4 – Conceito de Herança

Hierarquia de classes e mecanismo de ligação

Herança – Uma classe pode herdar operações de uma superclasse e as suas operações podem ser herdadas por subclasses.

O mecanismo de herança permite definir uma nova classe em termos de uma classe existente, com modificações e/ou extensões de comportamento.

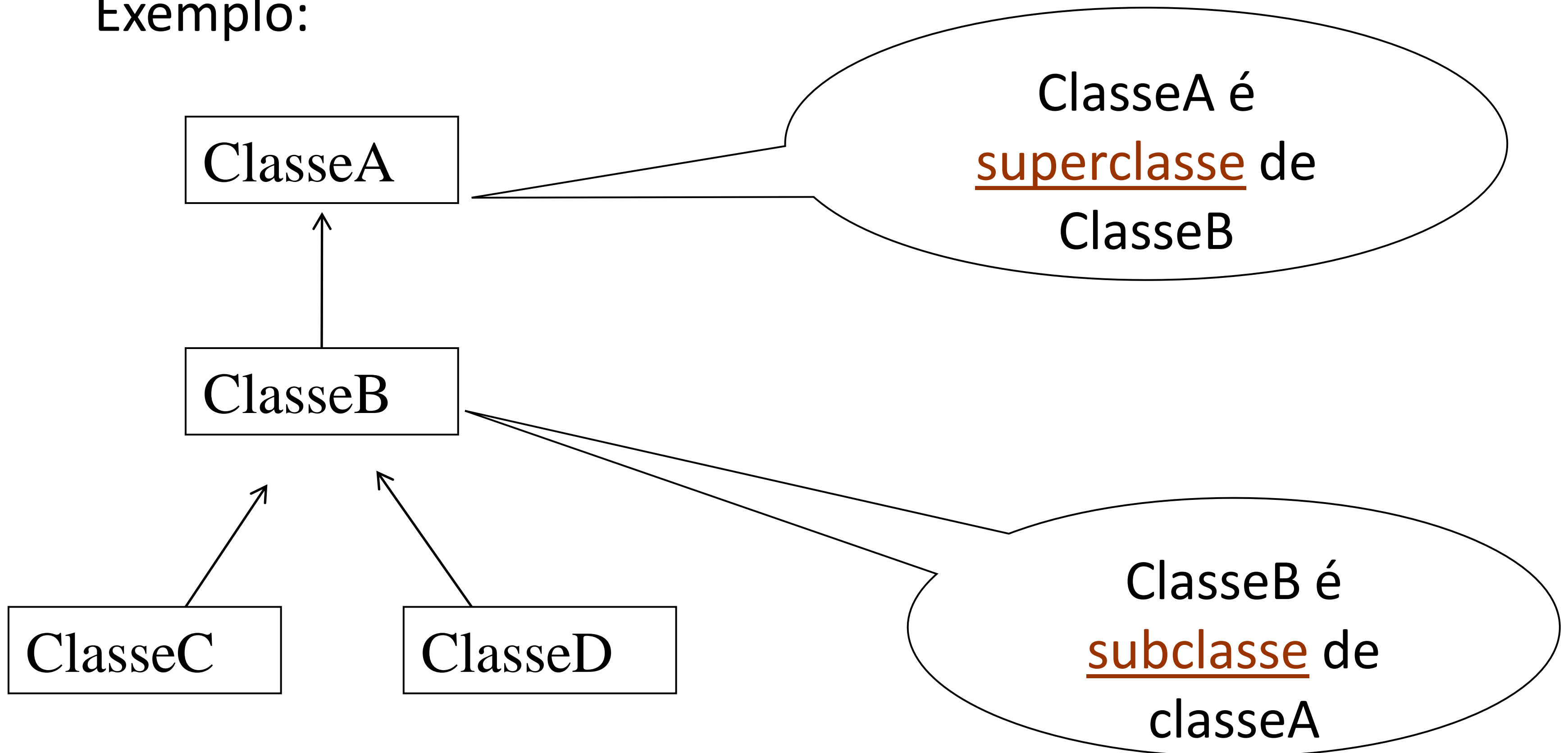
A nova classe é a subclasse da anterior ou classe derivada.

A classe inicial é a superclasse ou classe base.

- Pode repetir-se o processo, definindo uma nova classe a partir da classe derivada anterior ...

Construindo uma hierarquia de classes ➔

Exemplo:



Todos os métodos e atributos da superclasse vão ser herdados pela subclasse.

À subclasse, podem ser adicionados novos métodos e novos atributos num processo de especialização sucessiva.

Dada uma hierarquia de classes,

. uma instância de uma subclasse vai conter ➔

- as variáveis de instância da superclasse (ou superclasses)

mais

- as variáveis de instância declaradas na classe derivada (subclasse).

. O comportamento dessa instância está definido

na sua classe

e

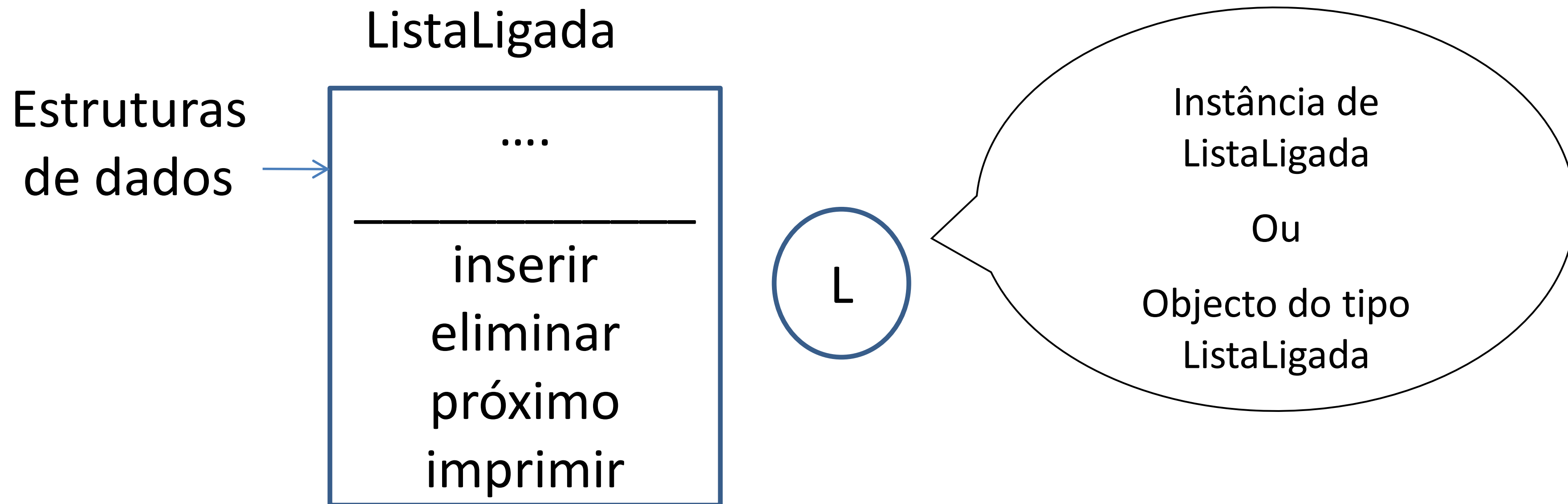
no conjunto das suas superclasses.

Quando um método é invocado, isto é, quando é enviada uma mensagem a um objecto, torna-se necessário ligar a mensagem à correspondente implementação:

(Por outras palavras, associar a assinatura do método ao código que o implementa)

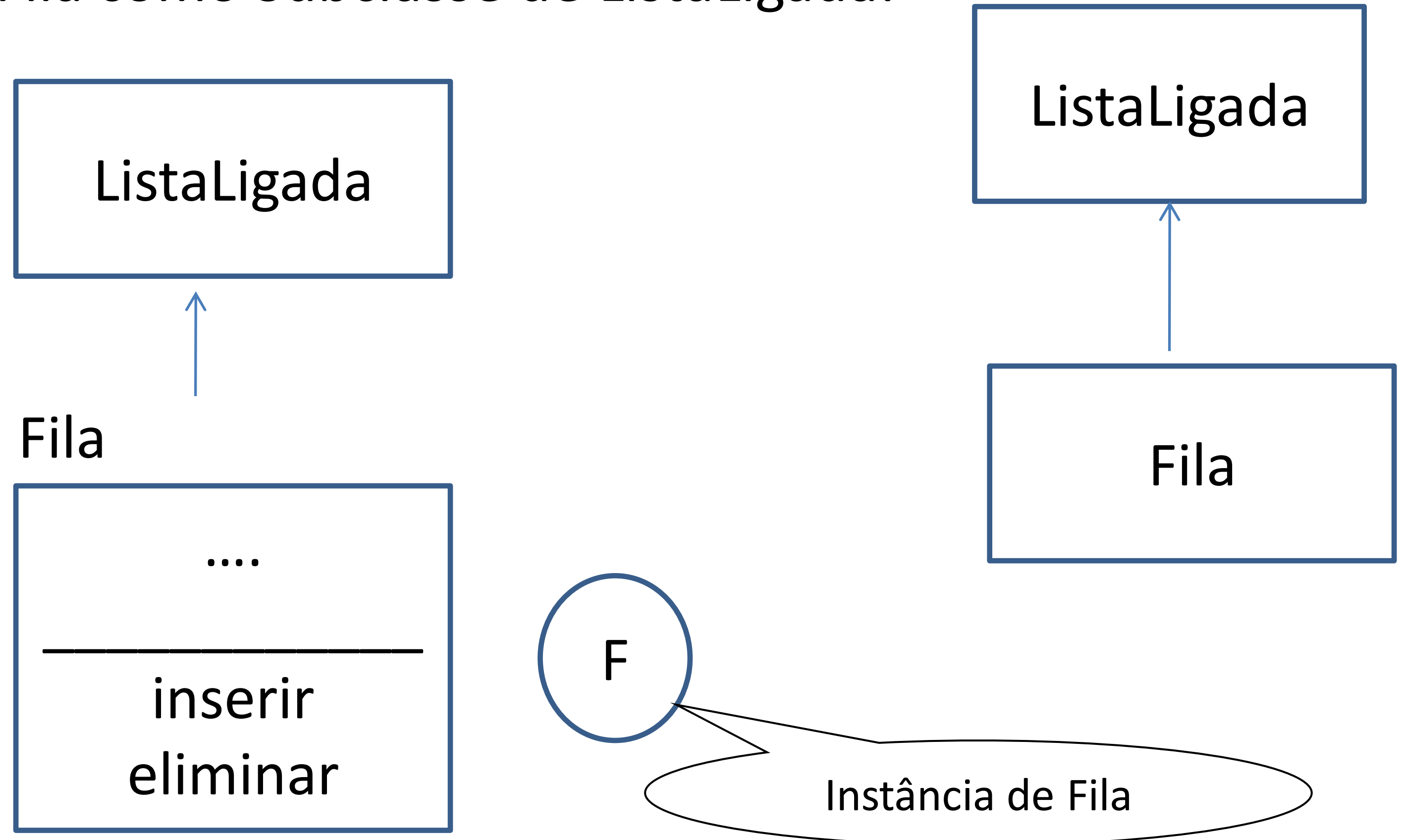
Mecanismo de Ligação ➔

Suponhamos uma classe ListaLigada:



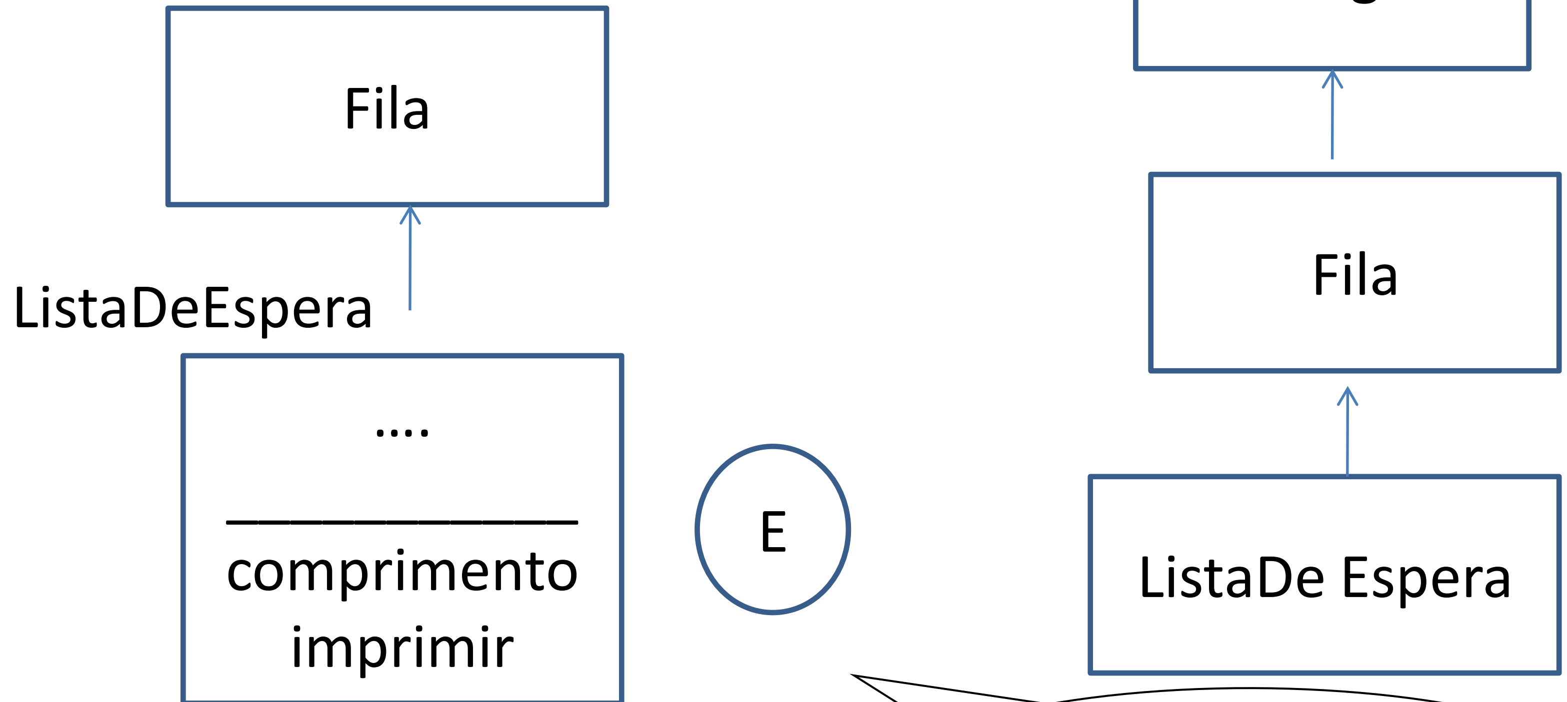
- Uma Fila pode ser facilmente implementada a partir de (isto é, reutilizando a implementação de) uma lista ligada desde que se imponham as restrições adequadas à manipulação dos seus elementos.

Definimos Fila como Subclasse de ListaLigada:



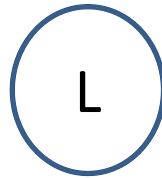
- Redefinimos os métodos Inserir e Eliminar para reflectirem a semântica da Fila.

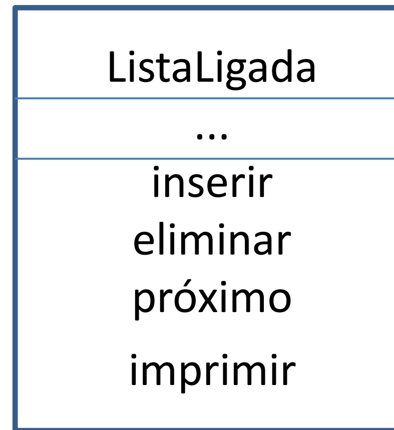
Podemos definir uma classe ListaDeEspera
Como subclasse de Fila:



- Adicionamos o método comprimento
- Redefinimos o método imprimir

Construímos a hierarquia:

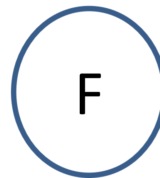


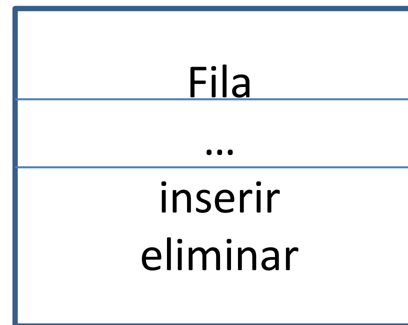


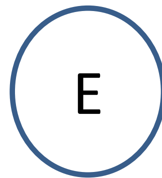
← Classe

← Atributos

← Métodos







Suponhamos agora as situações:

1º - A mensagem "imprimir" é enviada ao objecto F

```
F.imprimir()
```

- Primeiro é pesquisada a classe Fila e só depois a classe ListaLigada onde o método é encontrado.

2º - A mensagem “imprimir” é enviada ao objecto E

E.imprimir()

- O método é imediatamente encontrado na classe ListaDeEspera.

3º - A mensagem “inserir” é enviada ao objecto E

E.inserir()

- A classe ListaDeEspera é pesquisada em primeiro lugar, segue-se a classe Fila onde o método é encontrado.

Resumindo:

A hierarquia é pesquisada, em direcção à superclasse, com início na classe do objecto que recebe a mensagem.

O método mais próximo é o executado.

Observação: Algumas linguagens permitem explicitar o ponto de início da pesquisa através da especificação da superclasse juntamente com o nome do método

(ex.lo: C++), Java?

Tipos de ligação

A ligação do nome de um método a uma implementação pode ser feita

- em tempo de compilação (ligação estática)

ou

- em tempo de execução (ligação dinâmica)

Ligação estática

Abordagem mais simples.

- O compilador constrói uma tabela de classes e métodos associados. O código produzido contém as ligações entre os nomes dos métodos e as correspondentes implementações .

Vantagem:

- ligações erradas (isto é, chamadas de métodos não existentes) são detectadas em tempo de compilação.

Desvantagem:

- para introduzir alterações na ligação é necessário recompilar todo o código.

Ligação dinâmica

A correspondência entre o nome do método e a sua implementação é feita em cada invocação.

- A hierarquia é pesquisada, se o método não existir é devolvida uma mensagem do tipo “método desconhecido”.

Vantagem:

- alterações na hierarquia reflectem-se na ligação sem necessidade de recompilar todas as classes.

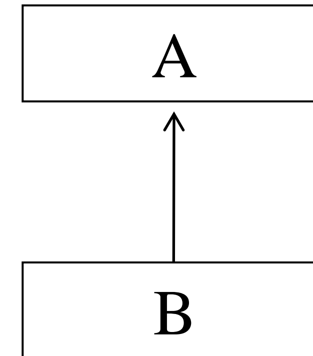
Ligação dinâmica (cont …)

Desvantagens:

-a pesquisa na hierarquia provoca alguma degradação no desempenho do sistema;

- necessidade de manipular mensagens "MétodoDesconhecido" em tempo de execução.

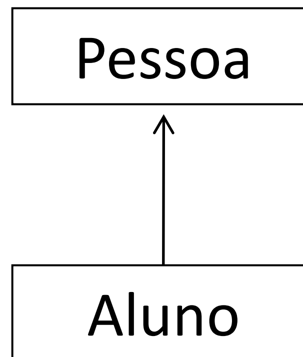
Herança de classes em Java

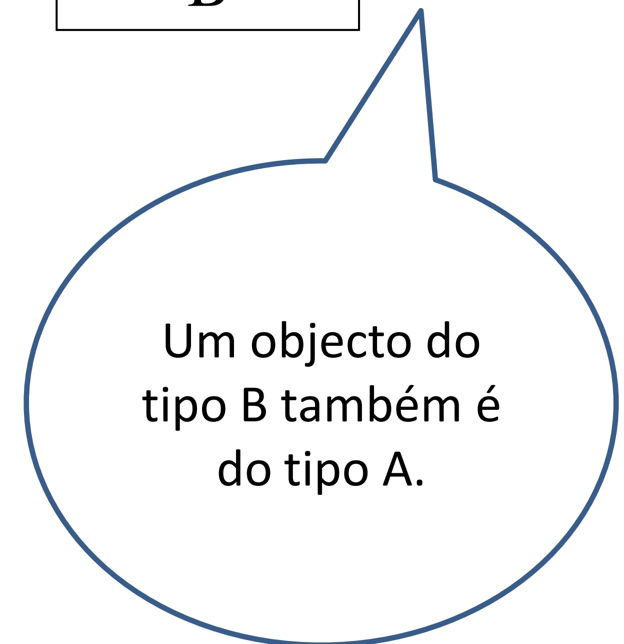


Declarar B como subclasse de A:

public class B **extends** A { ...

Ex.lo:





Um Aluno também é Pessoa.

Notas:

- Cada classe possui uma e uma só superclasse directa.
- Apenas a superclasse directa é identificada na cláusula *extends*

- A classe de topo da hierarquia é a classe **Object.**
- Quando a cláusula extends não é usada significa que a classe é subclasse directa da classe Object.

A classe Object

define o comportamento comum a todas as classes.

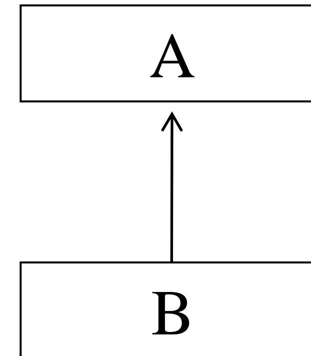
```
public class Object {

        public final Class getClass()
                                        // devolve a classe do objecto
        public String toString()
                                   // representação textual do objecto
        public boolean equals( Object obj) ...
                                          // igualdade de referências
        protected Object clone()
                          // clonagem, cria uma cópia do objecto
        …
}
```

A classe Object define métodos genéricos que normalmente necessitam de ser redefinidos.

- Qualquer instância de qualquer classe pode responder às mensagens correspondentes aos métodos públicos da classe Object.

- Se algum método não foi redefinido na classe do utilizador será executado o código definido na classe Object.

Dada uma classe A e uma subclasse B,



- B tem acesso directo a todas as variáveis e métodos da instância de A que não sejam declarados como private.

- B pode definir novas variáveis e novos métodos.

- B pode redefinir variáveis e métodos herdados.

- Uma instância de B pode responder

 a <u>mensagens</u> que correspondam a

todos os métodos públicos de B e de A.

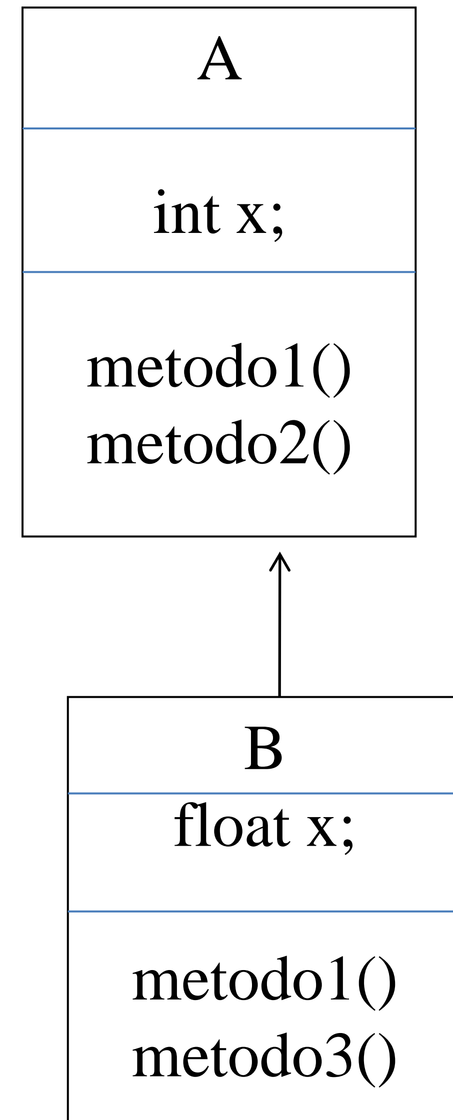
-Os <u>atributos</u> de uma instância de B

são os atributos definidos nas classes A e B.

Supondo,

B b = new B();

Quais são as variáveis e os métodos de b **?**



Principio da substitutividade:

"Declarada uma variável como sendo de uma dada classe é permitido atribuir-lhe um valor de sua classe ou de qualquer sua subclasse".

```
A a1, a2;

a1= new A();

a2 = new B();  // atribuição válida
```

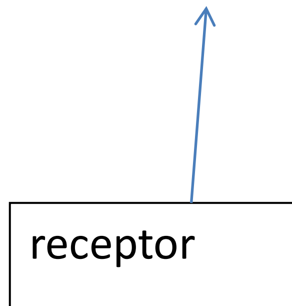
Métodos equals e clone

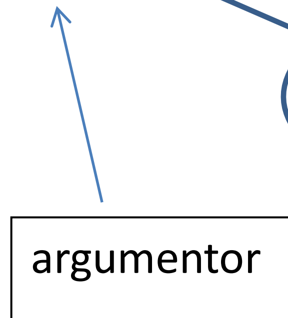
<u>Comparar objectos: método equals</u>

-o método “public boolean equals (Object)” da classe Object compara a referência do objecto, que recebe como argumento, com a referência do objecto receptor.

```
boolean b =   c1.equals( obj );
```

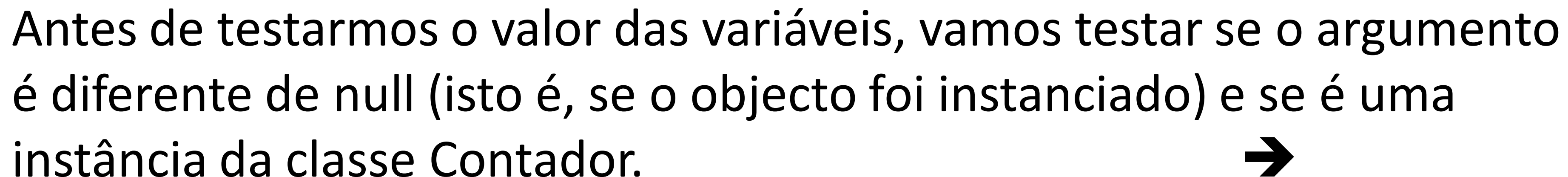
Devolve true se as referências forem iguais,
false caso contrário





Vamos redefinir o método “equals” para a classe Contador de tal forma que dois objectos do tipo Contador são iguais se o seu estado for igual.

Isto é, dois objectos do tipo Contador serão iguais, se as suas variáveis **conta** tiverem o mesmo valor.



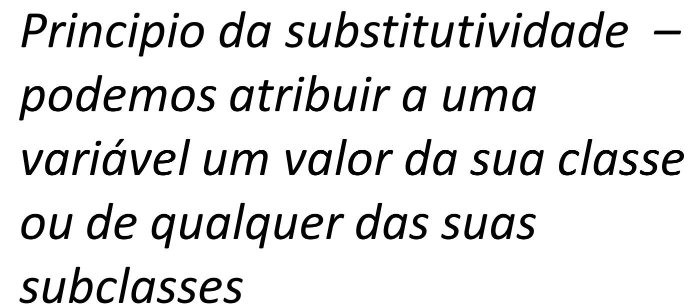
Antes de testarmos o valor das variáveis, vamos testar se o argumento é diferente de null (isto é, se o objecto foi instanciado) e se é uma instância da classe Contador. ➔

```
public boolean equals (Object  obj ) {
    if (obj != null  &&  obj instanceof Contador ){
        // compara as variáveis de instância dos dois objectos
        return ( this.conta = = ( (Contador) obj ).conta; )
    } else {
        return false;
    }
}
```

(&&) E condicional

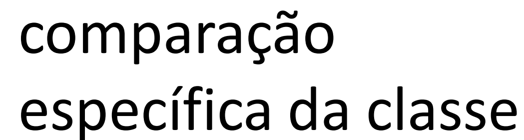
Utilização do método:

```
...
Contador c1,c2;

c1= new Contador();
c2= new Contador ();

...

boolean iguais;
iguais = c1.equals(  c2  );
...
```



Para uma qualquer classe Exemplo:

```
public boolean equals (Object obj) {

        if ( obj != null && obj instanceof Exemplo ){

                return ...
         }

        return false;

}
```



O método Clone

Queremos definir um método que crie e devolva uma cópia do objecto receptor.

-Essa cópia deve ser tal que o objecto criado e o objecto que recebe a mensagem:

1 – não são o mesmo objecto

```
y = x.clone() ;    //   y != x;
```

2 – são instâncias da mesma classe

```
x.clone().getClass()  ==  x.getClass()
```

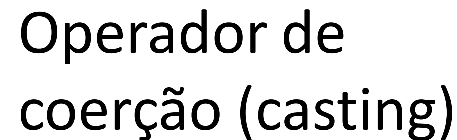
3 – têm o mesmo valor nas variáveis de instância

```
x.clone().equals( x )    ==   true
```

```
public Object clone() {

        Contador c = new Contador ( this.conta);

        return c;
}
```

Utilização:

```
Contador c1,c2;
c1=new Contador();
...
c2 = (Contador) c1.clone();
```



Comparação de Strings

1 - Valores constantes do tipo String têm a mesma referência.
"XPTO" = = "XPTO" ➔ expressão com valor true

2 – Strings construídas em tempo de compilação são tratadas como valores constantes do tipo String.

String s1 = "XPTO";
String s2 = "XPTO"

s1 = = s2 ➔ expressão com valor true

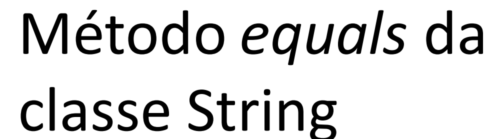
3 – Strings construídas em tempo de execução têm referências distintas:

String s1 = new String(“XPTO”);
String s2 = new String (“XPTO”);

s1 = = s2 ➔ false

mas

s1.equals(s2) ➔ true



Ex.lo 1

```
public class Exemplo {

    static String s0;

    public static void setS0 (String s){

        s0 = s;

    }
```

// ainda na classe exemplo.

```
public static void main (String [] args) {
   String s = "XPTO";
   setS0 (s);

  System.out.println ( s + "    "  +  s0);

  System.out.println ( s = =  s0);

  s = "XX";              //é criada uma nova instância da String s;

  System.out.println ( s + "    "  +  s0)
 }
} // fim da classe exemplo
```

Output:

XPTO   XPTO
true
XX XPTO

E se no método **setS0**

substituirmos

s0 = s

por **s0= new String (s);**

O que acontece?

s = = s0 ???

s.equals (s0) ????

Exercício: testar exemplo anterior

Qual o output do seguinte programa:

```
public static void main(String[] args) {
    String s1 = new String ("XPTO");
    String s2 = "XPTO";
    System.out.println ( s1 == s2);
    System.out.println ( s1.equals(s2) );

    String s3 = "XPTO";
    System.out.println ( s2 == s3);
    System.out.println ( s2.equals(s3) );
}
```

false
true
true
true

Regra:

- comparar Strings sempre com o método equals definido na classe String.

A classe Vector

A principal limitação dos arrays advém do seu carácter estático. É necessário estabelecer a dimensão do array aquando da sua definição e não é possível exceder este limite máximo.

Que acontece em problemas para os quais não é possível determinar, à partida, esta dimensão?

O ideal seria utilizar uma estrutura (dinâmica) cuja dimensão se adapte às necessidades de armazenamento durante a execução do programa ...

Temos pois duas alternativas:

i) implementar uma classe com a funcionalidade pretendida

Ou

ii) (re)utilizar uma classe com as características desejadas, se a mesma já existir!

Neste caso podemos (devemos) optar pela segunda escolha uma vez

que no pacote java.util temos disponível a implementação da classe

Vector que se distingue dos arrays pelas seguintes características:

- Um Vector pode crescer ou decrescer de tamanho.

- Os vectores armazenam objectos (não podem armazenar tipos simples! A menos que sejam “embrulhados” em objectos... Lembram-se das classes **I**nteger, **D**ouble,...?).

- Um Vector pode conter objectos de diferentes tipos.

Em conclusão, a classe Vector implementa uma abstracção de dados que representa uma estrutura linear indexada a partir do índice 0 (deste ponto de vista, análoga ao array) sem limite de dimensão.

Vejamos **alguns dos métodos da classe Vector**
(para uma referência completa estudar a API da classe):

```
Vector()          // construtor vector vazio, dimensão inicial zero.

Vector( int capacidadeInicial)
                  // construtor vector vazio, com dimensão inicial.

void addElement( Object elemento)
                  // adiciona o elemento especificado ao final do vector.

void insertElementAt( Object obj, int indice)
                  // insere o elemento especificado na posição índice.
```

void removeElementAt (int indice)
/ / remove o elemento na posição índice.

void setElementAt (Object obj, int indice)
// substitui o elemento da posição índice pelo objecto dado.

Object elementAt (int indice)
// devolve o componente presente no índice.

void clear()   // remove todos os elementos do vector.

Object clone()   // devolve uma cópia do vector.

boolean contains(Object elemento)
/ verifica se o objecto especificado é um componente do vector

Object firstElement()
// devolve o primeiro componente (indice 0) do vector.

Object lastElement()
// devolve o último componente do vector.

int indexOf(Object elemento)
// procura o índice da 1ª ocorrência de elemento (utiliza o método equals).

int indexOf(Object elemento, int indice)
// inicia a procura anterior na posição índice.

boolean isEmpty()
// verifica se o vector não tem componentes

int size()
// devolve a dimensão actual.

boolean equals(Object o)
//permite a comparação de 2 vectores.

Exemplo de utilização de objectos do tipo Vector

```
…
public static void main (String [] args) {

Vector v = new Vector();

v.addElement( “Maria”);
 v.addElement (“João”);
String s =  (String) v.firsElement();
System.out.println (v.toString()  + “ , ”+   s);
…
```

Output: [Maria, João] , Maria

O Vector v pode conter objectos de qualquer tipo.
Não há verificação de tipos.

A partir da versão 5 do Java, a verificação de tipos pode ser feita durante a compilação, usando **tipos genéricos**:

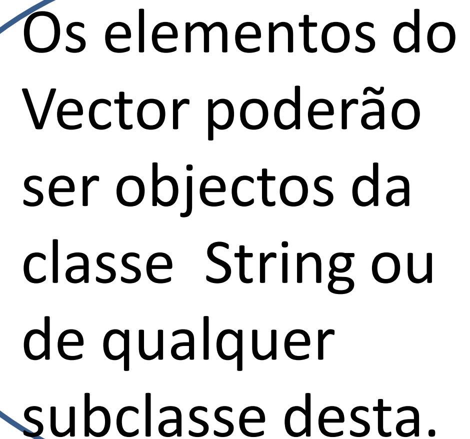
Um tipo genérico é um tipo referenciado que usa na sua definição um ou mais tipos de dados como parâmetros.

Por exemplo, o tipo vector Vector<E> em que **E** pode ser qualquer classe (ou interface!!)

A instanciação de um tipo genérico para um valor concreto de E, dá origem a um tipo parametrizado.

Por exemplo, o código :

```
Vector<String> v1;

v1 = new Vector<String> ();

v1.addElement("Joana");
v1.addElement("Manuel");

String ss = v1.firstElement();

System.out.println (v1.toString() + " ," + ss);
```



Usa o tipo Parametrizado Vector <String>, com verificação estática de tipos (isto é, em tempo de compilação).