Unidade II

3 LIDANDO COM AS VARIÁVEIS E SEUS ACESSOS

3.1 Promotion e casting

Nesta unidade, inicialmente, veremos como a linguagem Java trabalha com seus valores numéricos, quando misturamos tipos primitivos. Em seguida, iremos nos aprofundar na Teoria de Orientação a Objeto.

Veremos que o objetivo de todo esse conteúdo está relacionado ao processo de programação e a como devemos pensar a construção de um sistema para que ele atinja, da melhor forma possível, as necessidades identificadas pelas regras de negócio definidas pelo cliente.

Ao programarmos um sistema, muitas vezes trabalhamos em equipe. Devemos ter em mente que esse sistema que estamos construindo poderá ser alterado, corrigido ou implementado por diversos outros programadores e que futuramente poderão dar continuidade a ele. Assim, utilizamos os vários recursos de programação existentes na orientação a objetos (que veremos neste livro-texto), com o intuito de determinar uma linha de ação e de compreensão do sistema às possíveis futuras equipes que venham a trabalhar nesse (ou com esse) mesmo sistema.

Quando trabalhamos com linguagens fortemente tipadas, devemos tomar alguns cuidados ao misturarmos tipos de dados diferentes em algumas ocasiões. Isso é muito comum quando lidamos com fórmulas matemáticas, em que misturamos tipos inteiros e tipos reais em uma mesma equação. Para algumas situações, o sistema se resolve automaticamente, mas para outras precisamos forçar a alteração da tipagem dos dados, a fim de que os cálculos ocorram corretamente. A ideia é que o compilador trabalha com variáveis de valores numéricos diversos, utilizando quantidades de memória e processamentos diferenciados, de acordo com a tipagem definida para aquela variável.

O tipo numérico que se utiliza de menos memória e menos processamento é o tipo byte e o que se utiliza de mais memória, além de mais processamento, é o tipo double.

A **promotion** é a "elevação" dos tipos numéricos para um tipo que pode representar uma maior grandeza.

As regras definem que um valor ou uma variável de tipo primitivo de menor capacidade numérica pode ser utilizado no lugar em que seria utilizado um valor ou uma variável de tipo com maior capacidade numérica.

A promotion não provoca perda de valores. Ela é feita implicitamente (automaticamente) pelo compilador. Assim, quando reservamos um espaço maior e preparamos o sistema para trabalhar com

um determinado tipo de dado, caso tenhamos um tipo de dado inferior, isso não alterará em nada o processamento previsto. Isso tudo significa que para o compilador trabalhar com um dado que possua um tipo inferior ao que foi reservado, o compilador fará a "promoção" automática daquela informação e a processará no mesmo espaço de memória que já estava reservado para isso, o que significa que a promoção acontece automaticamente.

Portanto, a promotion é implícita (automática) e ocorre ao trabalharmos com tipos "menores" quando poderíamos trabalhar com tipos "maiores". A ordem sequencial de promotion (do tipo de menor capacidade numérica para o de maior capacidade) é como segue:

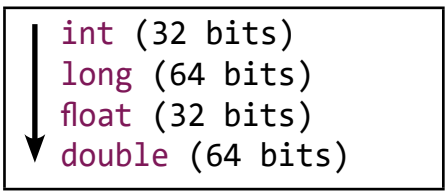

Figura 17

Exemplos de promotions:

```
int f = 4;
long g = 5L + f; //promotion do valor de f, que é int; para long
float h = 3.4f + g; //promotion do valor de g, que é long, para float
double i = 2.5 + h; //promotion do valor de h, que é float, para double
```

Observação

Devemos entender que a promoção ocorre para o valor da variável, e não para a variável (o tipo da variável não se altera).

O **casting** é a mudança de um tipo para outro de forma explícita (forçada) por meio do operador de casting (ou seja, o "nome do tipo" entre parênteses).

Para que possamos trabalhar com tipos que necessitam de memória de processamento maiores do que aquele que foi reservado inicialmente (na declaração do tipo), teremos antes que transformar aquela informação para que ela possa ser trabalhada naquele espaço que foi reservado (que é menor). Essa transformação pode causar a perda de dados (perda de valores ou de precisão), por isso deve ser determinada explicitamente pelo desenvolvedor.

Quando ocorre algum tipo de "erro" nesse sentido, o compilador pode, durante a compilação, detectar a impossibilidade da operação (não realizando-a). Do contrário, não podendo detectá-la no código, poderá ocorrer um erro em tempo de execução, caso a conversão do valor não seja possível de realizar.

Assim, a demotion (que chamamos de **casting**) é uma espécie de "rebaixamento" do tipo numérico para um tipo que representa uma menor grandeza.

A ordem sequencial de demotion (de um tipo de maior capacidade numérica para um de menor capacidade) é como segue:

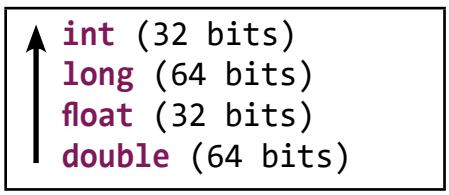

Figura 18

Exemplo de casting:

```
double j = 2.5;
float k = 5.2f + (float)j; //casting para float
long l = 3L + (long)k; //casting para long
int m = 7 + (int)l; //casting para int
short n = (short)m; //casting para short
byte o = (byte)n; //casting para byte
```

Lembrete

Um casting deve ser utilizado sempre que atribuímos a um tipo de menor abrangência um valor de um tipo de maior abrangência.

Assim como a promoção ocorre sobre um valor, o casting também ocorrerá sobre o valor da variável, e não sobre a variável (o tipo da variável não se altera).

3.2 Modificadores de acesso

Os modificadores de acesso são termos que utilizamos junto à declaração das classes, dos atributos e dos métodos e que modificam a forma de acessarmos esses elementos por meio dos códigos de programas.

Vimos que um sistema gerado em linguagem Java é baseado inteiramente na programação de classes e que uma classe possui, basicamente, atributos (dados) e métodos (ações). Vimos também que cada classe compilada é um arquivo com extensão .class.

Quando criamos um sistema, nele é gerada uma infinidade de classes, de forma que diversos arquivos são gerados e utilizados na sua construção. Dessa forma, podemos organizar nosso sistema em pacotes, ou packages (que veremos mais adiante), e em cada pacote inserimos as classes que pertencem a uma

determinada categoria. Dessa forma, podemos definir se classes de diversos pacotes podem acessar-se mutuamente ou se uma determinada classe (ou seus elementos) foi construída para ser acessada apenas por um determinado grupo de classes, e não por outros.

Para controlarmos os acessos das classes, utilizamos quatro modificadores de acesso disponibilizados, que são:

- public;
- private;
- protected;
- (default) – ou sem modificador.

Observação

Esses modificadores de acesso controlam não só o acesso à alteração do valor da variável, mas também o acesso à leitura desse valor.

O modificador public (público)

Uma declaração (de classes, atributos ou métodos) com o modificador public indica que o elemento pode ser acessado de qualquer lugar e por qualquer entidade (objeto) que possa visualizar a classe a que esse elemento pertence.

Exemplo:

```
public int idadePessoa;
public void imprimirDados() {...}
```

O modificador private (privado)

Uma declaração (de atributos ou métodos) com o modificador private indica que o elemento não pode ser acessado (visualizado) ou utilizado por nenhuma outra classe, a não ser apenas por métodos da própria classe a que pertence.

Exemplo:

```
private String nomeAluno;
private double calcularPeso() {...}
```

O modificador protected (protegido)

O modificador protected torna o elemento (atributo ou método) acessível às classes do mesmo pacote ou às classes relacionadas por herança (que veremos mais adiante neste curso). Seus elementos não são acessíveis a outras classes fora do pacote em que foram declarados.

Exemplo:

```
protected JButton b01;
protected String getEnderecoCliente() {...}
```

O (default) (padrão ou sem modificador)

Uma declaração (de classes, atributos ou métodos) em que não há definição de modificador possui o que chamamos de modificador padrão (default), fazendo com que sejam acessíveis somente por classes do mesmo pacote.

Exemplo:

```
class Principal { ... }
```

Ou, ainda:

```
boolean permiteProsseguimento;
double verificarAlturaRelativa() {...}
```

A tabela de níveis de acesso resume os itens explicativos sobre os modificadores de acesso:

Quadro 1 – Tabela de níveis de acesso

Modificador	Classe	Package (pacote)	Subclasse (classe filha)	Qualquer classe
public	Sim	Sim	Sim	Sim
protected	Sim	Sim	Sim	Não
(default)	Sim	Sim	Não	Não
private	Sim	Não	Não	Não

4 ELEMENTOS CARACTERÍSTICOS DA ORIENTAÇÃO A OBJETO

Alguns conceitos formam o que chamamos de bases da orientação a objetos. Veremos neste item alguns de seus elementos.

4.1 Encapsulamento

O encapsulamento, considerado um dos pilares da orientação a objetos, é um recurso em que "protegemos" os atributos de uma classe, fazendo com que todo o controle de acesso à leitura ou alteração de seu valor seja feito por meio dos métodos da classe.

Para termos uma ideia de como esse recurso é comum e de grande importância na programação, vários frameworks se utilizam dele na sua funcionalidade.

Observação

Um framework é um conjunto de códigos, classes e bibliotecas de classes, geralmente utilizado como ferramenta para o desenvolvimento de sistemas e criado para solucionar problemas comuns. Existem muitos frameworks criados para programação em linguagem Java e que podem agilizar o processo de construção de softwares, diminuindo o tempo de desenvolvimento e, consequentemente, o custo do projeto. Alguns frameworks Java são muito conhecidos, como o Hibernate (para persistência de dados), o Spark (para desenvolvimento de serviços web) e o Spring (que dispõe uma série de recursos que simplificam o desenvolvimento de aplicações Java).

Basicamente, para que uma classe seja encapsulada, basta que todos os seus atributos fiquem com o modificador de acesso private e que se gerem os métodos setters e getters para cada um deles.

- Os métodos setters são os métodos que vão alterar o valor dos atributos (sem retorno de valor).
- Os métodos getters são os métodos que vão retornar o valor dos atributos (com retorno de valor).

No encapsulamento, o nome do método deve iniciar com a palavra set ou get (em minúsculo) e mais o nome exato do atributo iniciado com maiúsculo (setAtributo ou getAtributo).

A sintaxe padrão desses métodos é a seguinte:

```
public void setAtributo (tipoAtributo variavel) {
   atributo = variavel;
}
public tipoAtributo getAtributo () {
   return atributo;
}
```

Exemplo de um encapsulamento: encapsulando a classe Pessoa (que já possui os atributos nome e idade):

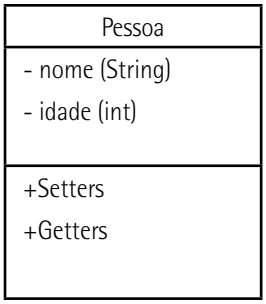

Figura 19 – Representação (UML) da classe Pessoa encapsulada

```
public class Pessoa {
   private String nome;
   private int idade;
     // Métodos Setters e Getters
   public void setNome (String s) {
     nome = s;
   }
   public String getNome () {
     return nome;
   }
   public void setIdade (int i) {
     idade = i;
   }
   public int getIdade () {
     return idade;
   }
}
```

Observação

Os métodos getters e setters são públicos, enquanto os atributos da classe são privados.

Na figura anterior, os elementos privados são representados iniciando-se com o sinal - (menos) e os elementos públicos são representados iniciando-se com o sinal + (mais). Essa é uma das características da UML.

4.2 Sobrecarga de métodos

A **assinatura** de um método é o conjunto formado pelo seu nome e pelos seus parâmetros. Assim, na declaração do método temos:

```
modificador(es) tipoDeRetorno nomeDoMetodo(Parâmetros) {
    (assinatura do método)
}
```

Sabe-se que uma classe não permite que haja dois métodos com a mesma assinatura. Sendo assim, quando há, numa mesma classe, dois ou mais métodos com o mesmo nome, ambos devem obrigatoriamente estar diferenciados em seus parâmetros, caracterizando uma sobrecarga do método.

Desse modo, **sobrecarga de métodos** é um tipo de polimorfismo caracterizado pela existência de mais de um método com mesmo nome em uma mesma classe (ou em classes que caracterizam herança – conceito que veremos adiante), diferenciados em seus parâmetros (na **quantidade** ou na **tipagem** dos parâmetros).

Exemplo: classe Calculadora com três métodos de nome "somar", porém diferenciados nos parâmetros:

```
// Um exemplo de Sobrecarga de Métodos (do método "somar")
public class Calculadora {
   // Método "somar" com 2 parâmetros do tipo "int"
   public int somar (int a, int b) {
     return (a + b);
   }
   // Método "somar" com 3 parâmetros do tipo "int"
   public int somar (int a, int b, int c) {
     return (a + b + c);
   }
   // Método "somar" com 2 parâmetros do tipo "double"
   public double somar (double a, double b) {
     return (a + b);
   }
}
```

Percebe-se que não há problema dos três métodos ("somar") coexistirem numa mesma classe, já que há diferenças na definição dos seus parâmetros, seja na quantidade de parâmetros, seja nos tipos nele existentes.

4.3 Método construtor

A instância de uma classe é a geração do objeto que a representa na memória RAM. Sabe-se que podemos gerar várias instâncias a partir de uma mesma classe, de modo que cada objeto gerado conterá as mesmas características (mesmos tipos de informações, podendo realizar as mesmas ações definidas pela classe), porém contendo, na maioria das vezes, informações diferentes.

Um método construtor é o método que é acionado toda vez que uma classe é instanciada, e toda classe possui um método construtor.

Quando **não** criamos um método construtor explicitamente em uma classe, mesmo assim ela possuirá um método construtor padrão sem parâmetros do qual o compilador se utilizará para gerar a classe em memória (instanciar a classe gerando o objeto).

O objetivo do método construtor é dar as características iniciais de um objeto assim que ele é gerado na memória. Podemos, com o método construtor de uma classe, tanto dar valores iniciais aos seus atributos quanto preparar uma situação inicial (rodando outros métodos, inclusive de outras classes) para que ele opere de acordo com o exigido pelo sistema.

As características de um método construtor são:

- ele deve possuir exatamente o mesmo nome da classe;
- ele não deve possuir qualquer definição de tipo de retorno;
- ele deve ser público.

Sua sintaxe é:

```
public NomeDaClasse (parametros) {
   ...
}
```

Observação

Uma vez determinados um ou mais métodos construtores para uma classe, para que ela seja instanciada deve-se utilizar um dos métodos construtores nela definidos.

Uma classe pode possuir mais de um método construtor (caracterizando, nesse caso, sobrecarga de métodos construtores).

Exemplo:

```
public class Pessoa {
   // atributos da Classe Pessoa
   private String nome;
   private int idade;
   private double altura;
   // #####  Métodos Construtores (MC)  #####
   // MC com apenas um parâmetro
   public Pessoa(String s) {
     nome = s;
```

```
    }
    // MC com três parâmetros equivalentes a seus atributos
    public Pessoa(String s, int i, double d) {
      nome = s;
      idade = i;
      altura = d;
    }
}
```

A classe Pessoa do exemplo que acabamos de apresentar possui dois métodos construtores, de modo que, para instanciar um objeto a partir dessa classe, o programador deverá se utilizar de um desses dois métodos construtores, assim como visto na sintaxe a seguir:

```
nomeClasse nomeObjeto = new MetodoConstrutor();
```

Ou, ainda:

```
nomeClasse nomeObjeto;
...
nomeObjeto = new MetodoConstrutor();
```

O último termo da declaração da instância é uma chamada a um método construtor, que muitas vezes é apenas o método construtor padrão quando ele não é especificado na classe.

4.4 Herança

Assim como o encapsulamento, a herança é considerada um dos pilares da orientação a objetos, em que se caracteriza a especialização e generalização de classes.

A herança é um recurso de orientação a objetos que permite a criação de novas classes a partir das características de outra classe já criada. Assim, há um reaproveitamento de código, de modo que uma classe herdará todos os métodos e todos os atributos de outra classe.

Desta forma, as classes que herdam as características de outras classes são chamadas de **subclasses** (ou classes filhas) e as classes já existentes mas que deram origem às subclasses são chamadas de **superclasses** (classes mãe). Gera-se então uma hierarquia de classes, criando assim classes mais genéricas (as superclasses) com as características mais gerais e classes mais específicas (as subclasses) com as características mais individuais.

Na herança em Java, uma classe filha pode herdar apenas uma classe mãe, já que em Java não há herança múltipla. Essa classe filha possuirá todos os atributos e todos os métodos da classe mãe, independentemente do modificador de acesso que cada um deles tiver. Imaginando que a classe mãe herde elementos de outra classe, se uma classe a herdar, tornando-se filha, herdará também tudo o que as superclasses da primeira tiver. Para que uma classe herde outra classe, deve-se inserir na declaração da primeira (classe filha) a palavra extends seguida do nome da classe que ela está herdando (classe mãe).

É importante se ter em mente que, pelo conceito de herança, quando uma classe herda outra classe (quando uma classe filha herda uma classe mãe), ela passa a "ser" essa outra classe (assim, a classe filha é a classe mãe).

Vejamos um exemplo de herança. Imaginemos que vamos criar as classes para controlar a relação comercial de uma empresa em que existem os funcionários e os clientes (pessoas físicas). Podemos com isso criar as duas classes (classe Funcionario e classe Cliente), de forma que ambas terão atributos e métodos em comum, além dos seus atributos e métodos individuais. Dessa forma, pode-se fazer com que as características comuns (gerais) fiquem em uma classe mais genérica (por exemplo, a classe pessoa) e as características individuais (específicas) permaneçam nas próprias classes filhas, especificando-as.

Podemos identificar, neste caso, os conceitos de generalização e especificação das características dessas classes, de forma a termos o seguinte cenário:

A classe Pessoa:

```
public class Pessoa {
    private String nome;
    private String cpf;
    //... e demais características comuns
}
```

A classe Funcionario, que herda a classe Pessoa:

```
public class Funcionario extends Pessoa {
    private String funcional;
    private double salario;
    //... e demais características específicas
}
```

A classe Cliente, que herda a classe Pessoa:

```
public class Cliente extends Pessoa {
    private String projeto;
    private String areaNegocio;
    //... e demais características específicas
}
```

Observação

Pelos conceitos que acabamos de apresentar, ambas as classes (Funcionario e Cliente) conterão os atributos nome e cpf existentes na classe Pessoa, assim como podemos dizer que Funcionario é Pessoa e Cliente é Pessoa.

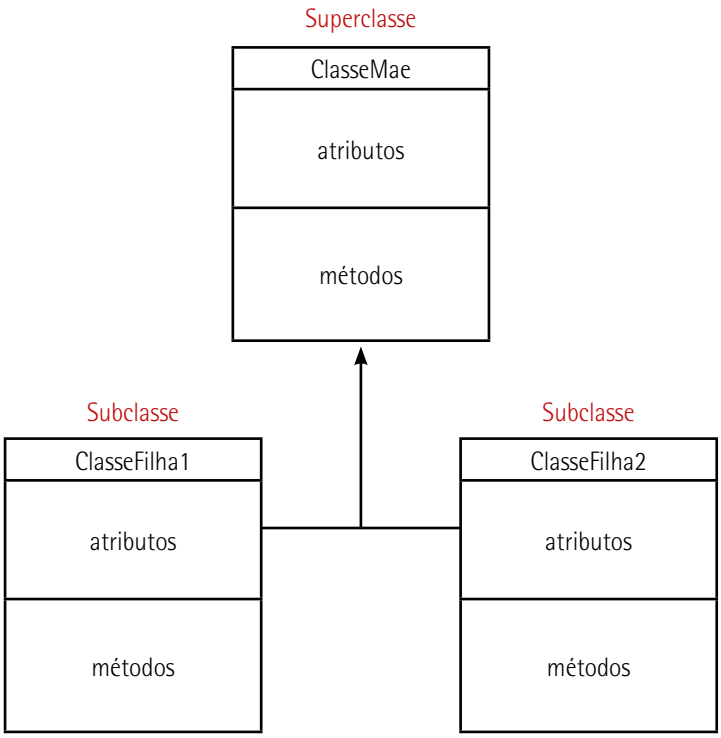

Figura 20 – Representação de herança com uma superclasse e duas subclasses

A linguagem Java não admite **herança múltipla** (uma subclasse herdando várias superclasses ao mesmo tempo), diferentemente de outras linguagens (como C++ e Python).

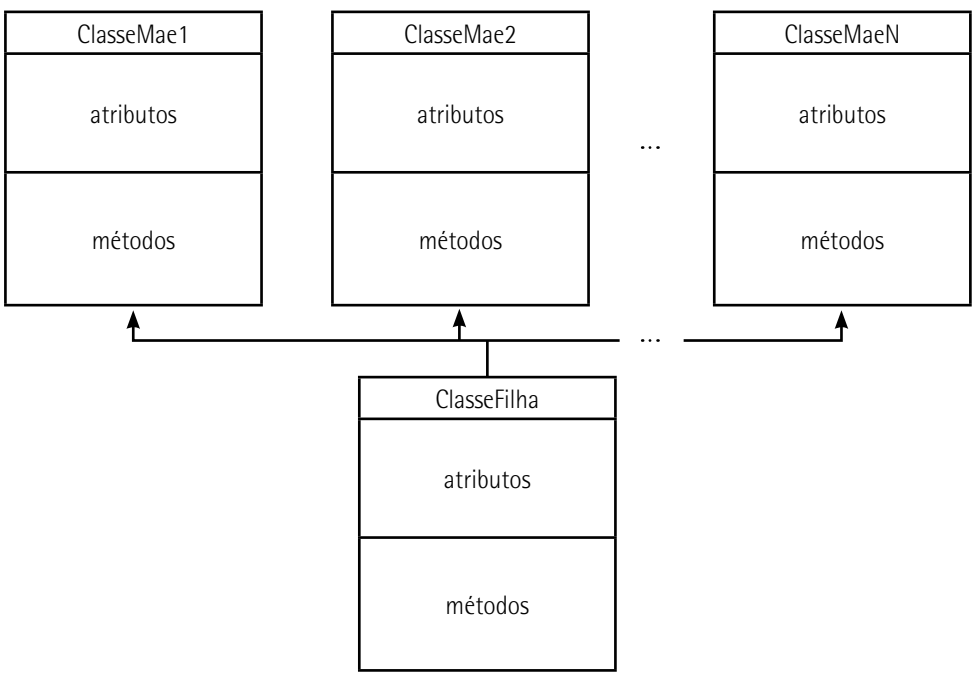

Figura 21 – Representação de herança múltipla para linguagens como C++ e Python

Apesar de não ser possível se fazer uma herança múltipla, o Java permite o que chamamos de **herança encadeada**, em que uma classe herda outra classe, que por sua vez herda outra classe, e assim por diante (veja a imagem a seguir).

Figura 22 – Representação de herança encadeada permitida pelo Java, em que a ClasseA herda a ClasseB, que herda a ClasseC

Nessa última representação, temos que a ClasseA conterá **todos** os elementos (atributos e métodos) das classes ClasseB e ClasseC. A classe ClasseB foi herdada diretamente pela ClasseA, e a classe ClasseC por encadeamento pela ClasseA.

Observação

Na linguagem Java, a classe "mãe de todas as classes" é a classe Object. Quando criamos uma classe em Java, mesmo que não explicitemos uma herança (com o termo extends), ela automaticamente irá herdar a classe Object.

Saiba mais

Saiba mais sobre herança em Java no capítulo 8, itens 8.2 e 8.4, da obra a seguir:

BARNES, D. J.; KÖLLING, M. *Programação orientada a objetos com Java*: uma introdução prática utilizando o BlueJ. São Paulo: Pearson Prentice Hall, 2004.

4.5 Sobrescrita de métodos

Uma classe filha (ou subclasse) herdará todos os métodos da classe mãe (ou superclasses). Dessa forma, se instanciarmos uma classe filha, podemos acionar os métodos existentes na classe mãe, desde que eles não sejam privados. No entanto, algumas vezes precisamos que aquele método tenha uma ação diferente, para a classe filha, daquela prevista e codificada na classe mãe, a fim de permitir que a subclasse tenha um comportamento mais específico e individual do que aquele esperado e definido pelo método da superclasse.

Para isso, recriamos na subclasse aquele ou aqueles métodos da superclasse com a mesma assinatura, porém implementados com as alterações específicas ao "negócio" da própria subclasse. Assim, ao instanciarmos a classe filha e acionarmos aquele método, acionaremos o método da classe filha, e não o da classe mãe.

Deve-se lembrar que, depois de reescrito, não é mais possível, a partir de uma outra classe, acionarmos diretamente, para um objeto gerado com a classe filha, aquele método que pertence à classe mãe.

Observação

Somente é possível invocar o método sobrescrito da classe mãe invocando-o a partir de um objeto da classe mãe, ou de dentro da classe filha (utilizando a palavra super).

Assim, sobrescrita de métodos é a existência de métodos com exatamente a mesma assinatura, tanto na superclasse quanto na subclasse, de forma que o método na subclasse estará sobrescrevendo (substituindo) o método da superclasse.

Lembrete

A assinatura de um método é a parte da declaração que comporta o nome do método e seus parâmetros.

Exemplo:

A classe Funcionario:

```
public class Funcionario {
   public double salario;
   //... e demais atributos
   public void mostraSalario () {
     System.out.println("Salário: " + salario);
   }
   public double calculaBonificacao() {
     double bonif = salario * 0.0165;
     return bonif;
   }
}
```

A classe Gerente herda a classe Funcionario:

```
public class Gerente extends Funcionario {
   // para o Gerente, a bonificação é
   // maior que a dos Funcionários
   public double calculaBonificacao() {
     double bonif = salario * 0.0205;
     return bonif;
   }
}
```

Assim, a partir de uma instância da classe Gerente podemos rodar os métodos mostraSalario() e calculaBonificacao(). Porém, no caso desse último método, o que será acionado será o da própria classe Gerente, e não aquele da classe Mãe, substituindo-o.

Exemplo de aplicação

De acordo com os conceitos de orientação a objetos, podemos afirmar que:

I – Utilizamos o encapsulamento para criar os métodos construtores que encapsulam os objetos em memória.

II – Métodos construtores são aqueles que geram os objetos em memória e somente são acionados no momento da instância da classe, sendo que uma classe pode conter vários métodos construtores, o que caracterizaria uma sobrecarga de métodos.

III – Herança é um recurso que permite a generalização e especificação das classes, caracterizando também o reaproveitamento de código.

IV – Sabendo que não pode haver mais de um método com mesma assinatura em uma classe, o fato de uma subclasse "ser" uma superclasse impede também que haja na subclasse um método que possua a mesma assinatura de um método da superclasse.

V – Uma classe chamada ClasseA contém um atributo xyz de um determinado tipo, mas sem o modificador na sua declaração. Sabendo que a ClasseA está no pacote pac01, então a ClasseB, que está no pacote pac02, não poderá acessar diretamente o atributo xyz, a não ser por meio de métodos da ClasseA.

Quais das afirmações são corretas?

A) Somente I, III e IV.

B) Somente II, III e V.

C) Somente II, III e IV.

D) Somente I, II e III.

E) Somente I, IV e V.

Resposta correta: alternativa B.

Resolução

I – Afirmativa incorreta.

Justificativa: utilizamos encapsulamento para garantir um controle de acesso aos atributos da classe, fazendo com que eles só possam ser acessados a partir de métodos da própria classe.

II – Afirmativa correta.

Justificativa: devemos sempre lembrar que quando estamos instanciando uma classe, acionamos o seu método construtor. Caso criemos explicitamente métodos construtores no corpo da classe, somos obrigados a utilizar algum deles para gerar a sua instância, de forma que uma classe pode ter sobrecarga de métodos construtores, ou seja, mais de um método construtor, diferenciados entre si por seus parâmetros, seja na quantidade deles, seja no tipo de pelo menos um deles.

III – Afirmativa correta.

Justificativa: na herança de classes temos uma classe que é mais genérica (a superclasse), cujos elementos pertencerão a todas as subclasses que a herdam, e temos também as classes mais específicas (as subclasses), que, além de possuírem todos os elementos da superclasse (atributos e métodos), caracterizando uma forma de reaproveitamento de código (já que não se precisa codificá-los novamente

nas subclasses), podem possuir seus próprios atributos e métodos, que são restritos a elas, e por isso são específicos de cada uma delas.

IV – Afirmativa incorreta.

Justificativa: a incoerência desta afirmação está no fato de uma subclasse poder possuir um método exatamente com a mesma assinatura da superclasse, caracterizando com isso a sobrescrita de método. Isso pode acontecer quando o método da superclasse não é satisfatório para aquela subclasse, de forma que, ao se gerar outro método com mesma assinatura na subclasse, se está substituindo aquele da superclasse. No entanto o conceito de uma subclasse "ser" uma superclasse é correto, já que ele possui tudo o que a superclasse possui.

V – Afirmativa correta.

Justificativa: devemos lembrar que um atributo sem a declaração de seu modificador de acesso significa que aquele atributo possui o modificador default, o qual permite acesso direto apenas às classes do mesmo pacote (package). Uma classe que esteja em outro pacote somente poderá acessar aquele atributo a partir de métodos públicos da classe que o contém.

4.6 ArrayList e Vectors

O ArrayList é implementado como um array de objetos. Ele é dimensionado dinamicamente (com relação à memória utilizada, sempre que for necessário, o seu tamanho aumenta proporcionalmente a 50% do tamanho da lista). Além disso o ArrayList permite que seus elementos sejam adicionados por meio do método add(), acessados diretamente pelo método get() e removidos pelo método remove().

Vector e ArrayList são muito semelhantes na sua utilização. Ambos geram vetores de objetos, porém há uma diferença na sua utilização no que diz respeito à utilização de threads (item que vamos ver adiante), de modo que vários threads podem operar em um ArrayList ao mesmo tempo (e por isso é considerado não sincronizado, ou assíncrono); no entanto apenas um único thread pode operar em um Vector por vez (e por isso é considerado sincronizado, ou síncrono).

No exemplo a seguir, é gerada a classe ClasseTipo, com seus atributos e métodos:

```
public class ClasseTipo {
   public int atrib01;
   public String atrib02;
   public double atrib03;
   public ClasseTipo(int i, String s, double d) {
     atrib01 = i;
     atrib02 = s;
     atrib03 = d;
   }
```

```
    public void mostraCT() {
      System.out.println(atrib01 + ": " + atrib02
                                              + " (" + atrib03 + ")");
    }
}
```

Assim, a partir da classe ClasseTipo, vamos gerar um ArrayList (um vetor) de objetos gerados a partir daquela classe (ClasseTipo).

Observação

Os objetos, criados utilizando-se seu método construtor, serão gerados a partir de uma mesma variável (ct), porém cada objeto é guardado independente e separadamente tanto no ArrayList quanto no Vector.

Com ArrayList:

```
ArrayList<ClasseTipo> conjunto = new ArrayList();
ClasseTipo ct;
ct = new ClasseTipo(1, "Primeiro", 25.6);
conjunto.add(ct);
ct = new ClasseTipo(2, "Segundo", 25.7);
conjunto.add(ct);
ct = new ClasseTipo(3, "Terceiro", 25.8);
conjunto.add(ct);
ct = null;
int tam = conjunto.size();
// lendo cada um dos objetos do vetor
for (int i = 0; i < tam; i++) {
   conjunto.get(i).mostraCT();
}
```

Observação

A declaração de ArrayLists leva os símbolos < e >, os quais envolvem o nome da classe que representa todos os objetos daquele ArrayList (<Classe>), indicando o tipo do objeto que será inserido no array.

O objeto ct é alterado a cada nova instância da ClasseTipo, porém pelo fato de serem a cada momento adicionados no ArrayList, os endereços de memória desses objetos ficarão guardados no array mantendo esses objetos ativos e acessíveis dentro do ArrayList, podendo o objeto ct ter seu endereço alterado para a nova instância.

Com Vector:

```
Vector<ClasseTipo> conjunto = new Vector();
ClasseTipo ct;
ct = new ClasseTipo(1, "Primeiro", 25.6);
conjunto.add(ct);
ct = new ClasseTipo(2, "Segundo", 25.7);
conjunto.add(ct);
ct = new ClasseTipo(3, "Terceiro", 25.8);
conjunto.add(ct);
ct = null;
int tam = conjunto.size();
for (int i = 0; i < tam; i++) {
   conjunto.get(i).mostraCT();
}
```

Observação

Assim como a declaração de ArrayLists, a declaração de Vectors também leva os símbolos < e >, os quais envolvem o nome da classe que representa todos os objetos daquele Vector (<Classe>).

Também para o Vector, tem-se a mesma observação em relação ao objeto ct descrita nas observações do ArrayList.

Percebe-se que a forma, ou estrutura, de utilização de cada um deles (ArrayList e Vector) é semelhante.

4.7 Trabalhando com strings

Sabemos que a classe String representa os tipos de dados alfanuméricos como palavras, frases e textos em geral. Na linguagem Java, esse tipo de dado não é um tipo primitivo, de forma que sua classe corresponde a um conjunto de caracteres, os quais contêm uma série de métodos que permitem a manipulação dessas strings de diversas formas.

Podemos, a partir dos métodos que pertencem à classe String, verificar seu tamanho (a quantidade de caracteres que possui), dividi-la em partes, alterar uma parte dela substituindo por outra, verificar se existe um texto específico dentro dela e descobrir em que posição está, entre outras ações.

Estes métodos são úteis quando vamos trabalhar com organização de dados, mineração de dados, inteligência artificial, aprendizado de máquina (machine learning), análise de sentimentos e muito mais.

Neste item, vamos ver alguns métodos (os mais comuns) e mostrar como trabalhar com cada um deles a fim de aprendermos a analisar textos.

O método length()

Esse método retorna um valor do tipo int que representa o tamanho de uma string, ou seja, a quantidade de caracteres que a string possui, lembrando que, por caracteres, estão compreendidos números, letras maiúsculas, letras minúsculas, símbolos (@, !, #, $ ...) e o espaço.

Sua sintaxe é:

<nome_da_String>.length()

Exemplo:

```
"...
String nome = "Fulano de Tal";
int tam = nome.length(); // no caso: tam = 13
..."
```

Neste exemplo, a variável tam receberá o tamanho da variável nome, já que o objeto nome representa uma string (é um objeto do tipo string).

O método toUpperCase() e toLowerCase()

Esses dois métodos retornam o valor da variável alterado de duas formas:

- **toUpperCase()**: retorna o texto inteiramente em maiúsculo (caixa alta);
- **toLowerCase()**: retorna o texto inteiramente em minúsculo (caixa baixa).

É importante saber que esses dois métodos não alteram o valor da variável, mas sim **retornam** o valor equivalente à sua característica.

Sua sintaxe é:

<nome_da_String>.toUpperCase()
<nome_da_String>.toLowerCase()

Exemplo:

```
"...
String nome = "Fulano de Tal";
String txt1 = nome.toUpperCase();
// no caso, txt1 recebe: FULANO DE TAL
```

```
String txt2 = nome.toLowerCase();
// no caso, txt2 recebe: fulano de tal
// mas a variável "nome" continua com o valor inicial:
// Fulano de Tal
..."
```

O método trim()

Esse método retorna o texto original (um valor do tipo string) no qual são removidos todos os espaços em branco que aparecem no início e no final do texto original. É importante observar que esse método não altera o valor da variável original, mas apenas **retorna** o valor alterado. Outra observação importante é a de que os espaços internos, que porventura estejam a mais, continuarão intactos (sem alteração).

Sua sintaxe é:

<nome_da_String>.trim()

Exemplo:

```
"...
String texto = "   um    texto qualquer    ";
String txt = texto.trim();
System.out.println("[" + txt + "]");
// imprimirá: [um    texto qualquer]
System.out.println("[" + texto + "]");
// imprimirá: [   um    texto qualquer    ]
..."
```

No exemplo que acabamos de apresentar, o texto contém mais de um espaço antes e depois das palavras, e na variável txt, que recebe o valor alterado da variável texto, não contém esses espaços iniciais e finais (foram retirados inteiramente). É importante observar que entre a palavra um e a palavra texto contém mais de um espaço e que ele foi mantido nas duas variáveis (texto e txt), já que o método trim() não mexe nos espaços internos.

Observação

Os colchetes da impressão foram inseridos apenas no ato da impressão das variáveis, e não em seus valores internos, para que fosse possível verificar de forma fácil a alteração ocorrida sobre o texto a partir da utilização do método.

O método charAt(...)

Diferentemente de outras linguagens de programação, na linguagem Java uma string não é um array de caracteres, de modo que para acessarmos um determinado caractere no interior da string, devemos nos utilizar de métodos próprios da classe.

O método charAt(...) permite então que acessemos um caractere numa determinada posição do interior de uma string. A posição desejada é um valor numérico inteiro, com um valor equivalente ao de um índice de arrays (ou seja, começando com o valor 0), e que deve ser indicado no parâmetro do método. Esse método retornará um valor do tipo char que representará o caractere existente na posição indicada no parâmetro.

Sua sintaxe é:

<nome_da_String>.charAt(<num_posicao>)

Exemplo:

```
"...
String texto = "palavra";
char c1 = texto.charAt(0); // c1 recebe 'p'
char c2 = texto.charAt(6); // c2 recebe 'a' (o último 'a')
char c3 = texto.charAt(8);
// este último resulta em ERRO pois é uma posição
// inexistente, gerando uma exceção do tipo
// "StringIndexOutOfBoundsException"
..."
```

O método indexOf(...)

Utilizamos esse método quando precisamos procurar um trecho de String (uma substring) ou até uma letra dentro da String. O método retornará um número inteiro equivalente ao valor da posição inicial da substring procurada. É importante ressaltar que caso haja repetição da substring, ou da letra, no texto original, o método indexOf() retornará a posição apenas do primeiro caso encontrado do trecho a partir da posição inicial (que pode ser o próprio valor dessa posição, dependendo do caso).

Sua sintaxe é:

<nome_da_String>.indexOf(<substring>)

ou:

<nome_da_String>.indexOf(<substring>, <posic_inicial>)

Exemplo:

O exemplo a seguir mostra a posição de todas as letras a que aparecem no texto inicial, utilizando-se de um laço de repetição, sempre renovando a posição inicial de busca da substring (no caso, a posição da letra a):

```
"...
String texto = "aa batata";
int pos = texto.indexOf("a");
while (pos >= 0) {
   System.out.print(pos + "\t");
   pos = texto.indexOf("a", pos + 1);
}
// imprimirá: 0      1       4       6       8
// Porém:
System.out.print(texto.indexOf("w"));
// Imprimirá: -1
// pois esta String "w" não existe no texto original
..."
```

O método replace(...)

Esse método retorna um valor do tipo string, de modo que toda ocorrência encontrada com o trecho indicado no parâmetro (substring procurada) será substituído pela nova substring. Vale ressaltar que a variável original não sofre a alteração, ou seja, o método replace() **retorna** um valor alterado sem alterar o valor da variável original.

Caso o método não encontre a substring procurada, ele não resulta em erro, simplesmente retornando o valor original sem alterações.

Sua sintaxe é:

<nome_da_String>.replace(<substr_procurada>, <nova_substr>)

Exemplo:

```
"...
String texto = "palavra";
String txt = texto.replace("a", "x");
// txt receberá: pxlxvrx
// onde TODA ocorrência da substring "a" foi alterada,
// de modo que o valor da variável "texto"
// continua sendo "palavra".
```

```
txt = texto.replaceAll("k", "x");
// neste caso txt receberá: palavra
// já que a substring "k" não existe na String original.
..."
```

O método substring(...)

Esse método retorna um valor do tipo string equivalente a um trecho da string original. Os parâmetros deste método devem receber um número inteiro equivalente ao índice relativo à posição inicial e à posição final+1 (este último, opcional).

Sua sintaxe é:

<nome_da_String>.substring(<posic_inicial>)

ou:

<nome_da_String>.substring(<posic_inicial>, <posic_final>)

No primeiro caso, só com o valor da posição inicial, o texto resultante equivale ao trecho que inicia no índice indicado e termina no último índice (vai até o final da string original).

No segundo caso, o valor da posição inicial equivalerá ao índice exato do caractere que fará parte da substring resultante, e a posição final indica que a substring resultante terá todos os caracteres até um índice anterior ao valor indicado (vide exemplo a seguir).

É importante salientar que caso um dos valores de posição (seja o inicial ou o final) não exista no texto original (lembrando que o valor final pode sempre ser um valor acima do valor do último índice), esse comando resultará em um erro, interrompendo a execução do código e provocando o que chamamos de exceção (do tipo StringIndexOutOfBoundsException).

Exemplo:

```
"...
String texto = "um texto qualquer";
txt = texto.substring(3);
// txt receberá: "texto qualquer"
// já que na posição 3 está a
// primeira letra t da palavra "texto"
txt = texto.substring(3, 10);
// txt receberá: "texto q"
// já que no índice 9 (um antes do valor 10) temos
// a primeira letra "q" da palavra "qualquer".
```

```
txt = texto.substring(3, 30);
// Este comando resulta em ERRO,
// já que a posição 30 não existe na String original.
..."
```

O método split(...)

Esse método é muito utilizado em leitura de arquivos de texto externos ao programa, para identificar partes de seu texto que geralmente contêm dados específicos sobre configurações ou de banco de dados.

Este método retorna um array de strings (uma matriz de strings) que contém, em cada uma de suas posições, trechos (substrings) do texto original "recortado em partes". Esse recorte é realizado segundo uma referência, ou, ainda, uma substring, indicada no parâmetro do método.

Observação

O fato de retornar um array facilita sua manipulação, já que podemos ler um array utilizando uma estrutura for (vide exemplo a seguir).

Sua sintaxe é:

<nome_da_String>.split(<referencia>)

Quanto à substring de referência, podemos ter três situações:

- **caso essa substring seja encontrada**: neste caso ele gerará um array com todas as substring separadas pela referência, na ordem que elas aparecem (observação: a string de referência é retirada ao gerar as substrings – entende-se que esse valor é apenas uma referência para a identificação e separação das substrings);

- **caso essa substring** não seja encontrada: neste caso ele gerará um array com apenas um elemento, que é o próprio texto completo da string original;

- **caso seja uma string vazia ("")**: neste caso, retornará um array de strings em que cada um dos caracteres da string original ocupará uma posição no array resultante, de modo que o tamanho do array será igual ao tamanho da string original.

Exemplo:

```
"...
String texto = "um texto qualquer";
String[] matStr = texto.split("");
```

```
for (int x = 0; x < matStr.length; x++) {
   System.out.println(matStr[x]);
}
// Este laço de repetição irá imprimir letra a letra
// do texto original, uma abaixo da outra:
// u
// m
//
// ... e assim por diante, até
// e
// r
matStr = texto.split(" ");
for (int x = 0; x < matStr.length; x++) {
   System.out.println(matStr[x]);
}
// Este laço de repetição imprimirá:
// um
// texto
// qualquer
// ...já que a referência é um espaço (" "), e com isso,
// tudo que NÃO é "espaço", vira um elemento do array.
matStr = texto.split("w");
for (int x = 0; x < matStr.length; x++) {
   System.out.println(matStr[x]);
}
// Este laço de repetição imprimirá:
// um texto qualquer
// ...já que a referência ("w") é um valor
// inexistente no texto original.
...”
```

O método toCharArray()

Este método retorna um array de caracteres (valores do tipo char) com todos os caracteres da string original, cada um deles ocupando uma das posições do array, na ordem em que aparecem no texto original.

Cada elemento do array será considerado um dado do tipo char, e como tal deverá ser trabalhado.

Sua sintaxe é:

<nome_da_String>.split(<referencia>)

Exemplo:

```
"...
String texto = "um texto qualquer";
char[] matChar = texto.toCharArray();
for (int x = 0; x < matChar.length; x++) {
   System.out.println(matChar[x]);
}
// Este laço de repetição irá imprimir letra a letra
// do texto original, uma abaixo da outra:
// u
// m
//
// ... e assim por diante, até
// e
// r
..."
```

O método equals(...)

Quando uma string for instanciada como se fosse uma classe, as comparações realizadas com o operador **==** não funcionarão como funcionam com strings criadas na forma literal.

```
"...
// ##### Criando de forma literal:
String txt1 = "qualquer coisa";
String txt2 = "qualquer coisa";
if (txt1 == txt2) {
   System.out.println("São iguais");
} else {
   System.out.println("São diferentes");
}
// Neste caso, como ambas foram criadas de forma literal
// a estrutura condicional aceitará a comparação com o
// operador ==, e imprimirá que "São iguais".
// Obs.: Isto também funcionará para a situação
// de comparação com o método equals(...).
// ##### Criando na forma de Classes:
String txt3 = new String("qualquer coisa");
String txt4 = new String("qualquer coisa");
if (txt1 == txt2) System.out.println("Iguais – comp. 1");
if (txt1.equals(txt2)) System.out.println("Iguais – comp. 2");
if (txt1 == txt3) System.out.println("Iguais – comp. 3");
if (txt3 == txt4) System.out.println("Iguais – comp. 4");
```

```
if (txt1.equals(txt3)) System.out.println("Iguais – comp. 5");
if (txt3.equals(txt4)) System.out.println("Iguais – comp. 6");
// Nestes casos acima, as únicas saídas serão:
// Iguais – comp. 1
// Iguais – comp. 2
// Iguais – comp. 5
// Iguais – comp. 6
// Isto ocorre por que as comparações com o operador == só
// funcionarão se as duas variáveis tiverem sido criadas
// na sua forma literal.
// Caso a variável (qualquer uma delas) tenha sido criada na
// forma de instância da Classe String, a única forma de
// comparação possível será utilizando o método equals(...)
..."
```

O método format(...)

Esse método, que retorna um valor do tipo string, permite a formatação de dados como datas ou números, a fim de adequá-los à sua forma de amostragem, dependendo de como devem aparecer em relatórios ou textos em geral.

Existem várias maneiras de utilizarmos o método format(...). Aqui, vamos apresentar um conteúdo básico (mas suficiente) para conseguirmos configurar textos de relatórios.

O método se utiliza de alguns símbolos, interpretando-os à medida que os encontra ao longo da string. Vejamos um primeiro exemplo:

```
String nome = "José";
int dia = 4;
String mes = "maio";
int ano = 1950;
// ------------------------
String txtGeral = "O Sr. %s, nasceu em %s de %s de %s.";
String msg = String.format(txtGeral, nome, dia, mes, ano);
System.out.println(msg);
/*
Este programa imprimirá na tela da Console:
O Sr. José, nasceu em 4 de maio de 1950.
*/
```

Esse exemplo mostra como podemos criar uma mensagem que inicialmente não contém alguns de seus dados, mas que podem ser preenchidos ao longo do programa a fim de completar a frase a ser amostrada.

O símbolo %s que aparece no valor (texto) da variável txtGeral indica ao compilador que ali deverá ser inserido um valor (a ser interpretado como uma string). Perceba que há quatro desses símbolos no texto, de modo que o comando dado com o método format(...) contém como parâmetro cinco variáveis, todas elas separadas por vírgula, sendo a primeira o texto a ser interpretado, e as demais deverão contemplar todos os símbolos que aparecem ao longo do texto inicial.

Dessa forma, cada símbolo será substituído pelo valor da variável localizada após a variável que representa o texto original e relativa à sua ordem de aparição, ou seja: o primeiro símbolo será substituído pelo valor da segunda variável; o segundo símbolo será substituído pelo valor da terceira variável; e assim por diante, lembrando que **todos** os símbolos devem ter um valor correspondente definido após o texto.

Sabemos que a solução poderia também ser realizada utilizando-se de concatenação simples, como:

```
"...
String txtGeral = "O Sr. " + nome + ", nasceu em " + dia + " de " + mes + " de " + ano + ".";
..."
```

No entanto, muitas vezes nos deparamos com textos que exigem uma formatação específica de amostragem, como quando queremos que as datas fiquem num formato do tipo dd/mm/aaaa.

Nesse caso, se utilizarmos uma concatenação simples:

```
"...
int dia = 4;
int mes = 5;
int ano = 1950;
// ....
String msg = "No dia " + dia + "/" + mes + "/" + ano + ", nasceu...";
System.out.println(msg);
/*
Este programa imprimirá na tela da Console:
No dia 4/5/1950, nasceu...
*/
..."
```

Percebam que a data não apareceu num formato padrão que possamos utilizar em documentos ou relatórios.

Sendo assim, apresentamos alguns exemplos de utilização do método format(...), seguidos da explicação dos símbolos utilizados:

1º exemplo:

```
"...
String txt = "No dia %02d/%02d/%04d, nasceu...";
String msg = txt.format(txt, dia, mes, ano);
System.out.println(msg);
/*
Este programa imprimirá na tela da Console:
No dia 04/05/1950, nasceu...
*/
..."
```

Neste caso foi utilizado o símbolo %02d, de modo que:

- o d indica que se está trabalhando com números inteiros;
- o 0 indica que o valor textual deverá ser completado com zeros à esquerda;
- o 2 indica a quantidade de dígitos que devem compor a representação do número (assim como o 4 no símbolo %04d).

Outro exemplo da utilidade desse formato é a impressão de números de conta em banco, código (ou número de funcional) de funcionários etc.:

```
int icc = 572; // número da conta
int idv = 4; // dígito verificador
String cc1 = "Conta Corrente: %06d-%02d";
String ccFinal = cc1.format(cc1, icc, idv);
System.out.println(ccFinal);
/*
Este programa imprimirá na tela da Console:
Conta Corrente: 000572-04
*/
```

2º exemplo:

```
"...
int num = 1234567;
String val = "Valor: %,1d";
System.out.println(val.format(val, num));
/*
```

```
Este programa imprimirá na tela da Console:
Valor: 1.234.567
*/
..."
```

Neste caso, utilizamos o símbolo %,1d de modo que:

- A vírgula (,) indica que deverá ser impresso o separador de **milhares**. Ele pode aparecer como no sistema inglês, com vírgula, ou como no sistema brasileiro, com ponto, dependendo da configuração geral do micro do usuário.

- O 1 indica que ao menos um dígito, o da unidade, deverá aparecer. Os demais só aparecem se existirem no número.

3º exemplo:

```
"...
double med = 12345.67;
String medS = "Medida: %,1.5f";
System.out.println(medS.format(medS, med));
/*
Este programa imprimirá na tela da Console:
Medida: 12.345,67000
*/
..."
```

Neste caso, utilizamos o símbolo %,1.5f, de modo que:

- O f indica que estamos formatando um dado numérico de ponto flutuante, ou seja, um valor do tipo float ou double.

- O .5 indica que queremos que apareçam de forma fixa apenas cinco casas decimais depois da vírgula, que são completadas com zeros caso o número não tenha essa quantidade de casas decimais.

Saiba mais

Para saber mais sobre as diversas formatações que o método format(...) fornece, acesse o link a seguir, do site oficial do tutorial do Java (The Java™ Tutorials), que mostra as possíveis formatações:

ORACLE. *The Java™ Tutorials*. [s.d.]b. Disponível em: https://cutt.ly/2TDTFDr. Acesso em: 22 nov. 2021. Acesso em: 3 ago. 2021.

Exemplo de aplicação

Imagine que você foi contratado para dar manutenção em um sistema. O cliente solicitou que você acrescentasse uma nova funcionalidade em um código já existente, a qual consiste em analisar um conjunto de informações vindas de um arquivo texto na forma de array de strings, e que em cada posição desse array existe um texto que reúne diversas informações separadas pelo símbolo # (hashtag, cerquilha ou sustenido).

Exemplo de um trecho daquele array:

"...
matriz[25] = "Projeto FGH#Fulano de Tal#Filial de São Paulo#..."
matriz[26] = "Projeto XYZ#Beltrano dos Anzóis#Filial de Vitória#..."
matriz[27] = "Projeto KLM#..."
...
matriz[41] = "Projeto PQR#Ciclano das Couves#Filial de Salvador#..."
..."

Esse trecho do array mostra como os dados estão dispostos e dizem respeito a dados específicos como o nome dos projetos em andamento, quem é o responsável, em qual filial da empresa está sendo desenvolvido esse projeto, entre muitas outras informações.

Além disso, as informações deverão ser analisadas de várias formas, pois o sistema que as processa limita o tamanho de cada uma das informações, além de comparar valores numéricos existentes nas informações da string, com intervalos numéricos definidos em regras de negócio específicas de cada área da empresa.

Sendo assim, qual dos grupos de estruturas e métodos a seguir seria necessário para destrinchar cada um dos dados do array de informações que acabamos de mostrar, em relação às problemáticas descritas, e permitiria trabalhar com cada uma das informações individualmente?

A) estrutura for, estrutura if, método split() das strings, método legth() das strings.

B) estrutura if, estrutura switch – case, método replace() das strings e método substring() das strings.

C) estrutura for, estrutura switch – case, método replace() das strings e método length() das strings.

D) estrutura while, estrutura if, método toLowerCase() das strings e método trim() das strings.

E) estrutura for, estrutura switch – case, método trim() das strings e método toUpperCase() das strings.

Resposta correta: alternativa A.

Resolução

A) Alternativa correta.

Justificativa: a estrutura for nos permitiria acessar cada um dos elementos do array. O método split nos permitiria destrinchar cada string (cada elemento do array), a partir do caractere de referência (#). O método length nos permitiria definir o tamanho do array, além de informar o tamanho das informações, já que há restrições nas regras de negócio. A estrutura if nos permitiria realizar comparações sobre os valores numéricos e sobre os intervalos (segundo o enunciado).

B) Alternativa incorreta.

Justificativa: está faltando um laço de repetição para ler a matriz de strings. A estrutura switch – case permite apenas comparações de igualdade, não sendo possível verificar intervalos numéricos, segundo o que diz o enunciado. O método replace a princípio não auxilia nenhuma das necessidades elencadas no enunciado. O método substring só seria possível se soubéssemos a posição exata das strings #, o que seria possível com o método indexOf.

C) Alternativa incorreta.

Justificativa: a estrutura switch – case permite apenas comparações de igualdade, não sendo possível verificar intervalos numéricos, segundo o que diz o enunciado. O método replace a princípio não auxilia nenhuma das necessidades elencadas no enunciado.

D) Alternativa incorreta.

Justificativa: os métodos toUpperCase e trim a princípio não auxiliam nenhuma das necessidades elencadas no enunciado.

E) Alternativa incorreta.

Justificativa: os métodos toLowerCase e trim a princípio não auxiliam nenhuma das necessidades elencadas no enunciado.

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

Exemplos de aplicação

1 – Criar uma classe encapsulada de nome Produto com os seguintes atributos: codigo (int), nome (String), qtdEstoque (int), este último representando a quantidade de produto em estoque. Nessa classe, criar um método construtor que recebe como parâmetro cada um dos atributos. Criar um método de nome mostrarDados(), que deve imprimir os valores de todos os atributos.

2 – Criar uma classe encapsulada de nome Alimento que herda a classe Produto com o seguinte atributo: peso (double). Nessa classe, criar um método construtor que deve receber como parâmetro um valor para cada um dos atributos (lembrar que essa classe possui uma superclasse, que também deve ser acionada). Criar um método de nome mostrarDados(), que deve imprimir os valores de todos os atributos (inclusive os da superclasse).

3 – Criar uma classe encapsulada de nome Bebida que herda a classe Produto com o seguinte atributo: volume (double). Nessa classe, criar um método construtor que recebe como parâmetro cada um dos atributos (lembrar que essa classe possui uma superclasse, que também deve ser acionada. Criar um método de nome mostrarDados(), que deve imprimir os valores de todos os atributos (inclusive os da superclasse).

4 – Criar uma classe TesteProduto que possui o método main, gerando nele uma instância da classe Alimento e uma instância da classe Bebida (em ambos com informações a sua escolha) e acionando o método mostrarDados().

5 – Criar uma classe encapsulada de nome Data com os seguintes atributos: dia (inteiro); mes (inteiro); ano (inteiro); e bissexto (lógico).

Regras específicas:

- Os métodos setters deverão ser privados, e os getters deverão ser públicos (o método set do atributo bissexto não deve ser criado).
- O programa não deve aceitar valores de mês menores do que 1 nem maiores do que 12.
- O programa não deve aceitar valores de ano ou de dia menores do que 1.
- Os métodos setters deverão sempre (além de validarem e inserirem o valor no respectivo atributo) retornar um valor lógico (true ou false) de acordo com a validação, de forma que se o valor for válido retornará true e se for inválido retornará false. Observação: caso se tente incluir um valor não permitido, deve-se imprimir na console o seguinte texto: "Data Inválida!".
- Ao receber o valor de um ano (no método setAno(...)), deve-se preencher também o valor do atributo bissexto (com true ou false), de modo que se o valor do ano for divisível por 4 (não negativo), deve-se imprimir na tela da console "Ano Bissexto", e o atributo bissexto deverá ficar com o valor true; caso contrário, deverá ficar com o valor false. Observação: existem várias outras regras para se determinar se um ano é ou não bissexto. Além disso, pela regra verdadeira nem todo valor de ano divisível por 4 representa um ano bissexto (como os anos que terminam com 00 – por exemplo, o ano 2100 não será bissexto). Mas essas e outras regras não estão sendo levadas em consideração neste exercício.

- Implementar o método que inclui valor ao atributo dia para que, além da validação já existente, valide-o de acordo com os meses do ano, de forma que:

  – meses 1, 3, 5, 7, 8, 10 e 12: deverá aceitar até no máximo o dia 31;

  – meses 4, 6, 9 e 11: deverá aceitar até no máximo o dia 30;

  – mês 2: deverá aceitar até o dia 28 (para qualquer ano) e até o valor 29 para os anos bissextos (para isso, deve-se verificar o valor do atributo bissexto).

- Criar o método alterarData(...) (público e sem retorno), que deve receber como parâmetro os valores de ano, mês e dia (nessa ordem) e deve incluir os valores aos respectivos atributos utilizando os métodos set já existentes na classe (com as devidas validações já programadas). Os valores devem ser incluídos nos atributos de acordo com a seguinte sequência: primeiro o ano, depois o mês e depois o dia, de forma que caso resulte alguma inconsistência de data, ela não deverá ser alterada.

- Criar um método construtor para a classe Data, que possui os parâmetros para receber os valores do ano, mês e dia (nessa ordem), e que deve colocar automaticamente os valores enviados como parâmetros nos respectivos atributos. Para isso, pode-se utilizar do método alterarData(...) já construído. Caso a data inserida seja inválida, o sistema deverá colocar como data 01/01/01.

- Criar um método getData, que retorna o valor da data como uma string no formato dd/mm/aaaa.

Resolução

1 –

```
public class Produto {
   private int codigo;
   private String nome;
   private int qtdEstoque;
   public Produto (int cod, String nom, int qtd) {
     codigo = cod;
     nome = nom;
     qtdEstoque = qtd;
   }
   public void mostrarDados() {
     String txt = "Código: " + codigo + "\n";
     txt += "Nome: " + nome + "\n";
     txt += "Quantidade em Estoque: " + qtdEstoque;
     System.out.println(txt);
   }
```

```
// Métodos Setters e Getters do Encapsulamento
    public void setCodigo(int codigo) {
      this.codigo = codigo;
    }
    public int getCodigo() {
      return codigo;
    }
    public void setNome(String nome) {
      this.nome = nome;
    }
    public String getNome() {
      return nome;
    }
    public void setQtdEstoque(int qtdEstoque) {
      this.qtdEstoque = qtdEstoque;
    }
    public int getQtdEstoque() {
      return qtdEstoque;
    }
}
```

2 –

```
public class Alimento extends Produto {
    private double peso;  // .. (em gr - gramas)
    public Alimento (int cod, String nom, int qtd, double p) {
      super(int cod, String nom, int qtd);
      peso = p;
    }
    public void mostrarDados() {
      super.mostraDados();
      String txt = "Peso: " + peso + "(gr)";
      System.out.println(txt);
    }
    // Métodos Setters e Getters do Encapsulamento
    public void setPeso(double peso) {
      this.peso = peso;
    }
    public double getPeso() {
      return peso;
    }
}
```

3 –

```
public class Bebida extends Produto {
   private double volume;  // .. (em ml - mililitro)
   public Bebida (int cod, String nom, int qtd, double v) {
     super(int cod, String nom, int qtd);
     volume = v;
   }
   public void mostrarDados() {
     super.mostraDados();
     String txt = "Volume: " + volume + "(ml)";
     System.out.println(txt);
   }
   // Métodos Setters e Getters do Encapsulamento
   public void setVolume(double volume) {
     this.volume = volume;
   }
   public double getVolume() {
     return volume;
   }
}
```

4 –

```
public class TesteProduto {
   public static void main (String[] args) {
     Alimento a1 = new Alimento(132, "Biscoito", 200, 180);
     Bebida b1 = new Bebida(5141, "Suco", 500, 90);
     a1.mostrarDados();
     b1.mostrarDados();
   }
}
```

5 –

Existem várias soluções para esse exercício. Segue uma possível solução:

```
public class Data {
   private int dia;
   private int mes;
   private int ano;
   private boolean bissexto;
   // Método Construtor
   public Data(int d, int m, int a) {
     dia = 1;
```

```
  mes = 1;
  ano = 1;
  alterarData(d, m, a);
}
// Método alterarData
public void alterarData(int d, int m, int a) {
  int d0 = dia;
  int m0 = mes;
  int a0 = ano;
  boolean x = false;
  x = setAno(a);
  if (x) x = setMes(m);
  if (x) x = setDia(d);
  if (!x) {
          dia = d0;
          mes = m0;
          ano = a0;
  }
}
// Métodos Setters e Getters
public int getDia() {
  return dia;
}
private boolean setDia(int dia) {
  boolean res = false;
  if (dia >= 1) {
          if (mes == 2) {
                  if (bissexto && dia <= 29) res = true;
                  else if (dia <= 28) res = true;
          } else if (mes==4 || mes==6 || mes==9 || mes==11) {
                  if (dia <= 30) res = true;
          } else {
                  if (dia <= 31) res = true;
          }
  }
  if (res) this.dia = dia;
  eles System.out.println("Data Inválida!");
  return res;
}
public int getMes() {
  return mes;
}
private boolean setMes(int mes) {
  boolean res = false;
```

```
    if (mes >= 1 && mes <= 12) {
            res = true;
            this.mes = mes;
    } else {
            System.out.println("Data Inválida!");
    }
    return res;
  }
  public int getAno() {
    return ano;
  }
  private boolean setAno(int ano) {
    boolean res = false;
    bissexto = false;
    if (ano >= 1) {
            res = true;
            this.ano = ano;
            if (ano % 4 == 0) {
                    bissexto = true;
                    System.out.println("Ano Bissexto");
            }
    } else {
            System.out.println("Data Inválida!");
    }
    return res;
  }
  public String getData() {
    String res = "";
    String sData = "%02d/%02d/%04d";
    res += sData.format(sData, dia, mes, ano);
    return res;
  }
}
```

Resumo

Nesta unidade, vimos inicialmente que o sistema permite que misturemos tipos numéricos em fórmulas matemáticas, segundo suas equações. No entanto, dependendo do tipo da variável que receberá o valor, o sistema resolverá ou não automaticamente o tipo do valor numérico utilizado. Quando o tipo da variável receptora tem uma abrangência maior, o sistema (compilador) se encarrega automaticamente de promover ao tipo maior os valores de abrangência menor, fazendo o que chamamos de promotion. Quando o tipo da variável receptora tem uma abrangência menor, o sistema (compilador) exige que o programador indique para qual tipo os valores das variáveis de abrangência maior devem ser alterados, e esse programador deve fazer o que chamamos de casting.

```
int f = 4;
long g = 5 + f; //promotion do valor de f (que é int) para long
int h = 7 + (int)g; //casting de g (que é long) para int
```

Podemos controlar o acesso (em relação à programação) a determinados elementos (classe, atributo ou método). Em Java, trabalhamos com quatro modificadores de acesso: public, private, protected ou (default). Assim, um elemento public pode ser acessado por qualquer classe. Por sua vez, um elemento private só pode ser acessado por métodos da própria classe. Já um elemento protected pode ser acessado por classes herdeiras (subclasses) ou pelas classes do mesmo pacote. Por fim, um elemento (default) pode ser acessado apenas pelas classes do mesmo pacote.

O encapsulamento é um recurso em que restringimos o acesso (via código) aos atributos de uma classe, colocando para eles o modificador de acesso private, de modo a terem o acesso controlado por métodos da própria classe: os métodos setters e getters, cuja sintaxe geral é:

```
public void setAtributo (tipoAtributo variavel) {
        atributo = variavel;
}
public tipoAtributo getAtributo () {
        return atributo;
}
```

Uma classe não pode conter mais de um método que tenha a mesma assinatura (nome_do_método + parâmetros). No entanto, podemos gerar numa mesma classe diversos métodos que possuam o mesmo nome, desde

que eles estejam diferenciados nos seus parâmetros (seja no tipo dos parâmetros, seja na quantidade deles). Quando uma classe possui mais de um método com o mesmo nome, mas diferentes parâmetros, dizemos que nela há uma sobrecarga de métodos (overloading).

Toda instância é gerada a partir de um método construtor. Quando geramos um objeto em memória, o compilador se utiliza do método construtor daquela classe para criar a instância, de modo que se aquele não existir explicitamente no código, o compilador se utilizará de um método construtor padrão para isso. Desse modo, o método construtor é o método que gera na memória a instância de uma classe. Criamos um ou mais métodos construtores numa classe para que, ao instanciarmos aquela classe, o sistema possa automaticamente iniciar processos ou inicializar atributos com informações necessárias para o funcionamento do sistema. Se uma classe possui explicitamente um ou mais métodos construtores, ao gerarmos suas instâncias somos obrigados a utilizá-los na codificação. São características de métodos construtores:

- possuir exatamente o mesmo nome da classe;
- não possuir a definição de tipo de retorno;
- ser público.

Sua sintaxe é:

```
public NomeDaClasse (parâmetros) {
    ...
}
```

A herança é uma característica da orientação a objetos que permite que uma classe adquira as características (atributos e métodos) de outra classe, de forma que a classe que **herda** é chamada de subclasse (ou classe filha) e a classe que **é herdada** é chamada de superclasse (ou classe mãe). O termo que indica que uma classe está herdando outra é o termo extends, e sua declaração deve ser da seguinte forma:

```
public class Subclasse extends Superclasse { ... }
```

Uma classe que herda uma outra passa a "ser" essa classe, e ela possuirá todos os atributos e métodos da superclasse, independentemente do modificador de cada um deles. Na linguagem Java não há herança múltipla, mas sim encadeamento de heranças, de forma que ao herdar uma classe, a subclasse herdará todas as superclasses de sua superclasse. Esse recurso (herança) remete ao conceito de generalização e especificação, de

modo que as subclasses serão classes mais específicas e a superclasse terá características mais genéricas e abrangentes.

Por conta da especificidade das subclasses, é comum termos uma situação em que alguns métodos herdados da superclasse precisam estar mais adequados às suas características. Assim, podemos recriar esses métodos na subclasse (com a mesma assinatura), de forma que, caso sejam executados, se executará aquele da subclasse, e não o da superclasse. Essa característica, que é a existência de métodos com a mesma assinatura na superclasse e na subclasse, é chamada de sobrescrita de métodos (ou overriding).

Quando precisamos trabalhar com arrays de objetos, é comum trabalharmos com as classes ArrayList ou Vector, que são classes próprias para essas situações. A forma de se trabalhar com essas classes é basicamente a mesma, diferenciando-se apenas nos casos em que temos a utilização de threads (em que se convém trabalhar com ArrayList). Sua declaração é como segue:

```
ArrayList<UmaClasse> objArrayList = new ArrayList();
```

Ou:

```
Vector<UmaClasse> objVector = new Vector();
```

Em ambos os casos, podemos adicionar nesse objeto, a partir do método add, outros objetos criados a partir da classe neles definida (no caso, a classe UmaClasse):

```
objArrayList.add(objUmaClasse);
```

Ou:

```
objVector.add(objUmaClasse);
```

A vantagem de utilizar este tipo de array de objetos é não precisarmos guardar o objeto de origem, cujo endereço de memória foi adicionado ao array.

Também vimos nesta unidade alguns métodos da classe String, que são úteis quando vamos trabalhar com mineração de dados, classificação de informações, busca de textos, entre outras utilidades. Dessa classe, vimos como utilizar cada um dos seguintes métodos: length(), toUpperCase(), toLowerCase(), charAt(...), trim(), indexOf(...), replace(...), replaceAll(...), substring(...), equals(...), format(...), split(...) e toCharArray().

Exercícios

Questão 1. (FCC/2019. Adaptada) Um analista de TI está programando em Java e deseja relacionar a classe AcompanhaManancial de tal maneira que ela herde tudo que a classe Manancial tem, criando uma relação de superclasse e subclasse. Isso é conseguido em Java usando, inicialmente:

A) @Override public class Manancial { public class AcompanhaManancial {

B) public class AcompanhaManancial extends Manancial {

C) public class Manancial { @Override public class AcompanhaManancial {

D) protected interface AcompanhaManancial extends Manancial {

E) protected class Manancial includes AcompanhaManancial {

Resposta correta: alternativa B.

Análise da questão

De maneira geral, uma classe pode ser entendida como um "modelo" destinado à geração de objetos. A sintaxe básica de declaração de uma classe em Java é demonstrada a seguir.

```
public class NomeDaClasse {
   //...
}
```

A herança, considerada um dos pilares da orientação a objetos, é um recurso que permite a criação de novas classes a partir das características de outra classe já criada. Há, portanto, um reaproveitamento de código, de modo que uma classe filha (ou subclasse) herdará os métodos e os atributos da classe mãe (ou superclasse). Para isso, deve-se inserir, na declaração da classe filha, a palavra extends, seguida do nome da classe mãe.

Pelo enunciado, sabemos que AcompanhaManancial é a classe filha, que herda as características da Manancial, que é a classe mãe. Portanto, para declarar a classe filha realizando a herança proposta, devemos usar, inicialmente, o trecho:

```
public class AcompanhaManancial extends Manancial {
```

Questão 2. (Ibade/2019. Adaptada) Os modificadores de acesso são padrões de visibilidade de acessos às classes, atributos e métodos. Esses modificadores são palavras-chave reservadas pelo Java, ou seja, não podem ser usadas como nome de métodos, classes ou atributos. Os modificadores de acesso são classificados conforme as descrições a seguir:

1 - Indica que método ou variável só pode ser acessado de dentro da classe que os criou. Uma classe que herde de uma superclasse com atributos declarados de acordo com esse modificador só poderá ter acesso a eles por meio dos métodos públicos da própria superclasse, caso contrário, não haverá acesso a esses atributos.

2 - Indica que o método ou a variável assim declarada pode ser acessado somente dentro do pacote em que está contido, ou por meio de uma subclasse.

3 - Indica que a classe, o método ou a variável assim declarada pode ser acessado em qualquer lugar e a qualquer momento da execução do programa.

Os modificadores de acesso descritos como 1, 2 e 3 são denominados, respectivamente:

A) protected, public e static.

B) dynamic, private e protected.

C) public, static e dynamic.

D) private, protected e public.

E) static, dynamic e private.

Resposta correta: alternativa D.

Análise da questão

Conforme estudado, um sistema gerado em linguagem Java é baseado na programação de classes. Uma classe possui, basicamente, atributos (dados) e métodos (ações). Pode-se organizar o sistema em pacotes e, em cada pacote, inserimos as classes pertencentes a determinada categoria.

Além disso, podemos tornar componentes da nossa aplicação mais ou menos acessíveis por outras partes do programa. Para controlar os acessos, estão disponíveis quatro modificadores básicos de acesso. Eles estão elencados a seguir, do menos restritivo ao mais restritivo.

- **public**: é o modificador de acesso menos restritivo dos quatro. Uma declaração (classe, atributo ou método) feita com esse modificador indica que o elemento pode ser acessado de qualquer lugar e por qualquer objeto que possa visualizar a classe a que esse elemento pertence. Dessa forma, a descrição 3 do enunciado da questão diz respeito a esse modificador.

- **protected**: torna um elemento (atributo ou método) acessível apenas às classes do mesmo pacote, ou a classes relacionadas por herança. A descrição 2 corresponde e esse modificador.

- **default** (padrão, ou sem modificador): uma declaração (classe, atributo ou método) feita sem um modificador de acesso explícito assume o modificador padrão. Nesse caso, o elemento só pode ser acessado por outras classes definidas do mesmo pacote. O modificador default é mais restritivo do que o public e o protected, mas é menos restritivo do que o private. O modificador default nem mesmo aparece no texto das alternativas da questão; logo, nenhuma das descrições do enunciado corresponde a ele.

- **private**: é o modificador de acesso mais restritivo. Indica que o elemento (atributo ou método) não pode ser acessado ou utilizado por nenhuma outra classe, a não ser apenas por métodos da própria classe a que pertence. A descrição 1 corresponde a esse modificador.