estgoh
Politécnico de Coimbra

Programação
Programação III

Programação Orientada a Objectos
Herança e Polimorfismo

Marco Veloso
marco.veloso@estgoh.ipc.pt

Agenda

Programação Orientada a Objectos – Herança e Polimorfismo

Introdução à Linguagem Java
- Paradigmas de Programação
- Linguagem Java

Programação Orientada a Objectos
- Objectos
- Classes
- Herança
- Polimorfismo

Tratamento de Excepções

Estruturas de dados
- Tabelas unidimensionais
- Tabelas multidimensionais
- Vectores
- Dicionários (Hashtables)
- Collections

Ficheiros
- Manipulação do sistema de ficheiros
- Ficheiros de Texto
- Ficheiros Binários
- Ficheiros de Objectos
- Leitura de dados do dispositivo de entrada

Herança

Conceito de Herança

Um dos **conceitos fundamentais em programação orientada para objectos** é a **herança**

Quando utilizada correctamente **permite a reutilização de código e facilita o desenho de software**

A **herança permite derivar uma nova classe a partir de outra já existente**

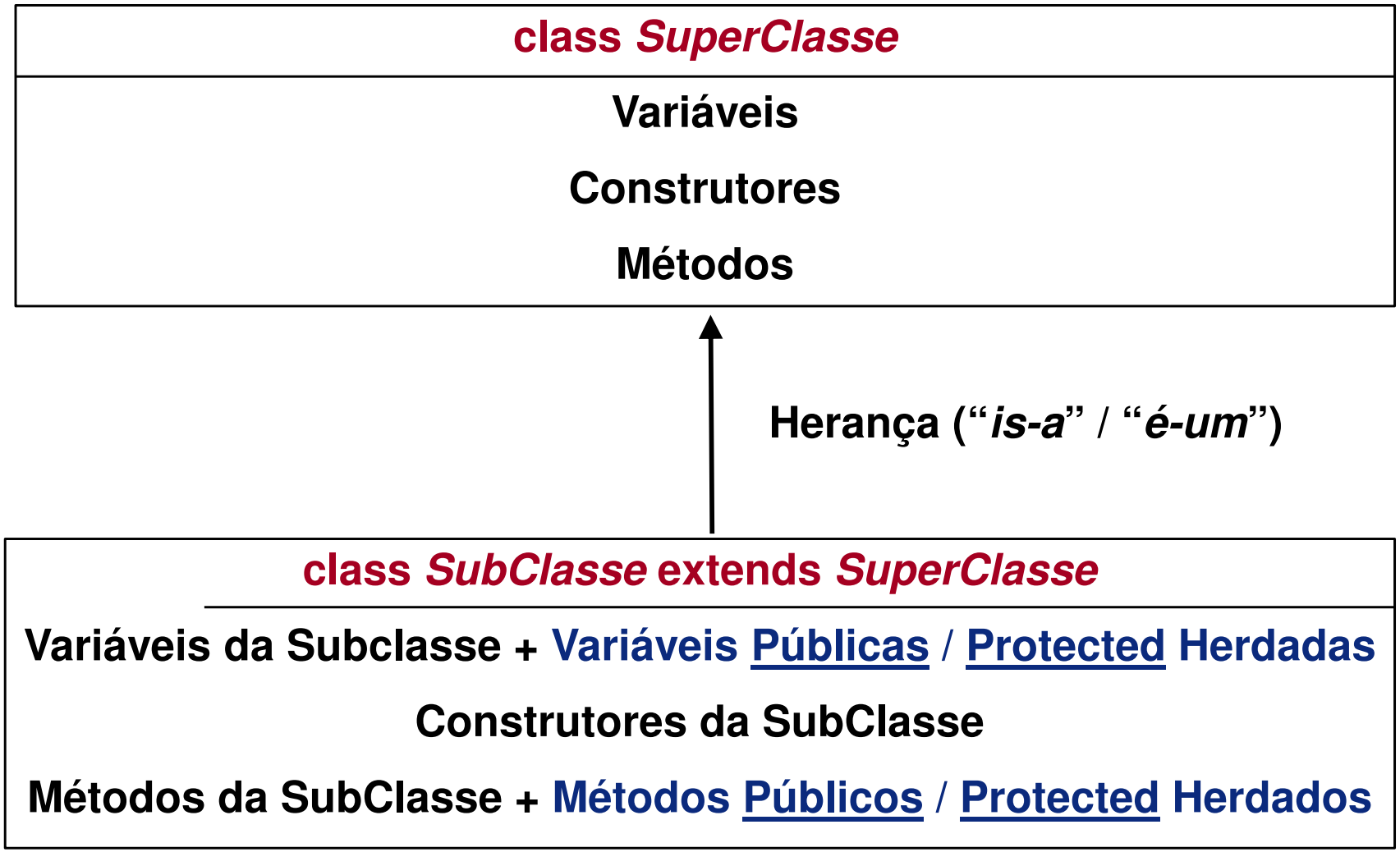
À **classe já existente** dá-se o nome de **superclasse**

À **nova classe** chama-se **subclasse**

Também se pode usar o **conceito *pai-filho*** para descrever a relação entre uma subclasse e a sua superclasse

Uma subclasse herda características e comportamentos da sua superclasse (**herda as variáveis de instância e os métodos nela definidos**, desde que não possuam o modificador `private`)

Representação de Herança



Palavra reservada *extends*

A relação de herança deve criar uma **relação *é-um* (*is-a*)**, significando que a **subclasse é uma versão mais específica da superclasse**

Exemplo: **dicionário** ***é-um*** **livro**, pelo que uma classe que represente um dicionário pode ser subclasse de uma classe mais genérica que representa um livro

Em Java, **a relação de herança é estabelecida usando a palavra reservada `extends`**

```
class Dicionario extends Livro {
        // conteúdo da classe
}
```

Exemplo

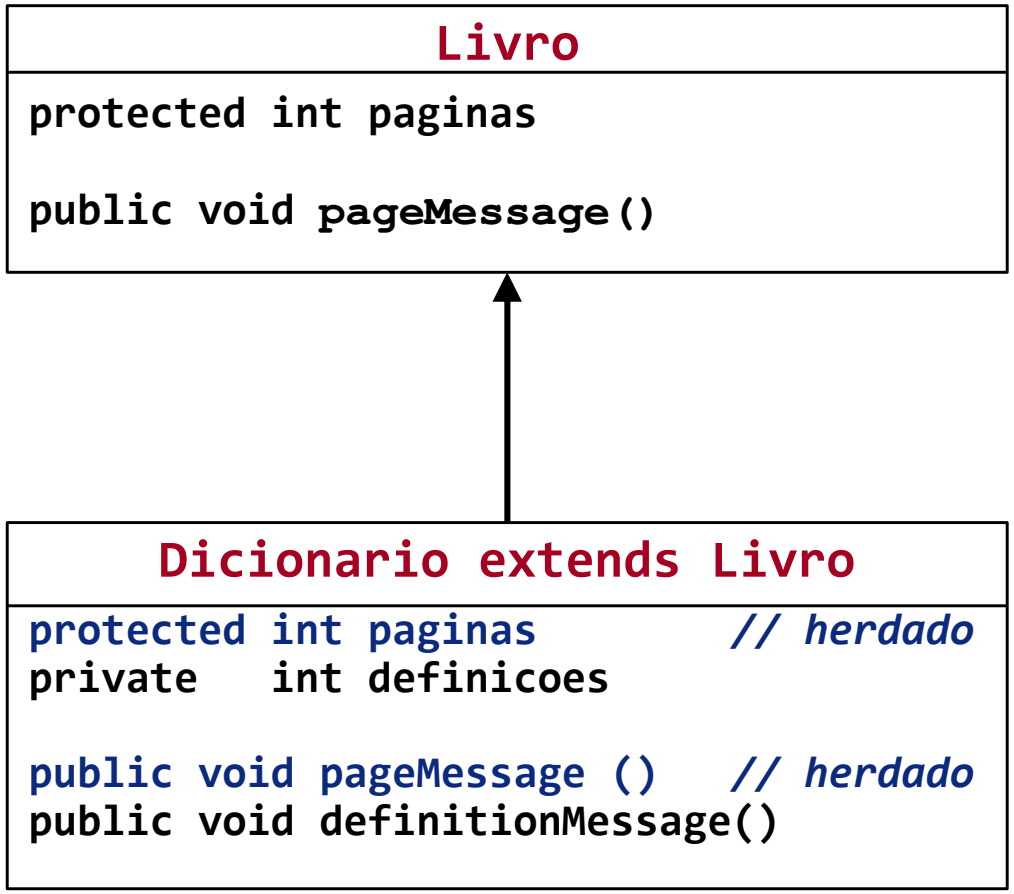

```
// Classe Livro será uma superclasse
class Livro {
  protected int paginas = 1500;

  public void pageMessage () {
    System.out.println ("Número de páginas: " + paginas);
  }
}


// Classe Dicionario herda os métodos e variáveis da classe Livro
class Dicionario extends Livro {
  private int definicoes = 52500;

  public void definitionMessage () {
    System.out.println ("Número de definicoes: " + definicoes);
    System.out.println ("Média por página: " +
                                        definicoes/paginas);
  }
}
```

Exemplo


Livro
protected int paginas
public void pageMessage()
Dicionario extends Livro
protected int paginas // herdado
private int definicoes
public void pageMessage () // herdado
public void definitionMessage()

Exemplo

```
class Palavras {
   public static void main (String[] args) {
      Dicionario larousse = new Dicionario ();
      larousse.definitionMessage(); // método da classe Dicionario
      larousse.pageMessage();       // método da classe Livro
                  //Dicionario herda o método pageMessage() de Livro
   }
}
```

Notar que **não foi criado explicitamente um objecto da classe `Livro`**

No entanto, **a criação de um objecto da classe `Dicionario` implica também a criação de um objecto da classe `Livro`**, já que esta é a sua superclasse

Como já foi visto, **a utilização de modificadores de visibilidade serve para controlar o acesso aos métodos** e variáveis de uma classe

O modificador **`public`** **indica que a variável ou método pode ser acedida a partir de qualquer classe**

O modificador **`private`** **indica que a variável ou método só pode ser acedida a partir da própria classe**

Isto significaria que **seria necessário declarar os métodos e variáveis como `public` para que fossem herdados.** No entanto, isto viola os princípios do encapsulamento

Podemos usar o modificador **`protected`** para indicar que **os métodos e variáveis podem ser acedidos a partir de subclasses**, mas **não das restantes classes**

Acesso aos construtores da superclasse

Apesar de terem visibilidade **`public`**, os **construtores não são herdados**

Por vezes é necessário **invocar o construtor da superclasse a partir da subclasse**, por forma a inicializar devidamente as variáveis herdadas (reutilizar a implementação)

A referência **`super`** pode ser usada para **referenciar a superclasse** (**`super.metodo()`**) e é **frequentemente usada para invocar o seu construtor** (**`super()`**), devendo ser a **primeira instrução no construtor**

A referência **`super`** quando usada num construtor procura **referenciar o construtor da superclasse**, e quando usada fora do construtor pretende **invocar um método da superclasse** (que não o construtor)

Exemplo

```
class Livro {
    protected int paginas;

    // Construtor, inicializa paginas com um valor dado
    public Livro (int numPaginas) {
        paginas = numPaginas;
    }

    public void pageMessage () {
        System.out.println ("Número de páginas: " + paginas);
    }
} // classe Livro
```

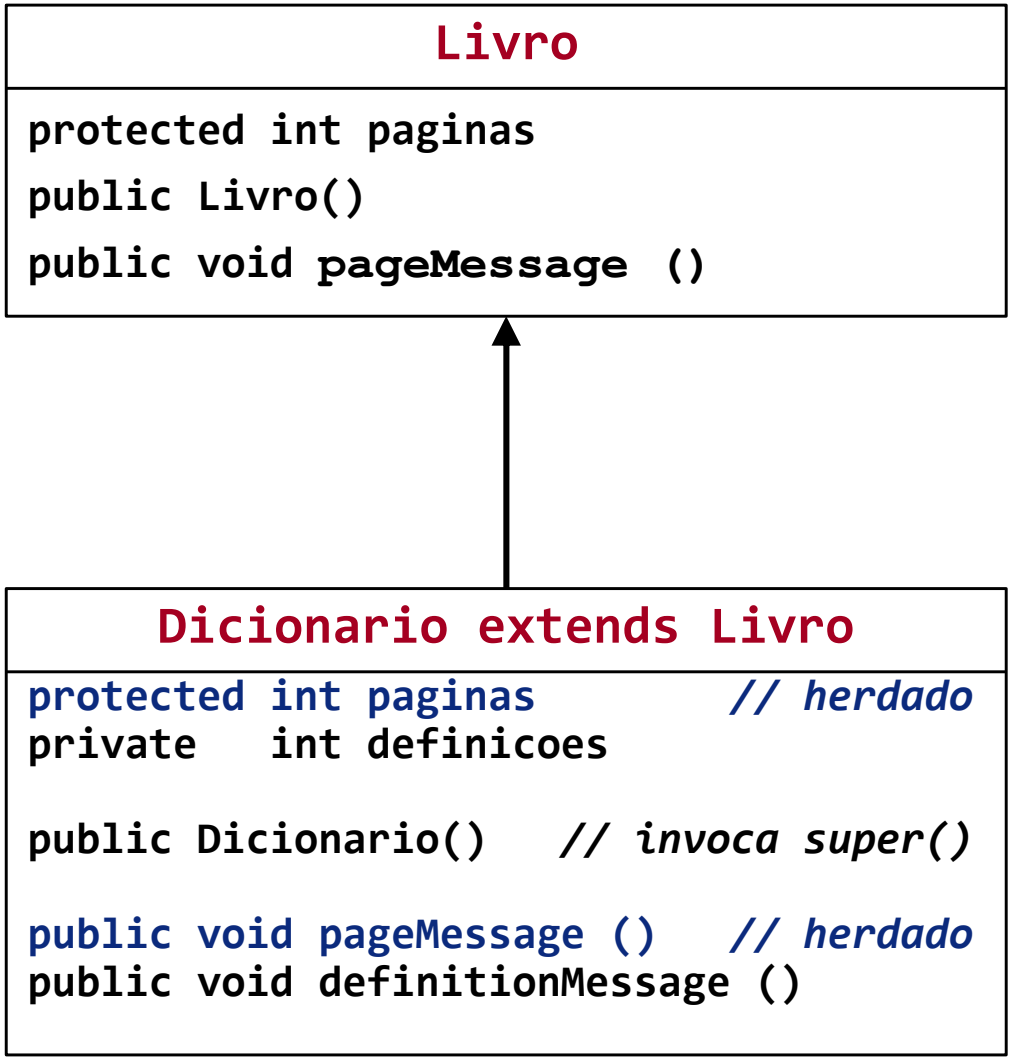
Exemplo

```
class Dicionario extends Livro {
    private int definicoes;


    // Construtor de dicionario. Tem que chamar o construtor de Livro
    // para inicializar o número de páginas
    public Dicionario (int numPaginas, int numDefinicoes) {
        super (numPaginas);      // primeira instrução do construtor
        //paginas = numPaginas
        definicoes = numDefinicoes;
    } // construtor Dicionario


    public void definitionMessage () {
        System.out.println ("Número de definições: " +
                                                definicoes);
        System.out.println ("Média por página: " +
                                              definicoes/paginas);
    }
} // classe Dicionario
```

Exemplo


Livro
protected int paginas
public Livro()
public void pageMessage ()
Dicionario extends Livro
protected int paginas // herdado
private int definicoes
public Dicionario() // invoca super()
public void pageMessage () // herdado
public void definitionMessage ()

```
class Words2 {
    public static void main (String[] args) {
        Dicionario larousse = new Dicionario (1500, 52500);
        larousse.pageMessage();
        larousse.definitionMessage();
    }
}
```

Um membro herdado por uma subclasse pode ser invocado directamente, tal como se tivesse sido nela declarado

Mas, os membros não herdados estão também definidos na subclasse, **podendo ser utilizados indirectamente através dos métodos herdados da superclasse**

```
class Comida {
    final private   int CALORIAS_POR_GRAMA = 9;
          private   int gordura;
          protected int fatias;

    Comida (int numGramasGord, int numFatias) {
          gordura = numGramasGord;
          fatias = numFatias;
    }


    // Devolve o número de calorias de um item (Método privado)
    private int calorias() {
        return gordura * CALORIAS_POR_GRAMA;
    }


    // Devolve o núm. de gramas de gordura por fatia (público, usa privado)
    public int caloriasPorFatia() {
          return (calorias() / fatias);
    }
}
```

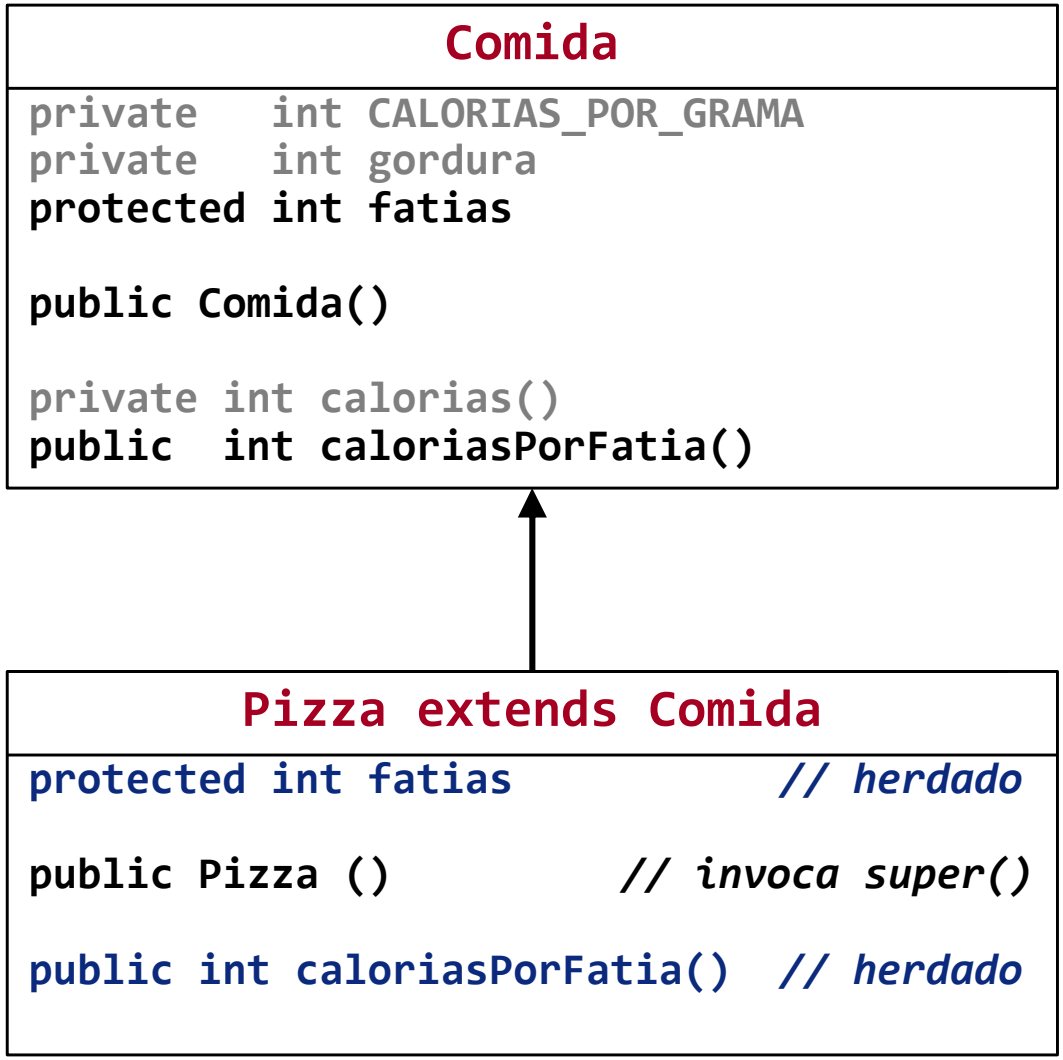
Exemplo


```
class Pizza extends Comida {
     // Representa uma pizza com uma dada quant. de gordura, assumindo 8 fatias
     public Pizza (int quantGordura) {
          super (quantGordura, 8);
     }
} // class Pizza


class Comendo {
     public static void main (String[] args) {
          Pizza especial = new Pizza (275);
          System.out.println ("Calorias por fatia: "+
                                   especial.caloriasPorFatia());
     }
} // class Comendo
```

Chama um método herdado da classe `Comida` que, por sua vez, chama um método não herdado

Exemplo


Comida
private int CALORIAS_POR_GRAMA
private int gordura
protected int fatias
public Comida()
private int calorias()
public int caloriasPorFatia()
Pizza extends Comida
protected int fatias // herdado
public Pizza () // invoca super()
public int caloriasPorFatia() // herdado

Uma **subclasse pode sobrepor um dos seus métodos à definição de um método herdado da sua superclasse** (***method overriding***)

Ou seja, **uma subclasse pode redefinir um método que herdou**

O **novo método tem que ter a mesma assinatura que o herdado**, mas pode ter conteúdo diferente (caso se pretenda no entanto invocar o método da superclasse e não o implementado, recorre-se à invocação `super.metodo()`)

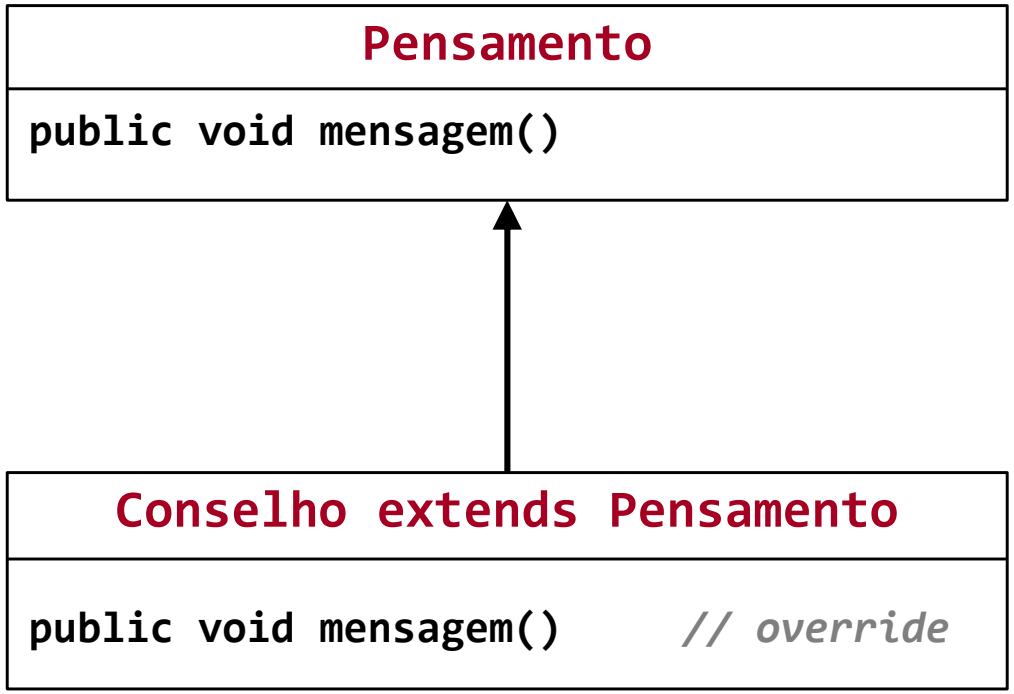
O tipo de objecto usado na chamada determina qual dos métodos é usado (se da superclasse, se da subclasse)

Exemplo


```
class Pensamento {
  public void mensagem() {
    System.out.println ("Aproxima-se a Queima das Fitas...");
  }
} // end of class Pensamento


class Conselho extends Pensamento {
  public void mensagem() { // overriding do método na subclasse
    System.out.println ("Atenção, Junho está à porta....");
  }
} // end of class Conselho


class Mensagens {
  public static void main (String[] args) {
    Pensamento pensamento = new Pensamento();
    Conselho   conselho   = new Conselho();
    pensamento.mensagem();        // método da classe Pensamento
    conselho.mensagem();          // método da classe Conselho
  }
} // end of class Mensagens
```


Pensamento
public void mensagem()
Conselho extends Pensamento
public void mensagem() // override

Invocação de métodos da superclasse

Vimos já que a **palavra reservada** `super` **referencia a superclasse** da classe onde é utilizada

Vimos também que esta palavra reservada pode ser utilizada para **invocar o construtor da superclasse** a partir do construtor da subclasse

Podemos agora generalizar e dizer que a referência `super` **pode ser usado para invocar qualquer método da superclasse**

Esta capacidade só é **útil quando ao método da superclasse que se pretende invocar foi sobreposto outro na subclasse**

Assim, para **explicitar que se quer executar o método da superclasse** podemos fazer `super.método(parametros);`

O estabelecimento de relações de **herança permite a criação de hierarquias de classes**, uma vez que nada impede uma **subclasse de ser, por sua vez, superclasse de uma nova classe**

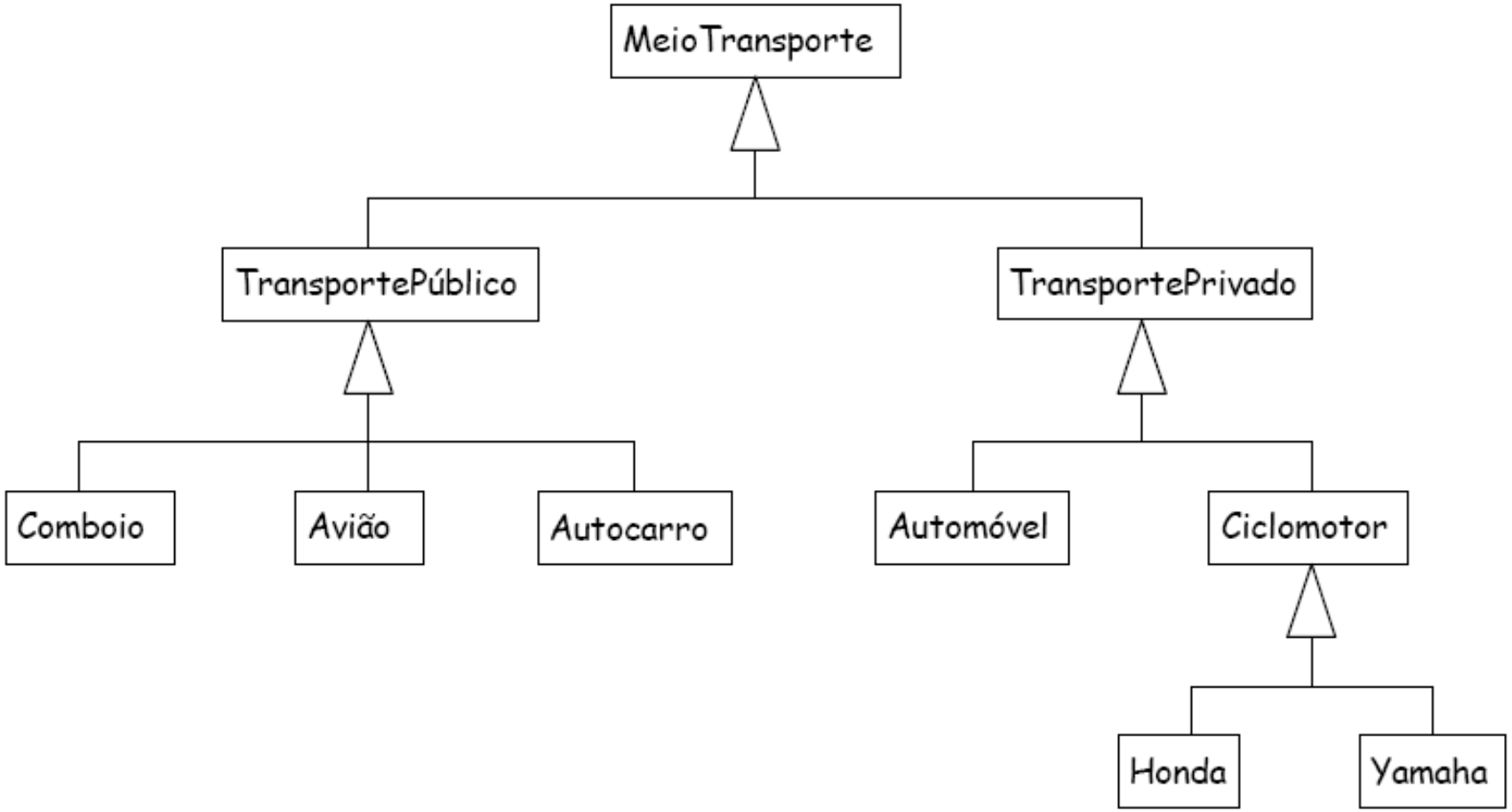
De igual modo nada impede que **várias classes derivem da mesma superclasse**

A definição da forma como a hierarquia de classes existente deve ser expandida para resolver um dado problema é uma questão central em programação orientada a objectos

Geralmente aceita-se que os **comportamentos comuns a um conjunto de classes devem ser colocados tão acima quanto possível na hierarquia**

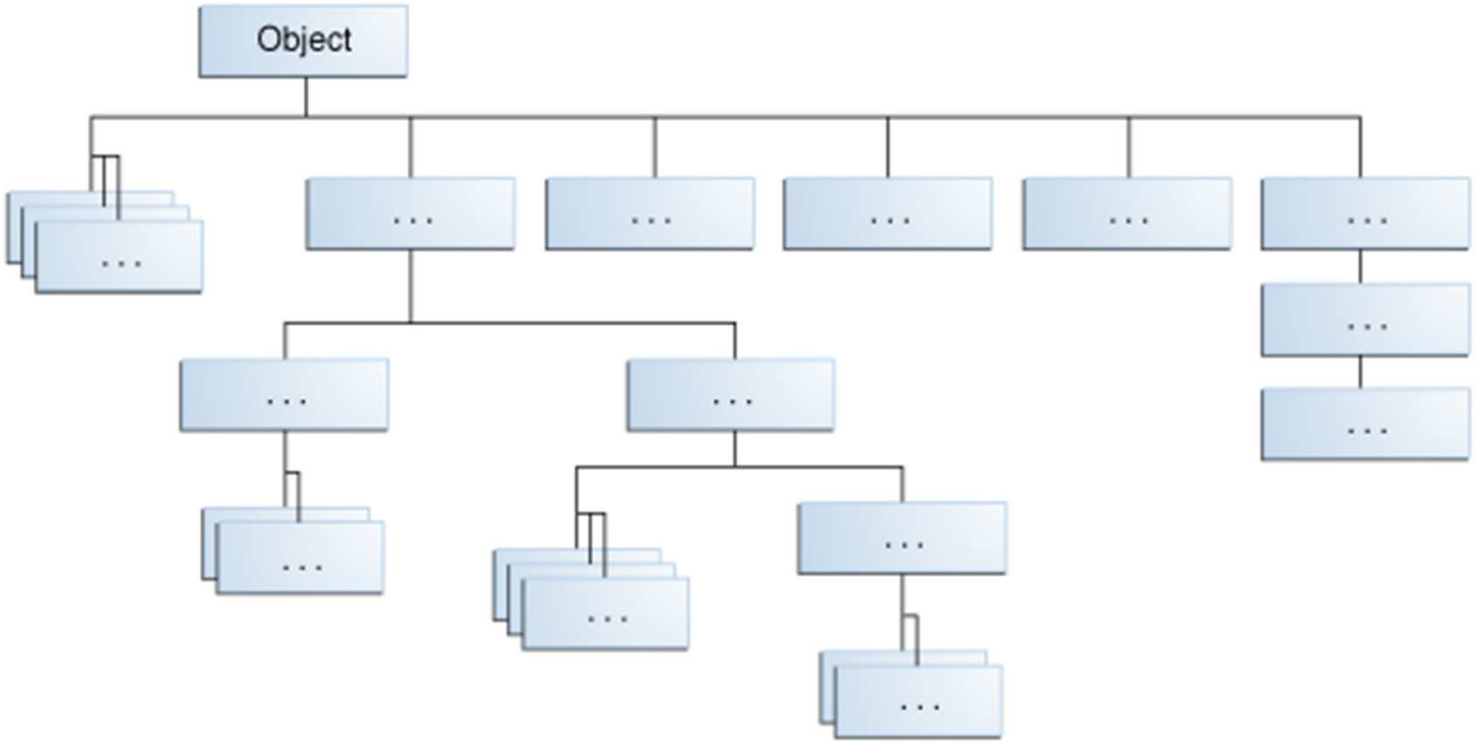
Exemplo de heranças

Programação Orientada a Objectos – Herança e Polimorfismo



Hierarquia de classes

Programação Orientada a Objectos – Herança e Polimorfismo



https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

Em Java a **raiz da hierarquia de classes** é ocupada pela classe `Object`

Isto significa que **todos os objectos são também derivados desta classe** (ainda que longinquamente em muitos casos)

Quando nada é dito (**não existe `extends` na definição da classe**) **o sistema assume que essa classe descende directamente de `Object`**

A **classe `Object` contém alguns métodos** (como **`toString()`** ou **`equals()`**) que são **herdados por todos os objectos**

Para que estes métodos tenham **comportamentos específicos** da nossa classe ela terá **que sobrepor a sua versão à herdada de `Object`**

Palavra reservada *final* aplicada a métodos

Tornar um **método `final` impede as descendentes da classe**, onde tal seja definido, **de fazer a sua própria versão do método (impede o *method overriding*)**

Dito de outro modo, faz com que essa seja a **"última" definição possível para o método**

Para definir um método **`final`**, basta colocar a *keyword* **`final`** no início da declaração

Palavra reservada *final* aplicada a classes

Uma classe marcada como `final` não pode ter descendentes

Desta forma **todos os métodos nela definidos são também**, de uma forma implícita, **do tipo `final`**

Para definir uma classe **`final`**, basta colocar a *keyword* **`final`** no início da declaração, tal como nos métodos

Criação de objectos a partir de classes derivadas

Vimos já que **uma referência pode referir-se a um objecto da sua classe**

Devido ao processo de herança **uma referência também se pode referir a um objecto da sua subclasse**, uma vez que essa possui todas as inicializações necessárias

Por exemplo, imaginando uma classe `Ferias`:

```
Ferias dia = new Ferias ();
```

Se existir uma classe `FeriasDeVerao` derivada a partir de `Ferias`

```
class FeriasDeVerao extends Ferias {
      // ...
}
```

é possível invocar:

```
Ferias dia = new FeriasDeVerao ();
```

Ou seja, **uma referência pode referir a sua própria classe ou qualquer outra classe relacionada com ela por relações de herança**

Como regra geral diz-se que se **pode usar um objecto de uma subclasse onde um objecto da sua superclasse possa ser utilizado**

Embora o **inverso também seja verdadeiro**, essa situação é menos comum e normalmente menos útil

Quando **obtemos um objecto de um Vector** este é devolvido pelo método `elementAt()` como sendo da classe `Object`, pelo que é **necessário proceder ao *cast*** para o tipo de dados pretendido

Porém, um **Vector pode armazenar objectos de diversos tipos**. Coloca-se a questão: como determinar o tipo correcto?

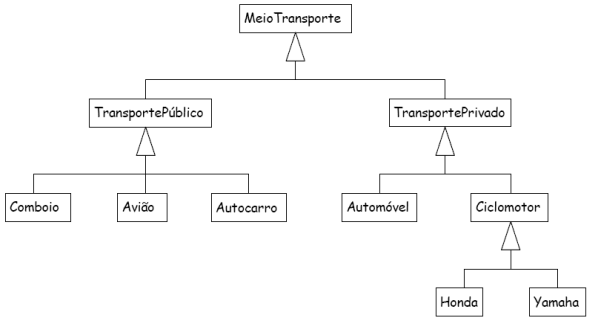
Existindo uma **hierarquia de classes**, podemos sempre **realizar o *cast* para o tipo da classe superior**

Palavra reservada *instanceof*

Se pretendermos definir o tipo do objecto especifico de cada subclasse podemos recorrer à *keyword* **`instanceof`** para testar o seu tipo

Exemplo: verifica se o objecto **`meioTransp`** **é do tipo (*é uma instância*) de `TranspPublico`**

```
MeioTransporte meioTransp = (meioTransp) lista.nextElement()
if (meioTransp instanceof TranspPublico) {
        (…)
}
```



ou em alternativa, **verificando o nome da classe**:

```
if (meioTransp.getClass().getName().equalsIgnoreCase("TranspPublico"){
        (…)
}
```

O método **`getClass().getName()`** **devolve o nome da classe e sua superclasse** (caso exista). Caso pretenda **só o nome da classe (sem incluir a superclasse)** deve usar o método **`getClass().getSimpleName()`**

Polimorfismo

Conceito de Polimorfismo

O **polimorfismo** ("*várias formas*") é um outro conceito central em programação orientada a objectos

Uma **referência polimórfica** é aquela que **se pode referir a um de vários possíveis métodos**

Considere a seguinte implementação:

```
class Ferias {
      public void celebrar() {
            System.out.println("Finalmente Férias!");
      }
} // end of class Ferias


class FeriasDeVerao extends Ferias {
      public void celebrar() {
            System.out.println("Férias de Verão!");
      }
} // end of class FeriasDeVerao
```

A instrução **`dia.celebrar();`** qual das duas versões invocará?

Se **`dia`** referir um objecto **`FeriasDeVerao`** será essa versão, mas se **`dia`** referir um objecto **`Ferias`** será a versão respectiva

Em geral, **é o tipo do objecto referido** (e não o tipo da referência) **que define qual a versão do método que é invocada**

Considerando de novo as classes **`Pensamento`** e **`Conselho`** já utilizadas anteriormente:

```
class Pensamento {
   public void mensagem() {
      System.out.println ("Vem aí Queima das Fitas!...");
   }
} // end of class Pensamento


class Conselho extends Pensamento {
   public void mensagem() {
      System.out.println ("Atenção, Junho está aí!...");
   }
} // end of class Conselho
```

```
class Mensagens {
      public static void main (String[] args) {
            Pensamento pensamento = new Pensamento();
            Conselho   conselho   = new Conselho();

            pensamento.mensagem();        // 1
            conselho.mensagem();          // 2
            pensamento = conselho;        // 3
            pensamento.mensagem();        // 4
      }
} // class Mensagens
```

Escreverá:

```
Vem aí Queima das Fitas!…          // Pensamento (passo 1)
Atenção, Junho está aí!…           // Conselho   (passo 2)
Atenção, Junho está aí!…           // Conselho   (passos 3 e 4)
```

É de notar que, caso a **invocação polimórfica de um método esteja dentro de um ciclo**, é possível que a mesma linha de código **invoque métodos diferentes** em momentos (iterações do ciclo) diferentes

Assim, as **referências polimórficas são definidas no momento da execução e não no momento da compilação**

Imaginemos que queríamos **implementar um programa de gestão de *stocks*** de uma loja de material fotográfico

Vamos imaginar que esta loja vende **lentes**, **sensores** e **câmaras fotográficas**

O nosso sistema necessita de guardar informação sobre os diversos itens:

– As **lentes** têm uma **distância focal** e podem ou não ter **zoom**

– Os **sensores** têm uma **sensibilidade** e uma **tamanho** (MP)

– As **câmaras** podem vir ou não com **lente**, têm uma dada **velocidade máxima** e uma dada **cor**

Todos os elementos necessitam ainda de:

- **Descrição** do item
- Um **código identificador**
- A **quantidade** em *stock*
- O **preço** do item

Precisamos de **criar classes que representem cada tipo de item**, guardando as informações respectivas nas suas variáveis de instância

Exemplo – Criação de classes


```
class Lente {
    private String  descricao;
    private int     numInvent;
    private int     quantidade;
    private int     preco;
    private boolean temZoom;
    private double  distFocal;

    public Lente (...) {…}; //Construtor
    ...
    //Métodos específicos de Lente
    public String getDescricao ()  {…};
    public int    getQuantidade () {…};
    public int    getPreco()       {…};
    ...
}
```

Exemplo – Criação de classes


```
class Sensor {
    private String  descricao;
    private int     numInvent;
    private int     quantidade;
    private int     preco;
    private int     sensibilidade;
    private int     tamanho;

    public Sensor(...) {…}; //Construtor
    ...
    //Métodos específicos de Sensor
    ...
    public String getDescricao ()   {…};
    public int    getQuantidade () {…};
    public int    getPreco()        {…};
}
```

Exemplo – Criação de classes


```
class Camara{
    private String  descricao;
    private int     numInvent;
    private int     quantidade;
    private int     preco;
    private boolean temLentes;
    private int     velMaxima;
    private String  cor;

    public Camara(...) {…}; //Construtor
    ...
    //Métodos específicos de Sensor
    ...
    public String getDescricao ()   {…};
    public int    getQuantidade () {…};
    public int    getPreco()        {…};
}
```

Como se pode ver há uma grande **sobreposição entre as classes**, já que **têm diversas variáveis e métodos comuns**

Se for necessário **guardar mais alguma informação comum aos três itens será necessário adicioná-la às três classes**

Cada uma das classes modela dois comportamentos relacionados, mas distintos:

- Um elemento de inventário
- Um elemento específico

A **herança pode ajudar a simplificar** esta situação

Podemos **criar uma superclasse que contenha os elementos comuns, derivando depois as classes específicas** a partir desta

Exemplo – superclasse


```
class ItemInventario {
        protected String  descricao;
        protected int     numInvent;
        protected int     quantidade;
        protected int     preco;
        ...
        public ItemInventario (...) {…}; //Construtor
        ...
        public String getDescricao ()   {…};
        public int    getNumInvent ()   {…};
        public int    getQuantidade () {…};
        public int    getPreco ()       {…};
        ...
}
```

Esta classe não “sabe” nada sobre as especificidades de cada item, mas é responsável pelo comportamento comum a todos os itens presentes no inventário da loja

Podemos agora definir as **sub classes** mais específicas:

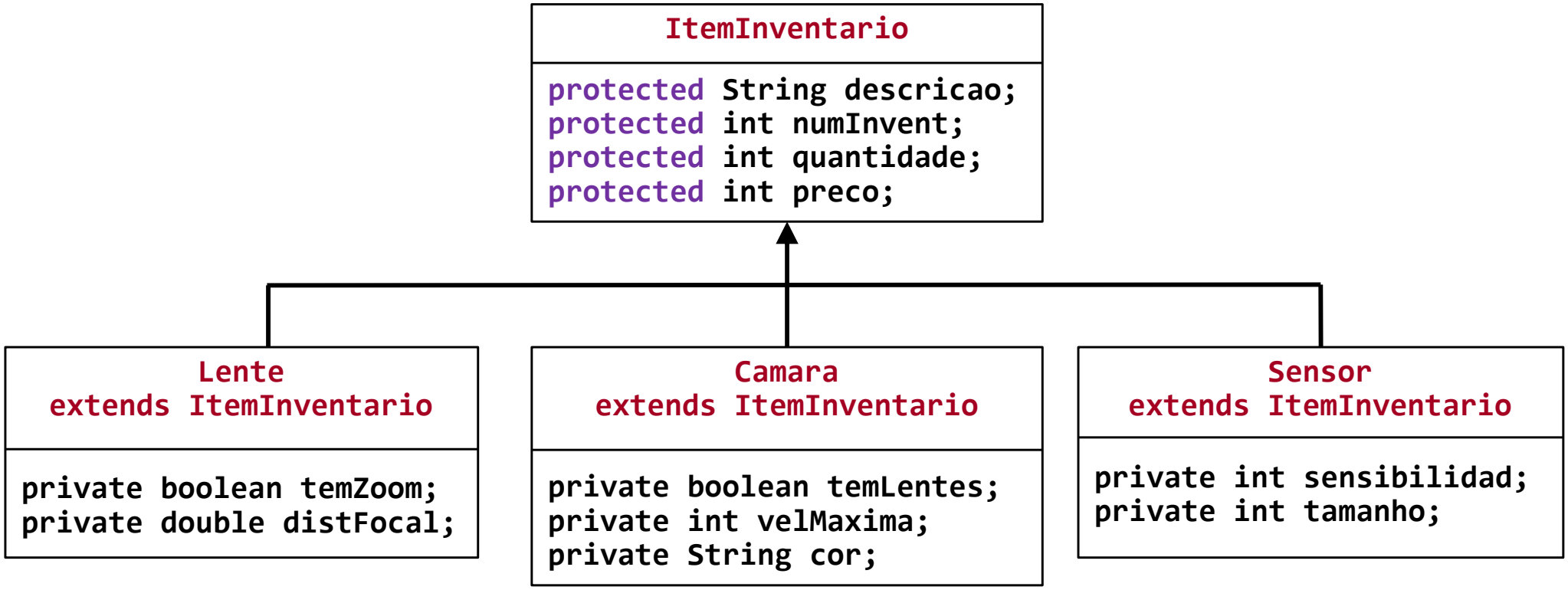
```
class Lente Extends ItemInventario{
      private boolean temZoom;
      private double  distFocal;
      ...
      public Lente (...) {…}; //Construtor
      ...
      //Métodos específicos de Lente
      ...
}
```

Exemplo – Sub Classes


```
class Sensor Extends ItemInventario{
     private int sensibilidade;
     private int tamanho;
     ...
     public Sensor (...) {…}; //Construtor
     ...
     //Métodos específicos de Filme
     ...
}

class Camara Extends ItemInventario{
     private boolean temLentes ;
     private int     velMaxima;
     private String  cor;
     ...
     public Camara (...) {…}; //Construtor
     ...
     //Métodos específicos de Camara
     ...
}
```

Exemplo – Estrutura de Classes



Estas três classes modelam as especificidades e herdam o comportamento comum a partir de `ItemInventario`

Esta opção é mais correcta do ponto de vista do *design* de classes e apresenta ainda algumas vantagens ao nível da manipulação dos objectos

Dado que **`Sensor`** ***é-um*** **`ItemInventario`** (tal como **`Lente`** e **`Camara`**), podemos invocar:

```
ItemInventario item;
item = new Lente  (…);        // ou
item = new Sensor (…);        // ou
item = new Camara (…);
```

Um resultado interessante deste tipo de organização pode ser visto no código seguinte:

```
ItemInventario[] invent = new ItemInventario[5];
invent [0] = new Lente  (…);
invent [1] = new Sensor (…);
invent [2] = new Camara (…);
invent [3] = new Sensor (…);
invent [4] = new Camara (…);
…
// percorrendo toda a estrutura
System.out.println ("Listagem de material em stock:");
for (int i = 0; i < inventario.length; i++)
      System.out.println (  invent[i].getDescricao() +":"+
                            invent[i].getQuantidade() );
```

Referências

"***Programação Orientada a Objectos***"
António José Mendes
Departamento de Engenharia Informática, Universidade de Coimbra

"***Java in a Nutshell***", 4ª Edição, Capítulo 3 "*Object-Oriented Programming in Java*"
David Flanagan
O'Reilly, ISBN: 0596002831

"***Thinking in Java***, ", 4ª Edição, Capítulo 6 "*Reusing Classes*"; Capítulo 7 "*Polymorphism*"
Bruce Eckel
Prentice Hall, ISBN: 0131872486

"***The Java Tutorial – Learning the Java Language: Interfaces and Inheritance***"
Java Sun Microsystems
http://java.sun.com/docs/books/tutorial/java/landl/index.html

Bibliografia complementar

Programação Orientada a Objectos – Herança e Polimorfismo

"***Fundamentos de Programação em Java 2***", Capítulo 11 "*Noções Avançadas sobre Classes e Objectos*"
António José Mendes, Maria José Marcelino
FCA, ISBN: 9727224237

"***Java 5 e Programação por Objectos***", Capítulo 5 "*Hierarquia de Classes e Herança*"
F. Mário Martins
FCA, ISBN: 9727225489