objeto do tipo NSDictionary (ou dicionário) contém associações de pares chave-valor. Cada par desses possui um objeto que representa a chave (geralmente do tipo NSString) e o outro objeto, derivado de NSObject, que é o valor recuperado por essa chave. Toda chave do dicionário deve ser única, ou seja, não pode existir dois pares que possuam a mesma chave para objetos diferentes. Todo objeto da classe NSDictionary é imutável portanto deve ter todos as suas chaves e valores atribuídos no momento de sua inicialização.

Para isso, temo os seguintes métodos:

- initWithDictionary: retorna um dicionário com os mesmos pares de chave-valor que o dicionário passado como parâmetro.
- initWithObjects:forKeys: retorna um dicionário com os pares chave-valor criados com os objetos dos arrays passados como parâmetro.
- initWithObjectsAndKeys: retorna um dicionário com os pares chave-valor definidos pelos objetos passados como parâmetro (primeiro objeto e depois chave), terminado com nil.
- dictionary: retorna um dicionário vazio.
- dictionaryWithDictionary: retorna um dicionário (autorelease) com os mesmos pares de chave-valor que o dicionário passado como parâmetro.
- dictionaryWithObject:forKey: retorna um dicionário (autorelease) com o par chave-valor criado com os objetos passados como parâmetro.
- dictionaryWithObjects:forKeys: retorna um dicionário (autorelease) com os pares chave-

valor criados com os objetos dos arrays passados como parâmetro.

- dictionaryWithObjectsAndKeys: retorna um dicionário (autorelease) com os pares chave-valor definidos pelos objetos passados como parâmetro (primeiro objeto e depois chave), terminado com nil.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSArray *objetos = [NSArray arrayWithObjects:@"objeto1", @"objeto2", nil];

NSArray *chaves = [NSArray arrayWithObjects:@"chave1", @"chave2", nil];

NSDictionary *d1 = [[NSDictionary alloc] initWithObjects:objetos forKeys:chaves];

NSDictionary *d2 = [[NSDictionary alloc] initWithDictionary:d1];

NSDictionary *d3 = [[NSDictionary alloc] initWithObjectsAndKeys:@"objeto1",
@"chave1", @"objeto2", @"chave2", nil];

NSDictionary *d4 = [NSDictionary dictionary];

NSDictionary *d5 = [NSDictionary dictionaryWithDictionary:d5];

NSDictionary *d6 = [NSDictionary dictionaryWithObject:@"objeto1" forKey:@"chave1"];

NSDictionary *d7 = [NSDictionary dictionaryWithObjects:objetos forKeys:chaves];

      NSDictionary *d8 = [NSDictionary dictionaryWithObjectsAndKeys: @"objeto1",
@"chave1", @"objeto2", @"chave2", nil];
```

Assim como acontece na criação de um array, nos exemplos dos dicionários d3 e d8 precisamos terminar a lista de objetos e chaves passados como parâmetro com nil para infor-

mar quando termina a lista. Importante ressaltar que tipo primitivos (int, float, double, BOOL, etc) não podem ser guardados no dicionário e nem utilizados como chave pois eles não são derivados da classe NSObject.

Abaixo temos alguns métodos para lidarmos com o conteúdo do dicionário:

- count: retorna o número de pares chave-valor do dicionário.
- allKeys: retorna um array com todas as chaves do dicionário.
- allValues: retorna um array com todos os objetos do dicionário.
- objectForKey: retorna o objeto que está associado com a chave passada como parâmetro.
- description: retorna uma string com a representação dos pares chave-valor do dicionário.

Importante notar que a ordem em que as chaves estão dispostas no array de todas as chaves não possui uma ordem. O mesmo acontece com o array de todos os objetos. Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSDictionary *d1 = [NSDictionary dictionaryWithObjectsAndKeys: @"objeto1",
@"chave1", @"objeto2", @"chave2", nil];

int c = [d1 count];

// 2

NSArray *a1 = [d1 allKeys];

// (@"chave1", @"chave2")
```

```
NSArray *a2 = [d1 allValues];

// (@"objeto1", @"objeto2")

NSString *s1 = [d1 objectForKey:@"chave1"];

// @"objeto1"

NSString *s2 = [d1 description];

// @"{chave1 = objeto1; chave2 = objeto2;}"
```

9.2) NSMutableDictionary

A classe mutável do NSDictionary é a NSMutableDictionary. Com ela é possível adicionar, alterar e remover os pares de chave-valor do dicionário. Para isso temos os seguintes métodos:

- setObject:forKey: adiciona um novo par chave-valor ao dicionário ou, caso a chave já exista, ele altera a associação da chave para o novo objeto passado como parâmetro.
- removeObjectForKey: remove o par chave-valor do dicionário cuja chave é igual ao parâmetro.
- removeAllObjects: remove todos os pares chave-valor do dicionário.

Abaixo seguem exemplos de como utilizamos os métodos acima:

```
NSMutableDictionary *md = [NSMutableDictionary dictionaryWithObjectsAndKeys:
@"objeto1", @"chave1", @"objeto2", @"chave2", nil];
```

```
[md setObject:@”novo objeto” forKey:@”chave1”];

// {@”chave1” = @”novo objeto”; @”chave2” = @”objeto2”;}

[md removeObjectForKey:@”chave2”];

// {@”chave1” = @”novo objeto”;}

[md setObject:@”objeto3” forKey:@”chave3”];

// {@”chave1” = @”novo objeto”; @”chave3” = @”objeto3”;}

[md removeAllObjects];

      // {}
```

Capítulo 10

10.1) NSBundle

A classe NSBundle guarda a localização de um pacote de arquivos no sistema de arquivos do aparelho, entretanto, nos aplicativos de iOS, só é possível acessar o próprio pacote. Com ele é possível localizar e carregar dinamicamente seus recursos. Para utilizarmos essa classe, não devemos instanciar um novo objeto, mas sim utilizarmos o objeto que já existe:

- mainBundle: retorna o pacote atual, de onde o aplicativo está sendo executado (geralmente o .app).

Com o objeto dessa classe, podemos fazer algumas tarefas, como carregar recursos e obter informações do pacote:

- bundlePath: retorna uma string com o caminho do pacote no sistema de arquivos do aparelho.
- pathForResource:ofType: retorna uma string com o caminho para o recurso passado como parâmetro (o nome do arquivo deve estar separado de sua extensão).

Supondo um projeto chamado "ExemploNSBundle", teríamos o seguinte resultado para os métodos acima:

```
NSBundle *bundle = [NSBundle mainBundle];

NSString *path = [bundle bundlePath];

// @"/Users/iai/Library/Application Support/iPhone Simulator/5.0/
Applications/7CC80812-947A-453A-B10D-6FC5BF7B3D8E/ExemploNSBundle.app"

NSString *resource = [bundle pathForResource:@"Parrot" ofType:@"tif"];
```

```
// @"/Users/iai/Library/Application Support/iPhone Simulator/5.0/
Applications/7CC80812-947A-453A-B10D-6FC5BF7B3D8E/ExemploNSBundle.app/Parrot.tif"
```

10.2) NSHomeDirectory()

O iOS é um sistema fechado, e por isso não é todo lugar que podemos alterar arquivos, deletar. Por exemplo, não conseguimos modificar os arquivos que estão dentro do pacote do aplicativo, eles são somente para leitura.

Para isso temos a função NSHomeDirectory(), que retorna uma string com o caminho da pasta que contém o pacote do aplicativo, ou seja, a pasta anterior à do pacote. Nessa pasta podemos escrever e ler arquivos, entretanto não é recomendado que façamos dessa maneira. O ideal é utilizarmos duas pastas recomendadas pela Apple: tmp, que é uma pasta temporária cujo conteúdo é apagado a cada três dias caso necessário, e a pasta Documents, que é uma pasta para salvar documentos que são acessados por toda execução do aplicativo.

Supondo o mesmo projeto anterior chamado "ExemploNSBundle", utilizaríamos a função da seguinte maneira:

```
NSString *home = NSHomeDirectory();

// @"/Users/baldi/Library/Application Support/iPhone Simulator/5.0/
Applications/7CC80812-947A-453A-B10D-6FC5BF7B3D8E"

NSString *documents = [home stringByAppendingString:@"/Documents"];

// @"/Users/baldi/Library/Application Support/iPhone Simulator/5.0/
Applications/7CC80812-947A-453A-B10D-6FC5BF7B3D8E/Documents"
```

```
NSString *tmp = [home stringByAppendingString:@"/tmp"];

      //    @"/Users/baldi/Library/Application    Support/iPhone    Simulator/5.0/
Applications/7CC80812-947A-453A-B10D-6FC5BF7B3D8E/tmp"
```

10.3) NSFileManager

A classe NSFileManager fornece um conjunto de métodos para executarmos operações relacionadas com o sistema de arquivos. Com ele somos capazes de localizar, criar, remover e mover arquivos e diretórios. Para utilizarmos o gerenciador de arquivos, temos duas maneiras:

- init: retorna uma nova instância da classe NSFileManager, geralmente utilizado quando for necessário um gerenciador específico para determinada situação.
- defaultManager: retorna o gerenciador de arquivos padrão do aparelho.

E para gerenciarmos de fato os arquivos e pastas, utilizamos os seguintes métodos:

- contentsOfDirectoryAtPath:error: retorna um array com o conteúdo da pasta passada como parâmetro.
- subpathsOfDirectoryAtPath:error: retorna um array com o caminho de todas os arquivos e pastas dentro da pasta passada como parâmetro, recursivamente. É preciso tomar cuidado ao usar esse método pois ele pode ser muito custoso dependendo da quantidade de subpastas.
- createDirectoryAtPath:withIntermediateDirectories:attributes:error: cria uma nova

pasta para o caminho passado como parâmetro. Os outros parâmetros indicam se devem ou não ser criadas as pastas intermediárias, caso elas não existam, e seus atributos (consultar a documentação da classe NSFileManager para a lista de atributos válidos). Importante notar as permissões de leitura e escrita dos itens envolvidos.

- createFileAtPath:contents:attributes: cria um arquivo no caminho passado como parâmetro, junto de seus conteúdo e atributos. Importante notar as permissões de leitura e escrita dos itens envolvidos.

- copyItemAtPath:toPath:error: copia um arquivo de um caminha para outro, ambos passados como parâmetro nessa ordem. Importante notar as permissões de leitura e escrita dos itens envolvidos.

- moveItemAtPath:toPath:error: move um arquivo de um caminha para outro, ambos passados como parâmetro nessa ordem. Importante notar as permissões de leitura e escrita dos itens envolvidos.

- fileExistsAtPath: verifica se existe um arquivo ou pasta no caminho passado como parâmetro.

- isReadableFileAtPath: verifica a permissão de leitura do arquivo ou pasta passado como parâmetro.

- isWritableFileAtPath: verifica a permissão de escrita do arquivo ou pasta passado como parâmetro.

- isDeletableFileAtPath: verifica a permissão para deleção do arquivo ou pasta passado como parâmetro.

Como podemos notar, alguns dos métodos acima recebem um parâmetro de erro. Quando chamamos esses métodos, devemos passar uma variável do tipo NSError como referência para que, no caso de erro, essa variável seja atualizada pelo erro que ocorreu. Por exemplo, se tentamos recuperar o conteúdo de uma pasta que não existe, o objeto será atualizado com um erro "The operation couldn't be completed. No such file or directory" junto de outras informações que possam ser importantes.

Supondo o mesmo projeto anterior chamado "ExemploNSBundle", utilizaríamos os métodos acima da seguinte maneira:

```
NSFileManager *fm = [NSFileManager defaultManager];

NSError *erro = nil;

[fm createDirectoryAtPath:[documents stringByAppendingFormat:@"/pasta1/pasta2/
pasta3/"] withIntermediateDirectories:YES attributes:nil error:&erro];

// cria as pastas corretamente

[fm createFileAtPath:[documents stringByAppendingFormat:@"/exemplo.txt"]
contents:[@"texto para o arquivo" dataUsingEncoding:NSUTF8StringEncoding]
attributes:nil];

// cria o arquivo corretamente

erro = nil;

NSArray *a1 = [fm contentsOfDirectoryAtPath:[documents
stringByAppendingFormat:@"/pasta/"] error:&erro];
```

```
// erro: “The operation couldn’t be completed. No such file or directory”

erro = nil;

NSArray *a2 = [fm contentsOfDirectoryAtPath:documents error:&erro];

// (@”exemplo.txt”, @”pasta1”)

erro = nil;

NSArray *a3 = [fm subpathsOfDirectoryAtPath:documents error:&erro];

// (@”exemplo.txt”, @”pasta1”, @”pasta1/pasta2”, @”pasta1/pasta2/pasta3”)

erro = nil;

[fm copyItemAtPath:[documents stringByAppendingFormat:@”/exemplo.txt”]
toPath:[documents stringByAppendingFormat:@”/pasta1/pasta2/exemplo.txt”]
error:&erro];

// copia corretamente

erro = nil;

[fm moveItemAtPath:[documents stringByAppendingFormat:@”/pasta1/pasta2/exemplo.
txt”] toPath:[documents stringByAppendingFormat:@”/pasta1/exemplo.txt”]
error:&erro];

// move corretamente

[fm fileExistsAtPath:[documents stringByAppendingFormat:@”/pasta1/pasta2/exemplo.
txt”]];
```

```
// não existe esse arquivo

[fm isReadableFileAtPath:[documents stringByAppendingFormat:@"/exemplo.txt"]];

// o arquivo pode ser lido

[fm isWritableFileAtPath:[documents stringByAppendingFormat:@"/exemplo.txt"]];

// o arquivo pode ser editado

[fm isDeletableFileAtPath:[documents stringByAppendingFormat:@"/exemplo.txt"]];

      // o arquivo pode ser deletado
```

Capítulo 11

11.1) @protocol

Um protocolo define um conjunto de mensagens que um objeto pode receber, independente da classe do objeto, ou seja, elas podem ser implementadas por qualquer classe que implemente esse protocolo. Os protocolos são usados quando queremos definir um comportamento para uma classe e esperamos que ela saiba responder a elas.

Vamos imaginar uma classe Pessoa e duas classes que estendem Pessoa: Homem e Advogado. Agora vamos imaginar o protocolo para churrasqueiro. Quem sabe fazer churrasco precisa "acender a churrasqueira", "temperar a carne" e "tirar a carne no ponto". Dessa forma, é possível que tanto o engenheiro quanto o advogado possam fazer churrasco sem ter a necessidade de criar uma nova classe Churrasqueiro ou Engenheiro-Churrasqueiro.

Um protocolo não implementa nenhum método, apenas os definem. Portanto, uma definição de protocolo seria:

```
@protocol Churrasqueiro <NSObject>

-(void)acenderChurrasqueira;

-(void)temperarACarne;

-(void)tirarACarneNoPonto;

    @end
```

Por padrão, todos os métodos do protocolo são obrigatórios, ou seja, toda classe que implemente o protocolo deve implementá-los. Entretanto, podemos definir métodos que são opcionais através da diretiva @optional. Por exemplo, não é necessário saber "fazer costela no

bafo" para ser churrasqueiro. Portanto, poderíamos atualizar o protocolo para o seguinte:

```
@protocol Churrasqueiro <NSObject>

-(void)acenderChurrasqueira;

-(void)temperarACarne;

-(void)tirarACarneNoPonto;

@optional

-(void)fazerCostelaNoBafo;

    @end
```

Todos os métodos declarados abaixo da diretiva @optional serão opcionais para a implementação, a não ser que exista uma diretiva @required, que implica que os métodos abaixo dela serão obrigatórios. Para fazer que uma classe implemente um protocolo, fazemos da seguinte maneira:

```
@interface Pessoa : NSObject <Churrasqueiro>
```

Ou seja, a classe Pessoa, que estende a classe NSObject, implementa o protocolo Churrasqueiro. Poderíamos definir outros protocolos para uma mesma classe, por exemplo o protocolo Origami:

```
@interface Pessoa : NSObject <Churrasqueiro, Origami>
```

Quando adicionamos o protocolo para uma classe, não precisamos definir seus métodos no arquivo de interface da classe, apenas implementamos seus métodos no arquivo de imple-

mentação. Portanto o arquivo Pessoa.h ficaria assim:

```
@interface Pessoa : NSObject <Churrasqueiro, Origami>

      @end
```

E o arquivo Pessoa.m ficaria assim:

```
@implementation Pessoa

#pragma mark - Churrasqueiro methods

-(void)acenderChurrasqueira {

    // ...

}

-(void)temperarACarne {

    // ...

}

-(void)tirarACarneNoPonto {

    // ...

}

-(void)fazerCostelaNoBafo {

    // ...

}
```

```
#pragma mark - Origami methods

-(void)fazerOrigami {

    // ...

}

    @end
```

Assim como os métodos e atributos, quando estendemos uma classe, a classe filha implementa também os protocolos que a classe pai implementa. E em determinadas ocasiões precisaremos saber se a classe do objeto que estamos trabalhando implementa um protocolo, seja por herança ou não. Para isso existe o seguinte método da classe NSObjetct:

- conformsToProtocol: retorna sim caso a classe implemente o protocolo passado como parâmetro.

Para recuperarmos um objeto do tipo Protocol passado como parâmetro acima, utilizamos a mesma diretiva que é utilizada para declarar o protocolo, entretanto passamos o nome do protocolo entre parênteses da seguinte forma:

```
@protocol(Churrasqueiro);
```

Ou seja, para sabermos se a classe implementa ou não o protocolo, fazemos assim:

```
BOOL b = [Pessoa conformsToProtocol:@protocol(Churrasqueiro)];

        // YES
```

11.2) SEL e @selector

O selector (ou SEL) é um id único gerado pelo compilador para cada método do aplicativo. Entretanto, métodos com o mesmo nome possuem o mesmo id, o que é essencial para o polimorfismo, permitindo que uma mesma mensagem seja enviada para objetos de diferentes classes. Ou seja, o selector é utilizado, por exemplo, para passar a assinatura de um método como parâmetro para uma função.

Para recuperar o selector de um método, utilizamos a diretiva @seletor() e passamos o nome do método como parâmetro. Dessa forma podemos verificar se um objeto de uma classe responde a uma chamada, e para isso utilizamos o método abaixo:

- respondsToSelector: retorna sim caso o objeto implemente o selector passado como parâmetro ou caso ele responda a esse selector por herança.

Suponha o exemplo das classes Pessoa, Engenheiro e Advogado. A pessoa implementa o método "fazer contas de matemática", assim como o engenheiro, mas de uma forma muito mais trabalhada e confiável. Se fizermos o seguinte:

```
BOOL b1 = [pessoa respondsToSelector:@selector(fazerContasDeMatematica)];

// YES

BOOL b2 = [engenheiro respondsToSelector:@selector(fazerContasDeMatematica)];

// YES

BOOL b3 = [advogado respondsToSelector:@selector(fazerContasDeMatematica)];

      // YES
```

O objeto da classe Pessoa implementa o método acima, o objeto da classe Engenheiro reimplementa o método da classe pessoa e o objeto da classe Advogado responde a esse método por herança.