Parte 4.3 – Herança e Polimorfismo

Fernando Barros, Karima Castro, Luís Cordeiro,
Marília Curado, Nuno Pimenta

Os conteúdos desta apresentação baseiam-se nos materiais produzidos por António José Mendes para a unidade curricular de Programação Orientada a Objetos.

Quaisquer erros introduzidos são da inteira responsabilidade dos autores.

HERANÇA

Herança

- Um dos conceitos fundamentais em programação orientada aos objetos é a **herança**
- Quando utilizada corretamente 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** ou **classe derivada**
- Também se pode usar o conceito pai-filho para descrever esta relação

Herança

- Uma **subclasse herda características da sua superclasse** (herda as variáveis de instância e os métodos nela definidos)
- 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 represente um livro
- Em Java a relação de herança é estabelecida usando a palavra reservada **extends**:

```
class Dicionario extends Livro {...}
```

Herança

- Exemplo:

```
//  Classe Livro será uma superclasse
class Livro {
  private int paginas = 700;
  public int paginas() {
      return paginas;
  }
  public void pageMessage() {
    System.out.println("Número de páginas: " + paginas);
  }
}
```

Herança

- Exemplo (cont.):

```
//  Classe Dicionario herda os métodos e os atributos da
//  classe Livro
class Dicionario extends Livro {
  private int entradas = 5500;
  public void entradasMessage() {
     System.out.println("Número entradas: " + entradas);
     //Herda método paginas() da classe Livro
     System.out.println("Média por página: " +
                                    entradas/paginas());
   }
}
```

Herança

- Exemplo (cont.):

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

- Não foi criado explicitamente um objeto da classe Livro
- No entanto, o objeto da classe Dicionario herda o(s) método(s) (**`pageMessage()`**) e atributo(s) (**`paginas`**) da classe Livro, já que esta é a sua superclasse

Modificadores de Visibilidade/Acessibilidade

- 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
- Mas isto viola os princípios do **encapsulamento**…

Modificadores de Visibilidade/Acessibilidade

- Podemos usar o modificador **protected** para indicar que os métodos e variáveis podem ser acedidos em todas as classes da mesma Package e em subclasses
- Apesar de terem visibilidade **public**, os construtores não são herdados

Modificadores de Visibilidade/Acessibilidade

- Por omissão (quando não é utilizado qualquer modificador), um membro de uma classe (atributo ou método) é visível, e está acessível, em todas as classes da mesma package
- A visibilidade / acessibilidade pode ser alterada, utilizando um dos seguintes modificadores: **private**, **protected** e **public**

Modificadores de Visibilidade/Acessibilidade

- **private**
  - Acessível apenas na própria classe.
- **Sem modificador (por omissão)**
  - Acessível em todas as classes da mesma Package.
- **protected**
  - Acessível em todas as classes da mesma Package e em subclasses (mesmo que pertençam a outras packages).
- **public**
  - Acessível em todas as classes de qualquer package.

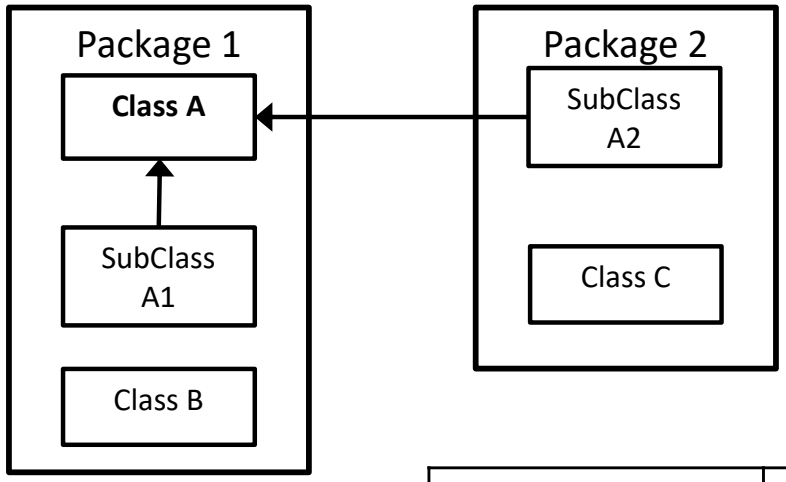
Modificadores de Visibilidade/Acessibilidade

Modificador	**Classe**	**Package**	**Subclasse** (outras packages)	**Tudo**
private	Visível	---	---	---
Omissão	Visível	Visível	---	---
protected	Visível	Visível	Visível	---
public	Visível	Visível	Visível	Visível

Fonte: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Modificadores de Visibilidade/Acessibilidade

- Exemplo:



Modificador na Class A	A	A1	B	A2	C
private	Visível	---	---	---	---
sem modificador	Visível	Visível	Visível	---	---
protected	Visível	Visível	Visível	Visível	---
public	Visível	Visível	Visível	Visível	Visível

Fonte: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Keyword **super**

- Por vezes é necessário invocar o construtor da **superclasse** a partir da **subclasse**, por forma a inicializar devidamente as variáveis herdadas (como foi referido, os construtores, apesar de serem públicos, não são herdados)
- O termo **super** pode ser usado para referenciar a **superclasse** e é frequentemente usada para invocar o seu construtor, mas não só
- A palavra reservada **super** está disponível em todos os métodos não static de uma classe descendente
- É utilizada no acesso às variáveis e métodos membro da superclasse e funciona como uma referência para o objeto corrente enquanto instância da sua superclasse

```
super.metodoDaSuperclasse()
```

Keyword **super** – Porquê?

- Para invocar construtores da superclasse, porque não são herdados;
- Para invocar métodos da superclasse, porque a classe derivada pode ter métodos com a mesma assinatura e é necessário distingui-los (mais à frente);
- Para invocar variáveis da superclasse, porque a classe derivada, ou o método em execução, podem ter variáveis com o mesmo nome.

Keyword **super** – utilização nos construtores

- Tal como a palavra reservada **this**, a palavra reservada **super** assume um significado especial nos construtores:
  - **super()** serve para invocar explicitamente o construtor por omissão (sem argumentos) da superclasse. Da mesma forma podem invocar-se os restantes construtores, passando os argumentos necessários.
  - Caso se pretenda usar a invocação explícita de um dos construtores da superclasse tem que ser a primeira instrução do construtor da classe descendente.
  - O Java invoca automaticamente o construtor por omissão da superclasse, caso não se invoque explicitamente nenhum dos construtores disponíveis.

Keyword **super** – Exemplo

```
class Livro {
    private String nome;
    private int paginas = 700;
    public String nome() {
        return nome;
    }
    public int paginas() {
        return paginas;
    }
    public Livro(String nome, int paginas) {
        this.nome = nome;
        this.paginas = paginas;
    }
```

Keyword **super** – Exemplo (cont.)

```
    public void pageMessage() {
        System.out.println("Número de páginas: " + paginas);
    }
    public String toString() {
        return nome + " " + paginas;
    }
} //class Livro
```

Keyword **super** – Exemplo (cont.)

```
class Dicionario extends Livro {
    private int entradas;
    public Dicionario(String nome, int paginas, int entradas) {
        super(nome, paginas);
        this.entradas = entradas;
    }
    public void entradasMessage() {
        System.out.println("Número entradas: " + entradas);
        System.out.println("Média por página: " +
                entradas / paginas());
    }
    public String toString() {
        return super.toString() + " " + entradas;
    }
}
```

Keyword **super** – Exemplo (cont.)

```
class Palavras {
  public static void main(String[] args) {
      Dicionario larousse = new Dicionario("Latim-
Portugês", 600, 8000);
      larousse.pageMessage();
      larousse.entradasMessage();
      System.out.println(d);
  }
}
```

POLIMORFISMO

Polimorfismo

- A Herança estabelece uma relação entre uma superclasse e a(s) sua(s) descendente(s)
- Uma classe descendente é uma das variantes possíveis para os membros da superclasse
- As classes descendentes têm as suas próprias características, mas partilham características semelhantes, herdadas da superclasse
- As classes descendentes herdam comportamentos, que vão ser semelhantes entre si, podendo não ser exatamente iguais
- Os comportamentos (métodos) herdados podem ser adequados a cada classe que os herda

Polimorfismo

- O **polimorfismo** é um outro conceito central em programação orientada a objetos
- **Polimorfismo** quer dizer várias formas. Quando aplicado à orientação a objetos quer dizer que um mesmo método pode ter muitas formas
- Sendo um método reconhecido pelo seu nome e pelo tipo dos seus argumentos (a sua assinatura), quer isto dizer que à mesma assinatura podem corresponder várias implementações, em diferentes classes descendentes, e dependendo do objeto que está a executar o método

Polimorfismo

- Uma **referência polimórfica** é aquela que se pode referir a um de vários possíveis métodos
  - Imaginemos que a classe **Ferias** tem um método chamado **celebrar** e a classe **Verao** (descendente de Ferias) lhe sobrepõe um outro método (com o mesmo nome e parâmetros, claro)
  - A instrução `dia.celebrar();` qual das duas versões invocará?
  - Se **dia** referenciar um objeto da classe **Ferias** será a versão dessa classe, mas se dia referenciar um objeto da classe **Verao** será a versão respetiva (sobreposta à herdada)
- Em geral, é o tipo do objeto (e não o tipo da referência / variável) que define qual o método que é invocado

Polimorfismo

- Considere as classes **Pensamento** e **Conselho**:

```
class Pensamento {
  public void mensagem() {
       System.out.println ("Aproxima-se a Latada...");
  }
}  // class Pensamento
```

Polimorfismo

```
class Conselho extends Pensamento {
   public void mensagem() {
     System.out.println("--------");
     super.mensagem();
     System.out.println("Mas atenção, há trabalhos para
entregar....");
  }
  public void mensagem2() {…}
}  // class Conselho
```

Polimorfismo

```
class Mensagens {
    public static void main (String[] args) {
            Pensamento pensa = new Pensamento();
            Conselho exame = new Conselho();
            pensa.mensagem();
            exame.mensagem();
            pensa = exame;
            pensa.mensagem();
        }
}  // class Mensagens
```

- Escreverá:

```
Aproxima-se a Latada…
--------
Aproxima-se a Latada…
Mas atenção, há trabalhos para entregar ....
--------
Aproxima-se a Latada…
Mas atenção, há trabalhos para entregar ....
```

Polimorfismo

- 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

Polimorfismo

```
class MeioTransporte {
  private String condutor = "Zé";
  public void movimenta() {
      System.out.println("MeioTransporte em movimento!!");
  }
  public String condutor() {
      return condutor;
  }
}
```

Polimorfismo

```
class TransportePublico extends MeioTransporte {
  private String[] passageiros = {"Xico", "Manel", "João"};
  public void movimenta() {//herdado
      System.out.println("TransportePublico em movimento!!");
  }
  public void picaBilhete() {
      System.out.println("PicaBilhete TransportePublico!!");
  }
  public void imprimePassageiros() {
      System.out.print("passageiros: ");
      for (String p: passageiros)
          System.out.print(p +  ", ");
      System.out.println();
  }
}
```

Polimorfismo

```
class Comboio extends TransportePublico {
  private int numCarruagens = 5;
  //herdado
  public void movimenta() {
      System.out.println("Comboio em movimento!!");
  }
  public void picaBilhete() {
      System.out.println("PicaBilhete de Comboio!!");
  }
  public void apita() {
      System.out.println("tutuuu!!");
  }
  public int numCarruagens() {
      return numCarruagens;
  }
}
```

Polimorfismo

- Se executarmos o código:

```
System.out.println("---MeioTransporte---:");
MeioTransporte m = new MeioTransporte();
System.out.println("condutor: " + m.condutor());
m.movimenta();
```

- Obtemos:

```
---MeioTransporte:---
Condutor: Zé
MeioTransporte em movimento!!
```

Polimorfismo

- Se executarmos o código:

```
System.out.println("\n\n---TransportePublico:---");
TransportePublico t = new TransportePublico();
System.out.println("condutor: " + t.condutor());
t.movimenta();
t.imprimePassageiros();
t.picaBilhete();
```

- Obtemos:

```
---TransportePublico:---
condutor: Zé
TransportePublico em movimento!!
passageiros: Xico, Manel, João,
PicaBilhete TransportePublico!!
```

Polimorfismo

- Se executarmos o código:

```
System.out.println("\n\n---Comboio:---");
Comboio c = new Comboio();
System.out.println("condutor: " + c.condutor());
c.movimenta();
c.imprimePassageiros();
c.picaBilhete();
c.apita();
System.out.println("número de carruagens: " +
    c.numCarruagens());
```

Polimorfismo

- Obtemos:

```
---Comboio:---
Condutor: Zé
Comboio em movimento!!
passageiros: Xico, Manel, João,
PicaBilhete de Comboio!!
tutuuu!!
número de carruagens: 5
```

Upcasting e *Downcasting*

- **Upcasting:** um membro de uma classe descendente pode ser sempre visto como um membro da sua superclasse

```
class Instrumento {
  static void afina (Instrumento i) { //método da classe
      System.out.println("A afinar um " + i);
  }
}
class Sopro extends Instrumento { // Igual a Instrumento
}
class Musica {
  public static void main (String[] args) {
      Sopro flauta = new Sopro();
      Instrumento.afina(flauta); //upcasting:flauta é Sopro
  }
}
```

Upcasting e *Downcasting*

- **Downcasting:** transformar um *handle* de uma classe mais genérica (ascendente) numa classe mais específica (descendente), APENAS SE o *handle* apontar na realidade para um objeto do tipo mais específico

```
Instrumento f = new Sopro();
Sopro s = (Sopro)f; //funciona
Instrumento g = new Instrumento();
Sopro t = (Sopro)g; //vai dar origem a uma exceção
```

*Keyword **final***

- Um método pode ser definido como **final**, o que impede que as classes descendentes da classe onde foi definido possam implementar a sua própria versão do método
- Uma classe também pode ser definida como **final**, o que a impede de ter classes descendentes. Implicitamente, todos os seus métodos são, igualmente, final
- Para definir uma classe ou método como **final**, basta colocar a palavra reservada **final** no início da declaração

*Keyword **final***

- **Exemplo:**

```
class Aviao extends TransportePublico {
  public final void descola() {
      System.out.println("Avião a descolar!!");
  }
  public final void aterra() {
      System.out.println("Avião a aterrar!!");
  }
}

final class Autocarro extends TransportePublico {
  private int minutosAtraso;
}
```

  - Classes descendentes de Aviao não podem implementar novas versões dos métodos `void descola()` e `void aterra()`
  - Classe Autocarro não pode ter classes descendentes