



Curso de **Java8** para **Web**

Professor
Antonio Benedito Coimbra Sampaio Jr

abc  | Treinamentos

www.abctreinamentos.com.br

Primeira Disciplina

JAVA 8 - Fundamentos Teóricos e Orientação a Objetos

- **UNIDADE 1:** Introdução à Tecnologia Java
- **UNIDADE 2:** Introdução à Sintaxe Java
- **UNIDADE 3:** Programação Orientada a Objetos em Java (Parte I)
- **UNIDADE 4: Programação Orientada a Objetos em Java (Parte II)**

UNIDADE 4

PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA (PARTE II)

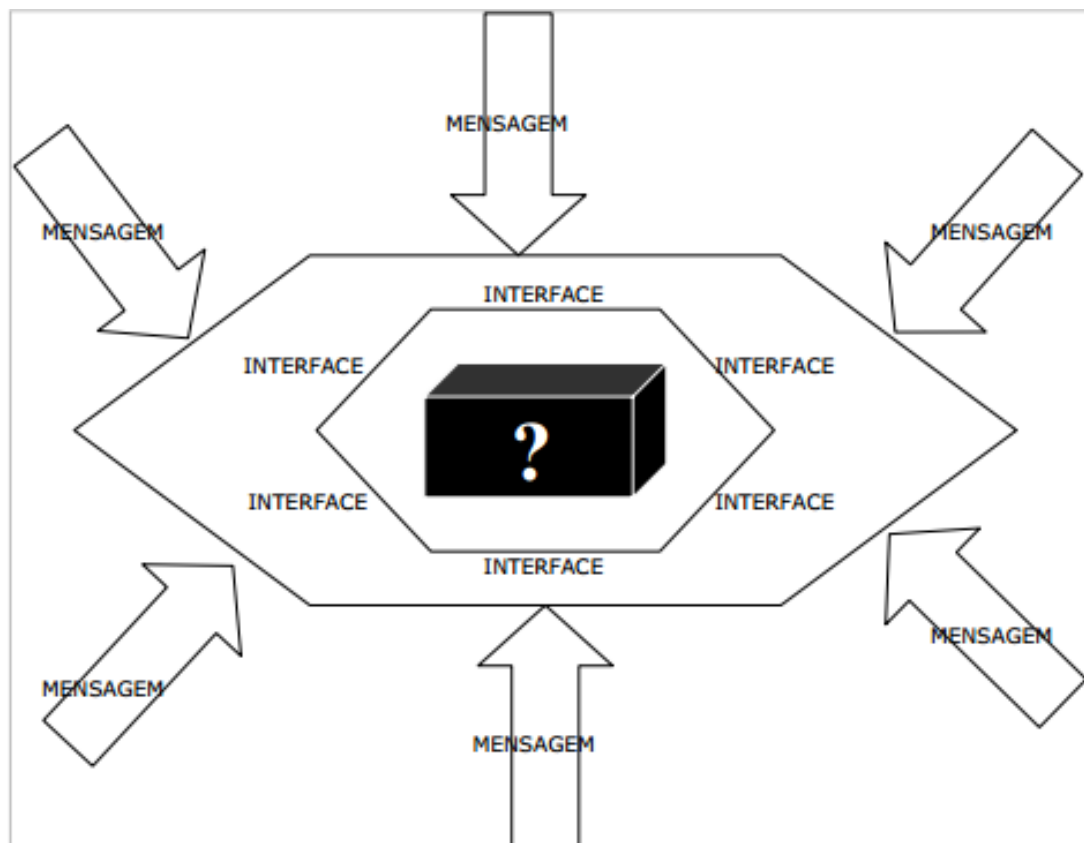
Encapsulamento e os Modificadores de Acesso

“Pilares” da Orientação a Objetos

- São três os “pilares” da Orientação a Objetos:
 - **ENCAPSULAMENTO**
 - **HERANÇA**
 - **POLIMORFISMO**
- O **Encapsulamento** serve para controlar o acesso aos atributos e métodos de uma classe, tendo por objetivo criar um software mais flexível, fácil de modificar e de criar novas implementações.
- A **Herança** permite a criação de novas classes (subclasses) com as mesmas características (atributos) e os mesmos comportamentos (métodos) de uma classe já existente (superclasse).
- O **Polimorfismo** é o princípio pelo qual um objeto pode ser referenciado de várias formas.

Encapsulamento

- O **encapsulamento** permite a visualização de uma entidade de software (neste caso, **uma classe**) como uma caixa preta. Neste caso, sabe-se o que a classe faz, sem ter acesso ao seu comportamento interno, possibilitando esconder os detalhes da implementação realizada.



Encapsulamento

- O acesso às funcionalidades dessas classes é feito via **troca de mensagens**, isto é, a chamada aos seus métodos.
- O encapsulamento é fundamental para garantir que as mudanças em uma determinada classe, não afete o funcionamento das outras classes que se relacionam com ela, visto que as regras de negócios ficam definidas em apenas um único lugar.
- Simplifica o acesso a um determinado objeto, expondo apenas a sua interface essencial.
- Por definição, **os atributos de uma classe devem ser de acesso restrito, e os seus métodos devem ser de acesso público.**

Modificadores de Acesso

- Os tipos de acesso aos atributos e métodos de uma classe são definidos pelo uso adequado dos seus **Modificadores de Acesso**.
- No Java são quatro os **Modificadores de Acesso**:
 - **PUBLIC (+)**
 - **PRIVATE (-)**
 - **PACKAGE (~)**
 - **PROTECTED (#)**

Modificadores de Acesso

PUBLIC (+)

- **Acessível** na própria classe, nas subclasses, nas classes do mesmo pacote e por todas as outras classes.
- Resumindo: é acessível por todo mundo!
- **Deve ser** utilizado preferencialmente para construtores e métodos que fazem parte da interface do objeto.
- **Evite usar** em construtores, métodos de uso restrito e campos de dados de objetos.

Classe
+campoPublico: tipo
+metodoPublico: tipo

PRIVATE (-)

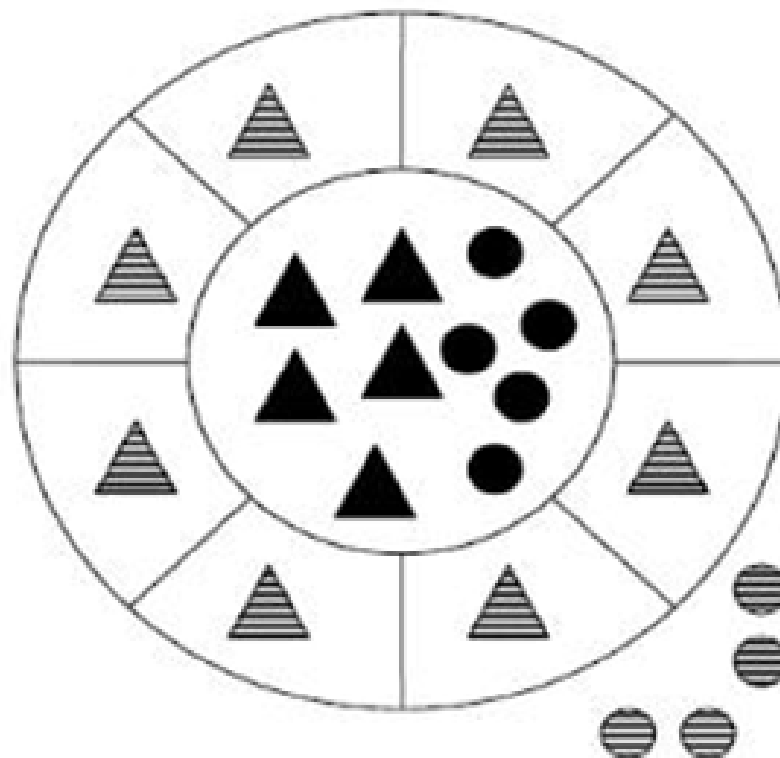
- **Acessível apenas na própria classe!**
- **Use** para métodos que não fazem parte da interface do objeto e **para campos de dados de objetos**.

Classe
-campoPrivate: tipo
-metodoPrivate: tipo

Public x Private

CLASSE

-  métodos públicos
-  métodos privados
-  dados privados
-  dados públicos (não recomendável)



© [Tiago Eugenio](#)

Exemplo

```
class Pessoa {  
    private String nome; private int idade;  
    public String obs;  
    Pessoa (String nome, int idade) {  
        this.nome = nome;  
    }  
    public void imprimeNome() {  
        System.out.println("Nome: " + nome);  
    }  
}
```

```
...  
Pessoa p1 = new Pessoa("Joao",10);  
p1.nome = "Raul";//ERRO DE COMPILAÇÃO!  
p1.obs = "bom garoto"; //OK!  
p1.imprimeNome(); //Nome: Joao
```

Modificadores de Acesso

PROTECTED (#)

- **Acessível** na própria classe, nas subclasses e nas classes do mesmo pacote.
- **Use** para métodos que devem ser sobrepostos.
- **Evite usar** em métodos com restrição à sobreposição e em campos de dados de objetos.

Classe
#campoProt: tipo
#metodoProt: tipo

PACKAGE (~)

- **Acessível** na própria classe, nas subclasses e nas classes do mesmo pacote.
- Se não houver outro modificador de acesso, o acesso é desse tipo.
- **Use** para construtores e métodos que só devem ser chamados pelas classes e subclasses do pacote.
- **Evite usar** em campos de dados de objetos.

Classe
~campoAmigo: tipo
~metodoAmigo: tipo

Modificadores de Acesso

- Resumo

Modifier	Class	Package	Subclass	World
<code>public</code>	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	✗
<code>no modifier*</code>	✓	✓	✗	✗
<code>private</code>	✓	✗	✗	✗

Métodos Getters e Setters

- Para permitir o acesso aos **atributos restritos (private)** de uma forma controlada, utilizam-se dois métodos: o primeiro para recuperar o valor do atributo (**get**) e o segundo para atribuir o valor do atributo (**set**).
- A convenção para esses métodos é de colocar a palavra **get** ou **set** antes do nome do atributo.

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

- É importante ressaltar que é uma má prática criar uma classe e, logo em seguida, criar **getters** e **setters** para todos seus atributos. Métodos **getters** e **setters** só se houver real necessidade.

Java Bean

- Quando se cria uma classe com todos os seus atributos privados, todos os seus métodos **get** e **set** e um construtor vazio (padrão), na verdade está se construindo um **componente Java (Java Bean)**.

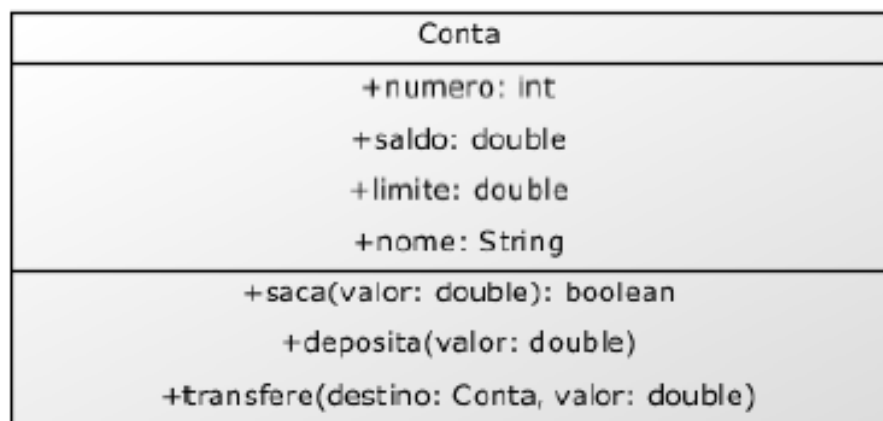
```
public class MyBean implements Serializable {  
    private int margin;  
  
    public MyBean() { }  
    public int getMargin() { return margin; }  
    public void setMargin(int margin) { this.margin = margin; }  
}
```

- Um Java Bean não é um EJB (*Enterprise Java Beans* - tecnologia de objetos distribuídos em Java).

```
@Stateless  
public class CustomerService {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    public void addCustomer(Customer customer) {  
        entityManager.persist(customer);  
    }  
}
```

Exercícios

- 1) Transforme as classes (**Cliente**, **Locacao** e **Carro**) do sistema de informação que gerencie o aluguel (**sisalucar**) de uma frota de carros em **Java Beans**. Faça as correções necessárias na classe **SisalucarApp**.
- 2) Transforme a classe Conta em um **Java Bean**. Implemente todos os seus três métodos, considerando a seguinte Regra de Negócio:
 - **RN01 – Só é possível sacar até saldo+limite.**



- 3) Responda se era necessário criar todos os métodos **get** e **set** da classe **Conta**.

Reuso

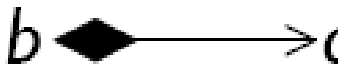
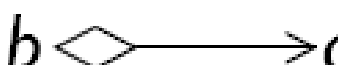
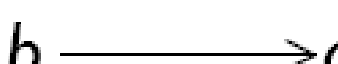

Reuso

- O **Reuso** é inerente ao processo de solução de problemas utilizado pelos seres humanos.
- Na medida em que soluções são encontradas, estas são utilizadas em problemas similares.
- Reutilização de software é o processo de criação de sistemas de software a partir de software existente ao invés de construí-los a partir do zero.
- Para entregar software de qualidade em menos tempo, é preciso reutilizar.
- O Reuso veio para agilizar o processo de desenvolvimento de software, aumentando a sua confiabilidade e diminuindo o seu custo, de uma forma simples e contemplando as boas práticas de engenharia de software.

Reuso

- Reuso é uma das principais vantagens anunciadas pela Orientação a Objetos.

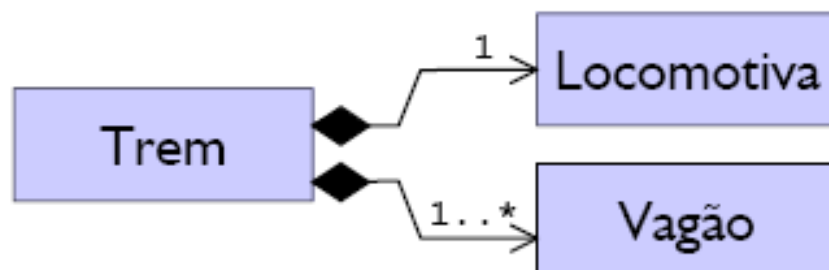
- São 04 as suas principais formas de reuso:

- Composição: *a “é parte essencial de” b* 
- Agregação: *a “é parte de” b* 
- Associação: *a “é usado por” b* 
- Herança: *b “é” a* (substituição pura) 
ou *b “é um tipo de” a* (substituição útil, extensão)

© Helder da Rocha

Reuso - Composição

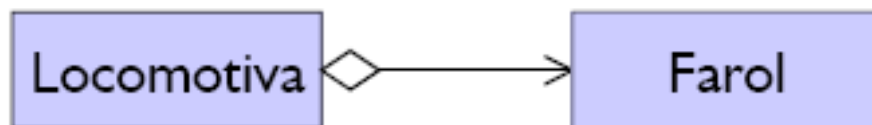
- Criação de uma nova classe usando classes existentes como atributos.
- **Composição:** *um trem é formado por locomotiva e vagões*



```
import java.util.*;
class Trem
{
    Locomotiva locomotiva = new Locomotiva();
    Collection<Vagao> vagao = new ArrayList<Vagao>();
    ...
}
```

Reuso - Agregação

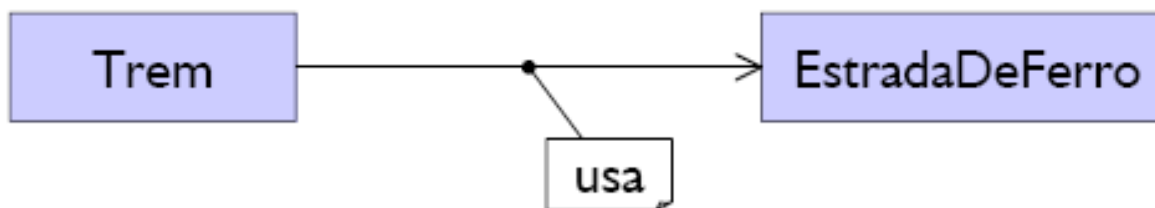
- Criação de uma nova classe usando classes existentes como atributos.
- **Agregação:** *uma locomotiva **tem** um farol (mas não vai deixar de ser uma locomotiva se não o tiver)*



```
import java.util.*;
class Locomotiva
{
    Collection<Farol> farol = new ArrayList<Farol>();
    ...
}
```

Reuso - Associação

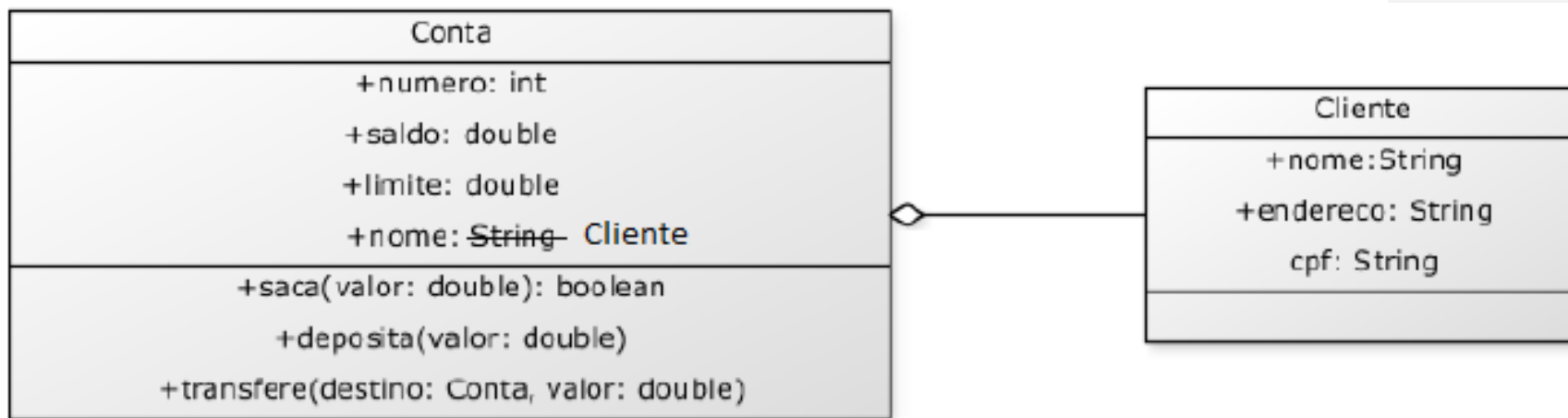
- Criação de uma nova classe fazendo uso de outras usando classes (acoplamento menor).
- **Associação:** *um trem **usa** uma estrada de ferro (não faz parte do trem, mas ele depende dela)*



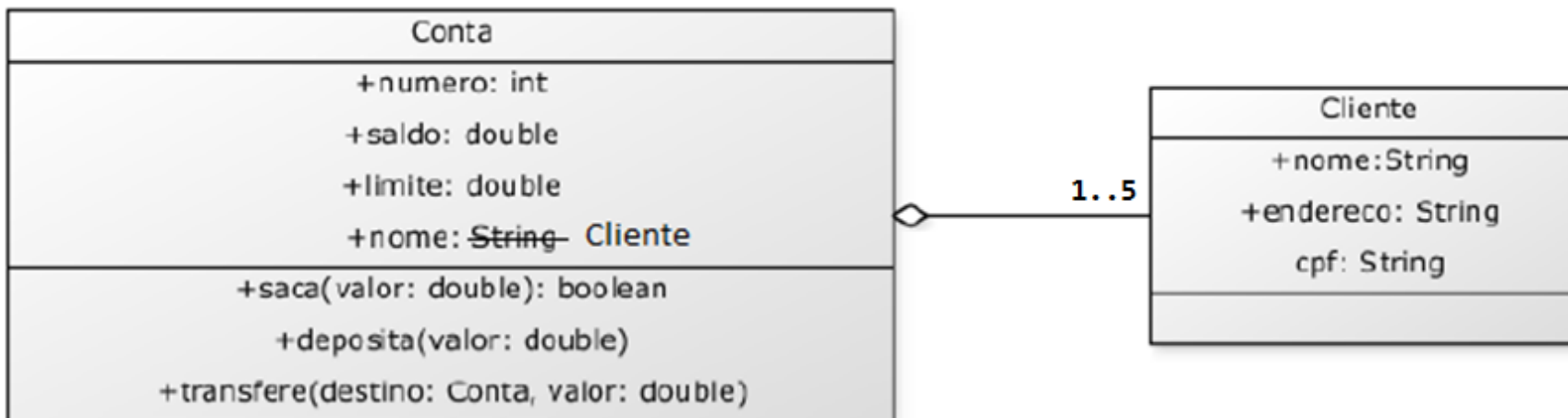
```
class Trem
{
    EstradaDeFerro estrada = new EstradaDeFerro();
    ...
}
```

Exercícios

- 1) Utilize a técnica de **Reuso** mais adequada para representar o relacionamento das classes **Conta** e **Cliente**.



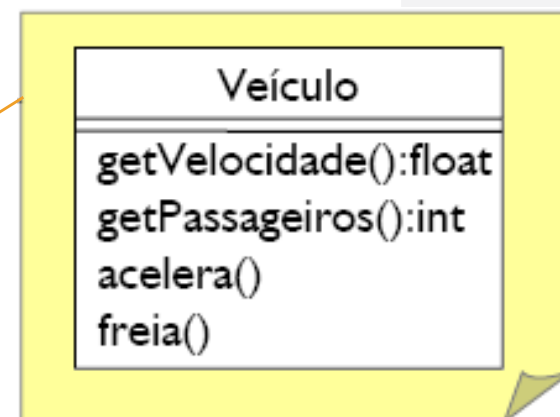
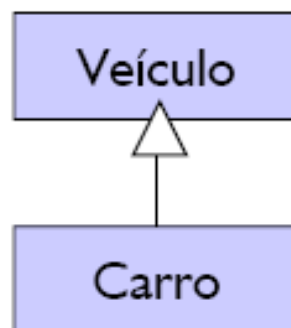
- 2) Qual é a alteração de código necessária para refletir a imagem abaixo:



Reuso - Herança

- Criação de novas classes estendendo classes existentes. O relacionamento “é um [subtipo de]”.

■ *Um carro **é um** veículo*



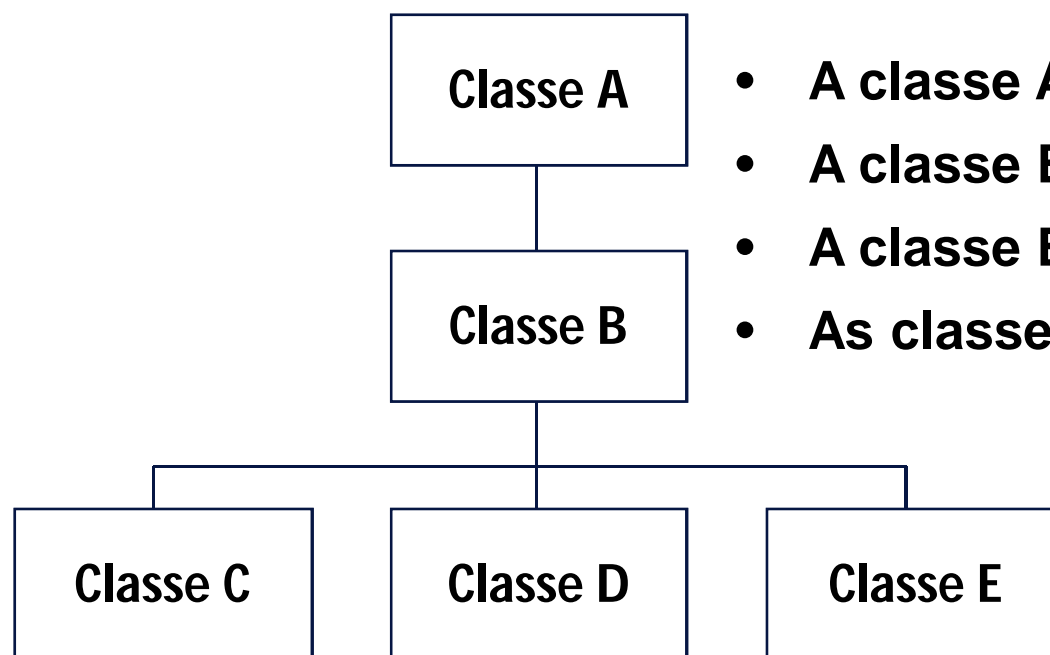
*representação UML
detalhada de 'Veículo'*

```
class Veiculo
{ ... }

class Carro extends Veiculo
{ ... }
```


Reuso - Herança

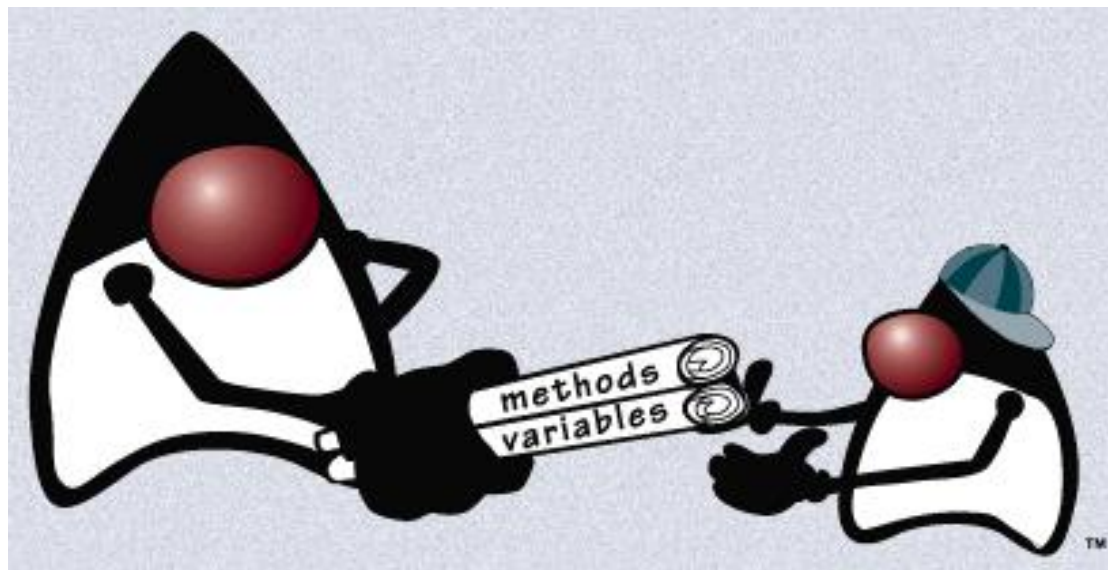
- Herança é um mecanismo que permite a uma **classe herdar todos os métodos e os atributos de outra classe**.
- Uma classe que herda de outra classe é chamada **subclasse** e a classe que fornece a herança é chamada **superclasse**.



- A classe A é a superclasse de B.
- A classe B é uma subclasse de A.
- A classe B é a superclasse de C, D e E.
- As classes C, D e E são subclasses de B.

Reuso - Herança

- As superclasses definem atributos e métodos genéricos que serão herdados pelas classes derivadas.



- Um método herdado de uma superclasse pode ser redefinido pela classe derivada, mantendo o mesmo nome mas agindo de forma diferente.
- Normalmente, os atributos de um objeto só podem ser consultados ou modificados através dos seus métodos (**getters** e **setters**).

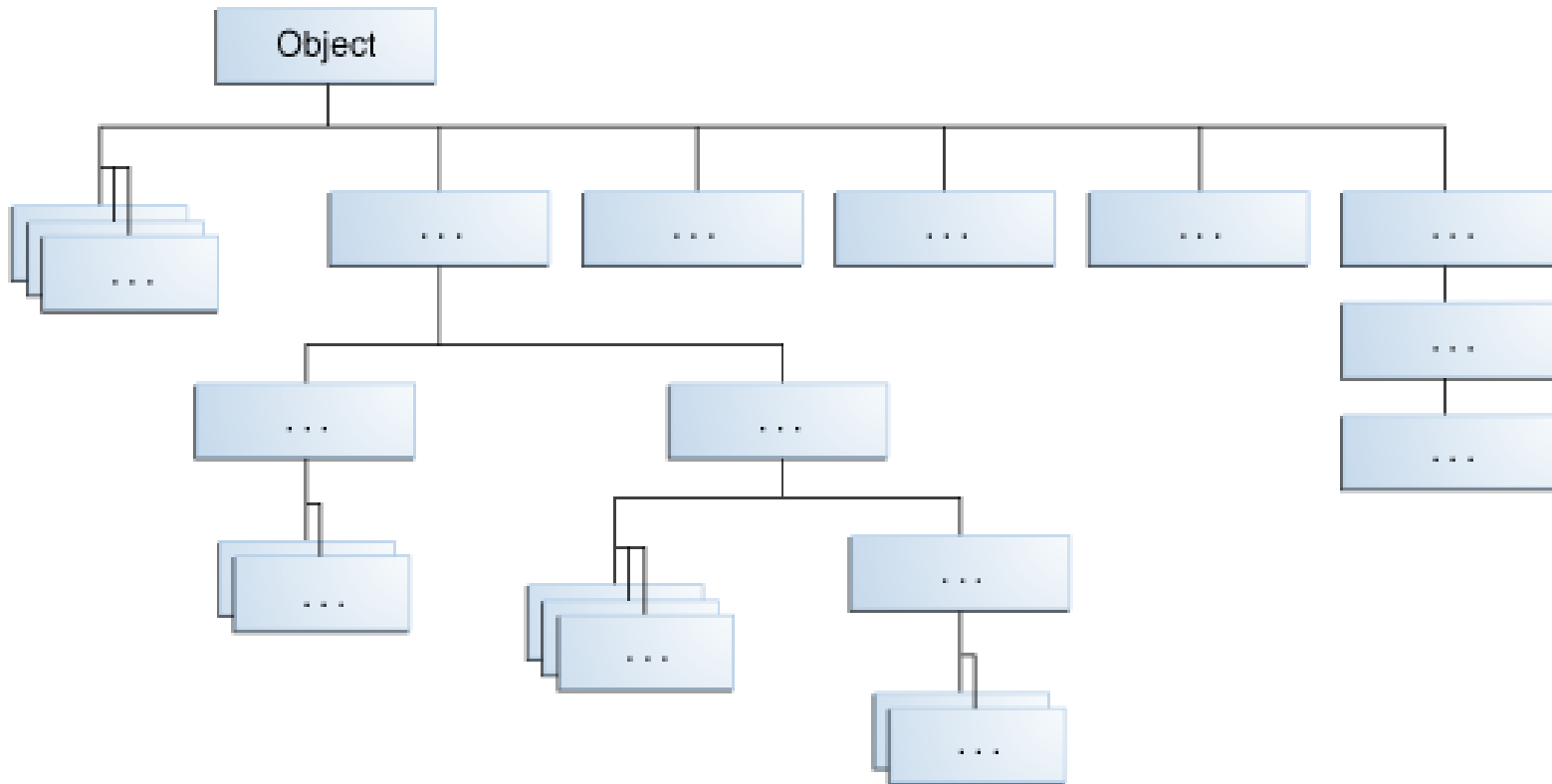
Exercício

- 1) [ESAF - 2008 - CGU] Na linguagem de programação Java, pode-se definir a visibilidade dos métodos e atributos. Com relação a essa característica, é correto afirmar que:
 - a) métodos declarados como **public** em uma superclasse, quando herdados, precisam ser **protected** em todas as subclasses dessa classe.
 - b) métodos declarados como **protected** em uma superclasse, quando herdados, precisam ser **protected** ou **public** nas subclasses dessa classe.
 - c) o nível de acesso **protected** é mais restritivo do que o nível de acesso default.
 - d) métodos declarados como **public** só podem ser acessados a partir dos métodos da própria classe ou de classes derivadas.
 - e) métodos declarados como **default** só podem ser acessados a partir dos métodos da própria classe.

Herança em Java

Herança em Java

- Java adota o modelo de árvore, cuja classe **Object** é a raiz dessa hierarquia de classes.

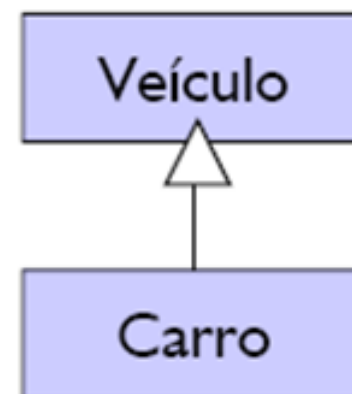


© <http://docs.oracle.com>

Herança em Java

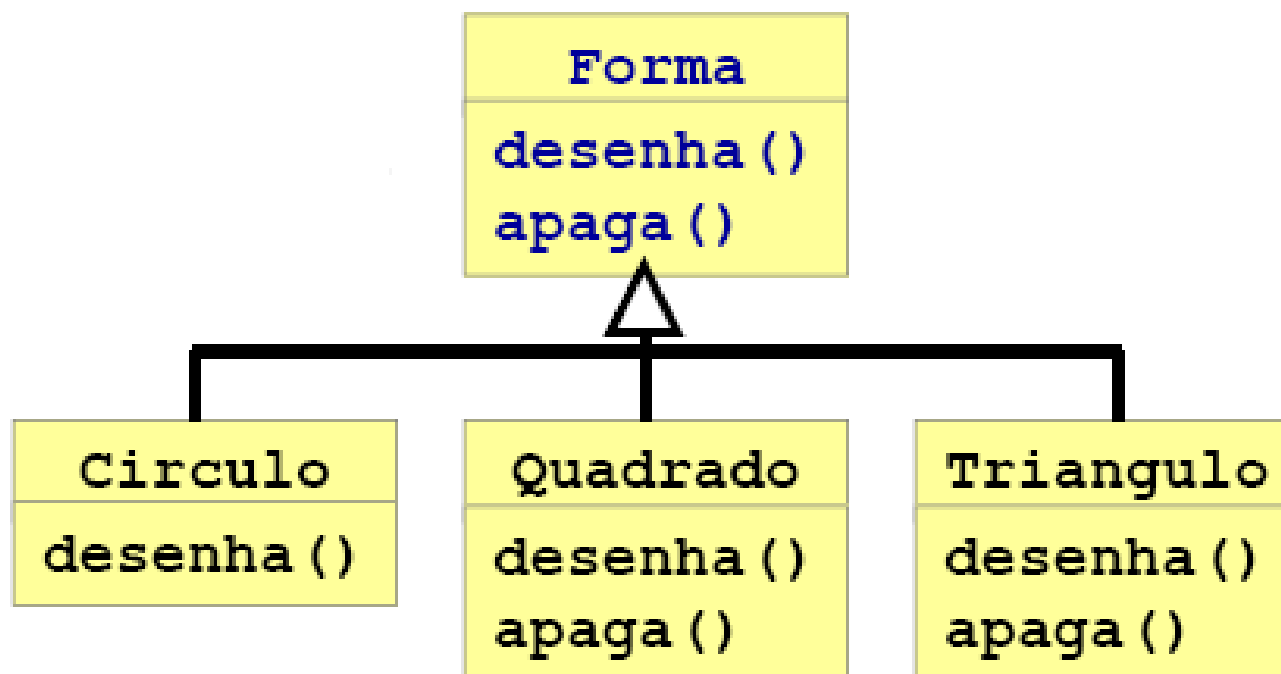
- Uma classe Java só pode estender uma outra classe (**herança simples**) com o uso da palavra reservada **extends**.

```
class Veiculo  
{ ... }  
  
class Carro extends Veiculo  
{ ... }
```



- Quando uma classe não declara nada, esta (implicitamente) estende **Object**.
- Os construtores das subclasses não herdam os construtores das superclasses. Portanto, a chamada a eles deve ser feita pelo uso do comando **super()**.

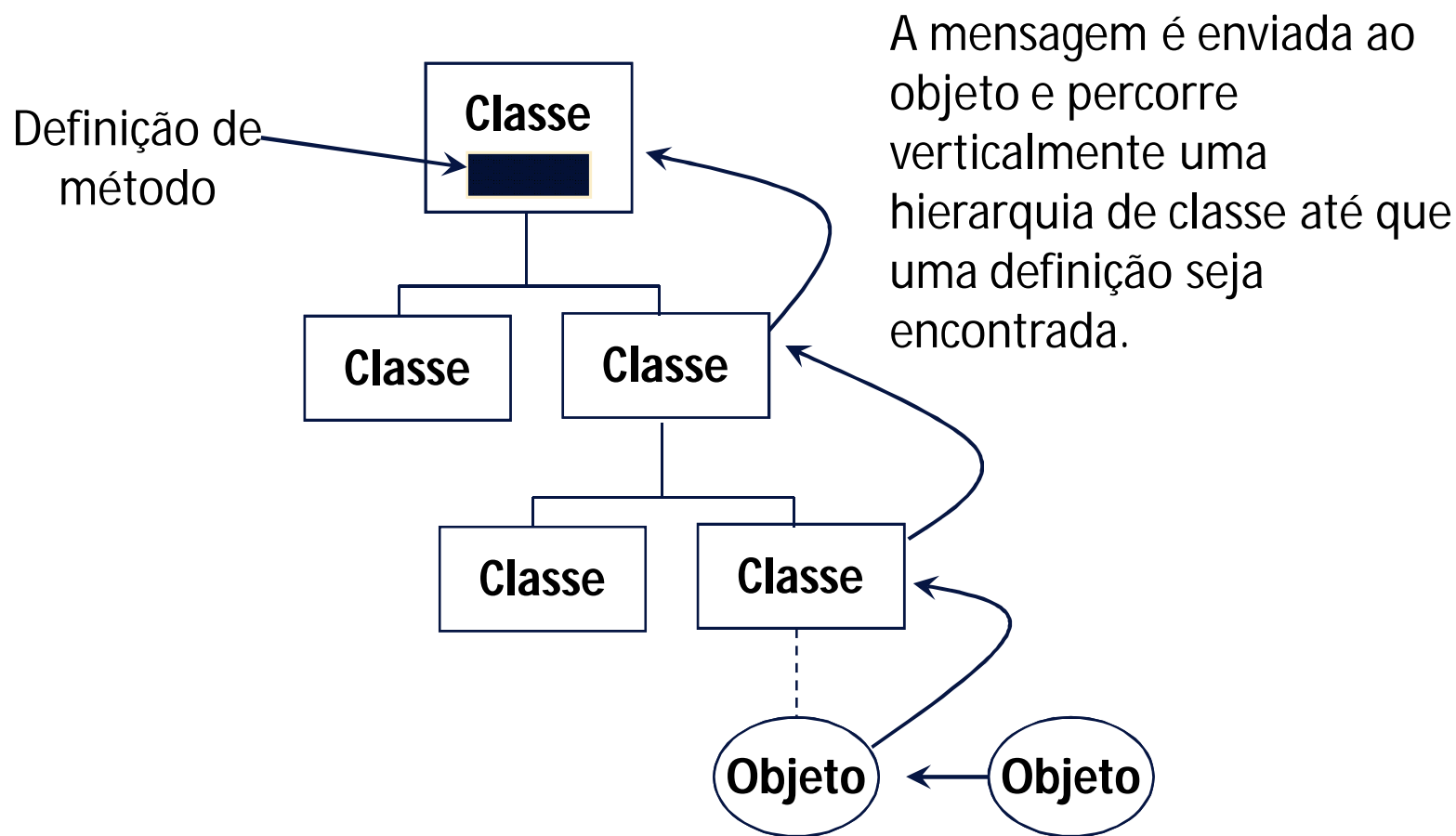
Herança em Java



```
class Forma {...}
class Circulo extends Forma {...}
class Quadrado extends Forma {...}
class Triangulo extends Forma {...}
```

Herança em Java

- Como é realizada a chamada de métodos em uma hierarquia de classes.



Herança em Java

- O uso de herança aumenta o acoplamento entre as classes, isto é, a dependência de uma classe em relação a outra.
- A relação entre classe 'pai' e 'filha' é muito forte e isso acaba fazendo com que o desenvolvedor tenha que conhecer a implementação da superclasse e das suas subclasses.
- O **forte acoplamento da Herança** dificulta a mudança pontual no sistema.
- Mesmo depois de reescrever um método da superclasse, a subclasse ainda pode acessar o método da superclasse.

Sintaxe padrão: **super.método()**.

Herança em Java

DIFERENÇAS **super** e **super()**

- A palavra reservada **super** é utilizada para fazer referência aos membros (atributos e métodos) da superclasse.
- O comando **super()** é utilizado para chamar construtores da superclasse.
- Por definição, o **super()** é chamado (implicitamente) pelo construtor da subclasse. Se houver necessidade de chamá-lo explicitamente, os seus argumentos deverão ser informados.
- Por fim, toda a chamada a **super()** deverá ser feita na primeira instrução do construtor da subclasse.

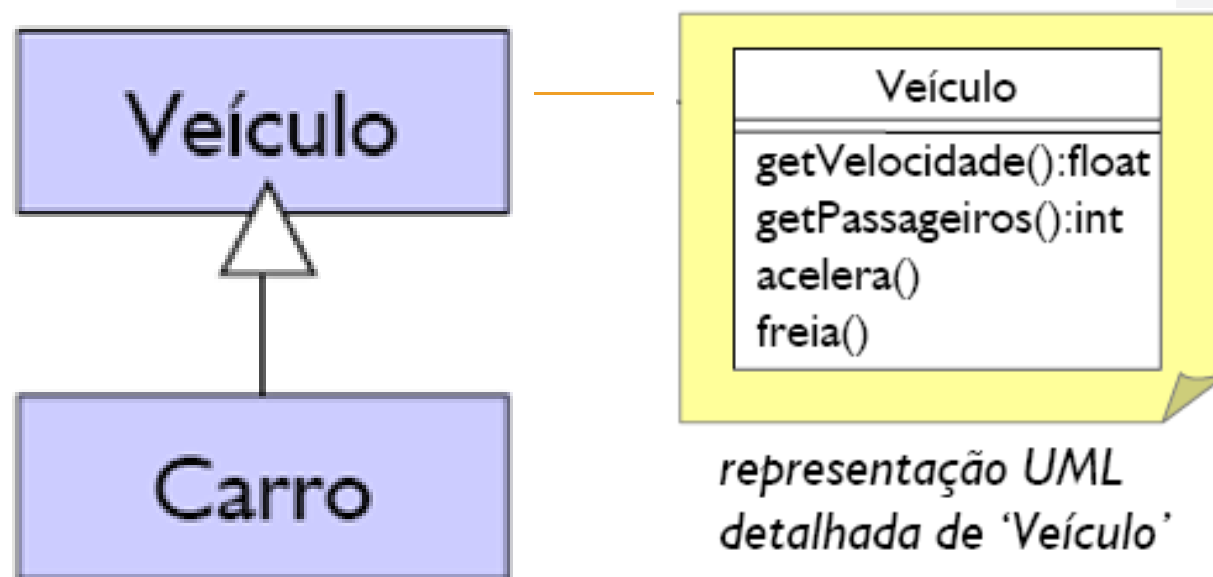
Herança em Java

DIFERENÇAS **super** e **super()**

```
class Circulo extends Forma {  
    public float raio;  
    Circulo(float raio){  
        super(raio);  
        ...  
    }  
    public void apaga() ){  
        super.apaga( );  
        ...  
    }  
}
```

Exercícios

- 1) Modifique a classe **Carro** do **sisalucar** conforme ilustra a imagem abaixo.



- 2) Altere a aplicação **sisalucar** para permitir a locação de outros veículos, tais como: motos e ônibus.
- 3) Crie duas subclasses de Conta: **ContaCorrente** e **ContaPoupanca**.

Sobrecarga e Anulação

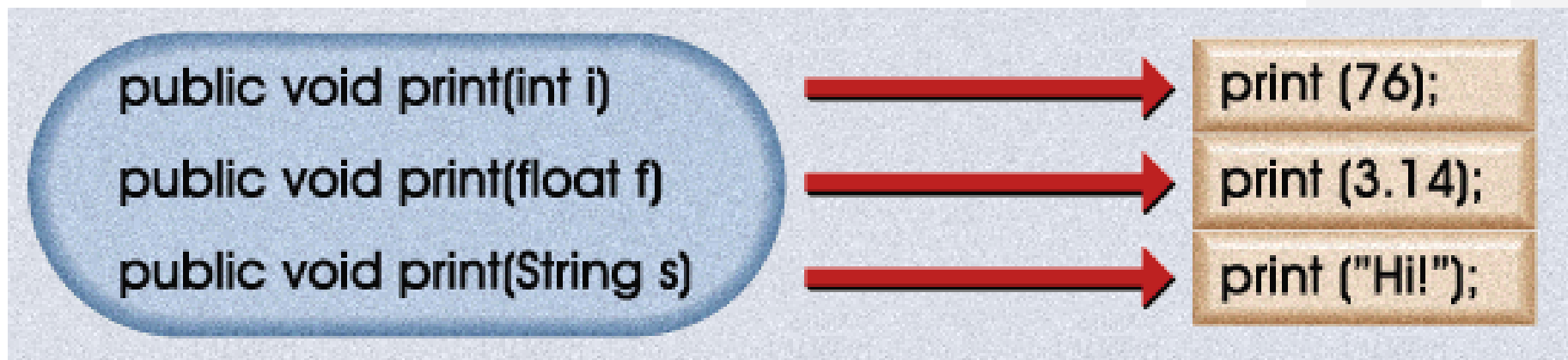
Herança em Java

MANIPULAÇÃO DE MÉTODOS NAS SUBCLASSES

- **Sobrecarga (*overloading*)**
 - Ocorre quando uma subclasse define um método com o mesmo nome do método herdado da superclasse, contudo com a sua assinatura diferente.
- **Anulação ou Sobreposição (*overriding*)**
 - Ocorre quando uma subclasse define um método com o mesmo nome e a mesma assinatura do método herdado da superclasse.
 - Métodos constantes (**final**) não podem ser sobrepostos.
- **Extensão**
 - Ocorre quando uma subclasse define novos métodos, sem qualquer relação com os métodos herdados da superclasse.

Sobrecarga (*overloading*)

EXEMPLO DE SOBRECARGA



- Uma boa prática é usar a sobrecarga, somente, nos métodos que possuam a mesma funcionalidade.
- A sobrecarga pode ser feita igualmente nos métodos construtores.

Sobrecarga (*overloading*)

EXEMPLO

```
void mover (int dx, int dy) {  
    x += dx;  
    y += dy;  
}  
void mover (int raio, float ang) {  
    x += raio*Math.cos(ang);  
    y += raio*Math.sen(ang);  
}
```

SUPERCLASSE

SUBCLASSE

```
void doIt(int ...v){  
    //instruções  
}  
void doIt(boolean ...v){  
    //instruções  
}
```

SUPERCLASSE

SUBCLASSE

Sobrecarga (*overloading*)

EXEMPLO - CONSTRUTOR

```
class Ponto {  
    int x, y;  
    Ponto () { }  
  
    Ponto (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

- A sobrecarga de construtores visa definir diferentes formas de criar um objeto.

- Exemplo:

```
Ponto p1 = new Ponto(); //p1 está em (0,0)  
Ponto p2 = new Ponto(1,2); //p2 está em (1,2)
```

Sobrecarga (*overloading*)

ENCADEAMENTO DE CONSTRUTOR

- O “encadeamento de construtor” ocorre quando um construtor faz referência a outro. Isto é feito pelo uso do comando **this()**.

```
Ponto () {  
    this(0,0);  
}  
Ponto (int x, int y) {  
    this.x = x;  
    this.y = y;  
}
```

- Essa chamada deverá ser feita na primeira linha do construtor!

Sobrecarga (*overloading*)

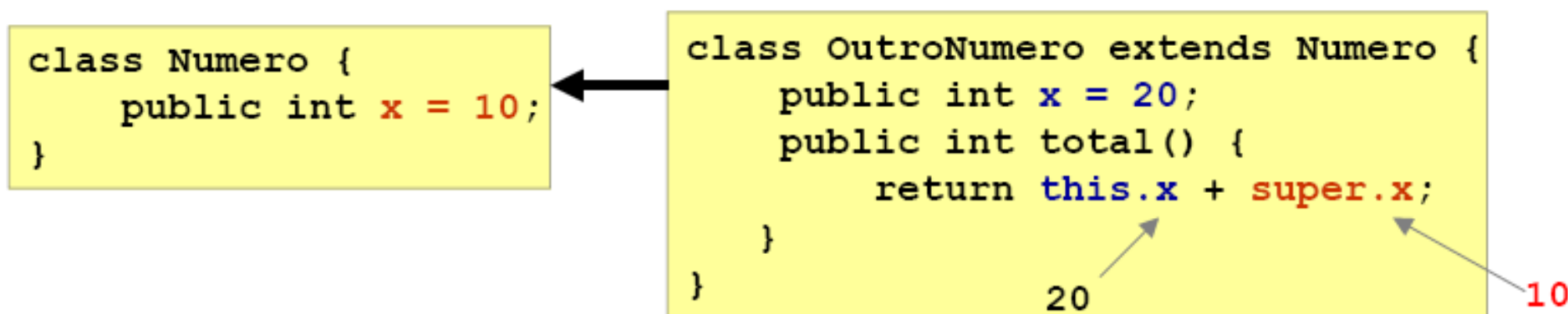
DIFERENÇAS **this** e **this()**

- A palavra reservada **this** é utilizada para informar que a variável manipulada é um atributo da classe, e não uma simples variável.
- No exemplo anterior, os atributos x e y de Ponto recebem os valores passados pelas variáveis x e y.
- O comando **this()** é utilizado para chamar construtores.

Sobrecarga (*overloading*)

DIFERENÇAS **this** e **super**

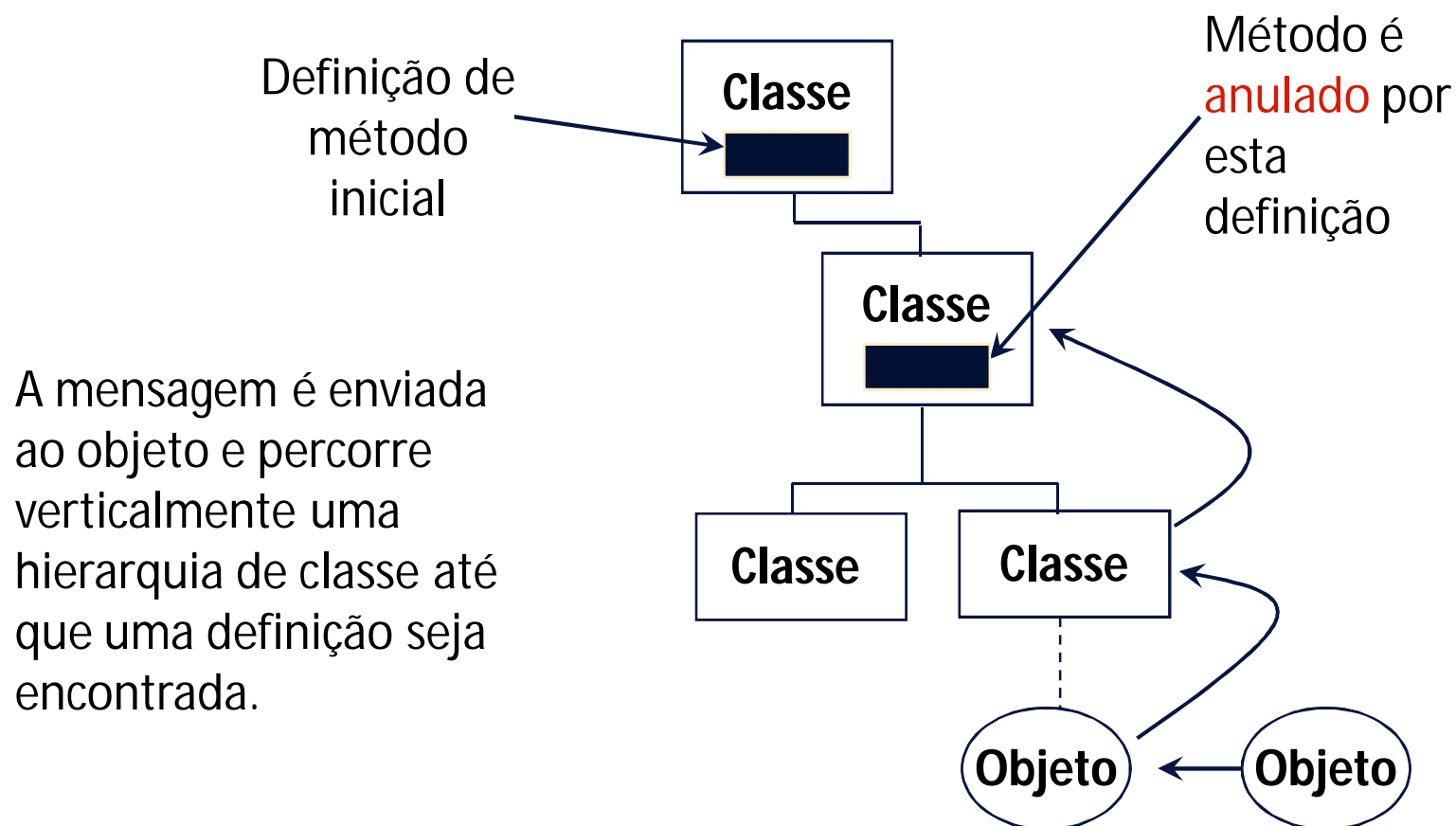
- Usados quando for necessário referenciar explicitamente a instância (**this**) ou a superclasse (**super**).



- Não confundir **this** e **super** com **this()** e **super()**. Estes últimos são usados apenas em chamadas de construtores!

Anulação (*overriding*)

- Funcionamento quando uma subclasse define um método que possui a mesma assinatura do método herdado da superclasse.



Anulação (*overriding*)

EXEMPLO

```
void mover (int dx, int dy) {  
    x += dx;  
    y += dy;  
}  
  
void mover (int dx, int dy) {  
    x += dx - dy;  
    y += dy - dx;  
}
```

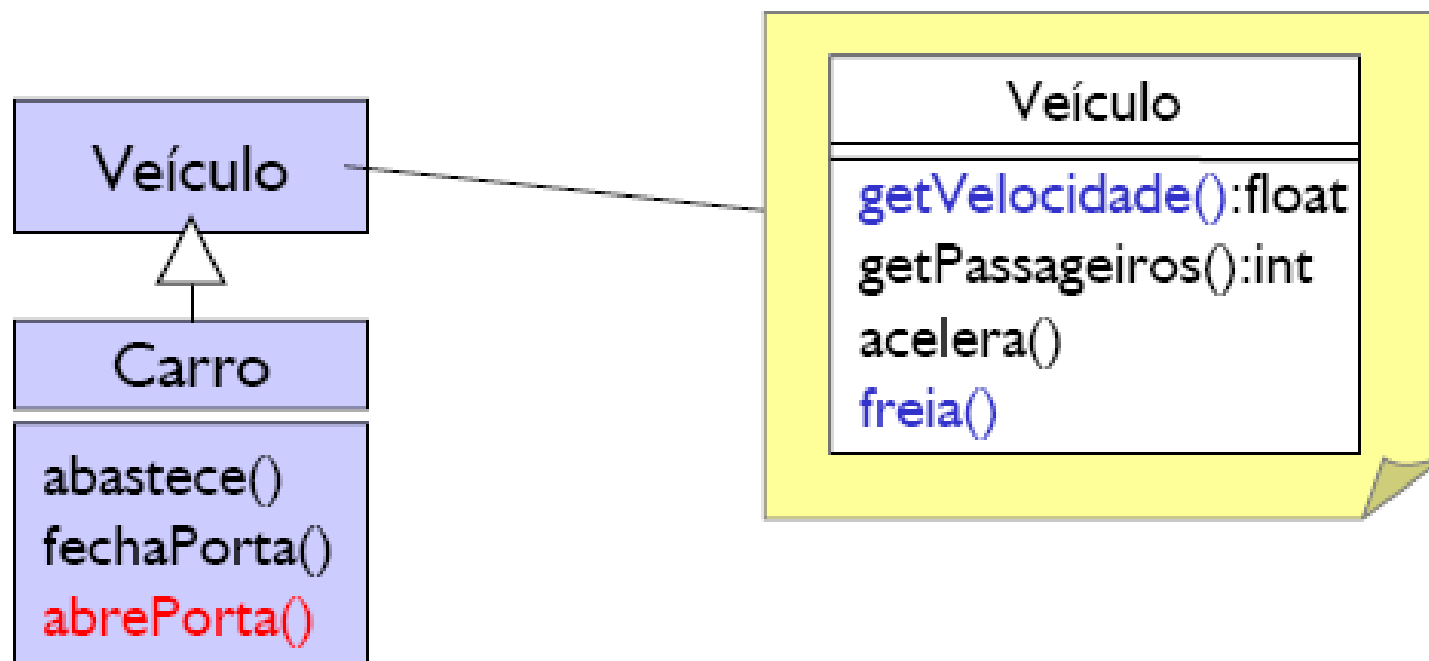


SUPERCLASSE

SUBCLASSE

Extensão

- Novos métodos são criados na subclasse.



Métodos Constantes

- São os métodos que não podem ser anulados pelas subclasses.
- Deve ser utilizado a palavra reservada **final**.

```
final void mover (int dx, int dy) {  
    x += dx;  
    y += dy;  
}  
void mover (int dx, int dy) {  
    x += dx;  
    y += dy;  
}
```



SUPERCLASSE

SUBCLASSE

Classes Constantes

- São as classes que não podem ser estendidas (serem superclasses).

```
public final class A {  
    ...  
}
```

- Todos os métodos dessa classe são finais. Muito útil em classes que contém funções utilitárias e constantes.

```
java.lang  
Class Math  
  
java.lang.Object  
    java.lang.Math  
-----  
public final class Math  
    extends Object
```

Exercícios

- 1) Altere os métodos herdados pelas subclasses **Carro**, **Moto** e **Onibus** da aplicação **sisalucar**, fazendo uso das técnicas de **sobrecarga** e **anulação**.
- 2) Crie novos métodos (extensão) na subclasse Carro.



- 3) Adicione um método na classe **Conta**, que atualiza essa conta de acordo com uma taxa percentual fornecida. Posteriormente, realize a anulação desse método nas subclasses **ContaCorrente** e **ContaPoupanca**: a primeira deve atualizar-se com o dobro da taxa e a segunda com o triplo da taxa.

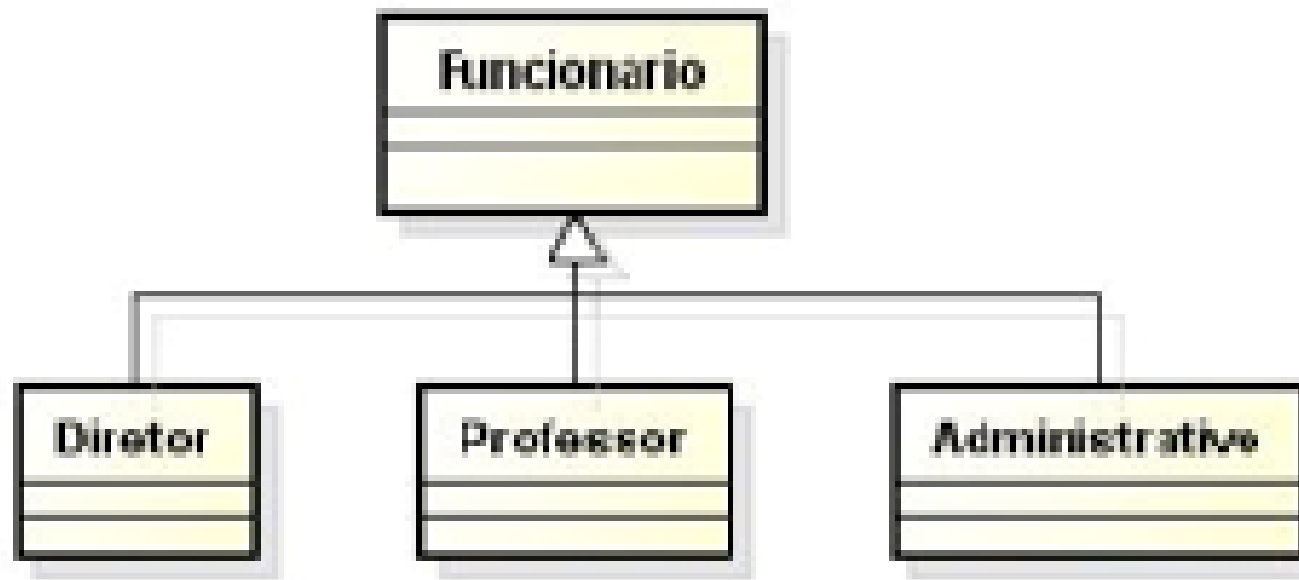
Polimorfismo

Polimorfismo

- Polimorfismo é conhecido como o terceiro pilar da programação orientada a objeto, após o encapsulamento e a herança.
- Polimorfismo é uma palavra grega que significa “muitas formas” e **é a capacidade de um objeto poder ser referenciado de várias formas.**
- Por meio do Polimorfismo é possível:
 - **1. Trazer clareza ao código;**
 - **2. Diminuir as linhas de programação;**
 - **3. Desenvolver aplicações flexíveis.**

Polimorfismo

- Considere a hierarquia de classes abaixo:



© <http://tdsb2014.blogspot.com>

- No exemplo acima, é possível perceber que um objeto do tipo **Funcionário** também pode ser um objeto do tipo **Diretor**, **Professor** ou **Administrativo**.
- Poder adquirir formas mais especializadas de **Funcionário** em tempo de execução é uma das principais vantagens do polimorfismo.

Polimorfismo

- **EXEMPLO**

```
public class Funcionario{}  
public class Professor extends Funcionario{}  
public class Diretor extends Funcionario{}  
//Código Principal  
    Funcionario jose = new Professor();  
//Promoção  
    jose = new Diretor();
```

- No exemplo acima, o objeto **jose** (do tipo **Funcionario**) pode assumir formas “mais especializadas”, como **Professor** e, posteriormente, **Diretor**.

Polimorfismo

- **EXEMPLO**

```
public class Funcionario{}  
public class Diretor extends Funcionario{  
    String departamento;  
}  
//Código  
Funcionario jose = new Diretor();  
jose.departamento = "Financeiro"; //Erro
```

- No exemplo acima, o atributo 'departamento' é visível apenas na classe **Diretor**, sendo que o objeto **jose** é do tipo **Funcionario**.
- A correção se dá com **Casting de Objetos** que será estudado no final desta Unidade.

Polimorfismo

- Polimorfismo é o nome formal para o fato de que, quando se precisa de um objeto de determinado tipo, pode-se utilizar uma versão mais especializada dele.
- Ao se estender ou especializar uma classe, não se perde a compatibilidade com a superclasse.

Polimorfismo

OPERADOR **instanceof**

- É utilizado para identificar se um determinado objeto pertence a uma hierarquia de classes.
- **Sintaxe Padrão:**

```
if (objeto instanceof Funcionario){  
    ...  
}
```

- O resultado da condição será **true** se objeto for do tipo **Funcionario** ou das suas subclasses (**Diretor**, **Professor** e **Administrador**). Caso contrário, o resultado será **false**.

Polimorfismo

OPERADOR **instanceof**

```
void verificarTipoFuncionario(Funcionario objeto)
{
    if (objeto instanceof Diretor)
        System.out.println("Objeto Diretor");
    else if (objeto instanceof Professor)
        System.out.println("Objeto Professor");
    else
        System.out.println("Objeto Administrador");
}
```

Exercício

- 1) Criar a hierarquia de classes de **Funcionário** e implementar o método **verificarTipoFuncionario(...)**.
- 2) [FCC - 2010 – MPE] Uma operação pode ter implementações diferentes em diversos pontos da hierarquia de classes, desde que mantenham a mesma assinatura. Na orientação a objetos, este é o conceito que embasa:
 - a) a multiplicidade
 - b) o encapsulamento
 - c) o protótipo
 - d) o polimorfismo
 - e) o estereótipo.

Casting, Conversão de Objetos e Tipos Primitivos

Casting, Conversão de Objetos e Tipos Primitivos

- Java é uma linguagem **fortemente tipada** pois exige a declaração de um tipo.
- Quando se definem argumentos em métodos ou variáveis em expressões, a utilização dos mesmos com os tipos de dados corretos é obrigatório!
- **POR EXEMPLO:**

```
void realizarConversao(int valor)
{ ... }
//Execução
Objeto.realizarConversao("antonio");
//Erro de compilação => tipo requerido é 'int'
```

Casting, Conversão de Objetos e Tipos Primitivos

- Muitas vezes, haverá uma definição na classe, no método, no atributo ou na variável cujo tipo não é o ideal em determinada situação.
- Pode ser a classe errada ou o tipos de dados errados, como por exemplo um *float* quando se precisa de um *int*.
- Nesta situação, utiliza-se o *Casting* para converter o valor de um tipo para outro.
- O uso de *Casting* no Java é dividido em três partes:
 - ***Casting* entre tipos primitivos;**
 - ***Casting* de Objetos;**
 - ***Casting* de tipos primitivos para objetos e vice-versa.**

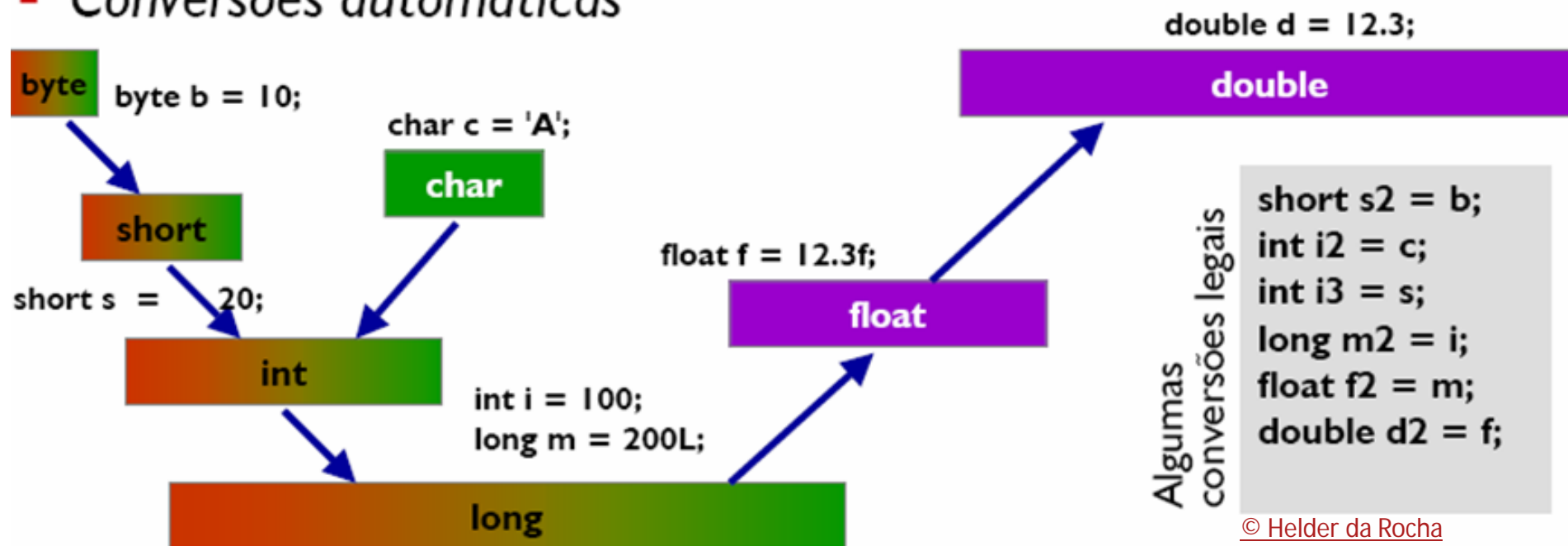
Casting de Tipos Primitivos

- O **Casting** entre tipos primitivos permite converter o valor de um tipo para outro.

```
int a; byte b;  
b = (byte) a; //Downcasting
```

- O Java converterá automaticamente um tipo de dados em outro (**Upcasting**) sempre que houver garantia de não haver perda de informação.

■ Conversões automáticas



© Helder da Rocha

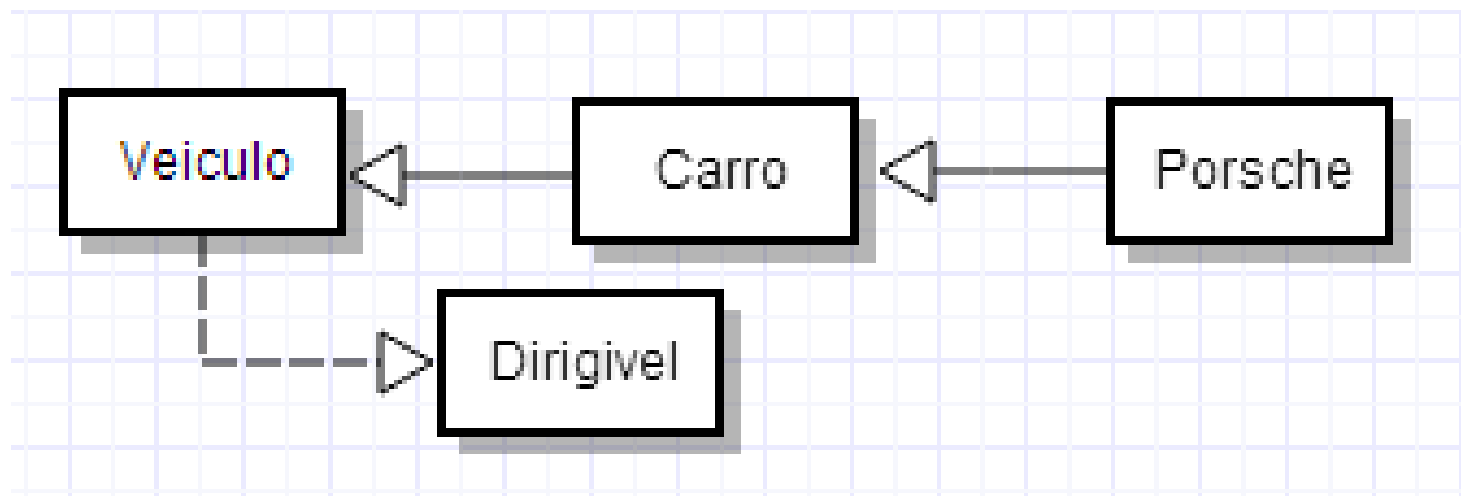
Casting de Objetos

- O **Casting** de Objetos ocorre quando se deseja associar um objeto da **classe Filho** a uma declaração da **classe Pai**.

```
Pai objeto = new Filho();
```

- Esta conversão é automática desde que a **classe Pai** seja uma superclasse (direta ou indireta) de **Filho** ou seja uma interface implementada por **Pai**.

- Exemplo:**



Casting de Objetos

```
class Carro extends Veiculo {...}
```

```
class Veiculo implements Dirigivel {}
```

```
class Porsche extends Carro {...}
```

Algumas conversões legais

```
Carro c = new Carro();  
Veiculo v = new Carro();  
Object o = new Carro();  
Dirigivel d = new Carro();  
Carro p = new Porsche();
```

Casting de Objetos

- Pode-se utilizar uma versão mais especializada quando se precisa de um objeto de certo tipo. Neste caso, a conversão é automática.

```
Veiculo v = new Carro();
```

- No sentido inverso, quando for necessário fazer a conversão de volta ao tipo mais especializado, a conversão deverá ser explícita!

```
Carro c = (Carro)v;
```

//Código do slide 55

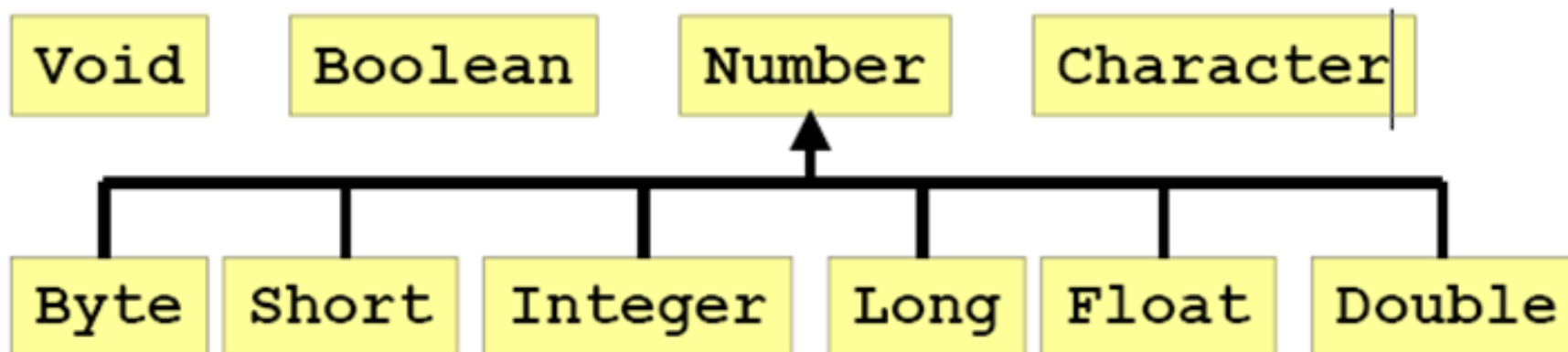
```
Funcionario jose = new Diretor();
```

```
jose.departamento = "Financeiro"; //Erro
```

```
((Diretor)jose).departamento = "Financeiro"; //OK
```

Casting de Tipos Primitivos para Objetos e Vice-versa

- No Java os tipos primitivos e os objetos são muito diferentes entre si. Portanto, para que haja uma conversão adequada entre eles, devem-se utilizar as classes Java conhecidas como *Wrappers*.



- Estas classes são utilizadas para encapsular tipos primitivos em objetos Java.

Casting de Tipos Primitivos para Objetos e Vice-versa

AUTOBOXING e AUTOUNBOXING

- O **Autoboxing** é o processo que transforma automaticamente um tipo primitivo em seu Objeto equivalente.
- O **Autounboxing** é o processo inverso que transforma automaticamente um Objeto no seu tipo primitivo.

```
//ANTES
```

```
Integer iob = new Integer(100);
```

```
//COM AUTOBOXING
```

```
Integer iob = 100;
```

```
//ANTES
```

```
int i = iob.intValue();
```

```
//COM AUTOUNBOXING
```

```
int i = iob;
```

- A principal vantagem do recurso de **AUTOBOXING** e **UNBOXING** é simplificar o código e prevenir erros de conversão.

Casting de Tipos Primitivos para Objetos e Vice-versa

OBSERVAÇÃO: Não se deve abandonar o uso de tipos primitivos para utilizar apenas objetos! É pouco eficiente!

Exercício

- 1) Qual é o Casting de tipos necessários para permitir a atribuição dos valores abaixo?

```
char a = 'a'; int b = 'b'; float c = 100;  
int d = 5.1987; float e = 5.0;  
int f = (a + 5); char g = 110.5;
```

- 2) Qual são as opções corretas?

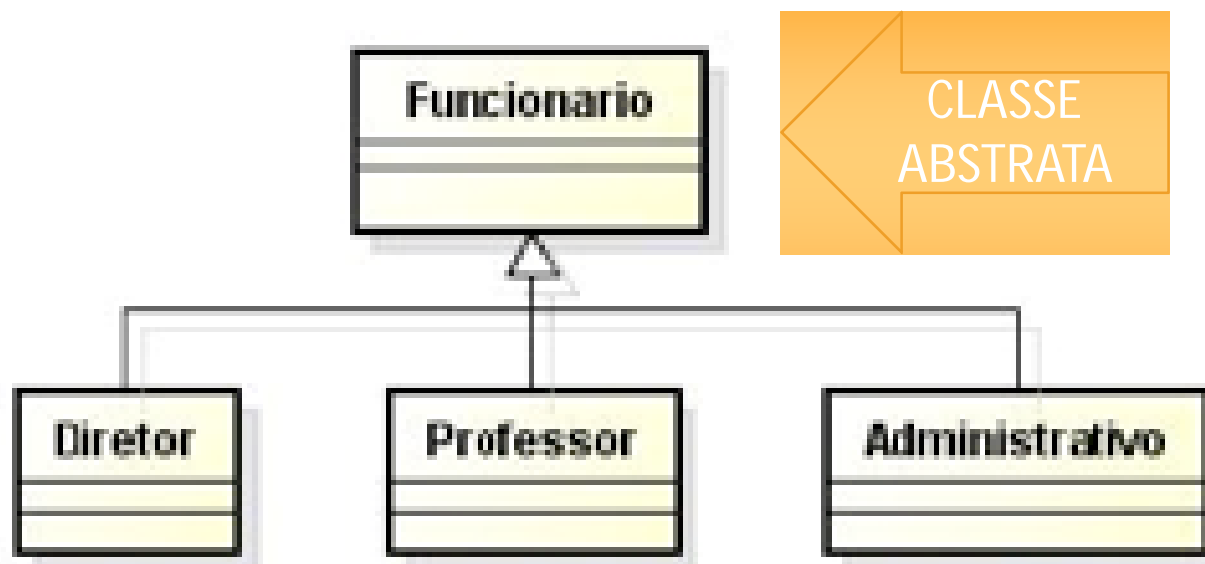
```
interface Pet{  
class Dog implements Pet{  
class Beagle extends Dog{
```

- a) Pet a = new Dog();
- b) Pet b = new Pet();
- c) Dog f = new Pet();
- d) Dog d = new Beagle();
- e) Pet e = new Beagle();
- f) Beagle c = new Dog();

Classes Abstratas

Classes Abstratas

- Ao se criar uma classe para ser estendida, é muito comum não se ter idéia de como codificar os seus métodos, isto é, somente as suas subclasses saberão implementá-los.
- Uma classe deste tipo não pode ser instanciada pois sua funcionalidade está incompleta. Tal classe é dita **abstrata**.



Classes Abstratas

- No exemplo em questão, não faz sentido criar objetos do tipo **Funcionário**, mas sim objetos do tipo **Diretor**, **Professor** ou **Administrador**.
- Nesta caso, especificando **Funcionario** como classe abstrata, economiza-se código e ganha-se o poliformismo para a criação de métodos genéricos que servirão a diversos objetos.
- **Java suporta o conceito de classes abstratas**. Pode-se declarar uma classe abstrata usando o modificador **abstract**.
- Métodos também podem ser declarados abstratos com o modificador **abstract**. As suas implementações serão feitas nas subclasses.
- As classes abstratas podem ter métodos concretos, campos de dados e construtores. Os objetos das suas subclasses poderão fazer uso deles.

Classes Abstratas

```
abstract class Funcionario {  
    public abstract double getbonificacao();  
}  
public class Professor extends Funcionario{  
    public double getBonificacao()  
    {  
        return this.salario * 1,4;  
    }  
}
```

Exercício

- 1) Altere a superclasse **Funcionário** para classe abstrata. Defina o método **getBonificacao(...)** como abstrato. Realize a sua implementação nas subclasses de Funcionário.
- 2) [FUNRIO – 2009 - FURNAS] Na orientação a objetos, classes que NÃO geram instâncias diretas (objetos) são denominadas classes:
 - a) abstratas
 - b) primárias
 - c) virtuais
 - d) básicas
 - e) derivadoras.

Interfaces

Interfaces

- Todos os métodos públicos (**public**) de uma classe podem ser acessados por objetos de outras classes. Este conjunto de métodos define a interface de acesso a uma classe.
- No Java existe o conceito de interface que define uma série de métodos, sem conter as suas implementações. A interface só expõe o que o objeto deve fazer, e não como ele faz, nem o que ele tem. Como ele faz vai ser definido em uma implementação dessa interface.

```
interface Autenticavel {  
    boolean autentica(int senha);  
}
```

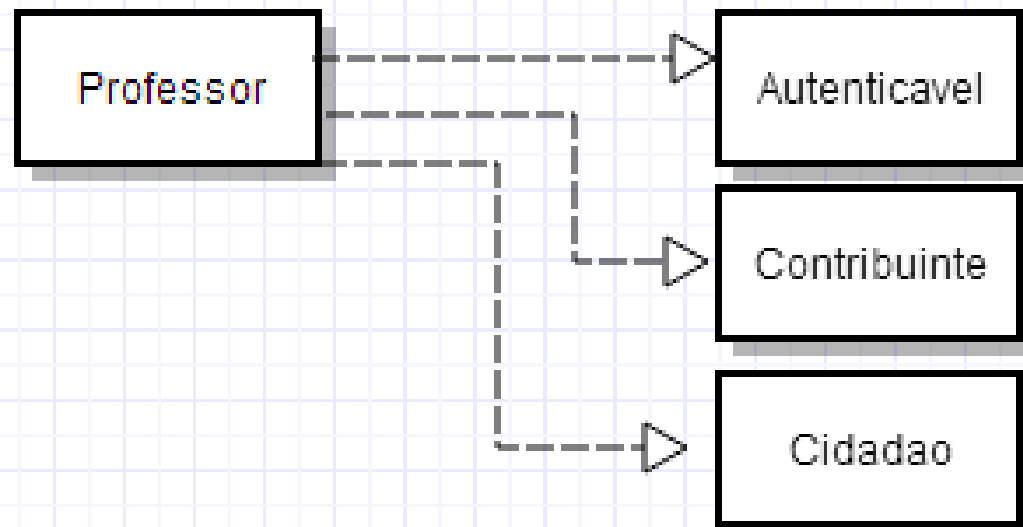
Interfaces

- Para implementar uma interface em uma classe utiliza-se a palavra reservada **implements**.

```
interface Autenticavel {  
    boolean autentica(int senha);  
}  
  
public class Professor extends Funcionario  
implements Autenticavel {  
    boolean autentica(int senha){  
  
        ...  
    }  
}
```

Interfaces

- Em Java, uma classe pode estender uma outra classe e implementar zero ou mais interfaces.



```
public class Professor extends Funcionario
implements Autenticavel, Contribuinte, Cidadao
{ ... }
//Código
Autenticavel a = new Professor();
```

Interfaces

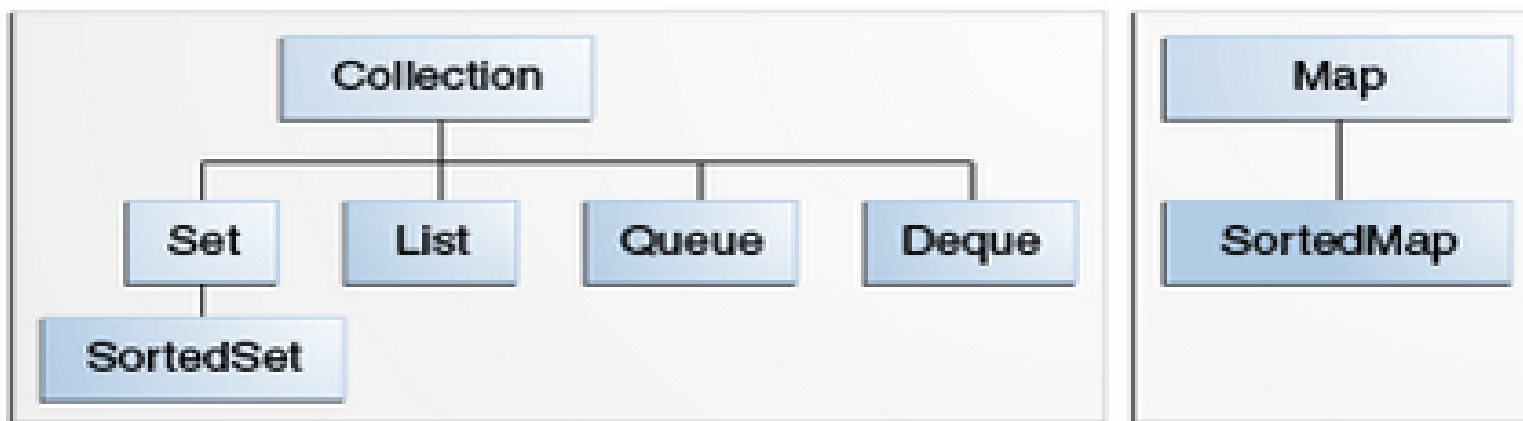
- O Java não permite herança múltipla com herança de código, porém torna possível herdar múltiplas interfaces.
- Uma vez que uma interface não possui implementação, deve-se notar que:
 - seus campos devem ser públicos, estáticos e constantes;
 - seus métodos devem ser públicos e abstratos.
- Como esses qualificadores são fixos, não é necessário a sua declaração.

OBSERVAÇÃO: No Java 8 já é possível definir implementações de métodos em interfaces. Recurso este conhecido como métodos default.

Interfaces

HIERARQUIA DE INTERFACES NO JAVA API

- O framework Collections é todo baseado em interfaces.



- Interfaces Java servem para fornecer **polimorfismo sem herança!**
- Deve ser utilizado interfaces sempre que possível, pois o código fica mais flexível.

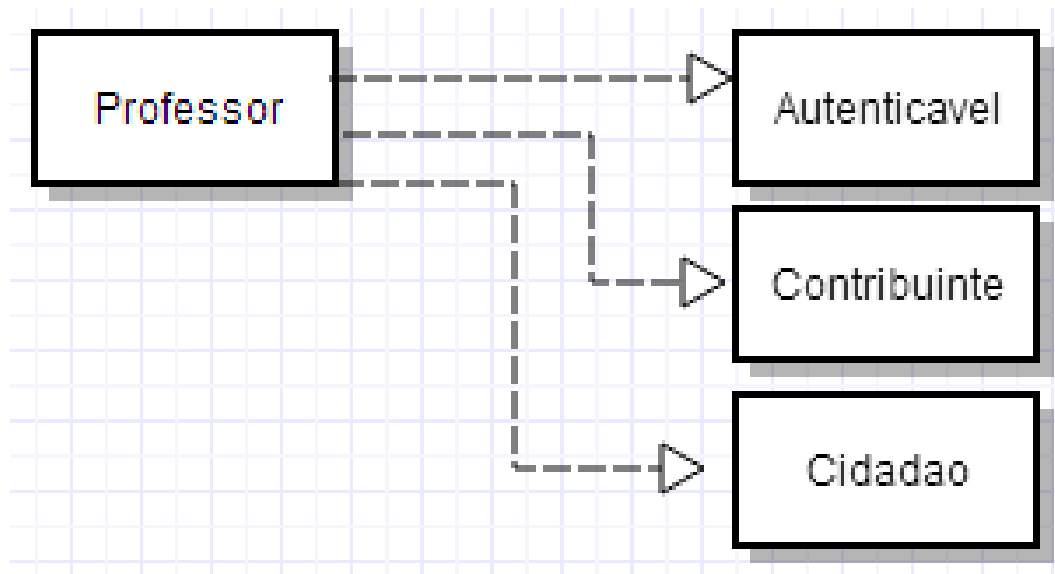
Interfaces

CONCLUSÃO

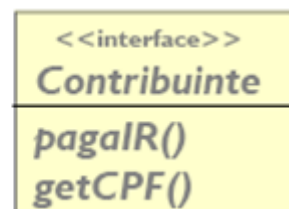
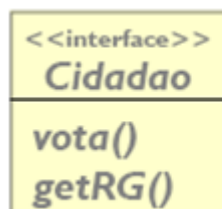
- É mais fácil evoluir classes concretas que interfaces.
- Não é fácil acrescentar métodos a uma interface depois que ela já estiver em uso, visto a necessidade de alteração de todas as classes que fazem uso da mesma.
- Quando a evolução for mais importante que a flexibilidade oferecida pelas **interfaces**, deve-se utilizar **classes abstratas**.

Exercícios

- 1) Implemente a estrutura abaixo:



obs: o modelo das interfaces **Contribuinte** e **Cidadao** está descrito abaixo.



Exercícios

- 2) O que há de errado na Interface abaixo?

```
public interface SomethingIsWrong {  
    void aMethod(int aValue){  
        System.out.println("Hi Mom");  
    }  
}
```

- 3) A Interface abaixo é válida?

```
public interface Marker {  
}
```

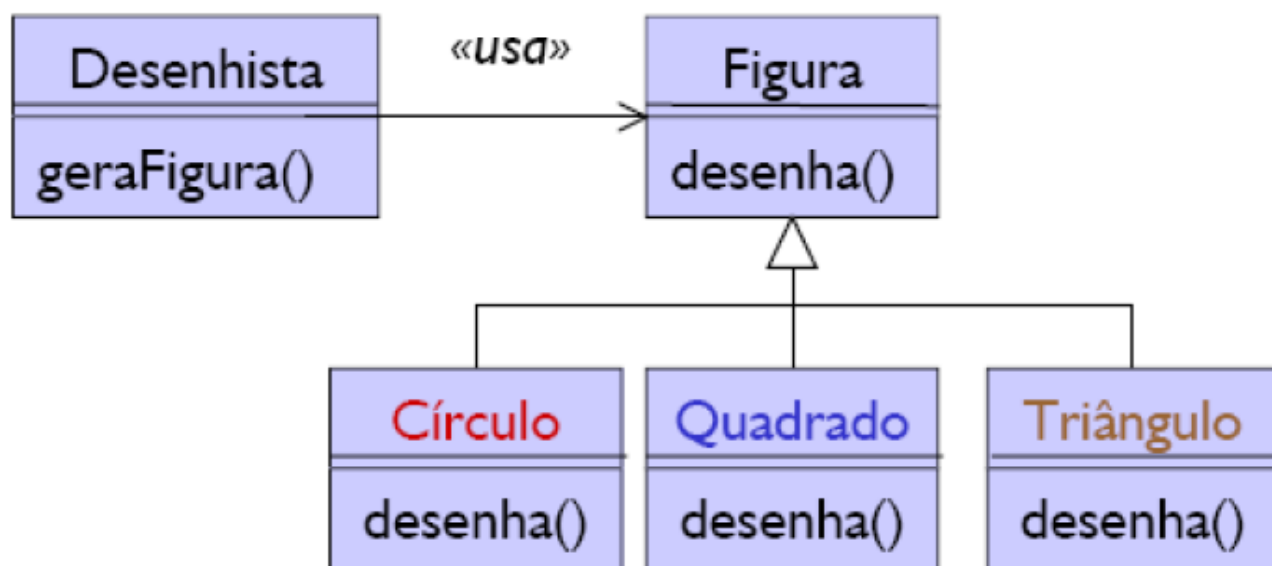
RESUMO

TÓPICOS APRESENTADOS

- Nesta aula nós estudamos:
 - **Encapsulamento e os Modificadores de Acesso**
 - **Reuso**
 - **Herança em Java**
 - **Sobrecarga e Anulação**
 - **Polimorfismo**
 - **Casting, Conversão de Objetos e Tipos Primitivos**
 - **Classes Abstratas**
 - **Interfaces**

ATIVIDADES PARA SE APROFUNDAR

- 1) Pesquisar qual a importância do uso das classes abstratas e interfaces no *framework collections* (java.util).
- 2) Implementar a hierarquia de classes abaixo:



- 3) No exemplo acima, implemente a classe **Figura** como abstrata. Posteriormente, analise a hipótese de implementá-la como Interface?