Unidade III

5 POLIMORFISMO

Considerado outro pilar da orientação a objetos, o polimorfismo é uma característica que define que alguns elementos da linguagem orientada a objetos (como suas variáveis, os atributos, os métodos e as classes) podem ter formas diferentes de comportamentos.

Para o Java, o polimorfismo é a capacidade de um determinado elemento se comportar de formas diferentes, dependendo de como ele é declarado, ou de como está sendo utilizado, a partir da sua codificação.

5.1 Polimorfismo de variáveis

Em Java, algumas variáveis (como os atributos) têm "alcance global" na classe (variáveis globais, ou seja, aquelas que podem ser utilizadas por qualquer método da classe) e outras são geradas e declaradas em blocos, podendo ser utilizadas apenas naquele bloco.

No entanto, é possível termos duas variáveis com mesmo nome num mesmo bloco, mas com comportamentos diferentes (global e local). Como exemplo deste caso, é muito comum encontrarmos essa situação nos encapsulamentos, quando nos métodos set... temos o parâmetro com o mesmo nome do atributo. Neste caso, temos duas variáveis com o mesmo nome, mas com comportamentos diferentes (uma sendo o atributo da classe e a outra sendo a variável local, definida no parâmetro).

Para diferenciarmos as duas, colocamos juntamente à variável que representa o atributo a palavra this., como no exemplo a seguir:

```
public class Carro {
   public double velocidade;
   //... e demais atributos
   public void setVelocidade (double velocidade) {
     this.velocidade = velocidade;
   }
   //... e demais métodos
}
```

Observação

Na IDE do Eclipse é possível gerar automaticamente as classes setters e getters. Seus métodos setters são definidos como o exemplo que acabamos de apresentar.

Percebe-se nesse exemplo que o método setVelocidade(...) recebe como parâmetro um valor numérico do tipo double, e que o parâmetro será recebido pela **variável local** velocidade. Esse método insere o valor, recebido pelo seu parâmetro, no **atributo** velocidade. Assim, para diferenciar as varáveis (atributo velocidade da variável local velocidade), utilizamos a palavra reservada this, de forma que:

- **this.velocidade**: equivale ao atributo velocidade da classe;
- **velocidade**: equivale à variável local do método.

Ou seja, são duas formas diferentes de variáveis que possuem o mesmo nome existentes num mesmo bloco (o bloco do método setVelocidade(...)).

5.2 Polimorfismo de métodos

Vimos anteriormente que uma mesma classe não aceita métodos que possuam a mesma assinatura entre si. No entanto, vimos também (em itens anteriores) que existem duas situações em que há semelhanças de assinaturas entre métodos, no que diz respeito ao seu nome, representando as duas formas de polimorfismo de métodos. São elas:

- **Sobrecarga de métodos (ou overloading)**: a presença, numa mesma classe ou em classes com herança, de dois ou mais métodos com o mesmo nome e diferenciados nos parâmetros.
- **Sobrescrita de métodos (ou overiding)**: a presença, neste caso apenas em classes com herança, de dois ou mais métodos com a mesma assinatura (mesmo nome e mesmos parâmetros), um na superclasse e outro na subclasse.

5.3 Polimorfismo de classes

O polimorfismo de classes é um tipo de polimorfismo em que uma classe se comporta como se fosse outra classe. Isso só é possível quando há herança de classes, em que uma superclasse se comporta como uma subclasse.

Sabemos que uma classe (classe mãe, ou superclasse) pode ser herdada por diversas outras classes (classes filhas, ou subclasses). Dessa forma, podemos fazer com que um objeto representando uma classe mãe receba (ou seja instanciado com) a instância de uma de suas classes filhas, caracterizando assim o polimorfismo de classe.

Exemplo:

```
public class Carro {
   private String tipoClasse = "Carro";
   //... e demais atributos
   public String getTipoClasse () {
     return tipoClasse;
   }
   public void mostraDados(Carro crr) {
     System.out.println(crr.getTipoClasse());
   }
}
public class Fusca extends Carro {
   private String tipoClasse = "Fusca";
   //... e demais atributos
   public String getTipoClasse () {
     return tipoClasse;
   }
}
public class Ferrari extends Carro {
   private String tipoClasse = "Ferrari";
   //... e demais atributos
   public String getTipoClasse () {
     return tipoClasse;
   }
}
```

Vejamos então a ocorrência do polimorfismo de classe:

```
public class Teste {
   public static void main(String[] args) {
     Carro c1 = new Carro();
     Fusca fu = new Fusca();
     Ferrari fe = new Ferrari();
     c1.mostraDados(c1);
     c1.mostraDados(fu);
     c1.mostraDados(fe);
     /*
      * imprimirá:
      *   Carro
      *   Fusca
      *   Ferrari
      */
   }
}
```

Neste exemplo, o método mostraDados(...) da classe Carro recebe como parâmetro um objeto do tipo Carro. Porém na classe Teste geramos três instâncias, uma instância para cada uma das classes (Carro, Fusca e Ferrari). Com essas instâncias, chamamos o método mostraDados(...) da classe Carro (que recebe a classe Carro como parâmetro), e enviamos cada uma daquelas instâncias por vez, observando que a saída é diferente para cada uma delas. Isso mostra que o objeto crr do parâmetro daquele método se comporta de formas diferentes para cada um dos objetos enviados.

Exemplo de aplicação

Um dos pilares da orientação a objetos é a possível existência de polimorfismo entre os seus elementos, de forma que a linguagem permite o polimorfismo de métodos e de classes.

De acordo com o conceito de polimorfismo, qual dos itens a seguir está **incorreto**?

A) A sobrecarga é um tipo de polimorfismo de métodos em que, numa mesma classe ou entre superclasses e suas subclasses, temos a presença de métodos com o mesmo nome, mas com parâmetros diferentes na mesma classe ou entre superclasses e subclasses.

B) A sobrescrita é um tipo de polimorfismo de métodos em que há ocorrência de métodos com mesma assinatura e, portanto, só pode ocorrer entre superclasses e subclasses.

C) Quando um método tem como parâmetro um ou mais objetos que representam classes consideradas superclasses, esses parâmetros podem ser "preenchidos" (na chamada ao método) com objetos que representam outras classes, desde que elas sejam subclasses daquela representada pelo parâmetro.

D) O polimorfismo é uma característica de OO em que um mesmo elemento pode ser apresentado, ou utilizado, de formas diferentes, dependendo de condições específicas.

E) O polimorfismo é um recurso que permite que uma subclasse se utilize de qualquer método privado de uma superclasse, sem que seja preciso instanciá-la.

Resposta correta: alternativa E.

Resolução

A) Alternativa correta.

Justificativa: a sobrecarga, também conhecida como overloading, é observada quando temos ou numa mesma classe, ou em classe que caracteriza herança, mais de um método com o mesmo nome, mas com parâmetros diferenciados. Criamos esses métodos para que o conceito envolvido pela ação do método possa ser utilizado com tipos ou quantidades de informações diferentes.

B) Alternativa correta.

Justificativa: a sobrescrita, também conhecida como overiding, pode ser observada quando um método de uma classe filha (ou subclasse) possui a mesma assinatura da classe mãe (ou superclasse). Assim, quando instanciamos a classe filha, sabemos que ela contém todos os métodos da classe mãe, os quais podem ser acionados com aquele objeto. No entanto, se acionarmos o método que sofreu sobrescrita, será o método da classe filha que estaremos acionando, pois ele substitui o da classe mãe.

C) Alternativa correta.

Justificativa: o polimorfismo de classe é bastante observado em parâmetros de métodos quando esses parâmetros representam classes que possuem subclasses, de modo que esses objetos (os parâmetros) podem se comportar como um objeto que foi instanciado com qualquer uma de suas subclasses.

D) Alternativa correta.

Justificativa: a ideia do polimorfismo é a de apresentar formas diversas de se trabalhar com um mesmo elemento. Sua observância é mais simples quando pensamos em polimorfismo de métodos, em que temos métodos que possuem o mesmo nome (e, portanto, uma mesma chamada), mas que atuam de formas diferentes dependendo da necessidade do programa, dependendo dos valores enviados nos parâmetros.

E) Alternativa incorreta.

Justificativa: um método privado não pode ser utilizado (chamado) diretamente por nenhum método que não seja da própria classe. Isso não quer dizer que uma subclasse não o contenha, já que ela herda todos os métodos da superclasse. A subclasse apenas não poderá acionar diretamente esse método privado, mas poderá indiretamente, ao acionar outro método na superclasse que se utiliza daquele método privado. O polimorfismo, como se pode observar nas explicações que acabamos de oferecer, contém um conceito que não está relacionado a esse tipo de recurso.

Saiba mais

Saiba mais sobre polimorfismo no capítulo 7, itens 7.13, de:

FURGERI, S. *Java 7*: ensino didático. 2. ed. São Paulo: Érica, 2012.

6 MODIFICADORES DE COMPORTAMENTO E ALGUMAS CLASSES ÚTEIS

Os modificadores de comportamento são aqueles que alteram a forma de se trabalhar com os elementos e, consequentemente, o comportamento do elemento na funcionalidade do sistema.

Veremos que esses modificadores são utilizados com o intuito de controlar a forma de se programar o sistema, ou de como outros sistemas podem se utilizar dele. Esses modificadores são recursos que podem definir como a equipe de programadores deverá lidar com cada elemento do sistema.

6.1 Modificadores de comportamento

Esses modificadores são "somados" aos modificadores de acesso e alteram o comportamento dos elementos (classes, métodos e/ou atributos).

Lembrete

Modificadores de acesso (public, private, protected e (default)) são os termos iniciais das declarações de atributos e métodos, que controlam quais classes podem ou não acessá-los.

Tem-se os seguintes modificadores de comportamento:

- static;
- final;
- abstract.

Sua sintaxe, na declaração dos elementos, deve seguir a seguinte sequência:

modifAcesso modifComport declaracaoDoElemento

Exemplo:

```
public abstract class ClasseA {...}
public final void metodo01() {...}
```

Observação

Perceba que primeiro se declara o modificador de acesso e em seguida o modificador de comportamento.

6.2 Modificador final

O modificador final pode ser utilizado com classes, atributos e/ou métodos, alterando o comportamento desses elementos segundo as seguintes regras:

Uma classe final é uma classe que não pode ser herdada

Isso quer dizer que numa cadeia de heranças, uma classe final será a última classe da cadeia, já que não poderá ser herdada por outras classes.

Exemplo:

```
public final class ClasseA {
   //.. seus atributos e métodos
}
```

Isso quer dizer que se tentarmos criar outra classe que herda a classe ClasseA, o compilador acusará um erro de programação:

Exemplo:

```
// ### ERRO !!
public class ClasseB extends ClasseA {...}
//.. já que uma Classe final NÃO pode ser herdada ####
```

Observação

Esse tipo de erro (a ClasseB herdando a ClasseA) impede que o programa seja compilado (gera um erro de compilação), impedindo-o de ser acionado.

Um método final é um método que não pode ser sobrescrito

Quando uma classe herda outra classe, a primeira passa a possuir todos os métodos da classe herdada. O programador pode querer sobrescrever algum método da superclasse (recriá-lo na subclasse com a mesma assinatura), por conta de alguma característica específica daquela subclasse. No entanto, se o método da superclasse for um método final, essa substituição não pode ser realizada.

Exemplo:

```
public class ClasseA {
   public final void metodo01() {
     // ...
   }
```

```
    //.. seus atributos e demais métodos
}
public class ClasseB extends ClasseA {
    //.. seus atributos e métodos
    // ### ERRO !!
    public void metodo01() {...}
    //.. já que um método final NÃO pode ser sobrescrito ###
}
```

Observação

Esse tipo de erro (o metodo01(...) sendo sobrescrito) acentuado no exemplo impede que o programa seja compilado (gera um erro de compilação), impedindo-o de ser acionado.

Um atributo final é um atributo que não pode ser alterado (ou seja, é uma constante)

Normalmente, o atributo de uma classe é uma variável que adquire algum valor durante a sua existência em memória (na instância da classe) e que também pode ter seu valor alterado. No entanto, algumas vezes necessitamos estabelecer alguns valores que não podem ser alterados ao longo do sistema (valores como taxas, índices indicativos de uma empresa, elementos repetitivos de programação etc.). Nesse caso, trabalha-se com constantes identificadas como atributos finais. Pela convenção de programação, um atributo final (uma constante) deve ser descrito em caixa alta, ou seja, com todas as letras em maiúscula.

Exemplo:

```
public class CalculoMatematico {
    public final double PI = 3.1415926;
    //.. seus atributos e demais métodos
}
```

Ou, ainda:

```
public class Financeiro {
    public final double TAXA_IMPOSTO = 0.00175;
    //.. seus atributos e demais métodos
}
```

Neste caso, qualquer tentativa de alteração do valor dessa constante PI (pertencente à primeira classe) ou à constante TAXA_IMPOSTO (pertencente à segunda classe) que se faça ao longo do sistema (de uma forma geral) gerará um erro de compilação, impedindo que a classe seja compilada.

6.3 Modificador static

O modificador static pode ser utilizado com atributos e/ou métodos, de modo que todo elemento estático (static) possa ser acessado sem a necessidade de se instanciar a classe a que ele pertence.

Veja o exemplo a seguir:

```
public class Calculo {
   public double PI = 3.1415926;
   //.. e demais atributos
   public double calcularValor(double a, double b) {
     double resultado = 0;
     // processamento do valor – cálculo com "a" e com "b"
     return resultado;
   }
   //.. e demais métodos
}
```

No exemplo, se os métodos não forem estáticos, para se utilizar o atributo e o método da classe Calculo, deve-se antes instanciar a classe:

```
public class Teste {
   public static void main (String[] args) {
     // instanciando a Classe Calculo
     // ...para poder utilizar seus elementos:
     Calculo c = new Calculo();
     double x = 72.31 * c.PI;
     double y = c.calcularValor(1.75, 5.48);
     System.out.println(x + " :: " + y);
     //.. imprimirá: 227.168560906 :: 0.0
   }
}
```

Mas se os elementos da classe Calculo forem estáticos, para utilizarmos esses elementos na classe Teste não se faz necessária a instância daquela classe, e a chamada é feita diretamente a partir da classe, e não por uma instância da classe:

```
public class Calculo {
   public static double PI = 3.1415926;
   //.. e demais atributos
   public static double calcularValor(double a, double b) {
     double resultado = 0;
     // processamento do valor – cálculo com "a" e com "b"
     return resultado;
```

```
    }
    //.. e demais métodos
}
public class Teste {
    public static void main (String[] args) {
      // Utilizando os elementos sem instanciar a Classe
      double x = 72.31 * Calculo.PI;
      double y = Calculo.calcularValor(1.75, 5.48);
      System.out.println(x + " :: " + y);
      //.. imprimirá: 227.168560906 :: 0.0
    }
}
```

6.4 Modificador abstract

O modificador abstract pode ser utilizado com classes e/ou métodos, alterando o comportamento desses elementos segundo as seguintes regras:

Uma classe abstract é uma classe que não pode ser instanciada

Existem algumas razões para que possamos querer que uma classe não seja instanciada. Duas delas são as seguintes:

- quando criamos uma classe que conterá uma série de métodos estáticos e/ou atributos finais e estáticos, de forma que seus elementos (atributos e métodos) deverão ser acessados diretamente sem a instância da classe;

- quando criamos uma classe com elementos genéricos (uma classe mãe, ou superclasse), da qual geramos várias subclasses (herdando aquela classe mãe), e desejamos que a superclasse não seja instanciada, para ser utilizada somente a partir de suas classes filhas (instanciando-as).

Exemplo:

```
public abstract class ClasseA {
    // seus atributos
    public void metodo01() {
      // processamento do método
    }
    //.. e demais métodos
}
public class ClasseB extends ClasseA {
    // seus atributos e métodos
}
```

```
public class Teste {
   public static void main (String[] args) {
     // Instanciando a ClasseB,
     // ..já que a ClasseA não pode ser instanciada.
     ClasseB cb = new ClasseB ();
     cb.metodo01(); // método herdado da ClasseA
   }
}
```

Um método abstract é um método que não possui implementação, mas que exige a sua implementação na herança

Características importantes dos métodos abstratos:

- métodos abstratos só podem existir em classes abstratas;
- métodos abstratos não possuem implementação (percebam que métodos abstratos terminam com ponto-e-vírgula):

```
public abstract tipoRetorno nomeMetodo(parâmetros);
```

- os métodos abstratos necessitam obrigatoriamente ser implementados nas subclasses;
- caso tenhamos uma classe abstrata que herdou outra classe abstrata, os métodos abstratos desta última não precisam ser implementados na subclasse que a herdou, já que ela deverá ser herdada por outra classe.

Observação

Sabemos que um método abstrato só pode existir em uma classe abstrata, mas uma classe abstrata não precisa ter métodos abstratos, podendo ter tanto métodos comuns como também métodos abstratos – lembrando que uma classe abstrata é simplesmente uma classe que não pode ser instanciada.

Exemplo: imagine que estamos criando um jogo de corrida de carros no qual geramos as classes que representarão cada um dos carros que competirão juntamente na pista de corrida. Neste caso, geramos uma classe mais genérica (a classe Carro), que conterá as características que todos os tipos de carros de corrida terão no jogo. Essa classe, porém, não será uma classe instanciada diretamente, já que o jogador deverá poder escolher uma dentre as diversas opções de tipos de carro.

Na classe Carro (que será a superclasse com as características mais gerais que todos os carros terão no jogo), inserimos dois métodos abstratos (frear e acelerar), para que os programadores se lembrem de implementá-los em cada um dos tipos de carro que criarem para o jogo.

```
public abstract class Carro {
   // :: métodos abstratos
   public abstract void frear();
   public abstract void acelerar();
   // :: métodos implementados
   public void virar(String tipoVirada) {
     // :: construção da lógica
     if (tipoVirada.equals("direita")) {
              // .. lógica de curva para a direita
     }
     // ...
   }
   // .. e demais métodos
}
```

No exemplo, como a classe Carro é abstrata, não poderá ser instanciada. Isso quer dizer que outras classes irão herdá-la, especificando sua atuação no sistema. Nesse caso, os desenvolvedores que criarem as suas subclasses, ou seja, as classes que herdarem a classe Carro, deverão implementar os seus métodos abstratos. A preocupação em se fazer isso está em especificar cada uma dessas ações em cada uma das classes Filhas (como nas classes Fusca e Ferrari), de modo a obrigar que a cada uma dessas classes seja dado um comportamento específico dentro desses métodos.

```
public class Fusca extends Carro {
   // :: Implementação obrigatória dos métodos abstratos
   // .. existentes na Classe Carro herdada
   public void frear() {
     // .. lógica do método frear para o Fusca
   }
   public void acelerar() {
     // .. lógica do método acelerar para o Fusca
   }
   // .. e demais métodos
}
```

Observação

A IDE do Eclipse possui uma facilidade (um wizard) que faz com que de dentro da classe Fusca o desenvolvedor consiga puxar (gerar na classe) todos os métodos abstratos das classes herdadas. Clicando com o botão esquerdo do mouse sobre a classe Fusca e abrindo o menu Source – Override/ Implement Methods..., abrirá uma tela na qual os métodos abstratos das classes herdadas já estarão selecionados, bastando clicar sobre o botão OK desta tela, o qual irá gerar de forma padrão (com retornos padrões, quando necessário) os métodos necessários (com exatamente a mesma declaração definida no método abstrato, ou seja, mesmo tipo de retorno e mesmos parâmetros).

6.5 Tabela–resumo sobre os modificadores de comportamento

O quadro a seguir resume todos os conceitos que acabamos de definir referentes aos modificadores de comportamento:

Quadro 2 – Resumo sobre os modificadores de comportamento

	static	final	abstract
Classe	(Não existe)	Não pode ser herdada por outra classe	Não pode ser instanciada (deverá ser utilizada a partir de suas classes filhas)
Método	Pode ser **acionado** sem precisar instanciar a classe a que pertence	Não pode ser sobrescrito (não pode ser substituído por outro método em uma subclasse)	É um método sem implementação (só com a sua declaração) Ele deve ser implementado em uma classe filha Somente pode existir em uma classe abstrata
Atributo	Pode ser **lido** sem precisar instanciar a classe a que pertence (será um valor constante)	É constante (seu nome deve estar em caixa alta)	(Não existe)

Exemplo de aplicação

Quanto ao conceito de modificadores de comportamento, analise as afirmativas a seguir:

I – Uma classe estática é aquela destinada a conter somente constantes, a fim de não sofrer alterações nos valores de seus atributos.

II – O modificador final, de certa forma, é um modificador que proíbe ações com o elemento por ele caracterizado, de modo que uma classe final não pode ser herdada, um método final não pode ser sobrescrito e um atributo final não pode ter seu valor alterado.

III – Criam-se métodos abstratos para que, quando um desenvolvedor gerar uma subclasse daquela classe a que o método pertence, esses métodos sejam criados e tenham sua lógica devidamente desenvolvida, geralmente para suprir uma necessidade imposta por uma regra de negócio do sistema.

IV – Uma classe abstrata é uma classe que poderá ser utilizada de forma indireta, ou para servir de repositório de métodos estáticos, ou para servir de base e complementar a subclasse que a herda.

De acordo com as afirmativas, podemos dizer que estão corretas:

A) Apenas as afirmativas II, III e IV.

B) Apenas as afirmativas I, II e III.

C) Apenas as afirmativas I e IV.

D) Apenas as afirmativas II e IV.

E) Apenas as afirmativas I, III e IV.

Resposta correta: alternativa A.

Resolução

I – Afirmativa incorreta.

Justificativa: não se pode criar uma classe estática. Esse modificador somente é utilizado para atributos e métodos, de modo a fazer com que eles possam ser acessados sem a necessidade de se instanciar a classe a que pertencem.

II – Afirmativa correta.

Justificativa: o modificador final pode ser utilizado com classes, métodos e atributos, gerando respectivamente classes que não aceitam herança, métodos que não podem ser substituídos por outros em subclasses, impedindo o que conhecemos por overiding, e atributos considerados constantes e que, portanto, não podem ter seu valor alterado.

III – Afirmativa correta.

Justificativa: um método abstrato, obrigatoriamente construído em uma classe abstrata (que conceitualmente não pode ser instanciada, mas pode ser herdada), obriga a sua implementação nas subclasses que herdam a classe a que pertencem. Dessa forma, cria-se um método abstrato numa classe quando um desenvolvedor pretende definir regras que devem ser seguidas nas classes que a herdarem.

IV – Afirmativa correta.

Justificativa: uma classe abstrata, que não pode ser instanciada, pode servir de repositório de métodos estáticos ou como base para a construção de subclasses que conterão todos os seus atributos e métodos.

6.6 Interface

Uma interface é uma entidade da orientação a objetos **equivalente a** uma classe totalmente abstrata. Ela não é considerada uma classe, mas, assim como uma classe, também pode conter atributos e métodos. No entanto, em uma interface, todo atributo será um atributo estático e final, e todo método será um método abstrato.

Lembrete

Atributos estáticos e finais são considerados constantes, e um método abstrato é aquele que não possui implementação e que deve ser implementado na herança.

Como todo atributo e todo método de uma interface tem características como as que acabamos de definir, seus modificadores de comportamento não são descritos na declaração desses elementos.

Exemplo:

```
public interface Inter01 {
   public int ATRB01 = 11;
   public double ATRB02 = 22.33;
   public void metodo01(int x);
   public int metodo02();
}
```

Observação

Como pertencem a uma interface, os atributos ATBR01 e ATRB02 são estáticos e finais, e os métodos metodo01 e metodo02 são abstratos, apesar de essas características não estarem explícitas na declaração desses elementos.

Uma interface é implementada por uma classe por meio da palavra-chave implements, ao final da declaração da classe.

```
public class ClasseA implements InterfaceA {
   ...
}
```

O principal objetivo de uma interface é obrigar as classes que a implementaram a implementarem os seus métodos.

Algumas características da interface:

- uma interface somente possuirá métodos abstratos e/ou atributos finais (constantes);
- o nome de uma interface segue o mesmo padrão de nomes de classes (começa com letra maiúscula);
- uma classe pode implementar múltiplas interfaces:

```
public class ClasA implements InterfA, InterfB, InterfC {
   ...
}
```

No caso de implementações múltiplas de interfaces, a classe deverá implementar todos os métodos de todas as interfaces que ela está implementando.

Observação

Os métodos de uma interface não levam o modificador abstract, apesar de serem abstract. Já os atributos de uma interface não levam o modificador final, apesar de serem final.

6.6.1 Por que utilizamos interfaces?

Utilizamos interfaces para servirem de guias na implementação das regras de negócio das empresas.

Quando uma empresa possui uma ou mais regras de negócio que devem ser propagadas às diversas representações de seus produtos na forma de sistemas de informática, ao construirmos um sistema orientado a objeto, sempre que for criada uma classe que representa aqueles produtos, essas regras (métodos definidos na interface) deverão ser implementadas naquela classe. Isso, no mínimo, irá obrigar ao desenvolvedor conhecer (ou procurar conhecer) o produto no qual ele está trabalhando e suas particularidades, para depois então gerar suas representações no sistema.

Saiba mais

Saiba mais sobre interfaces no capítulo 8 de:

SCHILDT, H. *Java para iniciantes*: crie, compile e execute programas Java rapidamente. 6. ed. Porto Alegre: Bookman, 2015.

6.7 Classe JOptionPane

JOptionPane é uma classe da biblioteca do Java que possibilita a criação de uma caixa de diálogo padrão que ou solicita um valor para o usuário, ou retorna uma informação padrão, dependendo do método utilizado e da necessidade do desenvolvedor.

Quadro 3 – Métodos da classe JOptionPane

Métodos	Descrição
showConfirmDialog	Solicita uma confirmação como YES, NO, CANCEL
showInputDialog	Solicita algum valor (campo texto para preenchimento)
showMessageDialog	Mostra ao usuário uma informação
showOptionDialog	Unificação dos três tipos acima

Quadro 4 – Estilos de mensagens

Estilos da mensagem	Descrição
ERROR_MESSAGE	X num octógono
INFORMATION_MESSAGE	i num círculo
WARNING_MESSAGE	! num triângulo
QUESTION_MESSAGE	? num quadrado
PLAIN_MESSAGE	Sem ícone

Quadro 5 – Estilos de botões

Estilos do botão	Descrição
DEFAULT_OPTION	Botão OK
YES_NO_OPTION	Botões Sim e Não
YES_NO_CANCEL_OPTION	Botões Sim, Não e Cancelar
OK_CANCEL_OPTION	Botões OK e Cancelar

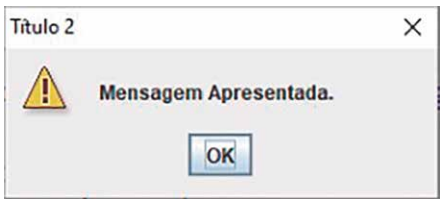
Quadro 6 – Tipos de respostas (ou retornos)

Tipos de respostas	Valor de retorno
`YES_OPTION` - (clicando em Sim)	0
`NO_OPTION` - (clicando em Não)	1
`CANCEL_OPTION` - (clicando em Cancelar)	2
`OK_OPTION` - (clicando em OK)	0
`CLOSED_OPTION` - (fechando a janela de mensagem, ou teclando em ESC, ou clicando no X que fecha a janela)	–1

Exemplos de utilização com sua respectiva visualização de tela de mensagem (observação: suponha JFrame frame = null;):

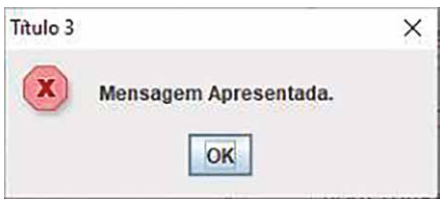

```
JOptionPane.showMessageDialog(frame,
      “Mensagem Apresentada.”,
      “Título 1”,
      JOptionPane.INFORMATION_MESSAGE);
```

Figura 23 – Tela de mensagem




```
JOptionPane.showMessageDialog(frame,
      “Mensagem Apresentada.”,
      “Título 2”,
      JOptionPane.WARNING_MESSAGE);
```

Figura 24 – Tela de mensagem

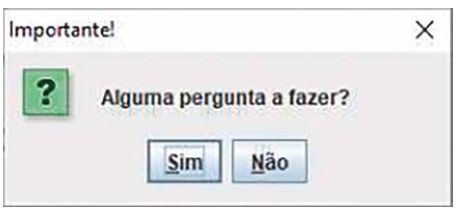



```
JOptionPane.showMessageDialog(frame,
      “Mensagem Apresentada.”,
      “Título 3”,
      JOptionPane.ERROR_MESSAGE);
```

Figura 25 – Tela de mensagem


```
JOptionPane.showMessageDialog(frame,
      “Mensagem Apresentada.”,
      “Título 4”,
      JOptionPane.PLAIN_MESSAGE);
```

Figura 26 – Tela de mensagem

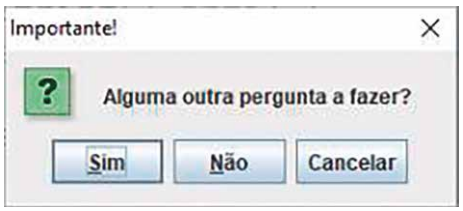



```
int n = JOptionPane.showConfirmDialog(
      frame,
      “Alguma pergunta a fazer?”,
      “Importante!”,
      JOptionPane.YES_NO_OPTION);
```

Figura 27 – Tela de mensagem

Observação

Clicando em Sim retorna 0, clicando em Não retorna 1.

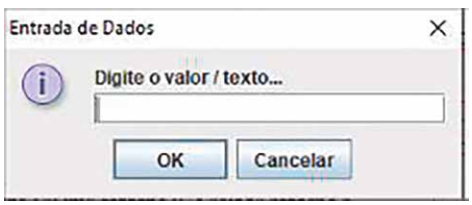



```
int n = JOptionPane.showConfirmDialog(
      frame,
      "Alguma outra pergunta a fazer?",
      "Importante!",
      JOptionPane.YES_NO_CANCEL_OPTION);
```

Figura 28 – Tela de mensagem

Observação

Clicando em Sim retorna 0, clicando em Não retorna 1, clicando em Cancelar retorna 2.




```
String s = JOptionPane.showInputDialog(
      frame,
      "Digite o valor / texto...",
      "Entrada de Dados",
      JOptionPane.INFORMATION_MESSAGE);
```

Figura 29 – Tela de mensagem

Nesse último exemplo, a string s receberá o texto digitado (independentemente do valor, se numérico ou alfanumérico). Se nada for digitado, ou apertar-se Cancelar, então s receberá nulo (null).

6.8 Wrapper classes

A palavra **wrapper** vem do inglês **wrap**, que significa **envolver**, e tem como principal função "envolver elementos" adicionando funcionalidades a eles.

O Java possui diversos wrappers que adicionam funcionalidades a outras classes, ou tipos primitivos. Há vários tipos, seja para tratar o fluxo de conexões, (ObjectInputStream), para fluxos de áudio (AudioInputStream), baseados em tipos primitivos (DataInputStream) ou para adicionar buffers (BufferedInputStream).

Com os wrappers de tipos primitivos consegue-se executar métodos como:

- parseTipo (utilizado na transformação de valores de strings);
- valueOf.

Quadro 7 – Relação entre os tipos primitivos e suas respectivas wrappers classes

Tipo primitivo	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

6.9 Transformando textos em números: utilizando as wrapper classes

É possível transformar textos (valores do tipo string) em números. Neste caso, transforma-se o valor de uma string em um tipo primitivo numérico a partir dos métodos da sua respectiva wrapper class, desde que os valores das strings representem efetivamente um valor do tipo numérico que se quer recuperar, de forma que caso não seja possível, uma exceção será gerada.

Exemplo:

```
String s1, s2, s3;
s1 = "27";
int i = Integer.parseInt(s1);
long l = Long.parseLong(s1);
s2 = "27.98";
float f = Float.parseFloat(s2);
double d = Double.parseDouble(s2);
//int i = Integer.parseInt(s2);
// este último geraria uma exceção pois o valor não é inteiro
s3 = "3956";
char c = s3.charAt(2); // pega o valor 5
int val = Character.getNumericValue(c);
```

Observação

É importante ficar atento quando se vai realizar a transformação de informações do tipo string para tipos numéricos, já que caso o valor da informação do tipo string não represente um valor numérico, o compilador acusará um erro que poderá interromper a execução do programa, parando-o imediatamente se não for tratado (o tratamento de exceções é um tópico explicado mais a frente neste livro-texto).

Também devemos ficar atentos ao fato de que, se tentarmos transformar valores numéricos do tipo real (double ou float) para variáveis do tipo inteiro (int, long etc.), também será gerado um erro que interrompe a execução do programa.

Para finalizar esta unidade, os exemplos de aplicação a seguir. As resoluções são apresentadas na sequência.

Exemplos de aplicação

1 – Criar a classe Pessoa (encapsulada) com as seguintes características:

Deve possuir os atributos:

- nome (String)
- endereco (String)

Deve possuir o método abstrato: cadastrarPessoa()

público e sem retorno

2 – Criar as classes PessoaFisica e PessoaJuridica (encapsuladas) com as seguintes características:

- devem herdar a classe Pessoa;
- devem ser classes finais.

A classe PessoaFisica deve possuir os atributos:

- cpf (String);
- idade (int).

A classe PessoaJuridica deve possuir os atributos:

- cnpj (String);
- nomeFantasia (String).

O método de cadastro de cada uma delas deverá permitir a inclusão de um valor em cada um de seus atributos (via JOptionPane – com input box).

3 – Criar uma classe abstrata de funções especiais com strings cujo nome deverá ser "StrEspecial".

Essa função deverá ter três métodos estáticos:

trimGeral(...)

Esta função deverá receber uma string e retornar a mesma string, porém tendo sido retirados os espaços iniciais e finais a mais (se houver) e todo e qualquer espaço a mais (extras) internamente à string.

Por exemplo, se receber a frase:

" Olá Mundo ! "

Deverá retornar:

"Olá Mundo !"

Perceba que os espaços iniciais e finais (que não deviam existir) foram totalmente retirados, e os espaços extras na parte interna se transformaram em espaço único.

trimToUpper(..)

Esta função deverá passar a string recebida pelo método anterior (trimGeral(...)) e depois transformar tudo em maiúsculo.

trimToLower(..)

Esta função deverá passar a string recebida pelo método anterior (trimGeral(...)) e depois transformar tudo em minúsculo.

Observação

Para trabalhar com a string, dê uma olhada no item "Trabalhando com strings" deste livro-texto.

Por fim, criar uma classe TesteStr com o método main, que deverá testar o texto a seguir:

" Olá Mundo ! "

E imprimi-lo das quatro seguintes formas:

- formato inicial (como o texto que acabamos de mostrar);
- "passado" pelo método trimGeral(...);

- "passado" pelo método trimToUpper(...);
- "passado" pelo método trimToLower(...).

1 –

Classe Pessoa:

```
public abstract class Pessoa {
   private String nome;
   private String endereco;
   public abstract void cadastrarPessoa();
   public String getNome() {
     return nome;
   }
   public void setNome(String nome) {
     this.nome = nome;
   }
   public String getEndereco() {
     return endereco;
   }
   public void setEndereco(String endereco) {
     this.endereco = endereco;
   }
}
```

2 –

Classe Pessoa Física:

```
import javax.swing.JOptionPane;
public class PessoaFisica extends Pessoa {
   private String cpf;
   private int idade;
   public void cadastrarPessoa() {
     this.setNome(JOptionPane.showInputDialog
   ("Digite o Nome:"));
     this.setEndereco(JOptionPane.showInputDialog
   ("Digite o Endereço:"));
     cpf = JOptionPane.showInputDialog("Digite o CPF:");
     idade = Integer.parseInt(JOptionPane.showInputDialog
   ("Digite a Idade:"));
   }
```

```
    public String getCpf() {
      return cpf;
    }
    public void setCpf(String cpf) {
      this.cpf = cpf;
    }
    public int getIdade() {
      return idade;
    }
    public void setIdade(int idade) {
      this.idade = idade;
    }
}
```

Classe Pessoa Jurídica:

```
import javax.swing.JOptionPane;
public class PessoaJuridica extends Pessoa {
    private String cnpj;
    private String nomeFantasia;
    public void cadastrarPessoa() {
      this.setNome(JOptionPane.showInputDialog
    ("Digite o Nome:"));
      this.setEndereco(JOptionPane.showInputDialog
    ("Digite o Endereço:"));
      cnpj = JOptionPane.showInputDialog("Digite o CNPJ:");
      nomeFantasia = JoptionPane.showInputDialog
    ("Digite o Nome Fantasia:");
    }
    public String getCnpj() {
      return cnpj;
    }
    public void setCnpj(String cnpj) {
      this.cnpj = cnpj;
    }
    public String getNomeFantasia() {
      return nomeFantasia;
    }
    public void setNomeFantasia(String nomeFantasia) {
      this.nomeFantasia = nomeFantasia;
    }
}
```

3 –

Classe StrEspecial:

```
public abstract class StrEspecial {
   // Método estático que retira espaços extras
   public static String trimGeral (String txt) {
     // retirando espaços iniciais e finais
     String auxTxt = txt.trim();
     // retirando espaços extras internos a String
     String espExt = "  ";
     String espFin = " ";
     int posic = auxTxt.indexOf(espExt);
     while (posic >= 0) {
              auxTxt = auxTxt.replaceAll(espExt, espFin);
              posic = auxTxt.indexOf(espExt);
     }
     return auxTxt;
   }
   // Método estático que deixa tudo em
   // maiúsculo e retira espaços extras
   public static String trimToUpper (String txt) {
     // retirando espaços gerais
     String auxTxt = trimGeral(txt);
     // deixando todo o texto em MAIÚSCULO
     auxTxt = auxTxt.toUpperCase();
     return auxTxt;
   }
   // Método estático que deixa tudo em
   // minúsculo e retira espaços extras
   public static String trimToLower (String txt) {
     // retirando espaços gerais
     String auxTxt = trimGeral(txt);
     // deixando todo o texto em minúsculo
     auxTxt = auxTxt.toLowerCase();
     return auxTxt;
   }
}
```

Classe TesteStr:

```
public class TesteStr {
   public static void main(String[] args) {
     String txt = "    Olá    Mundo    !    ";
```

```
        System.out.println("txt0 = [" + txt + "]");
        txt = StrEspecial.trimGeral(txt);
        System.out.println("txt1 = [" + txt + "]");
        txt = StrEspecial.trimToLower(txt);
        System.out.println("txt2 = [" + txt + "]");
        txt = StrEspecial.trimToUpper(txt);
        System.out.println("txt3 = [" + txt + "]");
    }
}
```

Resumo

Nesta unidade, vimos inicialmente que polimorfismo é um recurso da orientação a objetos que permite que mais de um elemento possa estar caracterizado de igual forma, mas cada um deles com comportamentos diferentes.

Já tínhamos visto que é possível o polimorfismo de métodos (sobrescrita ou sobrecarga). Nesta unidade, estudamos mais um tipo de polimorfismo, o polimorfismo de variáveis, em que duas variáveis podem coexistir com o mesmo nome, mas comportamentos diferentes (uma com comportamento global e outra com comportamento local), cuja diferença estará na utilização do termo this, de forma que a variável com esse termo será a variável global e a que estiver sem esse termo será a variável local.

```
variavel = valor1;  // esta é uma variável local
this.variavel = valor2;  // esta é uma variável global
```

Na linguagem Java existem alguns modificadores que alteram o comportamento ou a forma com que trabalhamos os elementos. Esses modificadores de comportamento permitem um controle da maneira com que uma equipe de desenvolvimento se utiliza de cada um daqueles elementos. São eles: o modificador final (para classes, métodos e atributos), sendo que uma classe final é uma classe que não pode ser herdada, um método final é um método que não pode ser sobrescrito e um atributo final é uma constante que não pode ter seu valor alterado; o modificador static (para métodos e atributos), o qual faz com que o elemento possa ser acessado sem que a classe a que pertence precise ser instanciada; e o modificador abstract (para classes e métodos), sendo que uma classe abstrata é uma classe que não pode ser instanciada.

Um método abstrato é um método sem implementação cuja declaração termina com ponto-e-vírgula que exige que a classe a que pertence seja

também abstrata e que exige a sua implementação nas subclasses da classe a que pertence.

Uma interface é um elemento que, assim como as classes, pode possuir atributos e métodos, porém todos os seus métodos são abstratos e todos os seus atributos são finais e estáticos (constantes). Os métodos de uma interface não levam o modificador abstract (apesar de serem abstratos) e os seus atributos não levam o modificador final nem static (apesar de serem finais e estáticos).

Uma classe implementa uma interface utilizando-se do termo implements, e ela pode implementar múltiplas interfaces, utilizando-se da seguinte sintaxe:

```
public class ClasA implements InterfA, InterfB, InterfC {
        ...
}
```

A classe que implementa uma ou mais interfaces deve implementar todos os métodos de todas as interfaces que ela está implementando.

Por sua vez, a classe JOptionPane permite que mostremos janelas de pop-up de mensagens onde aparecem os botões OK, Cancelar, Sim e Não alternadamente, além de ícones padrões de mensagens de tela. Essa classe possui os métodos showMessageDialog(), showConfirmDialog() e showInputDialog(), que nos permite variar entre pop-ups de simples mensagens, pop-ups que retornam um número inteiro equivalente ao botão apertado e pop-ups que retornam o texto digitado em seu campo de inclusão.

Existem algumas wrapper classes que representam tipos primitivos, como as classes: Integer (que representa o tipo int), Short, Long, Double, Float e Character (que representa o tipo char). Uma das utilidades dessas classes é a realização do parse, ou seja, a transformação de um valor do tipo string para o tipo primitivo da linguagem Java que representa.

Exercícios

Questão 1. (FCC/2019) Considere as linhas de um pseudocódigo abaixo, levando em conta que a variável consumo_mes tenha sido declarada anteriormente para receber valores inteiros:

```
[...]
Início
  Escreva ("Entre com o valor consumido:")
  Leia (consumo_mes)
    [...]
Fim
```

Usando Java SE em uma aplicação com as bibliotecas necessárias devidamente importadas, essas linhas podem ser substituídas por:

A) consumo_mes = Integet.toInt(Dialog.showDialogBox("Entre com o valor consumido:"));

B) consumo_mes = Integer.parseInt(JOptionPane.showInputDialog(null, "Entre com o valor consumido:"));

C) consumo_mes = Integer.toInteger(Dialog.input("Entre com o valor consumido:"));

D) consumo_mes = Int.parseInt(Console.input("Entre com o valor consumido:"));

E) consumo_mes = Int.parse(Console.write("Entre com o valor consumido:"));

Resposta correta: alternativa B.

Análise da questão

A classe JOptionPane permite que mostremos ao usuário janelas do tipo pop-up. Dentre os métodos dessa classe, temos o showInputDialog(), que solicita um valor ao usuário e retorna o texto digitado como um valor do tipo string. Porém, de acordo com o enunciado, a variável consumo_mes deve receber valores inteiros. É possível transformar textos (valores do tipo string) em valores de tipos numéricos utilizando wrapper classes. No caso, o comando Integer.parseInt() realiza essa modificação, utilizando o método parseInt() da classe Integer. Dessa forma, por meio do trecho

```
consumo_mes = Integer.parseInt(JOptionPane.showInputDialog(null, "Entre com o valor consumido:"));
```

a variável consumo_mes receberá um valor do tipo inteiro, correspondente ao texto digitado pelo usuário na janela pop-up.

Questão 2. (Idecan/2020. Adaptada) A orientação a objetos (OO) é um paradigma de programação para o qual "tudo é um objeto", sendo Java uma das principais linguagens que implementam esse paradigma.

Em relação à linguagem Java e à OO, avalie as afirmativas:

I – Uma classe Java pode implementar mais de uma interface Java.

II – Uma classe Java abstrata, obrigatoriamente, deve possuir um ou mais métodos abstratos.

III – Uma classe Java declarada como final não pode ser herdada (não pode ter subclasses Java).

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) I e III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: em Java, uma interface é um recurso utilizado para "obrigar" as classes que a implementam a implementarem, também, os seus métodos. Interfaces podem servir como guias na implementação das regras de negócio de empresas, por exemplo. Uma classe pode, sim, implementar mais de uma interface. Nesse caso, a classe em questão deverá implementar todos os métodos de todas as interfaces.

II – Afirmativa incorreta.

Justificativa: uma classe abstrata é uma classe que não pode ser instanciada. Ela pode ser herdada por subclasses, mas não pode, diretamente, instanciar objetos. Um método abstrato só pode existir em uma classe abstrata, mas uma classe abstrata não precisa, necessariamente, ter métodos abstratos.

III – Afirmativa correta.

Justificativa: uma classe final é uma classe que não pode ser herdada. Isso quer dizer que, em uma cadeia de heranças, uma classe final será a última classe da cadeia. Dessa forma, ela não pode ter subclasses.