FICHA 4.2 – HERANÇA E POLIMORFISMO

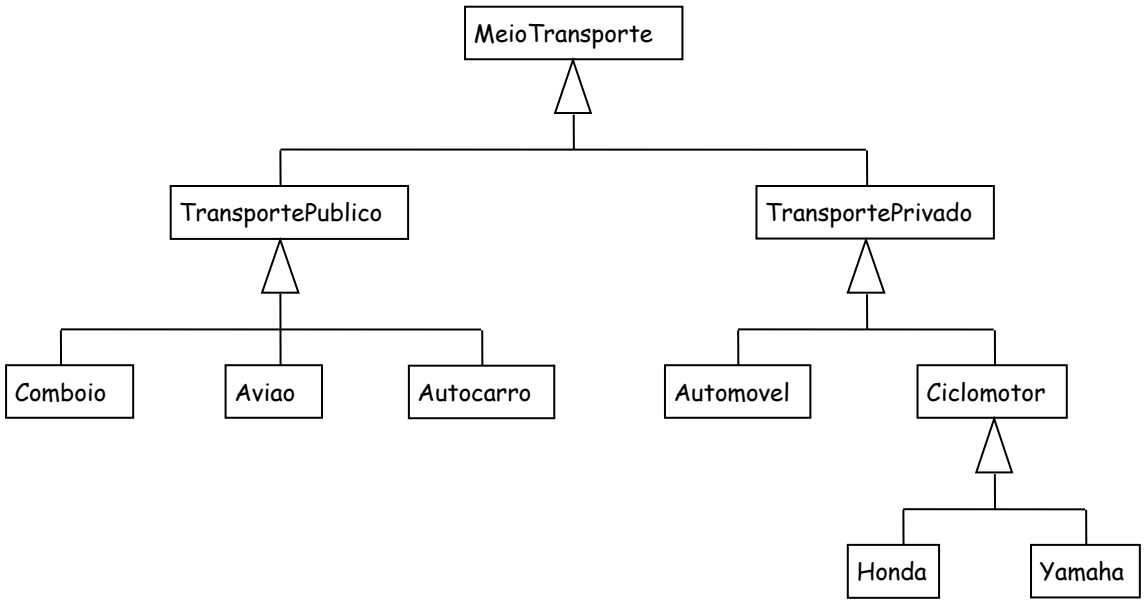
Um programa em Java é geralmente constituído por vários objetos de várias classes organizadas hierarquicamente.

Hereditariedade

Se olharmos para o mundo real, os objetos como as pessoas, os animais, as bicicletas, etc., são muitas vezes "agrupados" de acordo com determinadas características e comportamentos. Por exemplo, pessoas e animais são objetos animados, em oposição às bicicletas que são objetos inanimados.
Existem grupos de objetos que além de possuírem as suas características específicas, possuem características comuns a outros grupos. Os humanos, por exemplo, pertencem aos mamíferos, pois exprimem características e comportamentos próprios dos mamíferos, apesar de possuírem as suas características próprias. Dito de outro modo, um humano é um ser humano, mas também é um mamífero.
Em linguagens orientadas a objetos a técnica que nos permite a partir de uma classe utilizá-la na construção de novas classes é denominada hereditariedade, uma vez que a organização das classes é feita hierarquicamente.

Super classe e classes derivadas

A hereditariedade é a capacidade que uma classe, chamada **classe derivada** ou **classe descendente** ou **subclasse**, tem de adquirir as características e comportamentos de outra classe, chamada **classe base**, **superclasse** ou **classe antecessora**.
O modo mais correto de usar a hereditariedade é criar uma hierarquia de classes em árvore, que coloca o conceito mais genérico na raiz e os conceitos mais específicos nas folhas.
Na figura mostra-se um exemplo de uma estrutura hierárquica, que descreve vários tipos de transporte:



É importante compreender a relação entre as várias classes:

- Começando das folhas para a raiz, cada subclasse também **é um** dos elementos da superclasse (do mesmo modo que um humano também **é um** dos elementos dos mamíferos);
- um Autocarro **é um** TransportePublico, que, por sua vez, **é um** meio de Transporte;
- um Autocarro também **é um** meio de Transporte.

Implementação em Java

A linguagem Java utiliza a palavra **extends** para exprimir a noção de herança. Assim, se pretender criar uma classe B que seja derivada de uma classe A, pode fazer:

B extends A

Voltando ao exemplo dos meios de transporte, uma implementação das classes MeioTransporte, TransportePublico e Comboio poderia ser:

```
class MeioTransporte {
       protected String condutor = "Zé";
       void movimenta() {
```

```
            System.out.println("Em movimento!!");
      }
}

class TransportePublico extends MeioTransporte {
      protected String[] passageiros = {"Xico, Manel, João"};
      void picaBilhete() {
            System.out.println("\nA picar um bilhete!!");
      }
}

class Comboio extends TransportePublico {
      protected  int numCarruagens = 5;
      void apita() {
            System.out.println("tutuuu!!");
      }
}
```

Herança de atributos e métodos

O código acima implementa as classes MeioTransporte, TransportePublico e Comboio. Relativamente aos atributos e métodos que são herdados refere-se:

- Um TransportePublico é um MeioTransporte; por esta razão herda deste um condutor e o método movimenta();
- um Comboio é um TransportePublico, herdando deste a capacidade de transportar passageiros e o método picaBilhete(),
- mas é também um MeioTransporte, pelo que herda ainda um condutor e o método movimenta().

O código seguinte permite testar o mecanismo de herança:

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

System.out.println("\n\nTransportePublico:");
TransportePublico t = new TransportePublico();
System.out.println("condutor " + t.condutor);
t.movimenta();
//mas também:
System.out.println("passageiros:");
for (int i = 0; i < t.passageiros.length; i++)
      System.out.print(t.passageiros[i] +  "     ");
t.picaBilhete();

System.out.println("\n\nComboio:");
Comboio c = new Comboio();
System.out.println("condutor " + c.condutor);
c.movimenta();
```

```
System.out.println("passageiros:");
for (int i = 0; i < c.passageiros.length; i++)
       System.out.print(c.passageiros[i] +  "     ");
c.picaBilhete();
//.. e ainda:
c.apita();
System.out.println("num carruagens " + c.numCarruagens);
// (…)
```

A execução do código acima tem o seguinte resultado:

```
MeioTransporte:
condutor Zé
Em movimento!!

TransportePublico:
condutor Zé
Em movimento!!
passageiros:
Xico, Manel, João
A picar um bilhete!!

Comboio:
condutor Zé
Em movimento!!
passageiros:
Xico, Manel, João
A picar um bilhete!!
tutuuu!!
num de carruagens 5
```

A palavra reservada super

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 e funciona como uma referência para o objeto corrente enquanto instância da sua superclasse.

A chamada **super.método** invoca sempre a implementação do método da super classe e não a versão do método implementada pela classe corrente.
Por exemplo, acrescentado a cada uma das classes TransportePublico e Comboio um método específico void movimenta() e o método void testaSuper() à classe Comboio:

```
class TransportePublico extends MeioTransporte {
       //(…)
       void movimenta() {
              System.out.println("TransportePublico em movimento!!");
       }
       //(…)
```

```
}

class Comboio extends TransportePublico {
	//(…)
	void movimenta() {
		System.out.println("Comboio em movimento!!");
	}

	void testaSuper() {
		System.out.println("vou chamar o movimenta(): ");
		movimenta();
		System.out.println("vou chamar o super.movimenta(): ");
		super.movimenta();
	}
	//(…)
}
```

e agora executando o seguinte código:

```
		Comboio c = new Comboio();
		c.testaSuper();
```

Obtemos o seguinte resultado:

```
vou chamar o movimenta():
Comboio em movimento!!
vou chamar o super.movimenta():
TransportePublico em movimento!!
```

A palavra reservada super nos construtores

Tal como a palavra reservada **this**, a palavra reservada **super** assume um significado especial nos construtores:

- assim, **super()** serve para invocar explicitamente o construtor por omissão da super classe. Da mesma forma podem invocar-se os restantes construtores, passando os argumentos necessários.
- Caso se queira 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.

Vamos adicionar os seguintes construtores a TransportePublico e a Comboio:

```
class TransportePublico extends MeioTransporte {
	protected  String[] passageiros = {"Xico, Manel, João"};
```

```
	TransportePublico() {
		passageiros = new String[2];
		passageiros[0] = "Josefa";
		passageiros[1] = "Maria";
	}

	TransportePublico(String[] p) {
		passageiros = new String[p.length];
		passageiros = p;
	}
	// (…)
}

class Comboio extends TransportePublico {
	protected  int numCarruagens = 5;

	Comboio() {}
	Comboio(String[] passageiros) {
		super(passageiros);
	}
	// (…)
}
```

Agora vamos executar o seguinte código:

```
		// (…)
		Comboio c = new Comboio();
		System.out.println("passageiros do Comboio():");

		for (int i = 0; i < c.passageiros.length; i++)
			System.out.print(c.passageiros[i] +  "     ");
		System.out.println();

		String[] p = {"Carlo", "Felizberto", "Matias"};;
		Comboio b = new Comboio(p);
		System.out.println("\npassageiros do Comboio(p):");

		for (int i = 0; i < b.passageiros.length; i++)
			System.out.print(b.passageiros[i] +  "     ");
		System.out.println();
		// (…)
```

Obtemos então como resultado:

```
passageiros do Comboio():
Josefa Maria

passageiros do Comboio(p):
Carlo Felizberto Matias
```

A palavra reservada final aplicada a classes e métodos

Definir um método como **final** impede as classes descendentes da classe onde foi definido de fazer a sua própria versão do método. Dito de outro modo, faz com que essa seja a última definição possível para o método.
Uma classe marcada como **final** não pode ter descendentes. O que faz com que todos métodos nela definidos sejam também implicitamente **final**.
Para definir uma classe ou método como *final*, basta colocar a palavra reservada *final* no início da declaração.
Por exemplo:

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

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

tem como consequência:

- que os métodos *descola()* e *aterra()* são considerados final. Assim as classes descendentes da classe Avião podem, na melhor das hipóteses, usar a definição que já existe; podem, no entanto, alterar as definições dos restantes métodos (herdados da classe TransportePublico).
- A classe Autocarro, porque foi tornada final, não pode agora ter descendentes.

EXERCÍCIOS

1. Pessoas do DEI

A Comunidade do DEI é composta por pessoas com diferentes funções. As mais comuns são os Alunos e os Docentes. Qualquer destas pessoas tem um atributo em comum: um nome. Todos têm também pelo menos um comportamento comum: comunicar. Os alunos têm um número de aluno e os docentes têm um número mecanográfico. As missões/comportamentos do aluno e do docente também são

diferentes: o aluno aprende e o docente ensina (embora também aprenda).

Construa uma estrutura de classes que reflita estas propriedades (atributos e comportamentos), de acordo com o seguinte:

- Existem 3 classes: Pessoa, Aluno e Docente;
- A classe Pessoa tem um atributo comum a docentes e alunos: nome.
- Também na classe Pessoa, existe um comportamento/método comum a docentes e alunos, String comunica() que, uma vez chamado, devolve o seguinte texto: “[Nome da pessoa] a comunicar.”;
- As classes Aluno e Docente têm um método String missao() que, uma vez chamado, tem o seguinte comportamento:
    - Na classe Aluno, devolve o seguinte texto: “aprender”;
    - Na classe Docente, devolve o seguinte texto: “ensinar”.
- Todas as classes têm um método toString() que, recorrendo ao método missao(), devolve o seguinte texto:

  O [aluno/docente] com o número de aluno/mecanográfico [XXXX] tem uma missão de [aprender/ensinar]!

Para testar o seu programa, crie dois objetos de cada tipo (aluno e docente) e ”escreva-os” no ecrã, recorrendo ao método toString(). Chame, ainda, o método comunica() de cada objeto e imprima o texto devolvido no ecrã.

2. Coffee Shop

Um amigo seu abriu recentemente uma *Coffee Shop* em Coimbra e está agora contratar pessoas para atender ao balcão. Como se trata de uma *Coffee Shop* moderna, não vão servir apenas o tradicional café, leite e galão, mas também uma série de variantes tais como o *cappuccino* e o *latte macchiato*. Como nem toda a gente sabe fazer estas bebidas, o seu amigo pediu-lhe para você desenvolver um programa que ajude os novos empregados de balcão a preparar as bebidas.

Todas as bebidas têm um **nome**, uma **temperatura** (graus Celsius) e uma **quantidade** (ml). Há três tipos principais de bebidas: café, leite, e café com leite. À

partida, o café é servido em doses de 100ml a 60 graus, mas o café ainda pode ser de um subtipo mais específico — o café expresso — que é servido a 70 graus e pode ser "curto" (25ml), "cheio" (50ml) ou "normal" (35ml).

O leite é servido em doses de 150ml a 55 graus, e há ainda um tipo especial de leite — a espuma de leite — que é apenas usada na preparação de outras bebidas, mas que é sempre servida em quantidades de 80ml a 60 graus.

Há várias bebidas consideradas como café com leite e todas elas são compostas de algum tipo de café e de algum tipo de leite. Um exemplo é o galão, e outro é o *cappuccino* que é composto por um café expresso cheio e espuma de leite. Finalmente, há o *latte macchiato* que é feito como o *cappuccino* mas ainda leva mais leite normal para além da espuma de leite.

a) programe as classes que desenhou, incluindo o(s) método(s) toString que permitam obter uma descrição de cada tipo de bebida

b) Desenvolva um programa para testar as suas classes, criando as instâncias que fizerem sentido, e mostrando no ecrã as suas descrições textuais.

3. Banco

Imagine que para a gestão informatizada de um banco está devidamente armazenada informação sobre as contas dos seus clientes. Para cada cliente sabe-se o nome, o número e o tipo de conta (à ordem ou a prazo) e o respetivo saldo. Faça um programa que permita fazer repetidamente as seguintes operações:

a) Levantamentos de uma conta - só se houver saldo;

b) A listagem dos depósitos numa conta - sempre possíveis;

c) Consulta do saldo de uma conta - sempre possível;

d) A listagem dos clientes que têm contas com um saldo superior a x;

e) Quantos dias faltam para vencer o juro de uma conta a prazo (anual).

4. Concurso

Uma cadeia de supermercados decidiu organizar um concurso para comemorar os seus 15 anos de existência. O concurso destina-se a funcionários e clientes e é semelhante ao conhecido Totoloto sem recurso ao número suplementar: uma chave é constituída por 4 números de um universo de 10. Cada funcionário pode fazer 5 apostas, enquanto que cada cliente tem direito a 3 apostas. Um jogador (funcionário ou cliente) é identificado pelo respetivo nome e idade. Enquanto que para um cliente é necessário saber o seu telefone, um funcionário é ainda identificado pelo seu número interno na empresa. Os dados relativos a cada jogador (funcionário ou cliente), bem como as apostas/chaves de todos os jogadores devem ser armazenados de forma adequada. Desenvolva as classes necessárias para utilizar num programa que implemente esta situação.

5. Jantar/convívio da comunidade DElana

O Diretor do DEI pretende promover um jantar/convívio da comunidade atual do DEI, composta por funcionários, docentes e alunos, além dos respectivos acompanhantes.

Desenvolva uma aplicação que permita gerir as inscrições para este evento e, adicionalmente, recolher alguma informação sobre os inscritos, conforme os requisitos seguintes:

- Qualquer pessoa da comunidade deve indicar o seu nome e número de acompanhantes;
- Os funcionários devem indicar a sua categoria;
- Os docentes devem indicar o seu grau;
- Os alunos devem indicar o seu ano de entrada no curso que frequentam atualmente;
- Os alunos de doutoramento devem, adicionalmente, indicar o tema da sua tese;
- Os alunos de mestrado devem indicar a sua média de entrada neste curso;
- Os alunos de licenciatura devem indicar o número de ECTS já obtidos;

- Os preços praticados serão os representados na tabela seguinte:

Pessoa	**Preço**	**Preço Acompanhante**
Funcionário	10€	12€
Docente	10€	12€
Aluno PHD	8€	10€
Aluno MSC	6€	8€
Aluno BSC	6€	8€

Além de aceitar as inscrições, o programa deverá imprimir uma listagem de todas as pessoas da comunidade inscritas, uma por linha. A informação a apresentar sobre cada pessoa inscrita será toda a disponível (de acordo com os requisitos), número de acompanhantes e valor total a pagar.

Deverá, ainda, apresentar uma listagem dos valores totais recebidos referentes a funcionários, docentes e cada tipo de aluno. Estes valores devem incluir os valores dos respectivos acompanhantes.

6. Figuras geométricas

Desenvolva um programa em Java para lidar com figuras geométricas. O programa deve ter as seguintes características, utilizando herança sempre que for útil.

a) Defina uma classe com os dados relativos a figura geométrica (FiguraGeométrica). Deverá conter o nome da figura geométrica, enquanto que as restantes propriedades dependerão do tipo de figura geométrica. Todas as figuras geométricas deverão ter um método propriedades() e um método toString(). O método propriedades() devolverá as propriedades do objeto. No caso de uma figura geométrica FiguraGeometrica, este método retornará “Nome=X”, onde X é o valor da propriedade nome. O método toString() retornará uma String com as propriedades da figura. Por fim, deverão implementar o método area(), que devolverá a área da figura geométrica.

b) Defina uma classe com os dados relativos a **Ponto, SegmentoRecta, Circulo**, **Quadrado** e **Rectângulo**. Defina cada uma dessas como figuras geométricas e inclua as variáveis necessárias nas respectivas classes.
c) Implemente os métodos para cada uma das classes criadas. Por exemplo, se o objeto for um Circulo localizado no ponto (1,1,) e de raio 2, o método propriedades() e o método toString() deverão devolver a String "nome=X, raio=2, centro=(1,1)", e o método área() deverá calcular a área do Circulo.
d) Defina uma classe desenho geométrico (DesenhoGeométrico), o qual será também uma figura geométrica. Essa classe deverá ter um construtor que indique quantas figuras geométricas terá, e um método adiciona para adição de figuras geométricas uma a uma. Deverá também implementar correctamente os métodos da super-classe FiguraGeométrica. Para simplificar, a área de um desenho geométrico será a soma das áreas dos objetos que o constituem.
e) Desenvolva um programa que permita testar todas as funcionalidades das classes e métodos criados. O programa deverá criar um desenho geométrico, adicionando figuras geométricas. No final, a aplicação deverá mostrar as propriedades do desenho geométrico, incluindo a área e os constituintes.

Polimorfismo

Uma das facetas mais importantes da herança é a relação que se estabelece entre uma superclasse e uma classe descendente. Uma classe descendente é uma das variantes possíveis para os membros da superclasse.

Essa relação especial tem reflexos no modo como os métodos e variáveis existentes na superclasse passam para a classe descendente e no modo como um elemento de uma classe descendente se pode transformar num membro da super classe.

Voltando ao exemplo dos mamíferos, humanos e elefantes, sendo mamíferos, têm características semelhantes, como poder amamentar, mas diferem, por exemplo, no modo como andam. O mesmo comportamento (andar) difere nos humanos, que se deslocam na posição vertical, e nos elefantes, que se deslocam com as quatro patas.

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, dependendo do objeto que está a executar o método.

Se, usando de novo o exemplo dos transportes, redefinirmos em cada uma das classes os métodos herdados, obtemos:

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

class TransportePublico extends MeioTransporte {
	String[] passageiros = {"Xico", "Manel", "João"};
	//herdado
	void movimenta() {
		System.out.println("TransportePublico em movimento!!");
	}
	void picaBilhete() {
		System.out.println("\nA validar um bilhete de um
TransportePublico!!");
	}
}

class Comboio extends TransportePublico {
	int numCarruagens = 5;
	//herdado
	void movimenta() {
		System.out.println("Comboio em movimento!!");
	}
	void picaBilhete() {
		System.out.println("\nA validar um bilhete de Comboio!!");
	}
	void apita() {
		System.out.println("tutuuu!!");
	}
}
```

Se executarmos o código:

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

		System.out.println("\n\n--------TransportePublico:--------");
		TransportePublico t = new TransportePublico();
		System.out.println("condutor " + t.condutor);
		t.movimenta();
		System.out.println("passageiros:");
		for (int i = 0; i < t.passageiros.length; i++)
			System.out.print(t.passageiros[i] +  "     ");
		t.picaBilhete();

		System.out.println("\n\n----------Comboio:----------");
		Comboio c = new Comboio();
		System.out.println("condutor " + c.condutor);
		c.movimenta();
		System.out.println("passageiros:");
```

```
            for (int i = 0; i < c.passageiros.length; i++)
                  System.out.print(c.passageiros[i] +  "     ");
            c.picaBilhete();
            c.apita();
            System.out.println("num de carruagens " + c.numCarruagens);
```

Obtemos o resultado:

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

----------TransportePublico:----------
condutor Zé
TransportePublico em movimento!!
passageiros:
Xico, Manel, João
A validar um bilhete de um TransportePublico!!

----------Comboio:-----------
condutor Zé
Comboio em movimento!!
passageiros:
Xico, Manel, João
A validar um bilhete de Comboio!!
tutuuu!!
num de carruagens 5
```

De acordo com a classe do objeto que executa cada um dos métodos a versão correspondente é executada.

Além disso, existe ainda a possibilidade de um mesmo objeto ter várias versões de um método, desde que a assinatura seja diferente. O que quer dizer na prática que podemos ter vários métodos com o mesmo nome, desde que os tipos dos argumentos sejam diferentes (ver *overloading* de métodos).

Upcasting e downcasting

O **upcasting** e o **downcasting** são os mecanismos que nos permitem tratar uma instância de uma classe descendente como se fosse uma instância de uma classe ascendente, e vice-versa.

Upcasting

Já vimos que um membro de uma classe descendente pode ser sempre visto como um membro da sua super classe. Tomemos como exemplo a classe dos instrumentos musicais e a sua subclasse instrumentos de sopro:

```
class Musica {
      public static void main (String[] args) {
            Sopro flauta = new Sopro();
            // upcasting: flauta é um Sopro, mas é passado ao método afina,
            // que espera um Instrumento
```

```
                Instrumento.afina(flauta);
        }
}

class Instrumento {
        static void afina (Instrumento i) { //método da classe
                System.out.println("A afinar um " + i);
                i.toca();
        }
        public void toca () {       //método das instâncias
                System.out.println("A tocar um " + this);
        }
}

// Instrumentos de sopro. Têm os mesmos métodos que os outros Instrumentos.
class Sopro extends Instrumento {
}
```

A classe Instrumento possui um método de classe chamado afina() que recebe um Instrumento e possui ainda um método de instância toca. Na classe Musica é criado um instrumento de sopro, uma flauta, e é chamado o método da classe Instrumento, afina(), mas em vez de se passar o **handle** de um Instrumento, passa-se o handle de um Sopro (a flauta). O Java sabendo que um Sopro é um tipo de Instrumento, e portanto possui tudo o que um Instrumento deve possuir, converte automaticamente o handle do Sopro num handle de Instrumento e o código funciona sem problemas.

Chama-se **upcasting** ao ato de converter um handle de uma classe descendente num handle de uma classe superior.

Curiosamente, apesar de ao chamar o afina se converter implicitamente a flauta num handle de um Instrumento, a flauta em si continua a ser um Sopro, e, por isso, quando se executa o código do main de Musica obtém-se:

```
A afinar um Sopro@1ee7f9
A tocar um Sopro@1ee7f9
```

Como um objeto de uma classe descendente possui tudo o que possui um objeto da sua super classe (por herança desta) é sempre possível fazer o upcasting. Está-se simplesmente a passar de um tipo mais específico para um tipo mais geral. O objeto do tipo descendente pode conter mais atributos e mais métodos, mas não pode conter menos. Daí que não seja necessária nenhuma notação especial ou cast explícito para se fazer o upcasting.

Downcasting

O contrário de upcasting é o **downcasting** e consiste em transformar um handle de uma classe mais genérica (ascendente) numa classe mais específica (descendente).

O downcasting também é possível, mas só nalguns casos, nomeadamente se o handle apontar na realidade para um objeto do tipo mais específico. Por exemplo:

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

Criamos um Sopro, mas passamos a referi-lo com um handle de um Instrumento (upcasting). Mais à frente voltamos a reconvertê-lo num Sopro, s, (downcasting). Não há problema porque f é na realidade um Sopro, por isso, garantidamente possui tudo o que um Sopro deve possuir. No entanto, quando tentamos fazer o mesmo com um Instrumento, não funciona do mesmo modo. Na realidade a execução do código dará origem a uma exceção. Isto porque um objeto da classe Sopro, pertencendo a uma classe descendente, pode possuir mais métodos e atributos do que uma instância da classe Instrumento. Se assim for, como no caso de g, que na realidade aponta para um Instrumento e não possui a informação adicional que um Sopro deveria possuir, dá-se uma exceção. Dito de outra forma, embora seja verdade que um Sopro é sempre um Instrumento, nem sempre um Instrumento é um Sopro.

EXERCÍCIOS

1. Animais

Implemente um programa para caracterização de animais de vários tipos, de acordo com os seguintes requisitos:

- um Animal é caracterizado pelo tipo (mamífero, ave, réptil, peixe) e espécie (cão gato, galinha, cobra, bacalhau, etc.);
- quando se desloca, o animal anda, voa, rasteja ou nada, consoante o seu tipo.

a) Crie a classe Animal com os seus atributos tipo e id. Implemente o método desloca() que escreve no ecrã “Animal a deslocar-se.”.

b) Crie as classes Mamifero, Ave, Reptil e Peixe, descendentes da classe Animal. Em cada uma destas classes, redefina o método desloca(), de forma a indicar qual o tipo e espécie de animal e a forma como se desloca. Exemplo: “Réptil Cobra a rastejar.”.

c) Teste a sua estrutura de classes, mediante a criação de um array de objetos de diferentes tipos e espécies de animais, e a chamada dos respetivos métodos desloca().

2. Empresa

Uma empresa possui dois tipos de empregados: os que trabalham à comissão e os que trabalham à hora. Cada empregado é caracterizado pelo seu nome e número. Os ordenados dos empregados que trabalham à comissão são calculados adicionando ao ordenado base o total de vendas que efetuaram multiplicado pela sua percentagem de comissão. Os ordenados dos empregados que trabalham à hora são calculados multiplicando o número de horas que trabalharam pelo preço da hora de trabalho. Pretende-se que desenvolva um programa que permita:

a) atualizar os dados diários de um empregado conforme o seu tipo;

b) visualizar os dados de um empregado ao longo de um mês;

c) implementar e visualizar a contabilidade mensal (ordenados a pagar) da empresa.

3. Gestão de produtos alimentares

Uma empresa de transformação e comércio de produtos alimentares tem dois tipos de produtos: frescos e transformados.
Qualquer produto é caracterizado por uma descrição, origem, peso, taxa de IVA e preço (kg) de aquisição.
No caso dos produtos frescos, deverá haver um registo do custo do processo de embalamento, por kg.
À caracterização dos produtos transformados acresce o peso bruto (após transformação) e o custo médio, por kg, do processo de transformação daquele tipo de produto.
Os produtos transformados podem ser congelados ou conservas. No caso das conservas, acresce o valor, por kg, do produto utilizado na conserva (óleo, azeite ou outro). Considerando estes factos e o que aprendeu sobre polimorfismo, faça o seguinte:

a) Crie as classes necessárias à representação deste problema.

b) Também em cada classe criada, crie o respetivo método 'String toString()' que devolverá uma String que, além dos valores dos atributos de cada objeto, deverá conter o tipo de produto (fresco, congelado ou conserva).
c) Na classe Produto, crie o método 'double custoFinal()' que devolve o valor final de cada produto, tendo em conta as condições da sua aquisição, processo de embalamento ou transformação. Faça as intervenções necessárias nas restantes classes, de forma a obter este resultado para cada tipo de produto.
d) Crie um programa que guarde os produtos existentes para venda (pré-definidos), tendo em conta que cada registo corresponderá a uma embalagem, e peça ao utilizador um último produto a acrescentar ao conjunto de registos.
e) Adicione ao seu programa um método que faça uma listagem de todos os produtos registados, incluindo o respetivo custo final. Deverá, ainda, apresentar o custo total de todos os produtos existentes.

4. Conferência

Pretende-se um programa de suporte à organização de conferências. Uma conferência tem um nome, localização e data. Além disso, cada conferência tem um comité de organização e um comité de programa. Os membros dos dois comités são identificados pelo nome, endereço de correio eletrónico e instituição. Os membros do comité de organização são também caracterizados pelo papel que desempenham na conferência (finanças, espaços e refeições). Os membros do comité de programa são caracterizados por um parâmetro que reflete o seu desempenho na edição anterior da conferencia (mau, suficiente, bom, excelente). Além disso, os membros do comité de programa têm também associada uma lista dos artigos que devem rever.

Cada artigo é caracterizado pelo título, lista de autores e o resultado de avaliação (nota entre 1 e 5, sendo 5 a melhor classificação). A avaliação final de cada artigo é obtida pela média das avaliações realizadas por 2 membros do comité de programa. Os autores são caracterizados pelo nome, endereço de correio eletrónico, instituição e valor de inscrição a pagar, podendo ser de dois tipos: seniores (professores ou investigadores) ou estudantes. No caso dos estudantes deve considerar-se o grau que frequentam (licenciatura, mestrado ou doutoramento).

O custo de inscrição na conferência é de 400€ para seniores e 200€ para estudantes. Uma inscrição dá direito a apresentar um artigo. Autores seniores que apresentem mais do que um artigo devem pagar 50€ por cada artigo adicional. Considera-se que o primeiro autor de cada artigo será responsável pela sua apresentação na conferência.

Desenvolva um programa em Java que permita realizar as operações seguintes:

a) Listar os membros dos comités de organização e programa de determinada instituição, indicando o comité a que cada um pertence;
b) Apresentar estatísticas relativas às notas obtidas por todos os artigos, indicando quantos artigos tiveram nota média de 1, 2, 3, 4 e 5;
c) Determinar o valor realizado em inscrições, considerando que são aceites todos os artigos cuja avaliação seja igual ou superior a 3.5.

5. PoW

POAO of Warcraft (PoW) é um jogo RPG (Role Playing Game) onde personagens com diferentes características lutam em parcerias. As personagens têm um nome, nível de experiência (valor entre 1 e 25, aleatoriamente), e valores de força, inteligência e agilidade. As personagens podem ser de três tipos: guerreiros, magos e mercenários. Os guerreiros podem ou não ter armaduras e têm uma arma de curto alcance que podem ser: facas, machados ou espadas. Os magos têm uma mochila com sementes e folhas que usam para fazer poções. Os mercenários têm uma arma de longo alcance que podem ser pedras, arcos ou pistolas, e o número de munições. Para cada tipo de personagem, os níveis iniciais são os seguintes:

Tipo personagem	Experiência	Força	Agilidade	Inteligência
Guerreiro	Aleatório 1-25	10	5	3
Mago	Aleatório 1-25	2	4	9
Mercenário	Aleatório 1-25	4	10	4

Pretende-se desenvolver um programa para a gestão das personagens do jogo. Para isso, deve completar as seguintes alíneas:

a) Crie as classes necessárias para representar o problema descrito.

b) Crie a classe “PoW” e defina uma estrutura (ArrayList de Personagens) que contenha 5 personagens guerreiros, 5 personagens magos, e 5 personagens mercenários que permitam verificar as funcionalidades indicadas nas alíneas seguintes.

c) Desenvolva o código necessário para imprimir os mercenários com arcos, os magos com sementes de abóbora, e os guerreiros com armaduras.

d) Desenvolva o código necessário para subir o nível de experiência das personagens (deve mostrar estatísticas antes e depois de subir de nível), tendo em conta as regras seguintes:

Tipo personagem	Experiência	Força	Agilidade	Inteligência
Guerreiro	+1	Acrescenta 20%	Acrescenta 10%	Acrescenta 5%
Mago	+1	Acrescenta 5%	Acrescenta 10%	Acrescenta 20%
Mercenário	+1	Acrescenta 8%	Acrescenta 20%	Acrescenta 8%