Programação Orientada a Objetos

George Gomes Cabral

Cenário Estrutural

- Foco principal nas funções, depois nos dados, informações desencontradas.
- Idéias e necessidades dos usuários normalmente não ficam claras.
- Grandes arquivos de códigos fontes
  - Código difícil de manter
- Baixa modularização

Cenário OO

- Foco principal nos objetos do mundo real, com suas funções e dados agrupados.
- Deve-se fazer o levantamento de requisitos do sistema já pensando nos objetos do mundo real.
- Diminuição do tempo e custo do sistema
  - Reutilização de Código
  - Modularização
  - Facilidade na Manutenção

Objeto

- Um objeto denota uma entidade, seja ela de natureza física ou conceitual ou de software.
  - Entidades físicas: um carro, uma pessoa, uma casa, etc.
  - Entidade conceitual: um organograma de uma empresa.
  - Entidade de software: um botão em uma interface gráfica.

Objeto

- É uma entidade capaz de reter um estado e que oferece uma série de operações.
- É um conceito, uma abstração, algo com limites e significados nítidos em relação ao domínio de uma aplicação.
- Facilitam a compreensão do mundo real.
- Incorporam: Estado, Comportamento e Identidade.

Estado de um objeto

- O estado de um objeto representa uma das possíveis condições em que um objeto pode se encontrar.
- O estado é representado pelos valores das propriedades de um objeto em um determinado instante.
- Esse estado, geralmente, muda ao longo do tempo.
  - Nome: Maurício
  - Matrícula: 105018
  - Semestre de Ingresso: 2000A

Comportamento de um objeto

- O comportamento determina como o objeto pode responder a interações mediante à ativação de operações decorrentes de mensagens recebidas de outros objetos.
- Objeto: Aluno
- Comportamento: solicitaMatrícula()
  solicitaDispensa ()
  checaHistorico()
- Cada objeto tem um único identificador, mesmo que seu estado seja idêntico ao de outro objeto.

Classe

- Uma classe é a descrição de um grupo de objetos com propriedades semelhantes (atributos), mesmo comportamento (operações), mesmos relacionamentos com outros objetos e mesma semântica.
- Um objeto é uma instância de uma classe.
- Pode ser vista como ma abstração que:
  - Enfatiza características relevantes
  - Abstrai outras características

Classe

- Nomeando classes
  - Uma classe deve ter como nome um substantivo singular que melhor caracteriza a abstração.
  - Dificuldades na nomeação das classes podem indicar abstrações mal definidas.
  - Nomes devem surgir diretamente do domínio do problema.
  - Nomes devem começar com letra maiúscula.
  - Em caso de mais de uma palavra no nome da classe, cada palavra deve ter a primeira letra maiúscula. Ex: ControleAcademico.java.

Atributo

- O estado de um objeto é dado por valores de atributos e por ligações que tem com outros objetos
- Atributos são definidos a nível de classe, enquanto que valores de atributos são definidos a nível de instância.
- Exemplos:
  - Uma pessoa (classe) tem os atributos nome, data de nascimento e peso.
  - João é uma pessoa (instância da classe Pessoa) com nome "João da Silva", data de nascimento "18/03/1973" e peso "70kg".

Visibilidade de Atributos e Métodos

- Atributos e Métodos Públicos
  - São atributos e métodos dos objetos que podem ser visíveis externamente, ou seja, outros objetos poderão acessar os atributos e métodos desses objetos sem restrição.
- Atributos e Métodos privados (**private**)
  - Atributos e Métodos que só podem ser acessados por operações internas ao próprio objeto são ditos privados.
- Os métodos públicos de uma classe definem a interface da classe. Os métodos privados não fazem parte da interface pois não podem ser acessados externamente.

Herança e Hierarquia

- Hierarquia
  - Quando vamos trabalhar com um conjunto grande de classes de objetos, é necessário organizar estas classes de maneira ordenada de modo que tenhamos uma hierarquia.
  - Em uma hierarquia de classes teremos as classes mais genéricas no topo e as mais específicas na base.

Hierarquia

- Automóveis
  - Automóveis Utilitários
    - Utilitários urbanos
    - Utilitários off-road
  - Automóveis Passeio
    - Passeio família
    - Passeio esportivo
  - Automóveis de Carga
    - Carga inflamável
    - Carga com frigorífico

Herança

- Em uma hierarquia de classes semelhantes, podemos dizer que as classes mais específicas herdam as características das mais genéricas, ou seja, todo automóvel de passeio família é um automóvel de passeio.
- A classe de nível superior na associação de herança é chamada de super-classe e a inferior de sub-classe
- Automóvel de passeio família é uma sub-classe de automóvel de passeio.

Agregação x Composição

- Agregação
  - Um objeto contém uma lista de outros objetos
  - Os objetos contidos podem existir sem serem parte do objeto que os contém.
  - Ex.: Aluno → Disciplinas. Um aluno pode ter várias disciplinas, porém, uma disciplina pode existir sem um aluno.
- Composição
  - Um objeto contém uma lista de outros objetos
  - Os objetos contidos não fazem sentido fora do contexto do objeto que os contém.
  - Ex.:Pedido → Itens. Se você destruir o pedido, os itens são destruídos junto, eles não tem sentido fora do pedido.

Encapsulamento

- É o processo de ocultação de características internas do objeto.
  - Cuida para que certas características não possam ser vistas ou modificadas externamente.
- Ex.:
  - Podemos dizer que o motor do automóvel está encapsulado, pois normalmente não podemos ver ou alterar características internas do motor.

Encapsulamento

- No contexto de OO, o encapsulamento protege os dados que estão dentro dos objetos, evitando assim que os mesmos sejam alterados erroneamente.
- Os dados só poderão ser alterados pelas funções dos próprios objetos.
- Existe também o encapsulamento a nível de pacotes.

Polimorfismo

- Qualquer objeto Java que passar por mais de um teste *IS-A* pode ser considerado polimórfico.
  - Exceto *Object* qualquer objeto Java pode ser considerado polimórfico.
- Uma classe não pode herdar de mais de uma classe.

```
class PlayerPiece extends GameShape, Animatable{  // ERRADO !!!!
    //codigo
}
```

POLIMORFISMO

```
public interface Animatable {
    public void animate();
}

class PlayerPiece extends GameShape implements Animatable{
    public void movePiece(){
            System.out.println("moving game piece");
    }
    public void animate() {
            System.out.println("animating...");
    }
    ///mais codigo
}
```

POLIMORFISMO

- Isso significa que PlayerPiece pode ser tratado como um dos quatro tipos a qualquer momento:
  - Um Objeto (qualquer objeto herda de object)
  - Um GameShape (PlayerPiece herda de GameShape)
  - Um PlayerPiece (é o que PlayerPiece realmente é)
  - Um Animatable (PlayerPiece implementa um Animatable)
- Possíveis declarações:
  - PlayerPiece player = new PlayerPiece();
  - Object o = player;
  - GameShape shape = player;
  - Animatable mover = player;

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

- Sempre que se tem uma classe que herda de uma superclasse, existe a oportunidade de sobrescrever o método (exceto quando o método é marcado como *final* ou *static*).

```
public class Animal {
    public void eat() {
            System.out.println("animal generico comendo genericamente");
    }
}

class Horse extends Animal {
    public void eat() {
            System.out.println("Cavalo comendo coisas de cavalo como um cavalo")
    }
    public void buck();
}
```

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

- Quando marcado como *abstract* na superclasse, é obrigatória a sobrescrita do método na subclasse.
- Um exemplo não abstrato de polimorfismo.

```
public class TestAnimals{
    public static void main (String[] args){
            Animal a = new Animal();
            Animal b = new Horse();
            a.eat();  //roda a versao de eat() de animal
            b.eat();  //roda a versao de eat() de cavalo
    }
}
```

- O compilador olha somente para o tipo da referencia, não para o da instância.

```
Animal c = new Horse();
c.buck();    //buck() não pode ser invocado
             // a classe animal não tem o método
```

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

- O método sobrescrito da subclasse não pode ter um modificador de acesso mais restritivo que o da superclasse.

```
public class Animal {
    public void eat() {
            System.out.println("animal generico comendo genericamente");
    }
}

class Horse extends Animal {
    private void eat() { // errado !!
            System.out.println("Cavalo comendo coisas de cavalo como um cavalo")
    }
    public void buck();
}
```

- Invocando a versão do método escrita na superclasse
  - super.eat();

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

Illegal Override Code	Problem with the Code
`private void eat() { }`	Access modifier is more restrictive
`public void eat() throws IOException { }`	Declares a checked exception not defined by superclass version
`public void eat(String food) { }`	A legal overload, not an override, because the argument list changed
`public String eat() { }`	Not an override because of the return type, not an overload either because there's no change in the argument list

- Overload Methods (sobrecarga)
  - Métodos sobrecarregados devem mudar a lista de argumentos
  - Métodos sobrecarregados podem mudar o tipo retornado
  - Métodos sobrecarregados podem mudar o modificador de acesso
  - Métodos sobrecarregados podem declarar novas exceções

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

- Exemplos de Sobrecargas legais
  - public void changeSize(int size, String name, float pattern){ }
  - public void changeSize(int size, String name){ }
  - public void changeSize(int size, float pattern){ }
  - public void changeSize(String name, float pattern) throws IOException { }
- A decisão de qual método invocar é baseada nos argumentos

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

```
class Adder {
    public int addThem(int x, int y) {
            return x + y;
    }
    //sobrecarrega o método para adicionar doubles ao invés de ints
    public double addThem(double x, double y) {
            return x + y;
    }
}
//a partir de outra classe
public class TestAdder {
    public satic void main (String[] args) {
            Adder a = new Adder();
            int b = 27;
            int c = 3;
            int result = a.addThem(b, c); //qual método será invocado ?
    }
}
```

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

```
class Animal { }
class Horse extends Animal { }
class UseAnimals {
    public void doStuff(Animal a) {
            System.out.println("versao animal");
    }
    public void doStuff(Horse h) {
            System.out.println("versao cavalo");
    }
    public static void main(String[] args){
            UseAnimals ua = new UseAnimals();
            Animal a = new Animal();
            Horse h = new Horse();
            ua.doStuff(a);
            ua.doStuff(h);
    }
}

Animal animalRefToHorse = new Horse();
ua.doStuff(animalRefToHorse); //qual a saida ?
```

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

Method Invocation Code	Result
`Animal a = new Animal(); a.eat();`	`Generic Animal Eating Generically`
`Horse h = new Horse(); h.eat();`	`Horse eating hay`
`Animal ah = new Horse(); ah.eat();`	`Horse eating hay` Polymorphism works—the actual object type (Horse), not the reference type (Animal), is used to determine which eat() is called.
`Horse he = new Horse(); he.eat("Apples");`	`Horse eating Apples` The overloaded eat(String s) method is invoked.
`Animal a2 = new Animal(); a2.eat("treats");`	Compiler error! Compiler sees that Animal class doesn't have an eat() method that takes a String.
`Animal ah2 = new Horse(); ah2.eat("Carrots");`	Compiler error! Compiler *still* looks only at the reference, and sees that Animal doesn't have an eat() method that takes a String. Compiler doesn't care that the actual object might be a Horse at runtime.

SOBRESCRITA (OVERRIDE) E SOBRECARGA (OVERLOAD)

Diferenças entre sobrecarga e sobrescrita

	Overloaded Method	Overridden Method
Argument(s)	Must change.	Must not change.
Return type	Can change.	Can't change except for covariant returns.
Exceptions	Can change.	Can reduce or eliminate. Must not throw new or broader checked exceptions.
Access	Can change.	Must not make more restrictive (can be less restrictive).
Invocation	*Reference* type determines which overloaded version (based on declared argument types) is selected. Happens at *compile* time. The actual *method* that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the *signature* of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the *class* in which the method lives.	Object type (in other words, *the type of the actual instance on the heap*) determines which method is selected. Happens at *runtime*.

CASTING

- Força uma variável a ser de um determinado tipo

```
public class CastTest2 {
    public static void main(String args[]){
            Animal[] a = {new Animal(), new Dog(), new Animal()};
            for(Animal animal: a){
                    animal.makeNoise();
                    if (animal instanceof Dog) {
                            animal.playDead(); //tenta se comportar como um cachorro
                    }
            }
    }
}

class Animal{
    void makeNoise(){System.out.println("generic noise!");}
}
class Dog extends Animal{
    void makeNoise(){System.out.println("latido");}
    void playDead(){System.out.println("embola");}
}
```

CASTING

```
public class CastTest2 {
        public static void main(String args[]){
                Animal[] a = {new Animal(), new Dog(), new Animal()};
                for(Animal animal: a){
                        animal.makeNoise();
                        if (animal instanceof Dog) {
                                Dog d = (Dog) animal;
                                d.playDead(); //tenta se comportar como um
                                                //cachorro
                        }
                }
        }
}

class Animal{
    void makeNoise(){System.out.println("generic noise!");}
}
class Dog extends Animal{
    void makeNoise(){System.out.println("latido");}
    void playDead(){System.out.println("embola");}
}
```

- Downcast - cast para uma classe mais específica.

CASTING

```
class Animal{}
class Dog extends Animal{}
class DogTest{
    public void static main(String args[]){
            Animal animal = new Animal();
            Dog d = (Dog) animal; //compila mais falha em tempo de execução
            String s = (String) animal; //não compila !!!
    }
}
```

Java.lang.ClassCastException

```
class DogTest{
    public void static main(String args[]){
            Dog d = new Dog();
            Animal a1 = d;   //upcast sem cast explicito
            Animal a2 = (Animal) d; //upcast com cast explicito
    }
}
```

VARIÁVEIS E MÉTODOS ESTÁTICOS

- Imagine uma classe com um método que sempre roda da mesma maneira; sua única função é retornar, por exemplo, um número aleatório. Não interessa qual a instância da classe executou o método.
- Suponha que lhe interessa saber quantos objetos de uma determinada classe foram instanciados. Não funcionaria armazenar esse valor em uma variável de instância da classe pois a cada construção de uma nova instância essa variável seria reiniciada ao seu valor inicial.

VARIÁVEIS E MÉTODOS ESTÁTICOS

```
public class Frog {

    static int frogCount = 0;

    public Frog(){
            frogCount += 1;
    }
    public static void main(String[] args){
            new Frog();
            new Frog();
            new Frog();
            System.out.println("Frog count is now "+frogCount);
    }
}
```

VARIÁVEIS E MÉTODOS ESTÁTICOS

```
public class Frog {

    int frogCount = 0;

    public Frog(){
            frogCount += 1;
    }
    public static void main(String[] args){
            new Frog();
            new Frog();
            new Frog();
            System.out.println("Frog count is now "+frogCount);
    }
}
```

ERRO !!!!!

```
Frog.java:11: non-static variable frogCount cannot be referenced from a static context
    system.out.println("Frog count is now "+ frogCount);
    1 error                                      ^
```

ACESSANDO MÉTODOS E CARIÁVEIS ESTÁTICAS

```
public class Frog {

    static int frogCount = 0;

    public Frog(){
            frogCount += 1;
    }
}
class TestFrog{
    public static void main(String[] args){
            new Frog();
            new Frog();
            Frog f = new Frog();
            System.out.println("Frog count is now "+Frog.frogCount);
            System.out.println("Frog count is now "+f.frogCount); //também funciona

    }
}
```

Acoplamento e Coesão

- Relacionamento direto com a qualidade do design Orientado a Objetos.
- Bom design Orientado a objetos:
  - Baixo nível de acoplamento
  - Alto nível de Coesão
- Metas na criação de aplicações:
  - Facilidade na criação
  - Facilidade na manutenção
  - Facilidade em melhorias

Acoplamento

- Grau de conhecimento de uma classe sobre o desenho de outra classe.
- Se o único conhecimento que uma classe A tem sobre uma classe B é o que a classe B expôs através de sua interface, então as classes A e B têm um baixo nível de acoplamento.
- Se a classe A sabe sobre partes da classe B que não são interfaces de B, então as classes A e B têm um alto nível de acoplamento.
  - O que aconteceria com a classe A em um caso de melhoria na classe B ?

Acoplamento

- Aplicações não triviais utilizando OO são uma mistura de muitas classes e interfaces trabalhando juntas. Todas as interações entre objetos devem ser baseadas em contratos.

```
public class DoTaxes {
    float rate;
    float doColorado(){
            SalesTaxRates str = new SalesTaxRates();
            rate =  str.salesRate;
            //faz alguma coisa com rate
            return rate;
    }
}
class SalesTaxRates{
    public float salesRate;
    public float adjustedSalesRate;

    public float getSalesRates(String region){
            salesRate = new DoTaxes().doColorado();
            //faz calculos baseados na regiao
            return adjustedSalesRate;    } }
```

Coesão

- Tem a ver com o design de uma única classe.
- Termo utilizado para indicar o grau em que uma classe tem um único e bem definido propósito.
- Benefício de alta coesão: Classes fáceis de manter (e menos freqüentemente alteradas) e tendência a uma maior reusabilidade que classes com um menor nível de coesão.

Coesão

- Baixo Grau de Coesão !!
- Uma classe tem vários métodos com tarefas bem distintas.

```
class BudgetReport{
    void connectToRDBMS(){}
    void generateBudgetReport(){}
    void saveToFile(){}
    void print(){}
}
```

Coesão

- Alto grau de Coesão !!
- Cada classe tem seu papel bem definido.

```
class BudgetReport{
    Options getReportOptions(){}
    void generateBudgetReport(Options o){}
}

class ConnectToRDBMS{
    DBConnection getRDBMS(){}
}

class printStuff{
    PrintOptions getPrintOptions(){}
}

class FileSaver{
    SaveOptions getFileSaveOptions(){}
}
```

Exercícios

1. Qual afirmação é verdadeira ?
   1. Relacionamento *tem um* (HAS-A) sempre se associa a herança.
   2. Relacionamento *tem um* (HAS-A) sempre se associa a variáveis de instância.
   3. Relacionamento *tem um* (HAS-A) sempre requerem pelo menos dois tipos de classes.
   4. Relacionamento *tem um* (HAS-A) sempre se associa a polimorfismo.
   5. Relacionamentos *tem um* (HAS-A) sempre indicam forte acoplamento
2. Dado:

```
Class Clidders {
        public final void flipper(){print("clider")}
}
Public class Clidlets extends Clidders{
        public void flipper(){
          print("flip a clidlet");
          super.flipper();
        }
        public static void main(String[] args){
          new Clidlets().flipper();
        }
}
```

Exercícios

Qual o resultado ?

1. Flip a clidlet
2. Flip a clidder
3. Flip a clidder
   Flip aClidlet
4. Flip a clidlet
   Flip aClidder
5. Falha na compilação

3. Selecione a afirmação que melhor indicam baixo nível de acoplamento.
   1. Os atributos de uma classe são todos privados.
   2. A classe se refere a um pequeno número de outros objetos.
   3. Os objetos contêm apenas um pequeno número de variáveis.
   4. É improvável que mudanças realizadas em uma classe requeiram mudanças na outra classe.

Exercícios

4. Dado:

```
Class Clidders {
        private final void flipper(){print("clidder")}
}
Public class Clidlets extends Clidders{
        public final void flipper(){
          print("clidlet");
        }
        public static void main(String[] args){
          new Clidlets().flipper();
        }
}
```

Qual o resultado ?

1. clidlet
2. clidder
3. clidder
   clidlet
4. clidlet
   clidder
5. Falha na compilação

Exercícios

5. Dado:

```
class Zing{
        protected Hmpf h;
}
class Woop extends Zing{}
class Hmpf {}
```

Quais afirmativas são verdadeiras ?

1. Woop "é um" Hmpf e "tem um" Zing.
2. Zing "é um" Woop e "tem um" Hmpf.
3. Hmpf "tem um" Woop e Woop "é um" Zing
4. Woop "tem um" Hmpf e Woop "é um" Zing
5. Zing "tem um" Hmpf e Zing "é um" Woop.

Respostas

1. 2
2. 5 (um método final não pode ser sobrescrito)
3. 4
4. 1 (um metodo final não pode ser sobrescrito, poré, nesse caso ele é privado, não é visível na subclasse)
5. 4