

Programação Orientada a Objetos 2017/2



<https://goo.gl/F8dPBy>



GitHub


<https://github.com/mauro-hemerly/POO>

Mauro Hemerly (Hämmerli) Gazzani
maurog@kroton.com.br
mauro.hemerly@gmail.com

Release	Year
JDK Beta	1994
JDK 1.0	1996
JDK 1.1	1997
J2SE 1.2	1998
J2SE 1.3	2000
J2SE 1.4	2002
J2SE 5.0	2005
Java SE 6	2006
Java SE 7	2011
Java SE 8	2014



https://www.java.com/pt_BR/dowr



Pesquisar

Fazer Download Ajuda

Todos os Downloads do Java

Se você deseja fazer download do Java para outro computador ou Sistema Operacional, clique no link abaixo.
[Todos os Downloads do Java](#)

Reportar um problema

Download Gratuito do Java

Fazer o Download do Java para o seu computador desktop agora!

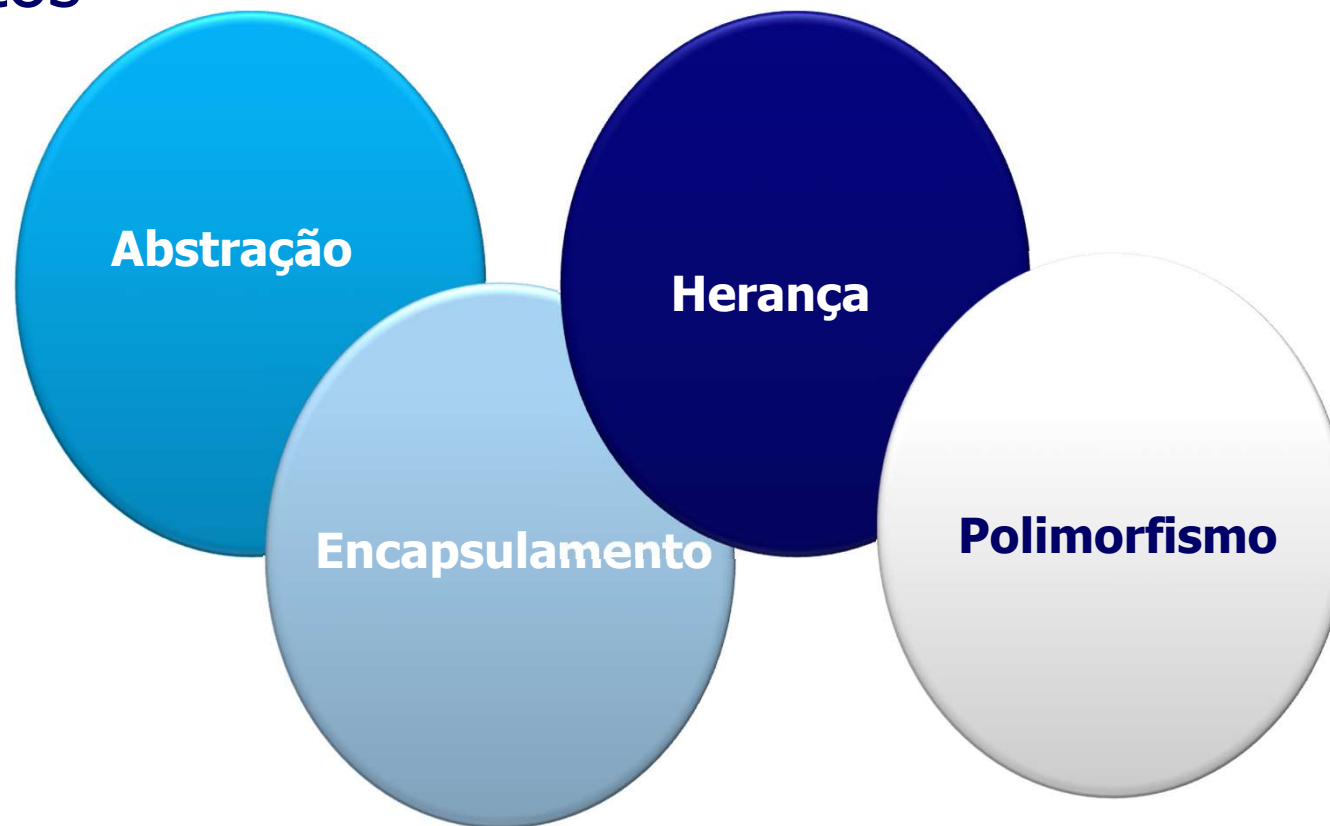
Version 8 Update 144
Data da release - 26 de julho de 2017

Download Gratuito do Java

[» O que é o Java?](#) [» Eu tenho Java?](#) [» Precisa de Ajuda?](#)

Pilares da POO

- Princípios Fundamentais do Paradigma Orientado a Objetos

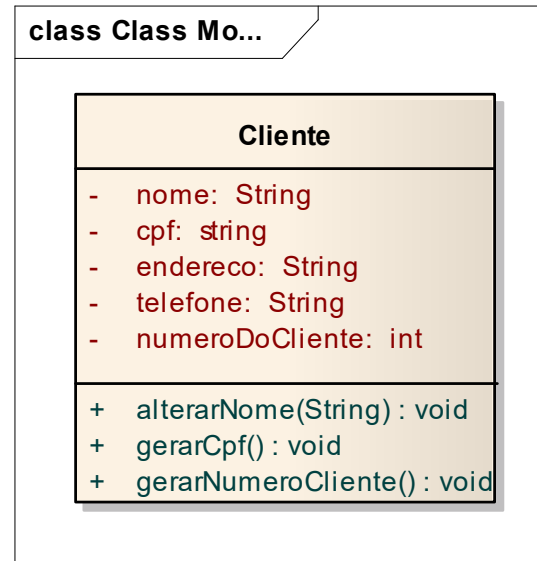
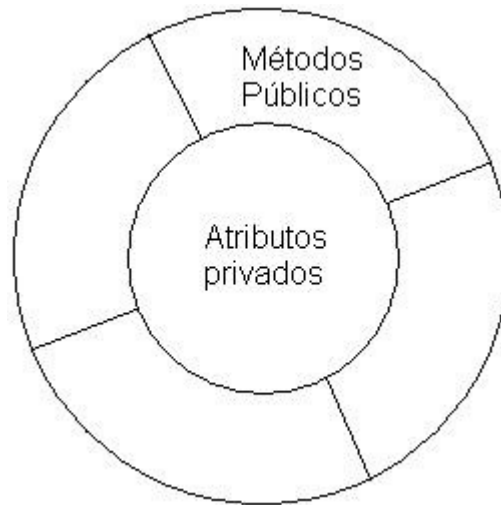


Abstração

- Nesta etapa “imaginamos” o nosso objeto: definindo a **identidade**, as **propriedades** e seus **métodos** (eventos).
 - ✓ **Identidade**: O nome do objeto a ser criado
 - ✓ **Propriedades**: São as características do objeto
 - ✓ **Métodos**: São os eventos, as ações que esse objeto irá executar (ou em outras palavras, como o estado do objeto será alterado).
- É a representação de algo real, na abstração analisamos o que é relevante conter em um objeto.

Encapsulamento

- O objetivo é ocultar as propriedades (atributos) de um objeto com a palavra reservada **private**
- Os valores dos atributos ficam acessíveis por meio de métodos.



Encapsulamento

■ Benefícios do encapsulamento

- ✓ Os atributos de uma classe podem ser feitas somente leitura ou somente gravação
- ✓ Uma classe pode ter total controle sobre o que está armazenado em seus atributos
- ✓ Os usuários de uma classe não sabem como a classe armazena seus dados. Uma classe pode alterar o identificador de um atributo, e os usuários da classe não precisam realizar qualquer alteração em seu código

Atributos e Métodos Estáticos

Atributos Estáticos (variáveis de classe)

- Há situações onde o valor de um atributo deve ser compartilhado entre todas as instâncias de uma classe;
- Exemplo: O valor da taxa CPMF cobrada por movimentações bancárias;
- Nestas situações utilizamos atributos estáticos; Ou seja, atributos cujos valores serão constantes para todas as instâncias de uma classe.

Atributos Estáticos

```
public class ClienteBanco {  
    String nome;  
    int conta;  
    float saldo;  
    static float taxa_CPMF = 0.01F; // Exemplo: 1%  
  
    void RealizaDeposito (float pValor) {  
        saldo = saldo + pValor*(1 - taxa_CPMF);  
    }  
}
```


Métodos Estáticos (métodos de classe)

- Analogamente aos atributos estáticos, podemos ter comportamentos que devem ser únicos (independente do objeto) em uma classe
- Por exemplo, um método para exibir dados estatísticos das contas: quantidade de clientes, volume de dinheiro total, etc.
- Ou seja, não faz sentido um objeto retornar uma consulta sobre um conjunto no qual ele está inserido
- Nestas situações utilizamos métodos estáticos
- Em livros costumamos encontrar os termos método de classe (estático) e métodos de instância (métodos comuns, aplicados a objetos)

Métodos Estáticos (métodos de classe)

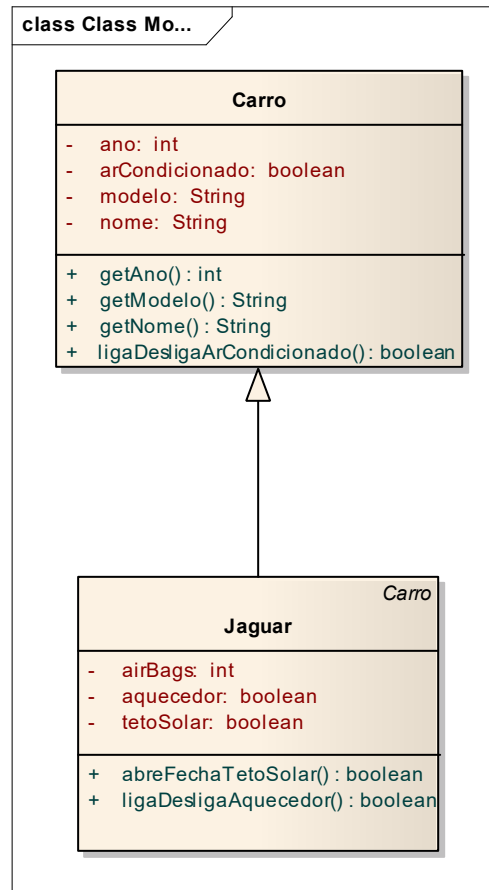
```
public class ClienteConta {  
    private String nome;  
    private int conta;  
    private float saldo;  
    private static float taxa_cpmf;  
    private static int qtd_clientes = 0;
```

```
    ClienteConta (String pNome, int pConta, float pSaldo) {  
        nome = pNome; conta = pConta; saldo = pSaldo;  
        qtd_clientes++;  
    }
```

```
    public static int QuantidadeClientes () {  
        return qtd_clientes;  
    }
```

```
public class Principal {  
    public static void main(String[] args) {  
        ClienteCC cliente1 = new ClienteCC("eu", 1, 5000, 500);  
        System.out.println("Quantidade de clientes: " +  
            ClienteConta.QuantidadeClientes());  
    }  
}
```

Herança



Herança

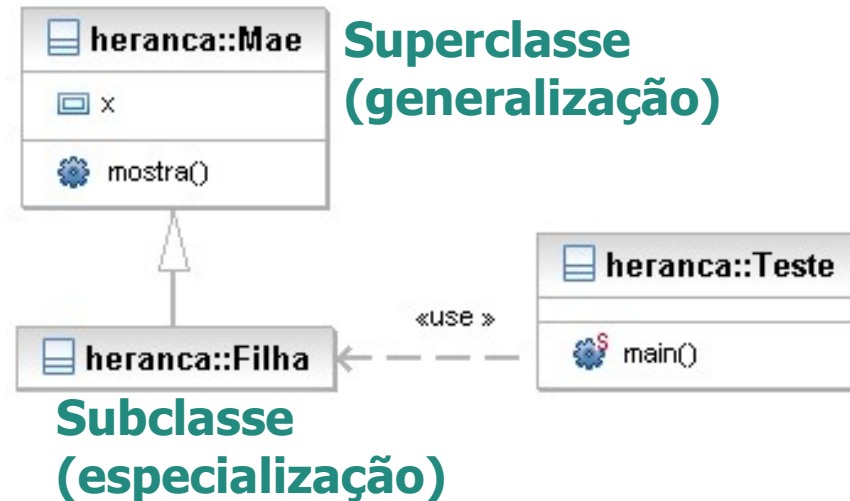
- A idéia é fornecer um mecanismo simples para que se defina novas classes a partir de uma já existente
- Assim sendo, dizemos que essas novas classes **herdam** todos os membros (propriedades+métodos) da **classe-mãe**
- Esta capacidade de **fatorar** as propriedades comuns de diversas classes em uma **superclasse** pode reduzir dramaticamente a repetição de código em um projeto ou programa, sendo uma das principais vantagens da abordagem de orientação a objetos

Herança

- A palavra **extends** indica que uma classe herda de outra classe

```
package heranca;  
class Mae { // superclasse  
    private int x = 12;  
    public void mostra() {  
        System.out.println(x);  
    };  
};
```

```
class Filha extends Mae {}; // subclasse  
public class Teste {  
    public static void main(String[] a) {  
        Filha f = new Filha(); f.mostra();  
    };  
};
```



Herança

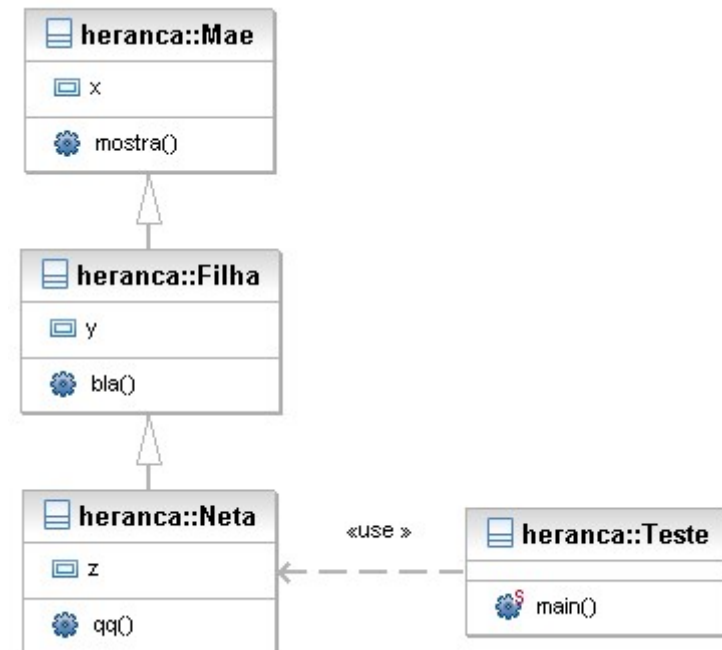
```
package heranca;

class Mae {
    private int x = 12;
    public void mostra() {
        System.out.println(x);
    };
};

class Filha extends Mae {
    private int y = 23;
    public void bla() {
        System.out.println(x+" "+y);
    };
};

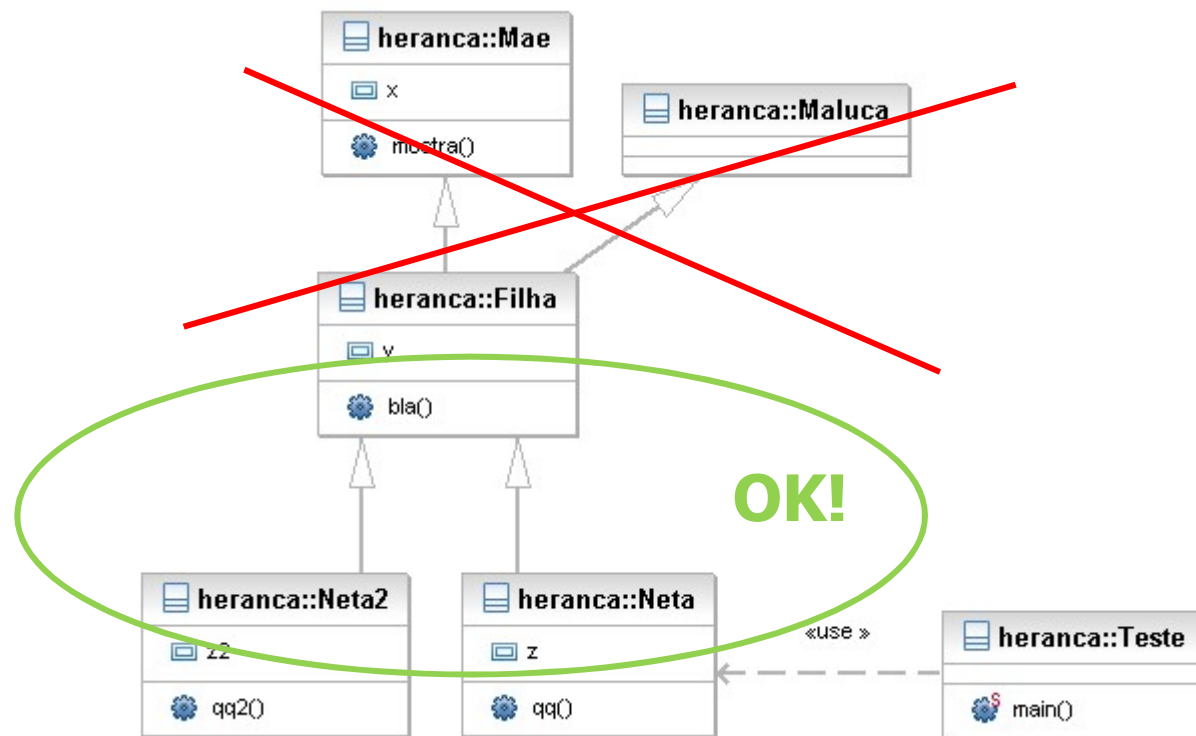
class Neta extends Filha {
    private int z = 1;
    public void qq() {
        System.out.println(x+" "+y+" "+z);
    };
};
```

```
public class Teste {
    public static void main(String[] args) {
        Neta neta = new Neta();
        neta.mostra(); // método mostra herdado da classe Mae por meio de Filha
        neta.bla();    // método bla herdado da classe Filha
        neta.qq();
    };
};
```



Herança


- Java **não suporta** herança múltipla como C++ e Object Pascal suportam. Assim uma classe Java só pode herdar de uma única classe



Herança

- O construtor de uma **subclasse** deve, obrigatoriamente, chamar o construtor da **superclasse** por meio da palavra reservada **super**
- Exemplo

```
class Mae {  
    int x;  
    public Mae(int x) { this.x = x }  
};  
  
class Filha extends Mae {  
  
    public Filha() {  
        super(10);  
    };  
};
```



Herança

- Quando não há uma chamada explícita no construtor da subclasse ao construtor da superclasse, a **JVM** adiciona uma chamada ao construtor padrão da superclasse
- Construtor padrão é um construtor que tem a lista de parâmetros vazia:

```
class Mae {  
    int x;  
    public Mãe() {};  
};
```

Sobrecarga

- Significa prover mais de uma versão de um mesmo método (qualquer método, inclusive métodos construtores)
- As versões devem, necessariamente, possuir listas de parâmetros diferentes, seja no tipo ou no número desses parâmetros
- **Tipo de retorno** diferente não configura sobrecarga

Sobrecarga

- Os métodos construtores podem ser sobrecarregados

```
class Ponto {  
    int x = 0;  
    int y = 0;  
    Ponto() { }  
    Ponto(int a, int b) {  
        x = a;  
        y = b;  
    };  
};
```

Sobrescrita

- Só ocorre em caso de **herança**
- Quando um método em uma **subclasse** possui o mesmo nome de um método na **superclasse**, mas com algumas características:
 - ✓ lista de argumentos iguais.
 - ✓ mesmo tipo de retorno
 - ✓ nível de acesso igual ou mais restritivo
- Não se pode sobrescrever método ***static*** e marcado com ***final*** (uma classe com ***final*** não pode ser estendida)
- Não se pode sobrescrever métodos ***private***, ou seja, se ele não pode ser herdado, não pode ser sobrescrito

Exercício

1) Dado o seguinte código,

```
1. public class Carga {  
2.   public int getPeso() {  
3.     return peso;  
4.   }  
5.   public void setPeso(int p) {  
6.     peso = p;  
7.   }  
8.   public int peso;  
9. }
```

O que é verdade sobre a classe acima (marque uma alternativa)?

- (A). A classe Carga é fortemente encapsulada.
- (B). Linha 2 está em conflito com encapsulamento.
- (C). Linha 5 está em conflito com encapsulamento.
- (D). Linha 8 está em conflito com encapsulamento.**
- (E). Linhas 5 e 8 estão em conflito com encapsulamento.
- (F). Linhas 2, 5 e 8 estão em conflito com encapsulamento.

Exercício

2) Dado o seguinte código,

```
class Teste {  
    String algumaCoisa(int x) { return "teste 1 2 3"; }  
}
```

e as seguintes opções de métodos de uma subclasse de Teste,

1. String algumaCoisa(int x) { return "teste"; }
2. String algumaCoisa(int x) { return "alo"; }
3. int algumaCoisa(int x) { return "42"; }
4. String algumaCoisa(String s) { return "Teste"; }
5. int algumaCoisa(String s) { return 42; }

Marque a alternativa correta:

- (A). Opções 4 e 5 são sobrescrita
- (B). Opções 1, 2 e 3 são sobrecarga
- (C). Opções 1, 2 são sobrescrita e 4, 5 são sobrecarga
- (D). Opções 1, 2 são sobrecarga e 4, 5 são sobrescrita
- (E). Opções 1, 3 e 5 são sobrescrita
- (F). Opções 1, 3 e 5 são sobrecarga

Exercício

3) Dado o seguinte código,

```
1. public class TesteHeranca {  
2.     public static void main(String [] args ){  
3.         Filho pai = new Filho();  
4.     }  
5. }  
6.  
7. class Pai {  
8.     public Pai() {  
9.         super();  
10.        System.out.println("instanciado um pai");  
11.    }  
12. }  
13.  
14. class Filho extends Pai {  
15.     public Filho() {  
16.         System.out.println("instanciado um filho");  
17.     }  
18. }
```

O que será impresso na execução do programa TesteHeranca?

- A. instanciado um filho
- B. instanciado um pai
- C. instanciado um filho
 instanciado um pai
- D. instanciado um pai
 instanciado um filho
- E. Nada será impresso.

Classes Abstratas

- Classes abstratas não podem ser instanciadas
- Uma classe abstrata é marcada com o modificador ***abstract***
- O seguinte código **NÃO** é válido:

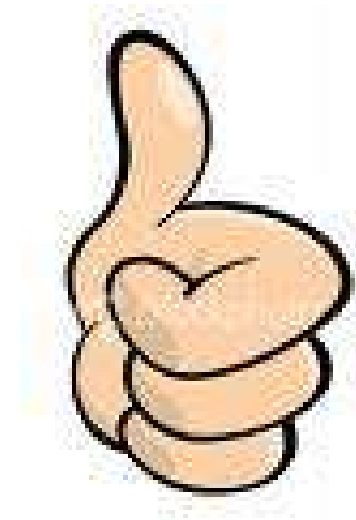
```
public abstract class Veiculo {  
    String tipo;  
    int nroEixos;  
    public void metodoConcreto() {}  
}
```

```
class Teste2 {  
    public static void main(String[] args) {  
        new Veiculo();  
    }  
}
```

O único propósito de “vida” de uma classe abstrata é ser especializada!

Classes Abstratas

```
public abstract class Veiculo {  
    String tipo;  
    int nroEixos;  
    public void metodoConcreto() {}  
}  
  
class Carro extends Veiculo {  
}  
  
class Teste2 {  
    public static void main(String[] a) {  
        new Carro();  
    }  
}
```



Classes Abstratas

```
abstract class Veiculo {  
    String tipo;  
    int nroEixos;  
    void metodoConcreto() {}  
    abstract void meio();  
}
```

Método abstrato!

```
class Carro extends Veiculo {  
    void meio() {  
        System.out.println("terrestre");  
    }  
}
```

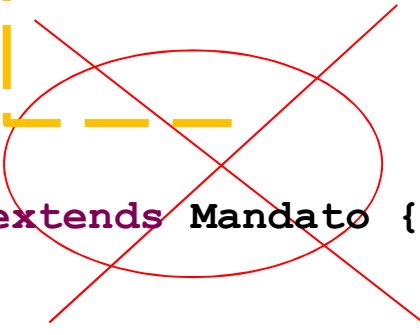
Toda primeira subclasse concreta de uma classe abstrata deve implementar todos os métodos abstratos herdados.

```
class Teste2 {  
    public static void main(String[] args) {  
        new Carro().meio();  
    }  
}
```

O modificador final

- Uma classe marcada como final não pode ser especializada.
- O código abaixo **NÃO** é válido:

```
package heranca;  
import java.util.Date;  
public final class Mandato {  
    Date inicio;  
    Date fim;  
}  
class NovoMandato extends Mandato {  
  
}
```



NÃO COMPILA!!!!

Interfaces

- Uma interface é um contrato!
- Quando você criar uma interface, estará definindo um contrato com o que a classe pode fazer, sem mencionar nada sobre como a classe o fará.
- Todos os **métodos** de uma interface são **públicos e abstratos**, e podem ou não usar os modificadores *public abstract*

Interfaces

- Toda classe que implementar (**implements**) o comportamento de uma interface deverá implementar todos os métodos da interface
- Uma classe pode implementar mais de uma interface
- Uma interface pode especializar (extends) de outra interface.
- Atributos de interfaces são sempre **public static final**. Vamos ver

Interface

```
package heranca;

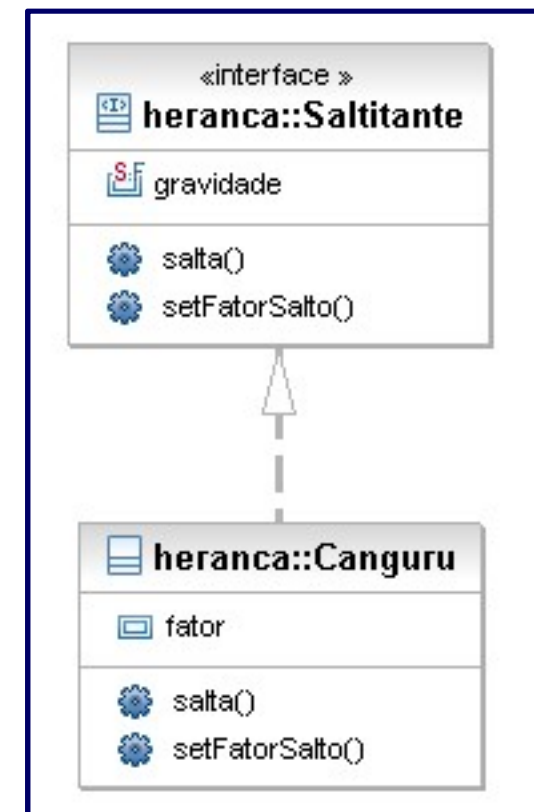
public interface Saltitante {
    double GRAVIDADE = 9.8; //é public static e final

    void salta(); //é public abstract
    void setFatorSalto(int fator); //é public abstract
}

class Canguru implements Saltitante {
    int fator;

    public void salta() {
        System.out.println("saltando!");
    }

    public void setFatorSalto(int fator) {
        this.fator = fator;
    }
}
```



Herança de interface:

```
interface Corredor {
    void corre();
}

interface Saltitante extends Corredor {
    double GRAVIDADE = 9.8; //é public static e final
    void salta();
    void setFatorSalto(int fator);
}

class Canguru implements Saltitante {
    int fator;
    public void corre() {
        System.out.println("correndo!");
    }
    public void salta() {
        System.out.println("saltando!");
    }
    public void setFatorSalto(int fator) {
        this.fator = fator;
    }
}
```



```

interface Saltitante extends Corredor {
    double GRAVIDADE = 9.8;
    void salta();
    void setFatorSalto(int fator);
}

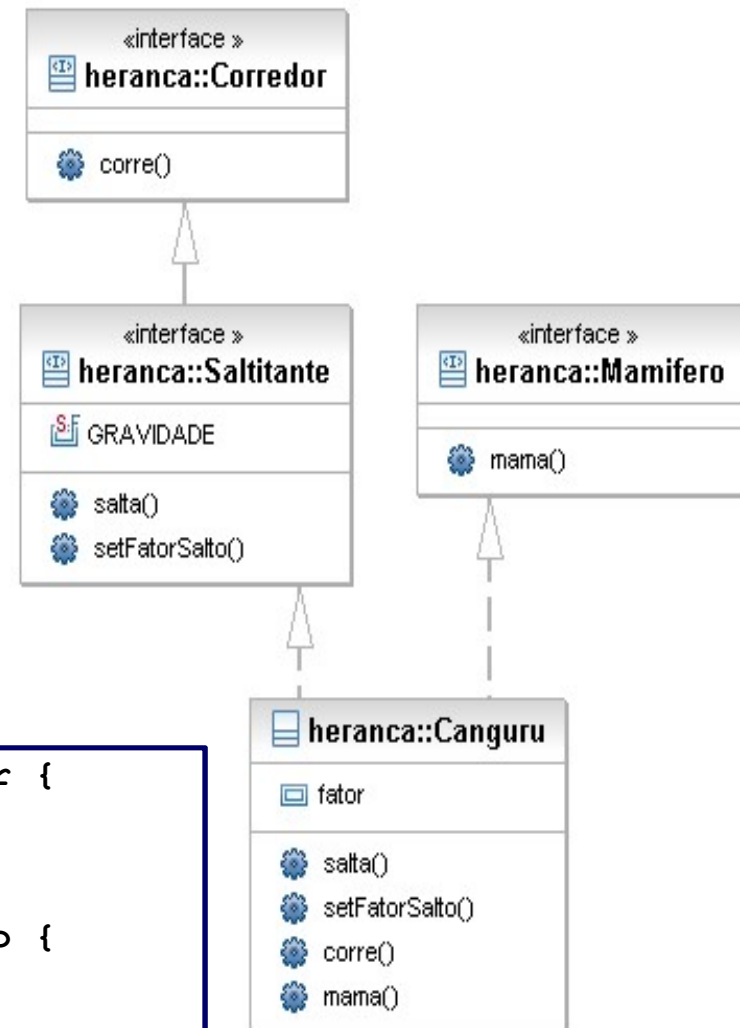
class Canguru implements Saltitante,
                        Mamifero {

    int fator;
    public void corre() {
        System.out.println("correndo!");
    }
    public void salta() {
        System.out.println("saltando!");
    }
    public void mama() {
        System.out.println("mamando!");
    }
    public void setFatorSalto(int fator) {
        this.fator = fator;
    }
}
  
```

```

interface Corredor {
    void corre();
}

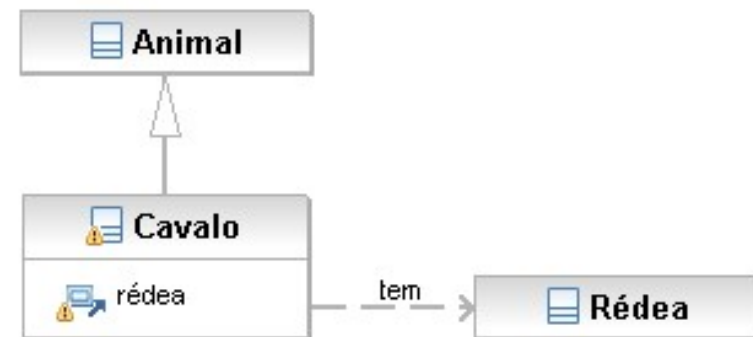
interface Mamifero {
    void mama();
}
  
```



Relacionamentos É-UM e TEM-UM:

- Na OO, o conceito É-UM é baseado na herança de classes ou na implementação de interfaces
- Os relacionamentos TEM-UM são baseados na utilização, em vez de na herança
- Vamos analisar um exemplo simples com 3 classes: Animal, Cavalo e Rédea
- O Cavalo É-UM Animal. O Cavalo TEM-UMA Rédea

```
public class Animal {  
}  
public class Cavalo extends Animal {  
    private Rédea rédea;  
}  
public class Rédea {  
}
```



Modificadores de Acesso

- Enquanto uma classe só pode usar 2 dos níveis de acesso (público ou default), um membro (atributo ou método) pode usar até quatro níveis:

público (public)

protegido (protected)

default

privativo (private)

Mais restrito

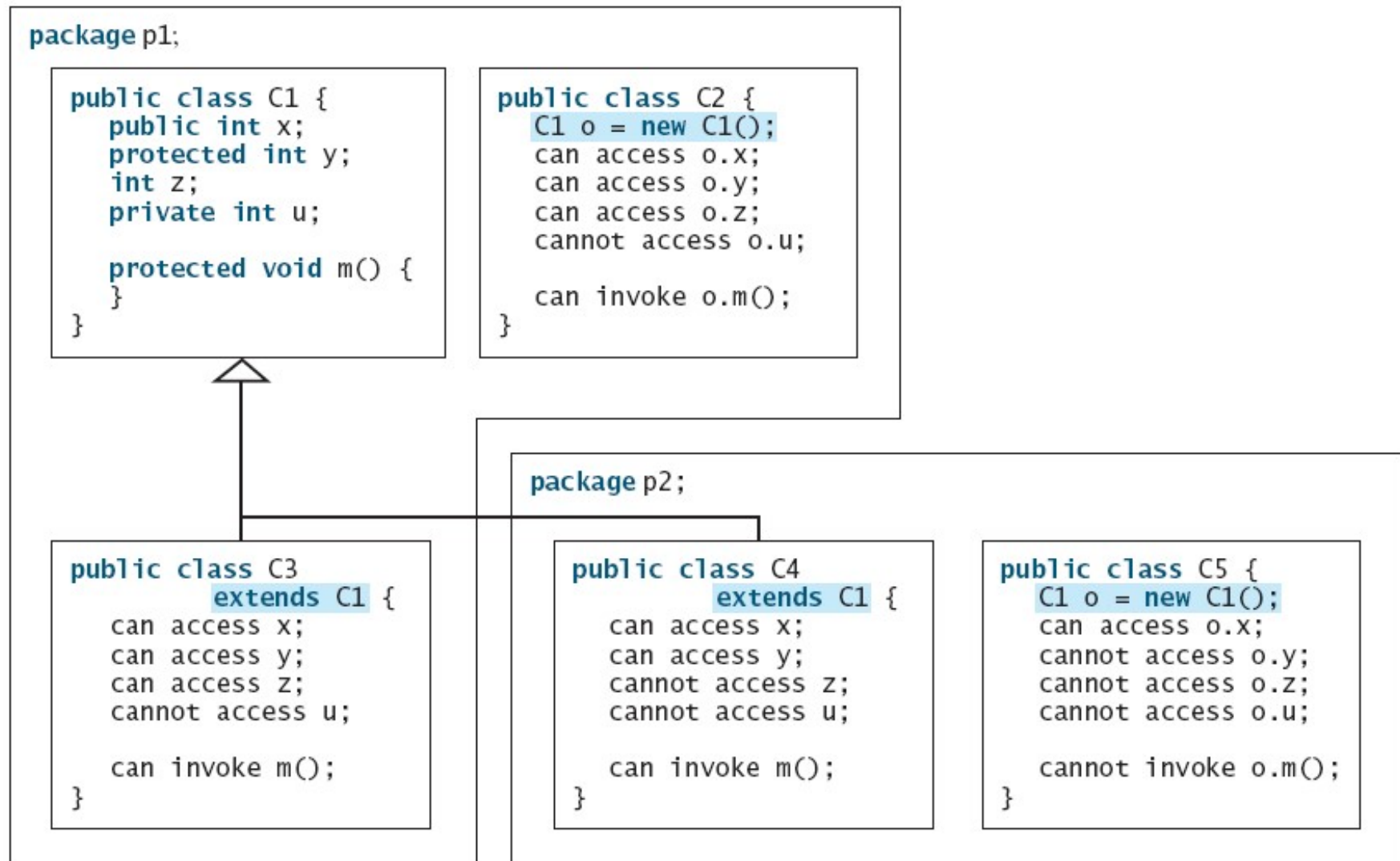


- Duas questões diferentes relativas ao acesso de membros:
 - ✓ se o código de um método em uma classe pode acessar um membro de outra classe
 - ✓ se uma subclasse pode herdar um membro da sua superclasse.

Modificadores de Acesso

- Os modificadores de acesso para membros respeitam antes o modificador de acesso da classe, assim sendo, primeiro a classe deve ser visível para depois verificar a visibilidade do membro.
 - ✓ **Membros private**
 - só podem ser acessados na classe em que foi declarado
 - ✓ **Membros default**
 - só podem ser acessado se a classe que o estiver acessando pertencer ao mesmo pacote
 - ✓ **Membros protected**
 - são similares aos membros default, porém podem ser acessados fora do pacote por meio de herança
 - ✓ **Membros public**
 - qualquer classe pode acessar o membro em questão desde que a própria classe seja visível.

Modificadores de Acesso



Modificadores de Acesso

Nesse exemplo a classe A não é visível à C e portanto mesmo tendo um atributo público nada da classe A será visto pela classe C, nem mesmo a classe:

```
package visibilidade;  
  
class A {  
    public String nome = "Classe A";  
}  
  
package visibilidade.outra;  
  
import visibilidade.A;  
  
public class C {  
    public static void main(String[] args) {  
        A a = new A();  
    }  
}
```

A classe visibilidade.A não é pública então não é "vista" pela classe visibilidade.outra.C!!

Obs.: se a classe C estivesse no mesmo pacote da classe A estaria tudo OK!!

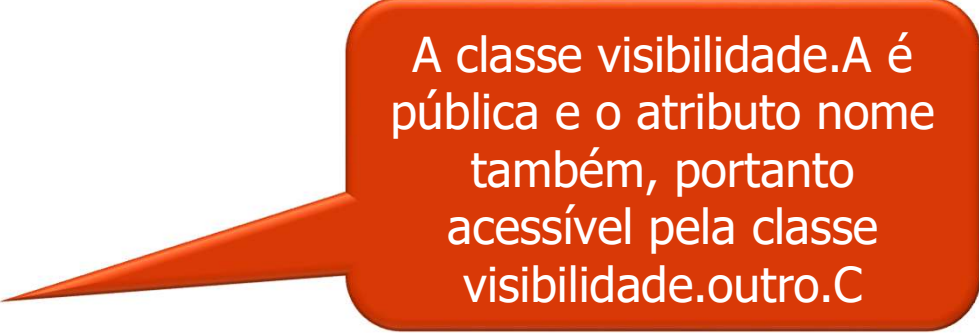
Modificadores de Acesso

Nesse exemplo a classe A é visível à C e portanto o atributo público nome da classe A é acessível à classe C:

```
package visibilidade;

public class A {
    public String nome = "Classe A";
}

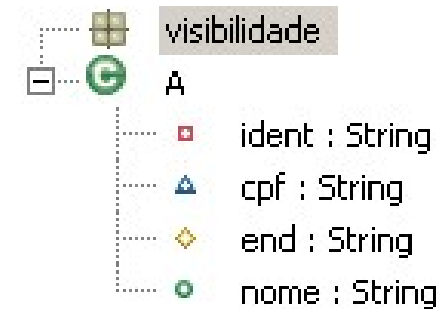
package visibilidade.outro;
import visibilidade.A;
public class C {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.nome);
    }
}
```



A classe visibilidade.A é pública e o atributo nome também, portanto acessível pela classe visibilidade.outro.C

Modificadores de Acesso

```
package visibilidade;  
  
public class A {  
    private String ident = "AA";  
    String cpf = "88";  
    protected String end = "Rua";  
    public String nome = "Classe A";  
}
```

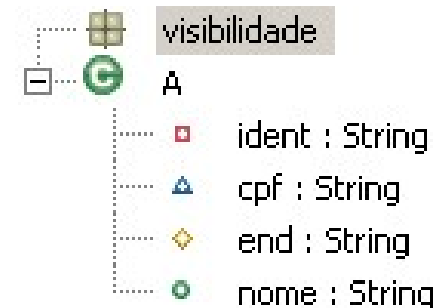


```
package visibilidade.outro;  
import visibilidade.A;  
  
public class C {  
    public static void main(String[] args) {  
        A a = new A();  
        System.out.println(a.nome);  
        System.out.println(a.ident);  
        System.out.println(a.end);  
        System.out.println(a.cpf);  
    }  
}
```

Apenas o atributo público
nome da classe visibilidade.A
é acessível da classe
visibilidade.outro.C!!

Modificadores de Acesso

```
package visibilidade;  
  
public class A {  
    private String ident = "AA";  
    String cpf = "88";  
    protected String end = "Rua";  
    public String nome = "Classe A";  
}
```

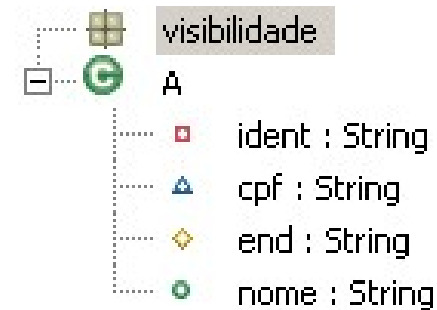


```
package visibilidade.outro;  
import visibilidade.A;  
public class D extends A {  
    public static void main(String[] args) {  
        D d = new D();  
        System.out.println(d.nome);  
        System.out.println(d.ident);  
        System.out.println(d.end);  
        System.out.println(d.cpf);  
    }  
}
```

Apenas os atributos nome (*public*) e end (*protected*) da classe visibilidade.A são acessíveis da classe visibilidade.outro.D!!

Modificadores de Acesso

```
package visibilidade;  
public class A {  
    private String ident = "AA";  
    String cpf = "88";  
    protected String end = "Rua";  
    public String nome = "Classe A";  
}
```



```
package visibilidade;  
public class E extends A {  
    public static void main(String[] args) {  
        E e = new E();  
        System.out.println(e.nome);  
        System.out.println(e.ident);  
        System.out.println(e.end);  
        System.out.println(e.cpf);  
    }  
}
```

Apenas o atributo ident
(*private*) da classe
visibilidade.A **NÃO** é acessível
da classe visibilidade.E!!

ex1) Preencher o quadro abaixo, com relação à visibilidade de membros de classe (considerar a classe visível)

Visibilidade	Public	Protected	Default	Private
A partir da mesma classe				
A partir de qq classe do mesmo pacote				
A partir de uma subclasse do mesmo pacote				
A partir de uma subclasse de fora do mesmo pacote				
A partir de qq classe que não seja subclasse e esteja fora do pacote				

Variáveis locais e modificadores de acesso

- Já vimos que os atributos de uma classe são sempre inicializados com valores default enquanto que variáveis locais NÃO são
- De forma similar NÃO existem modificadores de acesso aplicados às variáveis locais.
- Isso ocorre, naturalmente, porque as variáveis locais são vistas apenas no escopo local e não necessitam de modificador para isso.
- Na verdade, o único modificador que pode ser aplicado a variáveis locais é o final.
- Uma variável local final só pode ter uma inicialização!

ex2) os códigos abaixo compilam?

a)

```
public static void main(String[] args) {  
    final String teste;  
    teste = "asl";  
}
```

b)

```
public static void main(String[] args) {  
    final String teste = "";  
    teste = "afl";  
}
```

c)

```
public static void main(String[] args) {  
    final String teste = null;  
    teste = "sfl";  
}
```

Outros Modificadores

- Além dos modificadores de acesso temos outros modificadores.
- Para variáveis
 - ✓ final, static, transient e volatile
- Para métodos
 - ✓ final, static, abstract, synchronized, native, strictfp
- Alguns desses modificadores raramente, ou nunca, serão necessários em nossos sistemas diários.
- Os mais importantes são: final, static e abstract

Polimorfismo

- Tipo da referência: **Funcionario func**
- Tipo do objeto : **new Funcionario()**
- O tipo da referência, ou **tipo estático**, é o tipo que é declarado e usado pelo **compilador**
- O tipo do objeto, ou **tipo dinâmico**, é o tipo original do objeto e pode ser usado apenas pela **JVM** (ambiente de execução).

Funcionario func = new Funcionario();

Polimorfismo

■ Subtipagem

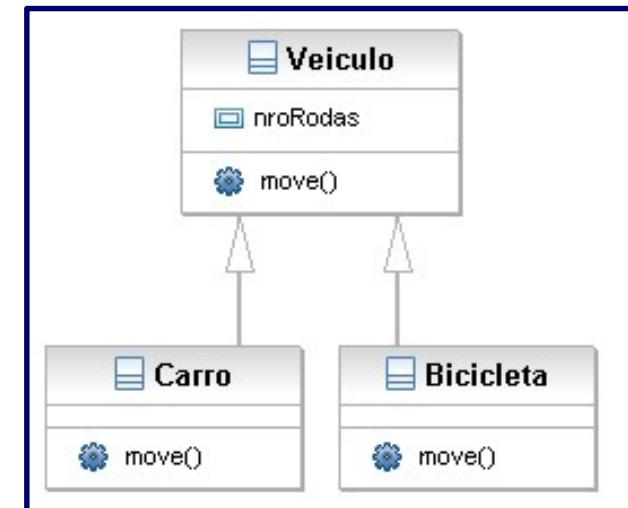
- ✓ Um objeto pode assumir, naturalmente, em tempo de execução, qualquer tipo da sua hierarquia Superior :

```
class Veiculo {  
    protected int nroRodas;  
    public void move() {  
        System.out.println("Coisa movimentando! "+this);  
    }  
}  
  
class Carro extends Veiculo {  
    public void move() {  
        System.out.println("Carro movendo! "+this);  
    }  
}  
  
class Bicicleta extends Veiculo {  
    public void move() {  
        System.out.println("Bicicleta andando! "+this);  
    }  
}
```

Polimorfismo

```
public class Teste {
    public static void main(String[] args) {
        Carro carro1 = new Carro();
        Carro carro2 = new Carro();
        Veiculo carro3 = new Carro(); // upcasting
        Bicicleta bike1 = new Bicicleta();
        Veiculo bike2 = new Bicicleta(); // upcasting
        Veiculo veiculo1 = new Veiculo();
        Bicicleta bike3 = (Bicicleta) new Veiculo(); // downcasting
        // downcasting: passa pela compilação, mas "dispara" exceção na execução !

        carro1.move();
        carro2.move();
        carro3.move();
        bike1.move();
        bike2.move();
        veiculo1.move();
    }
}
```



```
<terminated> Teste (4) [Java Application] C:\Arquivos de programas\Java\jre1.6.
Carro movendo! polimorfismo.Carro@42e816
Carro movendo! polimorfismo.Carro@9304b1
Carro movendo! polimorfismo.Carro@190d11
Bicicleta andando! polimorfismo.Bicicleta@a90653
Bicicleta andando! polimorfismo.Bicicleta@de6ced
Coisa movimentando! polimorfismo.Veiculo@c17164
```


Polimorfismo - Exercício

```
class Animal {  
    public void eat() {  
        System.out.println("Animal genérico comendo qq. coisa!");  
    }  
}  
  
class Cavalo extends Animal {  
    public void eat() {  
        System.out.println("Cavalo comendo grama!");  
    }  
    public void eat(String s) {  
        System.out.println("Cavalo comendo "+s);  
    }  
}
```

Obs.: lembrar que o tipo do objeto (**dinâmico - em tempo de execução**) decide qual método **sobrescrito** será invocado. No caso da **sobrecarga**, é o tipo da referência (**estático - em tempo de compilação**) que decide.

Exercício 3

Código de chamada de método	Resultado
<code>Animal a = new Animal(); a.eat();</code>	
<code>Cavalo h = new Cavalo(); h.eat();</code>	
<code>Animal ac = new Cavalo(); ac.eat();</code>	
<code>Cavalo he = new Cavalo(); he.eat("maça")</code>	
<code>Animal a2 = new Animal(); a2.eat("granola");</code>	
<code>Animal ah2 = new Cavalo(); ah2.eat("cenoura");</code>	

Respostas ex1

Visibilidade	Public	Protected	Default	Private
A partir da mesma classe	Sim	Sim	Sim	Sim
A partir de qq classe do mesmo pacote	Sim	Sim	Sim	Não
A partir de uma subclasse do mesmo pacote	Sim	Sim	Sim	Não
A partir de uma subclasse de fora do mesmo pacote	Sim	Sim (só por herança)	Não	Não
A partir de qq classe que não seja subclasse e esteja fora do pacote	Sim	Não	Não	Não

Respostas ex2

a) **Sim**

b) **Não**

c) **Não**

Respostas ex3

Código de chamada de método	Resultado
<pre>Animal a = new Animal(); a.eat();</pre>	Animal genérico comendo qq. coisa!
<pre>Cavalo h = new Cavalo(); h.eat();</pre>	Cavalo comendo grama!
<pre>Animal ac = new Cavalo(); ac.eat();</pre>	Cavalo comendo grama!
<pre>Cavalo he = new Cavalo(); he.eat("maça")</pre>	Cavalo comendo maçã!
<pre>Animal a2 = new Animal(); a2.eat("granola");</pre>	Erro de compilação!!!
<pre>Animal ah2 = new Cavalo(); ah2.eat("cenoura");</pre>	Erro de compilação!!!