Orientação a Objeto - Criação de Classes

Objetivos da Seção

- Aprender a pensar em testes antes de definir novas classes
- 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
- Apresentar "Test-Driven Development" (TDD)

A Primeira Classe

- A primeira classe que escreveremos é uma ContaSimples, que já usamos anteriormente
    - Na realidade, vamos fazer uma versão um pouco diferente, e é por isso que se chama ContaSimples1

A documentação da classe

- A documentação da classe que queremos escrever está aqui

Os testes

- Vamos adotar uma postura chamada "Test-Driven Development" (TDD) e escrever os testes antes de escrrver o código da classe
    - Antes???!!!?
    - Sim, antes, não depois!
- Vamos supor que a classe esteja pronta e que queiramos (foi escrita por alguém) e que queiramos 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
```

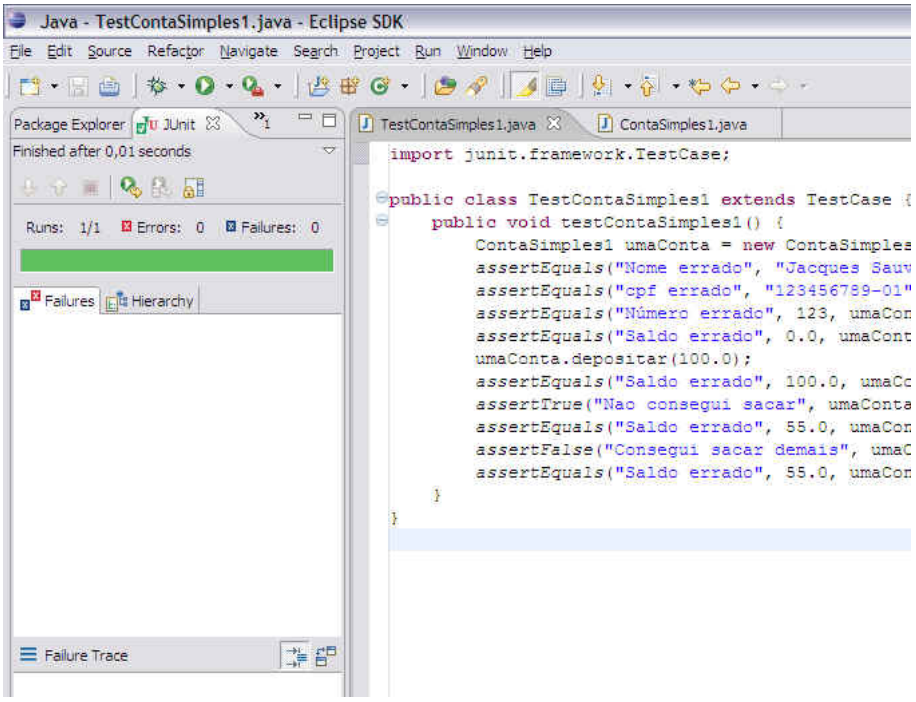
- Veremos adiante que esses testes não estão completos, mas vamos começar com eles
    - Você pode pensar em mais testes que deveriam ser feitos?
- Queremos agora fazer com que os testes sejam realizados de forma *automática*
    - Ter testes automáticos é muito, muito bom
    - Permite que você execute os testes a qualquer momento
        - Você pode repetir os testes centenas de vezes sem custo adicional
        - 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
        - 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
- Os testes estão em TestContaSimples1.java

```
import junit.framework.TestCase;

public class TestContaSimples1 extends TestCase {
        public void testContaSimples1() {
                ContaSimples1 umaConta = new ContaSimples1("Jacques Sauve", "12
                assertEquals("Nome errado", "Jacques Sauve", umaConta.getNome()
                assertEquals("cpf errado", "123456789-01", umaConta.getCPF());
                assertEquals("Número errado", 123, umaConta.getNúmero());
                assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
                umaConta.depositar(100.0);
                assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005)
                assertTrue("Nao consegui sacar", umaConta.sacar(45.0));
                assertEquals("Saldo errado", 55.0, umaConta.getSaldo(), 0.005);
                assertFalse("Consegui sacar demais", umaConta.sacar(56.0));
                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



O programa

- Vamos fazer de conta que a classe ainda não exista (remova-a!) e precisa ser escrita do zero
- Precisamos fazer os testes passar
  - Havendo Eclipse disponível na hora da aula, o professor pode construir a classe aos poucos para fazer os testes passar, um teste de cada vez
- Ver a solução final em ContaSimples1.java

```
/**
 * 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.
 */
public class ContaSimples1 {
  // atributos
  private String  nome;
  private String  cpf;
  private int     número;
  private double  saldo;

  // construtor

  /**
   * Cria uma conta a partir de um nome e cpf de pessoa física, e um número de
   * @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 número) {
    this.nome = nome;
    this.cpf = cpf;
    this.número = número;
    saldo = 0.0;
  }

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

  /**
   * 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 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) {
    // credita a conta
    saldo += valor;
  }

  /**
   * Efetua sacada na conta.
   * @param valor O valor a sacar.
   * @return O sucesso ou não da operação.
   */
  public boolean sacar(double valor) {
    // debita a conta
    if(saldo - valor >= 0) {
      saldo -= valor;
      return true;
    } else {
      return false;
    }
  }

  /**
   * Transforma os dados da conta em um String.
   * @return O string com os dados da conta.
   */
  public String toString() {
    return "numero " + número
           + ", 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) sairam na documentação

A declaração da classe

```
public class ContaSimples1 {
```

- Novas classes são definida com:

```
public class nome {
  corpo da classe
}
```

Atributos

```
  // atributos
  private String  nome;
  private String  cpf;
  private int     número;
  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

```
  // construtor

  /**
   * Cria uma conta a partir de um nome e cpf de pessoa física, e um número de
   * @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 número) {
    this.nome = nome;
    this.cpf = cpf;
    this.número = número;
    saldo = 0.0;
```

```
}
```

- Ao chamar "new ContaSimples(...)", o método construtor da classe é chamado
- Como qualquer método, pode ter parâmetros (aqui tem 3)
    - Porém, o construtor nunca retorna um valor com "return"
- O construtor 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 número da conta.
 * @return O número da conta.
 */
public int getNúmero() {
  return número;
}

/**
 * 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 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) {
  // credita a conta
  saldo += valor;
}

/**
 * Efetua sacada na conta.
 * @param valor O valor a sacar.
 * @return O sucesso ou não da operação.
 */
public boolean sacar(double valor) {
  // debita a conta
  if(saldo - valor >= 0) {
    saldo -= valor;
    return true;
  } else {
    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
    - Neste caso, uma ContaSimples1 deixa que se façam depósitos e saques na Conta
    - Esses métodos definem a interface da classe
    - (Na realidade, *todos* os métodos declarados public fazem parte da interface da classe)

O método toString

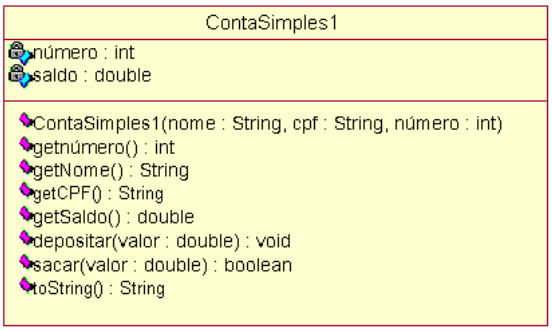
```
/**
 * Transforma os dados da conta em um String.
 * @return O string com os dados da conta.
 */
public String toString() {
  return "numero " + número
         + ", nome " + nome
         + ", saldo " + saldo;
}
```

- 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

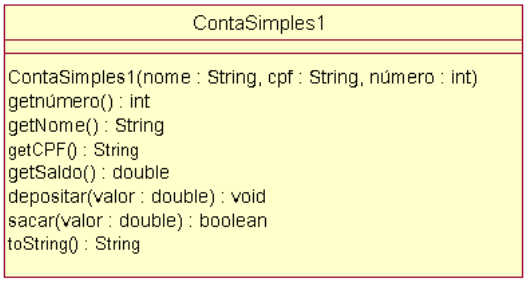
A documentação da classe com UML

- 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)



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



- 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* num programa

Usando a classe que acabamos de definir

- O fato de definir uma classe como ContaSimples1 não significa que haja objetos desta classe em existência: alguém precisa fazer "new ContaSimples1(...)"
- Seguem dois exemplos de programas que usam a classe ContaSimples1
- O primeiro exemplo está em Exemplo1.java

```
/*
 * 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
```

- O exemplo acima é praticamente idêntico ao exemplo Banco1.java
- Observe que o método main está em Exemplo1 e não em ContaSimples1
- O próximo exemplo está em ContaSimples1.java e consiste em colocar o main diretamente na classe que definimos acima

```
/**
 * 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.
 */
public class ContaSimples1 {
  // atributos
  private String  nome;
  private String  cpf;
  private int     número;
  private double  saldo;

  // construtor

  /**
   * Cria uma conta a partir de um nome e cpf de pessoa física, e um número de
   * @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 número) {
    this.nome = nome;
    this.cpf = cpf;
    this.número = número;
    saldo = 0.0;
  }

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

  /**
   * 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 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) {
      // credita a conta
      saldo += valor;
    }

    /**
     * Efetua sacada na conta.
     * @param valor O valor a sacar.
     * @return O sucesso ou não da operação.
     */
    public boolean sacar(double valor) {
      // debita a conta
      if(saldo - valor >= 0) {
        saldo -= valor;
        return true;
      } else {
        return false;
      }
    }

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

- 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.

Palavras adicionais sobre o exemplo

- 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 escodem os detalhes interno 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 e a caixa disponibiliza uma interface (os métodos) para manipular os dados que estão dentro da caixa
  - Isso 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

Vamos falar de testes novamente ...

- Os testes da classe ContaSimples1 não estão completos
- Principalmente, as condições de "exceção" não foram testadas
- Exemplos:
  - Construtor
    - O que ocorre se o nome for nulo ou vazio?
    - O que ocorre se o CPF for nulo ou vazio?
    - O que ocorre se o CPF for inválido?
    - O que ocorre se o número da conta não for positivo?
  - depositar
    - O que ocorre se o valor 0.0 for depositado?
    - O que ocorre se um valor negativo for depositado?
    - Depositar centavos funciona?
    - O que ocorre se depositar frações de centavos?
  - sacar
    - O que ocorre se o valor 0.0 for sacado?
    - O que ocorre se um valor negativo for sacado?
    - Sacar centavos funciona?
    - O que ocorre se sacar frações de centavos?
  - toString
    - toString não foi testado
    - Tem que testar com que valores de saldo?
      - Zero
      - Positivo
      - Com centavos
- Muitos testes são necessários para garantir que tratamos adequadamente de todas as

situações
- E 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.
    - 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
  - A palavra é throw
  - Da mesma forma que "return", "throw" retorna imediatamente do método
  - Diferentemente de "return", "throw" só retorna objetos especiais chamados exceções
  - A exceção pode conter uma mensagem indicando o erro que ocorreu
- "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
- Aqui está um exemplo do uso de exceções para você fuçar

```
package p2.exemplos;

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.getM
                }
        }
        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 co
                }
                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.getMess
                        // 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
- Eis a classe de testes com novos testes
  - Vamos testar a classe ContaSimples3.java que trata erros adequadamente
  - Os testes estão em TestContaSimples3.java

```
import junit.framework.TestCase;

public class TestContaSimples3 extends TestCase {
        public void testBasico() throws Exception {
                ContaSimples3 umaConta = new ContaSimples3("Jacques Sauve",
                                "123456789-01", 123);
                assertEquals("Nome errado", "Jacques Sauve", umaConta.getNome()
                assertEquals("cpf errado", "123456789-01", umaConta.getCPF());
                assertEquals("Número errado", 123, umaConta.getNúmero());
                assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
                umaConta.depositar(100.0);
                assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005)
                assertTrue("Nao consegui sacar", umaConta.sacar(45.0));
                assertEquals("Saldo errado", 55.0, umaConta.getSaldo(), 0.005);
                assertFalse("Consegui sacar demais", umaConta.sacar(56.0));
                assertEquals("Saldo errado", 55.0, umaConta.getSaldo(), 0.005);
        }

        public void testDeErrosNoConstrutor() {
                try {
                        ContaSimples3 umaConta = new ContaSimples3("", "1234567
                        fail("Esperava excecao de nome vazio");
                } catch (Exception ex) {
                        assertEquals("Mensagem errada", "Nome nao pode ser nulo
                                        ex.getMessage());
                }
                try {
                        ContaSimples3 umaConta = new ContaSimples3(null, "12345
                                        123);
                        fail("Esperava excecao de nome nulo");
                } catch (Exception ex) {
                        assertEquals("Mensagem errada", "Nome nao pode ser nulo
                                        ex.getMessage());
                }
                try {
                        ContaSimples3 umaConta = new ContaSimples3("Jacques Sau
                        fail("Esperava excecao de CPF vazio");
                } catch (Exception ex) {
```

```
                assertEquals("Mensagem errada", "CPF nao pode ser nulo
                        ex.getMessage());
        }
        try {
                ContaSimples3 umaConta = new ContaSimples3("Jacques Sau
                        123);
                fail("Esperava excecao de CPF nulo");
        } catch (Exception ex) {
                assertEquals("Mensagem errada", "CPF nao pode ser nulo
                        ex.getMessage());
        }
        try {
                ContaSimples3 umaConta = new ContaSimples3("Jacques Sau
                        "123456789-01", 0);
                fail("Esperava excecao de numero de conta errada");
        } catch (Exception ex) {
                assertEquals("Mensagem errada",
                        "Número da conta deve ser positivo", ex
        }
        try {
                ContaSimples3 umaConta = new ContaSimples3("Jacques Sau
                        "123456789-01", -1);
                fail("Esperava excecao de numero de conta errada");
        } catch (Exception ex) {
                assertEquals("Mensagem errada",
                        "Número da conta deve ser positivo", ex
        }
}

public void testDepositar() throws Exception {
        ContaSimples3 umaConta = new ContaSimples3("Jacques Sauve",
                        "123456789-01", 123);
        assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
        umaConta.depositar(100.0);
        assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005)
        umaConta.depositar(0.0);
        assertEquals("Saldo errado", 100.0, umaConta.getSaldo(), 0.005)
        try {
                umaConta.depositar(-0.01);
                fail("Esperava excecao no deposito");
        } catch (Exception ex) {
                assertEquals("Mensagem errada", "Deposito nao pode ser
                        ex.getMessage());
        }
        umaConta.depositar(0.01);
        assertEquals("Saldo errado", 100.01, umaConta.getSaldo(), 0.005
}

public void testSacar() throws Exception {
        ContaSimples3 umaConta = new ContaSimples3("Jacques Sauve",
                        "123456789-01", 123);
        assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
        assertTrue("Nao consegui sacar", umaConta.sacar(0.0));
        assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
        umaConta.depositar(100.0);
        assertTrue("Nao consegui sacar", umaConta.sacar(23.10));
        assertEquals("Saldo errado", 76.90, umaConta.getSaldo(), 0.005)
        assertFalse("Consegui sacar demais", umaConta.sacar(76.91));
        assertTrue("Nao consegui sacar", umaConta.sacar(76.90));
        assertEquals("Saldo errado", 0.0, umaConta.getSaldo(), 0.005);
```

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

        public void testToString() throws Exception {
                ContaSimples3 umaConta = new ContaSimples3("Jacques Sauve",
                                "123456789-01", 123);
                assertEquals("toString errado",
                                "numero 123, nome Jacques Sauve, saldo 0.0", um
                                                .toString());
                umaConta.depositar(1.23);
                assertEquals("toString errado",
                                "numero 123, nome Jacques Sauve, saldo 1.23", u
                                                .toString());
        }
}
```

- Agora, vamos ver a classe ContaSimples3

```
/**
 * Classe de conta bancária simples, com tratamento de erro
 *
 * @author Jacques Philippe Sauvé, jacques@dsc.ufcg.edu.br
 * @version 1.0 <br>
 *          Copyright (C) 2006 Universidade Federal de Campina Grande.
 */
public class ContaSimples3 {
        // atributos
        private String nome;

        private String cpf;

        private int número;

        private double saldo;

        // construtor
        /**
         * Cria uma conta a partir de um nome e cpf de pessoa física, e um nume
         * conta.
         *
         * @param nome
         *            O nome do titular da conta.
         * @param cpf
         *            O CPF do titular da conta.
         * @param número
         *            O numero da conta.
         * @throws Exception
         *             Se o nome for nulo ou vazio, o CPF for nulo ou vazio ou
         *             conta não for um número positivo
         */
        public ContaSimples3(String nome, String cpf, int número) throws Except
                if (nome == null || nome.equals("")) {
                        throw new Exception("Nome nao pode ser nulo ou vazio");
```

```
        }
        if (cpf == null || cpf.equals("")) {
                throw new Exception("CPF nao pode ser nulo ou vazio");
        }
        if (número <= 0) {
                throw new Exception("Número da conta deve ser positivo"
        }
        this.nome = nome;
        this.cpf = cpf;
        this.número = número;
        saldo = 0.0;
}

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

/**
 * 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 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
 *             Para um deposito negativo
 */
public void depositar(double valor) throws Exception {
        if (valor < 0.0) {
```

```
                throw new Exception("Deposito nao pode ser negativo");
        }
        // credita a conta
        saldo += valor;
}

/**
 * Efetua sacada na conta.
 *
 * @param valor
 *            O valor a sacar.
 * @return O sucesso ou não da operação.
 * @throws Exception
 *             Para um saque negativo
 */
public boolean sacar(double valor) throws Exception {
        if (valor < 0.0) {
                throw new Exception("Saque nao pode ser negativo");
        }
        // debita a conta
        if (saldo - valor >= 0) {
                saldo -= valor;
                return true;
        } else {
                return false;
        }
}

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

// O programa sempre tem um "método" main que é onde começa a execução
public static void main(String args[]) {
        try {
                // Abra uma conta de numero 1 para João com CPF 3091406
                // A conta será "referenciada" com a variável umaConta
                ContaSimples3 umaConta = new ContaSimples3("Joao", "309
                // 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 sa
                                + umaConta.getSaldo());
        } catch (Exception ex) {
                System.err.println(ex.getMessage());
```

```
                    System.exit(1);
                }
        } // fim do método main
}
```

- Você pode ver como a classe é usada no main
  - Rode o programa e veja o que ocorre se alterar o código para causar exceções

A Segunda Classe: Vários Construtores

A classe Pessoa

- Em primeiro lugar, queremos ver a implementação da classe Pessoa que usamos no passado
- Ver a solução em Pessoa.java
- <

  Exercício para casa:
  - Escreva testes para a classe Pessoa

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal de Campina Grande
 *
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 * Não redistribuir sem permissão.
 */
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 implements Serializable {
  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 contrári
   */
  public boolean equals(Object objeto) {
    if(! (objeto instanceof Pessoa)) {
      return false;
    }
    Pessoa outra = (Pessoa)objeto;
    return getNome().equals(outra.getNome())
            && getCPF().equals(outra.getCPF());
  }
}
```

- Esquece, por enquanto da palavra "Serializable"

        - Por enquanto, significa apenas que queremos gravar objetos dessa classe em arquivo
    - A linha com "package" diz a qual pacote pertence a classe
        - Falaremos mais disso adiante
    - 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 p1 = new Pessoa("joão", "12345678901");
Pessoa p2 = new Pessoa("joão");
```

- 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)
    - Não entenderemos toda a implementação do método equals() agora, mas pelo menos, 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 ContaSimples2.java

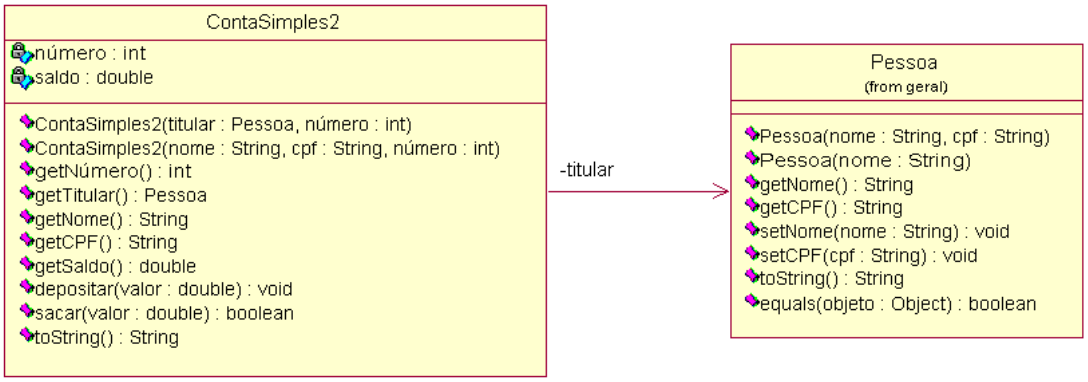
```
/**
 * Classe de conta bancária simples.
 *
 * @author   Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 */
```

```
import p1.aplic.geral.*;

public class ContaSimples2 {
  // atributos
  private int        número;
  private Pessoa     titular;
  private double     saldo;

  // construtores
  /**
   * Cria uma conta a partir de uma pessoa e número de conta.
   * @param titular O titular da conta.
   * @param número O número da conta.
   */
  public ContaSimples2(Pessoa titular, int número) {
    this.número = número;
    this.titular = titular;
    saldo = 0.0;
  }

  /**
   * Cria uma conta a partir de um nome e cpf de pessoa física, e um númer
   * @param nome O nome do titular da conta.
   * @param cpf O CPF do titular da conta.
   * @param número O número da conta.
   */
  // há sobrecarga do método construtor
  public ContaSimples2(String nome, String cpf, int número) {
    // aninhamento de método construtor
    this(new Pessoa(nome, cpf), número);
  }

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

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

  /**
   * Recupera o nome do titular da conta.
   * @return O nome do titular da conta.
   */
  /* feito para ajudar os principiantes escondendo a classe Pessoa no inic
  public String getNome() {
    // aninhamento de método
    return titular.getNome();
  }

  /**
```

```
     * Recupera o CPF do titular da conta.
     * @return O CPF do titular da conta.
     */
    /* feito para ajudar os principiantes escondendo a classe Pessoa no inic
    public String getCPF() {
      return titular.getCPF();
    }

    /**
     * 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) {
      // credita a conta
      saldo += valor;
    }

    /**
     * Efetua sacada na conta.
     * @param valor O valor a sacar.
     * @return O sucesso ou não da operação.
     */
    public boolean sacar(double valor) {
      // debita a conta
      if(saldo - valor >= 0) {
        saldo -= valor;
        return true;
      } else {
        return false;
      }
    }

    /**
     * Transforma os dados da conta em um String.
     * @return O string com os dados da conta.
     */
    public String toString() {
      return "numero " + número
              + ", 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" (ContaSimples2 conhece uma Pessoa)
- A seta indica a navegabilidade



Mais exemplos

- Vamos ver a implementação de duas classes que já usamos: Carta e Baralho
- A solução está em Carta.java e Baralho.java

Os testes da classe Carta

- Eis alguns testes que indicam como queremos que a classe Carta de comporte

```
public class TestaCarta extends TestCase {
        private Carta asPaus;
        private Carta asCopas;
        private Carta reiPaus;
        private Carta menorCarta;
        private Carta maiorCarta;

        private void setUp() {
                asPaus = new Carta(Carta.AS, Carta.PAUS);
                asCopas = new Carta(Carta.AS, Carta.COPAS);
                reiPaus = new Carta(Carta.REI, Carta.PAUS);
                menorCarta = new Carta(Carta.menorValor(), Carta.PAUS);
                maiorCarta = new Carta(Carta.maiorValor(), Carta.PAUS);
        }

        public void testEquals() {
                assertTrue(!asPaus.equals(null));
                assertTrue(!asPaus.equals(new Baralho()));
                assertEquals(asPaus, asPaus);
                assertEquals(new Carta(Carta.AS, Carta.PAUS), asPaus);
                assertTrue(!asPaus.equals(asCopas));
                assertTrue(!asPaus.equals(reiPaus));
        }

        public void testMenor() {
                assertEquals(asPaus, menorCarta);
        }

        public void testMaior() {
                assertEquals(reiPaus, maiorCarta);
        }

        public void testCompareTo() {
                assertEquals("1", 0, asPaus.compareTo(asPaus));
```

```
                assertEquals("2", 0, asPaus.compareTo(new Carta(Carta.AS,
                assertEquals("3", 0, asPaus.compareTo(asCopas));
                assertTrue("4", asPaus.compareTo(reiPaus) < 0);
                assertTrue("5", reiPaus.compareTo(asPaus) > 0);
        }

        public void testToString() {
                assertTrue(asPaus.toString().equals("AS de PAUS"));
        }
}
```

A classe Carta

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal de Campina Grande
 *
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 * Não redistribuir sem permissão.
 */

package p1.aplic.cartas;

/**
 * Uma carta de um baralho comum.
 * Num baralho comum, tem 52 cartas: 13 valores (AS, 2, 3, ..., 10, valete
 * de 4 naipes (ouros, espadas, copas, paus).
 * Cartas podem ser criadas, comparadas (quanto a seu valor), etc.
 *
 * @author Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 */
public class Carta {
  /**
   * Valor da carta AS. Usado para construir uma carta: new Carta(Carta.AS
   */
  public static final int AS = 1;
  /**
   * Valor da carta VALETE. Usado para construir uma carta: new Carta(Cart
   */
  public static final int VALETE = 11;
  /**
   * Valor da carta DAMA. Usado para construir uma carta: new Carta(Carta.
   */
  public static final int DAMA = 12;
  /**
   * Valor da carta REI. Usado para construir uma carta: new Carta(Carta.R
   */
  public static final int REI = 13;

  /**
   * Valor do naipe de PAUS. Usado para construir uma carta: new Carta(Car
   */
  public static final int PAUS = 0;
  /**
```

```
 * Valor do naipe de OUROS. Usado para construir uma carta: new Carta(Ca
 */
public static final int OUROS = 1;
/**
 * Valor do naipe de COPAS. Usado para construir uma carta: new Carta(Ca
 */
public static final int COPAS = 2;
/**
 * Valor do naipe de ESPADAS. Usado para construir uma carta: new Carta(
 */
public static final int ESPADAS = 3;

private int  valor;
private int  naipe;

/**
 * Construtor de uma carta comum.
 * @param valor O valor da carta (AS, 2, 3, ..., 10, VALETE, DAMA, REI).
 * @param naipe O naipe da carta (PAUS, OUROS, COPAS, ESPADAS).
 */
public Carta(int valor, int naipe) {
  this.valor = valor;
  this.naipe = naipe;
}

/**
 * Recupera o valor da carta.
 * @return O valor da carta.
 */
public int getValor() {
  return valor;
}

/**
 * Recupera o naipe da carta.
 * @return O naipe da carta.
 */
public int getNaipe() {
  return naipe;
}

/**
 * Recupera o valor da menor carta deste tipo que pode ser criada.
 * É possível fazer um laço de menorValor() até maiorValor() para varrer
 * todos os valores possíveis de cartas.
 * @return O menor valor.
 */
public static int menorValor() {
    return AS;
}

/**
 * Recupera o valor da maior carta deste tipo que pode ser criada.
 * É possível fazer um laço de menorValor() até maiorValor() para varrer
 * todos os valores possíveis de cartas.
 * @return O maior valor.
 */
public static int maiorValor() {
    return REI;
}
```

```
/**
 * Recupera o "primeiro naipe" das cartas deste tipo.
 * Ser "primeiro naipe" não significa muita coisa, já que naipes não tem
 * (um naipe não é menor ou maior que o outro).
 * Fala-se de "primeiro naipe" e "último naipe" para poder
 * fazer um laço de primeiroNaipe() até últimoNaipe() para varrer
 * todos os naipes possíveis de cartas.
 * @return O primeiro naipe.
 */
public static int primeiroNaipe() {
    return PAUS;
}

/**
 * Recupera o "último naipe" das cartas deste tipo.
 * Ser "último naipe" não significa muita coisa, já que naipes não tem v
 * (um naipe não é menor ou maior que o outro).
 * Fala-se de "primeiro naipe" e "último naipe" para poder
 * fazer um laço de primeiroNaipe() até últimoNaipe() para varrer
 * todos os naipes possíveis de cartas.
 * @return O primeiro naipe.
 */
public static int últimoNaipe() {
    return ESPADAS;
}

/**
 * Compare esta carta a outra.
 * @param outra A carta a comparar a esta.
 * @return Zero se forem iguais. Um valor < 0 se a carta for menor que a
 * Um valor > 0 se a carta for maior que a outra carta.
 */
public int compareTo(Carta outra) {
    return getValor() - outra.getValor();
}

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

 private static final String[] nomeDeCarta = {
  "", // queremos sincronizar o valor da carta e seu indice (AS == 1, et
  "AS",
  "DOIS",
  "TRES",
  "QUATRO",
  "CINCO",
  "SEIS",
  "SETE",
```

```
    "OITO",
    "NOVE",
    "DEZ",
    "VALETE",
    "DAMA",
    "REI",
  };

  private static final String[] nomeDeNaipe = {
    "PAUS",
    "OUROS",
    "COPAS",
    "ESPADAS",
  };

  /**
   * Representa a carta como String.
   * @return Um string representando a carta.
   */
  public String toString() {
    return nomeDeCarta[getValor()] + " de " + nomeDeNaipe[getNaipe()];
  }
}
```

- Verifique a definição de certas cartas (AS, REI, ...) e a definição dos naipes
  - São constantes simbólicas
  - Por convenção, usamos letras maiúsculas para constantes simbólicas
  - A palavra "final" diz que são constantes (o valor é final e não pode mudar)
  - A palavra "static" diz que isso pertence à classe e não precisa ser armazenado para cada objeto da classe
    - Observe que uso o nome da classe antes do "." e não uma referência a um objeto
    - Isso é por causa do "static"
  - A palavra "public" siz que posso usar Carta.AS, Carta.OUROS, mesmo fora da classe
- Poderíamos ter usado um String para os naipes, em vez de usar int, certo?
  - Qual você prefere?
  - O que mudaria no programa?
- Quais são os atributos de cada objeto da classe Carta?
- O método menorValor() também é static
  - Ele só acessa informaão estática, então pode ser static
  - Serve que que eu possa fazer Carta.MenorValor() e saber o menor valor que existe nas Cartas sem ter que instanciar um objeto primeiro
  - Chamamos isso de método de classe, ou método estático
  - Embora tenham seu lugar, métodos de classe devem ser evitados
- O método compareTo() existe em muitas classes e serve para poder comparar dois objetos
  - Vamos supor que eu tenha duas Cartas que peguei de uma Baralho
  - As Cartas têm referências suaCarta e minhaCarta
  - Então posso fazer o seguinte dentro de um programa:

```
if(suaCarta.compareTo(minhaCarta) > 0) {
  System.out.println("Voce ganha.");
  suasVitórias++;
} else if(suaCarta.compareTo(minhaCarta) < 0) {
  System.out.println("Eu ganho.");
  minhasVitórias++;
} else {
  System.out.println("Empate.");
}
```

- Observe como o método toString() é simples
  - São os arrays de nomes que simplificam tudo
  - Como você teria feito? Com muito código usando if-else ou switch??
- Como teste de conhecimento, o que ocorreria se os arrays nomeDeCarta e nomeDeNaipe não fossem estáticos?

Os testes da classe Baralho

- Primeiro, vamos ver o que Baralho promete oferecer na sua interface:
  - Clique aqui e veja o construtor e os métodos prometidos
- Agora, os testes ...

```
public class TestaBaralho extends TestCase {
        private Baralho b1; // fica intacto

        private void setUp() {
                b1 = new Baralho();
        }

        public void testNúmeroDeCartas() {
                assertEquals(1, b1.menorValor());
                assertEquals(13, b1.maiorValor());
                assertEquals(52, b1.númeroDeCartas());
        }

        public void testBaralhoNovo() {
                assertTrue(baralhoEstáCompleto(b1));
        }

        public void testBaralhar() {
                Baralho b2 = new Baralho();
                b2.baralhar();
                assertTrue(baralhoEstáCompleto(b2));
        }

        private boolean baralhoEstáCompleto(Baralho b) {
                List cartasJáVistas = new ArrayList();
                Iterator it = b.iterator();
                while (it.hasNext()) {
                        Carta c = (Carta) it.next();
                        // vê se carta está ok
                        int v = c.getValor();
                        int n = c.getNaipe();
                        assertTrue("Valor não ok", v >= c.menorValor()
                                        && v <= c.maiorValor());
                        assertTrue("Naipe não ok", n >= c.primeiroNaipe()
                                        && n <= c.últimoNaipe());
                        assertTrue("Carta já vista", !cartasJáVistas.conta
                        cartasJáVistas.add(c);
                }
                return cartasJáVistas.size() == 52;
        }

        public void testPegaCarta() {
                List cartasJáVistas = new ArrayList();
                Baralho b3 = new Baralho();
                Carta c;
                while ((c = b3.pegaCarta()) != null) {
                        // vê se carta está ok
                        int v = c.getValor();
```

```
                int n = c.getNaipe();
                assertTrue("Valor não ok", v >= c.menorValor()
                                && v <= c.maiorValor());
                assertTrue("Naipe não ok", n >= c.primeiroNaipe()
                                && n <= c.últimoNaipe());
                assertTrue("Carta já vista", !cartasJáVistas.conta
                cartasJáVistas.add(c);
            }
            assertEquals("Baralho não vazio", 0, b3.númeroDeCartas());
        }
}
```

A classe Baralho

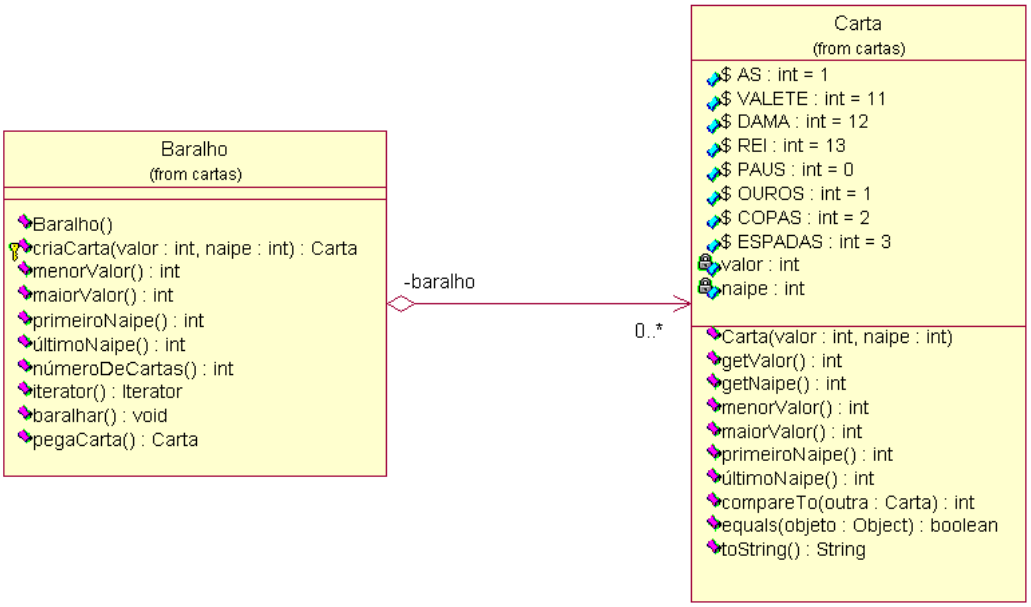
- Primeiro, vamos ver o que Baralho promete oferecer na sua interface:
  - Clique aqui e veja o construtor e os métodos prometidos
- Agora, vejamos como implementar Baralho.java

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal de Campina Grande
 *
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 * Não redistribuir sem permissão.
 */

package p1.aplic.cartas;

import java.util.*;
import java.lang.Math.*;

/**
 * Um baralho comum de cartas.
 * Num baralho comum, tem 52 cartas: 13 valores (AS, 2, 3, ..., 10, valete
 * de 4 naipes (ouros, espadas, copas, paus).
 *
 * @author Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 */
public class Baralho {
  /**
   * O baralho é armazenado aqui.
   */
  private List baralho;

  /**
   * Construtor de um baralho comum.
   */
  public Baralho() {
    // Usa uma List para ter um iterador facilmente
    baralho = new ArrayList();
    // enche o baralho
    for(int valor = menorValor(); valor <= maiorValor(); valor++) {
      for(int naipe = primeiroNaipe(); naipe <= últimoNaipe(); naipe++) {
        baralho.add(new Carta(valor, naipe));
      }
```

```
    }
  }

  /**
   * Recupera o valor da menor carta possível deste baralho.
   * É possível fazer um laço de menorValor() até maiorValor() para varrer
   * todos os valores possíveis de cartas.
   * @return O menor valor.
   */
  public int menorValor() {
    return Carta.menorValor();
  }

  /**
   * Recupera o valor da maior carta possível deste baralho.
   * É possível fazer um laço de menorValor() até maiorValor() para varrer
   * todos os valores possíveis de cartas.
   * @return O maior valor.
   */
  public int maiorValor() {
    return Carta.maiorValor();
  }

  /**
   * Recupera o "primeiro naipe" das cartas que podem estar no baralho.
   * Ser "primeiro naipe" não significa muita coisa, já que naipes não tem
   * (um naipe não é menor ou maior que o outro).
   * Fala-se de "primeiro naipe" e "último naipe" para poder
   * fazer um laço de primeiroNaipe() até últimoNaipe() para varrer
   * todos os naipes possíveis de cartas.
   * @return O primeiro naipe.
   */
  public int primeiroNaipe() {
    return Carta.primeiroNaipe();
  }

  /**
   * Recupera o "último naipe" das cartas que podem estar no baralho.
   * Ser "último naipe" não significa muita coisa, já que naipes não tem v
   * (um naipe não é menor ou maior que o outro).
   * Fala-se de "primeiro naipe" e "último naipe" para poder
   * fazer um laço de primeiroNaipe() até últimoNaipe() para varrer
   * todos os naipes possíveis de cartas.
   * @return O primeiro naipe.
   */
  public int últimoNaipe() {
    return Carta.últimoNaipe();
  }

  /**
   * Recupera o número de cartas atualmente no baralho.
   * @return O número de cartas no baralho.
   */
  public int númeroDeCartas() {
    return baralho.size();
  }

  /**
   * Baralha (traça) o baralho.
   */
```

```
  public void baralhar() {
    int posição;
    for(posição = 0; posição < númeroDeCartas() - 1; posição++) {
      // escolhe uma posição aleatória entre posição e númeroDeCartas()-1
      int posAleatória = posição + (int)((númeroDeCartas() - posição) * Ma
      // troca as cartas em posição e posAleatória
      Carta temp = (Carta)baralho.get(posição);
      baralho.set(posição, baralho.get(posAleatória));
      baralho.set(posAleatória, temp);
    }
  }

  /**
   * Retira uma carta do topo do baralho e a retorna. A carta é removida d
   * @return A carta retirada do baralho.
   */
  public Carta pegaCarta() {
    if(númeroDeCartas() == 0) return null;
    return (Carta)baralho.remove(númeroDeCartas()-1);
  }
}
```

- Observações sobre a classe Baralho
  - Estude como o Construtor funciona
    - Os dois "for" poderiam ser escritos assim também:
      for(int valor = AS; valor <= REI; valor++) {
      for(int naipe = PAUS; naipe <= ESPADAS; naipe++) {
  - Veja como menorValor() usa menorValor() da Carta
    - O menor valor de um Baralho cheio é o menor valor das Cartas que compõem o baralho, certo?
  - Veja a implementação da método númeroDeCartas()
  - Veja a implementação da método iterator()
  - Veja o uso de variáveis locais: valor e posição
  - Veja a implementação da método baralhar(): é muito instrutivo
    - Math.random() retorna um número randômico (ou pseudo-randômico) na faixa [0,1)
  - Veja a implementação da método pegaCarta()
    - É bom ter uma forma de avisar que não sobrou nada no Baralho, certo
    - Claro que é bom que o chamador verifique isso!!
    - Já vimos o que ocorre quando pegaCarta() é chamado com Baralho vazio (uma exceção)
- Em UML, temos o seguinte:
  - O losango chama-se agregação (um Baralho "possui" Cartas)
    - Usado para indicar uma relação de "todo-parte"
  - o..* chama-se a cardinalidade e indica que um baralho pode ter 0 ou mais Cartas



A classe MaiorCarta

- Se tiver tempo, estudar MaiorCarta em aula, senão o alunno estuda em casa
- Já vimos este jogo antes
- O programa está em MaiorCarta.java

```
/*
 * Desenvolvido para a disciplina Programacao 1
 * Curso de Bacharelado em Ciência da Computação
 * Departamento de Sistemas e Computação
 * Universidade Federal de Campina Grande
 *
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 * Não redistribuir sem permissão.
 */

package p1.aplic.cartas;

import java.util.*;

/**
 * Um jogo de cartas simples.
 * Cada jogador recebe uma carta do baralho.
 * A maior carta ganha.
 * Repete para cada rodada.
 *
 * @author Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.0
 * <br>
 * Copyright (C) 1999 Universidade Federal de Campina Grande.
 */
public class MaiorCarta {
  private int     suasVitórias;  // pontuação
  private int     minhasVitórias;
  private Baralho baralho;

  /**
   * Construtor do jogo.
   */
   public MaiorCarta() {
    suasVitórias = 0;
```

```
    minhasVitórias = 0;
    baralho = new Baralho();
    baralho.baralhar();
  }

  /**
   * Joga o jogo de Maior Carta.
   * @param rodadas O número de rodadas a jogar.
   */
  public void joga(int rodadas) {
    for(int i = 0; i < rodadas; i++) {
      Carta suaCarta = baralho.pegaCarta();
      System.out.print("Sua carta: " + suaCarta + " ");
      Carta minhaCarta = baralho.pegaCarta();
      System.out.print("Minha carta: " + minhaCarta + " ");
      if(suaCarta.compareTo(minhaCarta) > 0) {
        System.out.println("Voce ganha.");
        suasVitórias++;
      } else if(suaCarta.compareTo(minhaCarta) < 0) {
        System.out.println("Eu ganho.");
        minhasVitórias++;
      } else {
        System.out.println("Empate.");
      }
    }
    System.out.println("Voce ganhou " + suasVitórias +
      " vezes, eu ganhei " + minhasVitórias + " vezes, " +
      (rodadas-suasVitórias-minhasVitórias) + " empates.");
  }
}
```

Jogando o jogo ...

- Seguer um programa simples que joga o jogo MaiorCarta
- A colução está em Exemplo2.java

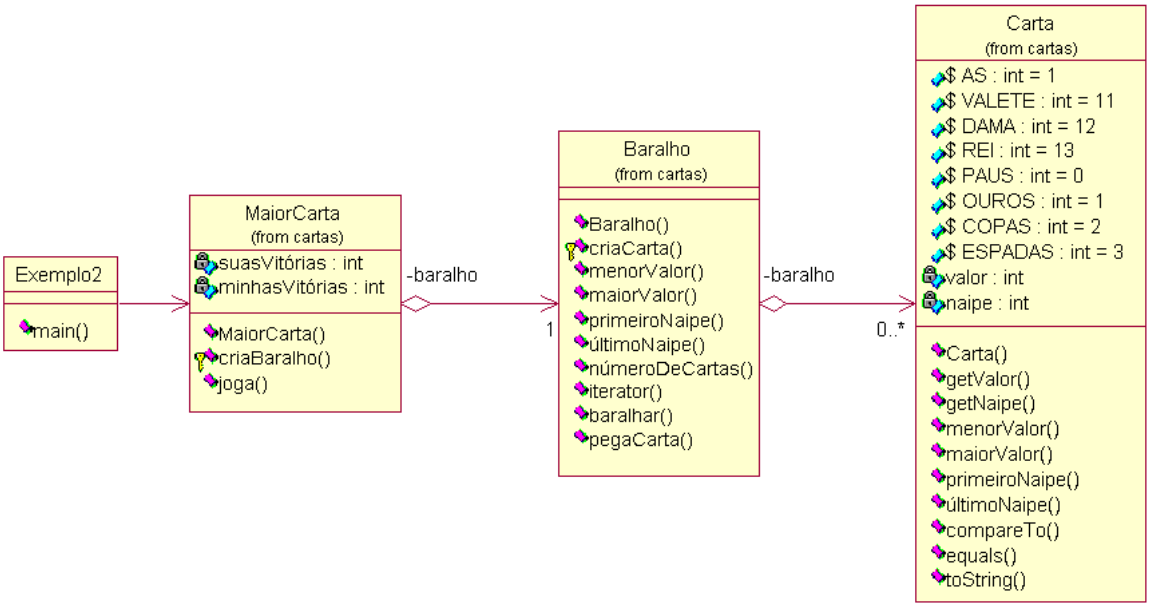
```
import p1.aplic.cartas.*;

public class Exemplo2 {
  public static void main(String args[]) {
    (new MaiorCarta()).joga(15);
  }
}
```

- A saída do programa:

```
Sua carta: CINCO de PAUS Minha carta: DEZ de COPAS Eu ganho.
Sua carta: DEZ de OUROS Minha carta: SETE de COPAS Voce ganha.
Sua carta: DAMA de PAUS Minha carta: DEZ de PAUS Voce ganha.
Sua carta: DOIS de PAUS Minha carta: QUATRO de ESPADAS Eu ganho.
Sua carta: TRES de ESPADAS Minha carta: OITO de ESPADAS Eu ganho.
Sua carta: DAMA de COPAS Minha carta: TRES de OUROS Voce ganha.
Sua carta: TRES de PAUS Minha carta: VALETE de OUROS Eu ganho.
Sua carta: AS de PAUS Minha carta: DAMA de ESPADAS Eu ganho.
Sua carta: REI de COPAS Minha carta: QUATRO de OUROS Voce ganha.
Sua carta: CINCO de OUROS Minha carta: DEZ de ESPADAS Eu ganho.
Sua carta: AS de ESPADAS Minha carta: VALETE de PAUS Eu ganho.
Sua carta: SETE de OUROS Minha carta: DOIS de OUROS Voce ganha.
Sua carta: SEIS de ESPADAS Minha carta: CINCO de ESPADAS Voce ganha.
Sua carta: REI de PAUS Minha carta: SEIS de PAUS Voce ganha.
```

```
Sua carta: REI de ESPADAS Minha carta: CINCO de COPAS Voce ganha.
Voce ganhou 8 vezes, eu ganhei 7 vezes, 0 empates.
```

- Veja como objeto de uma classe usam objetos de outras classe e assim sucessivamente
  - Como exercício, trace um diagrama dos relacionamentos entre classes e entre objetos no mundo das Cartas
- Em UML, temos:



- Exercício para a sala de aula: nos testes de Carta e Baralho, não faltou testar condições de erro? Quais?

oo-3 programa anterior próxima