

# Orientação a Objetos – Criação de Classes

---

Programação 2 – Aulas 5, 6, 7

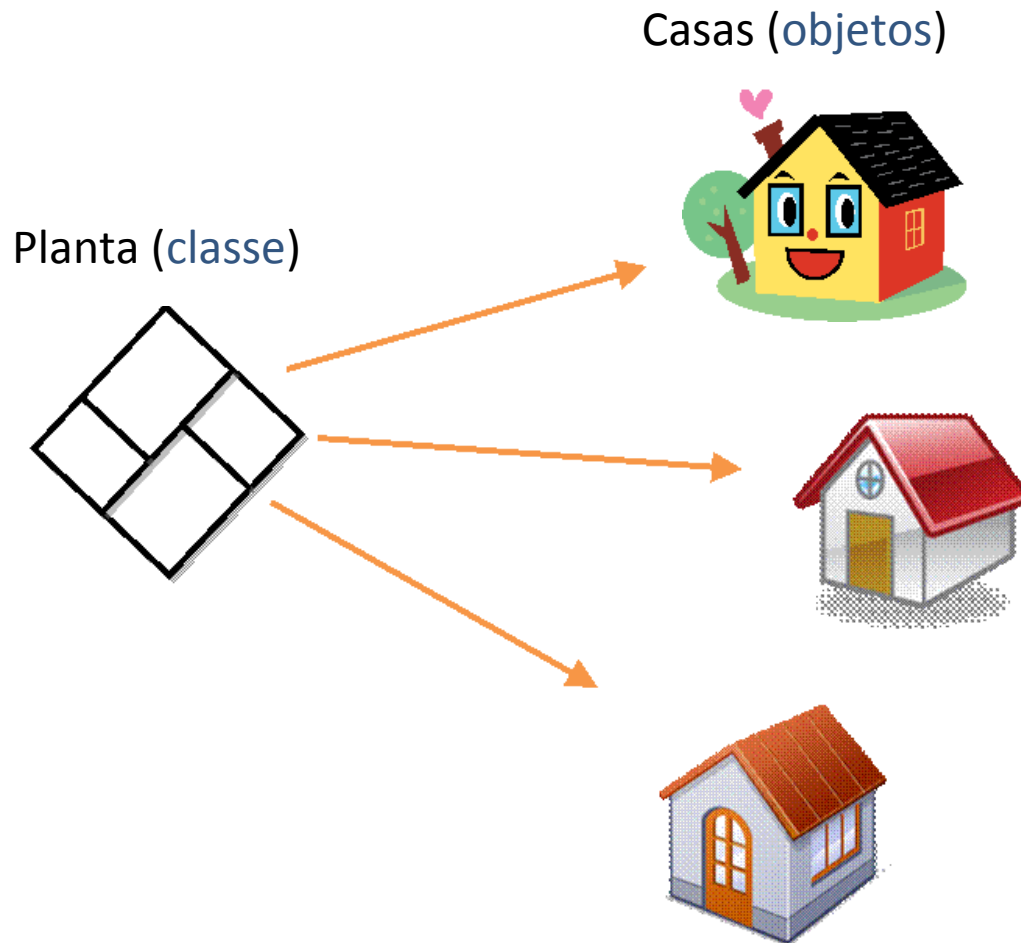
## Objetivos da seção

- Aprender a pensar em testes antes de definir novas classes
  - Apresentar *Test-Driven Development* (TDD)
- Aprender a definir novas classes
  - Entender conceitos de atributos, construtores, métodos, parâmetros e valor de retorno, encapsulamento, métodos *accessor* e *mutator*, *this*, métodos-função e métodos-procedimento, aninhamento de métodos, *this()*, escopo de atributos e variáveis locais, sobrecarga de métodos, métodos de classe, atributos de classe, escopo de atributos de classe, constantes

## Primeiramente uma introdução sobre Orientação a Objetos

- Pode-se pensar sobre o mundo real como uma coleção de objetos relacionados
- Exemplos de objetos do mundo bancário:
  - Contas correntes
  - Contas poupança
  - Clientes
  - Caixas
  - Agências
  - Cheques
  - Extratos
- Objetos podem ser agrupados em classes: Conta corrente, Conta de poupança, Cliente, Caixa, Agência, Cheque, Extratos

- Observe que existem várias Contas correntes de uma mesma classe "Conta corrente". A diferença entre classe e objeto:
  - "Classe" é um gabarito (como a planta de uma casa)
  - "Objeto" é a concretização do gabarito (casas feitas a partir da mesma planta)



- Objetos de uma certa classe têm **atributos**
  - Uma Conta tem um número, um saldo, um histórico de transações
  - Um Cliente tem um nome, um endereço

- Um Cheque tem um valor
- Objetos de uma mesma classe têm um mesmo **comportamento**
  - Clientes entram numa agência
  - Clientes fazem depósitos e saques
  - Clientes emitem cheques
- No mundo real, certos objetos não têm comportamento
  - Contas não são vivas: não "fazem" nada
- **Objetos podem estar relacionados**
  - Um cliente possui várias Contas
- **Podemos usar objetos ao fazer software também!**
- Há várias vantagens de fazer isso
  - É um pouco difícil entender todas as vantagens de OO agora. Mencionemos apenas duas:
    - i) É **mais fácil conversar com o cliente** que pediu o software se falarmos com objetos que existem no mundo dele (o mundo do software fica mais perto do mundo real)
    - ii) O software feito com OO pode ser feito com **maior qualidade** e pode ser mais fácil de escrever e, principalmente, alterar no futuro
- Esses pontos não ficarão claros na disciplina de Programação 2 mas certamente o ficarão adiante no curso
- Vamos pensar juntos: quais são os objetos no mundo Facebook?

## **Nossa primeira classe**

- A primeira classe que escreveremos representa uma conta bancária e será chamada de ContaSimples1

## **A documentação da classe**

- A documentação da classe que queremos escrever está [aqui](#)

## Como usaríamos a classe ContaSimples1?

— Observe o programa a seguir que manipula uma conta corrente realizando operações sobre a mesma.

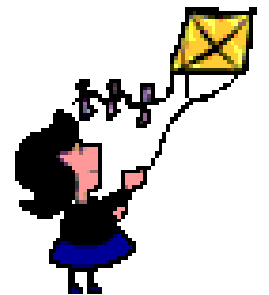
— Alguns pontos importantes:

- Um objeto é criado com a palavra **new**, especificando sua classe
  - i) Diz-se "chamamos o **construtor** da classe"
  - ii) A operação também se chama **instanciar** o objeto
- A variável `umaConta` é do tipo da classe `ContaSimples1` e armazena uma referência ao objeto



- É como se o objeto fosse uma pipa e a referência fosse uma linha amarrada à pipa
- Ou como se o objeto fosse uma televisão e a referência fosse o seu controle remoto
- É semelhante a ponteiros em outras linguagens

- Um objeto existe enquanto houver pelo menos uma referência a ele
  - i) Depois que não houver mais referências, o objeto some (que nem a pipa sem linha!)
- A sintaxe `umaConta.depositar(1000.0)` significa que estamos chamando o método `depositar()` do objeto `umaConta`



- i) Um método é como um sub-programa, subrotina ou função de outras linguagens

- Também se fala que estamos enviando a mensagem "depositar" para o objeto "umaConta"
- Certos métodos podem ter parâmetros e outros não
- Podemos imprimir um objeto como um todo!

```
// Programa Exemplo1
// Abra uma conta de número 1 para João com CPF 309140605-06
// A conta será "referenciada" com a variável umaConta
// Nesta conta, deposite R$1000,00
// Imprima o saldo da conta de João
// Saque R$300,00 desta conta
// Imprima o objeto umaConta
// Imprima o saldo da conta de João
```

```
package p2.exemplos;

/*
 * Movimentação simples de uma conta bancária
 */

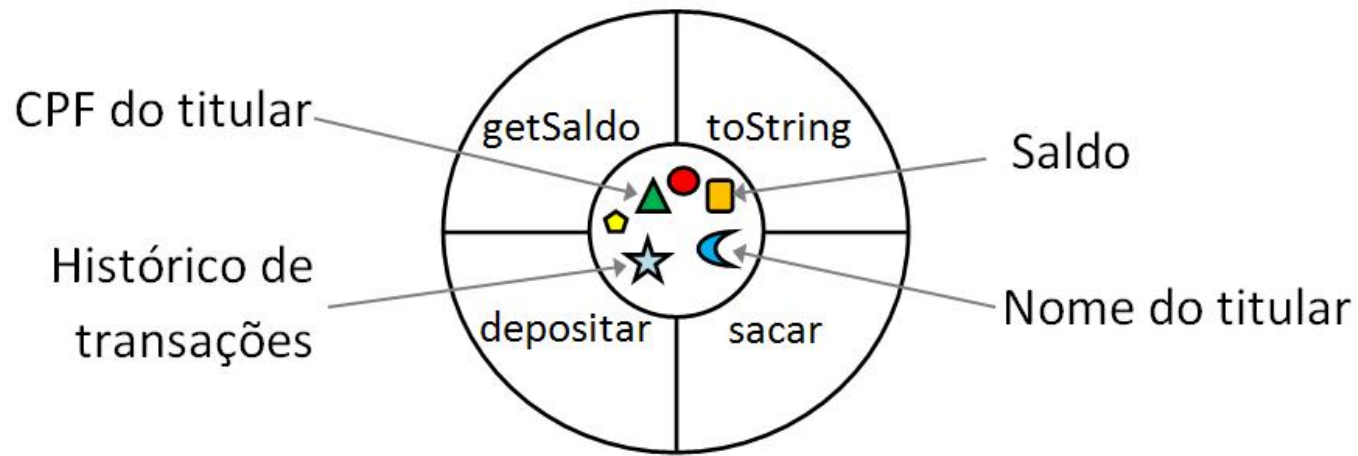
// Programa Exemplo1
public class Exemplo1 {
    // O programa sempre tem um "método" main que é onde começa a execução
    public static void main(String args[]) {
        // Abra uma conta de número 1 para João com CPF 309140605-06
        // A conta será "referenciada" com a variável umaConta
        ContaSimples1 umaConta = new ContaSimples1("Joao", "30914060506", 1);
        // Nesta conta, deposite R$1000,00
        umaConta.depositar(1000.0);
    }
}
```

```
// Imprima o saldo da conta de João
double saldo = umaConta.getSaldo();
System.out.print("Saldo da conta de Joao antes do saque: ");
System.out.println(saldo);

// Saque R$300,00 desta conta
umaConta.sacar(300.0);
// Imprima o objeto umaConta
System.out.println(umaConta);
// Imprima o saldo da conta de João
System.out.println("Saldo da conta de Joao depois do saque: "
    + umaConta.getSaldo());
} // fim do método main
} // fim da classe Exemplo1
```

## Não esquecer jamais: Encapsulamento e Ocultação da Informação

- Muito importante: conseguimos usar objetos da classe ContaSimples1 sem saber nada sobre como ContaSimples1 está escrita em Java!
  - Isso se chama **Ocultação de Informação** e é muito importante na programação
  - É a forma básica de lidar com a complexidade dos programas
- Também podemos dizer que a Classe ContaSimples1 **encapsula dados (estado) e comportamento em cima desses dados (que modifica o estado dos objetos)**
  - O estado do objeto são os atributos escondidos de nós (saldo, histórico de transações, dados do titular, ...)
  - O comportamento são os métodos que podemos chamar para manipular o estado do objeto, i.e., as mensagens que podemos mandar para os objetos
- Só podemos "mexer" no objeto através de seus métodos



Portanto, vê-se que classes como `ContaSimples1` definem um comportamento

— Mais precisamente, os objetos dessa classe é que têm comportamento definido pela classe

## Agora podemos criar nossa primeira classe: `ContaSimples1`

```
package p2.exemplos;

/**
 * Classe de conta bancária simples.
 *
 * @author Jacques Sauvé
 * @version 1.0
 */
public class ContaSimples1 {
    private String nome;

    private String cpf;

    private int numero;
```

```
private double saldo;

// construtor
/**
 * Cria uma conta a partir de um nome e cpf de pessoa física, e um número
 * de conta.
 *
 * @param nome
 *         O nome do titular da conta.
 * @param cpf
 *         O CPF do titular da conta.
 * @param número
 *         O número da conta.
 */
public ContaSimples1(String nome, String cpf, int numero) {
    this.nome = nome;
    this.cpf = cpf;
    this.numero = numero;
    saldo = 0.0;
}

// métodos

/**
 * Recupera o nome do titular da conta.
 *
 * @return O nome do titular da conta.
 */
public String getNome() {
    return nome;
}
```



```
/**
 * Recupera o CPF do titular da conta.
 *
 * @return O CPF do titular da conta.
 */
public String getCPF() {
    return cpf;
}
```

```
/**
 * Recupera o número da conta.
 *
 * @return O número da conta.
 */
public int getNumero() {
    return numero;
}
```

```
/**
 * Recupera o saldo da conta.
 *
 * @return O saldo da conta.
 */
public double getSaldo() {
    return saldo;
}
```

```
/**
 * Efetua um depósito numa conta.
 *
```

```
* @param valor
*          O valor a depositar.
*/
public void depositar(double valor) {
    saldo += valor;
}

/**
 * Efetua sacada na conta.
 *
 * @param valor
 *          O valor a sacar.
 * @return O sucesso ou não da operação.
 */
public boolean sacar(double valor) {
    if (saldo >= valor) {
        saldo -= valor;
        return true;
    }
    return false;
}

/**
 * Transforma os dados da conta em um String.
 *
 * @return O string com os dados da conta.
 */
public String toString() {
    return "numero " + numero + ", nome " + nome + ", saldo " + saldo;
}
```

```
}
```

## Os comentários Javadoc

— Há vários comentários iniciando com `/**`, por exemplo:

```
/**
 * Classe de conta bancária simples.
 *
 * @author    Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 */
```

— Esses comentários servem para criar [documentação automática](#) do seu programa através de uma ferramenta chamada javadoc

- Exemplo: ao rodar o seguinte comando: `javadoc -d docContaSimples1 -version -author ContaSimples1.java`

— A saída é [esta](#)

— Observe como os tags (`@author`, `@version`, `@param`, `@return`) saíram na documentação

## A declaração da classe

```
public class ContaSimples1 {
```

— Novas classes são definidas com:

```
    public class ContaSimples1 {
        corpo da classe
```

```
}
```

## Atributos

```
private String nome;  
private String cpf;  
private int numero;  
private double saldo;
```

- Atributos são definidos depois do primeiro { e fora de qualquer corpo de método
- "Método" significa função ou subrotina ou sub-programa
- Normalmente, atributos são colocados no início da definição da classe, ou talvez bem no final, antes do } final
- Os atributos de uma classe são equivalentes aos campos de um "record" ou "struct" em outras linguagens
- **A diferença básica é que com OO, a classe conterá também código para manipular esses dados**
- O adjetivo de **visibilidade** "private" significa que o atributo só pode ser "visto" (usado) pelo código *dentro* da classe
- "public" significa que todo mundo "vê", mesmo fora do corpo da classe
- Falaremos mais sobre visibilidade adiante
- Os atributos possuem um valor diferente para cada objeto instanciado
- Cada ContaSimples1 tem um valor diferente (armazenado em lugar diferente da memória) para o nome de titular, CPF, número de conta e saldo

## O construtor

```
/**
```

```

* Cria uma conta a partir de um nome e cpf de pessoa física, e um
* número de conta.
*
* @param nome
*         O nome do titular da conta.
* @param cpf
*         O CPF do titular da conta.
* @param numero
*         O número da conta.
*/
public ContaSimples1(String nome, String cpf, int numero) {
    this.nome = nome;
    this.cpf = cpf;
    this.numero = numero;
    saldo = 0.0;
}

```

- Ao chamar "new ContaSimples(...)", o **construtor** da classe é chamado
  - De forma semelhante aos outros métodos, pode ter parâmetros (aqui tem 3)
  - Chamamos de construtor default aquele que não recebe atributos e cria um objeto default da classe
  - Porém, o construtor nunca retorna um valor com "return"
- O construtor é normalmente usado para inicializar atributos
  - **this** é uma referência especial a este objeto
  - Portanto, this.nome se refere ao atributo "nome" do presente objeto ContaSimples1
- Se o parâmetro nome se chamasse outra coisa, digamos nomeTitular, a linha poderia ser mudada para:

```
nome = nomeTitular;
```

— Observe que, aqui, **nome** referencia o atributo sem precisar usar **this**

## Métodos accessor

```
// métodos

/**
 * Recupera o nome do titular da conta.
 *
 * @return O nome do titular da conta.
 */
public String getNome() {
    return nome;
}

/**
 * Recupera o CPF do titular da conta.
 *
 * @return O CPF do titular da conta.
 */
public String getCPF() {
    return cpf;
}

/**
 * Recupera o número da conta.
 *
 * @return O número da conta.
 */
public int getNumero() {
    return numero;
}
```

```
}

/**
 * Recupera o saldo da conta.
 *
 * @return O saldo da conta.
 */
public double getSaldo() {
    return saldo;
}
```

- Estamos vendo 4 métodos acima
- Observe como um método retorna um valor
  - "return expressão" automaticamente pára a execução do método e retorna o valor da expressão para o chamador do método
- Como todos esses métodos fazem apenas retornar o valor de um atributo, eles são chamados "**accessor methods**"

## Métodos de comportamento

```
/**
 * Efetua um depósito numa conta.
 *
 * @param valor
 *         O valor a depositar.
 */
public void depositar(double valor) {
    saldo += valor;
}
```

```

/**
 * Efetua sacada na conta.
 *
 * @param valor
 *         O valor a sacar.
 * @return O sucesso ou não da operação.
 */
public boolean sacar(double valor) {
    if (saldo >= valor) {
        saldo -= valor;
        return true;
    }
    return false;
}

```

— Os dois métodos acima mostram idéias importantes

- Primeiro, a passagem de parâmetros (nos dois métodos)
- Segundo, o método que não retorna nada (indicando com tipo de retorno void)
- Terceiro, o fato de que o saldo da conta não é mexido "de fora"

— Quem sabe mexer com o saldo é a ContaSimples1, em si

- Quem usa o objeto "de fora", não tem acesso direto aos atributos do objeto
- Só tem acesso aos métodos que definem o "comportamento" de objetos dessa classe
  - i) Neste caso, uma ContaSimples1 deixa que se façam depósitos e saques na Conta

— Esses métodos, juntamente com os outros métodos declarados como public, definem a **interface** da classe

## O método toString

```

/**

```



```

    * Transforma os dados da conta em um String.
    *
    * @return O string com os dados da conta.
    */
    public String toString() {
        return "numero " + numero + ", nome " + nome + ", saldo " + saldo;
    }

```

- Em Java, todo objeto deve ter uma representação como String
  - Facilita a impressão com System.out.println()
  - Facilita a depuração
- Definimos no método toString() o String a retornar para representar o objeto como String
- Normalmente, imprimem-se todos os atributos do objeto, em algum formato

## Outra forma de usar a classe ContaSimples1

- O método main pode ser colocado na própria classe ContaSimples1.java

```

package p2.exemplos;

/**
 * Classe de conta bancária simples.
 *
 * @author Jacques Sauvé
 * @version 2.0
 */
public class ContaSimples1 {
    private String nome;

    private String cpf;

```

```
private int numero;
```

```
private double saldo;
```

```
// construtor
```

```
/**
```

```
 * Cria uma conta a partir de um nome e cpf de pessoa física,  
 * e um número de conta.
```

```
 *
```

```
 * @param nome
```

```
 *           O nome do titular da conta.
```

```
 * @param cpf
```

```
 *           O CPF do titular da conta.
```

```
 * @param número
```

```
 *           O número da conta.
```

```
 */
```

```
public ContaSimples1(String nome, String cpf, int numero) {
```

```
    this.nome = nome;
```

```
    this.cpf = cpf;
```

```
    this.numero = numero;
```

```
    this.saldo = 0.0;
```

```
}
```

```
// métodos
```

```
/**
```

```
 * Recupera o nome do titular da conta.
```

```
 *
```

```
 * @return O nome do titular da conta.
```

```
 */
```

```
public String getNome() {
    return this.nome;
}

/**
 * Recupera o CPF do titular da conta.
 *
 * @return O CPF do titular da conta.
 */
public String getCPF() {
    return this.cpf;
}

/**
 * Recupera o número da conta.
 *
 * @return O número da conta.
 */
public int getNumero() {
    return this.numero;
}

/**
 * Recupera o saldo da conta.
 *
 * @return O saldo da conta.
 */
public double getSaldo() {
    return this.saldo;
}
```

```
/**
 * Efetua um depósito numa conta.
 *
 * @param valor
 *         O valor a depositar.
 */
public void depositar(double valor) {
    this.saldo += valor;
}

/**
 * Efetua sacada na conta.
 *
 * @param valor
 *         O valor a sacar.
 * @return O sucesso ou não da operação.
 */
public boolean sacar(double valor) {
    if (this.saldo >= valor) {
        this.saldo -= valor;
        return true;
    }
    return false;
}

/**
 * Transforma os dados da conta em um String.
 *
 * @return O string com os dados da conta.
 */
public String toString() {
```

```

        return "numero " + numero + ", nome " + nome + ", saldo " + saldo;
    }

    // O programa sempre tem um "método" main que é onde começa a execução
    public static void main(String args[]) {
        // Abra uma conta de número 1 para João com CPF 309140605-06
        // A conta será "referenciada" com a variável umaConta
        ContaSimples1 umaConta = new ContaSimples1("Joao", "30914060506", 1);
        // Nesta conta, deposite R$1000,00
        umaConta.depositar(1000.0);

        // Imprima o saldo da conta de João
        double saldo = umaConta.getSaldo();
        System.out.print("Saldo da conta de Joao antes do saque: ");
        System.out.println(saldo);

        // Saque R$300,00 desta conta
        umaConta.sacar(300.0);
        // Imprima o objeto umaConta
        System.out.println(umaConta);
        // Imprima o saldo da conta de João
        System.out.println("Saldo da conta de Joao depois do saque: "
            + umaConta.getSaldo());
    } // fim do método main
}

```

## Entendendo a palavra reservada static

— Palavra reservada **static**

- Você quer ter um campo armazenado em um mesmo espaço de memória, independentemente de quantos objetos da classe existem
  - i) Se vários objetos forem criados eles compartilham os mesmos campos estáticos
- Você quer um método que não está associado a qualquer objeto específico
  - i) Só acessa campos e/ou métodos de classe
  - ii) Os demais precisam estar associados a um objeto específico
- Observe que "main" é um método de classe (por causa da palavra "static")
  - Pode executar sem ter objeto em existência ainda
  - É assim que a bola começa a rolar e objetos são criados, etc.

## **Documentação de classe com UML**

- Além de Javadoc, uma outra forma de mostrar o que a classe faz é de usar uma representação gráfica numa linguagem chamada Unified Modeling Language (UML)

ContaSimples1
<ul style="list-style-type: none"> <li>- numero : int</li> <li>- nome : String</li> <li>- cpf : String</li> <li>- saldo : double</li> </ul>
<ul style="list-style-type: none"> <li>+ sacar(valor : double) : boolean</li> <li>+ depositar(valor : double) : void</li> <li>+ getSaldo() : double</li> <li>+ getNome() : String</li> <li>+ getCpf() : String</li> <li>+ getNúmero() : int</li> </ul>

powered by astah® 

— Também podemos mostrar apenas a parte pública, sem revelar detalhes internos que não interessam aos *clientes* da classe

ContaSimples1
<ul style="list-style-type: none"> <li>+ sacar(valor : double) : boolean</li> <li>+ depositar(valor : double) : void</li> <li>+ getSaldo() : double</li> <li>+ getNome() : String</li> <li>+ getCpf() : String</li> <li>+ getNúmero() : int</li> </ul>

powered by astah® 

- UML é também chamada de linguagem de "modelagem visual"
  - Um modelo é uma representação do mundo real que nos interessa
  - UML permite criar modelos visuais
  - Um programa é um modelo mais elaborado que consegue *executar*
- Veja mais informações sobre UML [aqui](#)

## Palavras adicionais sobre a nossa primeira classe

- Como na programação não OO, o “método” é uma técnica de **ocultação de informação**
  - Para poder diminuir a complexidade, focando o programador numa coisa só
  - Observe como os métodos escondem os detalhes internos do objeto
  - Para quem está “fora”, só usando o objeto, sabemos que podemos fazer um saque e um depósito mas nada sabemos sobre *como* isso ocorre, internamente
  - Isso é uma chave da programação!
- Também estamos vendo a técnica de **encapsulamento** em ação
  - Dados (saldo, etc.) foram encapsulados numa caixa preta junto com uma interface (os métodos) para manipular os dados que estão dentro da caixa
- A combinação de ocultação da informação e encapsulamento é poderosa:
  - Manter dados privados e disponibilizar junto deles alguns métodos públicos para manipulá-los é melhor do que acessar diretamente os dados para manipulação
  - É melhor perguntar a alguém o que ele tomou no café da manhã ou enfiar sua mão goela



abaixo e puxar a gosma para descobrir ...?

— Observe como os **métodos são pequenos**

- Isso é normal na orientação a objeto
- Você deve desconfiar de métodos grandes: devem ser complicados demais e deveriam ser quebrados

## Testes: como fazer testes automáticos em Java?

— Agora que a classe está pronta, queremos testá-la

- Como faríamos?

— Eis alguns testes que podemos fazer, escritos em português

Cria um conta com nome "Jacques Sauve", cpf 123456789-01, e número 123  
Verifique que o nome da conta é Jacques Sauve  
Verifique que o cpf da conta é 123456789-01  
Verifique que o número da conta é 123  
Verifique que o saldo da conta é 0.0  
Deposite R\$100,00  
Verifique que o saldo é R\$100,00  
Saque R\$45,00  
Verifique que o saldo é R\$55,00  
Tente sacar R\$56,00 e verifique que não é possível  
Verifique que o saldo continua em R\$55,00

— Veremos adiante que estes testes não estão completos, mas vamos começar com eles

- Você pode pensar em mais testes que deveriam ser feitos?
- Queremos agora fazer com que os testes sejam realizados de forma *automática*
  - Ter testes automáticos é muito, muito bom
  - Permite que você execute os testes a qualquer momento
    - i) Você pode repetir os testes centenas de vezes sem custo adicional
    - ii) Imagine a situação se os testes fossem "manuais"
  - Permite saber exatamente quando a implementação está terminada (quando os testes rodam)
  - Permite fazer alterações ao código ao longo do tempo e assegurar-se de que nada quebrou
  - Os próprios testes servem de documentação para a classe
    - i) Se você quiser saber como uma classe funciona ou como pode/deve ser usada, examine os testes
- Vamos automatizar os testes usando um testador chamado JUnit
  - JUnit ajuda a fazer "testes de unidade" (uma unidade = uma classe)
  - Tem outros tipos de testes que veremos em outro momento
- Vejamos uma classe de testes

```
package p2.exemplos.tests;  
  
import org.junit.Assert;  
import org.junit.Before;
```

```

import org.junit.Test;

import p2.exemplos.ContaSimples1;

public class TestaContaSimples1 {

    private ContaSimples1 umaConta;

    @Before public void criaConta() {
        umaConta = new ContaSimples1("Jacques Sauve", "123456789-01", 123);
    }

    @Test public void testaContaSimples1() {
        Assert.assertEquals("Nome errado", "Jacques Sauve",
umaConta.getNome());
        Assert.assertEquals("cpf errado", "123456789-01", umaConta.getCPF());
        Assert.assertEquals("Número errado", 123, umaConta.getNumero());
        Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
        umaConta.depositar(100.0);
        Assert.assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005);
        Assert.assertTrue("Nao consegui sacar", umaConta.sacar(45.0));
        Assert.assertEquals("Saldo errado", 55.0, umaConta.getSaldo(), 0.005);
        Assert.assertFalse("Conseguir sacar demais", umaConta.sacar(56.0));
        Assert.assertEquals("Saldo errado", 55.0, umaConta.getSaldo(), 0.005);
    }
}

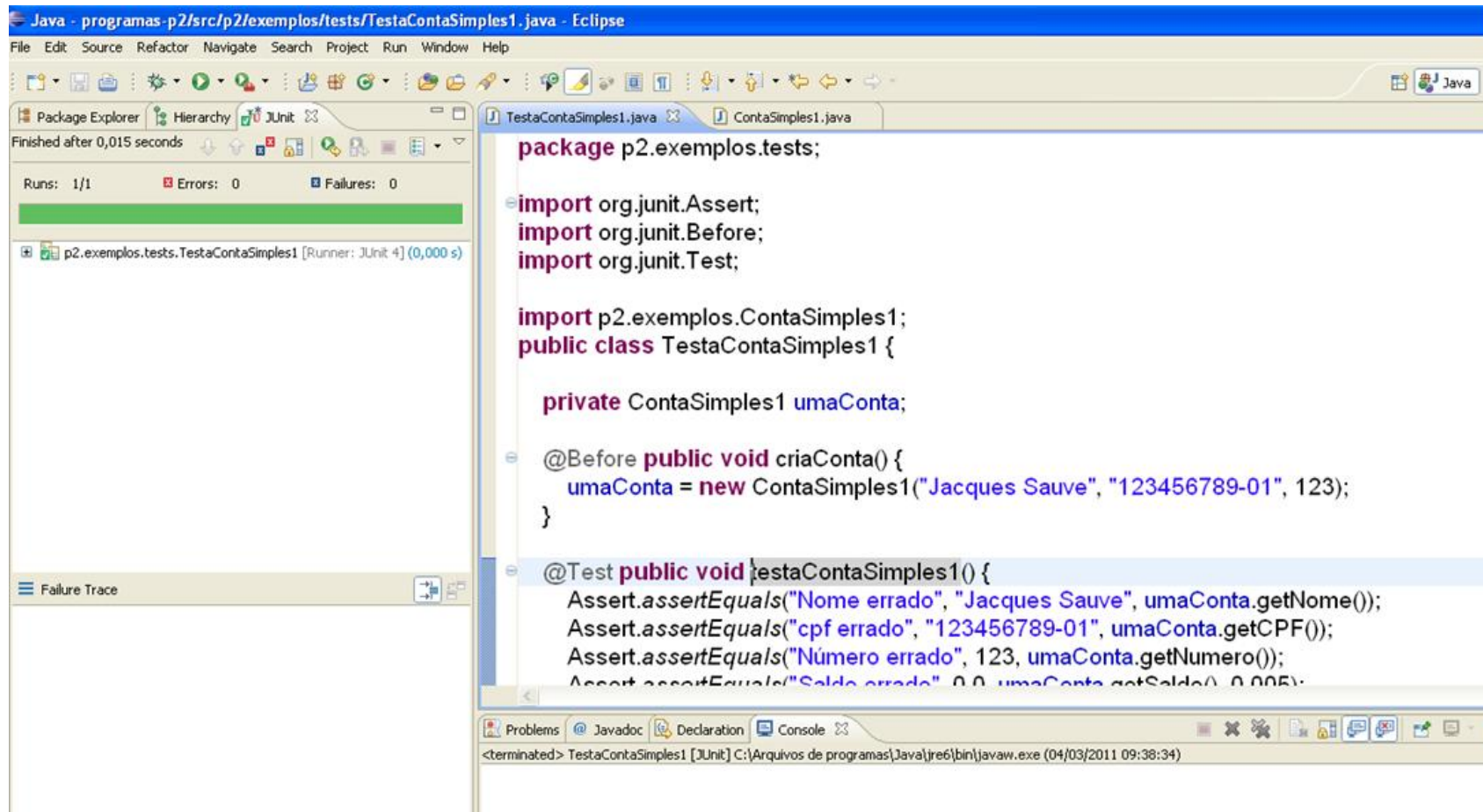
```

- assertEquals, assertTrue, assertFalse são métodos do pacote JUnit e servem para realizar testes
  - assertEquals("Mensagem de erro se o teste falhar", string esperado, string a testar)
  - assertEquals("Mensagem de erro se o teste falhar", valor double esperado, valor double a

testar, precisão)

- assertTrue("Mensagem de erro se o teste falhar", valor a testar que deve retornar true)
- assertFalse("Mensagem de erro se o teste falhar", valor a testar que deve retornar false)

- Examine os testes com muita atenção antes de continuar
- Tente rodar os testes com JUnit e a classe ContaSimples1.java pronta
- Os testes devem rodar (veja figura abaixo)
- Introduza erros nos testes e veja o que ocorre



```
Java - programas-p2/src/p2/exemplos/tests/TestaContaSimples1.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer Hierarchy JUnit
Finished after 0,015 seconds
Runs: 1/1 Errors: 0 Failures: 0
p2.exemplos.tests.TestaContaSimples1 [Runner: JUnit 4] (0,000 s)

Failure Trace

package p2.exemplos.tests;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import p2.exemplos.ContaSimples1;
public class TestaContaSimples1 {

    private ContaSimples1 umaConta;

    @Before public void criaConta() {
        umaConta = new ContaSimples1("Jacques Sauve", "123456789-01", 123);
    }

    @Test public void testaContaSimples1() {
        Assert.assertEquals("Nome errado", "Jacques Sauve", umaConta.getNome());
        Assert.assertEquals("cpf errado", "123456789-01", umaConta.getCPF());
        Assert.assertEquals("Número errado", 123, umaConta.getNumero());
        Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
    }
}
```

<terminated> TestaContaSimples1 [JUnit] C:\Arquivos de programas\Java\jre6\bin\javaw.exe (04/03/2011 09:38:34)

Mas queremos usar "*Test-Driven Development*" (TDD) ...

- TDD é uma postura que consiste em escrever os testes antes de escrever o código da classe
  - Antes?!?!?!?
  - Sim, antes, não depois!
- Então, você escreveria a classe de testes e, então, a classe desejada

### **Conversando mais sobre testes ...**

- Os testes da classe ContaSimples1 não estão completos
- Principalmente, as condições de “exceção” não foram testadas
- Exemplos:
  - Construtor
    - i) O que ocorre se o nome for nulo ou vazio?
    - ii) O que ocorre se o CPF for nulo ou vazio?
    - iii) O que ocorre se o CPF for inválido?
    - iv) O que ocorre se o número da conta não for positivo?
  - depositar
    - i) O que ocorre se o valor 0.0 for depositado?
    - ii) O que ocorre se um valor negativo for depositado?
    - iii) Depositar centavos funciona?

- iv) O que ocorre se depositar frações de centavos?
  - sacar
    - i) O que ocorre se o valor 0.0 for sacado?
    - ii) O que ocorre se um valor negativo for sacado?
    - iii) Sacar centavos funciona?
    - iv) O que ocorre se sacar frações de centavos?
  - toString
    - i) toString não foi testado
    - ii) Tem que testar com que valores de saldo?
      - (1) Zero
      - (2) Positivo
      - (3) Com centavos
- Muitos testes são necessários para garantir que tratamos adequadamente de **todas** as situações
  - Para que testar se não for assim?
- É o que veremos agora ...

## Tratamento de Erros

- É importante diferenciar o **descobrimento** do erro e o **tratamento** do erro
  - É muito frequente descobrir algo errado em um lugar mas querer tratar o erro em outro lugar
  - Por exemplo, tratar o erro de nome vazio em ContaSimples1() é ruim porque é um método de "baixo nível" que não sabe sequer que tipo de interface está sendo usada (gráfica, a caractere), etc.
    - i) Não seria apropriado fazer um println e exit
- A solução OO: **Exceções**
- Vamos usar um mecanismo novo para **retornar erros**
  - O retorno normal de valores por um método usa "return"
  - O retorno anormal (indicando erro) usa outra palavra para retornar do método: **throw**
  - Da mesma forma que "return", "throw" retorna imediatamente do método
  - Diferentemente de "return", "throw" só retorna objetos de tipos especiais chamados **exceções**
  - A exceção pode conter uma mensagem indicando o erro que ocorreu para que o cliente da classe (aquele que chamou o método que lançou a exceção) tenha informação suficiente para tratar o erro adequadamente
- "throw" faz com que **todos** os métodos chamados retornem, até o ponto em que algum método **captura a exceção** para tratar o erro

- Essa captura é feita com um bloco `try-catch`

— Segue um exemplo de uso de exceções como introdução à sintaxe de tratamentos de erros em Java

```
package p2.exemplos;

import java.util.Random;
import java.util.Scanner;

/**
 * Um jogo de sorte para bebês. <br>
 * Importante: Este programa é apenas ilustrativo. Um exemplo de uso
 * de exceções para os alunos irem se familiarizando com a sintaxe.
 * Na pratica, exceções são usadas para indicar erros.
 *
 * @author raquel
 *
 */
public class LancaExcecoes {

    public static final int NUM_JOGADAS = 5;
    private static int piques;
    private static int bombas;

    public static void main(String[] args) {
        doIt();
        // System.out.printf("Sorte = %d; Num. de derrotas = %d\n", piques,
        // bombas);
    }
}
```



```

private static void doIt() {
    int jogada = 1;
    while (jogada++ <= NUM_JOGADAS) {
        prompt();
        try {
            lanca();
            piques++;
        } catch (Exception e) {
            bombas++;
            System.out.println(e.getMessage());
        } finally { //codigo optativo
            System.out.println("Fim da jogada " + (jogada-1));
            System.out.printf("Sorte = %d; Num. de derrotas = %d\n",
                               piques, bombas);
        }
    }
}

```

```

private static void lanca() throws Exception {
    Random rand = new Random();
    if (rand.nextBoolean()) {
        throw new Exception("Bomba!");
        //qq codigo aqui dah erro de compilacao
        //System.out.println("vai dar erro :P");
    }
    System.out.println("Picolé!");
}

```

```

private static void prompt() {
    Scanner scn = new Scanner(System.in);
    System.out.println("Aperte <enter> para a proxima jogada.");
}

```

```
        scn.nextLine();
    }
}
```

— Segue um outro exemplo mais complexo do uso de exceções para você [fuçar em casa](#)

```
package p2.exemplos;

/**
 * Classe que mostra o uso de exceções.
 *
 * @author Jacques Philippe Sauvé
 *
 */
public class TesteDeExcecoes {
    public static void main(String[] args) {
        new TesteDeExcecoes().doIt();
        System.out.println("bye, bye");
    }

    private void doIt() {
        try {
            System.out.println("doIt: chama a()");
            a(false);
            System.out.println("doIt: a() retornou");
            System.out.println("main: chama b()");
            b(false);
            System.out.println("doIt: b() retornou");
            System.out.println("doIt: nao capturou excecao");
        } catch (Exception ex) {
            System.out.println("doIt: capturou excecao: " + ex.getMessage());
        }
    }
}
```

```

    }
}

private void a(boolean lanca) throws Exception {
    System.out.println("a: chama c(" + lanca + ")");
    c(lanca);
    System.out.println("a: c() retornou");
}

private void c(boolean lanca) throws Exception {
    System.out.println("c: inicio");
    if (lanca) {
        System.out.println("c: vai lancar");
        throw new Exception("bomba!");
    }
    // System.out.println("c: lancou"); // causaria erro de compilação
    System.out.println("c: fim");
}

private void b(boolean lanca) throws Exception {
    try {
        System.out.println("b: chama c(" + lanca + ")");
        c(lanca);
        System.out.println("b: c() retornou sem excecao");
    } catch (Exception ex) {
        System.out.println("b: capturou excecao: " + ex.getMessage());
        // tire o comentário abaixo para ver o comportamento
        // throw ex;
        // ou então
        // throw new Exception("granada!");
    } finally {

```

```

        System.out.println("b: finally");
    }
}
}

```

—Agora, vamos ver como montar esse circo

```

/*
 * setUp():
 * cria uma conta para "Jacques Sauve", com cpf "123456789-01" e
 * número 123
 */

/*
 * testaErrosNoConstrutor():
 * cria conta com nome " " CPF "123456789-01" e numero 123
 * Verifica que exceção deve ser lançada: "Nome nao pode ser nulo ou vazio"
 * cria conta com nome null CPF "123456789-01" e numero 123
 * Verifica que exceção deve ser lançada: "Nome nao pode ser nulo ou vazio"
 * cria conta com nome "jacques" CPF " " e numero 123
 * Verifica que exceção deve ser lançada: "CPF nao pode ser nulo ou vazio"
 * cria conta com nome "jacques" CPF null e numero 123
 * Verifica que exceção deve ser lançada: "CPF nao pode ser nulo ou vazio"
 * cria conta com nome "jacques" CPF "123456789-01" e numero 0
 * Verifica que exceção deve ser lançada: "Número da conta deve ser
 *      positivo"
 * cria conta com nome "jacques" CPF "123456789-01" e numero -1
 * Verifica que exceção deve ser lançada: "Número da conta deve ser
 *      positivo"
 */

```

```
/*
 * testaDepositar()
 * verifica que o saldo inicial é zero
 * deposita 100
 * verifica que o saldo é 100
 * tenta depositar -0.01
 * verifica que exceção é lançada: "Deposito nao pode ser negativo"
 * deposita 0.01
 * verifica que o saldo é 100.1
 */
```

```
/*
 * testaSacar():
 * verifica que o saldo inicial é zero
 * verifica que não consegue sacar 0.0
 * verifica que o saldo continua sendo 0.0
 * deposita 100
 * verifica que pode sacar 23.10
 * verifica que o saldo é 76.90
 * verifica que não consegue sacar 76.91
 * verifica que o saldo é 76.90
 * verifica que consegue sacar 76.90
 * verifica que o saldo é 0.0
 * tenta sacar -0.01
 * verifica que exceção é lançada: "Saque nao pode ser negativo"
 * verifica que o saldo é 0.0
 */
```

```
/*
 * testaToString():
```

```
* verifica que a representação em string da conta criada é "numero 123,  
*     nome Jacques Sauve, saldo 0.0"  
* deposita 1.23  
* verifica que a representação em string da conta criada é "numero 123,  
* nome Jacques Sauve, saldo 1.23"  
*/
```

— Eis a classe de testes com novos testes

- Vamos testar a classe ContaSimples2 que trata erros adequadamente
- Os testes estão em TestaContaSimples2.java

```
package p2.exemplos.tests;  
  
import org.junit.Assert;  
import org.junit.Before;  
import org.junit.Test;  
  
import p2.exemplos.ContaSimples2;  
  
public class TestaContaSimples2 {  
  
    private ContaSimples2 umaConta;  
  
    @Before  
    public void criaConta() throws Exception {  
        umaConta = new ContaSimples2("Jacques Sauve", "123456789-01", 123);  
    }  
  
    @Test  
    public void testaContaSimples1() throws Exception {  
        Assert.assertEquals("Nome errado", "Jacques Sauve",
```

```

        umaConta.getNome());
Assert.assertEquals("cpf errado", "123456789-01", umaConta.getCPF());
Assert.assertEquals("Número errado", 123, umaConta.getNúmero());
Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
umaConta.depositar(100.0);
Assert.assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005);
Assert.assertTrue("Nao consegui sacar", umaConta.sacar(45.0));
Assert.assertEquals("Saldo errado", 55.0, umaConta.getSaldo(), 0.005);
Assert.assertFalse("Consegui sacar demais", umaConta.sacar(56.0));
Assert.assertEquals("Saldo errado", 55.0, umaConta.getSaldo(), 0.005);
}

```

@Test

```

public void testaErrosNoConstrutor() {
    try {
        ContaSimples2 umaConta = new ContaSimples2("", "123456789-01", 123);
        Assert.fail("Esperava excecao de nome vazio");
    } catch (Exception ex) {
        Assert.assertEquals("Mensagem errada",
            "Nome nao pode ser nulo ou vazio", ex.getMessage());
    }
    try {
        ContaSimples2 umaConta = new ContaSimples2(null, "123456789-01",
            123);
        Assert.fail("Esperava excecao de nome nulo");
    } catch (Exception ex) {
        Assert.assertEquals("Mensagem errada",
            "Nome nao pode ser nulo ou vazio", ex.getMessage());
    }
    try {
        ContaSimples2 umaConta = new ContaSimples2("Jacques Sauve", "",

```

```
123);  
    Assert.fail("Esperava excecao de CPF vazio");  
} catch (Exception ex) {  
    Assert.assertEquals("Mensagem errada",  
        "CPF nao pode ser nulo ou vazio", ex.getMessage());  
}  
try {  
    ContaSimples2 umaConta = new ContaSimples2("Jacques Sauve", null,  
        123);  
    Assert.fail("Esperava excecao de CPF nulo");  
} catch (Exception ex) {  
    Assert.assertEquals("Mensagem errada",  
        "CPF nao pode ser nulo ou vazio", ex.getMessage());  
}  
try {  
    ContaSimples2 umaConta = new ContaSimples2("Jacques Sauve",  
        "123456789-01", 0);  
    Assert.fail("Esperava excecao de numero de conta errada");  
} catch (Exception ex) {  
    Assert.assertEquals("Mensagem errada",  
        "Número da conta deve ser positivo", ex.getMessage());  
}  
try {  
    ContaSimples2 umaConta = new ContaSimples2("Jacques Sauve",  
        "123456789-01", -1);  
    Assert.fail("Esperava excecao de numero de conta errada");  
} catch (Exception ex) {  
    Assert.assertEquals("Mensagem errada",  
        "Número da conta deve ser positivo", ex.getMessage());  
}  
}
```



@Test

```
public void testaDepositar() throws Exception {
    Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
    umaConta.depositar(100.0);
    Assert.assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005);
    umaConta.depositar(0.0);
    Assert.assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005);
    try {
        umaConta.depositar(-0.01);
        Assert.fail("Esperava excecao no deposito");
    } catch (Exception ex) {
        Assert.assertEquals("Mensagem errada",
            "Deposito nao pode ser negativo", ex.getMessage());
    }
    umaConta.depositar(0.01);
    Assert.assertEquals("Saldo errado", 100.01, umaConta.getSaldo(),
0.005);
}
```

@Test

```
public void testaSacar() throws Exception {
    Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
    Assert.assertTrue("Nao consegui sacar", umaConta.sacar(0.0));
    Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
    umaConta.depositar(100.0);
    Assert.assertTrue("Nao consegui sacar", umaConta.sacar(23.10));
    Assert.assertEquals("Saldo errado", 76.90, umaConta.getSaldo(), 0.005);
    Assert.assertFalse("Consegui sacar demais", umaConta.sacar(76.91));
    Assert.assertTrue("Nao consegui sacar", umaConta.sacar(76.90));
    Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
}
```

```

    try {
        umaConta.sacar(-0.01);
        Assert.fail("Esperava excecao no saque");
    } catch (Exception ex) {
        Assert.assertEquals("Mensagem errada",
            "Saque nao pode ser negativo", ex.getMessage());
    }
    Assert.assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
}

@Test
public void testaToString() throws Exception {
    Assert.assertEquals("toString errado",
        "numero 123, nome Jacques Sauve, saldo 0.0", umaConta
            .toString());
    umaConta.depositar(1.23);
    Assert.assertEquals("toString errado",
        "numero 123, nome Jacques Sauve, saldo 1.23", umaConta
            .toString());
}
}

```

- Vamos escrever a classe ContaSimples2 com base nestes testes?
  - Lembram que os testes são o roteiro de como a classe funciona?
- A classe ContaSimples2 está a seguir

```

package p2.exemplos;

/**
 * Classe de conta bancária simples.
 *

```

```
* @author Jacques Sauvé
* @version 2.0
*/
public class ContaSimples2 {
    private static final String STRING_VAZIA = "";

    private String nome;

    private String cpf;

    private int numero;

    private double saldo;

    // construtor
    /**
de    * Cria uma conta a partir de um nome e cpf de pessoa física, e um número
    * conta.
    *
    * @param nome
    *         O nome do titular da conta.
    * @param cpf
    *         O CPF do titular da conta.
    * @param número
    *         O número da conta.
    * @throws Exception
    *         Se o nome for nulo ou vazio, o CPF for nulo ou vazio ou a
    *         conta não for um número positivo
    */
    public ContaSimples2(String nome, String cpf, int numero) throws Exception
```

```
{  
    if (nome == null || nome.equals(STRING_VAZIA)) {  
        throw new Exception("Nome nao pode ser nulo ou vazio");  
    }  
    if (cpf == null || cpf.equals(STRING_VAZIA)) {  
        throw new Exception("CPF nao pode ser nulo ou vazio");  
    }  
    if (numero <= 0) {  
        throw new Exception("Número da conta deve ser positivo");  
    }  
    this.nome = nome;  
    this.cpf = cpf;  
    this.numero = numero;  
    this.saldo = 0.0;  
}  
  
// métodos  
  
/**  
 * Recupera o nome do titular da conta.  
 *  
 * @return O nome do titular da conta.  
 */  
public String getNome() {  
    return nome;  
}  
  
/**  
 * Recupera o CPF do titular da conta.  
 *  
 * @return O CPF do titular da conta.
```

```
 */
public String getCPF() {
    return cpf;
}

/**
 * Recupera o número da conta.
 *
 * @return O número da conta.
 */
public int getNumero() {
    return numero;
}

/**
 * Recupera o saldo da conta.
 *
 * @return O saldo da conta.
 */
public double getSaldo() {
    return saldo;
}

/**
 * Efetua um depósito numa conta.
 *
 * @param valor
 *         O valor a depositar.
 * @throws Exception
 *         Quando o valor a depositar é negativo.
 */
```

```
 */
public void depositar(double valor) throws Exception {
    if (valor < 0.0) {
        throw new Exception("Deposito nao pode ser negativo");
    }
    saldo += valor;
}
```

```
/**
 * Efetua saque na conta.
 *
 * @param valor
 *         O valor a sacar.
 * @return O sucesso ou não da operação.
 * @throws Exception
 *         Para um valor de saque negativo
 */
```

```
public boolean sacar(double valor) throws Exception {
    if (valor < 0.0) {
        throw new Exception("Saque nao pode ser negativo");
    }
    if (saldo >= valor) {
        saldo -= valor;
        return true;
    }
    return false;
}
```

```
/**
 * Transforma os dados da conta em um String.
 *
```

```

    * @return O string com os dados da conta.
    */
    public String toString() {
        return "numero " + getNumero() + ", nome " + getNome() + ", saldo " +
getSaldo();
    }

// O programa sempre tem um "método" main que é onde começa a execução
public static void main(String args[]) {
    // Abra uma conta de número 1 para João com CPF 309140605-06
    // A conta será "referenciada" com a variável umaConta
    ContaSimples2 umaConta = null;
    try {
        umaConta = new ContaSimples2("Joao", "30914060506", 1);
        // Nesta conta, deposite R$1000,00
        umaConta.depositar(1000.0);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }

    // Imprima o saldo da conta de João
    double saldo = umaConta.getSaldo();
    System.out.print("Saldo da conta de Joao antes do saque: ");
    System.out.println(saldo);

    // Saque R$300,00 desta conta
    try {
        umaConta.sacar(300.0);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

```

```
// Imprima o objeto umaConta
System.out.println(umaConta);
// Imprima o saldo da conta de João
System.out.println("Saldo da conta de Joao depois do saque: "
    + umaConta.getSaldo());
} // fim do método main
}
```

## Nossa segunda classe: com vários construtores

### A classe Pessoa

- Imagine uma classe que representa pessoas.
- Vamos pensar antes nos testes para esta classe Pessoa

```
/*
caso de teste1():
- cria uma Pessoa com nome "Maria" e cpf "111"
- verifica se nome é "Maria"
- verifica se cpf é "111"
- modifica nome para "Maria da Silva e Silva"
- verifica se nome é "Maria da Silva e Silva"

caso de teste2():
- cria uma Pessoa com nome "Jose"
- verifica se nome é "Jose"
- verifica se cpf é ""
- modifica cpf para "112"
- verifica se cpf é "112"

caso de teste3():
```



```
- cria uma Pessoa com nome "Joao" e cpf "113"
- cria outra Pessoa com nome "Joao" e cpf "113"
- verifica se as duas pessoas são iguais pelo nome e
cpf
*/
```

### — Exercício para casa:

- Escreva testes para a classe Pessoa

```
package p1.aplic.geral;

import java.io.*;

/**
 * Classe representando uma pessoa física.
 *
 * @author Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0 <br>
 *      Copyright (C) 1999 Universidade Federal de Campina Grande.
 */

public class Pessoa {
    private String nome;
    private String cpf;

    // Construtores
    /**
     * Constroi uma pessoa com nome e CPF dados.
     *
     * @param nome
     *      O nome da pessoa.
     */
}
```

```
*
* @param cpf
*         O CPF da pessoa.
*/
public Pessoa(String nome, String cpf) {
    this.nome = nome;
    this.cpf = cpf;
}

/**
 * Constroi uma pessoa com nome dado e sem CPF.
 *
 * @param nome
 *         O nome da pessoa.
 */
public Pessoa(String nome) {
    this(nome, "");
}

/**
 * Recupera o nome da pessoa.
 *
 * @return O nome da pessoa.
 */
public String getNome() {
    return nome;
}

/**
 * Recupera o CPF da pessoa.
 *
```

```
    * @return O CPF associado à pessoa.
    */
    public String getCPF() {
        return cpf;
    }

    /**
     * Ajusta o nome da pessoa.
     *
     * @param nome
     *         O nome da pessoa.
     */
    public void setNome(String nome) {
        this.nome = nome;
    }

    /**
     * Ajusta o CPF da pessoa.
     *
     * @param cpf
     *         O CPF associado à pessoa.
     */
    public void setCPF(String cpf) {
        this.cpf = cpf;
    }

    /**
     * Representa a pessoa como string
     */
    public String toString() {
        return "Nome " + nome + ", cpf " + cpf;
    }
}
```

```

}

/**
 * Testa a igualdade de um objeto com esta pessoa.
 *
 * @param objeto
 *         O objeto a comparar com esta pessoa.
 * @return true se o objeto for igual a esta pessoa, false caso contrário.
 */
public boolean equals(Object objeto) {
    if (!(objeto instanceof Pessoa)) {
        return false;
    }
    Pessoa outra = (Pessoa) objeto;
    return getNome().equals(outra.getNome())
        && getCPF().equals(outra.getCPF());
}
}

```

— Temos dois construtores

- Há um overload do nome Pessoa

```

// Construtores
/**
 * Constroi uma pessoa com nome e CPF dados.
 *
 * @param nome
 *         O nome da pessoa.
 *
 * @param cpf
 *         O CPF da pessoa.

```

```

    */
    public Pessoa(String nome, String cpf) {
        this.nome = nome;
        this.cpf = cpf;
    }

    /**
     * Constroi uma pessoa com nome dado e sem CPF.
     *
     * @param nome
     *         O nome da pessoa.
     */
    public Pessoa(String nome) {
        this(nome, "");
    }

```

— Uma Pessoa pode ser criada de duas formas: com e sem CPF

```

Pessoa pessoa1 = new Pessoa("Raquel", "98765432-11");
Pessoa pessoa2 = new Pessoa("Raquel");

```

- Observe também que o segundo construtor chama this() como se this fosse um método
  - this(...) é a chamada a um construtor da classe, neste caso com dois argumentos
  - Isto é, Pessoa(String) chama Pessoa(String, String), passando o string nulo como CPF
  - É uma forma de não duplicar código (fatorando o que é igual)
- Observe a existência de métodos "mutator" (que alteram o valor dos atributos)
- Finalmente, é muito comum uma classe incluir um método equals() para testar se outro objeto qualquer é igual a este (que foi chamado)
  - Dá para ver que dois objetos Pessoa são iguais se tiverem nome e CPF iguais
  - Entenderemos adiante que Object é um objeto de classe geral e instanceof diz se um objeto "é" de uma certa classe

## Usando a classe Pessoa

— Agora vamos usar a classe Pessoa, através de uma outra classe ContaSimples3.java

```
package p2.exemplos;

import p1.aplic.geral.Pessoa;

/**
 * Classe de conta bancária simples.
 *
 * @author Jacques Sauv  
 * @author Raquel Lopes
 * @version 3.0
 */
public class ContaSimples3 {
    private static final String STRING_VAZIA = "";

    private Pessoa titular;

    private int numero;

    private double saldo;

    // construtores

    /**
     * Cria uma conta a partir de uma pessoa e n  mero de conta.
     *
     * @param titular
     *            O titular da conta.
     * @param n  mero
```

```

*           O número da conta.
*/
public ContaSimples3(Pessoa titular, int numero) throws Exception {
    if (titular.getNome() == null ||
titular.getNome().equals(STRING_VAZIA)) {
        throw new Exception("Nome nao pode ser nulo ou vazio");
    }
    if (titular.getCPF() == null || titular.getCPF().equals(STRING_VAZIA))
{
        throw new Exception("CPF nao pode ser nulo ou vazio");
    }
    if (numero <= 0) {
        throw new Exception("Número da conta deve ser positivo");
    }
    this.titular = titular;
    this.numero = numero;
    this.titular = titular;
    saldo = 0.0;
}

/**
* Cria uma conta a partir de um nome e cpf de pessoa física, e um número
de
* conta.
*
* @param nome
*           O nome do titular da conta.
* @param cpf
*           O CPF do titular da conta.
* @param número
*           O número da conta.

```

```
* @throws Exception
*         Se o nome for nulo ou vazio, o CPF for nulo ou vazio ou a
*         conta não for um número positivo
*/
public ContaSimples3(String nome, String cpf, int numero) throws Exception
{
    this(new Pessoa(nome, cpf), numero);
}

// métodos

/**
 * Recupera o nome do titular da conta.
 *
 * @return O nome do titular da conta.
 */
public String getNome() {
    return titular.getNome();
}

/**
 * Recupera o CPF do titular da conta.
 *
 * @return O CPF do titular da conta.
 */
public String getCPF() {
    return titular.getCPF();
}

/**
 * Recupera o número da conta.
```



```
*
* @return O número da conta.
*/
public int getNumero() {
    return numero;
}

/**
 * Recupera o titular da conta.
 *
 * @return O titular da conta.
 */
public Pessoa getTitular() {
    return titular;
}

/**
 * Recupera o saldo da conta.
 *
 * @return O saldo da conta.
 */
public double getSaldo() {
    return saldo;
}

/**
 * Efetua um depósito numa conta.
 *
 * @param valor
 *           O valor a depositar.
 * @throws Exception
```

```

*           Quando o valor a depositar é negativo.
*
*/
public void depositar(double valor) throws Exception {
    if (valor < 0.0) {
        throw new Exception("Deposito nao pode ser negativo");
    }
    saldo += valor;
}

/**
 * Efetua saque na conta.
 *
 * @param valor
 *           O valor a sacar.
 * @return O sucesso ou não da operação.
 * @throws Exception
 *           Para um valor de saque negativo
 */
public boolean sacar(double valor) throws Exception {
    if (valor < 0.0) {
        throw new Exception("Saque nao pode ser negativo");
    }
    if (saldo >= valor) {
        saldo -= valor;
        return true;
    }
    return false;
}

/**

```

```
    * Transforma os dados da conta em um String.  
    *  
    * @return O string com os dados da conta.  
    */  
    public String toString() {  
        return "numero " + numero + ", nome " + getNome() + ", saldo " + saldo;  
    }  
}
```

- Observe particularmente os seguintes pontos:
  - Os dois construtores com overload da palavra Pessoa como método
  - Como o segundo construtor chama o primeiro
  - Como variáveis temporárias são evitadas no segundo construtor
  - O que getTitular() retorna
  - Como getNome() e getCPF() fazem seu trabalho
- Você vê por qual motivo toString() chama getNome() em vez de usar titular.getNome()?
- Em UML, a relação entre as duas classes pode ser vista assim:
  - A linha é uma associação (ou relação) entre classes
  - Neste caso, é uma associação de “conhecimento” (ContaSimples3 conhece uma Pessoa)
  - A seta indica a navegabilidade

