

Disciplina: LPOO

Prof. Anderson V. de Araujo

Aula 07: Conceitos Avançados de Programação Orientada a Objetos

andvicoso@facom.ufms.br

<http://prof.facom.ufms.br/~andvicoso/>

4 *Pilares da POO* (Conceitos Fundamentais)

- Herança
- Abstração
- Polimorfismo
- Encapsulamento

Precisamos lembrar do funcionamento dos **modificadores de acesso!**
(`private`, `protected`, `package` e `public`)

Herança

Herança

- A habilidade de criar uma **nova classe a partir de uma classe existente**
- É uma forma de **reutilização de código** na qual uma nova classe é criada, absorvendo membros de uma classe existente
 - Podendo ser aprimorada com novos membros ou modificar os existentes
- **Economiza tempo** no desenvolvimento
- **Aumenta a qualidade**

Herança (2)

- Ao declarar uma classe, é possível definir **UMA** classe para herdar (ou estender) seus membros internos
 - Os membros que serão visíveis nos filhos dependem do **modificador de acesso** associado
- A classe existente é chamada de **superclasse** (ou classe pai)
 - Superclasse direta: Primeira na hierarquia
 - Superclasse indireta: Qualquer outra classe na hierarquia (classe pai da classe pai, ...)
- A nova classe sendo criada é chamada de **subclasse** (ou classe filha)
 - A subclasse é mais específica que a superclasse e representa um grupo mais especializado de objetos

Herança - Sintaxe

- Superclasse

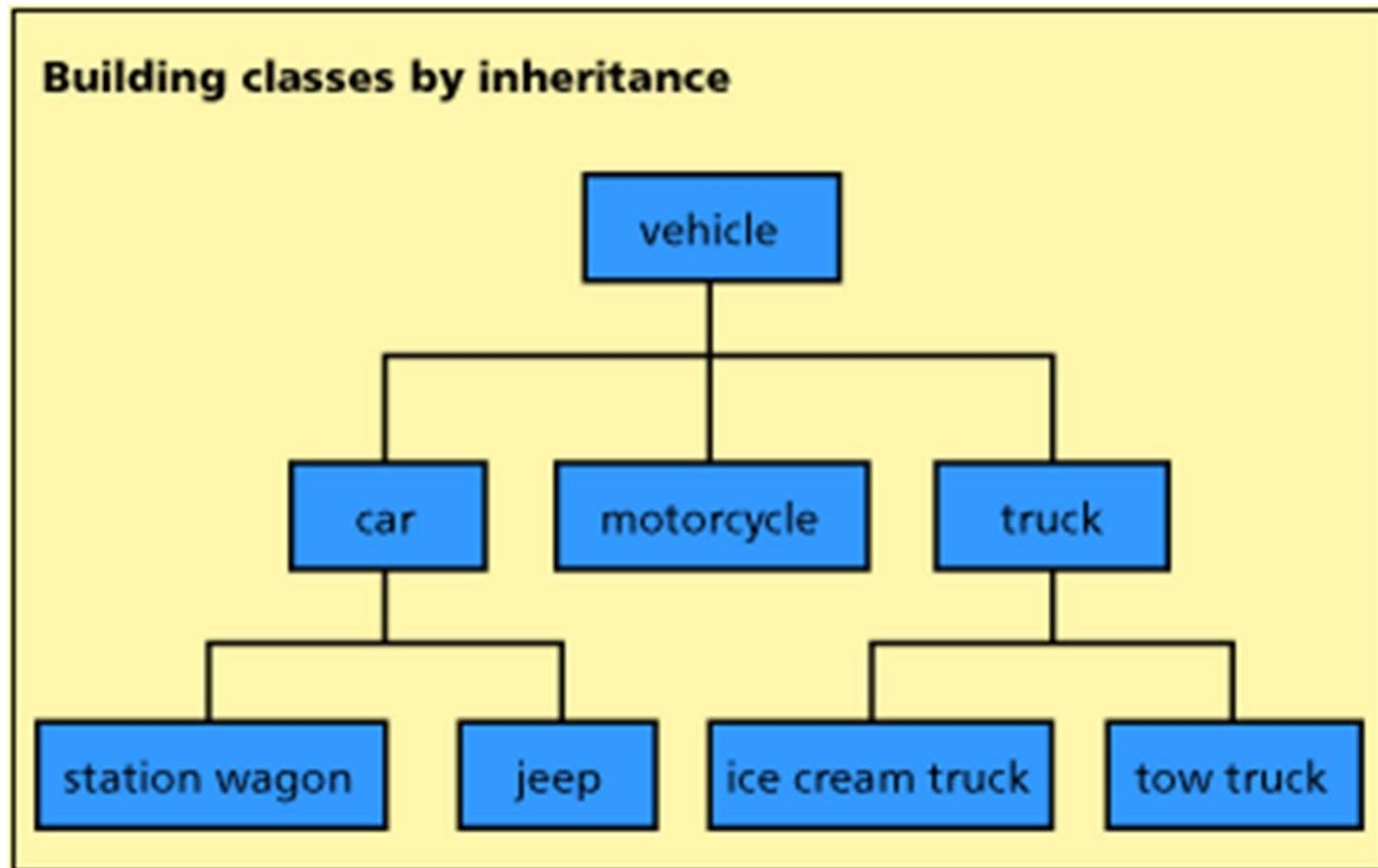
```
class Esporte{  
    String nome;  
}
```

- Subclasse

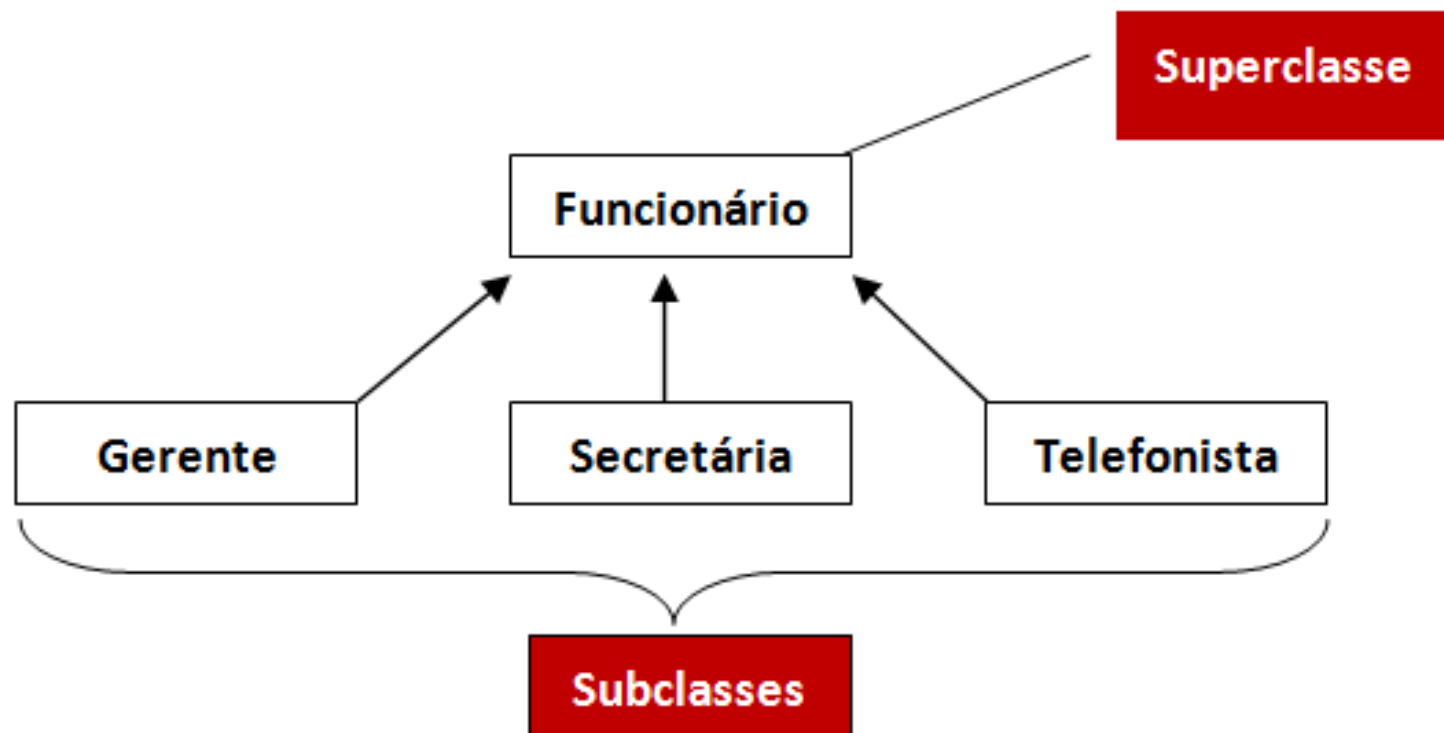
- A cláusula opcional **extends** indica a superclasse de onde os membros vão ser herdados

```
class EsporteAquatico extends Esporte {}
```

Herança - Exemplo



Herança – Exemplo (2)



Herança (3)

- A primeira vista a herança ajuda a solucionar vários problemas
 - Mas não devemos usar herança pra tudo...
- Preferir usar
 - Composição ou Agregação
- Java **não possui herança múltipla para classes**
 - Ou seja, uma classe não pode herdar características de diferentes superclasses
 - `class A extends B, C, D{`



Errado!

A Palavra Reservada *super*

- A palavra reservada ***super*** indica que é possível acessar um membro (atributo ou método) de uma classe pai a partir de uma classe filha
 - Claro que isso depende do **modificador de acesso** do membro
 - Pode chamar o construtor da classe pai também
 - Ex: `super(nome, idade);`
- Similar ao ***this*** só que referencia os métodos da classe pai
- Fica mais claro quando usado com **Polimorfismo**

Exemplo - Super

```
public class CellPhone {  
    public void print() {  
        System.out.println("I'm a cellphone");  
    }  
}  
  
public class TouchPhone extends CellPhone {  
    public void printTouch() {  
        super.print();  
        System.out.println("I'm a touch screen cellphone");  
    }  
  
    public static void main (String[] args) {  
        TouchPhone p = new TouchPhone();  
        p.printTouch();  
    }  
}
```

Exemplo – Super (2) - Atributo

```
class TestAttrib {  
    public static void main(String args[]) {  
        Subclass obj = new Subclass();  
        obj.printNumber();  
    }  
}  
  
class Parentclass {  
    int num = 100;  
}  
  
class Subclass extends Parentclass {  
    int num = 110;  
  
    void printNumber() {  
        System.out.println(num);  
        System.out.println(super.num);  
    }  
}
```

A Classe *Object*

- A classe **Object** é a classe pai de todas as classes definidas
 - Mesmo que você **não** tenha definido isso (através da palavra reservada **extends**)
- Ou seja, todas as classes que criar serão filhas de **Object**
 - Com isso, a sua classe tem acesso aos membros internos visíveis da classe Object
 - Inclusive vetores
- A classe Object possui os métodos:
 - **boolean** equals(Object obj)
 - **void** finalize() **throws** Throwable
 - **final** Class getClass()
 - **int** hashCode()
 - String toString()
 - Etc...

Abstração

Abstração

- Habilidade de expor características essenciais de uma classe enquanto se esconde outros detalhes irrelevantes
- Por que usar abstração?
 - **Reduz a complexidade do código**
 - Organiza o projeto/código
- Tornar uma classe **abstrata** (ou seja, uma classe que não pode ser instanciada) **ou** implementar uma **interface**
- -- *Abstraction is more about '**What**' a class can do.*
[Idea]

Abstração (2)

- A definição da abstração **ocorre durante a fase de planejamento/definição das classes**
- Se uma classe é abstrata e não pode ser instanciada, a classe não tem muita utilidade, a não ser que seja estendida (herança)
- A classe pai contém a funcionalidade comum de uma coleção de classes filhas (como na herança simples) mas não tem função sozinha
- Na classe abstrata é possível ter métodos normais e **ter métodos abstratos** também

Abstração (3)

- Modificadores de **classe**:
 - ***final***: nenhuma classe pode estender (herdar) da classe final
 - ***abstract***: a classe não pode ser instanciada
- Modificador de **método**:
 - ***abstract***: o método não vai ser instanciado na classe abstrata, mas deve ser obrigatoriamente implementado em suas subclasses concretas

Exemplo - Abstração

```
abstract class Instrumento {  
    protected double peso;  
    public abstract void tocar();  
}
```

```
abstract class InstrumentoDeCordas extends Instrumento {  
    protected int numeroDeCordas;  
}
```

```
public final class Guitarra extends InstrumentoDeCordas {  
    public Guitarra () {  
        peso = 3.5;  
        numeroDeCordas = 6;  
    }  
    public void tocar(){  
        //...  
    }  
}
```

Polimorfismo

Polimorfismo

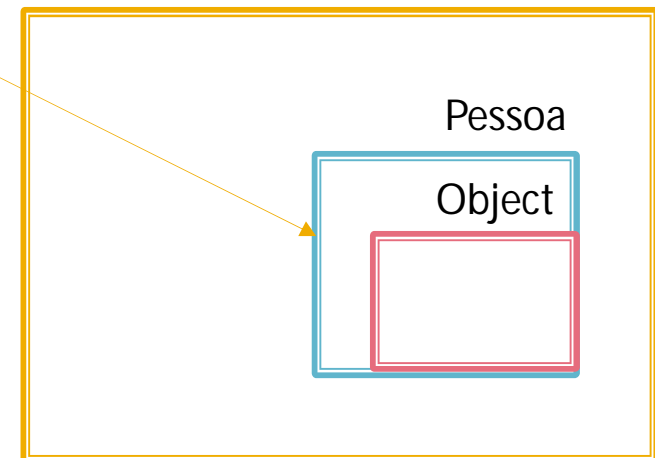
- Poli=Muitas; Morfo = Formas
- É a capacidade de um objeto poder ter muitas formas
- Está fortemente associada à **herança**
- A subclasse pode definir seu próprio comportamento único e ainda compartilhar as mesmas funcionalidades de seu pai
 - Mas uma **classe pai não pode** ter o comportamento **da** sua **subclasse**

Polimorfismo – *Object* - Exemplo

- `Object o1 = new Object();`
 - `//o1` pode referenciar qualquer subtipo
- `Object o2 = new String();`
- `Object o3 = new Pessoa();`

Neste exemplo a variável de referência `o3` só irá conseguir *enxergar* a parte referente a classe *Object* da instância criada

Memória



Polimorfismo - Tipos

- Funciona somente sobre **métodos**
 - Não serve para atributos
- Tipos:
 - Sobrecarga
 - Sobrescrita

Polimorfismo - Sobrecarga

- Ter vários métodos dentro de uma classe com o mesmo nome, mas com **assinaturas diferentes**
- Também chamado de *compile time polymorphism* ou *static binding*
- É possível:
 - Ter um número diferente de parâmetros
 - Ter tipos diferentes para os parâmetros
- Não é possível:
 - Ter somente o tipo de retorno diferente*
 - Claro, o tipo de **retorno não faz parte da assinatura** do método
 - Mudar apenas os nomes dos parâmetros
 - Se for de pai pra filho, vira **sobrescrita**
- **Sobrescrita de Métodos**
 - Prover uma **nova implementação para um método existente e visível da classe pai**
 - Também chamado de *Runtime polymorphism* ou *dynamic binding*

Polimorfismo – Sobrecarga - Exemplos

```
//compiler error - can't overload based on the type returned
//(one method returns int, the other returns a float)
int changeDate(int year) {...};
float changeDate(int year) {...};

//compiler error - can't overload by changing just
//the name of the parameter (from Year to Month)
int changeDate(int year) {...};
int changeDate(int month) {...};

//valid case of overloading, since the methods
//have different number of parameters
int changeDate(int year, int month) {...};
int changeDate(int year) {...};

//also a valid case of overloading, since the
//parameters are of different types
int changeDate(float year) {...};
int changeDate(int year) {...};

//also a valid case of overloading, since the
//parameters AND return type are different
int changeDate(float year) {...};
double changeDate(double var) {...};
```


Polimorfismo – Sobrecarga – Exemplo (2)

```
class Sobrecarga{  
    int method(float x) {  
        return 4;  
    }  
  
    int method(double x) {  
        return 2;  
    }  
  
    public static void main(String[] args) {  
        Sobrecarga d = new Sobrecarga();  
        System.out.println(d.method(4));  
    }  
}
```

Ampliação

Polimorfismo - Sobrescrita

- Provê uma **nova implementação** para um método existente e **visível** da classe pai
- Depende da **Herança**
- Também chamado de *Runtime polymorphism* ou *dynamic binding*

Polimorfismo – Exemplo Sobrescrita

```
class Parent {  
    public int someMethod() {  
        return 3;  
    }  
}  
  
class Child extends Parent {  
    public String name;  
  
    public int someMethod() {  
        return 4;  
    }  
}
```

```
Parent p = new Parent();  
p.someMethod(); //???
```

```
Parent p = new Child();  
p.someMethod(); //???
```

```
Child c = new Parent();  
c.someMethod(); //???
```



Errado!

Polimorfismo – Exemplo (2)

Sobrescrita

```
class Top {  
    static void myTop() {  
        System.out.println("myTop method in Top class");  
    }  
}  
  
public class Down extends Top {  
    void myTop() {  
        System.out.println("myTop method in Down class");  
    }  
    public static void main(String[] args) {  
        Top t = new Down();  
        t.myTop();  
    }  
}
```

1. Compila?
2. Se tirar o static?
3. Se colocar o static no myTop de Down?

Polimorfismo – Exemplo (3)

Sobrescrita

```
class A {
    int i = 10;

    public void demo(){
        System.out.println("Inside class A");
    }
}

class B extends A{
    int i = 20;
    public void demo(){
        System.out.println("Inside class B");
    }
}

public class Test{

    public static void main(String[] args) {
        A a = new B();
        System.out.println(a.i);
        a.demo();

        B b = (B)a;
        System.out.println(b.i);
    }
}
```

1. Compila?
2. O que sai impresso no console?
3. Se adicionar no método demo de B:
 - `sysout(super.i);`

Testar no Eclipse

Polimorfismo – Exemplo (4)

Sobrescrita e Super

```
public class CellPhone {  
    public void print() {  
        System.out.println("I'm a cellphone");  
    }  
}  
  
public class TouchPhone extends CellPhone {  
    public void print() {  
        super.print();  
        System.out.println("I'm a touch screen cellphone");  
    }  
  
    public static void main (String[] args) {  
        TouchPhone p = new TouchPhone();  
        p.print();  
    }  
}
```

Polimorfismo – Exemplo (5)

Sobrescrita

```
class Animal {  
    public void makeNoise(){  
        System.out.println("Some sound");  
    }  
}  
  
class Dog extends Animal{  
    public void makeNoise(){  
        System.out.println("Bark");  
    }  
}  
  
class Cat extends Animal{  
    public void makeNoise(){  
        System.out.println("Meawoo");  
    }  
}
```

Polimorfismo – Exemplo (5) (2)

Sobrescrita

```
abstract class Animal {  
    public abstract void makeNoise();  
}  
  
class Dog extends Animal{  
    public void makeNoise(){  
        System.out.println("Bark");  
    }  
}  
  
class Cat extends Animal{  
    public void makeNoise(){  
        System.out.println("Meawoo");  
    }  
}
```


Polimorfismo – Exemplo (5) (3)

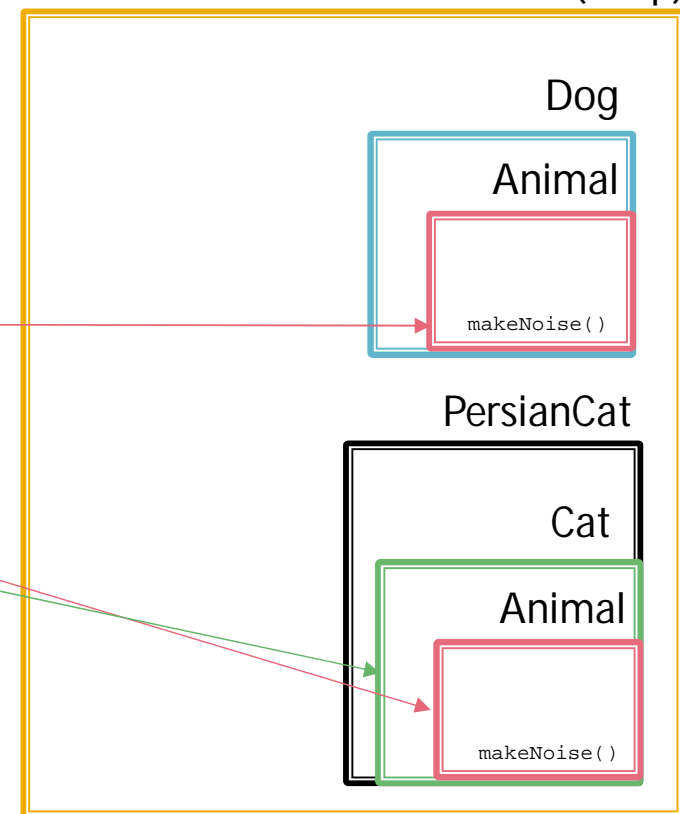
Sobrescrita

```
class PersianCat extends Cat{
    public void makeNoise(){
        System.out.println("Meawoooooooo");
    }
}

class Test{
    public static void main(String[] args){
        Animal an1 = new Dog();
        Animal an2 = new PersianCat();
        Cat cat = (Cat)an2;

        an1.makeNoise();
        cat.makeNoise();
    }
}
```

Memória (heap)



- As variáveis de referência **an1** e **an2** apontam para os campos **visíveis** da classe **Animal** para a instância criada (Dog)
- A variável de referência **cat** aponta para os campos **visíveis** da classe **Cat** para a instância (PersianCat) criada

Herança + Abstração + Polimorfismo

Exemplo

```
abstract class Instrumento {  
    protected double peso;  
    public abstract void tocar();  
}
```

```
abstract class InstrumentoDeCordas extends  
Instrumento {  
    protected int numeroDeCordas;  
}
```

```
final class Violao extends InstrumentoDeCordas{  
    public Violao () {  
        peso = 2.1;  
        numeroDeCordas = 6;  
    }  
    public void tocar(){  
        //bater nas cordas...  
    }  
}
```

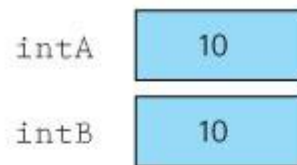
```
final class Piano extends Instrumento {  
    public Piano () {  
        peso = 200.0;  
    }  
    public void tocar(){  
        //apertar as teclas...  
    }  
}
```

```
final class Guitarra extends  
InstrumentoDeCordas {  
    public Guitarra () {  
        peso = 3.5;  
        numeroDeCordas = 6;  
    }  
}
```

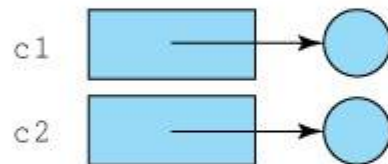
```
class Test{  
    public static void main(String[] args){  
        Instrumento g1 = new Piano();  
        InstrumentoDeCordas g2 = new Guitarra();  
        Guitarra g3 = new Guitarra();  
  
        g3.tocar();//?  
        g1.tocar();//?  
    }  
}
```

Relembrando...

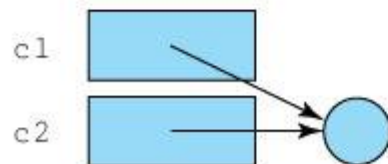
Como Comparar Elementos?



`"intA == intB"` evaluates to true



`"c1 == c2"` evaluates to false



`"c1 == c2"` evaluates to true

Comparando

OPERADOR ==

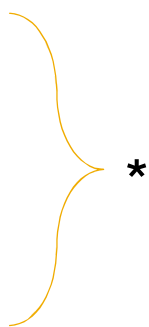
- Compara os valores em variáveis de tipos primitivos
- Em objetos, serve para comparar referências, inclusive com *null*
 - Retorna *true* se as variáveis de referências apontam para o mesmo objeto

MÉTODO `equals`

- É um método da superclasse *Object*
- Deve-se sobrescrever o método *equals* para proceder de acordo com a regra de igualdade
 - Por exemplo, na classe *String*, o *equals* é sobrescrito para comparar caractere por caractere, verificando se ambas as strings têm todos caracteres iguais
- Se não for definido pela classe, funciona exatamente igual o operador ==

Exemplo equals ()

```
class Pessoa {  
    public boolean equals(Object obj) {  
        // trata regra de igualdade  
        Pessoa pes = (Pessoa) obj;  
        if (pes.cpf.equals(this.cpf))  
            return true;  
        else  
            return false;  
    }  
}
```



*Ou somente: `return pes.cpf.equals(this.cpf);`

hashCode

- Implementa a função de hash para a classe
- É um método presente na superclasse *Object* que deve ser *sobrescrito*
- Ao implementar *equals*, deve-se implementar *hashCode* também
 - Quando *a.equals(b)*, então *a.hashCode()* deve ser o mesmo de *b.hashCode()*
 - As variáveis usadas para fazer a comparação em *equals* devem ser usadas para gerar o *hashCode*
- É usado para dar suporte as coleções que usam *hash*, tais como:
 - Hashtable, HashMap, HashSet, etc

hashCode - Exemplo

- Geralmente é gerado usando números primos grandes (13, 17, 31, ...) multiplicados com os campos* de tipo primitivo ou hashCode dos objetos do objeto sendo comparados

```
public class Employee{
    int        employeeId;
    String      name;
    Department  dept;

    public int hashCode() {
        int hash = 1;
        hash = hash * 17 + employeeId;
        hash = hash * 31 + name.hashCode();
        hash = hash * 13 + (dept == null ? 0 : dept.hashCode());
        return hash;
    }
}
```

equals e hashCode - Exercícios Práticos

- Criar uma classe livro, com nome, autor e ISBN
- Criar método *equals* e *hashCode* para o livro
 - Para verificar se eles são iguais, usando as suas propriedades
- Criar um método *main* para comparar se os livros são iguais

Operador *instanceof*

- Verifica se uma variável de referência aponta para um objeto que é uma instância de uma **classe** ou implementa uma **interface**

```
public static void main(String[] args){  
    Animal animal = new Dog();  
    Dog dog = (Dog)animal;  
  
    if(animal instanceof Dog)  
        System.out.println("The animal is a dog!");  
    if(animal instanceof Cat)  
        System.out.println("The animal is a cat!");  
    if(animal instanceof CanFly)  
        System.out.println("It can fly!")  
}
```

Encapsulamento

Encapsulamento

- Técnica para fazer com que os **campos de uma classe sejam escondidos** e acessá-los via métodos públicos
 - Se um campo é declarado privado, ele não pode ser acessado fora da classe, escondendo, assim, os campos dentro da classe
- Pode ser descrito como uma barreira protetora que **impede que o código** e os dados **sejam acessados por outro código** definido fora da classe
- Provê fácil manutenção, flexibilidade e extensibilidade para o código
- Também conhecido como *Data Hiding*
- A forma mais comum de uso do Encapsulamento é através dos ***getters*** e ***setters***
- -- *Encapsulation is more about 'How' to achieve that functionality. [Implementation]*

Encapsulamento - Exemplo 1

```
public class Car{
    public float speed = 0;
    public boolean started = false;
    public boolean reverse = false;
}

public class Test{

    public static void main(String[] args){
        Car car = new Car();
        car.started = true;
        car.speed = 100;
        car.speed = 0;
        car.speed = 200;
        car.reverse = true;
        car.started = false;
    }
}
```

Encapsulamento - Exemplo 1

```
class Car {  
    private float speed = 0;  
    private boolean reverse = false;  
    private boolean started = false;  
  
    public void start() {  
        if (started) {  
            // the car is already started  
        } else {  
            started = true;  
        }  
    }  
    public void accelerate() {  
        if (started) {  
            speed += 10;  
        } else {  
            // car is not started yet  
        }  
    }  
    public void break() {  
        if (started) {  
            speed -= 10;  
        } else {  
            // car is not started yet  
        }  
    }  
    public void off() {  
        if (started) {  
            started = false;  
        } else {  
            // car is already off  
        }  
    }  
}
```

```
class Test{  
    public static void main(String[] a){  
        Car car = new Car();  
        car.off(); //car is already off  
        car.start();  
        car.accelerate();  
        car.accelerate();  
        car.accelerate();  
        car.break();  
        car.break();  
        car.break();  
        car.off();  
    }  
}
```

Qualquer classe que queira acessar os atributos "escondidos" deve fazê-lo através dos **métodos públicos**

Encapsulamento – Exemplo 2

```
class Person{
    private String name;
    private int age;

    public int getAge(){
        return age;
    }

    public String getName(){
        return name;
    }

    public void setAge(int newAge){
        age = newAge;
    }

    public void setName(String newName){
        name = newName;
    }
}
```

- **Getter:** Método usado para retornar o valor de um atributo privado*

```
public Tipo getNomeAtributo(){
    return nomeAtributo;
}
```

- **Setter:** Método usado para alterar o valor de um atributo privado*

```
public void setNomeAtributo(Tipo novo){
    nomeAtributo = novo;
}
```

*Em geral privado

```
public static void main(String args[]){
    Person p = new Person();
    p.setName("James");
    p.setAge(20);

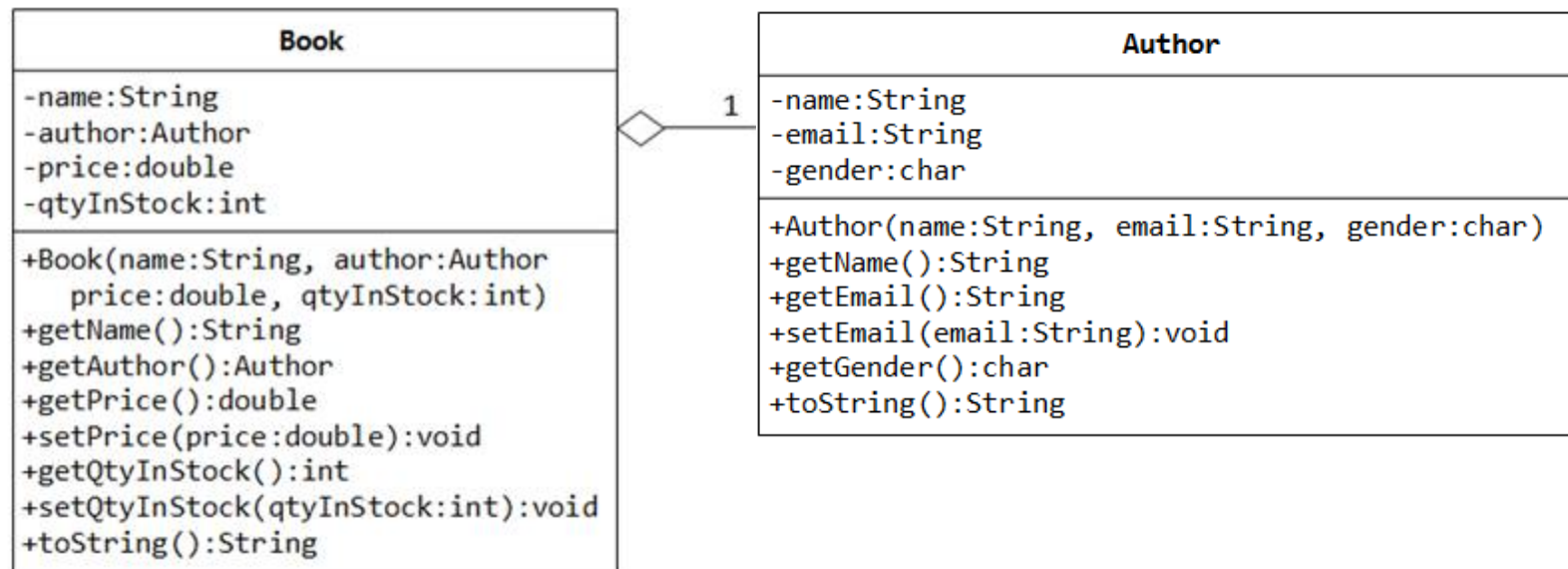
    System.out.print("Name : " +
p.getName() + " Age : " + p.getAge());
}
```

Encapsulamento - Benefícios

- Os campos de uma classe podem ser somente leitura ou somente escrita
 - Ou seja, podemos ter somente um dos métodos *getter* ou *setter*
 - **Não é só *getter* ou *setter***
- Uma classe tem **total controle** sobre o que está armazenado nela
- Quem usa a classe não sabe como a classe armazena seus dados
 - Uma classe pode alterar o tipo de um campo e os usuários da classe não precisam alterar seu código

Exercícios

Exercício Prático 1 – Conceitos OO

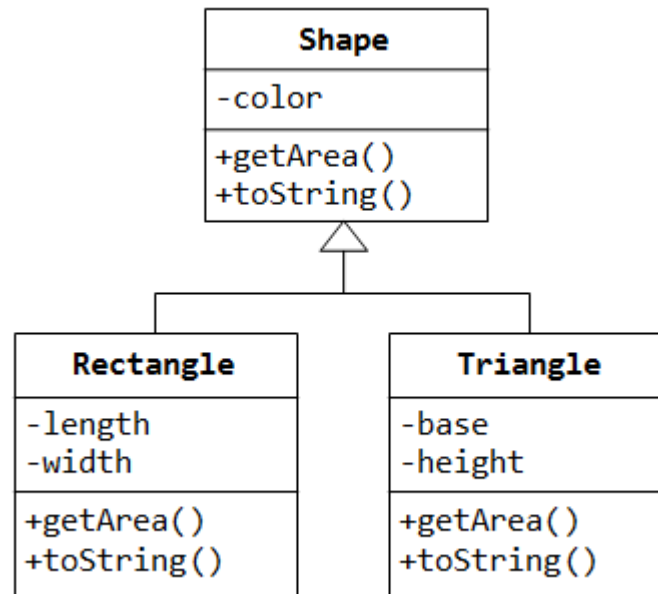


- Criar as duas classes e seu relacionamento
- Criar uma classe separada para testes que cria uma instância de cada uma das classes
 - Note que para criar um livro é preciso criar um autor
 - Imprimir o nome do livro, preço e quantidade no estoque
 - Através da instância de livro, recuperar o autor e imprimir o seu nome, e-mail e sexo
- Modificadores de acesso: + public e - private

Exercício Prático 2 – Herança e Abstração

- Criar as classes
 - Pessoa
 - CPF, nome e idade
 - Funcionario que estende de pessoa
 - Núm. de cadastro na empresa e salário
 - Gerente que estende de Funcionario
 - Vetor de funcionários e bônus anual
 - Cliente que estende de pessoa
 - Endereço e telefone
 - Cliente especial que estende de cliente
 - Desconto fixo em todas as compras
- Definir corretamente os modificadores de acesso
 - public, protected, package-private e private
- Lembrar das palavras reservadas:
 - **extends** e **abstract**

Exercício Prático 3 - Herança e Polimorfismo



- Criar as classes de acordo com o diagrama acima
 - Lembrar das palavras reservadas: **extends** e **abstract**
- Modificadores de acesso: + public e - private

Exercício Prático 4 - Encapsulamento

- Crie uma classe chamada Endereco que possua uma String rua e outra CEP
- Use o conceito de **Encapsulamento** para armazenar o valor do CEP de forma que a String que corresponda ao número do CEP fique sempre no formato: "XXXXXX-XXX" onde X representa um número de 0 a 9