Programação Orientada a Objetos I

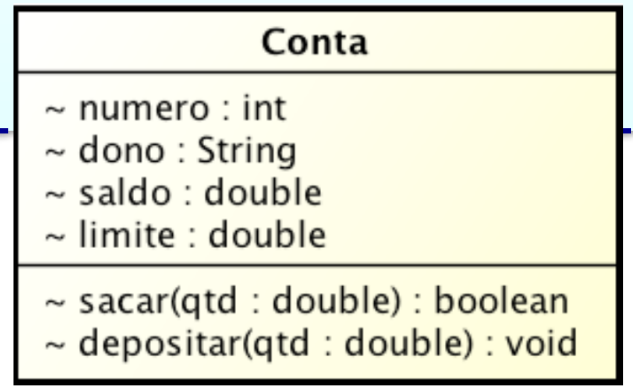
CÁSSIO CAPUCHO PEÇANHA – 07

Manipulação de Objetos

Regras de negócio

- Para uma conta corrente:

```
class Conta {
  int numero;
  String dono;
  double saldo;
  double limite;
  // ...
}
```



RN1: quando negativo, o valor absoluto do saldo não pode ser superior ao do limite.

Implementando a regra de negócio

```
class Conta {
  // Restante da classe...

  void sacar(double qtd) {
     double novoSaldo = this.saldo - qtd;
     this.saldo = novoSaldo;
  }
}

public class TesteConta {
  public static void main(String[] args) {
     Conta c = new Conta();
     c.saldo = 1000.0;
     c.limite = 1000.0;
     c.sacar(5000);       // Vai gerar inconsistência!
  }
}
```

Implementando a regra de negócio

```
class Conta {
  // Restante da classe...

  void sacar(double qtd) {
     double novoSaldo = this.saldo - qtd;
     this.saldo = novoSaldo;
  }
}

public class TesteConta {
  public static void main(String[] args) {
     Conta c = new Conta();
     c.saldo = 1000.0;
     c.limite = 1000.0;
     c.sacar(5000);       // Vai gerar inconsistência!
  }
}
```

Implementando a regra de negócio

```
public class TesteConta {
  public static void main(String[] args) {
     Conta c = new Conta();
     c.saldo = 1000.0;
     c.limite = 1000.0;

     // Vamos verificar antes de sacar...
     double valorASacar = 5000.0;
     if (valorASacar < c.saldo + c.limite)
        c.sacar(valorASacar);
  }
}
```

A responsabilidade está com a classe certa?

Implementando a regra de negócio

Conta

```
boolean sacar(double qtd) {
   double novoSaldo = this.saldo - qtd;
   if (novoSaldo >= -limite) {
      this.saldo = novoSaldo;
      return true;
   }
   else return false;
}
```

TesteConta

```
   Conta c = new Conta();
   c.saldo = 1000.0;
   c.limite = 1000.0;

   // Agora sim!
   if (c.sacar(5000)) System.out.println("Consegui");
   else System.out.println("Não deu...");

   c.saldo = -3000.0;  // Só que não...
```

Encapsulamento

- Permitir o acesso direto aos atributos:
  - Exige disciplina dos clientes da classe Conta;
  - Pode levar a inconsistências;
- Solução: impedir o acesso externo ao atributo:
  - Atributo privativo;
  - Externo = qualquer outra classe, exceto a proprietária do atributo
    - (ex.: Conta para o atributo saldo).
- Vantagens:
  - Objetos trocam mensagens com base em contratos;
  - Modificações na implementação não afetam clientes
    - (ex.: adicionar CPMF nos saques de conta-corrente).

Encapsulamento

- Usamos objetos sem saber seu funcionamento interno;
- Assim também deve ser em nossos sistemas OO:
  - Maior manutenibilidade;
  - Maior reusabilidade.

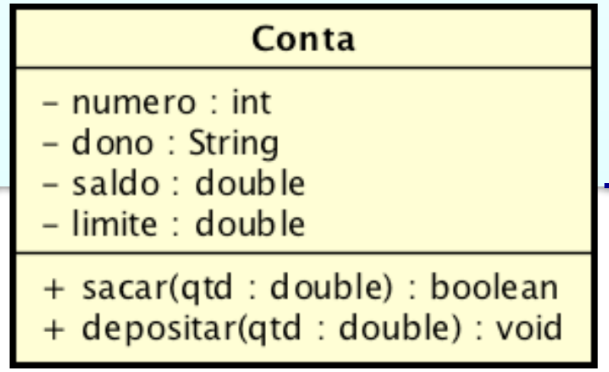
Encapsulamento

```
class Conta {
  private int numero;
  private String dono;
  private double saldo;
  private double limite;

  public boolean sacar(double qtd) {
    // ...
  }

  // . . .
}
```

Modificador de acesso / visibilidade!



Modificadores de acesso

- Determinam a visibilidade de um determinado membro da classe com relação a outras classes;
- Há quatro níveis de acesso:
  - Público (**public**);
  - Privado/privativo (**private**);
  - Protegido (**protected**);
  - Amigo ou privativo ao pacote (friendly ou **packageprivate**).

Modificadores de acesso

- Três palavras-chave especificam o acesso:
  - public
  - private
  - protected
- O nível de acesso package-private é determinado pela ausência de especificador;
- Devem ser usadas antes do nome do membro que querem especificar;
- Não podem ser usadas em conjunto.

Modificadores de acesso

Acesso	Público	Protegido	Amigo	Privado
A própria classe	Sim	Sim	Sim	Sim
Classe no mesmo pacote	Sim	Sim	Sim	Não
Subclasse em pacote diferente	Sim	Sim	Não	Não
Não-subclasse em pacote diferente	Sim	Não	Não	Não

Encapsulamento

Conta

```
class Conta {
  // ...
  private double saldo;

  public boolean sacar(double qtd) {
    // ...
  }
}
```

TesteConta

```
    Conta c = new Conta();
    c.depositar(1000.0);
    c.saldo = -3000.0;

    // Não compila!
    // error: saldo has private access in Conta
    // c.saldo = -3000.0;
    //  ^
```

Estrutura interna fica inacessível.

Interface do objeto é pública.

Mas e se eu precisar acessar um atributo? GET/SET

- Atributos devem ser privativos;
- Se precisarem ser lidos ou alterados, prover métodos **get/set**.

```
public class Cliente {
  // ...
  private String nome;

  public String getNome() {
     return nome;
  }

  public void setNome(String nome) {
     this.nome = nome;
  }
}
```

IDEs geram esses métodos automaticamente.

Planeje bem a interface da classe

- É importante observar que:
  - O método getAtr() não tem que necessariamente retornar apenas o atributo atr;
  - Não crie automaticamente métodos get/set para todos os atributos! Para alguns não faz sentido...

```
// O limite faz parte do saldo (só que cobra juros)!
public double getSaldo() {
   return saldo + limite;
}

// Não há método para mudar o saldo. Tem que sacar()
// public void setSaldo(double saldo)
```

Inicialização

- Neologismo criado para indicar tarefas que devem ser efetuadas ao iniciarmos algo;
- Quando criamos objetos, podemos querer inicializá-lo com alguns valores;
- Poderíamos criar um método para isso:

```
class Aleatorio {
  private int numero;

  public void inicializar() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
}
```

Construtores

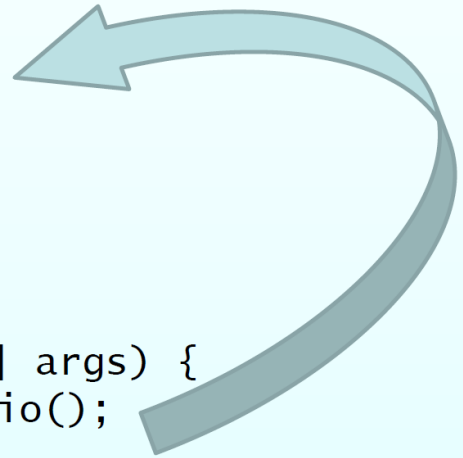
- Problema do método inicializar(): podemos esquecer de chamá-lo!
- Por isso, Java provê o mecanismo de construtores:
- São chamados automaticamente pelo Java quando um objeto novo é criado;
- Construtores não tem valor de retorno e possuem o mesmo nome da classe.

Construtores

- Quando um novo objeto é criado:
  1. é alocada memória para o objeto;
  2. o construtor é chamado.

```
class Aleatorio {
  private int numero;
  public Aleatorio() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
}

public class Teste {
  public static void main(String[] args) {
     Aleatorio aleat = new Aleatorio();
  }
}
```



Construtores podem ter argumentos

- Se definidos argumentos, devem ser passados na criação do objeto com new:

```
class Aleatorio {
  private int numero;
  public Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
}

public class Teste {
  public static void main(String[] args) {
     Aleatorio aleat1 = new Aleatorio(20);
     Aleatorio aleat2 = new Aleatorio(50);
  }
}
```

Pode haver múltiplos construtores

- Nossas classes podem ter quantos construtores quisermos (com argumentos diferentes):

```
class Aleatorio {
  private int numero;
  public Aleatorio() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
  public Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
}
public class Teste {
  public static void main(String[] args) {
     Aleatorio aleat1 = new Aleatorio();
     Aleatorio aleat2 = new Aleatorio(50);
  }
}
```

Atributos independentes de objetos

- Vimos até agora que atributos pertencem aos objetos:
- Não se faz nada sem antes criar um objeto (new)!
- No entanto, há situações que você quer usá-los sem ter que criar objetos:

```
public class TesteConta {
  public static void main(String[] args) {
    int qtdContas = 0;
    Conta c1 = new Conta();
    qtdContas++;

    Conta c2 = new Conta();
    qtdContas++;
    // ...
  }
}
```

A responsabilidade está com a classe certa?

Atributos independentes de objetos

- Se acertamos a responsabilidade, voltamos a depender de um objeto para usar o atributo:

```
class Conta {
  // ...
  public int qtdContas = 0;

  public Conta() {
     qtdContas++;     // Outras inicializações...
  }
}
```

```
    Conta c1 = new Conta();
    Conta c2 = new Conta();

    // Quantas contas foram criadas?
    System.out.println(c2.qtdContas);
```

TesteConta

Atributos static

- Usando a palavra-chave **static** você define um atributo de classe (“estático”):
- Pertence à classe como um todo;
- Pode-se acessá-los mesmo sem ter criado um objeto;
- Objetos podem acessá-los como se fosse um membro de objeto, só que compartilhado.

```
class Conta {
  // ...
  public static int qtdContas = 0;

  public Conta() {
    qtdContas++;     // Outras inicializações...
  }
}
```

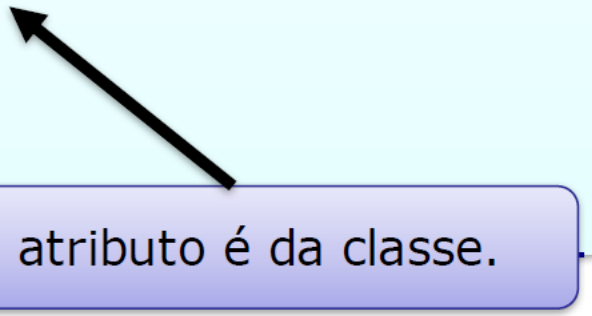
Acesso a atributos static

- Não precisamos mais de um objeto pra acessar:

```
public class TesteConta {
  public static void main(String[] args) {
     Conta c1 = new Conta();
     Conta c2 = new Conta();

     System.out.println(Conta.qtdContas);

     // ...
  }
}
```



Acesso a atributos static

- Se acertamos a visibilidade do atributo, precisamos então de um método para acessá-lo:

```
class Conta {
  // ...
  private static int qtdContas = 0;

  public Conta() {
     qtdContas++;     // Outras inicializações...
  }

  public int getQtdContas() {
     return qtdContas;
  }
}
```

Mas não sendo `static`, vou precisar de um objeto pra acessar o atributo de novo!?!

Métodos static

- Métodos também podem ser static:

```
class Conta {
  // ...
  private static int qtdContas = 0;

  public Conta() { qtdContas++; /* ... */ }

  public static int getQtdContas() {
     return qtdContas;
  }
}
```

```
    Conta c1 = new Conta();
    Conta c2 = new Conta();

    // Quantas contas foram criadas?
    System.out.println(Conta.getQtdContas());
```

TesteConta

Contexto static

- Métodos static não podem acessar membros não-static:

```
public class Teste {
  private int atributo;
  private static int atributoStatic;

  public void metodo() { }

  public static void metodoStatic() { }

  public static void main(String[] args) {
    // ...
    System.out.println(atributoStatic);
    System.out.println(Teste.atributoStatic);
    metodoStatic();
    Teste.metodoStatic();
    // Não pode:
    // System.out.println(atributo);
    // System.out.println(Teste.atributo);
    // metodo();
    // Teste.metodo();

    // Preciso de um objeto:
    Teste t = new Teste();
    System.out.println(t.atributo);
    t.metodo();
  }
}
```