Material Complementar

1 Herança

Um dos benefícios mais importantes que a orientação a objetos propicia é a **reusabilidade** de código. Um dos principais recursos presentes em linguagens com suporte à orientação a objetos é a herança. Uma classe pode fazer a definição de elementos (atributos, métodos) que sejam comuns a diversas outras classes e então, por meio da herança, elas obtêm uma cópia desses elementos sem ter que defini-los novamente.

1.1 (Criando um projeto) Como de costume, começamos criando um projeto. Escolha um nome apropriado para o contexto de estudo, que te ajude depois a lembrar do que se trata essa solução.

1.2 (Um sistema acadêmico sem herança) Vamos implementar partes de um sistema acadêmico (como o gerenciador acadêmico de uma universidade). O primeiro passo é identificar nosso "mini mundo". O que há no mundo real que também é de interesse implementar no sistema? Algumas classes possivelmente serão: Aluno e Professor. Note, porém, que há alguns tipos específicos de alunos bem como de professores. Por exemplo, uma universidade pode ter alunos de graduação e de pós. Pode ser, também, que existam professores horistas, professores pesquisadores e assim por diante. Assim, começamos nossa implementação conforme mostram as listagens de 1.2.1 a 1.2.4.

Listagem 1.2.1

```
public class AlunoDeGraduacao {
        private String nome;
        private int idade;
        private double nota1, nota2, notaFinal;
        //getters/setters
}
```

Listagem 1.2.2

```
public class AlunoDePosGraduacao {
        private String nome;
        private int idade;
        private char conceito;
        //getters/setters
}
```

Listagem 1.2.3

```
public class ProfessorHorista {
        private String nome;
        private int idade;
        //getters/setters
}
```

Listagem 1.2.4

```
public class ProfessorPesquisador {
		private String nome;
		private int idade;
		//getters/setters
	public void pesquisar () {
			System.out.println("Pesquisando..");
		}
}
```

1.3 (Aplicando a herança, crie um novo projeto) Note que o sistema possui diversas classes, cada uma responsável por representar um tipo específico, o que está de acordo com o princípio da **alta coesão**. Apesar disso, as classes têm muito em comum e cada uma delas se encarrega de fazer suas próprias definições. Por exemplo, os atributos **nome e idade estão presentes em todas as classes**. O que ocorreria com esse sistema caso fosse necessária a adição/remoção/alteração de mais atributos para todos os seres humanos que fazer parte dele? Para cada atributo seria necessário abrir cada uma das classes e fazer a adição/remoção/alteração. Neste passo iremos aplicar o conceito conhecido como **herança** para definir elementos comuns a diversas classes uma única vez e permitir que elas os reutilizem.

Nota: Iremos momentaneamente deixar de utilizar o **encapsulamento** para depois aplicá-lo e ver seu impacto quando envolvido com a herança.

1.3.1 Primeiro precisamos identificar o que as classes têm em comum. Todos os professores e alunos são pessoas e têm nome e idade em comum, certo? Sendo assim iremos criar uma classe para guardar essas partes em comum. Elas serão definidas na classe Pessoa da Listagem 1.3.1.1.

Listagem 1.3.1.1

```
public class Pessoa {
		String nome;
		int idade;
}
```

1.3.2 Agora iremos definir cada classe de interesse. A primeira será a classe AlunoDeGraduacao. Note que todo aluno de graduação é uma pessoa, evidentemente. Assim, faz sentido estabelecer o relacionamento entre classes conhecido como **É-UM (do inglês IS-A)** entre AlunoDeGraduacao e Pessoa, o que se faz por meio da herança. O operador da herança é o **extends**. Uma vez que esse relacionamento seja definido, tudo aquilo que é definido em Pessoa é herdado por AlunoDeGraduacao. Veja a Listagem 1.3.2.1.

Listagem 1.3.2.1

```
public class AlunoDeGraduacao extends Pessoa {
		//não definimos nome e idade aqui mais..eles são herdados.
		double nota1, nota2, notaFinal;
}
```

Perceba, no teste da Listagem 1.3.2.2, que objetos do tipo AlunoDeGraduacao têm acesso aos atributos nome e idade, embora a classe AlunoDeGraduacao não os defina. Isso se dá graças à herança. Dizemos que AlunoDeGraduacao herdou os atributos (isso também vale para métodos) da classe Pessoa. Tudo que é definido pela classe Pessoa também faz parte da classe AlunoDeGraduacao, graças ao uso do operador **extends** (que caracteriza o uso da herança).

Listagem 1.3.2.2

```
public class TesteComHeranca1 {
	public static void main(String[] args) {
		AlunoDeGraduacao aluno = new AlunoDeGraduacao ();
		aluno.nome = "José";
		aluno.idade = 19;
		System.out.printf("nome: %s, idade: %d", aluno.nome, aluno.idade);
	}
}
```

1.3.3 As listagens de 1.3.3.1 a 1.3.3.3 mostram as demais definições, agora usando herança.

Listagem 1.3.3.1

```
public class AlunoDePosGraduacao extends Pessoa{
	char conceito;
}
```

Listagem 1.3.3.2

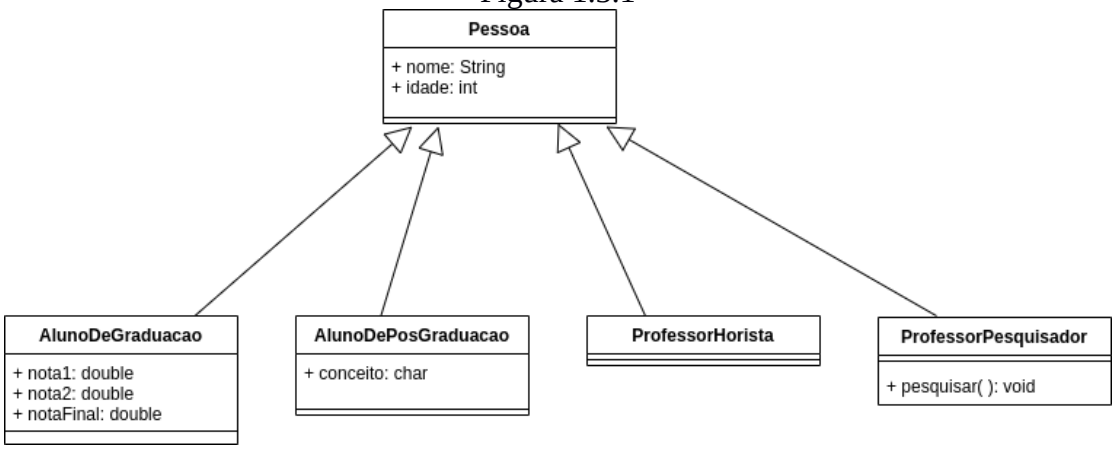
```
public class ProfessorHorista extends Pessoa{

}
```

Listagem 1.3.3.3

```
public class ProfessorPesquisador extends Pessoa{
    public void pesquisar () {
		System.out.println("Pesquisando..");
    }
}
```

Um diagrama de classes pode ser bastante útil para ilustrar a relação entre classes. A Figura 1.3.1 representa graficamente o que programamos até então.

Figura 1.3.1



1.4 (Refinando a solução com herança, crie um novo projeto) Note que alguns relacionamentos existentes no mundo real estão sendo ignorados. Por exemplo, professores horistas e pesquisadores têm algo em comum: eles são professores. É importante observar essa relação pois caso seja necessário algum ajuste que se aplique a todos os professores no sistema, isso poderá ser feito em uma única classe, desde que ela exista, evidentemente. O mesmo ocorre com alunos de graduação e de pós. Como professores, são pessoas. Porém, diferente dos professores, são alunos. As listagens de 1.4.1 a 1.4.7 exibem a nova implementação.

Listagem 1.4.1

```
public class Pessoa {
	String nome;
	int idade;
}
```

Listagem 1.4.2

```
public class Professor extends Pessoa{
	int matricula;
}
```

Listagem 1.4.3

```
public class Aluno extends Pessoa {
	int ra;
}
```

Listagem 1.4.4

```
public class AlunoDeGraduacao extends Aluno{
	double nota1, nota2, notaFinal;
}
```

Listagem 1.4.5

```
public class AlunoDePosGraduacao extends Aluno{
	char conceito;
}
```

Listagem 1.4.6

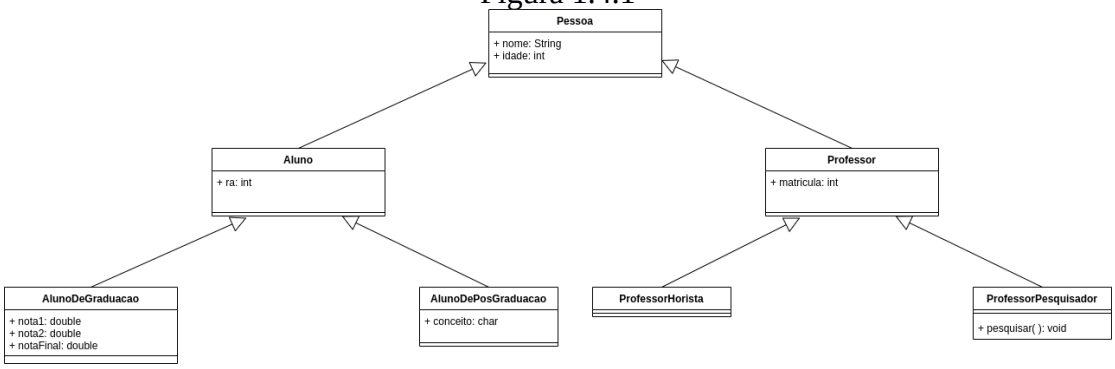
```
public class ProfessorHorista extends Professor{

}
```

Listagem 1.4.7

```
public class ProfessorPesquisador extends Professor{
    public void pesquisar () {
                    System.out.println("Pesquisando..");
    }
}
```

O diagrama de classes referente a essa nova solução é exibido na Figura 1.4.1.

Figura 1.4.1



1.5 (Nomenclatura utilizada) Alguns termos técnicos são comuns quando uma solução computacional envolve a herança. Os mais comuns são os seguintes. Usamos o exemplo da Figura 1.4.1 para simplificar.

- Pessoa é **superclasse** ou **classe-mãe** ou **classe-base** de todas que herdam direta ou indiretamente dela (Aluno, AlunoDeGraduacao, Professor etc).

- Professor é **subclasse** ou **classe-filha** ou **classe derivada** da classe Pessoa. Também é verdade que Professor é **superclasse (classe-mãe ou classe-base)** de ProfessorHorista e de ProfessorPesquisador.

- A classe Aluno **herda** da classe **Pessoa**.

- A classe pessoa é **estendida** pela classe Professor.

- A classe Professor **herda diretamente** da classe Pessoa.

- A classe ProfessorPesquisador **herda indiretamente** da classe Pessoa.

1.6 (Herança e encapsulamento) Vamos adicionar um método chamado **lecionar** à classe Professor. Quando ele for chamado, desejamos que o nome do professor seja exibido. Como todo professor leciona, esse método pode ser definido na classe Professor e automaticamente herdado por todas as classes que herdam direta ou indiretamente de professor. Veja a Listagem 1.6.1.

Listagem 1.6.1

```
public class Professor extends Pessoa{
        int matricula;
        public void lecionar () {
                System.out.println(nome + " lecionando...");
        }
}
```

A Listagem 1.6.2 mostra um teste para o método lecionar. Note que todo professor pode lecionar.

Listagem 1.6.2

```
public class TesteLecionar {
        public static void main(String[] args) {
                ProfessorHorista profHorista = new ProfessorHorista();
                profHorista.nome = "Rodrigo";
                ProfessorPesquisador profPesq = new ProfessorPesquisador();
    profBD.nome = "José";
    profHorista.lecionar();
                profPesq.lecionar();
        }
}
```

1.6.3 Desejamos, porém, fazer uso do encapsulamento, pois sabemos que um benefício imediato obtido por sua aplicação é o **baixo acoplamento** entre classes. Assim, os atributos nome e idade da classe Pessoa passarão a ser marcados como **private**. Veja a Listagem 1.6.3.

Listagem 1.6.3

```
public class Pessoa {
        private String nome;
        private int idade;
}
```

Inspecione a classe Professor e veja o que aconteceu. O método lecionar não compila mais. Isso ocorre, pois, **membros marcados como private têm acesso restrito à classe que os define**. Como vimos na aula de encapsulamento, classes cliente não podem acessá-los. Agora vemos que nem mesmo subclasses podem acessá-los.

Como resolver esse problema? A solução já conhecemos. Embora os atributos não possam ser acessados diretamente, o acesso pode ser realizado por meio dos métodos de acesso, os getters e setters. Veja a refatoração da classe Professor na Listagem 1.6.4 e o teste na Listagem 1.6.5.

Listagem 1.6.4

```
public class Professor extends Pessoa{
	int matricula;
	public void lecionar () {
		System.out.println(getNome() + " lecionando...");
	}
}
```

Listagem 1.6.5

```
public class TesteLecionar {
	public static void main(String[] args) {
		ProfessorHorista profHorista = new ProfessorHorista();
		profHorista.setNome("Rodrigo");
		ProfessorPesquisador profPesq = new ProfessorPesquisador();
	profBD.setNome("José");
	profHorista.lecionar();
		profPesq.lecionar();
	}
}
```

1.7 (Herança e construtores, crie um novo projeto) Quando definimos uma classe ela sempre tem um construtor. Caso o programador não escreva um explicitamente, o compilador cria por conta o conhecido **construtor padrão**. Quando uma classe herda de outra é importante observar quais construtores estão definidos na superclasse para que a herança possa acontecer corretamente. Vejamos o seguinte exemplo.

1.7.1 Crie uma classe chamada Pessoa. Pessoas têm, novamente, nome e idade. O único construtor existente é o padrão. Veja a Listagem 1.7.1.1.

Listagem 1.7.1.1

```
public class Pessoa {
	private String nome;
	private int idade;
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public int getIdade() {
		return idade;
	}
	public void setIdade(int idade) {
		this.idade = idade;
	}
}
```

1.7.2 A classe Aluno herda da classe Pessoa, como mostra a Listagem 1.7.2.1.

Listagem 1.7.2.1

```
public class Aluno extends Pessoa{

}
```

1.7.3 Agora vamos definir os construtores padrão de Pessoa e de Aluno. Ambos mostram uma mensagem. A Listagem 1.7.3.1 mostra ambos.

Listagem 1.7.3.1

```
//na classe Pessoa
public Pessoa () {
        System.out.println("Construindo Pessoa...");
}
//na classe Aluno
public Aluno () {
        System.out.println("Construindo Aluno...");
}
```

1.7.4 A Listagem 1.7.4.1 constrói um objeto do tipo Pessoa e outro, do tipo Aluno. Analise o resultado.

Listagem 1.7.4.1

```
public class TesteConstrutoresPadrao {
        public static void main(String[] args) {
                Pessoa p = new Pessoa();
                Aluno a = new Aluno();
        }
}
```

Perceba que a mensagem “Construindo Pessoa...” apareceu duas vezes. Isso ocorreu pois a chamada ao construtor da classe Aluno inclui, implicitamente, uma chamada ao construtor de sua **super**classe, a classe Pessoa. Essa chamada é feita por meio do operador **super** (veja como o nome é bem escolhido). Lembra-se de que o construtor padrão é criado automaticamente pelo compilador caso o programador não escreva nenhum. Pois é, em seu corpo, o compilador também adiciona uma chamada super, como mostra a Listagem 1.7.4.2.

Listagem 1.7.4.2

```
//na classe Aluno
public Aluno () {
        super();
        System.out.println("Construindo Aluno...");
}
```

A chamada super implica em uma chamada ao construtor da **super**classe e ela sempre existe, mesmo que o programador não a escreva explicitamente.

1.8 (Escrevendo um construtor personalizado na classe Pessoa) Digamos que queremos construir objetos do tipo Pessoa já informando os valores a serem atribuidos às variáveis nome e idade, como mostra a Listagem 1.8.1. Além disso, não queremos a existência do construtor padrão, obrigando que esses valores sejam sempre informados.

Listagem 1.8.1

```
//na classe Pessoa
public Pessoa (String nome, int idade) {
        System.out.println("Construindo Pessoa...");
        setNome(nome);
        setIdade(idade);
}
```

Note que a classe Aluno não compila mais. Isso ocorre por uma razão muito simples. Seu construtor padrão tenta colocar em execução o construtor padrão da classe Pessoa. Porém ele não existe mais, já que criamos um personalizado e ele não se encontra escrito explicitamente na classe Pessoa. As listagens 1.8.2 e 1.8.3 mostram duas soluções. Na primeira, o construtor padrão de Aluno envia valores fixos por meio da chamada ao construtor da super classe. Embora funcione, isso pode não ser muito útil, já que esses valores potencialmente serão diferentes a cada objeto criado. A segunda é mais flexível. O construtor de Aluno recebe os valores e os repassa ao construtor da superclasse.

Listagem 1.8.2

```
//na classe Aluno
public Aluno () {
        super("Maria", 19); //funciona, pouco flexível
        System.out.println("Construindo Aluno...");
}
```

Listagem 1.8.3

```
//na classe Aluno
public Aluno (String nome, int idade) {//recebe os valores
        super(nome, idade); //somente repassa, mais flexível
        System.out.println("Construindo Aluno...");
}
```

A Listagem 1.8.4 mostra um teste que usa os novos construtores.

Listagem 1.8.4

```
public class TesteNovosConstrutores {
        public static void main(String[] args) {
                Pessoa pessoa = new Pessoa ("João", 37);
                Aluno aluno = new Aluno ("Maria", 19);
        }
}
```

Note que a chamada super deve ser a primeira instrução no corpo de um construtor. Caso o programador não a escreva explicitamente, ela já está ali e é a primeira. Caso o programador a escreva, deverá tomar cuidado para que nenhuma instrução ocorra antes dela. Veja a Listagem 1.8.5. Ela ilustra um trecho de código inválido, que viola essa característica. Intuitivamente, obrigar super a ser a primeira instrução, no nosso exemplo, indica que um aluno, antes de ser aluno, é uma pessoa e, portanto, suas características de pessoa devem ser estabelecidas antes de suas características de aluno.

Listagem 1.8.5

```
public class Aluno extends Pessoa{
        //na classe Aluno
        public Aluno (String nome, int idade) {//recebe os valores
                System.out.println("Construindo Aluno...");
                super(nome, idade); //inválido, nao faça isso
        }
}
```

1.9 (Uma nota sobre a classe Object) Na API do Java, existe uma classe chamada Object da qual todas herdam. Ela define diversos métodos muito úteis sobre os quais aprenderemos ao longo do curso. No momento, basta saber que TODA CLASSE herda direta ou indiretamente de Object e, portanto, tem acesso a tudo aquilo (que não for private) que ela define, como métodos chamados **equals**, **hashcode**, **toString**, **notify**, **notifyAll etc**. Em breve saberemos mais sobre alguns deles. Veja a documentação de Object em:

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Exercícios

1 Escreva uma hierarquia de classes envolvendo as seguintes classes: Figura, Quadrado, Retangulo, Circulo, Losango e Trapézio. Figura deve ser superclasse de todas. Todas as demais devem herdar diretamente de Figura.

2 Refine a solução do Exercício 1. Agora devem existir as classes Figura2D e Figura 3D. Ambas (e somente elas) herdam diretamente de Figura. As demais devem todas herdar de Figura2D, de Figura3D ou ainda de alguma outra. Devem existir também as classes Quadrilatero, Cubo e Esfera.

3 Escreva uma hierarquia de classes para representar animais. Deve haver uma classe Animal que define o nome e a quantidade de patas do animal. Ela deve ter três construtores: o padrão, um que recebe uma string a ser atribuída ao nome do animal e um terceiro que recebe o número de patas e o nome. Escreva uma classe chamada Mamifero que herda de Animal. Ela deve ter um construtor somente. Ele recebe o nome e a quantidade de patas do Animal e os repassa para o construtor da superclasse. Escreva duas classes chamadas Cachorro e Gato que herdam de Mamifero. Ambas definem um único construtor que recebe o nome do Animal e repassa para a superclasse. Internamente, ambos devem repassar o valor 4 como quantidade de patas.

4 Escreva uma classe de Teste que instancia um Cachorro e um Gato.

5 Escreva uma classe chamada Oviparo. Pesquise sobre animais oviparos e encaixe dois tipos de oviparos como subtipo de Oviparo. Coloque dois atributos em Oviparo que sejam comuns a todos os oviparos.

6. Considere a classe Ornitorrinco. Embora sejam considerados oviparos, as fêmeas produzem leite e poderiam ter características de interesse já definidas na classe mamífero. Responda: De qual classe Ornitorrinco deveria herdar?

7 Faça um diagrama de classes que mostra todas as classes da hierarquia de animais, incluindo ornitorrinco. Lembre-se: no Java só existe herança simples. Isso quer dizer que uma classe só pode herdar diretamente de uma única classe.

8 Faça uma pesquisa sobre o fenômeno conhecido como **Deadly Diamond of Death** e escreva cinco linhas sobre ele. Faça um diagrama de classes em que ele aparece.

Referências

DEITEL, P. e DEITEL, H. **Java Como Programar**. 8ª Edição. São Paulo, SP: Pearson, 2010.

LOPES, A. e GARCIA, G. **Introdução à Programação – 500 Algoritmos Resolvidos**. 1ª Edição. São Paulo, SP: Elsevier, 2002.