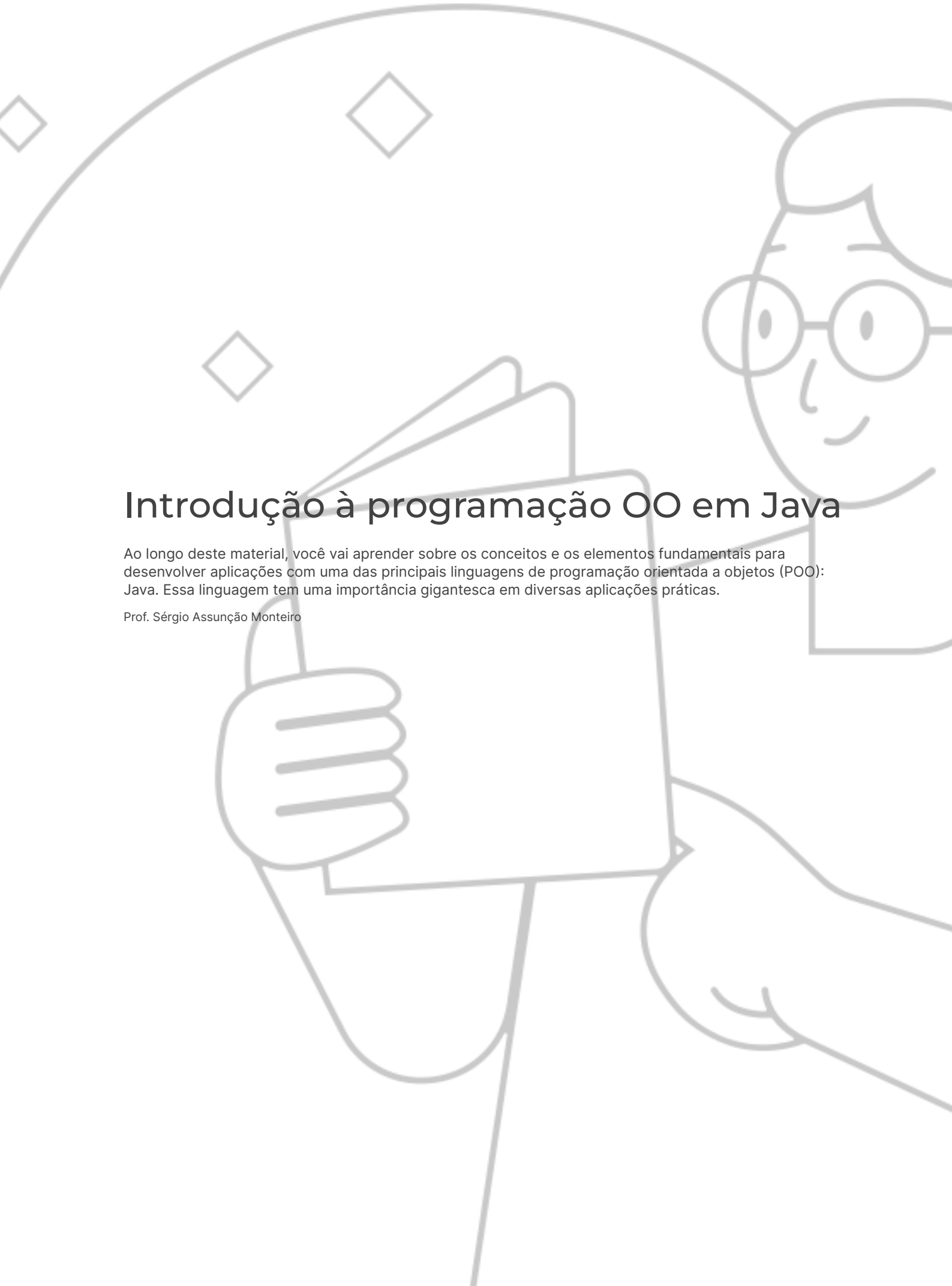


Introdução à programação OO em Java

Ao longo deste material, você vai aprender sobre os conceitos e os elementos fundamentais para desenvolver aplicações com uma das principais linguagens de programação orientada a objetos (POO): Java. Essa linguagem tem uma importância gigantesca em diversas aplicações práticas.

Prof. Sérgio Assunção Monteiro

Preparação

É importante instalar o Java JDK adequado para a versão do seu sistema operacional no site oficial da **Oracle**, na sua máquina de trabalho e utilizar uma IDE. No nosso caso, vamos utilizar o **Eclipse**.

Objetivos

- Descrever a definição, a manipulação e as nuances de classes e objetos em Java.
- Descrever o mecanismo de herança e polimorfismo em Java.
- Descrever os mecanismos de agrupamento de objetos em Java.
- 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 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.

Neste estudo, 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. O nosso objetivo é obtermos os conhecimentos necessários para nos destacarmos nesse mercado que possui uma grande demanda de profissionais tecnicamente capacitados e com habilidades para resolver problemas demandados pelo mercado.

Assista ao vídeo e entenda a linguagem de programação Java, com enfoque na programação orientada a objetos (POO). Descubra a sintaxe e como ela incorpora os princípios da POO, além de explorar os motivos que tornaram o Java tão relevante no mercado. Amplie seu entendimento sobre Java e sua aplicação no mundo da programação.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Classes e objetos em Java

Vamos começar estudando os conceitos de classes na linguagem de programação Java. Neste momento, o nosso objetivo é entender o potencial do Java e já iniciar a implementar alguns exemplos. No entanto, vamos aprofundar os conceitos mais adiante. Por isso, se você ainda não conhece essa linguagem de programação, não se preocupe, pois vamos analisá-la com muitos detalhes mais adiante. Aproveite para conhecer uma das linguagens de programação mais importantes da atualidade e que, durante muito tempo, foi considerada a referência para programação orientada a objetos.

Assista e obtenha uma visão abrangente da linguagem de programação Java e seu incrível potencial. Explore a criação de classes e sua implementação em Java, além de aprender sobre símbolos de declaração e a utilização de modificadores.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Classe e sua realização em Java

Em POO, uma classe é uma maneira de se criar objetos que possuem mesmo comportamento e mesma estrutura. Ou seja, uma classe é uma estrutura definida por:

- Dados
- Métodos que operam sobre esses dados
- Mecanismo de instanciação dos objetos

O conjunto formado por dados e métodos (assinatura e semântica) estabelece o contrato existente entre o desenvolvedor da classe e o seu usuário.

O código a seguir mostra um exemplo de uma classe em Java:

```
java

public class Aluno {
    private String nome;
    public void inserirNome(String nn) {
        nome = nn;
    }
    public String recuperarNome() {
        return nome;
    }
    public static void main(String args[]){
        Aluno a = new Aluno();
        a.inserirNome ("Pessoa");
        System.out.println("saida: "+a.recuperarNome());
    }
}
```

Em Java, cada classe pública deve estar em um arquivo com o mesmo nome da classe e extensão “java”. Logo, a classe do código superior deve ser salva em um arquivo de nome “Aluno.java”.

Dica

O site jdoodle possui um compilador java online, onde você pode copiar e executar os códigos.

A classe “Aluno”, possui um atributo do tipo “String” (nome) e dois métodos (inserirNome e recuperarNome). Além disso, podemos notar as palavras reservadas "private" e "public". Nós usamos essas instruções para modificar a acessibilidade de métodos, atributos e classes. O trecho mostrado é um exemplo bem simples de declaração de uma classe em Java. Pela especificação da linguagem (GOSLING *et. al.*, 2020), há duas formas de declaração de classes: Normal e Enum.

A partir disso, vamos nos deter apenas à forma normal.

```
java

[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. Logo, a forma mais simples possível de se declarar uma classe em Java é:

```
java

class ClasseSimples { }
```

Observamos nessa declaração a presença dos símbolos reservados (e obrigatórios) e a existência do Identificador (ClasseSimples).

Mais símbolos de declaração

Agora, vamos conhecer outros símbolos da declaração. Os modificadores podem ser qualquer elemento do seguinte conjunto:

```
java

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

Considerando que:

Annotation

É uma definição e não propriadamente um elemento. 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. Eles são os modificadores de acesso.

Abstract e final

São modificadores que relacionam-se com a hierarquia de classes.

Static

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

Strictfp

É um modificador que torna a implementação de cálculos de pontos flutuantes (números dos conjuntos dos reais) independentes da plataforma. Sem o uso desse modificador, as operações se tornam dependentes da plataforma sobre a qual a máquina virtual é executada.

Composição de modificadores

Podemos combinar alguns desses modificadores com outros. Por exemplo, podemos definir uma classe da seguinte forma:

```
java

public abstract class Teste { }
```

Outro elemento opcional são os "TipoParâmetros". Vale ressaltar que esses elementos estão relacionados à implementação de programação genérica em Java, e estão além do escopo abordado neste contexto.

O elemento opcional seguinte é a "Superclasse". Tanto a Superclasse quanto o "Superinterface" permitem 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" e que outras classes vão herdar as características dela.

Outro ponto importante nessa visão geral do Java, é a sintaxe do elemento "Superinterface". Ele utiliza a palavra-chave "implements IdentificadorInterface". Isso significa que a classe implementa a interface "IdentificadorInterface".

Veja a seguir um exemplo mais complexo que utiliza mais recursos de declaração de classe em Java:

```
java

@Deprecated @SuppressWarnings ("deprecation") public abstract strictfp class Aluno
extends Pai implements Idealizacao, Sonho {
    private String nome;
    public void inserirNome(String nn){
        nome = nn;
    }
    public String recuperarNome(){
       return nome;
    }
    ...//outros métodos ocultados por simplicidade
}
```

Esse exemplo código é apenas conceitual. A ideia é nos acostumarmos com o Java e obtermos o melhor do potencial que ele pode nos oferecer.

Atividade 1

O principal elemento estrutural de um programa em Java é a definição de uma classe. Afinal de contas, até hoje, o Java é considerado uma das linguagens de programação orientada a objetos mais importantes da atualidade e, certamente, vai continuar a ser por muitas décadas. Nesse sentido, selecione a opção correta que contém uma declaração de classe válida na linguagem Java.

A

Private class Aluno{}

B

Class Aluno{}

C

Protected class Aluno {}

D

public class Aluno {}

E

extends class Aluno {}

A alternativa D está correta.

Na declaração de uma classe, o modificador "public" é opcional, no entanto ele é correto. Além disso, devemos ficar atentos que o Java também diferencia os nomes de variáveis, comandos, métodos e classes por meio do uso de letras maiúsculas e minúsculas. Por isso, dizemos que o Java é uma linguagem de programação "case sensitive". Por exemplo, os termos "class" e "Class" são completamente diferentes para o Java.

Objetos: os produtos das classes

Aqui, vamos começar a conhecer as classes em Java com mais profundidade. Em especial, vamos dar destaque para dois aspectos muito importantes: o método construtor e a instanciação de classes. Dessa forma, vamos obter mais familiaridade com o Java e entender alguns dos motivos que o tornaram tão popular.

Neste vídeo, vamos explorar a instância de classes no Java, aprofundar nosso entendimento sobre o método construtor especial e sua relevância para o ciclo de vida dos objetos. Também discutiremos os diferentes estados de um objeto em Java.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Instanciação de classes

As classes são modelos. Para realmente realizarmos atividades em um programa, precisamos instanciá-las. Chamamos essas instâncias de objetos. Para compreendermos melhor o que é um objeto, vamos analisar seu ciclo de vida. A criação de um objeto ocorre em duas etapas:

1. Declaramos um objeto como sendo do tipo de uma classe.
2. Na sequência, instanciamos o objeto e passamos a utilizá-lo.

Aqui é importante destacarmos que o objeto tem todas as características da classe, ou seja, atributos e métodos. Uma forma de declararmos um objeto é dada por:

```
java

Aluno objetoAluno = new Aluno();
```

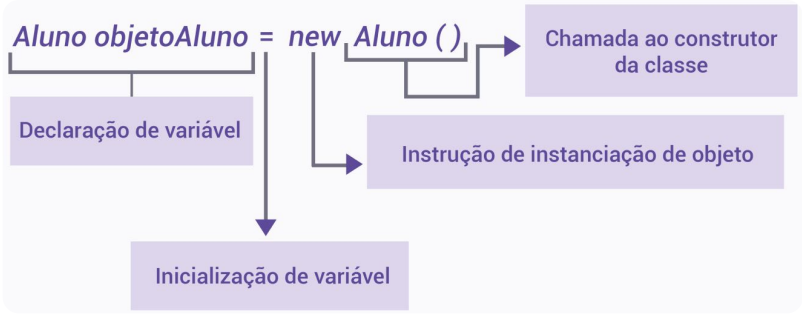
Podemos, ainda, reescrever esse código da seguinte maneira:

```
java

                Aluno objetoAluno;
                objetoAluno = new Aluno();
```

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 por meio de um método especial chamado construtor.

O método construtor é sempre executado quando fazemos a instanciação de um objeto e, obrigatoriamente, deve ter exatamente o mesmo nome 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”. Veja a seguir como ocorre esse processo de criação de objetos:



Exemplo de método construtor.

Exemplo de um código que utiliza construtor

O objetivo desse código é criarmos uma classe “Aluno” com atributos e alguns métodos. No entanto, o ponto mais importante que devemos observar é o uso do método construtor que possui o mesmo nome da classe, conforme podemos ver no código a seguir:

```
java

      import java.util.Random;
      public class Aluno{
      //Atributos
      private String nome;
      private int idade;
      private double codigo_identificador;
      private Random aleatorio;
      //Construtor
      public Aluno(String nome, int idade){
           aleatorio = new Random();
           this.nome = nome;
           this.idade = idade;
           this.codigo_identificador = aleatorio.nextDouble();
      }
      //Métodos
      public void definirNome(String nome){
           this.nome = nome;
      }
      public void definirIdade( int idade){
           this.idade = idade;
      }
    }
```

Devemos notar que o método construtor e classe possuem o mesmo nome. Além disso, passamos dois parâmetros para o construtor para estabelecer o comportamento inicial do objeto que vai instanciá-la. Outro ponto que precisamos observar é o escopo das variáveis dentro de uma classe. Por exemplo, vamos supor que tirássemos o trecho do seguinte código.

```
java

private Random aleatorio;
```

A partir disso, faríamos a modificação dos atributos da classe e alteraríamos a instanciação da variável "aleatorio" dentro do construtor para:

```java
Random aleatorio = new Random();
```

O que ocorreria nesse caso?

A variável “aleatorio” seria válida apenas no escopo local do método construtor, ou seja, não seria um atributo da classe "Aluno" e, portanto, só poderia ser usada dentro do construtor.

Estados de um objeto

O estado de um objeto é definido pelos seus atributos, enquanto seu comportamento é determinado pelos seus métodos. Por exemplo, vamos considerar a seguinte instanciação a partir da classe do código anterior:

```java
Aluno novoAluno = new Aluno("teste de instanciação", 50);
```

Após a execução do código, teremos 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 Aluno (String nome, int idade), public void definirNome (String nome) e public void definirIdade (int idade)”.

Por fim, chegamos à última etapa do ciclo de vida de um objeto: a sua destruição. Na linguagem Java, não é possível ao programador destruir manualmente um objeto. Em vez disso, o Java implementa o conceito de coletor de lixo no qual 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.

Atenção

O programador tem a possibilidade de solicitar à JVM a realização da coleta de lixo, através da invocação do método "gc()" da biblioteca System. Mas isso é apenas uma solicitação, ou seja, não é uma ordem de execução. Na prática, isso significa que a JVM tentará executar a coleta de lixo tão logo quanto possível, mas não necessariamente quando o método foi invocado.

Atividade 2

A linguagem de programação Java possui muitas particularidades sobre o ciclo de vida de um objeto. A ideia é reduzir as possibilidades de um programa causar riscos de segurança para o sistema, mas, na prática, essa forma de trabalhar impõe limites sobre o que o programador pode atuar no ciclo de vida dos objetos. Nesse sentido, sobre objetos em Java, selecione a opção correta:

A

O programador pode determinar o momento exato em que deseja que o objeto seja destruído.

B

Quando um objeto é passado como parâmetro em um método, um clone dele é gerado.

C

O programador não precisa se preocupar em desalocar a memória de um objeto destruído.

D

O método construtor não pode ser privado.

E

O coleto de lixo tem a finalidade de reciclar os objetos destruídos pelo programador.

A alternativa C está correta.

A reciclagem de espaço de memória em Java é feita pelo coletor de lixo.

Classes e o encapsulamento de código

Do ponto de vista da POO, o encapsulamento visa ocultar do mundo exterior os atributos e o funcionamento da classe.

Para realizarmos a interação do objeto com os demais módulos de um sistema, precisamos estabelecer métodos públicos da classe. No entanto, ainda temos outras formas de visibilidade dos métodos que restringem o acesso a eles. Tudo isso é o que chamamos de contrato, estabelecido entre a classe e o código que a utiliza. A ideia do encapsulamento é disponibilizar para os demais módulos do sistema apenas o que eles precisam para realizar suas tarefas. Portanto, o conceito de encapsulamento está fortemente relacionado ao de visibilidade.

A visibilidade de um método ou atributo define quem pode ou não os acessar, ou seja, ela afeta a forma como o encapsulamento funciona. Existem três tipos de visibilidade, representados pelos seguintes modificadores:

- “private”
- “protected”
- “public”

Quando tratarmos sobre as propriedades de herança e polimorfismo, vamos nos aprofundar mais sobre o uso desses modificadores. Por enquanto, deve ficar claro para nós 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.

Neste vídeo, você vai conhecer a propriedade de encapsulamento da orientação a objetos. Em especial, vamos estudar um exemplo prático no Java, para que você possa explorar melhor essa propriedade.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Passo 1

No caso do Java, o encapsulamento é um mecanismo que permite o agrupamento de dados e métodos em uma única unidade chamada classe que atende a dois propósitos principais. Vamos conferí-los!

Ocultação de dados

O estado interno de um objeto é oculto do acesso externo. Os membros de dados internos de uma classe são declarados como privados, impedindo o acesso direto de outras classes. Para obtermos acesso a esses membros, devemos usar métodos públicos (getters e setters), que controlam as operações de leitura e gravação nos dados. Isso garante que os dados sejam acessados e modificados de forma controlada, mantendo a integridade do estado do objeto.

Abstração

O encapsulamento permite o conceito de abstração ao fornecer uma interface simplificada e bem definida para interagir com um objeto. A classe expõe métodos públicos que definem seu comportamento, enquanto os detalhes da implementação interna ficam ocultos. Essa abstração facilita o uso da classe, pois os usuários não precisam conhecer as complexidades internas. O encapsulamento ajuda a gerenciar a complexidade e permite a programação modular dividindo o código em unidades menores (classes).

Passo 2

Agora, implemente o exemplo de código em Java que usa o encapsulamento:

```
java

import java.util.Random;
    //Classe
    public class Pessoa {
    //Atributos
    private String nome;
    private double codigo_identificador;
    private Random aleatorio;
    //Métodos
    public Pessoa (String nome){
         aleatorio = new Random();
         this.setNome(nome);
         this.codigo_identificador = aleatorio.nextDouble();
    }
    private void setNome (String nome) {
         this.nome = nome;
    }
    public String getNome () {
         return this.nome;
    }
    public double getCodigoIdentificador (){
         return this.codigo_identificador;
    }
    public static void main(String args[]){
        Pessoa p1 = new Pessoa("Teste A");
        System.out.println("Pessoa 1: "+p1.getNome());
    }
}
```

Passo 3

Execute o programa e observe o resultado:

Pessoa 1: Teste A

Os pontos mais importantes que devemos observar nesse código são:

O modificador “private”

Usado no método “setNome”, o qual indica que ele só pode ser usado por métodos dentro da própria classe.

O modificador “public”

Usado nos métodos “getNome” e “getCodigoIdentificador”, os quais indicam que podem ser chamados por objetos que instanciam a classe.

É interessante executar esse exemplo para que você comece a ganhar mais segurança sobre a sintaxe da linguagem Java e, além disso, perceba as vantagens de proteger os dados da classe e garantir um comportamento previsível.

Faça você mesmo!

Sem dúvidas, a melhor forma de aprender uma linguagem de programação é praticando. Nesse sentido, considere o código anterior sobre a classe “Pessoa”. Agora, modifique o programa, para que ele exiba o seguinte resultado:

[Pessoa 1]nome: Teste A, Código Identificador: valor numérico do código

[Pessoa 2]nome: Teste B, Código Identificador: valor numérico do código

Chave de resposta

Basta modificarmos a função mais com a criação de mais uma variável e a adaptação de como os dados vão ser exibidos na saída. Apresentamos uma possível solução no código a seguir:

```
java

   Pessoa p1 = new Pessoa("Teste A");
   Pessoa p2 = new Pessoa("Teste B");
   System.out.println("[Pessoa 1] nome: "+p1.getNome()+", Código Identificador:
"+p1.getCodigoIdentificador());
   System.out.println("[Pessoa 2] nome: "+p2.getNome()+", Código Identificador:
"+p2.getCodigoIdentificador());
}
```

O restante do código deve ficar inalterado.

Tipos de relações entre objetos

Na POO, temos os seguintes tipos de relações entre objetos. Vamos conferí-los!

Associação

É semanticamente a relação 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 “um para um”, “um para vários”, “vários para um” e “vários para vários”.

Agregação

Ocorre entre dois ou mais objetos, com cada um tendo seu próprio ciclo de vida, mas com um objeto (pai) contendo os demais (filhos). Precisamos compreender que, nesse caso, os objetos filhos podem sobreviver à destruição do objeto pai. Um exemplo de agregação se dá entre as classes “Escola” e “Aluno”, pois se uma escola deixar de existir, não significa que os alunos simplesmente deixarão de existir.

Composição

Difere sutilmente da agregação, pois ocorre quando há uma relação de dependência entre o(s) filho(s) e o objeto pai. Caso o pai deixe de existir, necessariamente o filho será destruído. Por exemplo, no caso da relação entre uma classe “Escola” e as classes “Departamentos”, certamente, que a extinção da escola implica a extinção dos departamentos.

A partir disso, veremos como os conceitos de associação, agregação e composição formam conjuntos que se relacionam. Confira!

Conjunto formado pela definição das relações.

Conforme podemos concluir, a composição é um caso especial de agregação e o conceito mais restritivo de todos, enquanto a associação é o mais abrangente.

Neste vídeo, você terá a oportunidade de identificar as diferentes relações que podem ser implementadas no Java, além de assistir a um exemplo prático que demonstrará como aplicar essas relações.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Passo 1

Veja a seguir um exemplo de código em Java que nos auxilia a compreender de forma mais clara as relações de agregação e composição entre objetos.

```
java

class Escola {
    //Atributos
    private String nome, CNPJ;
    private Endereco endereco;
    private Departamento departamentos [];
    private Aluno discentes [];
    private int nr_discentes , nr_departamentos;

    //Métodos
    public Escola ( String nome , String CNPJ) {
        this.nome = nome;
        this.CNPJ = CNPJ;
        this.departamentos = new Departamento[10];
        this.discentes = new Aluno[1000];
        this.nr_departamentos = 0;
        this.nr_discentes = 0;
    }
    public void criarDepartamento(String nomeDepartamento){
        if(nr_departamentos <= 10)
        {
            departamentos[nr_departamentos] = new Departamento
( nomeDepartamento);
            nr_departamentos++;
        } else {
            System.out.println ( "Nao e possivel criar outro Departamento." );
       }
    public void matricularAluno ( Aluno novoAluno ) {
       discentes [ nr_discentes ] = novoAluno;
    }
}
```

Passo 2

Agora, vamos destacar alguns pontos importantes e que devem ser observados:

Associação

Devemos notar uma associação entre a classe Escola” e as classes “Endereco”, “Departamento” e “Aluno”.

Agregação

Devemos notar uma relação entre a classe “Escola” e “Aluno”. Nesse caso, trata-se de uma **agregação**, pois os alunos ainda vão existir no caso de a escola ser extinta.

Composição

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**.

Faça você mesmo!

Acabamos de estudar sobre como funcionam as relações entre classes em Java. Então, com base no código da classe “Escola”, desenvolva a implementação da classe “Endereco” com os atributos “nome_rua” e “numero”.

Chave de resposta

Podemos implementar a classe Endereco da seguinte maneira:

```
java
    public class Endereco {
    private String nome_rua;
    private int numero;
    public Endereco (String nome_rua, int numero){
         this.setNomeRua(nome_rua);
         this.numero=numero;
    }
    private void setNomeRua(String nome_rua) {
         this.nome_rua = nome_rua;
    }
    public String getNomeRua () {
         return this.nome_rua;
    }
    private void setNumero(int numero) {
         this.numero = numero;
    }
    public int getNumero() {
         return this.numero;
    }
}
```

Para testarmos se a classe está correta, podemos acrescentar a função “main” dentro da classe Endereco da seguinte forma:

```
java

    public static void main(String args[]){
        Endereco ender = new Endereco ("rua X", 7);
        System.out.println(ender.getNomeRua()+", "+ender.getNumero());
    }
```

Cujo resultado da execução será:

rua X, 7

Uso de referência de objetos em Java

Em Java, não é possível criar variáveis do tipo ponteiro. A linguagem Java oculta esse mecanismo, de modo que toda variável de classe é uma referência para o objeto instanciado. Isso tem implicações importantes na forma de lidar com objetos. Analisando um exemplo, vamos entender como isso funciona na prática.

Neste vídeo, você aprenderá a utilizar referências de objetos em Java e entenderá as implicações ao utilizar esse recurso. Faremos uma análise detalhada de um exemplo e apresentaremos um exemplo completo de como referenciar objetos.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

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.

Passo 1

Implemente os seguintes códigos:

```java
class Aluno {

    //Atributos
    private String nome;
    private int idade;
    //Métodos
    public Aluno ( String nome , int idade ) {
         this.nome = nome;
         this.idade = idade;
    }
    public void definirNome ( String nome ) {
         this.nome = nome;
    }
    public void definirIdade ( int idade ) {
         this.idade = idade;
    }
    public String recuperarNome () {
         return this.nome;
    }
}
```

Passo 2

Implemente a classe “Referencia”, cujo código é apresentado a seguir:

```java
public class Referencia  {
       private Aluno a1 , a2;
       public Referencia  ( ) {
        a1 = new Aluno ( "Carlos" , 20);
        a2 = new Aluno ( "Ana" , 23 );
        System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
        System.out.println("O nome do aluno a2 é " + a2.recuperarNome());
        a2 = a1;
        a2.definirNome("Flávia");
        System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
        manipulaAluno ( a1 );
        System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
       }
       public void manipulaAluno ( Aluno aluno ) {
        aluno.definirNome("Márcia");
       }
       public static void main(String args[]) {
           Referencia r = new Referencia ();
           System.out.println("Fim da Execução ");
    }

}
```

Passo 3

Execute o programa e veja se o resultado da execução desse código é:

```
java

nome do aluno a1 é Carlos
nome do aluno a2 é Ana
nome do aluno a1 é Flávia
nome do aluno a1 é Márcia
```

Vamos entender o que acontece seguindo passo a passo a execução do código visto anteriormente.

A classe principal desse código é a classe “Referencia”. Logo, já sabemos que o nome do arquivo deve ser “Referencia.java”. Dentro dela, há o método estático “main”, no qual criamos um objeto do tipo “Referencia”. Nesse momento, a JVM passa a instanciar dois objetos do tipo Aluno. Sendo que o objeto “a1” fica com o estado dos atributos “nome” e “idade”, respectivamente, dados por "Carlos" e "20".

Já o objeto “a2” fica com “nome” recebendo “Ana” e “idade” recebendo “23”. Ou seja, temos dois objetos distintos (“a1” e “a2”), cujos estados também são distintos.

Mais à frente, executamos a linha:

```
java

        a2 = a1;
```

Que significa que “a1” e “a2” são referências para os objetos criados, ou seja, não é um caso de atribuição. Por isso que, ao modificarmos o estado do objeto “a2”, também afetamos o estado do objeto “a1”.

Outro ponto interessante ocorre quando chamamos o método “manipulaAluno”, pois, 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” que tem impacto também no objeto “a2”.

Apesar do Java oferecer esse recurso de referência de objetos, devemos evitá-lo, pois o código pode ficar confuso e difícil de dar manutenção posteriormente. Bem, agora, chegou a hora de praticar!

Faça você mesmo!

Como sabemos, trabalhar com referências a objetos em Java pode ser um pouco confuso. No entanto, não podemos deixar de conhecer esse tipo de recurso, pois muitos sistemas fazem uso dele e esse conhecimento pode ser bastante útil para fazer a manutenção em um sistema. Nesse sentido, modifique o código da classe “Referência” de modo que você simplesmente modifique a linha:

a2 = a1;

Por

a1 = a2;

Execute o programa modificado e responda se houve ou não alguma diferença no resultado. Justifique a sua resposta.

Chave de resposta

O programa completo deve ficar da seguinte maneira:

```java
class Aluno {
    private String nome;
    private int idade;
    public Aluno ( String nome , int idade ) {
         this.nome = nome;
         this.idade = idade;
    }
    public void definirNome ( String nome ) {
         this.nome = nome;
    }
    public void definirIdade ( int idade ) {
         this.idade = idade;
    }
    public String recuperarNome () {
         return this.nome;
    }
}
public class Referencia  {
       private Aluno a1 , a2;
       public Referencia() {
        a1 = new Aluno ( "Carlos" , 20);
        a2 = new Aluno ( "Ana" , 23 );
        System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
        System.out.println("O nome do aluno a2 é " + a2.recuperarNome());
        a1 = a2;
        a2.definirNome("Flávia");
        System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
        manipulaAluno ( a1 );
        System.out.println("O nome do aluno a1 é " + a1.recuperarNome());
       }
       public void manipulaAluno ( Aluno aluno ) {
        aluno.definirNome("Márcia");
       }
       public static void main(String args[]) {
           Referencia r = new Referencia ();
           System.out.println("Fim da Execução ");
    }
}
```

Onde temos o resultado:

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

Fim da Execução

Que é exatamente o código anterior. Isso ocorreu porque a mudança da ordem dos objetos não implica a mudança do papel que eles desempenham, ou seja, ambos continuam referenciando um ao outro.

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 subsequentes.

Neste vídeo, você terá a oportunidade de explorar os conceitos relacionados à herança e aprender como implementá-los de forma eficiente em Java. Além disso, abordaremos os elementos fundamentais da herança, a herança de interfaces e como lidar com múltiplas superclasses.

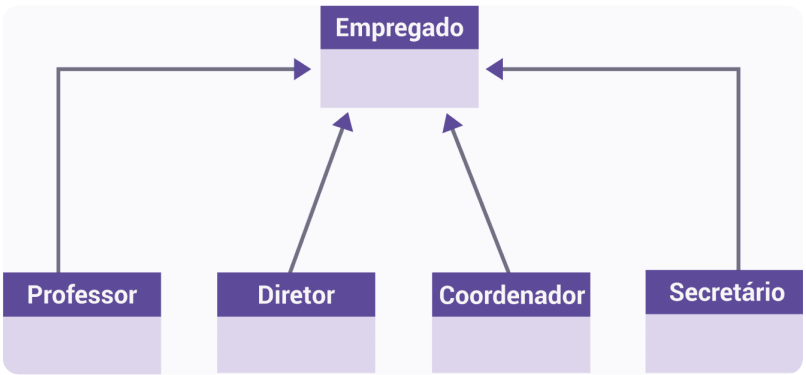
> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Elementos básicos da herança

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 em uma 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.

Observe a seguir um diagrama UML que modela uma hierarquia de herança.



Na figura acima, vemos que "Empregado" é pai (superclasse) das classes "Professor", "Diretor", "Coordenador" e "Secretario". Essas últimas são filhas (subclasses) da classe "Empregado".

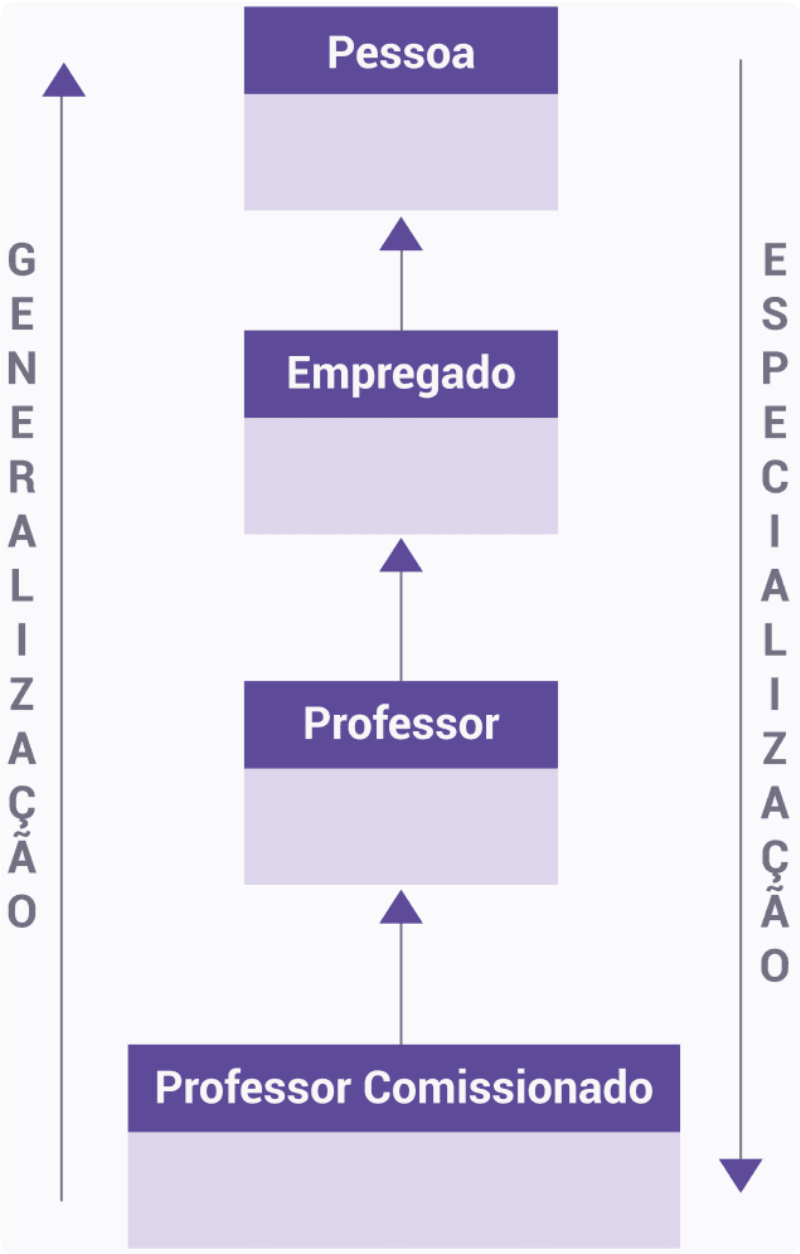
Como tratar mais de uma superclasse

Essa situação é denominada herança múltipla e, apesar de a POO aceitar a herança múltipla, a linguagem Java não oferece suporte para esse tipo de caso. No entanto, apesar de não permitir herança múltipla de classes, a linguagem permite que uma classe herde de múltiplas interfaces.

Atenção

É importante ressaltar que uma interface pode herdar de múltiplas interfaces pai. Ao contrário das classes, as interfaces não permitem a implementação de métodos, sendo responsabilidade da classe que as implementa realizar essa implementação.

Agora, vamos analisar a seguinte imagem:



Herança com vários níveis de ancestralidade.

Ao analisar a genealogia das classes, observamos que à medida que descemos na hierarquia, nos deparamos com classes cada vez mais específicas. Por outro lado, ao subir na hierarquia, nos deparamos com classes cada vez mais gerais. Essas características refletem os conceitos fundamentais de generalização e especialização da OO.

Agora, vamos visualizar um exemplo de código que demonstra a implementação da herança para a classe "ProfessorComissionado" com baseado na imagem anterior:

```
java

public class ProfessorComissionado extends Professor {
//…
}
```

Em outras palavras, ao utilizar a herança, é importante notar que ela é declarada apenas para a classe ancestral imediata. Com isto, podemos afirmar que:

1. A classe “**Professor**” deve declarar “**Empregado**” como sua superclasse.
2. O “**Empregado**” deve declarar “**Pessoa**” como sua superclasse.

Herança de interfaces

A sintaxe é análoga para o caso das interfaces, exceto que pode haver mais de um identificador de superinterface. O código consecutivo mostra um exemplo baseado no diagrama anterior, considerando que “ProfessorComissionado”, “Professor” e “Diretor” sejam interfaces.

```
java

public interface ProfessorComissionado extends Professor, Diretor {
//…
}
```

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

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”.

Resumindo

Na herança com vários níveis de ancestralidade, a classe "Pessoa" é uma subclasse de "Object" e, portanto, herda todos os métodos de "Object". Esses métodos são herdados pelas subclasses subsequentes até a base da hierarquia de classes. Consequentemente, um objeto da classe "ProfessorComissionado" terá acesso a todos os métodos disponíveis em "Object".

Agora, vamos trabalhar em um exercício conceitual.

Atividade 1

A herança é uma propriedade típica da programação orientada a objetos e é disponível pelo Java. Obviamente, sempre precisamos usar essa propriedade com cuidado, pois, caso contrário, podemos tornar o código bem complicado de entender. Neste sentido, sobre herança em Java, é correto afirmar apenas que:

A

Um atributo protegido da superclasse não é visível para a subclasse.

B

Um objeto instanciado da subclasse é também um objeto do tipo da superclasse.

C

A superclasse herda os métodos e atributos públicos da subclasse.

D

Uma superclasse só pode ter uma subclasse.

E

Um objeto instanciado da superclasse é também um objeto do tipo da subclasse.

A alternativa B está correta.

O mecanismo de herança oferece para subclasse a mesma estrutura da superclasse. Portanto, o objeto da classe filha tem as mesmas propriedades da classe mãe.

Herança e visibilidade

Quando dizemos que a classe “Pessoa” é uma generalização da classe “Empregado”, isso significa que ela reúne atributos e comportamentos 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. Aqui, vamos abordar esses pontos que já são um avanço do que podemos fazer com a propriedade herança.

Neste vídeo, você vai aprender sobre os modificadores em Java, aplicáveis a atributos, métodos e classes, auxiliando na compreensão da combinação das propriedades da programação orientada a objetos e na construção de projetos mais organizados e controlados.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Mecanismos de visibilidade

Inicialmente, precisamos compreender como os modificadores de acesso operam. Já vimos que esses modificadores alteram a acessibilidade de classes, métodos e atributos. Existem quatro níveis de acesso em Java. Vamos conhecê-los!

Default
É assumido quando nenhum modificador é usado. Define a visibilidade como deve ser restrita ao pacote do Java.

Privado
É declarado pelo uso do modificador “private”. A visibilidade dos atributos e métodos fica restrita à classe.

Protegido
É declarado pelo uso do modificador “protected”. A visibilidade é restrita ao pacote e a todas as subclasses.

Público
É 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 (classes, atributos ou métodos) fiquem visíveis aos demais elementos que compõem um código. Ainda temos o escopo definido por um pacote no Java, cuja ideia é que ele defina um espaço de nomes e seja usado para agrupar classes relacionadas.

O conceito de pacote contribui para a melhoria da organização do código de duas maneiras:

- Permite organizar as classes pelas suas afinidades conceituais.
- Previne conflito de nomes.

Devemos observar que evitar conflitos de nomes é um trabalho desafiador em um software que envolva diversos desenvolvedores e centenas de entidades e funções.

Pacotes em Java

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 também 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.

Já, o modificador de acesso “**public**” é o menos restritivo. Ele fornece acesso com escopo global. Qualquer classe do ambiente de desenvolvimento pode acessar as entidades declaradas como públicas.

A seguir, observe a tabela que 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

Níveis de acesso e escopo.
Marlos de Mendonça.

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.

Portanto, devemos observar também que **métodos** e **atributos privados** não são accessíveis às subclasses, e sua **acessibilidade** não é afetada pela herança.

Atividade 2

O Java oferece o uso de palavras-chave para que possamos aplicar os modificadores. A escolha do uso desses modificadores, no entanto, é de total responsabilidade do desenvolvedor. Nesse sentido, assinale a alternativa correta sobre qual o impacto positivo de uma boa escolha do uso dos modificadores:

A

Auxiliam a estabelecer um comportamento previsível para um objeto.

B

Tornam o código mais legível.

C

Facilitam a manutenção de código.

D

Aumentam a necessidade de um desenvolvedor justificar a necessidade de usar determinado método.

E

Restringem o uso das classes, de modo que elas tenham apenas um único objetivo.

A alternativa A está correta.

Os modificadores de acessibilidade são importantes para estabelecer qual o comportamento que esperamos que um objeto tenha durante o processo de execução. Eles estão fortemente relacionados a diversas outras propriedades da programação orientada a objetos, como um mecanismo de evitarmos comportamentos anômalos e garantir a segurança dos atributos e métodos das classes.

Exemplo prático de herança

Nós já estudamos os aspectos conceituais mais importantes sobre herança, encapsulamento e uso de modificadores de acesso. Agora, vamos aplicar esses conceitos com exemplos implementados no Java para ganharmos mais familiaridade com a linguagem e suas propriedades.

Neste vídeo, vamos explorar a implementação prática de códigos em Java, enfatizando as propriedades de herança, encapsulamento e modificadores de acesso. A partir disso, também utilizaremos os modificadores de acesso.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

O nosso objetivo aqui é implementar o modelo representado a seguir:

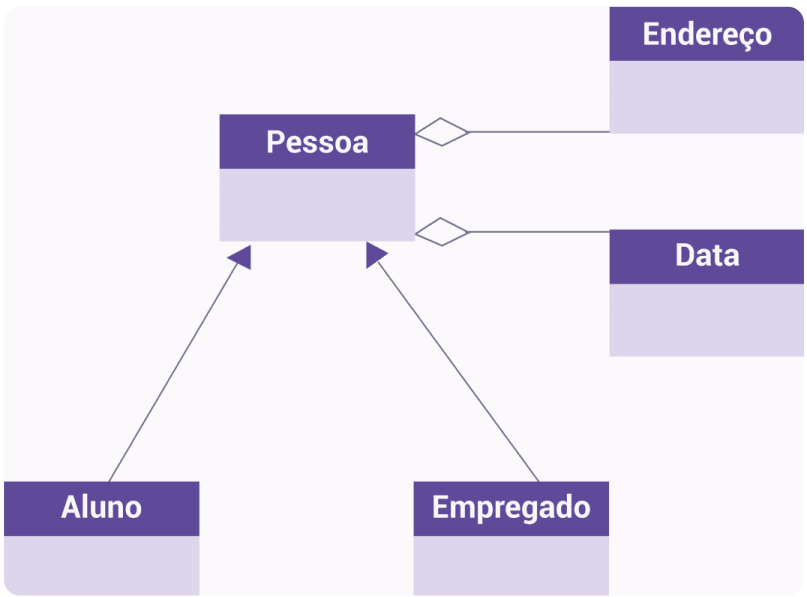


Diagrama de classes parcial de um sistema.

Basicamente, o que temos são relações entre classes que desempenham papéis bem distintos. Por exemplo, a classe “Pessoa” generaliza as classes “Empregado” e “Aluno”. Enquanto as classes “Endereço” e “Data” são exemplos de relações de agregação.

Passo 1: Implementação da classe Pessoa

A classe "Pessoa" é a “superclasse” do sistema que modelamos. Na sequência, você encontrará o código em Java que representa essa classe:

```
java
```

```
public class Pessoa {
    //Atributos
    private String nome;
    private int idade;
    private Calendar data_nascimento;
    private long CPF;
    private Endereco endereco;
    //Métodos
    public Pessoa(String nome, Calendar data_nascimento, long CPF, Endereco endereco){
        this.nome = nome;
        this.data_nascimento = data_nascimento;
        this.CPF = CPF;
        this.endereco = endereco;
        atualizarIdade();
    }
    protected void atualizarNome(String nome){
        this.nome = nome;
    }
    protected String recuperarNome(){
        return this.nome;
    }
    protected void atualizarIdade(){
        this.idade = calcularIdade();
    }
    protected int recuperarIdade() {
        return this.idade;
    }
    protected void atualizarCPF(long CPF){
        this.CPF = CPF;
    }
    protected long recuperarCPF(){
        return this.CPF;
    }
    protected void atualizarEndereco(Endereco endereco){
        this.endereco = endereco;
    }
    protected Endereco recuperarEndereco(){
        return this.endereco;
    }
    private int calcularIdade(){
        int lapso;
        Calendar hoje = Calendar.getInstance();
        lapso = hoje.get(YEAR) - data_nascimento.get(YEAR);
        if ((data_nascimento.get(MONTH) > hoje.get(MONTH)) ||
( data_nascimento.get(MONTH) ==
        hoje.get(MONTH) && data_nascimento.get(DATE) > hoje.get(DATE)))
            lapso--;
        return lapso;
    }
}
```

Também podemos observar que o código da **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”**.

Passo 2: Implementação da classe Aluno

A classe “Aluno” herda as características da classe “Pessoa”. Confira o respectivo código em Java:

```
java

public class Aluno extends Pessoa {
    //Atributos
    private String matricula;
    //Métodos
    public Aluno(String nome, Calendar data_nascimento, long CPF, Endereco endereco){
        super (nome, data_nascimento, CPF, endereco);
    }
}
```

Devemos observar o uso da palavra-chave “super” no construtor da classe “Pessoa” que significa que a classe Pessoa faz uso do construtor da classe Aluno.

Passo 3: Implementação da classe Empregado

A classe “Empregado” também herda as características da classe “Pessoa”. Veja o código em Java:

```
java

public class Empregado extends Pessoa {
    //Atributos
    protected String matricula;
    private Calendar data_admissao , data_demissao;
    //Métodos
    public Empregado(String nome, Calendar data_nascimento, long CPF, Endereco endereco) {
        super(nome, data_nascimento, CPF, endereco);
        this.matricula = gerarMatricula ();
        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;
    }
}
```

Semelhante à classe “Aluno”, devemos observar que o construtor da classe “Empregado” utiliza o “super” e que, além disso, ela estabelece os valores dos atributos “matrícula” e “data_admissao”.

Passo 4: Implementação da classe Principal

Agora, apresentamos o código da classe “Principal” que é responsável por gerenciar o nosso sistema:

```java
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");
	    aluno = new Aluno ("Marco Antônio", data ,901564098 , endereco);
	    aluno.atualizarIdade();
	    idade = aluno.recuperarIdade();
	}
}
```

Como observações finais, é importante estarmos atentos aos seguintes itens:

- 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.
- Isso significa que 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 ()”da superclasse. Porém, ela o faz por meio da invocação do método protegido “atualizarIdade()”, como vimos na classe “Aluno”.

Passo 5

Execute a atividade prática.

Faça você mesmo!

As principais vantagens da propriedade de herança são a reutilização e a padronização de código. Isso é muito importante, principalmente, quando trabalhamos com desenvolvimento de projetos em equipe. Nesse sentido, modifique o código da classe “Principal” para acrescentar um novo aluno com o nome de “Maria”, data de nascimento “07/julho/2007” e endereço “Brasil”, “Ceará”, “rua Canuto de Aguiar”, número “176”, cep “20252-060”, complemento “apto 307” e CPF “123456877-00”.

Chave de resposta

Para obtermos o resultado solicitado na questão, precisamos fazer a seguintes modificações na implementação classe “Principal”:

```
java
public class Principal {
       //Atributos
       private static Aluno aluno;
       private static Endereco endereco;
       private static Aluno aluno2;
       private static Endereco endereco2;
       //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");
           aluno = new Aluno ("Maria", data, 901564098, endereco);
           aluno.atualizarIdade();
           idade = aluno.recuperarIdade();
           //Inclusão do novo aluno
           Calendar data2 = Calendar.getInstance();
           data2.set(2007, 07, 07);
           endereco2 = new Endereco ();
           endereco2.definirPais("Brasil");
           endereco2.definirUF("CE");
           endereco2.definirCidade("Fortaleza");
           endereco2.definirRua("rua Canuto de Aguiar");
           endereco2.definirNumero("176");
           endereco2.definirCEP(20252060);
           endereco2.definirComplemento("Ap 307");
           aluno2 = new Aluno ("Maria", data , 123456877, endereco);
           aluno2.atualizarIdade();
       }
}
```

Basicamente, incluímos dois atributos na classe Principal – no caso, aluno2 e endereco2 – e repetimos as chamadas para os parâmetros dos métodos dos atributos com os respectivos dados informados na questão. De fato, poderíamos ter feito essa implementação de forma mais eficiente, mas, por enquanto, obtivemos o resultado que pretendíamos e isso está nos ajudando a entender melhor como programar com o Java.

Polimorfismo

O polimorfismo é a propriedade de um mesmo método se comportar de formas distintas para assinaturas de métodos diferentes. Ele pode se expressar de diversas maneiras. A sobrecarga de função, assim como a herança, é uma forma 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. Por exemplo, vamos considerar um objeto do tipo “Aluno”.

Comentário

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. 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 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.

Neste vídeo, conheceremos os aspectos essenciais do polimorfismo na programação orientada a objetos. A partir disso, veremos um exemplo prático que demonstra como o polimorfismo pode ser aplicado, incluindo a sobrecarga de métodos.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Passo 1

Implemente a classe “Diretor” que é subclasse de “Empregado”, sendo código dado por:

```
java

 public class Diretor extends Empregado {
   //Métodos
   public Diretor(String nome, Calendar data_nascimento, long CPF, Endereco endereco) {
        super(nome, data_nascimento, CPF, endereco);
   }
   protected void gerarMatricula(){
        matricula = "E-" + UUID.randomUUID( ).toString( );
   }
}
```

Passo 2

Implemente a modificação na classe “Principal”, cujo código é dado por:

```
java

public class Principal {
    //Atributos
    private static Empregado empregado, diretor;
    //Método main
    public static void main(String args[]) {
        Calendar data = Calendar.getInstance();
        data.set(1980, 10, 23);
        empregado = new Empregado("Clara Silva", data , 211456937 , null);
        empregado.gerarMatricula();
        diretor = new Diretor ("Marco Antônio", data , 901564098 , null);
        diretor.gerarMatricula();
        System.out.println ("A matrícula do Diretor é: " + diretor.recuperarMatricula());
        System.out.println ("A matrícula do Empregado é: " +
empregado.recuperarMatricula());
    }
}
```

Passo 3

Execute este código que 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. Essa variável, porém, está se referindo a um objeto da subclasse e o método em questão possui uma versão especializada na classe “Diretor” (ela sobrescreve o método “gerarMatricula ()” da superclasse). Dessa maneira, durante a execução, o método da subclasse será chamado. Outra forma de polimorfismo pode ser obtida por meio da sobrecarga de métodos.

Passo 4

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.

Implemente o código modificado da classe “Diretor” que utiliza sobrecarga de métodos:

```
java

public class Diretor extends Empregado {
   //Métodos
    public Diretor(String nome, Calendar data_nascimento, long CPF, Endereco endereco){
            super(nome, data_nascimento, CPF, endereco);
   }
   protected void gerarMatricula(){
            matricula = "E-" + UUID.randomUUID().toString();
   }
    protected void alterarMatricula(){
            gerarMatricula();
    }
    protected void alterarMatricula(String matricula){
            this.matricula = matricula;
    }
}
```

Nesse caso, a classe está preparada para tratar a chamada do método “alterarMatricula” de duas formas. Veja!

Uma chamada do tipo “alterarMatricula ()” invocará o método a seguir:

```
java

    protected void alterarMatricula(){
            gerarMatricula ();
    }
```

Por outro lado, caso seja feita uma chamada como “alterarMatricula (“M-202100-1000)”, o método chamado será:

```
java

protected void alterarMatricula(String matricula){
this.matricula = matricula;
}
```

A **diferença** entre qual dos dois métodos será chamado está na **passagem** ou não do **parâmetro**.

Faça você mesmo!

Agora, conhecemos mais uma importante propriedade programação orientada a objetos: o polimorfismo. De fato, há situações em que essa propriedade pode ser bastante útil, enquanto, em outros casos, pode ser bastante complicada e difícil de dar manutenção. Por isso, é muito importante realizarmos um processo de análise para decidirmos se devemos ou não aplicarmos o polimorfismo. Nesse sentido, implemente uma classe chamada de Principal que possua dois métodos que utilize **obrigatoriamente** a propriedade de polimorfismo para obter o maior elemento de dois números e o maior elemento de três números.

Chave de resposta

Agora, apresentamos uma solução para este problema.

```
java

public class Principal {
    public int maiorElem(int a, int b){
        int maior =a;
        if(b>maior){
            maior=b;
        }
        return maior;
    }
    public int maiorElem(int a, int b, int c){
        int maior = maiorElem(a, b);
        return maiorElem(maior, c);
    }
    public static void main(String args[]) {
      Principal objMaior = new Principal();
      int x=10;
      int y=25;
      int z=15;
      System.out.println("O maior elemento entre "+ x +" e " + y + " é: "+
objMaior.maiorElem(x,y));
      System.out.println("O maior elemento entre "+ x +", " + y + " e "+ z +" é: "+
objMaior.maiorElem(x,y,z));
    }
}
```

Basicamente, criamos dois métodos “maiorElem” na classe “Principal”, sendo que um deles utiliza o outro como parte integrante da própria solução. Trata-se de um exemplo simples, mas que nos dá uma noção clara de como podemos utilizar métodos mais simples para compor soluções mais complexas.

Classes abstratas

Uma classe abstrata é uma classe que não pode ser instanciada diretamente, mas serve como base para outras subclasses. Isso significa que ela pode ter atributos e métodos que podem ser herdados e estendidos por outras classes.

Além da questão de não poderem ser instanciadas, as classes abstratas podem conter métodos abstratos que são declarações de método sem uma implementação. Nesse caso, utilizamos as subclasses derivadas de uma classe abstrata para implementar esses métodos abstratos. Portanto, as classes abstratas são úteis quando precisamos definir uma interface comum para um grupo de classes relacionadas, mas deixamos os detalhes de implementação específicos para as subclasses.

Aqui, podemos perceber que a implementação dos métodos abstratos aplica o conceito de polimorfismo. Quando um método é implementado na classe abstrata, ele é chamado de método concreto. De forma semelhante, quando uma subclasse herda as características de uma classe abstrata e implementa os métodos dela, é chamada de classe concreta.

Neste vídeo, aprenderemos sobre classes abstratas e sua implementação em Java. A partir disso, veremos como esses recursos são valiosos para a definição de padrões de projetos em grande escala.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Passo 1

Implemente a classe abstrata “Animal” que possui um método abstrato chamado de “emitirSom” e um método concreto chamado de “dormir”. Na sequência, utilizamos duas subclasses herdeiras – as classes “Cachorro” e “Gato”- que vão implementar o método “emitirSom”. Acompanhe o código completo a seguir:

```
java

// Classe Abstrata
abstract class Animal {
    // metodo abstrato
    public abstract void emitirSom();

    // metodo concreto
    public void dormir() {
        System.out.println("Zzzz...");
    }
}

// subclasse concreta
class Cachorro extends Animal {
    public void emitirSom() {
        System.out.println("Latir!");
    }
}

// subclasse concreta
class Gato extends Animal {
    public void emitirSom() {
        System.out.println("Miar!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal cachorro = new Cachorro();
        Animal gato = new Gato();

        cachorro.emitirSom();
        cachorro.dormir();

        gato.emitirSom();
        gato.dormir();
    }
}
```

Passo 2

Execute o código que produz a seguinte saída:

Latir!

Zzzz...

Miar!

Zzzz...

Neste exemplo, é fundamental ressaltar que estamos aplicando vários conceitos que foram estudados até o momento, onde estão inclusos:

- Encapsulamento
- Herança
- Polimorfismo
- Classe abstrata

Agora é hora de colocar em prática todo o conhecimento adquirido!

Faça você mesmo!

Como vimos no exemplo da classe abstrata, todos os objetos das classes herdeiras eram do tipo “Animal”, pois eles implementavam exatamente os métodos que estavam definidos na classe mãe, ou seja, na classe “Animal”. Agora, o nosso desafio é implementar a classe “Leao” que também vai herdar da classe “Animal”, mas, além disso, ela deve ter um método “tipoDeAnimal” que imprima a mensagem: “É um animal selvagem”.

Chave de resposta

Basicamente, precisamos acrescentar a classe “Leao” derivada da classe “Animal”. No entanto, precisamos acrescentar o método “tipoDeAnimal”. Nesse caso, teremos que modificar a forma de declarar a classe “Leao”. A seguir, apresentamos o código completo:

java

```
// Classe Abstrata
abstract class Animal {
    // metodo abstrato
    public abstract void emitirSom();

    // metodo concreto
    public void dormir() {
        System.out.println("Zzzz...");
    }
}
// subclasse concreta
class Cachorro extends Animal {
    public void emitirSom() {
        System.out.println("Latir!");
    }
}
// subclasse concreta
class Gato extends Animal {
    public void emitirSom() {
        System.out.println("Miar!");
    }
}
// subclasse concreta
class Leao extends Animal {
    public void tipoDeAnimal() {
        System.out.println("É um animal Selvagem.");
    }
    public void emitirSom() {
        System.out.println("Rugir!");
    }
}
// Classe Principal
public class Main {
    public static void main(String[] args) {
        Animal cachorro = new Cachorro();
        Animal gato = new Gato();
        Leao leao = new Leao();
        // cachorro
        cachorro.emitirSom();
        cachorro.dormir();
        // gato
        gato.emitirSom();
        gato.dormir();
        // leao
        leao.emitirSom();
        leao.tipoDeAnimal();
        leao.dormir();
    }
}
```

No qual a saída é:

Latir!

Zzzz...

Miar!

Zzzz...

Rugir!

É um animal Selvagem.

Zzzz...

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. Esse é um dos motivos para o agrupamento nos interessar: a interação com conjuntos de dados.

Neste vídeo, aprenderemos os fundamentos do agrupamento de objetos em Java, abordando conceitos e práticas. Isso será útil para manipular grandes conjuntos de dados no dia a dia do desenvolvedor Java.

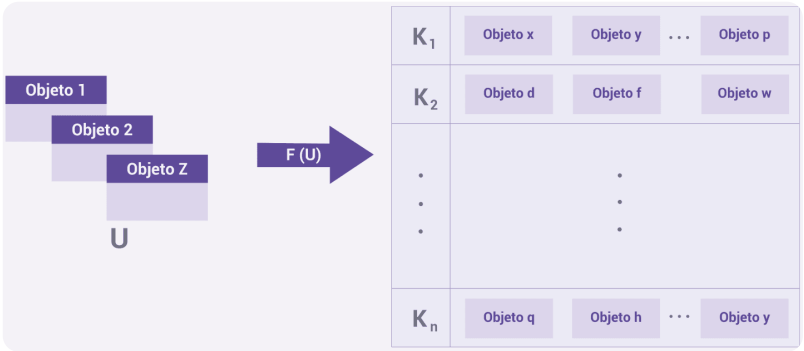
> **Conteúdo interativo**
>
> Acesse a versão digital para assistir ao vídeo.

Fundamentos do 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) = { [k1 ,], [k2 ,] , ..., [kn ,] }

A partir disso, vemos uma representação do que pretendemos fazer:



Representação gráfica do agrupamento de objetos.

Felizmente, a Java API oferece estruturas que facilitam o nosso 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.

Implementação de agrupamento de objetos

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. A estrutura “Map”, além de possuir métodos que nos auxiliarão, não permite a existência de chaves duplicadas. Veja, então, um exemplo da classe “Aluno” modificada:

```java
public class Aluno {
    //Atributos
    private String matricula,nome,naturalidade;
    //Métodos
    public Aluno(String nome,String naturalidade){
            this.nome=nome;
            this.naturalidade=naturalidade;
    }
        @Override
        public String toString(){
            return String.format("%s(%s)",nome,naturalidade);
        }
    }
}
```

Para executar o agrupamento, vamos implementar a classe “Escola”, conforme este código:

```java
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();
        this.discentes=new ArrayList();
    }
    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> agrupamento=new HashMap<>();
        for (Aluno a: discentes){
            if(!agrupamento.containsKey(a.recuperarNaturalidade())) {
                 agrupamento.put(a.recuperarNaturalidade(),new ArrayList<>());
            }
        agrupamento.get(a.recuperarNaturalidade()).add(a);
        }
    System.out.println ("Resultado do agrupamento por naturalidade: "+agrupamento);
    }
}
```

Por fim, implementamos a classe “Principal” que é responsável pelo gerenciamento do nosso sistema:

```java
public class Principal {
    // Atributos
    private static Aluno aluno1,aluno2,aluno3,aluno4,aluno5,aluno6,aluno7,aluno8,aluno9;
    private static Escola escola;
    // Método main
    public static void main(String args[]) {
        escola = new Escola("Escola Pedro Álvares Cabral", "42.336.174/0006-13");
        criarAlunos();
        matricularAlunos();
        escola.agruparAlunos();
    }
    //Métodos
    private static void criarAlunos( ){
        aluno1 = new Aluno("Marco Antônio","Rio de Janeiro");
        aluno2 = new Aluno("Clara Silva","Rio de Janeiro");
        aluno3 = new Aluno("Marcos Cintra","Sorocaba");
        aluno4 = new Aluno("Ana Beatriz","Barra do Pirai");
        aluno5 = new Aluno("Marcio Gomes","São Paulo");
        aluno6 = new Aluno("João Carlos","Sorocaba");
        aluno7 = new Aluno("César Augusto","São Paulo");
        aluno8 = new Aluno("Alejandra Gomez","Madri");
        aluno9 = new Aluno("Castelo Branco","São Paulo");
    }
    private static void matricularAlunos( ){
        escola.matricularAluno(aluno1);
        escola.matricularAluno(aluno2);
        escola.matricularAluno(aluno3);
        escola.matricularAluno(aluno4);
        escola.matricularAluno(aluno5);
        escola.matricularAluno(aluno6);
        escola.matricularAluno(aluno7);
        escola.matricularAluno(aluno8);
        escola.matricularAluno(aluno9);
    }
}
```

Análise do agrupamento de objetos

Vamos focar a nossa atenção no código da classe “Escola”, pois é nele que criamos uma lista de objetos do tipo “Aluno” por meio do método de agrupamento “agruparAlunos”. Nesse método, 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> >”).

Na sequência, temos um laço que 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. Ao final, adicionamos o objeto à lista correspondente à chave existente no mapa. A saída é dada por:

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)]

}

Podemos ver 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

Agora, vamos avançar ainda mais na manipulação de dados com o Java usando a classe "Collectors". Essa classe da API Java implementa vários métodos úteis de operações de redução, como o agrupamento dos objetos em coleções.

A operação de agrupamento é feita pelo método "groupingBy". Esse método possui três variantes sobrecarregadas, onde suas assinaturas são:

1. static <T,K> Collector<T,?,Map>> groupingBy(Function <? super T,? extends K> classifier)
2. static <T, K, D, A, M extends Map<K, D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)
3. static <T, K, A, D> Collector<T,?,Map> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

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.

Essas chaves de particionamento são os rótulos de cada grupo ou coleção de objetos formados. Dessa forma, conforme o agrupador da classe "Collectors" lê os objetos do fluxo de entrada, ele cria 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.

Atenção

A cláusula "static Collector" 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" é 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.

Neste vídeo, aprenderemos sobre a classe Collectors do Java, um recurso avançado para manipular grandes volumes de dados. Ela oferece diversas funcionalidades que aumentam a eficiência do sistema. Embora a sintaxe não seja trivial, a prática é essencial para uma melhor assimilação.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Passo 1: Uso da assinatura 1

Implemente a reescrita do método “agruparAlunos” usando a primeira assinatura de “groupingBy”, como podemos ver no código a seguir:

```
java

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();
        this.discentes=new ArrayList<>();
    }
    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> agrupamento=

discentes.stream().collect(Collectors.groupingBy(Aluno::recuperarNaturalidade));
            System.out.println("Resultado do agrupamento por naturalidade: ");
            agrupamento.forEach((String chave,List lista)->System.out.println(chave+" =
"+lista));
    }
}
```

Passo 2

Execute o código em que o resultado é:

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)]

No nosso exemplo, o 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).

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

Passo 3: Uso da assinatura 2

Implemente a 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”. Acompanhe!

```
java

public void agruparAlunos(int a) {
    Map>agrupamento =
discentes.stream().collect(Collectors.groupingBy(Aluno::recuperarNaturalidade,Collectors.t
oSet()));
    System.out.println("Resultado do agrupamento por naturalidade: ");
    agrupamento.forEach((String chave,Setconjunto)-> System.out.println(chave+" =
"+conjunto));
}
```

A saída é aquela mostrada para a execução do código anterior.

Passo 4: Uso da assinatura 3

Implemente a sobrecarga do método “agruparAlunos”, que utiliza a terceira assinatura de “groupingBy”:

```
java

public void agruparAlunos(double a) {
    Map>
agrupamento=discentes.stream().collect(Collectors.groupingBy(Aluno::recuperarNaturalidade,
TreeMap::new,Collectors.toSet()));
    System.out.println("Resultado do agrupamento por naturalidade: ");
    agrupamento.forEach((String chave,Setconjunto)-> System.out.println(chave+" =
"+conjunto));
}
```

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”).

Passo 5

Execute o código cujo resultado da execução é:

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)]

Devemos notar que a ordem das coleções está diferente do caso anterior. Isso porque o mecanismo "TreeMap" mantém as suas entradas ordenadas. No entanto, podemos perceber que os agrupamentos são iguais.

Faça você mesmo!

O uso dos coletores no Java exige bastante experiência. No entanto, o nosso objetivo é exatamente dominar a linguagem Java e nos destacar no mercado. Então, devemos nos esforçar para superar mais esse desafio. Vamos considerar o seguinte trecho incompleto de código em Java:

```
java

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class ExemploMapeamento {
    public static void main(String[] args) {
        List numeros = List.of(1, 2, 3, 4, 5);
        List quadradoNumeros = map(numeros, ???);
        System.out.println("Números Originais: " + numeros);
        System.out.println("Números ao Quadrado: " + quadradoNumeros);
    }
    public static  List map(List lista, Function mapa) {
        List resultado = new ArrayList<>();
        for (T item : lista) {
            R itemMapeado = mapa.apply(item);
            resultado.add(itemMapeado);
        }
        return resultado;
    }
}
```

Nós queremos que o programa exiba o seguinte resultado:

Números Originais: [1, 2, 3, 4, 5]

Números ao Quadrado: [1, 4, 9, 16, 25]

Nesse sentido, precisamos modificar o trecho "???" por um comando adequado. Então, mãos à obra e vamos superar mais esse desafio.

Chave de resposta

O programa completo vai ficar da seguinte forma:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

public class ExemploMapeamento {
    public static void main(String[] args) {
        List numeros = List.of(1, 2, 3, 4, 5);
        List quadradoNumeros = map(numeros, x -> x * x);
        System.out.println("Números Originais: " + numeros);
        System.out.println("Números ao Quadrado: " + quadradoNumeros);
    }
    public static  List map(List lista, Function mapa) {
        List resultado = new ArrayList<>();
        for (T item : lista) {
            R itemMapeado = mapa.apply(item);
            resultado.add(itemMapeado);
        }
        return resultado;
    }
}
```

Ou seja, trocamos a expressão “???” por “x → x * x” que significa que a função recebe um valor e calcula o quadrado dele.

Coleções em Java

Coleções, por vezes chamadas de containers, são objetos capazes de agrupar múltiplos elementos em uma única unidade. Elas têm por finalidade armazenar, manipular e comunicar dados agregados, de acordo com o Oracle America Inc. (2021). Por causa da importância dessas funcionalidades, vamos estudá-las sob o ponto de vista teórico e prático.

Neste vídeo, você aprenderá sobre as principais "coleções" disponíveis no Java. Exploraremos os diferentes tipos de coleções que o Java oferece e apresentaremos exemplos práticos de seu uso.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Seção 1: Tipos de coleções no Java

Em Java, nós utilizamos o termo "Coleções" para trabalharmos com uma estrutura ou conjunto de classes e interfaces que fornecem uma maneira de armazenar, manipular e acessar grupos de objetos. As coleções são, especialmente, importantes, pois elas nos permitem trabalhar com grupos de objetos como uma única entidade e fornecem várias estruturas de dados e algoritmos para gerenciar coleções de elementos com eficiência.

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”. Há, ainda, as coleções “SortedSet” e “SortedMap” que são, essencialmente, versões ordenadas de “Set” e “Map”, respectivamente.

Conheça a seguir alguns dos principais tipos de coleções suportados pelo Java:

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

Trata-se de uma coleção que implementa algo mais genérico, embora o nome faça referência ao conceito de filas. 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 observação, precisamos destacar que “Map” não é verdadeiramente uma coleção. Esse tipo de classe cria um mapeamento entre chaves e valores, conforme vimos nos exemplos anteriores. Além de não admitir chaves duplicadas, a interface “Map” mapeia uma chave para um único valor.

Seção 2: Exemplo prático de coleções

Nós já vimos alguns exemplos que trabalham com coleções. Agora, vamos implementar um exemplo completo que utiliza um ArrayList para gerenciar o ciclo de vida dos dados, ou seja, vamos inserir dados, fazer alterações, pesquisas e exclusão. A seguir, apresentamos o código completo em Java:

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList lst_numeros = new ArrayList<>();

        // Inserção dos elementos no ArrayList
        lst_numeros.add(10);
        lst_numeros.add(20);
        lst_numeros.add(30);
        lst_numeros.add(40);
        lst_numeros.add(50);

        // Acesso aos elementos no ArrayList
        System.out.println("Os elementos no ArrayList são:");
        for (int i = 0; i < lst_numeros.size(); i++) {
            System.out.println("lista["+i+"]= "+lst_numeros.get(i));
        }

        // Remove um elemento de um posição específica do ArrayList
        lst_numeros.remove(1);  // Remove o elemento da posição 2 do ArrayList

        // Alterar um elemento no ArrayList
        int x=57;
        lst_numeros.set(0, x);  // Coloca o elemento 57 na posição 0 do ArrayList

        // Verifica se o ArrayList contém um elemento específico
        int n = 100;
        String contem_elemento = lst_numeros.contains(n)?"Verdade":"Falso";
        System.out.println("O elemento "+n+" está the ArrayList? " + contem_elemento);

        // Iterar na lista através do laço for-each
        int k=0;
        System.out.println("Os elementos no ArrayList são:");
        for (int elemento : lst_numeros) {
            System.out.println("lista["+k+"]= "+elemento);
            k++;
        }

        // Limpar o ArrayList de todos os elementos
        System.out.println("Limpar o ArrayList. ");
        lst_numeros.clear();

        // Verifica se o ArrayList está vazio
        String eh_vazio = lst_numeros.isEmpty()?"Verdade":"Falso";
        System.out.println("O ArrayList está vazio? " + eh_vazio);
    }
}
```

Colocamos comentários sobre as funcionalidades. Ao executar o código, obtemos o seguinte resultado:

Os elementos no ArrayList são:

lista[0]= 10

lista[1]= 20

lista[2]= 30

lista[3]= 40

lista[4]= 50

O elemento 100 está the ArrayList? Falso

Os elementos no ArrayList são:

lista[0]= 57

lista[1]= 30

lista[2]= 40

lista[3]= 50

Limpar o ArrayList.

O ArrayList está vazio? Verdade

De fato, o uso de coleções em Java, como o ArrayList, facilita o trabalho de desenvolvimento e deixa o código mais legível para realizarmos manutenções posteriormente.

Atividade 1

Os agrupamentos de dados são um importante recurso que o Java que nos oferece com objetivo de manipular grandes volumes de dados que possuem características em comum, semelhante ao que fazemos na manipulação de dados em bancos de dados. Nesse sentido, avalie as seguintes afirmações feitas acerca da linguagem Java:

1. Ao utilizar o método “groupingBy” da classe “Collectors”, o programador tem de informar o atributo a ser usado para o agrupamento.
2. Os objetos agrupados são armazenados em um container que é mapeado para a chave de agrupamento.
3. O método “groupingBy” só armazena os objetos em coleções do tipo “List”.

Está correto apenas o que se afirma em:

A

I.

B

II.

C

III.

D

I e II.

E

II e III.

A alternativa B está correta.

O retorno do método “groupingBy” é um “Collector” que cria uma estrutura “Map”. Essa estrutura mantém um mapeamento entre a chave de agrupamento e o container que contém os objetos agrupados. As demais afirmativas são falsas.

Atividade 2

O Java oferece diversos tipos de coleções. De fato, isso pode ser muito útil para trabalhar com determinados cenários em que os dados devem obedecer a alguma política de manipulação, como por exemplo, uma estrutura do tipo FIFO (Primeiro a Entrar e Primeiro a Sair), ou LIFO (Último a Entrar, Primeiro a Sair). Nesse sentido, selecione a única alternativa verdadeira a respeito das coleções em Java:

A

As coleções em Java não admitem elementos duplicados.

B

O container “Queue” é uma fila FIFO.

C

Uma pilha pode ser implementada com o container “Deque”.

D

Nenhum container permite definir a posição do objeto a ser inserido.

E

Os containers não podem ser usados em programação concorrente.

A alternativa C está correta.

O container “Deque” possibilita inserção e remoção em ambas as extremidades, o que permite implementar uma pilha. É verdade, porém, que podemos utilizar um container “Deque” de outras formas também. De fato, o Java facilita bastante a manipulação de dados. Esse fato contribuiu bastante para a popularidade da linguagem e, durante muito tempo, ela continuará a ser bastante relevante.

Java versus C/C++: um breve comparativo

Neste estudo, faremos uma breve comparação das linguagens C/C++ com o Java. De fato, sob o aspecto cronológico, esta comparação faz bastante sentido, apesar de que essas linguagens possuem grandes diferenças sintáticas e, principalmente, têm escopos de uso bem distintos.

Neste vídeo, exploraremos a evolução das linguagens C e C++ em direção ao Java. Analisaremos como o Java foi influenciado por essas linguagens. Além disso, apresentaremos um exemplo prático em C++ que será útil para compararmos com a sintaxe do Java.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Alguns aspectos de C++

Segundo o criador da linguagem C++, Bjarne Stroustrup (2020), não devemos comparar Java com o C++, pois ambos possuem condicionantes de desenvolvimento bem distintos. Stroustrup aponta, também, que muito da simplicidade de Java é uma ilusão. De fato, essa colocação mostrou-se correta. Atualmente, um sistema desenvolvido em Java depende de vários frameworks e outras tecnologias que tornam um projeto muito complexo.

As diferenças mencionadas por Stroustrup se mostram presentes, por exemplo, no fato do C++ ter uma checagem estática fraca, enquanto o Java impõe mais restrições. O C++ possui essa característica para privilegiar a performance a interação com o hardware, enquanto o Java tem o foco em 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 o C++ não faz, conforme explica Sutter (2021).

Comentário

Talvez um dos pontos mais interessantes seja a questão da portabilidade. O C++ buscou alcançá-la por meio da padronização.

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

No caso do Java, a portabilidade também é um dos pontos fortes. Veja a seguir um apontamento importante.

> O Java não é uma linguagem independente de plataforma, e sim uma plataforma.
>
> —
>
> (STROUSTRUP, 2020)

O motivo dessa afirmação é que um software desenvolvido em Java não é executado pelo hardware, mas, sim, pela Máquina Virtual Java (JVM), que é uma plataforma emulada que abstrai o hardware sobre o qual a aplicação está executando.

A JVM expõe sempre para o programa a mesma interface, enquanto ela própria é um software acoplado a uma plataforma de hardware específica. A vantagem dessa abordagem é que um software em Java pode rodar em diferentes hardwares, por outro lado, o desempenho dele é inferior ao de um software desenvolvido em C++.

Alguns aspectos de C

A comparação entre Java e a linguagem C é mais difícil do que a comparação entre Java e C++. A linguagem C é conhecida por ter como ponto forte sua interação com sistemas de baixo nível, sendo amplamente utilizada em drivers de dispositivo. A diferença entre Java e C é maior do que entre Java e C++. Vamos conferi-las!

As linguagens Java e C++ são OO (Orientadas a Objeto).

A linguagem C é aderente ao paradigma de programação estruturado. Não possui conceito de classes e objetos.

Qualquer programa em Java precisa ter ao menos uma classe. Além disso, o Java ainda aceita declarações como interface e enum. Isso significa que não é possível aplicar o paradigma estruturado em Java. Comparar Java e C, portanto, serve apenas para apontar as diferenças dos paradigmas de programação orientada a objetos e estruturada.

Exemplo prático em C++

Aqui, apresentamos um programa em C++. O objetivo é encontrar o maior número de um vetor. Vamos observar o código a seguir:

```cpp
#include
using namespace std;
class Utilitario {
private:
  int* vetor;
  int tamanho;
public:
  Utilitario(int* vet, int tam) {
     vetor = vet;
     tamanho = tam;
  }
  int encontrarMaiorNumero() {
      int maior = vetor[0];
      for (int i = 1; i < tamanho; i++) {
      if (vetor[i] > maior) {
            maior = vetor[i];
      }
    }
    return maior;
  }
};
int main() {
  int numeros[]={5,7,9,10};
  int tam = end(numeros)-begin(numeros);
  std::cout << "\nA lista de numeros eh:" << std::endl;
  for (int i = 0; i < tam; i++) {
        std::cout << numeros[i]<<" ";
  }
  Utilitario objUtil(numeros, tam);
  int maximo = objUtil.encontrarMaiorNumero();
  std::cout << "\nO maior numero eh: " << maximo << std::endl;
  return 0;
}
```

Realmente, podemos notar semelhanças com um código em Java, mas a linguagem C++ ainda apresenta algumas características inseguras, como o acesso à memória. No entanto, ela continua sendo amplamente utilizada em aplicações que requerem alto desempenho, como no desenvolvimento de jogos.

Agora colocaremos em prática o que aprendemos! Vamos para a atividade prática!

Atividade 1

Java é uma linguagem de programação rápida, segura e confiável para codificar tudo, desde aplicações móveis e software empresarial até aplicações de big data e tecnologias do servidor. Agora, analise as afirmativas a seguir:

(I) O Java não é uma linguagem independente de plataforma, ela é uma plataforma.

PORQUE

(II) Um software desenvolvido em Java não é executado pelo hardware, mas, sim, pela Máquina Virtual Java (JVM).

A

As duas afirmações estão corretas e a segunda justifica a primeira.

B

As duas afirmações estão corretas e a segunda não justifica a primeira.

C

A primeira afirmação é correta e a segunda falsa.

D

A primeira afirmação é falsa e a segunda correta.

E

As duas afirmações são falsas.

A alternativa A está correta.

Java é uma linguagem multiplataforma, orientada a objetos e centrada em rede que é utilizada como uma plataforma em si.s classes.

Ambientes de desenvolvimento Java

Um ambiente de desenvolvimento integrado (IDE) desempenha um papel crucial no desenvolvimento Java. Isso ocorre porque uma IDE oferece assistência de codificação, recursos de depuração, teste integrado, integração de controle de versão, gerenciamento de projetos, suporte a documentação, extensibilidade e customização. Aqui, vamos conhecer identificar os principais elementos de uma IDE para desenvolver programas em Java.

Neste vídeo, aprenderemos sobre os principais ambientes de desenvolvimento de programação em Java e também entenderá elementos essenciais, como JVM e JRE.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Fundamentos sobre a JVM

No geral, um ambiente de desenvolvimento fornece um conjunto abrangente de ferramentas e recursos que aprimoram a produtividade, a qualidade do código e a colaboração para desenvolvedores Java. Ele simplifica e agiliza o processo de desenvolvimento, tornando mais fácil escrever, testar, depurar e gerenciar aplicativos Java com eficiência.

Para falarmos de ambientes de desenvolvimento, precisamos esclarecer alguns conceitos, tais como:

- A máquina virtual Java (MVJ) – em inglês, Java virtual machine (JVM)
- O Java runtime environment (JRE)

- O Java development kit (JDK)

Como já sabemos, a JVM é 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.

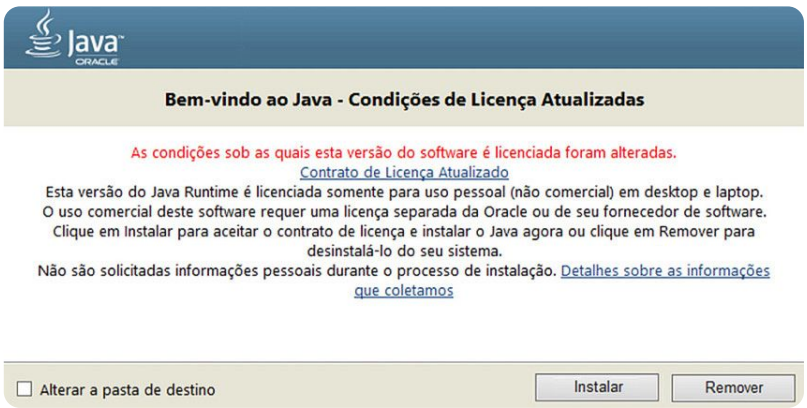
A JVM é um software que implementa a especificação mencionada e é sempre aderente à plataforma física. Contudo, ela provê para os programas uma plataforma padronizada, garantindo que o código Java sempre possa ser executado, desde que exista uma JVM. Ela não executa as instruções da linguagem Java, mas, 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. O JDK engloba o JRE e mais um conjunto de ferramentas de desenvolvimento, tais como:

1. Um interpretador Java (java)
2. Um compilador (javac)
3. Um programa de arquivamento (jar)
4. Um gerador de documentação (javadoc)

Dois JDK muito utilizados são o **Oracle JDK** e o **OpenJDK**. Vejamos a seguir um aviso legal exibido durante a instalação do Oracle JDK (Java SE 17), um software cujo uso gratuito é restrito.



Aviso legal na instalação do Oracle JDK.

Fundamentos sobre o ambiente de desenvolvimento integrado (IDE)

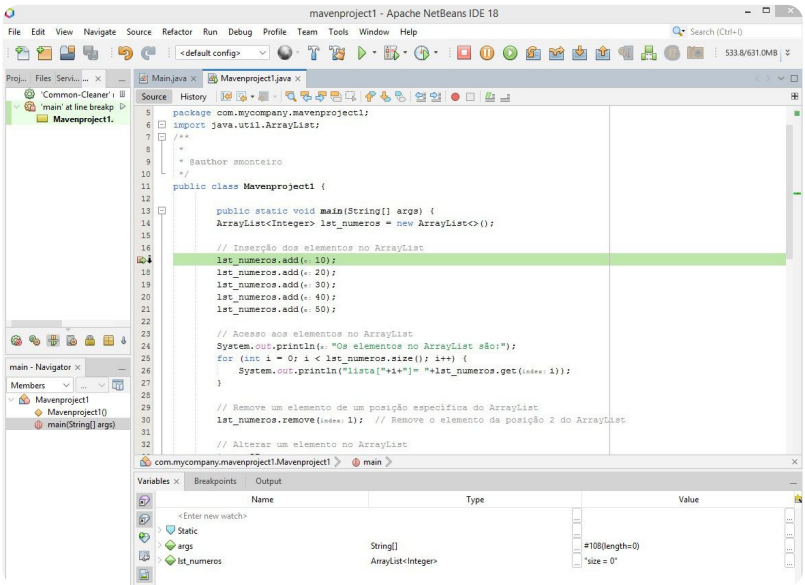
Um **integrated development environment** (IDE), ou ambiente integrado de desenvolvimento, é um software que reúne ferramentas de apoio e funcionalidades com o objetivo de facilitar e acelerar o desenvolvimento de software.

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.

Curiosidade

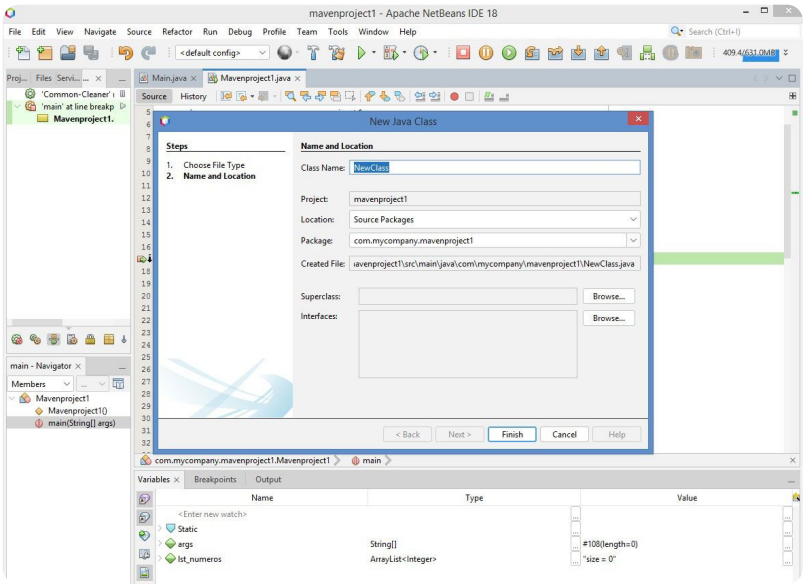
O Netbeans é um IDE mantido pela The Apache Software Foundation e licenciado segundo a licença Apache versão 2.0. De acordo com o site do IDE, ele é o IDE oficial do Java 8, mas também permite desenvolver em HTML, JavaScript, PHP, C/C++, XML, JSP e Groovy. É um IDE multiplataforma que pode ter suas funcionalidades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

Observe a seguir o IDE durante uma depuração.



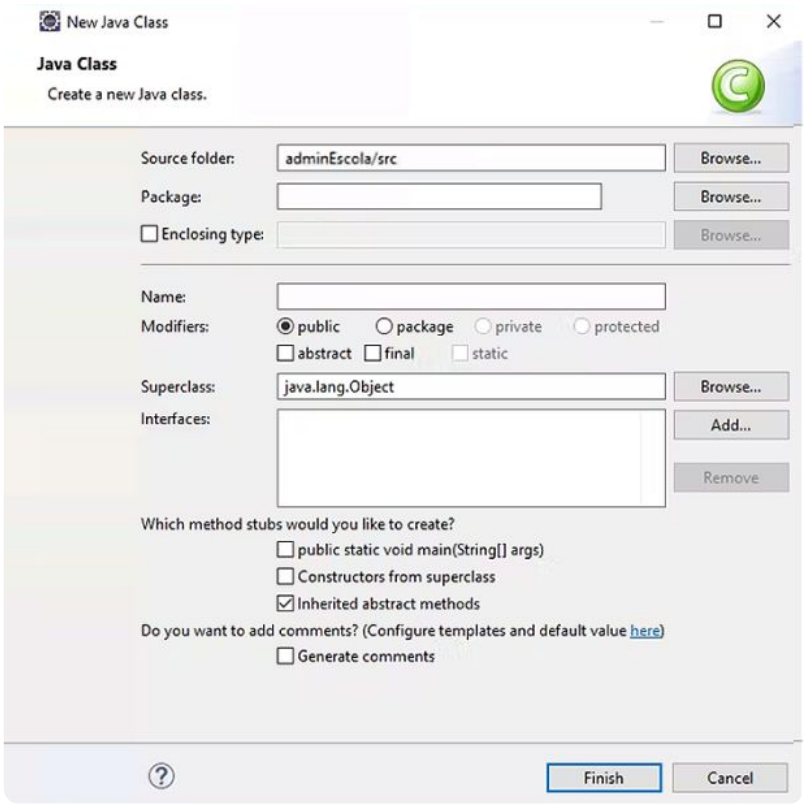
IDE Netbeans.

Apesar de possuir muitas funcionalidades, a parte de modelagem realiza apenas a declaração de classes, sem criação automática de construtor ou métodos, conforme observamos na imagem seguinte.



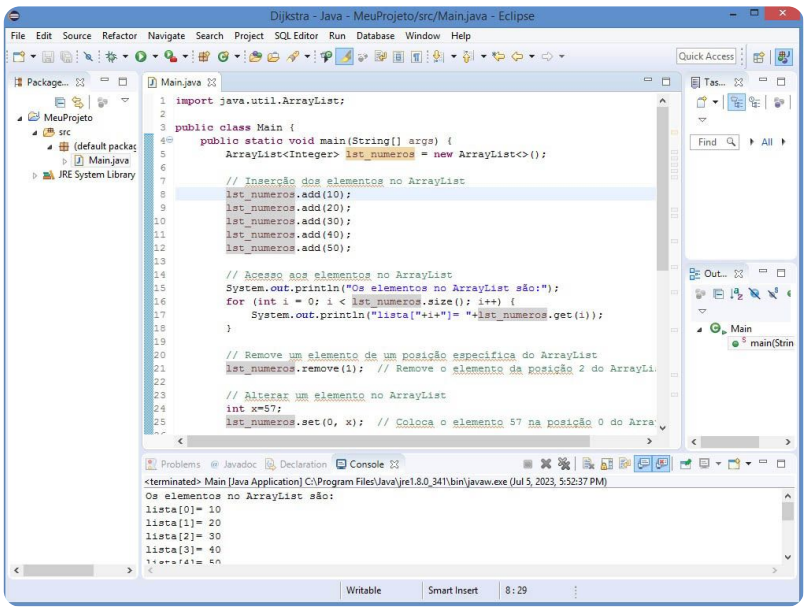
Criação de classe no Netbeans.

Outra IDE muito popular para desenvolver aplicações Java é o **Eclipse**. 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. Por exemplo, na imagem a seguir, apresentamos uma visão geral do Eclipse:



Criação de classe no Eclipse.

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 às do Netbeans. Sendo assim, vejamos na imagem a seguir um IDE Eclipse que mostra um exemplo de depuração no Eclipse.



IDE Eclipse.

Atividade 2

A linguagem Java oferece diversos recursos interessantes para o desenvolvimento de projetos sofisticados. Para isso, precisamos de alguns elementos básicos para criar esses programas. Nesse sentido, selecione a opção correta que contém todos os elementos que são imprescindíveis para realizar um desenvolvimento em Java:

A

JRE, IDE e máquina virtual Java.

B

Máquina Java, IDE e editor de código.

C

JDK e editor de código.

D

JRE, IDE e editor de código.

E

JDK, JRE e máquina virtual Java.

A alternativa C está correta.

O JDK contém o JRE e a máquina virtual Java, mas não possui aplicativo de edição de código, que precisa ser complementado.

Estrutura e principais comandos de um programa em Java

Na linguagem Java, o ponto de entrada para a execução do programa é a função “main”, mas não é obrigatório para compilar o programa. Ao longo deste estudo, vamos analisar alguns dos principais aspectos de uma classe do Java. Além de conhecermos melhor o comando “switch-case” que é bastante útil em algumas situações.

Neste vídeo, abordaremos os principais elementos de um programa em Java, focando no ciclo de vida de um objeto. Exploraremos a importância do padrão get/set para os métodos das classes e finalizaremos com uma análise do comando "switch-case".

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Classe padrão no Java

Uma classe padrão no Java não será executada a menos que haja um ponto de partida explícito. 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 (que vai rodar apenas em uma máquina), um ponto de entrada pode ser definido pelo método “main”. Entretanto, diferentemente de C/C++, esse método deverá pertencer a uma classe.

No código a seguir, a função “main” define o ponto de entrada da aplicação. Esse será o primeiro método a ser executado pela JVM:

```
java

public class Main {
    public static void main (String args[]) {
    //Código
    }
}
```

Normalmente, é no corpo da função “main” que fazemos a instanciação de classes, ou seja, criamos objetos.

Métodos de acesso

Não são comandos Java nem uma propriedade da linguagem, e sim consequências do encapsulamento. Eles apareceram em alguns códigos mostrados aqui. 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.

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, conheça a seguir quais são eles:

1. Um para obter seu valor (“get”).
2. Outro para inseri-lo (“set”).

Em seguida, apresentaremos no código dois métodos de acesso ao atributo “nome”.

```
java

public class Base {
    private private nome;
    public void setNome(String nome){
        this.nome=nome;
    }
    public String getNome(){
        return this.nome;
    }
}
```

No caso, o método “**setNome**” é utilizado para atribuir valor para o atributo “nome”, enquanto o método “**getNome**” é utilizado para obter o valor do atributo “nome”.

Comando switch

Um comando bastante útil no Java também é o “switch”, onde a sintaxe é a seguinte:

```
java

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 as “default” são opcionais, mas seu uso é uma boa prática de programação. Apresentaremos, a seguir, um exemplo de código que utiliza os comandos “**switch-case**”, “**break**” e “**default**”:

```
java

public class Base {
    private String linguagem = "JAVA ";
    public void desvio (){
        switch ( linguagem ) {
            case ( "C" ):
                    System.out.println("Suporta apenas programação estruturada");
                    break;
            case ( "C++" ):
                    System.out.println("Suporta programação estruturada e orientada a
objeto");
                    break;
            case ( "JAVA "):
                    System.out.println("Suporta apenas programação orientada a objeto");
                    break;
            default:
                    System.out.println("Erro!");
                    break;
        }
    }
}
```

Principalmente, em situações que precisamos fazer escolha entre muitas alternativas, o comando “switch” é uma opção muita prática.

Estruturas condicionais

Neste vídeo, você vai compreender as estruturas condicionais do JavaScript, entre elas as instruções if, else, else if e switch. Confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Segundo Flanagan (2011), as estruturas de decisão, também conhecidas como condicionais, são instruções que executam ou pulam outras instruções, dependendo do valor de uma expressão especificada. São os pontos de decisão do código, também conhecidos como ramos, uma vez que podem alterar o fluxo do código, criando um ou mais caminhos.

Aprofundando o conceito

Para melhor assimilação do conceito de estruturas condicionais, vamos usar um exemplo a partir do código construído anteriormente, como veremos a seguir:

```
javascript




 Resultado da Multiplicação:
```

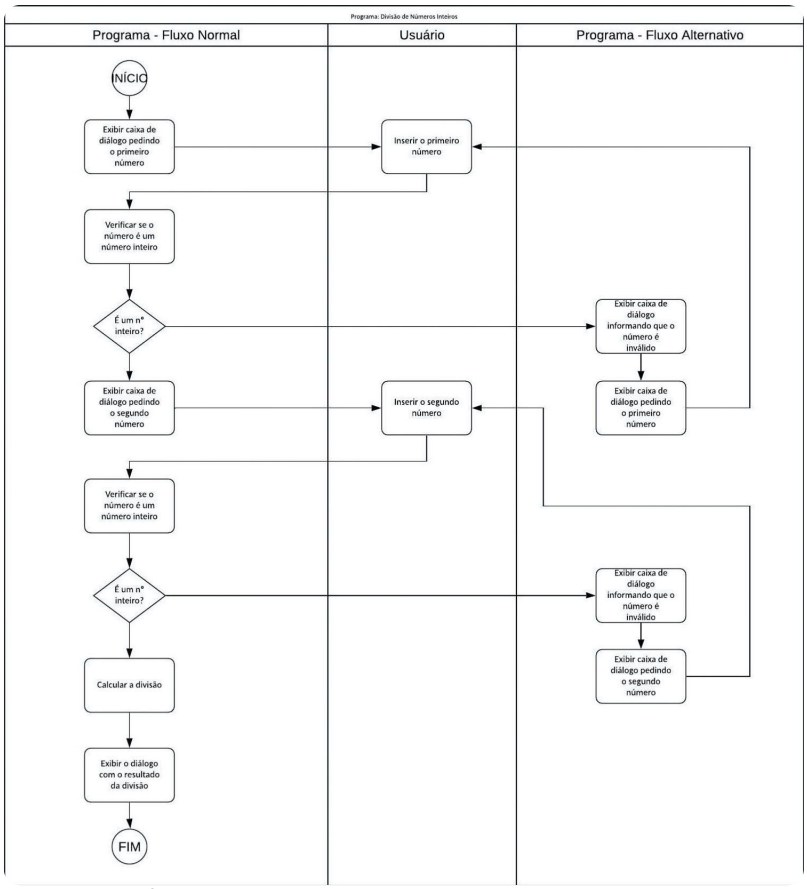
As orientações do programa afirmam que deve ser realizada a divisão de dois números inteiros positivos.

O que acontece se o usuário inserir um número inteiro que não seja positivo? Ou como forçá-lo a inserir um número positivo?

Para essa função, podemos utilizar uma condição, ou seja, se o usuário inserir um número inteiro não positivo, deve-se avisar que o número não é válido, solicitando que seja inserido um número válido.

Nesse caso, o fluxo normal do programa é receber dois números positivos, calcular a divisão e exibir o resultado. Perceba que a condição cria um novo fluxo, um novo ramo, em que outro diálogo é exibido, e o usuário é levado a inserir novamente o número.

O fluxo normal e o fluxo resultado da condicional podem ser vistos na imagem a seguir, em que são apresentados os passos correspondentes ao nosso exercício, separando as ações do programa e as do usuário.



Fluxo normal e fluxo alternativo.

Repare que a verificação “é um nº inteiro positivo” permite apenas duas respostas: “sim” e “não”. Essa condição, mediante a resposta fornecida, é responsável por seguir o fluxo normal do código ou o alternativo.

O fluxograma de exemplo foi simplificado para fornecer mais detalhes. Logo, a respectiva notação padrão não foi utilizada em sua confecção.

Nas linguagens de programação, utilizamos as instruções condicionais para implementar o tipo de decisão apresentado no exemplo. Em JavaScript, estão disponíveis as instruções "if/else" e "switch", como veremos a seguir.

Instrução “If”

A sintaxe da instrução "if/else" em JavaScript possui algumas formas. A primeira e mais simples é apresentada do seguinte modo:

if (condição) instrução

Nessa forma, é verificada uma única condição. Caso seja verdadeira, a instrução será executada. Do contrário, não. Antes de continuarmos, cabe destacar os elementos da instrução:

- É iniciada com a palavra reservada “if”.
- É inserida, dentro de parênteses, a condição (ou condições).
- É inserida a instrução a ser executada, caso a condição seja verdadeira.

Outro detalhe importante: caso exista mais de uma instrução para ser executada, é necessário envolvê-las em chaves. Veja o exemplo:

```javascript
if (condição1 && condição2){
instrução1;
instrução2;
}
```

Nesse segundo caso, além de mais de uma instrução, também temos mais de uma condição. Quando é necessário verificar mais de uma condição, em que cada uma delas precisa ser verdadeira, utilizamos os caracteres “&&”.

Na prática, as instruções 1 e 2 só serão executadas caso as condições 1 e 2 sejam verdadeiras. Vamos a outro exemplo:

```javascript
if (condição1 || condição2){
instrução1;
instrução2;
}
```

Repare que, nesse código, os caracteres “&&” foram substituídos por “||”. Esses últimos são utilizados quando uma ou outra condição precisa ser verdadeira para que as instruções condicionais sejam executadas.

> E o que acontece se quisermos verificar mais condições?

Nesse caso, podemos fazer isso tanto para a forma em que todas as condições precisam ser verdadeiras, separadas por “&&”, quanto para a forma em que apenas uma deve ser verdadeira, separadas por “||”. Além disso, é possível combinar os dois casos na mesma verificação. Veja o exemplo:

```javascript
if ( (condição1 && condição2) || condição3){
instrução1;
instrução2;
}
```

Nesse fragmento, as duas primeiras condições são agrupadas por parênteses. A lógica aqui é a seguinte:

Execute as instruções 1 e 2 SE as condições 1 e 2 forem verdadeiras OU se a condição 3 for verdadeira.

Por fim, há outra forma: a de negação.

> Como verificar se uma condição é falsa (ou não verdadeira)?

Veremos a seguir:

```
javascript

if (!condição1){
instrução1;
instrução2;
}
```

O sinal “!” é utilizado para negar a condição. As instruções 1 e 2 serão executadas caso a condição 1 não seja verdadeira.

Vamos praticar?

Nos três emuladores de código a seguir, apresentamos as estruturas de decisão vistas até o momento. No primeiro emulador, temos o uso da estrutura de decisão “if” de maneira simples, contendo apenas uma única condição:

Conteúdo interativo

esse a versão digital para executar o código.

Já no emulador seguinte, a estrutura de decisão “if” é implementada com duas condições, além dos operadores lógicos AND (&&) e OR (||):

Conteúdo interativo

esse a versão digital para executar o código.

Por fim, no emulador a seguir, temos a estrutura “if” sendo usada de uma maneira mais elaborada, com mais de duas condições, combinação dos operadores && e ll, assim como o uso do operador lógico de negação NOT (!):

Conteúdo interativo

esse a versão digital para executar o código.

Instrução “else”

A instrução “else” acompanha a instrução “if”. Embora não seja obrigatória, como vimos nos exemplos, sempre que “else” for utilizado, deve vir acompanhado de “if”. O “else” indica se alguma instrução deve ser executada caso a verificação feita com o “if” não seja atendida. Vejamos:

```
javascript

if(número fornecido é inteiro e positivo){
     Guarde o número em uma variável;
}else{
     Avise ao usuário que o número não é válido;
     Solicite ao usuário que insira novamente um número;
}
```

Perceba que o “else” (senão) acompanha o “if” (se). Logo, SE as condições forem verdadeiras, faça isto; SENÃO, faça aquilo.

No último fragmento, foi utilizado, de modo proposital, **português-estruturado** nas condições e instruções. Isso porque, mais adiante, você mesmo codificará esse "if/else" em JavaScript.

Português estruturado

Linguagem de programação ou pseudocódigo que utiliza comandos expressos em português.

Instrução “else if”

Veja o exemplo a seguir:

```
javascript

if (numero1 < 0){
     instrução1;
     }else if(numero == 0){
          instrução2;
     }else{
          instrução3;
     }
```

Repare que uma nova instrução foi usada no fragmento. Trata-se de “else if”, instrução utilizada quando queremos fazer verificações adicionais sem agrupá-las todas dentro de um único “if”. Além disso, ao utilizarmos essa forma, caso nenhuma das condições constantes no “if” e no(s) “if else” seja atendida, a instrução “else” será executada obrigatoriamente ao final.

Recomendação

Otimize os códigos presentes nos emuladores anteriores usando o “else if”. Como exemplo, apresentamos o código do primeiro emulador modificado, no qual as quatro estruturas de decisão com “if” foram transformadas em uma única estrutura de decisão.

Note que, antes, eram geradas duas saídas redundantes (“a é maior que b” e “b é menor que a”), pois se tratava de quatro estruturas independentes. Por isso, todas elas eram avaliadas. Isso não ocorrerá mais com o uso de uma estrutura de decisão composta de “if” e “else if”, pois, quando a primeira condição verdadeira for encontrada (“a é maior que b”), nenhuma das outras condições será avaliada. Logo, teremos:

```javascript
var a = 10;
var b = 3;

console.log ("if com uma única condição:");
if (a > b){
     console.log("a é maior que b");
} else if (a == b){
     console.log("a é igual a b");
} else if (a < b){
     console.log("a é menor que b");
} else if (b < a){
     console.log("b é menor que a");
}
```

Instrução “switch”

A instrução “switch” é bastante útil quando uma série de condições precisa ser verificada. É bastante similar à instrução “else if”. Vejamos:

```javascript
switch(numero1){
     case 0:
          instrução1;
          break;
     case 1:
          instrução2;
          break;
     default:
          instrução3;
          break;
}
```

De maneira geral, o switch é usado quando há uma série de condições, nas quais diversos valores para a mesma variável são avaliados. Vamos detalhar o código anterior:

- Após o “swicth” dentro de parênteses, temos a condição a ser verificada.
- A seguir, temos os “case”, em quantidade equivalente às condições que queremos verificar.
- Depois, dentro de cada “case”, temos a(s) instrução(ões) e o comando “break”.
- Por fim, temos a instrução “default”, que será executada caso nenhuma das condições representadas pelos “case” seja atendida.

Comandos iterativos

Outro grupo de comandos importantes da linguagem Java são as estruturas de repetição. As três estruturas providas pela Java são “while”, “do-while” e o laço “for”. Neste estudo, vamos analisar a sintaxe e alguns exemplos práticos de como utilizá-las. O comando “break”, visto na estrutura switch anteriormente, interrompe o laço/estrutura de controle atual, como o “while”, “for”, “do ... while”.

Neste vídeo, exploraremos as estruturas de repetição disponibilizadas pelo Java, tanto de forma conceitual quanto prática. Abordaremos os comandos "while" e "for", aprofundando nosso conhecimento sobre essas poderosas ferramentas de iteração.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Comando while

Semelhante a outras linguagens de programação, o Java possui o comando iterativo “while”, cuja estrutura tem a seguinte sintaxe:

```
java

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

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. No exemplo a seguir, apresentamos um código que imprime os valores de 0 até 9:

```
java

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

Para obter esse resultado, ele utiliza o valor da variável “controle” para verificar se a condição de teste é válida. Caso seja, ele exibe o valor da variável e, na sequência, incrementa.

O Java oferece, ainda, a estrutura “**do-while**”, onde o funcionamento é bem parecido. Porém, ao contrário de “**while**”, nessa estrutura o bloco sempre será executado ao menos uma vez. Veja sua sintaxe:

```
java

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

Como vemos pela sintaxe, o primeiro “bloco” é executado e, apenas depois disso, a expressão é avaliada. A repetição continua até que a expressão seja falsa. A seguir, veremos um exemplo do comando “do-while”:

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

Nesse caso, o programa vai exibir os valores de 0 até 9.

Comando for

A última estrutura de repetição que veremos é o laço “for”, que possui a seguinte sintaxe:

```java
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, o item “iteração” não é executado. Veja um exemplo do uso do comando “for”:

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

O Java oferece ainda mais uma possibilidade do comando “for” que é chamada de “for-each”. Esse laço é empregado para iterar sobre uma coleção de objetos, de maneira sequencial. Sendo assim, a sua sintaxe é dada por:

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

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

Agora, vamos para prática!

Atividade 4

A criação de programas interativos inclui o uso de alguns comandos que permitem que, dependendo das circunstâncias, os programas executem instruções diferente, tais como “while”, “for” e outros. Nesse contexto, o comando “break” tem a função de

A

interromper a execução de um loop.

B

condicionar a execução de um comando de atribuição a um operador lógico.

C

segmentar a execução de um loop em duas ou mais partes aninhadas.

D

estabelecer um intervalo de depuração durante a execução de um loop.

E

impossibilitar o aninhamento de loops não lógicos.

A alternativa A está correta.

O comando “break” é usado para interromper a execução de um laço de iteração ou de um comando “switch”.

Considerações finais

O que você aprendeu neste conteúdo?

- Os conceitos de programação orientada a objetos na linguagem de programação Java.
- Como utilizar na prática as propriedades de encapsulamento, herança e polimorfismo.
- Identificar o uso adequado de objetos e como utilizá-los para fazer referências.
- Trabalhar com agrupamentos de objetos em Java e coleções.
- Reconhecer as principais IDEs para desenvolver um programa em Java.
- Fizemos uma rápida comparação entre as linguagens Java e C/C++.
- Revisamos os principais comandos condicionais e iterativos do Java.

Explore +

- Acesse o site oficial da Oracle, pesquise por “Collectors”. Nele você vai aprender ainda mais sobre principais conceitos de Coletores e encontrará diversos exemplos que vão lhe ajudar aprofundar os seus conhecimentos.
- Ainda no site oficial da Oracle, procure por “Java Documentation”. Lá, você vai encontrar diversos exemplos de Java que vão ajudá-lo a se aprofundar mais nessa linguagem de programação que é bastante importante no mercado.

Referências

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

JAVA SE: Chapter 2 – The Structure of the Java Virtual Machine. Oracle America Inc, 2015. Consultado na internet em: 26 maio 2023.

JAVA Platform, Standard Edition & Java Development Kit Specifications – Version 20. Oracle America Inc. Consultado na internet em: 26 maio 2023.

SCHILDT, H. **Java**: The Complete Reference. Nova Iorque: McGraw Hill Education, 2014.

STROUSTRUP, B. **FAQ**. 2021. Consultado na internet em: 26 maio 2023.

SUTTER, H. **The C Family of Languages**: Interview with Dennis Ritchie, Bjarne Stroustrup, James Gosling. Consultado na internet em: 26 maio 2023.

THE APACHE FOUNDATION. **NetBeans IDE**: Overview. Consultado na internet em: 26 maio 2023.

THE JAVATM Tutorials. Lesson: Introduction to Collections. Oracle America Inc. Consultado na internet em: 26 maio 2023.