DESCRIÇÃO

Introdução à programação em Java, com apresentação de classes e objetos, o básico de implementação de herança e polimorfismo, agrupamento de objetos e ambientes de desenvolvimento Java e estruturas básicas da linguagem.

PROPÓSITO

Conhecer a linguagem de programação Java, uma das mais populares nos dias atuais por causa da sua portabilidade de código e rapidez de desenvolvimento, o que a torna largamente presente em dispositivos móveis. Juntamente com a orientação a objetos, é elemento indispensável para o profissional de Tecnologia da Informação.

PREPARAÇÃO

Antes de iniciar o conteúdo deste tema, recomenda-se o uso de computador com o Java Development Kit – JDK e um IDE (Integrated Development Environment) instalados.

OBJETIVOS

MÓDULO 1

Descrever a definição, a manipulação e as nuances de classes e objetos em Java

MÓDULO 2

Descrever o mecanismo de herança e polimorfismo em Java

MÓDULO 3

Descrever os mecanismos de agrupamento de objetos em Java

MÓDULO 4

Reconhecer os ambientes de desenvolvimento em Java e as principais estruturas da linguagem

INTRODUÇÃO

A Programação Orientada a Objetos (POO) é um paradigma que surgiu em resposta à crise do software. A POO buscou resolver diversos problemas existentes no paradigma de Programação Estruturada, como a manutenibilidade e o reaproveitamento de código. Essas deficiências tiveram papel central na crise, pois causavam o encarecimento do desenvolvimento e tornavam a evolução do software um desafio. A incorporação de novas funções em um software já desenvolvido vinha acompanhada do aumento de sua complexidade, fazendo com que, em certo ponto, fosse mais fácil reconstruir todo o software.

A solução visualizada à época tinha como objetivo facilitar o reaproveitamento do código e reduzir a sua complexidade. As classes e os objetos desempenham esse papel. Uma classe é um modelo a partir do qual se produzem os objetos. Ela define o comportamento e os atributos que todos os objetos produzidos segundo esse modelo terão. Com base nesses conceitos, outros foram desenvolvidos, como a herança, o polimorfismo e a agregação, dando maior flexibilidade e melhorando o reaproveitamento de código e a sua manutenção.

A POO tem nos conceitos de classes e objetos o seu fundamento; eles são centrais para o paradigma. Assim, não é mera coincidência que eles também tenham papel fundamental na

linguagem Java.

Quando se diz que uma linguagem é orientada a objetos (OO), diz-se que essa linguagem suporta nativamente os conceitos que formam a POO. Entretanto, a aderência das linguagens aos conceitos OO varia, assim como a forma que elas os implementam. O que deve ficar claro, nesse caso, é que POO é um paradigma de programação, não devendo ser confundido com uma linguagem.

Java possui semelhança com outras linguagens OO, como a C++ ou a C#. Mas, apesar de serem parecidas, as linguagens guardam significativa diferença entre si. Por exemplo, Java é uma linguagem fortemente tipada, ou seja, ela não aceita conversões implícitas entre tipos diferentes de dados como a C++ aceita.

Neste tema, começaremos pela forma como Java trata e manipula classes e objetos. Com isso, também traremos conceitos de orientação a objetos que são essenciais para compreender o funcionamento de um software em Java. Seguiremos mostrando como Java trata os conceitos de herança e polimorfismo e, em seguida, mostraremos os mecanismos de agrupamento de objetos. Assim, teremos percorrido conceitos e técnicas que formarão a base que o profissional poderá usar para alçar voos mais altos. Finalizaremos falando sobre ambientes de desenvolvimento em Java, quando abordaremos alguns aspectos úteis para formar um profissional com senso crítico e tecnicamente capacitado.

MÓDULO 1

⊙ **Descrever a definição, a manipulação e as nuances de classes e objetos em Java**

Aprender Java sem saber orientada a objeto (OO) é, no mínimo, ilógico.

Java é uma linguagem OO pura, logo o estudo dela implica o conhecimento do paradigma OO. Por isso, começaremos explorando os conceitos de classe e objeto ao mesmo tempo em que veremos como Java os trata. Para tanto, veremos o conceito de classes e as formas de emprego, o conceito de objetos e a forma de manipulá-los e, finalmente, um estudo de caso.

CLASSES E SUA REALIZAÇÃO EM JAVA

Em POO (Programação Orientada a Objetos), uma classe é uma maneira de se criar objetos que possuem mesmo comportamento e mesma estrutura.

Formalmente falando, uma classe é uma estrutura que define:

Os dados.

Os métodos que operam sobre esses dados e formam o comportamento do objeto.

O mecanismo de instanciação dos objetos.

O conjunto formado pelos dados e métodos (assinatura e semântica) estabelece o contrato existente entre o desenvolvedor da classe e o seu usuário. Isso porque é o método que será invocado pelo usuário e, para tanto, ele precisa saber sua assinatura e conhecer seu comportamento.

O Código 1 mostra um exemplo de definição de uma classe em Java.

Diferentemente de linguagens como a C e a C++, Java não possui arquivos de cabeçalho, portanto, a implementação dos métodos ocorre junto de sua declaração. Além disso, em Java cada classe pública deve estar num arquivo com o mesmo nome da classe e extensão "*java*". Logo, a classe do Código 1 deve ser salva num arquivo de nome "**Aluno.java**".

Código 1: Definição de uma classe em Java.

```
1  class Aluno {
2       //Atributo
3       private String Nome;
4
5       //Métodos
6       public void inserirNome ( ) {
7            Nome = System.in.toString ();
8       }
9       public String recuperarNome ( ) {
10           return Nome;
11      }
12 }
```

A classe do Código 1 chama-se "**Aluno**", possui um atributo do tipo "*String*" (Nome) e dois métodos (inserirNome e recuperarNome). Mas, além disso, podemos notar as palavras reservadas "*private*" e "*public*". Essas instruções modificam a visibilidade (ou acessibilidade) de métodos, atributos e classes. O trecho mostrado é um exemplo bem simples de declaração de uma classe em Java. Segundo Gosling *et. al*. (2020), pela especificação da linguagem há duas formas de declaração de classes: normal e enum. Vamos nos deter apenas na forma normal, mostrada a seguir:

[Modificador] **class** *Identificador* [TipoParâmetros] [Superclasse] [Superinterface] **{** [Corpo da Classe] **}**

Na sintaxe mostrada, os colchetes indicam elementos opcionais.

Os símbolos que não estão entre colchetes são obrigatórios, sendo que “Identificador” é um literal.

Em negrito estão os símbolos reservados da linguagem.

Logo, a forma mais simples possível de se declarar uma classe em Java é: **class Inutil { }**.

Observamos nessa declaração a presença dos símbolos reservados (e obrigatórios) e a existência do Identificador (Inutil). É óbvio que essa classe é inútil, uma vez que esse código não faz e não armazena nada. Mas é uma expressão perfeitamente válida na linguagem.

Olhemos agora os demais símbolos da declaração. Os modificadores podem ser qualquer elemento do seguinte conjunto:

{ *Annotation*, *public*, *protected*, *private*, *abstract*, *static*, *final*, *strictfp*}.

ANNOTATION

Não é propriamente um elemento, mas sim uma definição. Sua semântica implementa o conceito de anotações em Java e pode ser substituída por uma anotação padrão ou criada pelo programador.

PUBLIC, PROTECTED E PRIVATE

São os símbolos que veremos quando falarmos de encapsulamento; são modificadores de acesso.

STATIC

Este modificador afeta o ciclo de vida da instância da classe e só pode ser usado em classes membro.

ABSTRACT E FINAL

Já os modificadores *abstract e final* relacionam-se com a hierarquia de classes. Todos esses modificadores serão vistos oportunamente.

STRICTFP

Por fim, esse é um modificador que torna a implementação de cálculos de ponto flutuando independentes da plataforma. Sem o uso desse modificador, as operações se tornam dependentes da plataforma sobre a qual a máquina virtual é executada.

💬 COMENTÁRIO

É interessante observar que alguns desse modificadores podem ser compostos com outros. Por exemplo, você pode definir uma classe como *public abstract class* Teste *{ }*.

Outro elemento opcional são os "*TipoParâmetros"*. Tais elementos fazem parte da implementação Java da programação genérica e não serão abordados aqui.

O elemento opcional seguinte é a "*Superclasse"*.

Esse parâmetro, assim como o "*Superinterface",* permite ao Java implementar a herança entre classes e interfaces. O elemento "*Superclasse"* será sempre do tipo "*extends IdentificadorClasse"*, no qual "*extends"* (palavra reservada) indica que a classe é uma subclasse de "*IdentificadorClasse"* (que é um nome de classe). Por sua vez, "*IdentificadorClasse"* indica uma superclasse da classe.

A sintaxe do elemento "*Superinterface"* é "*implements IdentificadorInterface"*. A palavra reservada "*implements"* indica que a classe implementa a interface "*IdentificadorInterface"*.

Vejamos a seguir um exemplo mais completo de declaração de classe em Java.

Código 2: Definição de classe com a presença de modificadores.

```
@Deprecated @SuppressWarnings ("deprecation") public abstract strictfp class
      //Atributo
      private String Nome;

      //Métodos (parcialmente mostrados)
      public void inserirNome ( ) {
            Nome = System.in.toString ();
      }
      public String recuperarNome ( ) {
            return Nome
      ...//outros métodos ocultados por simplicidade
      }
}
```

OBJETOS: OS PRODUTOS DAS CLASSES

Anteriormente mencionamos que as classes eram modelos para a fabricação de objetos. Ao longo da seção anterior, vimos como podemos construir esse modelo, agora veremos como usá-lo.

Nem toda classe permite a criação de um objeto.

Uma classe definida como abstrata não permite a instanciação direta. Não vamos entrar em pormenores de classes abstratas nesse momento, mas é oportuno dizer que ela fornece um modelo de comportamento (e/ou estado) comum a uma coleção de classes. A tentativa de criar um objeto diretamente a partir de uma classe abstrata irá gerar erro de compilação.

Consideraremos, assim, apenas o caso de classes que permitam a sua instanciação.

O que é instanciar?

RESPOSTA

Instanciar uma classe significa realizar o modelo, e essa realização é chamada de objeto. Para compreender melhor o que é um objeto, vamos analisar seu ciclo de vida.

A criação de um objeto se dá em duas etapas:

Primeiramente, uma variável é declarada como sendo do tipo de alguma classe.

A seguir, o compilador é instruído a gerar um objeto a partir daquela classe, que será rotulado com o identificador que nomeia a variável.

Essas duas etapas podem ocorrer na mesma linha, como no seguinte exemplo:

```
1 | Aluno objetoAluno = new Aluno ( )
```

Ou esse código pode ser separado nas etapas descritas da seguinte forma:

```
1 | Aluno objetoAluno; //declaração
2 | objetoAluno = new Aluno ( ); //instanciação
```

O processo de criação do objeto começa com a alocação do espaço em memória.

E prossegue com a execução do código de construção do objeto.

Esse código, obrigatoriamente, deve ser implementado num método especial chamado **construtor**.

O método construtor é sempre executado quando da instanciação de um objeto e obrigatoriamente deve ter nome idêntico ao da classe.

Além disso, ele pode ter um modificador, mas não pode ter tipo de retorno. A instrução "*new*" é sempre seguida da chamada ao construtor da classe. Finalmente, a atribuição ("=") inicializa a variável com o retorno (referência para o objeto) de "*new*".

Revendo a instanciação, identificamos que ela pode ser decomposta da seguinte forma:

O construtor deve conter o código que precise ser executado quando da criação do objeto. Uma vez que esse código será executado por ocasião da instanciação, fica claro que se tal código for computacionalmente custoso, isso é, exigir muito processamento, a criação do objeto será impactada.

Observe as linhas 10 e 14 do Código 3.

Código 3: Instanciação de objeto com "new".

```
1  //Importações
2  import java.util.Random;
3
4      //Classe
5      public class Aluno {
6      //Atributos
7      private String nome;
8      private int idade;
9      private double codigo_identificador;
10     private Random aleatorio;
```

```
11
12        //Métodos
13        public void Aluno ( String nome , int idade ) {
14             aleatorio = new Random ();
15             this.nome = nome;
16             this.idade = idade;
17             this.codigo_identificador = aleatorio.nextDouble ();
18        }
19        public void definirNome ( String nome ) {
20             this.nome = nome;
21        }
22        public void definirIdade ( int idade ) {
23             this.idade = idade;
24        }
25   }
```

O que mudaria se a linha 10 fosse suprimida e a linha 14 substituída por "*Random aleatório = new Random ( );*"?

RESPOSTA

Nesse caso, a variável "*aleatorio*" seria válida apenas no escopo local do método construtor, não seria um atributo da classe "Aluno" e o objeto designado por ela não existiria fora do método "*Aluno (String nome , int idade)*".

Outra forma de se instanciar um objeto é através da clonagem.

Nesse mecanismo, cópias idênticas do objeto são feitas através da cópia do espaço de memória.

Cada objeto clonado terá um identificador diferente (pois são alocados em diferentes espaços de memória), mas seu conteúdo – isto é, o conteúdo de seus atributos – será igual.

A instanciação através de "*new*" é mais demorada do que a clonagem, pois executa todos os procedimentos que vimos anteriormente.

Assim, a maior rapidez da clonagem é uma vantagem quando se precisa criar muitos objetos do mesmo tipo.

O estado de um objeto é definido pelos seus atributos, enquanto seu comportamento é determinado pelos seus métodos. Considere a seguinte instanciação a partir da classe do Código 3:

```
1 | Aluno novoAluno = new Aluno ( Carlos Alberto, 16 )
```

Após todas as etapas descritas anteriormente serem executadas, haverá um objeto criado e armazenado em memória identificado por “*novoAluno*”.

O estado desse objeto é definido pelas variáveis “*nome, idade, codigo e aleatorio*”, e seu comportamento é dado pelos métodos “*public void Aluno (String nome , int idade)*, *public void definirNome (String nome)* e *public void definirIdade (int idade)*”.

Após a instanciação mencionada, o estado do objeto será “*nome = Carlos Alberto*”, “*idade = 16*”. A variável “*codigo*” terá como valor o retorno da função “*nextDouble*”, e a variável “*aleatório*” terá como valor uma referência para o objeto do tipo “*Random*” criado.

O comportamento modelado no construtor permite a construção inicial do objeto. Os métodos “*definirNome*” e “*definirIdade*” permitem, respectivamente, atualizar o nome e a idade. Então, se for executada a chamada “*novoAluno.defnirNome (“Carlos Roberto”)*”, a variável “nome” será atualizada, passando a valer “Carlos Roberto”, alterando, assim, o estado do objeto. Repare que a chamada indica ao compilador uma forma inequívoca de invocar o método. O compilador é informado que está sendo invocado o método “*definirNome*” do objeto “*novoAluno*”.

A última etapa do ciclo de vida de um objeto é sua destruição. Esse processo é necessário para se reciclar o espaço de memória usado. Quando um objeto é destruído, a JVM executa diversas operações que culminam com a liberação do espaço de memória usado pelo objeto. Esse é um ponto em que a Java difere de muitas outras linguagens OO.

★ EXEMPLO

Java não possui o conceito de destrutor. Em outras linguagens, como a C++, o método destrutor é invocado quando um objeto é destruído e a destruição do objeto é feita manualmente pelo programador.

Na linguagem Java, não é possível ao programador manualmente destruir um objeto. Em vez disso, a Java implementa o conceito de coletor de lixo. Periodicamente, a JVM varre o programa verificando objetos que não estejam mais sendo referenciados. Ao encontrar tais objetos, a JVM os destrói e libera a memória. O programador não possui qualquer controle sobre isso.

É possível ao programador, todavia, solicitar à JVM que a coleta de lixo seja realizada. Isso é feito através da invocação do método “*gc ( )*” da biblioteca “*System*”.

Essa, porém, é apenas uma solicitação, não é uma ordem de execução, o que significa que a JVM tentará executar a coleta de lixo tão logo quanto possível, mas não necessariamente quando o método foi invocado.

Mas e se o programador desejar executar alguma ação quando um objeto for destruído?

RESPOSTA

Apesar de a Java não implementar o destrutor, ela fornece um mecanismo para esse fim: o método “*finalize ( )*”. Este, quando implementado por uma classe, é invocado no momento em que a JVM for reciclar o objeto. Todavia, o método “*finalize ( )*” é solicitado quando o objeto estiver para ser reciclado, e não quando ele sair do escopo. Isso implica, segundo Schildt (2014), que o programador não sabe quando – ou mesmo se − o método será executado.

CLASSES E O ENCAPSULAMENTO DE CÓDIGO

O encapsulamento está intimamente ligado ao conceito de classes e objetos.

Do ponto de vista da POO, o encapsulamento visa ocultar do mundo exterior os atributos e o funcionamento da classe. Dessa maneira, os detalhes de implementação e variáveis ficam isolados do resto do código.

O contato da classe (e, por conseguinte, do objeto) com o resto do mundo se dá por meio dos métodos públicos da classe. Tais métodos formam o chamado **contrato**, que é estabelecido entre a classe e o código que a utiliza.

O conceito de encapsulamento também se liga ao de visibilidade.

A visibilidade de um método ou atributo define quem pode ou não ter acesso a estes. Ou seja, ela afeta a forma como o encapsulamento funciona. Há três tipos de visibilidade, representados pelos modificadores “*private"*, *"protected"* e "*public”*. Não entraremos em detalhes sobre eles nesse momento, deixando para explorar o assunto quando falarmos de herança e

polimorfismo. Mas convém dizer que “*private*” indica que o método ou atributo só pode ser acessado internamente à classe, enquanto “*public”* define que ambos são visíveis para todo o exterior.

TRABALHANDO COM CLASSES E OBJETOS

As classes e os objetos são trechos de código que refletem o produto da análise OO e interagem entre si, formando um sistema. Essa interação é a razão pela qual estabelecemos o contrato de uma classe. Esse contrato fornece a interface pela qual outras classes ou outros objetos podem interagir com aquela classe ou seu objeto derivado.

Isso pode ser visto na linha 17 do Código 3. Ao invocarmos o método “*nextDouble ( )*” da classe “*Random*” ao qual o objeto “*aleatorio*" pertence, estamos seguindo o contrato da classe para consumir um serviço prestado por ela mesma. Ainda examinando o Código 3, notamos na linha 10 que a classe “Aluno” possui como atributo uma variável do tipo “*Random*” (outra classe). Logo, um objeto do tipo “Aluno” conterá um objeto do tipo “*Random*”.

O parágrafo anterior nos mostra uma relação entre objetos. Em OO há diferentes tipos de relações.

ASSOCIAÇÃO

A Associação é semanticamente a mais fraca e se refere a objetos que consomem – usam – serviços ou funcionalidades de outros. Ela pode ocorrer mesmo quando nenhuma classe possui a outra e cada objeto instanciado tem sua existência independente do outro. Essa relação pode ocorrer com cardinalidade 1-1, 1-n, n-1 e n-n.

AGREGAÇÃO

Outro tipo de relacionamento entre classes é a Agregação, que ocorre entre dois ou mais objetos, com cada um tendo seu próprio ciclo de vida, mas com um objeto (pai) contendo os demais (filhos). É importante compreender que, nesse caso, os objetos filhos podem sobreviver à destruição do objeto pai. Um exemplo de Agregação se dá entre escola e aluno: se uma escola deixar de existir, não implica que o mesmo irá ocorrer com os seus alunos.

COMPOSIÇÃO

A Composição difere sutilmente da Agregação, pois ocorre quando há uma relação de dependência entre o(s) filho(s) e o objeto pai. Ou seja, caso o pai deixe de existir, necessariamente o filho será destruído. Voltando ao exemplo anterior, temos uma composição entre a escola e os departamentos. A extinção da escola traz consigo a extinção dos departamentos.

Os conceitos Associação, Agregação e Composição formam conjuntos que se relacionam, como visto na Figura 1. Vemos que a Composição é um caso especial de Agregação e o conceito mais restritivo de todos, enquanto a Associação é o mais abrangente.

Figura 1: Conjunto formado pela definição das relações do tipo Associação, Agregação e Composição.

Essas relações podem ser identificadas no Código *4*.

Observe as linhas 4, 5 e 6 e os métodos “*criarDepartamento*” e “*matricularAluno*”. Vamos analisá-las.

Código 4: Agregação e Composição.

```
1  class Escola {
2      //Atributos
3      private String nome, CNPJ;
4      private Endereco endereco;
5      private Departamento departamentos [];
6      private Aluno discentes [];
7      private int nr_discentes , nr_departamentos;
8
9      //Métodos
10         public Escola ( String nome , String CNPJ) {
11             this.nome = nome;
12             this.CNPJ = CNPJ;
13             this.departamentos = new Departamento [10];
14             this.discentes = new Aluno [1000];
15             this.nr_departamentos = 0;
16             this.nr_discentes = 0;
17         }
18         public void criarDepartamento ( String nomeDepartamento ) {
19             if ( nr_departamentos <= 10 )
20             {
21                 departamentos [ nr_departamentos ] = new Departamento (
22                 nr_departamentos++;
23             } else {
24                 System.out.println ( "Nao e possivel criar outro Departamento." );
25         }
26         public void matricularAluno ( Aluno novoAluno ) {
27             discentes [ nr_discentes ] = novoAluno;
28         }
29 }
```

Primeiro, devemos notar que as linhas 4, 5 e 6 mostram uma Associação entre a classe “Escola” e as classes “Endereco”, “Departamento” e “Aluno”.

Essas linhas apenas declaram variáveis do tipo das classes mencionadas.

A instanciação de objetos ocorre depois.

No caso de “Departamento”, objetos dessa classe são instanciados na linha 21.

A criação de um objeto da classe “Departamento” depende da existência de um objeto da classe “Escola”.

Ou seja, o objeto da classe “Escola” precede a criação de objeto do tipo “Departamento”.

Em segundo lugar, uma vez que o objeto do tipo “Escola” for destruído, necessariamente todos os objetos do tipo “Departamento” também serão destruídos. Isso mostra uma relação forte entre ambas as classes com o ciclo de vida dos objetos de “Departamento” subordinados ao ciclo de vida dos objetos da classe “Escola”, ilustrando uma relação do tipo Composição. É fácil entender essa semântica. Conforme mencionamos anteriormente, basta considerar que, se uma escola deixar de existir, não é possível seus departamentos continuarem funcionando.

O caso da classe “Aluno” difere do anterior. Semanticamente, o fechamento de uma escola não causa a morte dos alunos. Estes podem migrar para outro estabelecimento. No caso do Código *4*, esse comportamento encontra-se modelado no método “*matricularAluno*”. O que a linha 27 faz é armazenar no vetor “*discentes []*” uma referência para um objeto do tipo “Aluno” criado em outro local. Assim, a criação desse objeto pode ser feita independentemente da instanciação do objeto do tipo “Escola” que o recebe.

Da mesma maneira, a destruição do último não leva necessariamente à destruição do objeto do tipo “Aluno” se este estiver sendo referenciado em outra parte do código. Entretanto, um objeto da classe “Escola” contém objetos do tipo “Aluno”. Trata-se, portanto, de uma relação de Agregação.

Oportunamente essa situação nos permite explorar a referenciação de objetos em Java. Uma referência para um objeto é um valor de endereço de memória que indica a posição em memória na qual um objeto se encontra.

Em outras linguagens, como a C++, esse conceito é comumente implementado por ponteiros, que são variáveis específicas para manipulação de endereços de memória. Por exemplo, uma variável “*aux*” do tipo inteiro guardará um valor inteiro se fizermos “*aux = 0* ”. Todavia, uma variável ponteiro deve ser usada para guardar o endereço da variável “*aux*”.

📢 ATENÇÃO

Em Java, não é possível criar variáveis do tipo ponteiro! A linguagem Java oculta esse mecanismo, de forma que toda variável de classe é, de fato, uma referência para o objeto instanciado.

Isso tem implicações importantes na forma de lidar com objetos. A passagem de um objeto como parâmetro em um método, ou o retorno dele, é sempre uma passagem por referência. Isso não ocorre com tipos primitivos, que são sempre passados por valor.

Vejamos o Código 5.

Código 5: Referenciando objetos.

```
1
2  public class Referencia {
3      private Aluno a1 , a2;
4
5      public Referencia ( ) {
```

```
6           a1 = new Aluno ( "Carlos" , 20);
7           a2 = new Aluno ( "Ana" , 23 );
8           System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
9           System.out.println("O nome do aluno a2 é " + a2.recuperarNome());
10          a2 = a1;
11          a2.definirNome("Flávia");
12          System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
13          manipulaAluno ( a1 );
14          System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
15      }
16      public void manipulaAluno ( Aluno aluno ) {
17          aluno.definirNome("Márcia");
18      }
    }
```

A execução desse código produz a seguinte saída:

O nome do aluno a1 é Carlos

O nome do aluno a2 é Ana

O nome do aluno a1 é Flávia

O nome do aluno a1 é Márcia

Vamos entender o que acontece, seguindo passo a passo a execução do Código 5 a partir da linha 5. Essa linha instrui a JVM instanciar um objeto do tipo Aluno.

Isto é, a JVM reserva um espaço de memória para armazenar um objeto do tipo “*Aluno*” e o inicializa definindo as variáveis “*nome*” e “*idade*”, respectivamente, como "Carlos" e "20".

A linha 6 faz a mesma coisa, instanciando um segundo objeto com “*nome*” recebendo “Ana” e “*idade*” recebendo “23”. Após a execução da linha 6, existirão dois objetos distintos (“*a1*” e “*a2*”), cujos estados também são distintos.

Por isso, o resultado das linhas 7 e 8 é o mostrado nas saídas 1 e 2, respectivamente.

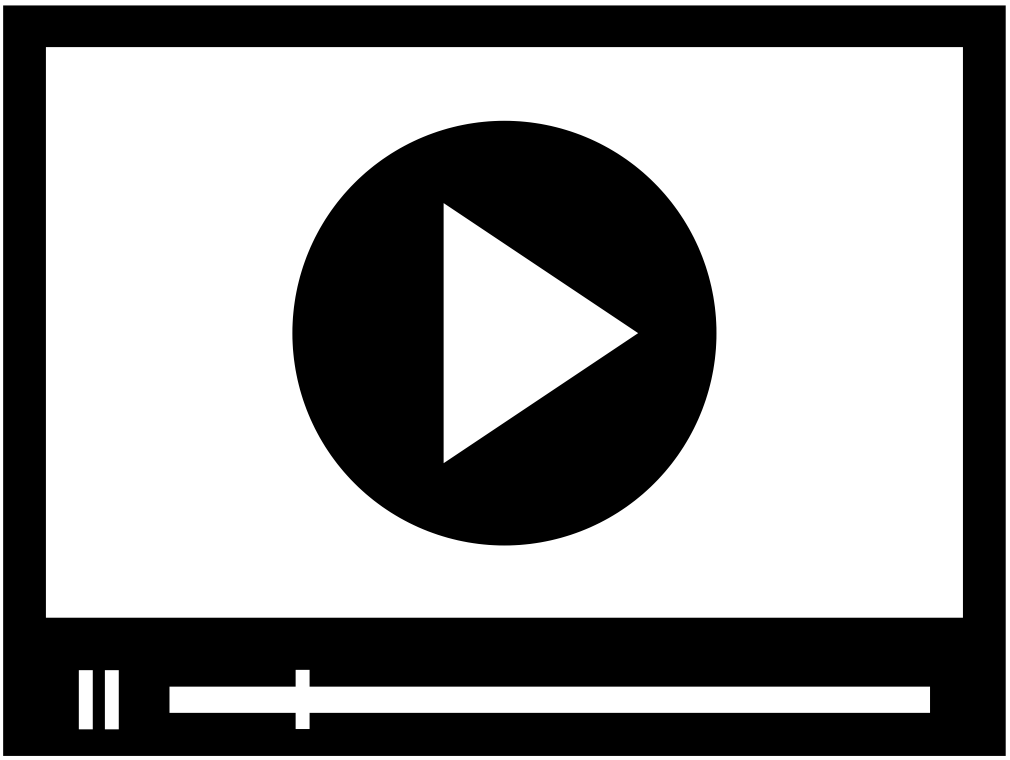
A linha 9 merece maior atenção. O segredo para entender o que ocorre nela é lembrarmos que “*a1*” e “*a2*” são referências para os objetos criados, e não o objeto em si.

Para deixar mais claro, quando atribuímos uma variável a outra, estamos atribuindo o seu conteúdo e, no caso, o conteúdo é uma referência para o objeto. Assim, ao fazermos “*a2 = a1*” estamos fazendo com que “*a2*” guarde a referência que “*a1*” possui. Na prática, “*a2*” passa a referenciar o mesmo objeto referenciado por “*a1*”.

Por isso, na linha 10, ao alterarmos o conteúdo da variável “*nome*” estamos fazendo-o no objeto referenciado por “*a1*” (e, agora, por “*a2*”).

Por esse motivo, a linha 11 exibe o nome “Flávia”. Na verdade, após a linha 9, o uso de “*a1*” e “*a2*” é indistinto.

O que acontece na linha 12 não é muito diferente. Como dissemos, a passagem de objetos é sempre feita por referência. Logo, a variável “*aluno*” na assinatura do método “*manipulaAluno*” vai receber a referência guardada por “*a1*”. Desse momento em diante, todas as operações feitas usando “*aluno*” ocorrem no mesmo objeto referenciado por “*a1*”.



CLASSES E OBJETOS EM JAVA

Neste vídeo, o especialista Marlos de Mendonça apresenta os conceitos de classes, estado e métodos, passagem de objetos como parâmetros, citando exemplos práticos com a linguagem Java.

VERIFICANDO O APRENDIZADO

MÓDULO 2

- **Descrever o mecanismo de herança e polimorfismo em Java**

Neste módulo, examinaremos os conceitos de herança e polimorfismo. Começaremos pelos aspectos elementares de herança e, em seguida, vamos aprofundar tal discussão, criando condições para analisarmos o polimorfismo.

HERANÇA: ASPECTOS ELEMENTARES

O termo **herança em OO** define um tipo de relação entre objetos e classes, baseado em uma hierarquia.

Dentro dessa relação hierárquica, classes podem herdar características de outras classes situadas acima ou transmitir suas características às classes abaixo.

Uma classe situada hierarquicamente acima é chamada de **superclasse**, enquanto aquelas situadas abaixo chamam-se **subclasses**.

Essas classes podem ser, também, referidas como classe base ou pai (superclasse) e classe derivada ou filha (subclasse).

A herança nos permite reunir os métodos e atributos comuns numa superclasse, que os leva às classes filhas. Isso evita repetir o mesmo código várias vezes.

Outro benefício está na manutenibilidade: caso uma alteração seja necessária, ela só precisará ser feita na classe pai, e será automaticamente propagada para as subclasses.

A Figura 2 apresenta um diagrama UML que modela uma hierarquia de herança. Nela, vemos que “Empregado” é pai (superclasse) das classes “Professor”, “Diretor”, “Coordenador” e “Secretario”. Essas últimas são filhas (subclasses) da classe “Empregado”.

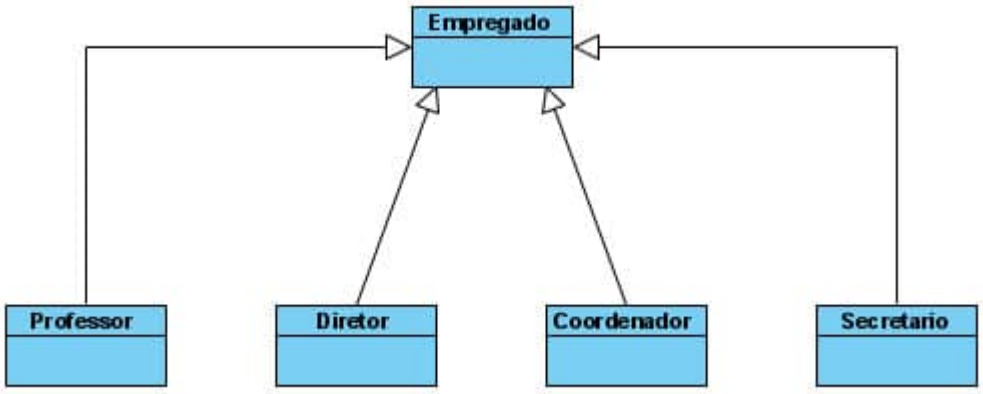


Figura 2: Diagrama de classes – herança.

E quando há mais de uma superclasse?

RESPOSTA

Essa situação é denominada herança múltipla e, apesar de a POO aceitar a herança múltipla, a linguagem Java não a suporta para classes. Veremos as implicações da herança múltipla em outra ocasião, quando explorarmos mais a fundo o conceito de herança.

Apesar de não permitir herança múltipla de classes, a linguagem permite que uma classe herde de múltiplas interfaces. Aliás, uma interface pode herdar de mais de uma interface pai. Diferentemente das classes, uma interface não admite a implementação de métodos. Esta é feita pela classe que implementa a interface.

Examinando a genealogia das classes da Figura *3*, podemos notar que, à medida que descemos na hierarquia, lidamos com classes cada vez mais específicas. De forma oposta, ao subirmos na hierarquia, encontramos classes cada vez mais gerais.

Essas características correspondem aos conceitos de generalização e especialização da OO.

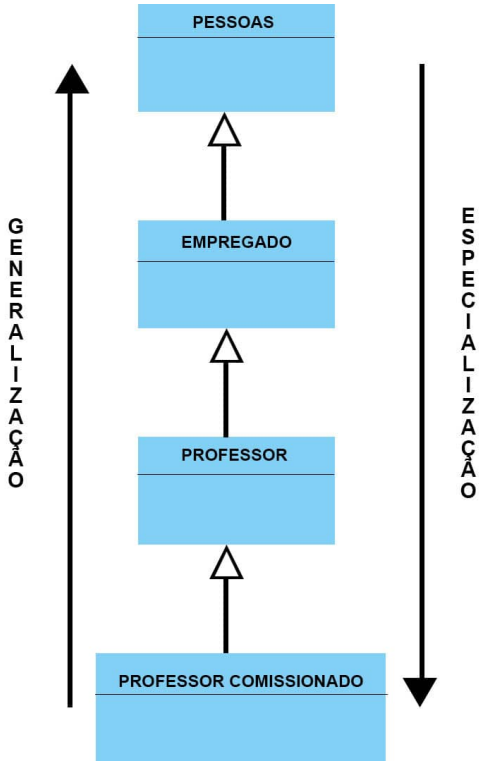


Figura 3: Herança com vários níveis de ancestralidade.

A sintaxe para declarar uma superclasse já foi mostrada no módulo anterior. O código a seguir mostra a implementação da herança, na Figura 3, para a classe “ProfessorComissionado”.

Código 6: Herança em classes.

```
1 public class ProfessorComissionado extends Professor {
2 …
3 }
```

No Código 6, podemos observar que a herança é declarada apenas para a classe ancestral imediata.

Em outras palavras:

> A classe “**Professor**” deve declarar “**Empregado**” como sua **superclasse**.
>
> “**Empregado**” deve declarar “**Pessoa**” como sua **superclasse**.

A sintaxe é análoga para o caso das interfaces, exceto que nesse caso pode haver mais de um identificador de superinterface. O código a seguir mostra um exemplo baseado na figura 2, considerando que “ProfessorComissionado”, “Professor” e “Diretor” sejam interfaces.

Nesse exemplo, a herança múltipla pode ser vista pela lista de superinterfaces (“Professor” e “Diretor”) que se segue ao modificador “*extends*”.

Código 7: Herança em interfaces.

```
1 public interface ProfessorComissionado extends Professor, Diretor {
2 …
3 }
```

Algo interessante de se observar é que em Java todas as classes descendem direta ou indiretamente da classe “Object”. Isso torna os métodos da classe “Object” disponíveis para qualquer classe criada. O método “*equals ()*”, da classe “Object”, por exemplo, pode ser usado para comparar dois objetos da mesma classe.

Se uma classe for declarada sem estender nenhuma outra, então o compilador assume implicitamente que ela estende a classe “Object”.

Se ela estender uma superclasse, como no código, então ela é uma descendente indireta de “Object”.

Isso fica claro se nos remetermos à Figura 3. Nesse caso, a classe “Pessoa” estende implicitamente “Object”, então os métodos desta são legados às subclasses até a base da hierarquia de classes. Logo, um objeto da classe “ProfessorComissionado” terá à sua disposição os métodos de “Object”.

HERANÇA E VISIBILIDADE

Quando dizemos que a classe “Pessoa” é uma generalização da classe “Empregado”, isso significa que ela reúne atributos e comportamento que podem ser generalizados para outras subclasses. Esses comportamentos podem ser especializados nas subclasses. Isto é, as subclasses podem sobrescrever o comportamento modelado na superclasse. Nesse caso, a assinatura do método pode ser mantida, mudando-se apenas o código que o implementa.

Nesta subseção, vamos entender o funcionamento desse mecanismo para compreendermos o que é polimorfismo.

Primeiramente, precisamos compreender como os modificadores de acesso operam. Já vimos que tais modificadores alteram a acessibilidade de classes, métodos e atributos. Há quatro níveis de acesso em Java:

DEFAULT

Assumido quando nenhum modificador é usado. Define a visibilidade como restrita ao pacote.

PRIVADO

Declarado pelo uso do modificador “*private”*. A visibilidade é restrita à classe.

PROTEGIDO

Declarado pelo uso do modificador “*protected*”. A visibilidade é restrita ao pacote e a todas as subclasses.

PÚBLICO

Declarado pelo uso do modificador “*public*”. Não há restrição de visibilidade.

Os modificadores de acessibilidade ou visibilidade operam controlando o escopo no qual se deseja que os elementos (classe, atributo ou método) aos quais estão atrelados sejam visíveis.

O maior escopo é o **global**, que, como o próprio nome indica, abarca todo o programa. Outro escopo é definido pelo pacote.

Um pacote define um espaço de nomes e é usado para agrupar classes relacionadas.

Esse conceito contribui para a melhoria da organização do código de duas formas: permite organizar as classes pelas suas afinidades conceituais e previne conflito de nomes.

💬 COMENTÁRIO

Evitar conflitos de nomes parece fácil quando se trabalha com algumas dezenas de entidades, mas esse trabalho se torna desafiador num software que envolva diversos desenvolvedores e centenas de entidades e funções.

Em Java, um pacote é definido pela instrução “*package*” seguida do nome do pacote inserida no arquivo de código-fonte. Todos os arquivos que contiverem essa instrução terão suas classes agrupadas no pacote. Isso significa que todas essas classes, isto é, classes do mesmo pacote, terão acesso aos elementos que tiverem o modificador de acessibilidade "*default"*.

O modificador “*private*” é o mais restrito, pois limita a visibilidade ao escopo da classe. Isso quer dizer que um atributo ou método definido como privado não pode ser acessado por qualquer outra classe senão aquela na qual foi declarado. Isso é válido mesmo para classes definidas no mesmo arquivo e para as subclasses.

O acesso aos métodos e atributos da superclasse pode ser concedido pelo uso do modificador "*protected*". Esse modificador restringe o acesso a todas as classes do mesmo pacote. Classes de outros pacotes têm acesso apenas mediante herança.

Finalmente, o modificador de acesso "*public*" é o menos restritivo. Ele fornece acesso com escopo global. Isto é, qualquer classe do ambiente de desenvolvimento pode acessar as entidades declaradas como públicas.

A Tabela 1 sumariza a relação entre os níveis de acesso e o escopo.

	default	**public**	**private**	**protected**
Subclasse do mesmo pacote	sim	sim	não	sim
Subclasse de pacote diferente	não	sim	não	sim
Classe (não derivada) do mesmo pacote	sim	sim	não	sim
Classe (não derivada) de pacote diferente	não	sim	não	não

Tabela 1: Níveis de acesso e escopo.

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

As restrições impostas pelos modificadores de acessibilidade são afetadas pela herança da seguinte maneira:

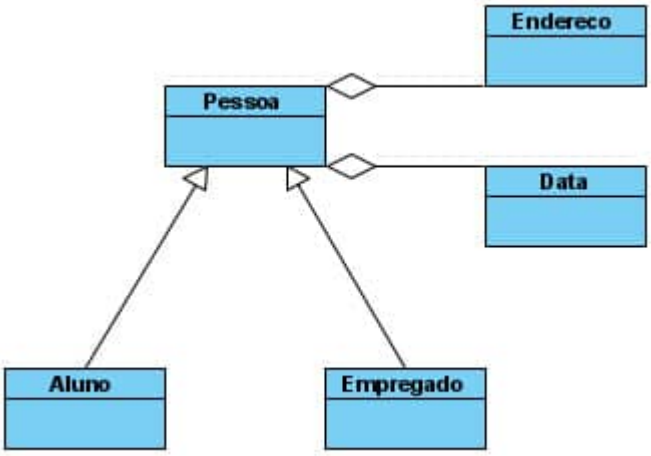
Métodos (e atributos) declarados públicos na superclasse devem ser públicos nas subclasses.

Métodos (e atributos) declarados protegidos na superclasse devem ser protegidos ou públicos nas subclasses. Eles não podem ser privados.

📢 ATENÇÃO

Métodos e atributos privados não são accessíveis às subclasses, e sua acessibilidade não é afetada pela herança.

Para entender melhor, examinaremos parte do código que implementa o modelo da figura 4:



📷 Figura 4: Diagrama de classes parcial de um sistema.

Como podemos observar no diagrama de classes exibido na figura 4, a classe “Pessoa” generaliza as classes “Empregado” e “Aluno”.

Os códigos que implementam as classes “Pessoa”, “Empregado”, “Aluno” e a classe principal do programa são mostrados respectivamente nos Código 8, Código 9, Código 10 e Código 11.

Código 8: Classe "Pessoa".

```
1
2  public class Pessoa {
3      //Atributos
4      private String nome;
5      private int idade;
6      private Calendar data_nascimento;
7      private long CPF;
8      private Endereco endereco;
9
10     //Métodos
11     public Pessoa ( String nome , Calendar data_nascimento, long CPF , Endereco e
12         this.nome = nome;
13         this.data_nascimento = data_nascimento;
14         this.CPF = CPF;
15         this.endereco = endereco;
16         atualizarIdade ();
17     }
```

```
18        protected void atualizarNome ( String nome ) {
19            this.nome = nome;
20        }
21        protected String recuperarNome ( ) {
22            return this.nome;
23        }
24        protected void atualizarIdade ( ) {
25            this.idade = calcularIdade ();
26        }
27        protected int recuperarIdade ( ) {
28            return this.idade;
29        }
30        protected void atualizarCPF ( long CPF ) {
31            this.CPF = CPF;
32        }
33        protected long recuperarCPF ( ) {
34            return this.CPF;
35        }
36        protected void atualizarEndereco ( Endereco endereco ) {
37            this.endereco = endereco;
38        }
39        protected Endereco recuperarEndereco ( ) {
40            return this.endereco;
41        }
42        private int calcularIdade ( ){
43            int lapso;
44            Calendar hoje = Calendar.getInstance();
45            lapso = hoje.get(YEAR) - data_nascimento.get(YEAR);
46            if ( ( data_nascimento.get(MONTH) > hoje.get(MONTH) ) || ( data_na
47                lapso--;
48            return lapso;
49        }
    }
```

Código 9: Classe "Empregado".

```
1
2   public class Empregado extends Pessoa {
3       //Atributos
4       protected String matricula;
5       private Calendar data_admissao , data_demissao;
6
7       public Empregado(String nome, Calendar data_nascimento, long CPF, Endereco e
8           super(nome, data_nascimento, CPF, endereco);
9           this.matricula = gerarMatricula ();
10          data_admissao = Calendar.getInstance();
```

```
    }
    public void demitirEmpregado () {
        data_demissao = Calendar.getInstance();
    }
    protected void gerarMatricula () {
        this.matricula = "Matrícula não definida.";
    }
    protected String recuperarMatricula () {
        return this.matricula;
    }
}
```

Código 10: Classe "Aluno".

```
public class Aluno extends Pessoa {
    //Atributos
    private String matricula;

    //Métodos
    public Aluno ( String nome , Calendar data_nascimento , long CPF , Endereco en
        super ( nome , data_nascimento , CPF , endereco );
    }
}
```

Código 11: Classe Principal (código parcial).

```

public class Principal {
    //Atributos
    private static Aluno aluno;
    private static Endereco endereco;

    //Método main
    public static void main (String args[]) {
        int idade;
        Calendar data = Calendar.getInstance();
        data.set(1980, 10, 23);
        endereco = new Endereco ();
        endereco.definirPais("Brasil");
        endereco.definirUF("RJ");
        endereco.definirCidade ("Rio de Janeiro");
        endereco.definirRua("Avenida Rio Branco");
        endereco.definirNumero("156A");
        endereco.definirCEP(20040901);
        endereco.definirComplemento("Bloco 03 - Ap 20.005");
```

```
20            aluno = new Aluno ("Marco Antônio", data , 901564098 , endereco);
21            aluno.atualizarIdade();
22            idade = aluno.recuperarIdade();
23
24        }
    }
```

As linhas 1 do Código 9 e do Código 10 declaram que as respectivas classes têm “Pessoa” como superclasse. Podemos notar que a classe “Pessoa” contém atributos que são comuns às subclasses, assim como os métodos para manipulá-los.

Outra coisa que podemos ver no Código 8 é que a classe “Pessoa” possui um construtor não vazio. Assim, os construtores das classes derivadas precisam passar para a superclasse os parâmetros exigidos na assinatura do construtor. Isso é feito pela instrução “*super*”, que pode ser notada na linha 7 das classes “Empregado” e “Aluno”.

Dissemos anteriormente que os métodos e atributos privados não são acessíveis às subclasses, mas se observarmos as linhas 19 e 20 do código 12, veremos que os atributos “*nome*, *idade*, *data_nascimento*, *CPF* e *endereço*”, que são privados, são definidos na instanciação da subclasse “Aluno” e imediatamente após. De fato, esses atributos não são herdados, mas a herança define uma relação “é tipo” ou “é um”. Ou seja, um objeto do tipo “Aluno” é também do tipo “Pessoa” (o inverso não é verdade).

Uma vez que foram fornecidos métodos protegidos capazes de manipular tais atributos, estes podem ser perfeitamente alterados pela subclasse. Em outras palavras, uma subclasse possui todos os atributos e métodos da superclasse, mas não tem visibilidade daqueles que são privados. Então, podemos entender que a subclasse herda aquilo que lhe é visível (ou acessível). Por isso, a subclasse "Aluno" é capaz de usar o método privado “*calcularIdade ()*” (linha 41 do Código 8) da superclasse. Mas ela o faz através da invocação do método protegido “*atualizarIdade()*”, como vemos na linha 20 da classe “Aluno”.

POLIMORFISMO

Em OO, polimorfismo é a capacidade de um objeto se comportar de diferentes maneiras.

Mais uma vez convém destacar que se trata de um princípio de OO que Java implementa, assim como várias linguagens OO.

O polimorfismo pode se expressar de diversas maneiras. A **sobrecarga de função**, assim como a **herança**, são formas de dar ao objeto uma capacidade polimórfica. No caso da herança, o polimorfismo surge justamente porque um objeto pode se comportar também como definido na superclasse.

⭐ EXEMPLO

Considere um objeto do tipo “Aluno”. Como vimos, todo objeto do tipo “Aluno” é do tipo “Pessoa”. Logo, ele pode se comportar como “Aluno” ou como “Pessoa”.

Todo objeto que possui uma superclasse tem capacidade de ser polimórfico. Por essa razão, todo objeto em Java é polimórfico, mesmo que ele não estenda explicitamente outra classe. A justificativa, como já dissemos, é que toda classe em Java descende direta ou indiretamente da classe “Object”.

O polimorfismo permite o desenvolvimento de códigos facilmente extensíveis, pois novas classes podem ser adicionadas com baixo impacto para o restante do software. Basta que as novas classes sejam derivadas daquelas que implementam comportamentos gerais, como no caso da classe “Pessoa”.

Essas novas classes podem especializar os comportamentos da superclasse, isto é, alterar a sua implementação para refletir sua especificidade, e isso não impactará as demais partes do programa que se valem dos comportamentos da superclasse.

O código 12 mostra a classe “Diretor”, que é subclasse de “Empregado”, e o código 13 mostra a classe “Principal” modificada.

Código 12: Classe "Diretor".

```
1  public class Diretor extends Empregado {
2      //Atributos
3
4      //Métodos
5      public Diretor(String nome, Calendar data_nascimento, long CPF, Endereco ende
6          super(nome, data_nascimento, CPF, endereco);
7      }
8      protected void gerarMatricula () {
9          matricula = "E-" + UUID.randomUUID( ).toString( );
10     }
11 }
```

Código 13: Classe "Principal" alterada.

```
1  public class Principal {
2      //Atributos
3      private static Empregado empregado , diretor;
4
5      //Método main
6      public static void main (String args[]) {
7
8          Calendar data = Calendar.getInstance();
9          data.set(1980, 10, 23);
10         empregado = new Empregado ("Clara Silva", data , 211456937 , null);
11         empregado.gerarMatricula();
12         diretor = new Diretor ("Marco Antônio", data , 901564098 , null);
13         diretor.gerarMatricula();
14         System.out.println ("A matrícula do Diretor é: " + diretor.recuperarMatricu
15         System.out.println ("A matrícula do Empregado é: " + empregado.recuperar
16     }
17 }
```

A execução desse código produz como saída:

A matrícula do Diretor é: E-096d9a3d-98e9-4af1-af61-a03d97525429

A matrícula do Empregado é: Matrícula não definida.

Observe que estamos invocando o método "*gerarMatricula ()*" com uma referência do tipo da superclasse (vide linha 3 do Código 14). Essa variável, porém, está se referindo a um objeto da subclasse (vide linha 12 do), e o método em questão possui uma versão especializada na classe "Diretor" (ela sobrescreve o método "*gerarMatricula ()*" da superclasse), conforme observamos na linha 8 do 13. Dessa maneira, durante a execução, o método da subclasse será chamado.

Outra forma de polimorfismo pode ser obtida por meio da **sobrecarga de métodos**.

A sobrecarga é uma característica que permite que métodos com o mesmo identificador, mas diferentes parâmetros, sejam implementados na mesma classe.

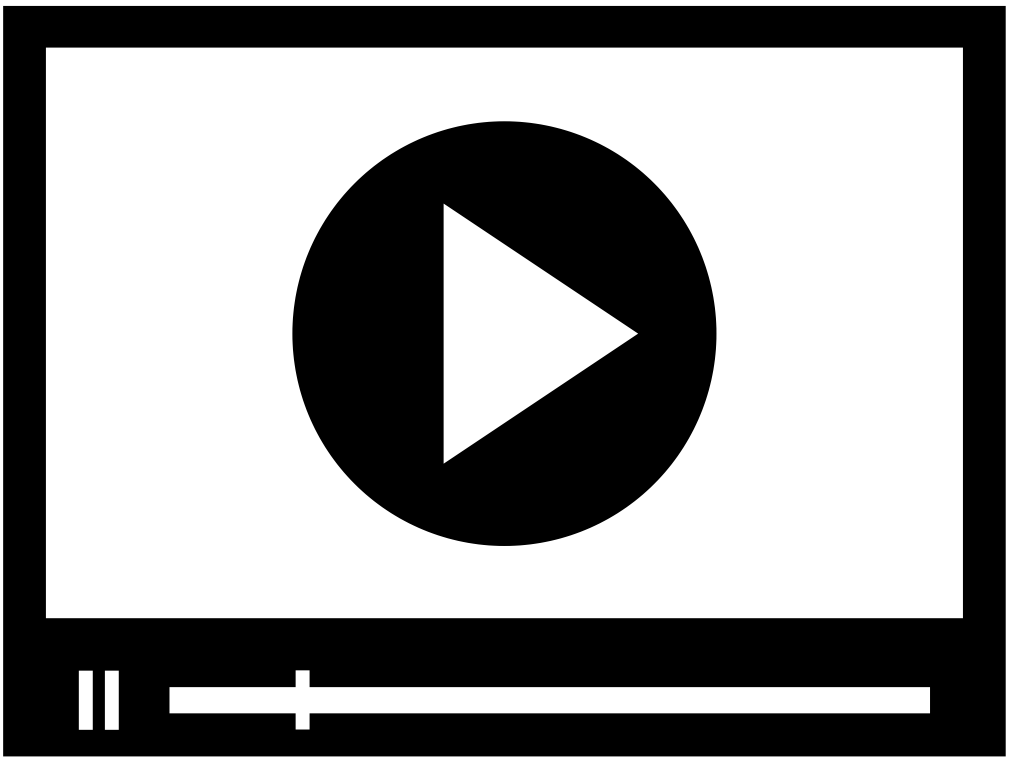
Ao usar parâmetros distintos em número ou quantidade, o programador permite que o compilador identifique qual método chamar. Veja o exemplo do código 14.

Código 14: Polimorfismo por sobrecarga de método.

```
1
2  public class Diretor extends Empregado {
3      //Atributos
```

```
4
5       //Métodos
6       public  Diretor(String nome, Calendar data_nascimento, long CPF, Endereco ende
7           super(nome, data_nascimento, CPF, endereco);
8       }
9       protected void gerarMatricula () {
10          matricula = "E-" + UUID.randomUUID( ).toString( );
11      }
12      protected void alterarMatricula () {
13          gerarMatricula ();
14      }
15      protected void alterarMatricula ( String matricula ) {
16          this.matricula = matricula;
17      }
    }
```

Nesse caso, uma chamada do tipo “*alterarMatricula ()*” invocará o método mostrado na linha 8 do Código 14. Caso seja feita uma chamada como “*alterarMatricula (“M-202100-1000)*”, o



HERANÇA E CLASSES ABSTRATAS EM JAVA

Neste vídeo, o especialista Marlos de Mendonça apresenta os conceitos de herança e classes abstratas, citando exemplos práticos com a linguagem Java.

VERIFICANDO O APRENDIZADO

MÓDULO 3

Descrever os mecanismos de agrupamento de objetos em Java

O propósito do agrupamento é permitir que, a partir de um universo de objetos, grupos de objetos afins sejam estabelecidos com base em determinado critério. Não se trata de um conceito de OO. Na verdade, é um conceito comum na interação com banco de dados. A cláusula “GROUB BY” da SQL, usada nas consultas ao banco de dados, faz exatamente isso. Esse é um dos motivos para o agrupamento nos interessar: a interação com banco de dados.

A lógica do agrupamento de objetos é simples: dado um universo de objetos e os critérios de particionamento, separá-los em grupos tais que em cada grupo todos os objetos possuam o mesmo valor para os critérios escolhidos. Se a lógica é simples, a implementação tem nuances que precisam ser observadas, como a checagem de referência para evitar acessos ilegais (*NullPointerException*). Isso porque, para fazer o particionamento, teremos de utilizar uma lista de listas ou outra estrutura de dados similar.

Apesar dos pontos levantados, é perfeitamente possível implementar o particionamento de objetos. Isso pode ser feito em poucas linhas (se você se valer das estruturas oferecidas pela API Java). De fato, essa era a maneira de se agrupar objetos até o lançamento da Java 8. A partir daí, a API Java passou a oferecer um mecanismo de agrupamento, simplificando o trabalho de desenvolvimento.

IMPLEMENTANDO O AGRUPAMENTO DE OBJETOS

No agrupamento, o estado final desejado é ter os objetos agrupados, e cada agrupamento deve estar mapeado para a chave usada como critério. Em outras palavras, buscamos construir uma função tal que, a partir de um universo de objetos de entrada, tenha como saída “n” pares ordenados formados pela chave de particionamento e a lista de objetos agrupados: F (U) = { [k_1 , <lista agrupada>, [k2 , <lista agrupada> , ..., [k_n , <lista agrupada> }.

A Figura 5 dá uma ideia do que se pretende.

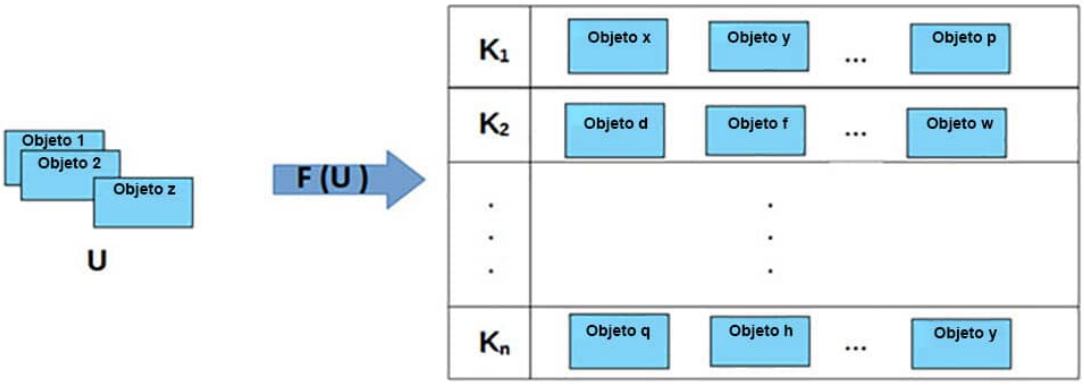


Figura 5: Representação gráfica do agrupamento de objetos.

Felizmente, a Java API oferece estruturas que facilitam o trabalho. Para manter e manipular os objetos usaremos o container “*List*”, que cria uma lista de objetos. Essa estrutura já possui métodos de inserção e remoção e pode ser expandida ou reduzida conforme a necessidade.

Para mantermos os pares de particionamento, usaremos o container “*Map*”, que faz o mapeamento entre uma chave e um valor. No nosso caso, a chave é o critério de particionamento e o valor é a lista de objetos particionados, exatamente conforme vemos na Figura 5. A estrutura “*Map*”, além de possuir métodos que nos auxiliarão, não permite a existência de chaves duplicadas. Vejamos, então, um exemplo.

Código 15: Classe "Aluno" modificada.

```
1
2  public class Aluno {
3      //Atributos
4      private String matricula , nome , naturalidade;
5
6      //Métodos
7      public Aluno ( String nome , String naturalidade ) {
```

```
        this.nome = nome;
        this.naturalidade = naturalidade;
    }
        @Override
        public String toString () {
            return String.format("%s(%s)", nome , naturalidade );
        }
    }
```

Código 16: Classe "Escola" – agrupamento de objetos pelo programador.

```
class Escola {
    //Atributos
    private String nome, CNPJ;
    private Endereco endereco;
    private List departamentos;
    private List discentes;

    //Métodos
    public Escola ( String nome , String CNPJ) {
        this.nome = nome;
        this.CNPJ = CNPJ;
        this.departamentos = new ArrayList<Departamento> ( );
        this.discentes = new ArrayList<Aluno> ( );
    }
    public void criarDepartamento ( String nomeDepartamento ) {
        departamentos.add ( new Departamento ( nomeDepartamento ) );
    }
    public void fecharDepartamento ( Departamento departamento ) {
        departamentos.remove ( departamento );
    }
    public void matricularAluno ( Aluno novoAluno ) {
        discentes.add ( novoAluno );
    }
    public void trancarMatriculaAluno ( Aluno aluno ) {
        discentes.remove ( aluno );
    }
    public void agruparAlunos ( ) {
        Map < String , List < Aluno > > agrupamento = new HashMap <> (
        for ( Aluno a : discentes ) {
            if(!agrupamento.containsKey ( a.recuperarNaturalidade ( ) )) {
                agrupamento.put( a.recuperarNaturalidade ( ) , new ArrayList<>
            }
        agrupamento.get(a.recuperarNaturalidade ( ) ).add(a);
        }
```

```
36        System.out.println ("Resultado do agrupamento por naturalidade: " + agrupamen
37        }
    }
```

Código 17: Classe Principal.

```
1   public class Principal {
2         //Atributos
3         private static Aluno aluno1 , aluno2 , aluno3 , aluno4 , aluno5 , aluno6
4         private static Escola escola;
5
6         //Método main
7         public static void main (String args[]) {
8               escola = new Escola ( "Escola Pedro Álvares Cabral" , "42.336.174/0006
9               criarAlunos ();
10        matricularAlunos ();
11        escola.agruparAlunos();
12  }
13
14        //Métodos
15        private static void criarAlunos ( ) {
16              aluno1 = new Aluno ( "Marco Antônio" , "Rio de Janeiro" );
17              aluno2 = new Aluno ( "Clara Silva" , "Rio de Janeiro" );
18              aluno3 = new Aluno ( "Marcos Cintra" , "Sorocaba" );
19              aluno4 = new Aluno ( "Ana Beatriz" , "Barra do Pirai" );
20              aluno5 = new Aluno ( "Marcio Gomes" , "São Paulo" );
21              aluno6 = new Aluno ( "João Carlos" , "Sorocaba" );
22              aluno7 = new Aluno ( "César Augusto" , "São Paulo" );
23              aluno8 = new Aluno ( "Alejandra Gomez" , "Madri" );
24              aluno9 = new Aluno ( "Castelo Branco" , "São Paulo" );
25        }
26        private static void matricularAlunos ( ) {
27              escola.matricularAluno(aluno1);
28              escola.matricularAluno(aluno2);
29              escola.matricularAluno(aluno3);
30              escola.matricularAluno(aluno4);
31              escola.matricularAluno(aluno5);
32              escola.matricularAluno(aluno6);
33              escola.matricularAluno(aluno7);
34              escola.matricularAluno(aluno8);
35              escola.matricularAluno(aluno9);
36        }
37  }
```

É o Código 16 que merece nossa atenção. Na linha 6, é criada uma lista de objetos do tipo Aluno. Para aplicarmos nossa função de agrupamento, precisaremos varrer essa lista, fazendo a separação de cada objeto encontrado segundo o critério de agrupamento.

O método de agrupamento encontra-se na linha 27. Na linha 28, temos a declaração de uma estrutura do tipo "*Map*" e a instanciação da classe pelo objeto "*agrupamentoPorNaturalidade*". Podemos observar que será mapeado um objeto do tipo "*String*" a uma lista de objetos do tipo "*Aluno*" ("*Map < String , List < Aluno > >*").

Em seguida, na linha 29, o laço implementa a varredura sobre toda a lista. A cada iteração, o valor da variável "*naturalidade*" é recuperado, e a função "*containsKey*" verifica se a chave já existe no mapa. Se não existir, ela é inserida. A linha 30, finalmente, adiciona o objeto à lista correspondente à chave existente no mapa.

Veja a saída:

Resultado do agrupamento por naturalidade: {

São Paulo=[Marcio Gomes(São Paulo), César Augusto(São Paulo), Castelo Branco(São Paulo)],

Rio de Janeiro=[Marco Antônio(Rio de Janeiro), Clara Silva(Rio de Janeiro)],

Madri=[Alejandra Gomez(Madri)],

Sorocaba=[Marcos Cintra(Sorocaba), João Carlos(Sorocaba)],

Barra do Pirai=[Ana Beatriz(Barra do Pirai)]

}

Vemos que nossa função agrupou corretamente os objetos. A chave é mostrada à esquerda do sinal de "=" e, à direita, entre colchetes, estão as listas de objetos, nas quais cada objeto encontra-se separado por vírgula.

AGRUPANDO OBJETOS COM A CLASSE "*COLLECTORS*" DA API JAVA

A classe "*Collectors*" é uma classe da API Java que implementa vários métodos úteis de operações de redução, como, por exemplo, o agrupamento dos objetos em coleções, de acordo com a Oracle America Inc. (2021). Com o auxílio da API Java, não precisaremos varrer

a lista de objetos. Vamos transformar esses objetos em um fluxo (“*stream*”) que será lido pela classe “*Collectors*”, a qual realizará todo o trabalho de classificação dos objetos, nos devolvendo um mapeamento como na situação anterior.

A operação de agrupamento é feita pelo método “*groupingBy*”.

Esse método possui três variantes sobrecarregadas, cujas assinaturas exibimos a :

```
1  static <T, K> Collector<T,?,Map>> groupingBy(Function<? super T,? extends K>
2  static <T, K, D, A, M extends Map<K, D>> Collector<T,?,M> groupingBy(Functi
3  static <T, K, A, D> Collector<T,?,Map>  groupingBy(Function<? super T,? exter
```

O agrupamento da classe “*Collectors*” usa uma função de classificação que retorna um objeto classificador para cada objeto no fluxo de entrada. Esses objetos classificadores formam o universo de chaves de particionamento. Isto é, são os rótulos ou as chaves de cada grupo ou coleção de objetos formados. Conforme o agrupador da classe “*Collectors*” vai lendo os objetos do fluxo de entrada, ele vai criando coleções de objetos correspondentes a cada classificador. O resultado é um par ordenado (Chave, Coleção), que é armazenado em uma estrutura de mapeamento “*Map*”.

Na assinatura 1, identificamos com mais facilidade que o método “*groupingBy*” recebe como parâmetro uma referência para uma função capaz de mapear T em K.

A cláusula “static <T, K> Collector<T,?,Map<K,List<T>>>” é o método (“*Collector*”) que retorna uma estrutura “*Map*”, formada pelos pares “*K*” e uma lista de objetos “*T*”. “*K*” é a chave de agrupamento e “*T*”, obviamente, um objeto agrupado. Então a função cuja referência é passada para o método “*groupingBy*” é capaz de mapear o objeto na chave de agrupamento.

A modificação trazida pela segunda assinatura é a possibilidade de o programador decidir como a coleção será criada na estrutura de mapeamento. Ele pode decidir usar outras estruturas ao invés de “*List*”, como a “*Set*”, por exemplo.

A terceira versão é a mais genérica. Nela, além de poder decidir a estrutura utilizada para implementar as coleções, o programador pode decidir sobre qual mecanismo de “*Map*” será utilizado para o mapeamento.

Vejamos o 8, que é uma versão modificada do Nessa nova versão, o método “*agruparAlunos*” foi reescrito usando a primeira assinatura de “*groupingBy*”.

Código 18: Agrupamento com o uso do método "groupingBy" (assinatura 1).

```
1
2  class Escola {
       //Atributos
```

```
3       private String nome, CNPJ;
4       private Endereco endereco;
5       private List departamentos;
6       private List discentes;
7
8       //Métodos
9       public Escola ( String nome , String CNPJ) {
10          this.nome = nome;
11          this.CNPJ = CNPJ;
12          this.departamentos = new ArrayList ( );
13          this.discentes = new ArrayList<> ( );
14      }
15      public void criarDepartamento ( String nomeDepartamento ) {
16          departamentos.add ( new Departamento ( nomeDepartamento ) );
17      }
18      public void fecharDepartamento ( Departamento departamento ) {
19          departamentos.remove ( departamento );
20      }
21      public void matricularAluno ( Aluno novoAluno ) {
22          discentes.add ( novoAluno );
23      }
24      public void trancarMatriculaAluno ( Aluno aluno ) {
25          discentes.remove ( aluno );
26      }
27      public void agruparAlunos ( ) {
28          Map < String , List < Aluno > > agrupamento = discentes.stream().
29          System.out.println ("Resultado do agrupamento por naturalidade: ");
30          agrupamento.forEach (( String chave , List < Aluno > lista) -> Sys
31      }
32  }
```

A execução desse código produz a saída a seguir:

Resultado do agrupamento por naturalidade:

São Paulo = [Marcio Gomes(São Paulo), César Augusto(São Paulo), Castelo Branco(São Paulo)]

Rio de Janeiro = [Marco Antônio(Rio de Janeiro), Clara Silva(Rio de Janeiro)]

Madri = [Alejandra Gomez(Madri)]

Sorocaba = [Marcos Cintra(Sorocaba), João Carlos(Sorocaba)]

Barra do Pirai = [Ana Beatriz(Barra do Pirai)]

Comparando essa saída com a produzida pelo Código 16, podemos perceber que os agrupamentos de objetos são rigorosamente os mesmos.

Analisemos a linha 28 do Código 18.

1

Inicialmente, identificamos que “*agrupamento*” é uma estrutura do tipo “*Map*” que armazena pares do tipo “*String*” e lista de “*Aluno*” (“*List<Aluno>*”).

O objeto “*String*” é a chave de agrupamento, enquanto a lista compõe a coleção de objetos.

2

3

A seguir, vemos o uso do método “*stream*”, que produz um fluxo de objetos a partir da lista “*discentes*” e o passa para “*Collectors*”.

Por fim, “*groupingBy*” recebe uma referência para o método que permite o mapeamento entre o objeto e a chave de agrupamento.

4

No nosso exemplo, esse método (“*Aluno::recuperarNaturalidade*”) é o que retorna o valor da variável “naturalidade” dos objetos alunos. Na prática, estamos agrupando os alunos pela sua naturalidade. Essa função é justamente a que mapeia o objeto “*Aluno*” à sua naturalidade (chave de agrupamento).

Vejamos agora o uso das demais assinaturas. Por simplicidade, iremos mostrar apenas a sobrecarga do método “*agruparAluno*”, uma vez que o restante da classe permanecerá inalterado.

Código 19: Agrupamento com o uso do método "groupingBy" (assinatura 2).

```
public void agruparAlunos ( int a ) {
    Map < String , Set < Aluno > > agrupamento = discentes.stream().collec
    System.out.println ("Resultado do agrupamento por naturalidade: ");
    agrupamento.forEach (( String chave , Set < Aluno > conjunto) -> Syste
}
```

Podemos ver que o Código 19 utiliza uma estrutura do tipo “*Set*”, em vez de “*List*”, para criar as coleções. Consequentemente, o método “*groupingBy*” passou a contar com mais um argumento − “*Collectors.toSet()*” – que retorna um *“Collector*” que acumula os objetos em uma estrutura “*Set*”. A saída é a mesma mostrada para a execução do código 18.

O Código 20 mostra uma sobrecarga do método “*agruparAlunos*”, que usa a terceira assinatura de “*groupingBy*”.

Código 20: Agrupamento com o uso do método "groupingBy" (assinatura 3).

```
public void agruparAlunos ( double a ) {
    Map < String , Set < Aluno > > agrupamento = discentes.stream().collec
    System.out.println ("Resultado do agrupamento por naturalidade: ");
    agrupamento.forEach (( String chave , Set < Aluno > conjunto) -> Syste
}
```

A diferença sintática para a segunda assinatura é apenas a existência de um terceiro parâmetro no método “*groupingBy*”: “*TreeMap::new*”. Esse parâmetro vai instruir o uso do mecanismo “*TreeMap*” na instanciação de “*Map*” (“*agrupamento*”).

Veja agora a saída desse código:

Resultado do agrupamento por naturalidade:

Barra do Pirai = [Ana Beatriz(Barra do Pirai)]

Madri = [Alejandra Gomez(Madri)]

Rio de Janeiro = [Clara Silva(Rio de Janeiro), Marco Antônio(Rio de Janeiro)]

Sorocaba = [João Carlos(Sorocaba), Marcos Cintra(Sorocaba)]

São Paulo = [Castelo Branco(São Paulo), César Augusto(São Paulo), Marcio Gomes(São Paulo)]

Podemos notar que a ordem das coleções está diferente do caso anterior. Isso porque o mecanismo "*TreeMap*" mantém as suas entradas ordenadas. Todavia, podemos perceber que os agrupamentos são iguais.

Uma observação sobre o método "*groupingBy*" é que ele não é otimizado para execuções concorrentes. Caso você precise trabalhar com agrupamento de objetos e concorrência, a API Java fornece uma versão apropriada para esse caso.

Todas as três variantes possuem uma contraparte chamada "*groupingByConcurrent*", destinada ao uso num ambiente *multithread*. As assinaturas, os parâmetros e o retorno – e, portanto, o uso − são exatamente os mesmos que na versão para desenvolvimento não paralelizado.

COLEÇÕES

Coleções, por vezes chamadas de *Containers,* são objetos capazes de agrupar múltiplos elementos em uma única unidade.

Sua finalidade é armazenar, manipular e comunicar dados agregados, de acordo com o Oracle America Inc., (2021). Neste módulo, nós as usamos ("*List*" e "*Set*") para armazenar os agrupamentos criados, independentemente dos métodos que empregamos.

Ainda de acordo com a Oracle America Inc. (2021), a API Java provê uma interface de coleções chamada *Collection Interface*, que encapsula diferentes tipos de coleção: "*Set*", "*List*", "*Queue* e "*Deque*" ("*Map*" não é verdadeiramente uma coleção) . Há, ainda, as coleções "*SortedSet*" e "*SortedMap*", que são, essencialmente, versões ordenadas de "*Set*" e "*Map*", respectivamente.

As interfaces providas são genéricas, o que significa que precisarão ser especializadas. Vimos isso ocorrer na linha 2 do 20, por exemplo, quando fizemos "*Set* < *Aluno* >". Nesse caso, instruímos o compilador a especializar a coleção "*Set*" para a classe "Aluno".

Cada coleção encapsulada possui um mecanismo de funcionamento, apresentado brevemente a seguir:

SET

Trata-se de uma abstração matemática de conjuntos. É usada para representar conjuntos e não admite elementos duplicados.

LIST

Implementa o conceito de listas e admite duplicidade de elementos. É uma coleção ordenada e permite o acesso direto ao elemento armazenado, assim como a definição da posição onde armazená-lo. O conceito de Vetor fornece uma boa noção de como essa coleção funciona.

QUEUE

Embora o nome remeta ao conceito de filas, trata-se de uma coleção que implementa algo mais genérico. Uma "*Queue*" pode ser usada para criar uma fila (FIFO), mas também pode implementar uma lista de prioridades, na qual os elementos são ordenados e consumidos segundo a prioridade e não na ordem de chegada. Essa coleção admite a criação de outros tipos de filas com outras regras de ordenação.

DEQUE

Implementa a estrutura de dados conhecida como Deque (*Double Ended Queue*). Pode ser usada como uma fila (FIFO) ou uma pilha (LIFO). Admite a inserção e a retirada em ambas as extremidades.

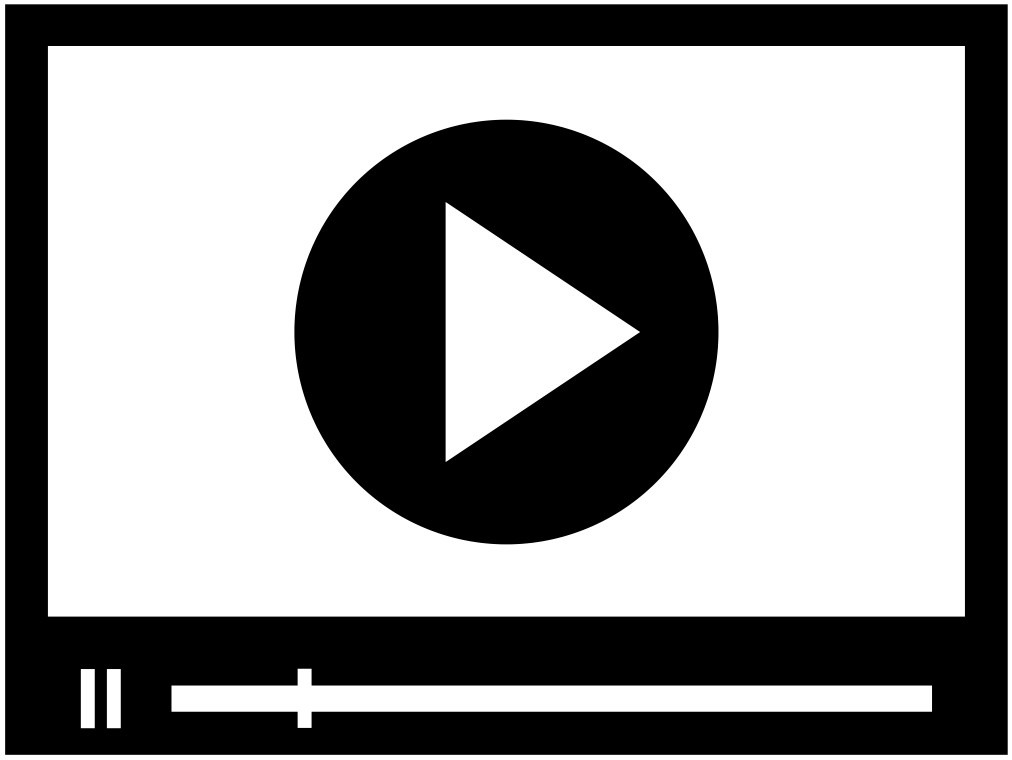
Como dissemos, "*Map*" não é verdadeiramente uma coleção. Esse tipo de classe cria um mapeamento entre chaves e valores, conforme vimos nos diversos exemplos da subseção anterior. Além de não admitir chaves duplicadas, a interface "*Map*" mapeia uma chave para um único valor.

Se observarmos a linha 2 do Código 20 por exemplo, veremos que a chave é mapeada em um único objeto que, no caso, é uma lista de objetos "*Aluno*". Uma tabela *hash* fornece uma boa noção do funcionamento de "*Map*".

Não é nosso propósito examinar em detalhe cada uma dessas coleções. A melhor forma de aprender sobre elas é consultando a documentação da API, mas uma noção de seu funcionamento foi mostrada nos exemplos anteriores. As linhas 4 do 20 e 29 do mostram formas de percorrer as estruturas.

Repare que o método "*foreach*", no primeiro caso, percorre todo o mapeamento sem que seja necessária a criação de um laço. Da mesma maneira, no segundo exemplo, nenhum controle de iteração é necessário para que o laço funcione e percorra todos os elementos da lista. Esses são apenas alguns exemplos das funcionalidades providas por essas estruturas.

Na linha 31 do 16, o método “*put*” é utilizado para inserir um elemento no mapeamento. A linha 33 desse código é ainda mais interessante: o método “*get*” é usado para recuperar a referência do objeto mapeado pela chave passada com parâmetro em “*get*”. Esse objeto é uma lista, que utiliza o método “*add*” para adicionar o novo elemento.



COLEÇÕES EM JAVA

Neste vídeo, o especialista Marlos de Mendonça explica o conceito de coleções e apresenta exemplos práticos com a linguagem Java.

VERIFICANDO O APRENDIZADO

MÓDULO 4

⦿ **Reconhecer os ambientes de desenvolvimento em Java e as principais estruturas da linguagem**

Neste módulo, falaremos sobre ambientes de desenvolvimento Java. Começaremos abordando alguns aspectos comerciais e prosseguiremos fazendo uma breve comparação com a linguagem C/C++. Em seguida, apresentaremos os ambientes de desenvolvimentos − sem esgotá-los −, alguns *Integrated Develompment Environment* (IDE) e a estrutura e os principais comandos de um programa Java.

JAVA *VERSUS* C/C++: UM BREVE COMPARATIVO

Segundo seu criador, Bjarne Stroustrup (2020), a linguagem C++ e a Java não deveriam ser comparadas, pois têm condicionantes de desenvolvimento distintas. Stroustrup aponta, também, que muito da simplicidade de Java é uma ilusão e que, ao longo do tempo, a linguagem se tornará cada vez mais dependente de implementações.

As diferenças mencionadas por Stroustrup se mostram presentes, por exemplo, no fato de a C++ ter uma checagem estática fraca, enquanto a Java impõe mais restrições. A razão de C++ ser assim é a performance ao interfacear com o hardware, enquanto Java privilegia um desenvolvimento mais seguro contra erros de programação. Nesse sentido, James Gosling, um dos desenvolvedores da Java, aponta a checagem de limites de *arrays*, feita pela Java, como um ponto forte no desenvolvimento de código seguro, algo que a C++ não faz, conforme explica Sutter (2021).

Talvez um dos pontos mais interessantes seja a questão da portabilidade.

A C++ buscou alcançá-la através da padronização.

Em 2021, a linguagem C++ segue o padrão internacional ISO/IEC 14882:2020, definido pelo *International Organization for Standardization,* e possui a *Standard Template Library*, uma biblioteca padrão que oferece várias funcionalidades adicionais como *containers*, algoritmos, iteradores e funções.

Logo, um programa C++, escrito usando apenas os padrões e as bibliotecas padrões da linguagem, em tese, possui portabilidade.

A portabilidade é um dos pontos fortes da Java.

Todavia, conforme Stroustrup (2020) aponta:

A Java não é uma linguagem independente de plataforma, ela é uma plataforma!

O motivo dessa afirmação é que um software Java não é executado pelo hardware, mas sim pela Máquina Virtual Java, que é uma plataforma emulada que abstrai o hardware subjacente. A JVM expõe sempre para o programa a mesma interface, enquanto ela própria é um software acoplado a uma plataforma de hardware específica. Se essa abordagem permite ao software Java rodar em diferentes hardwares, ela traz considerações de desempenho que merecem atenção.

Mas e a linguagem C?

Comparar Java com C é comparar duas coisas totalmente diferentes que têm em comum apenas o fato de serem linguagens de programação. A linguagem C tem como ponto forte a interação com sistemas de baixo nível. Por isso, é muito utilizada em drivers de dispositivo.

A distância que separa Java de C é ainda maior do que a distância entre Java e C++.

Java e C++ são linguagens OO.

A linguagem C é aderente ao paradigma de programação estruturado, ou seja, C não possui conceito de classes e objetos.

Em contrapartida, Java é uma linguagem puramente OO. Qualquer programa em Java precisa ter ao menos uma classe e nada pode existir fora da classe (na verdade, as únicas declarações permitidas no arquivo são classe, interface ou enum).

Isso quer dizer que não é possível aplicar o paradigma estruturado em Java.

Comparar Java e C, portanto, serve apenas para apontar as diferenças dos paradigmas OO e estruturado.

AMBIENTES DE DESENVOLVIMENTO JAVA

Para falarmos de ambientes de desenvolvimento, precisamos esclarecer alguns conceitos, como a Máquina Virtual Java (**MVJ**) – em inglês, *Java Virtual Machine* (JVM) –, o *Java Runtime Environment* (**JRE**) e o principal para nosso intento: o *Java Development Kit* (**JDK**).

Comecemos pela **JVM**, que já dissemos ser uma abstração da plataforma. Conforme explicado pelo Oracle America Inc. (2015), trata-se de uma especificação feita inicialmente pela Sun Microsystems, atualmente incorporada pela Oracle. A abstração procura ocultar do software Java detalhes específicos da plataforma, como o tamanho dos registradores da CPU ou sua arquitetura – RISC ou CISC.

A JVM é um software que implementa a especificação mencionada e é sempre aderente à plataforma física. Contudo, ela provê para os programas uma plataforma padronizada, garantindo que o código Java sempre possa ser executado, desde que exista uma JVM. Ela não executa as instruções da linguagem Java, mas sim os bytecodes gerados pelo compilador Java.

Adicionalmente, para que um programa seja executado, são necessárias bibliotecas que permitam realizar operações de entrada e saída, entre outras. Assim, o conjunto dessas bibliotecas e outros arquivos formam o ambiente de execução juntamente com a JVM. Esse ambiente é chamado de JRE, que é o elemento que gerencia a execução do código, inclusive chamando a JVM.

Com o **JRE**, pode-se executar um código Java, mas não se pode desenvolvê-lo. Para isso, precisamos do JDK, que é um ambiente de desenvolvimento de software usado para criar aplicativos e applets. O **JDK** engloba o JRE e mais um conjunto de ferramentas de desenvolvimento, como um interpretador Java (*java*), um compilador (*javac*), um programa de arquivamento (*jar*), um gerador de documentação (*javadoc*) e outros.

Dois JDK muito utilizados são o Oracle JDK e o OpenJDK. A Figura 6 mostra um aviso legal exibido durante a instalação do Oracle JDK (Java SE 15). Trata-se de um software cujo uso gratuito é restrito.

Figura 6: Aviso legal na instalação do Oracle JDK.

A Oracle também desenvolve uma implementação de referência livre chamada OpenJDK.

COMENTÁRIO

Há, contudo, diferenças entre o OpenJDK e o Oracle JDK (o coletor de lixo possui mais opções no último). Tais diferenças, porém, não impedem o uso do OpenJDK.

Uma vez instalado um ambiente de desenvolvimento, é possível começar o trabalho de desenvolvimento. Basta ter disponível um editor de texto. Os programas do ambiente podem ser executados a partir de chamadas feitas diretamente no console. Por exemplo, “javac aluno.java” irá compilar a classe “Aluno”, gerando um arquivo chamado “aluno.class”.

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Um *Integrated Development Environment,* ou ambiente integrado de desenvolvimento, é um software que reúne ferramentas de apoio e funcionalidades com o objetivo de facilitar e acelerar o desenvolvimento de software.

Ele normalmente engloba um editor de código, as interfaces para os softwares de compilação e um depurador, mas pode incluir também uma ferramenta de modelagem (para criação de classes e métodos), refatoração de código, gerador de documentação e outros.

O Netbeans é um IDE mantido pela The Apache Software Foundation e licenciado segundo a licença Apache versão 2.0. De acordo com o site do IDE, ele é o IDE oficial da Java 8, mas também permite desenvolver em HTML, JavaScript, PHP, C/C++, XML, JSP e Groovy. É um IDE multiplataforma que pode ter suas funcionalidades ampliadas pela instalação de plugins.

A Figura 7 mostra o IDE durante uma depuração.

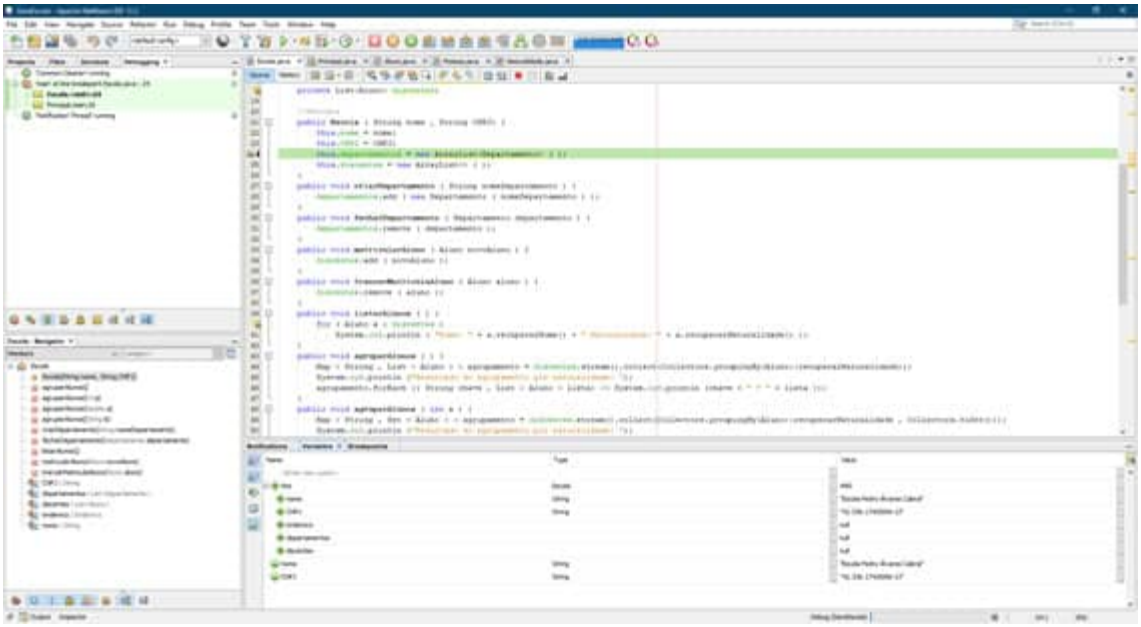


Figura 7: IDE Netbeans.

Apesar de possuir muitas funcionalidades, a parte de modelagem realiza apenas a declaração de classes, sem criação automática de construtor ou métodos, conforme se nota na Figura 8.

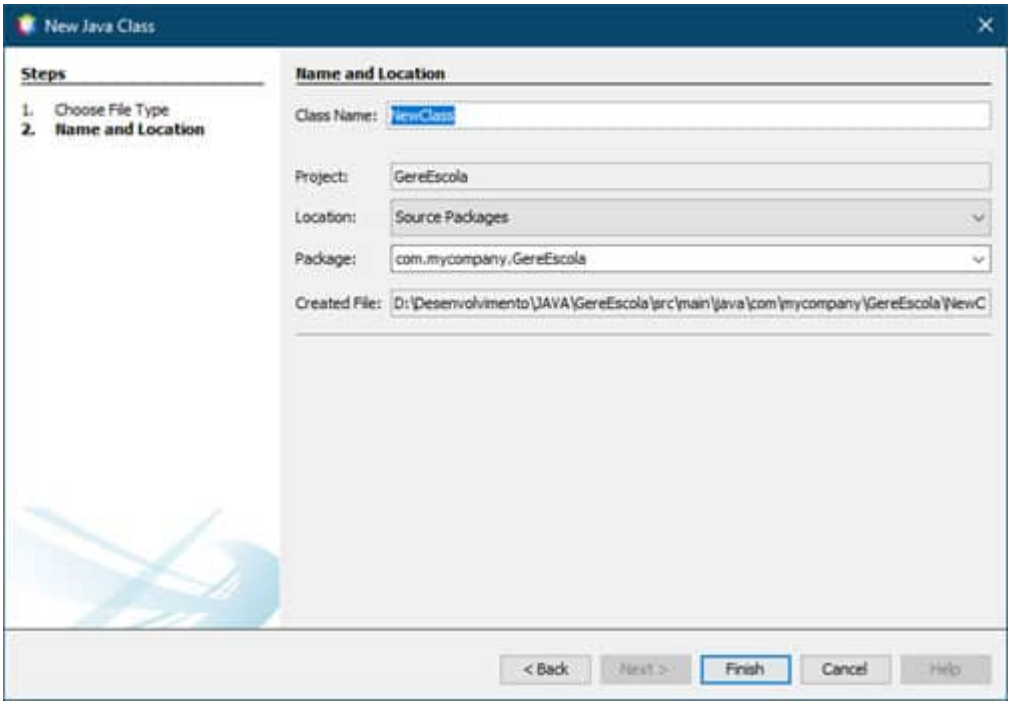


Figura 8: Criação de classe no Netbeans.

A parte de modelagem do IDE Eclipse, porém, é mais completa. Pode-se especificar o modificador da classe, declará-la como "*abstract*" ou "*final*", especificar sua superclasse e passar parâmetros para o construtor da superclasse automaticamente.

Veja a figura 9:

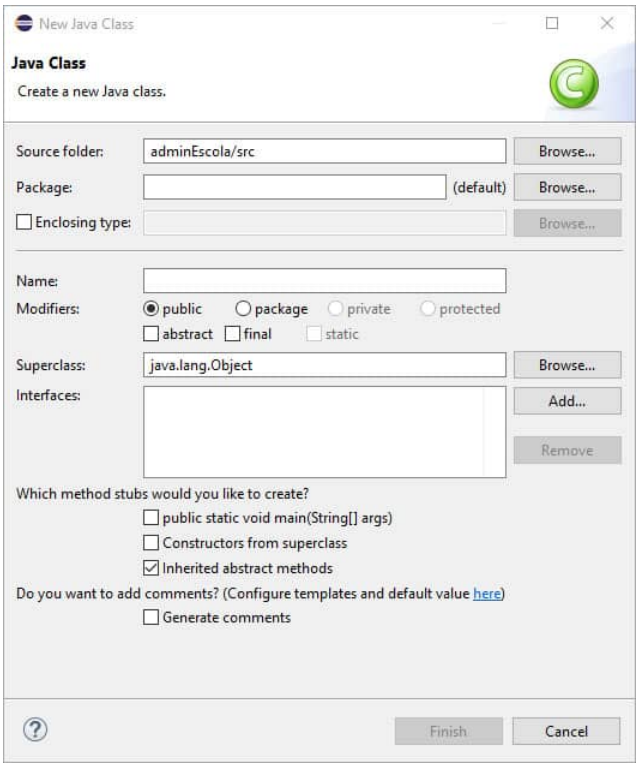


Figura 9: Criação de classe no Eclipse.

VOCÊ SABIA

O Eclipse é mantido pela Eclipse Foundation, que possui membros como IBM, Huawei, Red Hat, Oracle e outros.

Trata-se de um projeto de código aberto que, da mesma maneira que o Netbeans, suporta uma variedade de linguagens além da Java. O IDE também oferece suporte a plugins, e o editor de código possui características semelhantes ao do Netbeans. O Eclipse provê também um Web IDE chamado Eclipse Che, voltado para DevOps. Trata-se de uma ferramenta rica, que oferece terminal/SSH, depuradores, flexibilidade de trabalhar com execução multicontainer e outras características. A Figura 10 mostra um exemplo de depuração no Eclipse, e a Figura 11 mostra a tela do Eclipse Che.

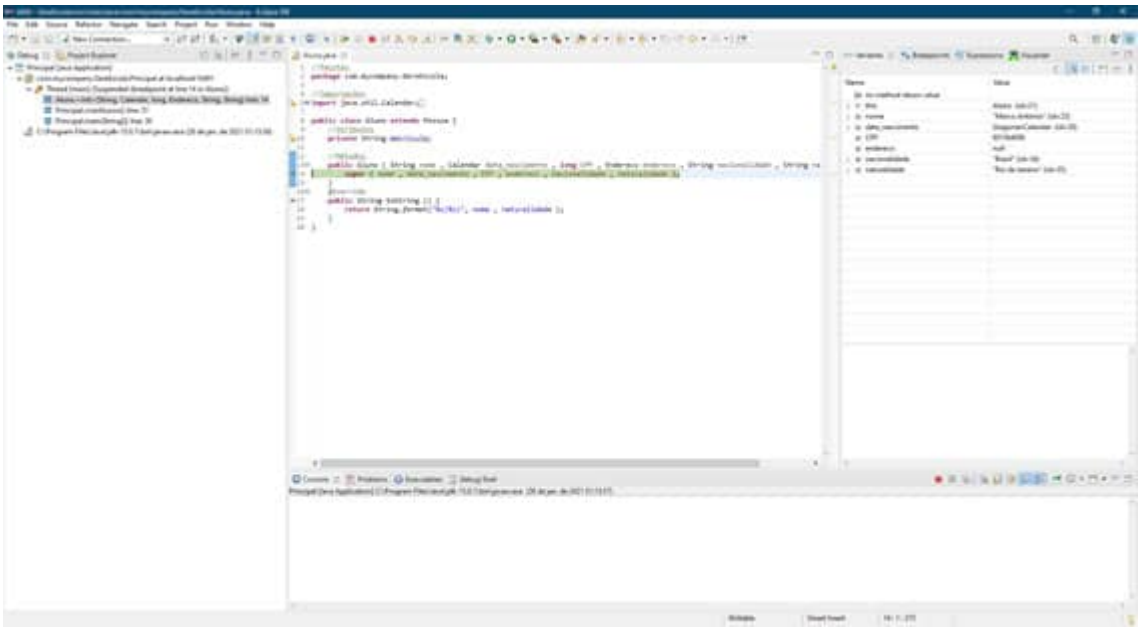


Figura 10: IDE Eclipse.

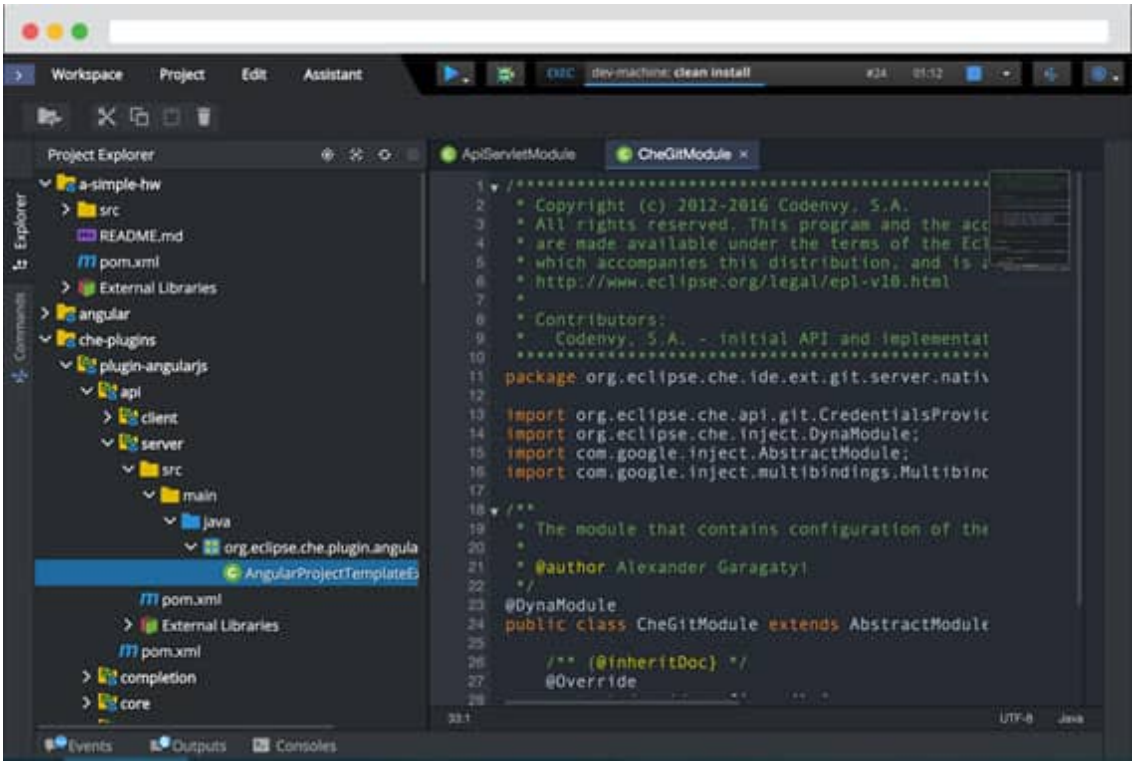


Figura 11: IDE Eclipse Che.

ESTRUTURA E PRINCIPAIS COMANDOS DE UM PROGRAMA EM JAVA

Como um programa em Java começa a sua execução?

Na linguagem C/C++, o ponto de entrada para a execução do programa é a função “*main”*.

Na Java, o ponto de entrada não é obrigatório para compilar o programa.

Isso quer dizer que, caso seu código possua apenas as classes “Aluno”, “Escola”, “Departamento” e “Pessoa”, por exemplo, ele irá compilar, mas não poderá ser executado.

Quando se trata de uma aplicação *standalone*, um ponto de entrada pode ser definido pelo método “*main*”. Entretanto, diferentemente de C/C++, esse método deverá pertencer a uma classe.

Veja o 21. A linha 3 mostra a função “*main*”, que define o ponto de entrada da aplicação. Esse será o primeiro método a ser executado pela JVM.

Código 21: Definição do ponto de entrada de um aplicativo Java.

```
1 public class Main {
2       public static void main (String args[]) {
3       //Código
4       }
5 }
```

A sintaxe mostrada na linha 3 é a assinatura do método "*main*". Qualquer modificação nela fará com que o método não seja mais reconhecido como ponto inicial de execução.

Mas há outro caso a ser considerado: as *Applets*, que são programas executados por outros programas, normalmente um navegador ou um *appletviewer*. Geralmente se apresentando embutidas em páginas HTML, as *Applets* não utilizam o método "*main*" como ponto de entrada.

Nesse caso, a classe deverá estender a classe "*Applet*", que fornece os métodos "*start*", "*init*", "*stop*" e "*destroy*". O método "*start*" é invocado quando a *Applet* é visitada. Em seguida, o método "*init*" é chamado no início da execução da *Applet*. Se o navegador sair da página onde a *Applet* está hospedada, é chamado o método "*stop*". Por fim, quando o navegador for encerrado, será chamado o método "*destroy*".

Outro elemento importante da estrutura de um programa Java são os métodos de acesso. Estes não são comandos Java e nem uma propriedade da linguagem, e sim consequência do encapsulamento. Eles apareceram em alguns códigos mostrados aqui, embora sem os nomes comumente usados em inglês.

Os **métodos de acesso** são as mensagens trocadas entre objetos para alterar seu estado. Eles, assim como os demais métodos e atributos, estão sujeitos aos modificadores de acesso, que já vimos anteriormente.

Todos os métodos utilizados para recuperar valor de variáveis ou para defini-los são métodos de acesso!

Na prática, é muito provável, mas não obrigatório, que um atributo dê origem a dois métodos: um para obter seu valor ("*get*") e outro para inseri-lo ("*set*"). Assim, no Código 8, as linhas 17 e 20 mostram métodos de acesso ao atributo "nome".

Vejamos agora alguns dos principais comandos em Java.

Vamos começar pelas estruturas de desvio, que realizam o desvio da execução do código com base na avaliação de uma expressão booleana. Java possui o tipo de dado "*boolean*", que pode assumir os valores "*true*" ou "*false*". Repare que, diferentemente de C/C++, um número inteiro não é convertido automaticamente pelo compilador em "*boolean*". Logo, o código 22 gera erro de compilação em Java.

Código 22: Estrutura de desvio − if.

```
1  public class Base {
2      private int temp = 1;
3
4      public void desvio (){
5          if ( temp )
6                  System.out.println("Verdade");
7          else
8                  System.out.println("Falsidade");
9      }
10 }
```

A sintaxe do comando "*if*" é a seguinte:

```
1  if ( < expressão> )
2      bloco;
3  [else
4      bloco;]
```

A expressão deve sempre ser reduzida a um valor booleano. Assim, se substituirmos a linha 2 do código 22 por "*private int temp =1, aux = 0;*", "*temp > aux*" é um exemplo de expressão válida. Da mesma maneira, se a substituição for por "*private boolean temp = true;*", a linha 5 do código se torna válida. Quando a expressão for verdadeira, executa-se o "bloco" que se segue ao comando "*if*". A cláusula "*else"* é opcional e desvia a execução para o "bloco" que a segue quando a expressão é falsa. Caso ela seja omitida, o programa continua na próxima linha. Quando "bloco" for composto por um único comando, o uso de chaves ("{ }") não é necessário, mas é obrigatório se houver mais de uma instrução. O comando "*if*" pode ser aninhado, como vemos no código 23.

Código 23: "if" aninhado.

```
1
2  class Base {
3          public static void main(String args[]) {
4              int mes = 4;
5              String trimestre;
6              if ( mes == 1 || mes == 2 || mes == 3 )
7                      trimestre = "Primeiro";
8              else if ( mes == 4 || mes == 5 || mes == 6 )
9                      trimestre = "Segundo";
10             else if ( mes == 7 || mes == 8 || mes == 9 )
11                 trimestre = "Terceiro";
12         else if ( mes == 10 || mes == 11 || mes == 12)
```

```
                trimestre = "Quarto";
            else
                trimestre = "Erro!";
            System.out.println("Abril está no " + trimestre + " trimestre.");
        }
    }
```

Embora o aninhamento do comando “*if*” seja útil, muitos casos podem ser melhor resolvidos pelo comando “*switch*”. A sintaxe de “*switch*” é a seguinte:

Código 24:

```
switch ( < expressão> ) {
    case < valor 1>:
        bloco;
        break;
.
.
.
    case :
        bloco;
    break;
    default:
        bloco;
            break;
}
```

No caso do comando “*switch*”, a expressão pode ser, por exemplo, uma “*String*”, “*byte*”, “*int*”, “*char*” ou “*short*”. O valor da expressão será comparado com os valores em cada cláusula “*case*” até que um casamento seja encontrado. Então, o “bloco” correspondente ao “*case*” coincidente é executado. Se nenhum valor coincidir com a expressão, é executado o bloco da cláusula “*default*”.

É interessante notar que tanto as cláusulas “*break*” quanto a “*default*” são opcionais, mas seu uso é uma boa prática de programação. Veja um exemplo no código 25. Além disso, “*switch*” também pode ser aninhado. Para isso, basta usar um comando “*switch*” em um dos blocos mostrados na sintaxe dele.

Código 25: Estrutura de desvio − "switch".

```

public class Base {
    private String linguagem = “JAVA”;
```

```
3
4       public void desvio (){
5           switch ( linguagem ) {
6               case ( "C" ):
7                       System.out.println("Suporta apenas programação estruturada"
8                       break;
9               case ( "C++" ):
10                      System.out.println("Suporta programação estruturada e orienta
11                      break;
12              case ( "JAVA "):
13                      System.out.println("Suporta apenas programação orientada a
14                      break;
15              default:
16                      System.out.println("Erro!");
17                      break;
18          }
19      }
20  }
```

Outro grupo de comandos importante são as estruturas de repetição. As três estruturas providas pela Java são “*while*”, “*do-while*” e o laço “*for*”.

A estrutura “*while*” tem a seguinte sintaxe:

Código 26

```
1   while ( < expressão> ){
2       bloco;
3   }
```

Observe que, nesse comando, antes de executar o “bloco” a expressão é avaliada. O “bloco” será executado apenas se a expressão for verdadeira e, nesse caso, a execução se repete até que a expressão assuma valor falso. O exemplo do Código 27 conta de 1 até 9, pois quando “*controle*” assume valor 10, a expressão “*controle* < *10*” se torna falsa e as linhas 5 e 6 não são executadas.

Código 27 Estrutura de repetição − "while".

```
1   class Base {
2       public static void main (String args []) {
3           private int controle = 0;
4           while ( controle < 10 ) {
5           System.out.println(controle);
6           controle++;
```

```
            }
        }
}
```

A estrutura “*do-while*” tem funcionamento parecido. Todavia, ao contrário de “*while*”, nessa estrutura o bloco sempre será executado ao menos uma vez. Veja sua sintaxe:

Código 28

```
do {
    bloco;
} while ( < expressão> );
```

Como vemos pela sintaxe, o primeiro “bloco” é executado e, somente após, a expressão é avaliada. Se for verdadeira, a repetição continua até que a expressão seja falsa. Caso contrário, a execução do programa continua na linha seguinte.

Vejamos, no Código 29, o código 27 modificado para o uso de “*do-while*”.

Código 29: Estruturas de repetição − "do-while".

```
class Base {
    public static void main (String args []) {
        private int controle = 0;
        do {
            System.out.println(controle);
            controle++;
        }while ( controle < 10 );
    }
}
```

A última estrutura de repetição que veremos é o laço “*for*”. A sua sintaxe é:

Código 30:

```
for ( inicialização ; expressão ; iteração ) {
    bloco;
}
```

O parâmetro “inicialização” determina a condição inicial e é executado assim que o laço se inicia.

A “expressão” é avaliada em seguida.

Se for verdadeira, a repetição ocorre até que se torne falsa.

Caso seja falsa no início, o “bloco” não é executado nenhuma vez e a execução do programa saltará para a próxima linha após o laço.

O último item, “iteração”, é executado após a execução do “bloco”.

• No caso de uma “expressão” falsa, “iteração” não é executado.

Vejamos, no Código 31, um exemplo a partir da modificação do Código *16*.

Código 31: Estruturas de repetição − "for".

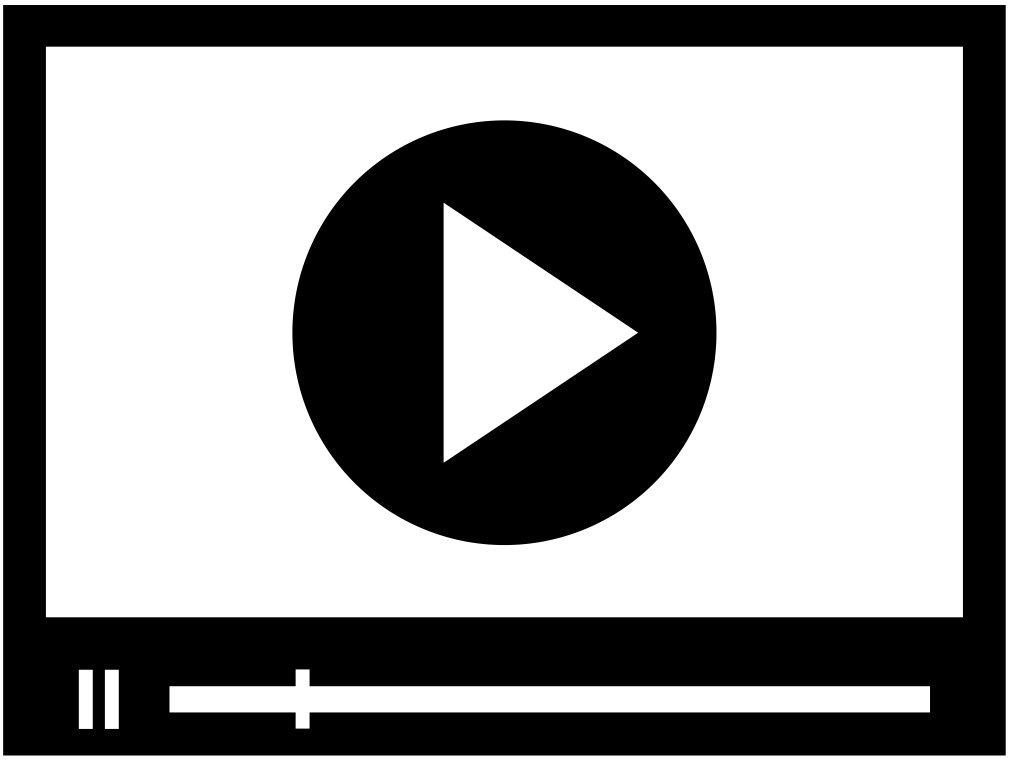
```java
class Base {
    public static void main (String args []) {
        for ( int controle = 0 ; controle < 10 ; controle++ ) {
            System.out.println(controle);
            controle++;
        }
    }
}
```

O laço “*for*” passou a ter uma segunda forma, desde a Java 5, chamada de “*for-each*”. Esse laço é empregado para iterar sobre uma coleção de objetos, de maneira sequencial, como vimos ao estudarmos o agrupamento de objetos.

Um exemplo do seu emprego pode ser visto na linha 29 do código 16. A sua sintaxe é:

```
for ( “tipo” “iterador” : “coleção ) {
    bloco;
;
```

Por fim, observamos que quando o “bloco” for formado por apenas uma instrução, o uso das chaves (“{ }”) é opcional.



JAVA X C/C++

Neste vídeo, o especialista Marlos de Mendonça faz uma breve comparação entre Java e C/C++, abordando não apenas as diferenças das linguagens, mas dos paradigmas de programação OO e estruturado.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Ao longo deste tema, você desenvolveu uma percepção crítica sobre a linguagem, que deve extrapolar para todas as linguagens. O bom profissional avalia tecnicamente as ferramentas à sua disposição e o problema a ser resolvido.

Nós apenas tocamos a superfície do desenvolvimento em Java, mas esta é uma linguagem com muitos recursos e que não se tornou tão popular por mero acaso! Introduzimos os conceitos de classe e objetos e a forma com trabalhá-los em Java. Prosseguimos no estudo aprendendo sobre o agrupamento de objetos, as formas mais complexas de manipulação e as ferramentas disponíveis para isso. Por fim, tecemos uma importante discussão sobre a linguagem e passamos por exercícios para sedimentar o conhecimento. Agora, você está pronto para se aprofundar neste tema!

PODCAST

⚠ PODCAST

Agora, o especialista Marlos de Mendonça encerra o tema falando sobre o histórico da linguagem Java e orientação objeto, as principais aplicações e perspectivas futuras.

REFERÊNCIAS

ECLIPSE FOUNDATION. **Eclipse Che**. Consultado em meio eletrônico em: 12 fev. 2021.

GOSLING, J. *et.al*. **The Java® Language Specification:** Java SE 15 Edition. *In*: Oracle America Inc, 2020.

ORACLE AMERICA INC. **Java SE:** Chapter 2 − The Structure of the Java Virtual Machine. 2015.

ORACLE AMERICA INC. **Java® Platform, Standard Edition & Java Development Kit Specifications − Version 15.** Consultado em meio eletrônico em: 12 fev. 2021.

ORACLE AMERICA INC. **The Java™ Tutorials**. Lesson: Introduction to Collections. Consultado em meio eletrônico em: 12 fev. 2021.

SCHILDT, Herbert. **Java:** The Complete Reference. 9th. ed. Nova Iorque: McGraw Hill Education, 2014.

STROUSTRUP, Bjarne. **Stroustrup:** FAQ. 2020.

SUTTER, Herb. **The C Family of Languages**: Interview with Dennis Ritchie, Bjarne Stroustrup, James Gosling. Consultado em meio eletrônico em: 12 fev. 2021.

THE APACHE FOUNDATION. **NetBeans IDE:** Overview. Consultado em meio eletrônico em: 12 fev. 2021.

EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, pesquise na internet:

Os processos envolvidos na implementação do método “*groupingBy*”. A documentação da API Java é um bom ponto de partida!

Artigos e entrevistas sobre os criadores do C++ (Bjarne Stroustrup) e da Java (James Gosling).

As disputas comerciais envolvendo a Oracle e o Google em torno do uso da Java no Android.

CONTEUDISTA

Marlos de Mendonça Corrêa

🔗 CURRÍCULO LATTES