Computação II - Java Prof. Adriano Joaquim de Oliveira Cruz
Aula Prática - Herança, Polimorfismo e Construtores

---

1 Exercícios

Exercício 1: Construtores e Testes

Analise a classe `Conta.java`, mostrada na listagem 1 e escreva uma classe que a teste usando todos os métodos da classe. Observe que nesta listagem aparecem construtores. Estes podem ser definidos como:

> **Construtores:** Uma classe pode conter construtores que são chamados para criar os objetos a partir da classe. Declarações de construtores parecem com declarações de métodos, exceto que eles usam o nome da classe e não têm tipo de retorno.

Em java não é necessário fornecer construtores para suas classes, mas você deve ser cuidadoso quando escreve uma classe sem eles. O compilador fornece automaticamente o construtor padrão, sem argumentos para qualquer classe sem construtores. Este construtor default chamará o construtor sem argumentos da sua superclasse. Caso sua classe não tenha uma superclasse explícita, o compilador usará como superclasse Object, que tem um construtor sem argumentos.

Você pode usar modificadores de acesso na declaração de um construtor para controlar que outras classes podem chamar o construtor.

Abaixo mostramos um possível exemplo de interação entre usuário e programa durante os testes.

```
adriano@adriano-Vostro-3550: java BankAccounts.TestaConta
Conta 1.0 tem saldo 0.0
Quanto quer retirar?
0
Retirada: não é possível retirar 0 ou valor negativo.
Conta 1.0 tem saldo 0.0
Quanto quer depositar?
200
Conta 1.0 tem saldo 200.0
```

Exercício 2: Herança e Testes

(a) **Importante: neste exercício a classe `Conta` não pode ser modificada de nenhuma maneira.** Usando a classe `Conta` como classe base, escreva duas classes derivadas chamadas `ContaPoupanca` e `ContaCorrente`. Um objeto

`ContaPoupanca`, além dos atributos de um objeto `Conta`, deve ter um chamado `juros` e um método que adiciona juros à conta. O atributo `juros` armazena a taxa de juros que será usada na conta. Um objeto `ContaCorrente`, além dos atributos de um objeto `Conta`, deve ter um chamado `limiteNegativo`. Este atributo garante que é possível a conta ficar negativa mas não abaixo de um certo valor. Garanta que você tenha sobrescrito os métodos necessários da classe `Conta` em ambas as classes derivadas.

(b) Escreva programas que testem todos os métodos das duas Classes.

Exercício 3: Herança e Testes

Agora crie uma classe chamada `Banco`. Um objeto desta classe contém dois `ArrayLists` de objetos. Um para `ContaCorrente` e outro para `ContaPoupanca`. Crie contas de vários tipos e as insira no `ArrayList` compatível.

Escreva um método que atualize a classe `Banco`. Este método deve iterar por todos os elementos dos `ArrayList` fazendo as atualizações da seguinte maneira:

(a) `ContaPoupanca` adicionar os juros via o método que você já construiu.

(b) `ContaCorrente` escreva uma mensagem se a conta está no negativo.

(c) Imprima o saldo de cada uma das contas.

Observações:

- A classe `Banco` requer métodos para abrir e fechar contas.
- Observe que o balanço de uma conta somente pode ser modificado através de um depósito ou retirada.
- A classe `Conta` não pode ser modificada de nenhuma maneira.
- Procure testar tudo o que fez a cada passo.

Exercício 4: Polimorfismo

A classe `Banco` do exercício anterior deve ser modificada de modo que haja somente um `ArrayList` que possa conter contas de todos os tipos (polimorfismo).

Escreva um método que atualize a classe `Banco`. Este método deve iterar por todos os elementos do array único fazendo as atualizações da maneira pedida no exercício anterior.

Neste exercício é permitido (e talvez obrigatório) modificar a classe `Conta` incluindo novos métodos. Isto irá permitir que a classe `Banco` ao percorrer a lista de contas não precise se preocupar que tipo de conta está processando.

Listagem 1: Listagem da classe Conta

```
package BankAccounts;

public class Conta {
   private double saldo;  //The current saldo
   private int numeroConta;  //The account number


   public Conta () { // Constructor
      saldo = 0.0;
   }

   public Conta (int numeroConta) {   // Constructor
      saldo=0.0;
      this.numeroConta = numeroConta;
   }

   public void setSaldo(double saldo) {
      this.saldo = saldo;
   }

   public double getSaldo() {
      return saldo;
   }

   public void setNumeroConta(int numeroConta) {
      this.numeroConta = numeroConta;
   }

   public double getNumeroConta() {
      return numeroConta;
   }

   public void deposito (double sum) {
      if (sum>0) {
         saldo +=sum;
      }
      else {
         System.err.println("Deposito: "
               +" não é possível depositar valor negativo.");
      }
   }

   public void retirada (double sum) {
      if (sum>0) {
         saldo-=sum;
      }
      else {
         System.err.println("Retirada: "
               +"não é possível retirar valor negativo.");
      }
   }

   public String toString() {
      return "Acc " + numeroConta + ": " + "saldo = " + saldo;
   }

   public final void print() {
      //Don't override this,
      //override the toString method
      System.out.println( toString() );
   }

}
```

Exercício 5: Considere o diagrama simplificado de classes mostrados na Figura 1. Execute as seguintes tarefas:

(a) Implemente as classes `Animal, Canine, Dog, Feline` e `Cat`.

(b) Crie um programa que use um `ArrayList` do tipo `Animal` e insira dois animais do tipo `Dog` e um do tipo `Cat`.

(c) Modifique o programa para que ele varra o `ArrayList` fazendo com que cada animal faça as coisas que sabe fazer.

(d) Modifique a classe `Dog` incluindo um novo método chamado `beNice()` e rode o programa novamente e veja o que acontece.

(e) Existe um teste em java que é executado da seguinte forma:

```
if (objeto instanceof  Classe) {
}
```

Modifique o seu programa para que caso o animal obtido da lista seja da classe `Dog` ele execute `beNice()`.

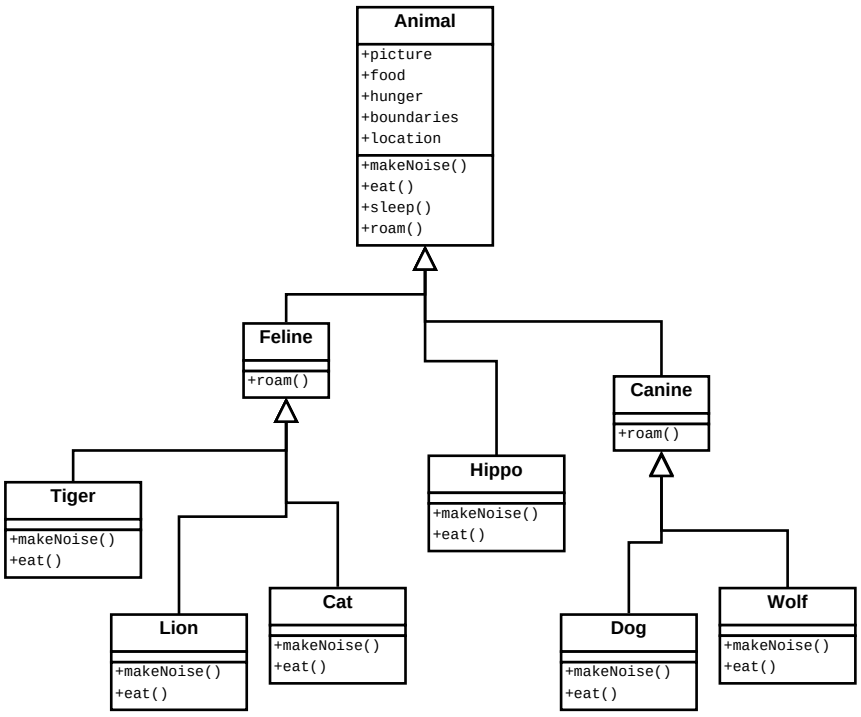

Figura 1: Diagrama de classes do reino Animal.

2 Classes úteis

Listagem 2: For melhorado.

```
int [] locationCells = {3, 4, 5};

for (int cell: locationCells) {
   if (guess == cell) {
      result = "hit";
      numberOfHits++;
      break;
   }
}
```

Listagem 3: Usando ArrayList.

```
// Make one
ArrayList<Egg> myList = new ArrayList<Egg>();
// Add on element
Egg s = new Egg();
myList.add(s);
// Add another
Egg b = new Egg();
myList.add(b);
// Find out how many this are in it.
int theSize = myList.size();
// Find out if it contains something
boolean isIn = myList.contains(s);
// Find out where something is
int idx = myList.indexOf(b);
// Find out it it is empty
boolean empty = myList.isEmpty();
// Remove
mylist.remove(s);
```

Listagem 4: Usando Scanner.

```
package TestScanner;
import java.util.Scanner;
public class TestScanner {
   public static void main(String[] args) {
      String inputLine = null;
      Scanner scan = new Scanner(System.in);
      double valor;
      System.out.print("Numero double? ");
      inputLine = scan.nextLine();
      valor = Double.parseDouble(inputLine);
      valor = valor + 1.0;
      System.out.println("Resultado " + valor);
      scan.close();
   }
}
```