# AULA TEÓRICA 9

**Tema 8. Herança (continuação)**

- Sobreposição de métodos.
- Variáveis e métodos estáticos.
- Hierarquia de classes.
- Polimorfismo

# Sobreposição de métodos

Em Java é permitido o uso de **vários métodos cujos nomes são iguais**. Isso é prático quando se pretende utilizar o mesmo método para diferentes tipos de dados.

Voltamos ao exemplo das classes **`Livro`** e **`Dicionario`**:

```
public class Livro
{ protected int paginas;
  public Livro(int p)
  { paginas = p; }

  public void descreve()
  { System.out.println("Livro com "+paginas +" paginas"); }
}
public class Dicionario extends Livro
{ private int entradas;
  public Dicionario(int pag, int ent)
  { super(pag);  //cria automaticamente o objecto da super classe
    entradas = ent;
  }
  public void descreve()
  { System.out.println("Dicionario");
    System.out.println(" Numero de paginas " + paginas);
    System.out.println(" Numero de entradas "+ entradas);
  }
}
```

As classes `Livro` e `Dicionario` incluem ambas um método `public` de nome `descreve()`. Uma vez que a subclasse herda os métodos não privados da super classe, a subclasse fica com acesso a dois métodos com a mesma assinatura (nome e parâmetros), o que poderia parecer errado.

Nesta situação, a **subclasse sobrepõe o seu método ao método herdado** da super classe, ou seja, a subclasse **redefine** o método que herdou. O novo método tem que ter a mesma assinatura que o herdado, mas pode ter conteúdo diferente.

Como já foi referido, a palavra reservada `super` referencia a superclasse da classe onde é utilizada. Esta palavra já foi utilizada para invocar o construtor da superclasse a partir do construtor da subclasse, mas pode também ser usada para invocar qualquer método da superclasse a partir de uma sua subclasse.

Esta capacidade só se torna útil quando ao método da superclasse que se pretende chamar foi sobreposto outro na subclasse, pois, caso contrário, a chamada pode ser feita normalmente, utilizando apenas o nome do método.

# Variáveis e Métodos estáticos

As declarações de variáveis efectuadas numa classe podem ser de dois tipos: **variáveis de instância** ou **variáveis da classe**.

**As variáveis de instância** servem para armazenar atributos específicos de cada objecto.

Contrariamente **as variáveis de classe não existem nos objectos**. A sua declaração leva o compilador a criar apenas uma variável em memória, que é partilhada por todos os objectos que vierem a ser criados a partir da classe. Desta forma, as alterações ao valor da variável que forem efectuadas por um objecto manifestam-se em situações em que seja necessário partilhar informação entre os objectos de uma classe.

No exemplo dos rectângulos, pode haver interesse em saber quantos objectos da classe `Rectangulo` foram criados até um determinado momento. Esta informação não pode ser guardada em cada objecto, pois estes só conhecem a sua própria existência. A solução é definir uma variável de classe que seja incrementada quando é criado um novo objecto:

```
public class ExemploVarEstatica {
  public static int contaRect = 0;      //variável de classe
  private int posX,posY,comp,larg;      //variáveis de instância
  .....
```

A palavra reservada **`static`** indica ao compilador que não deve produzir uma nova cópia desta variável quando for criado um novo objecto, mas que deve ser criada apenas uma variável e que todos os objectos da classe a devem partilhar.

Tendo sido incluída a declaração da variável `contaRect`, poderia agora ser alterado o respectivo construtor, de forma a contar quantos objectos são criados:

```
public ExemploVarEstatica(int pX, int pY, int c, int l){
  posX = pX;  posY = pY;
  comp = c;   larg = l;
  contaRect++;
}
```

A **identificação de uma variável da classe** é conseguida juntando o nome da classe, um ponto e o nome da variável:

```
ExemploVarEstatica.contaRect
```

Se o acesso for feito a partir da classe onde a variável foi definida, o nome da classe e o ponto podem ser omitidos.

Para além das variáveis, também os **métodos** podem ser declarados como **estáticos** e designam-se por **métodos de classe**. Eles funcionam de forma semelhante à das variáveis estáticas, ou seja, funcionam a nível da classe e não são copiados para os objectos criados a partir da classe.

Deste modo, podem ser chamados sem que haja necessidade de criar um objecto.
Por ex., os métodos da classe `Math` são chamados da forma: `Math.sqrt()`.

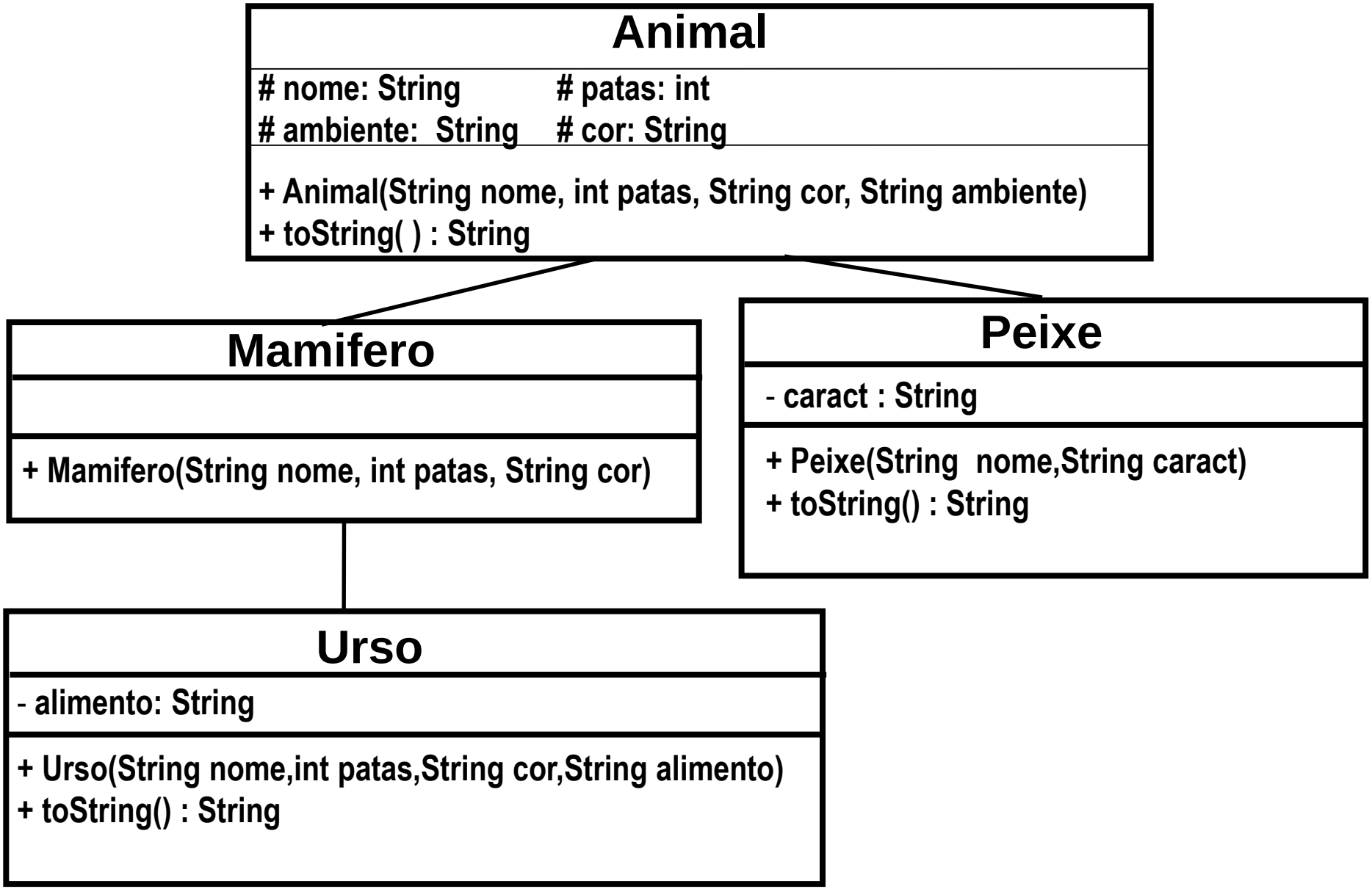
Sabemos que a execução do programa começa pelo chamada do seu método `main`(). Esta chamada é feita sem que seja necessário criar qualquer objecto, pelo que este método funciona ao nível da classe, ou seja, é um método estático:

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

Tendo em conta que os métodos estáticos funcionam sem que tenha que existir qualquer objecto da classe onde estão definidos, **não é permitido efectuar a chamada de métodos não estáticos a partir de métodos estáticos** ou utilizar variáveis de instância nestes métodos, porque tanto os métodos não estáticos como as variáveis de instância só existem nos objectos e os métodos estáticos podem ser chamados sem que haja criação de qualquer objecto.

# Hierarquia de classes

Uma subclasse pode ser super classe de uma outra classe. De igual modo, podem ser criadas várias classes a partir da mesma super classe.



```
class Animal //todo codigo está gravado num unico ficheiro AnimalExHierarq.java
{ protected String nome,cor,ambiente;
  protected int patas;

  public Animal(String nome,int patas,String cor,String ambiente)
  { this.nome=nome;
    this.patas=patas;
    this.cor=cor;
    this.ambiente=ambiente;
   }
  public String toString()
  { return "Animal:"+nome+"\nPatas:"+
           patas+"\nCor:"+cor+"\nAmbiente:"+ambiente+"\n****"; }
}
==================================================
class Peixe extends Animal
{ private String caract;
  public Peixe(String nome, String caract)
  { super(nome,0,"Cinzenta","Mar");
    this.caract = caract;
  }
  public String toString()
  { return super.toString()+"\nCaracteristica:"+caract+"\n====";}
}
```

```
class Mamifero extends Animal
{ public Mamifero(String nome,int patas,String cor)
  { super(nome,patas,cor,"Terra"); }
}
class Urso extends Mamifero
{ private String alimento;
  public Urso(String nome,int patas,String cor,String alimento)
  { super(nome,patas,cor);
    this.alimento = alimento;
  }
  public String toString() { return super.toString()+
                             "\nAlimento:"+alimento+"\n------"; }
}
=============================================================
public class AnimalExHierarq
{ public static void main(String args[])
  { Mamifero a1 = new Mamifero("Leao",4,"Amarelo");
    Peixe    a2 = new Peixe("Tubarao","Barbatanas e Caudas");
    Urso     a3 = new Urso("Urso-do-Canada",4,"Vermelho","Mel");
    System.out.println("ANIMAIS:");
    System.out.println(a1.toString());
    System.out.println(a2.toString());
    System.out.println(a3.toString());
  }
}
```

**Output:**

```
ANIMAIS:
Animal:Leao
Patas:4
Cor:Amarelo
Ambiente:Terra
**********************
Animal:Tubarao
Patas:0
Cor:Cinzenta
Ambiente:Mar
**********************
Caracteristica:Barbatanas e Caudas
======================
Animal:Urso-do-Canada
Patas:4
Cor:Vermelho
Ambiente:Terra
**********************
Alimento:Mel
----------------------
```

A definição da forma **como** a hierarquia de classes existente na linguagem deve ser expandida para resolver um determinado problema é uma questão em POO. Geralmente aceita-se que os dados e comportamentos comuns a um conjunto de classes devem ser colocados tão acima quanto possível na hierarquia, sendo herdados pelo conjunto de classes.

# Polimorfismo

O **polimorfismo** é mais um conceito fundamental de POO. No exercício sobre livros verificou-se que a referência `liv`, tendo sido declarada como referência para um objecto da classe `Livro`, pode também referir-se um objecto da classe `Dicionario`, pois esta descende de `Livro`. Estas classes têm definido um método com o mesmo cabeçalho, mas conteúdos diferentes: `public void` **`descreve`**`()`

Um aspecto importante a considerar é que **uma referência para um objecto de uma classe pode também ser utilizada para referenciar um objecto de uma classe relacionada com a sua herança.**

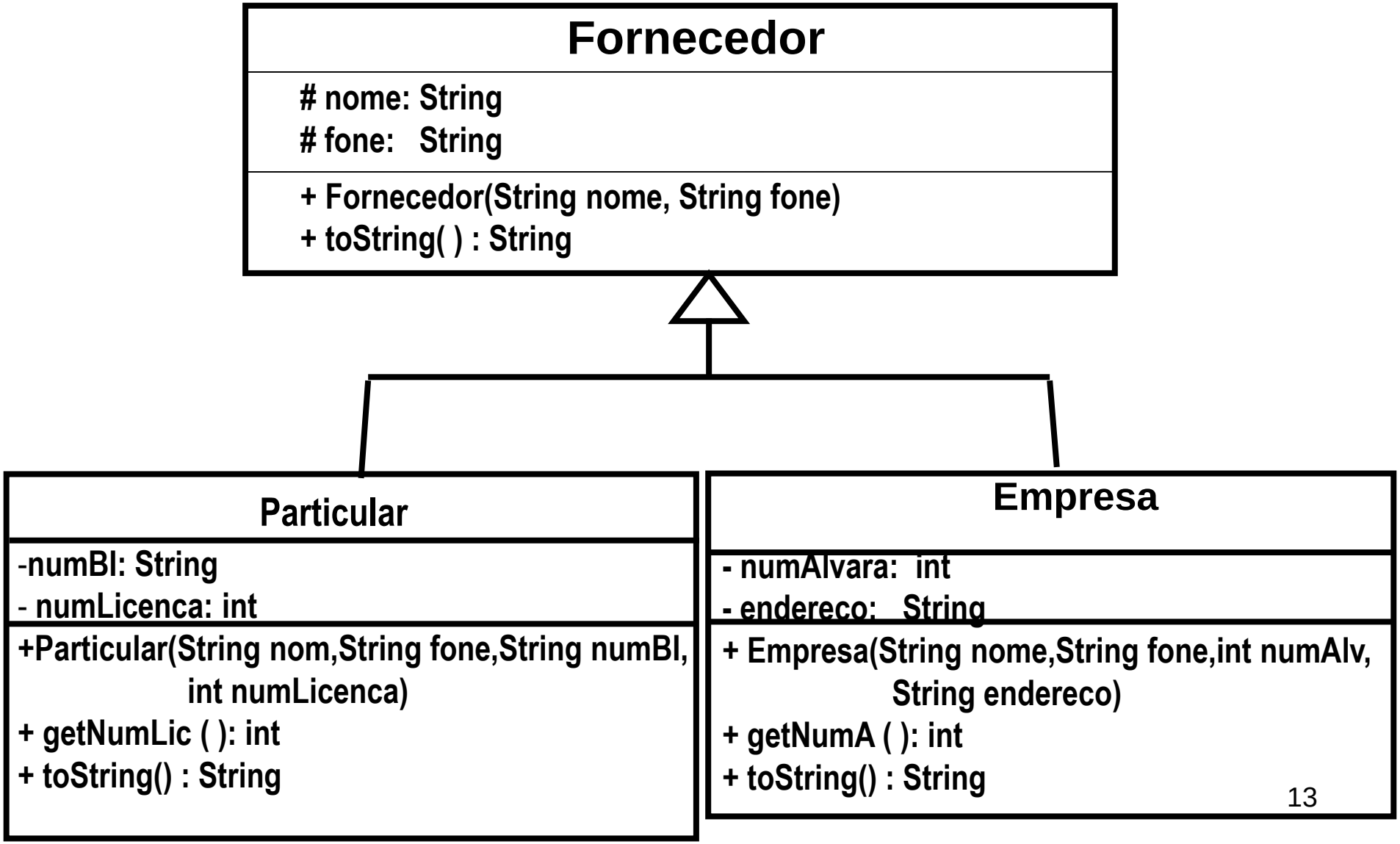
```
public class ExemploMesmaRefer {
  public static void main (String args[]){
    Livro liv;
    liv = new Livro (300);
    liv.descreve();
    liv = new Dicionario(600,10000);
    liv.descreve();
  }
}
```

A questão que se coloca reside em saber qual dos métodos vai ser executado. A resposta é que será executado o método definido na classe do objecto que no momento for referenciado por `liv`. Se `liv` referenciar um objecto da classe `Livro`, será o método `descreve()` definido nesta classe a ser executado. Se `liv` referenciar um objecto da classe `Dicionario`, será o método `descreve()` desta classe a ser executado.

A esta **capacidade da mesma instrução executar métodos diferentes em função dos objectos em causa chama-se polimorfismo**.

No caso da chamada de um método estar dentro de um ciclo, a mesma instrução pode chamar métodos diferentes em diferentes iterações do ciclo.

*Exemplo*: A classe **Fornecedor** representa duas características comuns para todos os fornecedores (nome e fone), a classe **Particular** adiciona duas características específicas que somente aqueles fornecedores que são pessoas possuem, a classe **Empresa** adiciona outras duas características que somente empresas possuem.



```
. . .
. . .

public class FornecedorTest
{ public static void main(String[] args)
  { Fornecedor empr[] = new Fornecedor[3];
    empr[0]=new Empresa("Delta","(84)397-2457",21,"COOP,PH7");
    empr[1]=new Particular("Ze Pedro","(45)277-2420",
                                  "324567F","A24");
    empr[2]=new Empresa("Ceres","(82)423-5566",132,"Lenine,43");
    for (int i=0;i<3;i++)
      System.out.println(empr[i].toString());
  }
}
```

O resultado da execução deste código:

```
Dados do Fornecedor:
Nome: Delta      Fone: (84)397-2457
***Dados do Fornecedor Empresa:
Numero Alvara: 21        Endereco: COOP,PH3
Dados do Fornecedor:
Nome: Ze Pedro   Fone: (45)277-2420
***Dados do Fornecedor Particular:
Numero de BI: 324567F    Numero de Licenca: A24
Dados do Fornecedor:
Nome: Ceres      Fone: (82)423-5566
***Dados do Fornecedor Empresa:
Numero Alvara: 132       Endereco: Lenine,43
```

**Há dois aspectos importantes que devem ser realçados:**

É a utilização de um array em que os elementos não são todos do mesmo tipo. Há elementos do array que armazenam referências para objectos da classe `Empresa` e outros que armazenam referências para objectos da classe `Particular`.

No entanto, este facto é coerente com a indicação de que **uma referência pode referenciar objectos de classes diferentes, desde que tenham relações de herança entre si.**

A instrução `empr[i].toString()` colocada no interior do ciclo `for`, não executa sempre o mesmo método.

Na 1ª iteração `(i=0)`, `empr[0]` refere um objecto da classe `Empresa`, pelo que é o método `toString()` desta classe que é executado.

Já na 2ª iteração `(i=1)`, `empr[1]` refere um objecto da classe `Particular`, sendo executado o método `toString()` desta classe e assim sucessivamente.

Nestes casos, em que se tira o partido das potencialidades do polimorfismo, a definição de qual o método que é executado só pode ser feita durante a execução, pois só nessa altura é que se pode saber qual a classe do objecto e, consequentemente, qual o método a utilizar.

**Referência bibliográfica**:

António José Mendes; Maria José Marcelino.
***"Fundamentos de programação em Java 2".*** FCA. 2002.

Elliot Koffman; Ursula Wolz.
***"Problem Solving with Java".*** 1999.

F. Mário Martins;
***"Programação Orientada aos objectos em Java 2",*** FCA, 2000,

John Lewis, William Loftus;
*"Java Software Solutions: foundation of program design",* **2nd edition, Addision-Wesley**

John R. Hubbard.
***"Theory and problems of programming with Java".*** Schaum's Outline series. McGraw-Hill.

H. Deitel; P. Deitel.
***"Java, como programar".*** 4 edição. 2003. Bookman.

Rui Rossi dos Santos.
**"*Programando em Java 2– Teoria e aplicações"***. Axcel Books. 2004