DESCRIÇÃO

Conceitos avançados de hierarquia e polimorfismo em linguagem Java e aplicação desses conceitos em interfaces e coleções. Os tipos estático e dinâmico de hierarquia e suas repercussões. Métodos importantes de objetos, implementação da igualdade, a conversão para o tipo *String* e o cálculo do *hash*, assim como o operador *instanceof*.

PROPÓSITO

Obter o conhecimento da linguagem Java, puramente orientada a objetos (OO) e que está entre as mais utilizadas no mundo, é imprescindível para o profissional de Tecnologia da Informação. A orientação a objetos (OO) é um paradigma consagrado no desenvolvimento de software e empregado independentemente da metodologia de projeto – ágil ou prescritiva.

PREPARAÇÃO

Para melhor absorção do conhecimento, recomenda-se o uso de computador com o *Java Development Kit* – JDK e um IDE (*Integrated Development Environment*) instalados e acesso à Internet para realização de pesquisa.

OBJETIVOS

MÓDULO 1

Identificar a hierarquia de herança em Java

MÓDULO 2

Empregar os principais métodos de objetos em Java

MÓDULO 3

Descrever polimorfismo em Java

MÓDULO 4

Aplicar a criação e o uso de interfaces em Java

INTRODUÇÃO

A orientação a objetos (OO) está no cerne da linguagem Java. A própria linguagem foi estruturada com foco no paradigma OO. Assim, Java está ligada de forma indissociável a esse paradigma. Para extrair todos os recursos que Java tem a oferecer, torna-se fundamental conhecer bem OO.

Na verdade, muito do poder da Java vem das capacidades que a OO trouxe para o desenvolvimento de softwares. Exemplos disso são, justamente, a herança e o polimorfismo.

Tais conceitos, próprios da OO e não de uma linguagem específica, foram empregados na concepção da Java, sendo responsáveis por muitas das funcionalidades oferecidas. Em Java, todas as classes descendem direta ou indiretamente da classe *Objects*.

Por esses motivos, vamos nos aprofundar no conhecimento de herança e polimorfismo na linguagem Java. Veremos como a Java emprega um conceito chamado Interfaces e estudaremos métodos de especial interesse para a manipulação de objetos.

Isso será feito com o apoio de atividades que irão reforçar o aprendizado e fornecer um *feedback*. Ao fim, teremos compreendido apenas uma parte do desenvolvimento em Java, mas que é indispensável para qualquer estudo mais avançado.

MÓDULO 1

Identificar a hierarquia de herança em Java

HERANÇA E A INSTANCIAÇÃO DE OBJETO EM JAVA

A palavra herança não é uma criação do paradigma de programação orientada a objetos (POO). Você deve estar familiarizado com o termo que, na área do Direito, designa a transmissão de bens e valores entre pessoas. Na Biologia, o termo significa a transmissão de características aos descendentes.

Independentemente da aplicação do seu conceito, a palavra herança tem em comum o entendimento de que envolve legar algo a alguém.

Em OO, herança é, também, a transmissão de características aos descendentes. Visto de outra forma, é a capacidade de uma “entidade” legar a outra seus métodos e atributos. Por legar, devemos entender que os métodos e atributos estarão presentes na classe derivada. Para melhor compreendermos, vamos entender a criação de um objeto em Java.

Quando definimos uma classe, definimos um mecanismo de criação de objetos.

Uma classe define um tipo de dado, e cada objeto instanciado pertence ao conjunto formado pelos objetos daquela classe.

Nesse conjunto, todos os objetos são do tipo da classe que lhes deu origem.

Uma classe possui métodos e atributos. Os métodos modelam o comportamento do objeto e atuam sobre o seu estado, que é definido pelos atributos. Quando um objeto é instanciado,

uma área de memória é reservada para acomodar os métodos e os atributos que o objeto deve possuir.

Todos os objetos do mesmo tipo terão os mesmos métodos e atributos, porém cada objeto terá sua própria cópia destes.

Consideremos, a título de ilustração, a definição de classe mostrada no Código 1.

CÓDIGO 1: DECLARAÇÃO DE CLASSE

```
1  //Pacotes
2  package com.mycompany.GereEscola;
3
4  //Classe
5  public class Pessoa {
6      //Atributos
7      protected String nome , nacionalidade , naturalidade;
8
9      //Métodos
10     public Pessoa ( String nome , String nacionalidade , String naturalidade
11         this.nome = nome;
12         this.nacionalidade = nacionalidade;
13         this.naturalidade = naturalidade;
14     }
15     protected void atualizarNome ( String nome ) {
16         this.nome = nome;
17     }
18     protected String recuperarNome ( ) {
19         return this.nome;
20     }
21     protected String recuperarNacionalidade ( ) {
22         return this.nacionalidade;
23     }
24     protected String recuperarNaturalidade ( ) {
25         return this.naturalidade;
26     }
27 }
```

Qualquer objeto instanciado a partir dessa classe terá os atributos “**nome**”, “**nacionalidade**” e “**naturalidade**”, além de uma cópia dos métodos mostrados. Na verdade, como “*String*” é um objeto de tamanho desconhecido em tempo de programação, as variáveis serão referências que vão guardar a localização em memória dos objetos do tipo “*String*”, quando esses forem criados.

Quando objetos se relacionam por associação, por exemplo, essa relação se dá de maneira horizontal. Normalmente são relações do tipo “contém” ou “possui”. A herança introduz um novo tipo de relacionamento, inserindo a ideia de “é tipo” ou “é um”. Esse relacionamento vertical origina o que se chama de “hierarquia de classes”.

Uma hierarquia de classe é um conjunto de classes que guardam entre si uma relação de herança (verticalizada), na qual as classes acima generalizam as classes abaixo ou, por simetria, as classes abaixo especializam as classes acima.

Vamos analisar a classe derivada mostrada no Código 2.

CÓDIGO 2: DECLARAÇÃO DE SUBCLASSE

```
1  //Pacotes
2  package com.mycompany.GereEscola;
3
4  //Classe
5  public class Aluno extends Pessoa {
6
7          public Aluno ( String nome , String nacionalidade , String naturalida
8              super ( nome , nacionalidade , String );
9          }
10 }
```

A classe “**Aluno**” estende a classe “**Pessoa**”, ou seja, “Aluno” é uma subclasse de “Pessoa”, formando assim uma relação hierárquica (“Aluno” deriva de “Pessoa”). Podemos aplicar a ideia introduzida com o conceito de herança para deixar mais claro o que estamos falando: “Aluno” **é um tipo de/é uma** “Pessoa”. Um objeto do tipo “Aluno” também é do tipo “Pessoa”.

Nesse sentido, como vimos, a classe “Pessoa” lega seus métodos e atributos à classe “Aluno”.

Verificando o Código 2, notamos que a classe “Aluno” não possui métodos e atributos definidos em sua declaração, exceto pelo seu construtor. Logo, ela terá apenas os componentes legados por “Pessoa”. A instanciação de um objeto do tipo “Aluno”, nesse caso, levará à reserva de um espaço de memória para guardar apenas os atributos “nome”, “nacionalidade” e “naturalidade”, e os métodos de “Pessoa” e o construtor de “Aluno”. No nosso exemplo, o objeto seria virtualmente igual a um objeto do tipo “Pessoa”.

Esse exemplo, porém, tem apenas um fim didático e busca mostrar na prática, ainda que superficialmente, o mecanismo de herança. Como os tipos do exemplo são praticamente idênticos, nenhuma utilidade prática se obtém - **“Aluno” não especializa “Pessoa”**.

Mas aí também atingimos outro fim: mostramos que a versatilidade da herança está na possibilidade de realizarmos especializações e generalizações no modelo a ser implementado. A especialização ocorre quando a classe derivada particulariza comportamentos. Assim, são casos em que as subclasses alteram métodos da superclasse.

No caso do Código 3, a classe “Aluno” definida possui um novo atributo – “**matrícula**” – que é gerado automaticamente quando o objeto for instanciado. Apesar de esse objeto possuir os métodos e atributos de “Pessoa”, ele agora tem um comportamento particular: na instanciação, além dos atributos definidos em “Pessoa”, ocorre também a definição de “matrícula”.

CÓDIGO 3: EXEMPLO DE ESPECIALIZAÇÃO

```
1  //Pacotes
2  package com.mycompany.GereEscola;
3
4  //Importações
5  import java.util.UUID;
6
7  //Classe
8  public class Aluno extends Pessoa {
9
10       //Atributos
11       private String matricula;
12
13       //Métodos
14       public Aluno ( String nome , String nacionalidade , String naturalidade )
15       super ( nome , nacionalidade , naturalidade );
16       matricula = UUID.randomUUID( ).toString( );
17       }
18 }
```

Vejamos a hierarquia de classes mostrada na Figura 1.

> Já sabemos que o atributo “**identificador**” e os métodos que operam sobre ele (“**atualizarID**” e “**recuperarID**”) são herdados pelas classes filhas.
>
> Também sabemos que um objeto da classe “**Fisica**” é também do tipo “Pessoa”.
>
> Mas vale destacar que a relação de herança se propaga indefinidamente, isto é, um objeto da classe “Aluno” também é um objeto das classes “Fisica” e “Pessoa”.

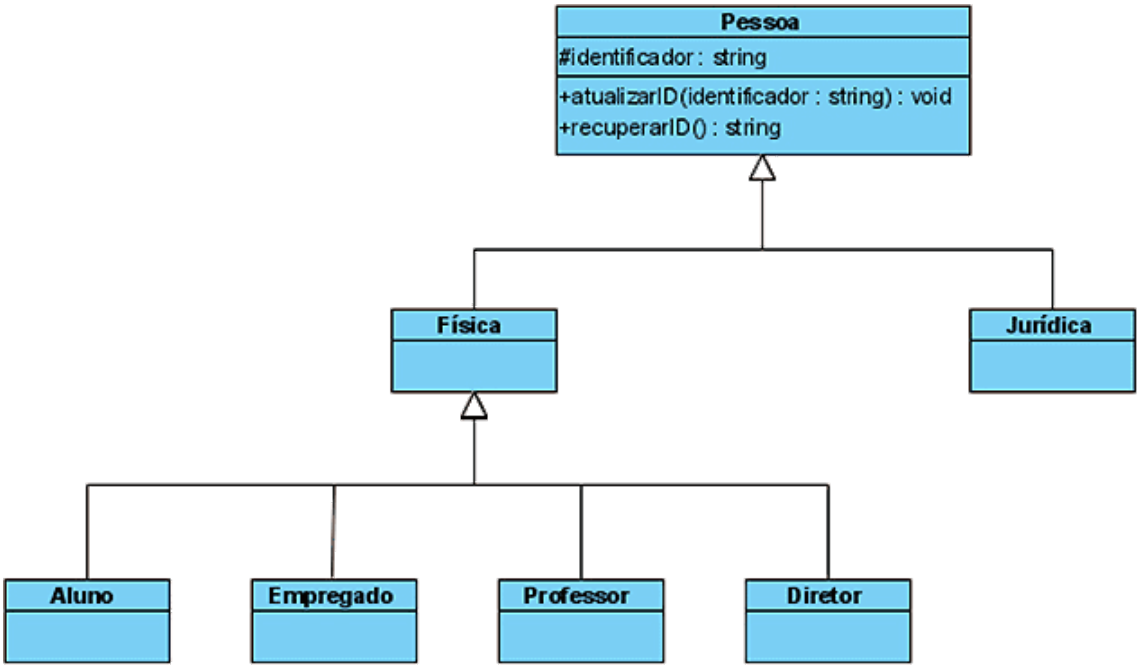


Figura 1: Especialização de comportamento.

A classe “Pessoa” implementa o comportamento mais genérico de todos. Ela possui um atributo de identificação definido como do tipo “*String*” (assim ele aceita letras, símbolos e números); e métodos que operam sobre esse atributo (veja o Código 4).

As classes “Fisica”, “Juridica” e “Aluno” especializam o método que define o atributo “Identificador”, particularizando-o para suas necessidades específicas, como observamos no Código 5, no Código 6 e no Código 7, que mostram as respectivas implementações.

CÓDIGO 4: CÓDIGO PARCIAL DA CLASSE "PESSOA" - MÉTODOS GENÉRICOS

```
1  //Classe
2  public class Pessoa {
3        ...
4        private String identificador;
5        ...
6        protected void atualizarID ( String identificador ) {
7             this.identificador = identificador;
8        }
9        protected String recuperarID ( ) {
10            return this.identificador;
11       }
12       ...
13 }
```

CÓDIGO 5: MÉTODO "ATUALIZARID" DA CLASSE “FISICA”

```
protected void atualizarID ( String CPF ) {
    if ( validaCPF ( CPF ) )
        this.identificador = CPF;
    else
        System.out.println ( "ERRO: CPF invalido!" );
}
```

CÓDIGO 6: MÉTODO "ATUALIZARID" DA CLASSE “JURIDICA”

```
protected void atualizarID ( String CNPJ ) {
    if ( validaCNPJ ( CNPJ ) )
        this.identificador = CNPJ;
    else
        System.out.println ( "ERRO: CNPJ invalido!" );
}
```

CÓDIGO 7: MÉTODO "ATUALIZARID" DA CLASSE "ALUNO"

```
public void atualizarID ( ) {
    if ( this.identificador.isBlank() )
        this.identificador = UUID.randomUUID( ).toString( );
    else
        System.out.println ( "ERRO: Codigo matricula ja existente!" );
}
```

📢 ATENÇÃO

Repare que cada código possui um comportamento específico. Apesar de não exibirmos a implementação do método “**validaCPF**” e “**validaCNPJ**”, fica claro que os comportamentos são distintos entre si e entre as demais implementações de “atualizarID”, pois o Cadastro de Pessoas Físicas (CPF) e o Cadastro Nacional de Pessoas Jurídicas (CNPJ) possuem regras de formação diferentes.

EXPLORANDO A HIERARQUIA DE HERANÇA

O exemplo anterior (Figura 1) pode ter despertado em você alguns questionamentos. Nele, “Aluno” está especializando o método “atualizarID”, que é da superclasse “Pessoa”.

Ao mesmo tempo, um objeto “Aluno” também é um tipo de “Fisica”, pois esta classe busca modelar o conceito de “pessoa física” (é razoável esperar que um aluno possua CPF além de seu código de matrícula). Será que essa situação não seria melhor modelada por meio de herança múltipla, com “Aluno” derivando diretamente de “Pessoa” e “Fisica”?

Já falamos que Java não permite herança múltipla e que é possível solucionar tais casos modificando a modelagem. Agora, entenderemos o problema da herança múltipla que faz a Java evitá-la. Para isso, vamos modificar o modelo mostrado na Figura 1, criando a classe “**ProfessorJuridico**”, que modela os professores contratados como pessoa jurídica.

Essa situação é vista na Figura 2. Nela, as classes “Fisica” e “Juridica” especializam o método “atualizarID” de “Pessoa” e legam sua versão especializada à classe “ProfessorJuridico”.

A questão que surge agora é: **quando “atualizarID” for invocado em um objeto do tipo “ProfessorJuridico”, qual das duas versões especializadas será executada?**

Essa é uma situação que não pode ser resolvida automaticamente pelo compilador. Por conta dessa ambiguidade, a Figura 2 é também chamada de “Diamante da morte”.

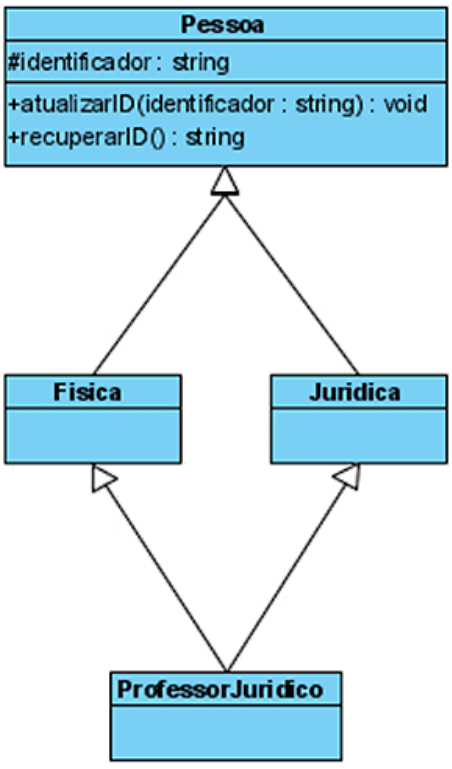


Figura 2: Diamante da morte.

Linguagens que admitem herança múltipla deixam para o desenvolvedor a solução desse problema.

Em C++, por exemplo, a desambiguação é feita fazendo-se um cast explícito para o tipo que se quer invocar.

Entretanto, deixar a cargo do programador é potencializar o risco de erro, o que Java tem como meta evitar.

Portanto, em Java, tal situação é proibida, obrigando a se criar uma solução de modelagem distinta.

HERANÇA, SUBTIPOS E O PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV

Nas seções anteriores, você compreendeu melhor o mecanismo de herança e como ele afeta os objetos instanciados. Falamos sobre especialização e generalização, mostrando como tais conceitos funcionam dentro de uma hierarquia de classes.

📢 ATENÇÃO

Se você estava atento, deve ter notado que o método “atualizarID” das classes “Pessoa”, “Fisica” e “Juridica” possuem a mesma assinatura. Isso está alinhado ao princípio de **Projeto por Contrato** (*Design by Contract*).

No Projeto por Contrato, os métodos de uma classe definem o contrato que se estabelece com a classe ao consumir os serviços que esta disponibiliza. Assim, para o caso em questão, o contrato “reza” que, para atualizar o campo “**identificador**” de um objeto, deve-se invocar o método “atualizarID”, passando-lhe um objeto do tipo “*String*” como parâmetro. E não se deve esperar qualquer retorno do método.

O método “atualizarID” da classe “Aluno” é diferente. Sua assinatura não admite parâmetros, o que altera o contrato estabelecido pela superclasse. Não há erro que seja conceitual ou formal, isto é, trata-se de uma situação perfeitamente válida na linguagem e dentro do paradigma OO, de maneira que “Aluno” é subclasse de “Fisica”. Porém, “Aluno” não define um subtipo de “Pessoa”, apesar de, corriqueiramente, essa distinção não ser considerada.

Rigorosamente falando, **subtipo e subclasse são conceitos distintos**.

Uma subclasse é estabelecida quando uma classe é derivada de outra.

Um subtipo tem uma restrição adicional.

Para que uma subclasse seja um subtipo da superclasse, faz-se necessário que todas as propriedades da superclasse sejam válidas na subclasse.

Isso não ocorre para o caso que acabamos de descrever. Nas classes “Pessoa”, “Fisica” e “Juridica”, a propriedade de definição do identificador é a mesma: o campo “identificador” é definido a partir de um objeto “*String*” fornecido. Na classe “Aluno”, o campo “identificador” é definido automaticamente.

A consequência imediata da mudança de propriedade feita pela classe “Aluno” é a modificação do contrato estabelecido pela superclasse. Com isso, um programador que siga o contrato da superclasse, ao usar a classe “Aluno”, terá problemas pela alteração no comportamento esperado.

Essa situação nos remete ao princípio da substituição de Liskov. Esse princípio foi primeiramente apresentado por Barbara Liskov, em 1987, e faz parte dos chamados princípios SOLID (*Single responsibility*, *Open-closed*, *Liskov substitution*, *Interface segregation and Dependency inversion*).

Ele pode ser encontrado com diversos enunciados, sempre afirmando a substitutibilidade de um objeto da classe base pela classe derivada, sem prejuízo para o funcionamento do *software*.

Podemos enunciá-lo da seguinte forma:

Seja um programa de computador P e os tipos *B* e *D*, tal que *D* deriva de *B*.

Se *D* for subtipo de *B*, então qualquer que seja o objeto *b* do tipo *B*, ele pode ser substituído por um objeto d do tipo *D* sem prejuízo para *P*.

Voltando ao exemplo em análise, as classes “Fisica” e “Juridica” estão em conformidade com o contrato da classe “Pessoa”, pois não o modificam, embora possam adicionar outros métodos.

Assim, em todo ponto do programa no qual um objeto de “Pessoa” for usado, objetos de “Fisica” e “Juridica” podem ser aplicados sem quebrar o programa, mas o mesmo não se dá com objetos do tipo “Aluno”.

HIERARQUIA DE COLEÇÃO

Coleções ou *containers* são um conjunto de classes e interfaces Java chamados de *Java Collections Framework*. Essas classes e interfaces implementam estruturas de dados comumente utilizadas para agrupar múltiplos elementos em uma única unidade. Sua finalidade é armazenar, manipular e comunicar dados agregados.

Entre as estruturas de dados implementadas, temos:

Set

List

Queue

Deque

Curiosamente, apesar de a estrutura “Map” não ser, de fato, uma coleção, ela também é provida pela Java Collections Framework. Isso significa que o framework é formado por duas árvores de hierarquia, uma correspondente às entidades derivadas de “Collection” (Figura 3) e a outra, formada pela “Map” (Figura 4).

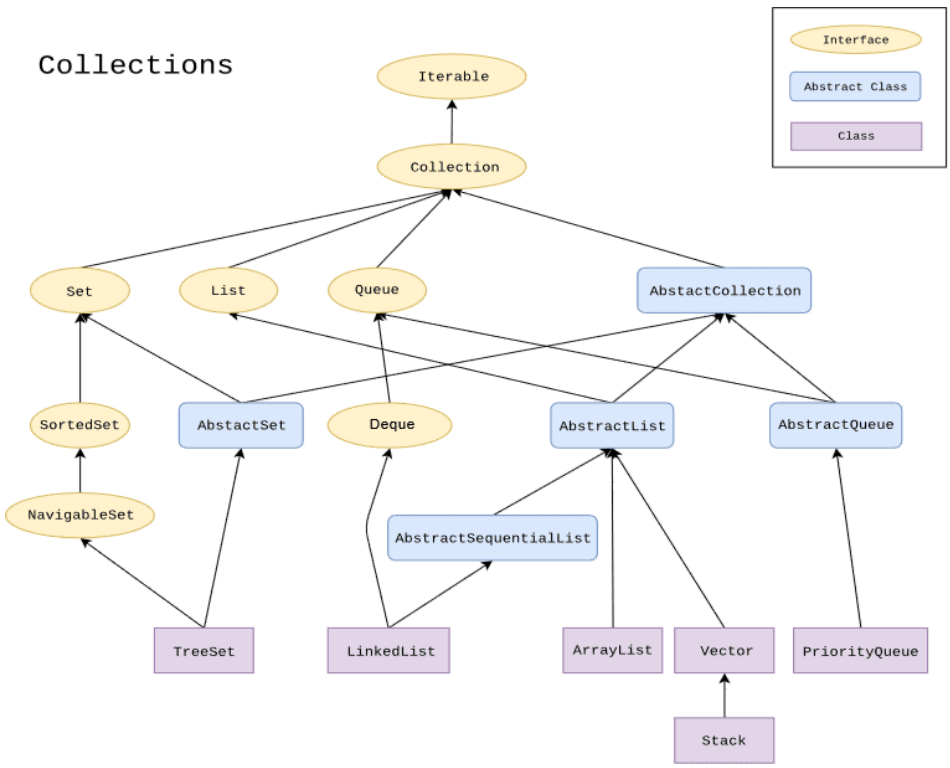
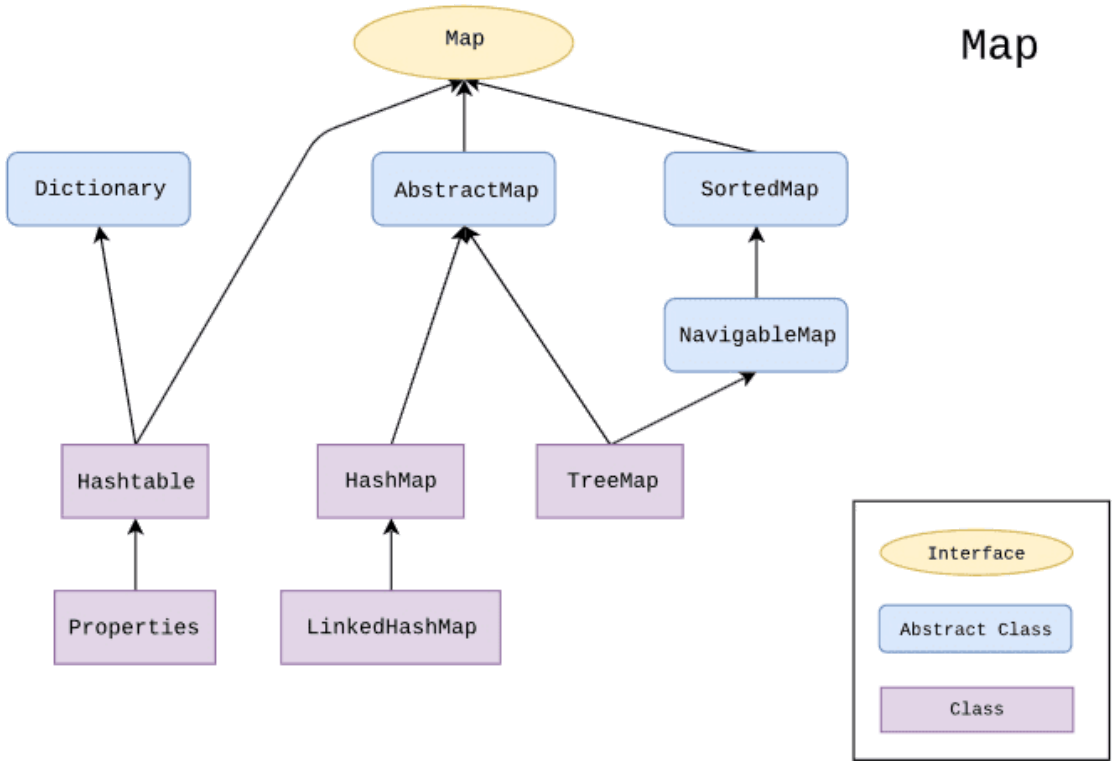


Figura 3: Hierarquia de herança – Collections.



📷 Figura 4: Hierarquia de herança - Map.

Como você pode observar pelas figuras que representam ambas as hierarquias de herança, essas estruturas são implementadas aplicando-se os conceitos de herança vistos anteriormente.

⭐ EXEMPLO

"*SortedSet*" e "*SortedMap*" são especializações de "*Set*" e "*Map*", respectivamente, que incorporam a ordenação de seus elementos.

A interface "*Collection*" possui os comportamentos mais genéricos (assim como "*Map*", na sua hierarquia). Por exemplo, o mecanismo para iterar pela coleção pode ser encontrado nela. Ela possui, também, métodos que fornecem comportamentos genéricos. É um total de 15 métodos, mostrados no quadro a seguir.

Método	Comportamento
add(Object e)	Insere um elemento na coleção.
addAll(Collection c)	Adiciona os elementos de uma coleção em outra.

clear()	Limpa ou remove os elementos de uma coleção.
contains(Object c)	Verifica se dado elemento está presente na coleção.
containsAll(Collection c)	Verifica se todos os elementos de uma coleção estão presentes em outra.
equals(Object e)	Verifica se dois objetos são iguais.
hashCode()	Retorna o hash de uma coleção.
isEmpty()	Retorna verdadeiro se a coleção estiver vazia.
Iterator()	Retorna um iterador.
remove(Object e)	Remove determinado elemento da coleção.
removeAll(Collection c)	Remove todos os elementos da coleção.
retainAll(Collection c)	Remove todos os elementos de uma coleção exceto os que correspondem a “c”.
size()	Retorna o número de elementos da coleção.
toArray()	Retorna os elementos de uma coleção em um vetor.
Objetct [ ] toArray (Object e)	Retorna um vetor que contém todos os elementos da coleção que o invoca.

Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal

Quadro: Métodos da interface Collection. Elaborado por Marlos de Mendonça.

As interfaces “Set”, “List”, “Queue” e “Deque” possuem comportamentos especializados, conforme as seguintes particularidades de cada tipo:

SET

Representa conjuntos e não admite elementos duplicados.

LIST

Implementa o conceito de listas e admite duplicidade de elementos. É uma coleção ordenada que permite o acesso direto ao elemento armazenado, assim como a definição da posição onde armazená-lo.

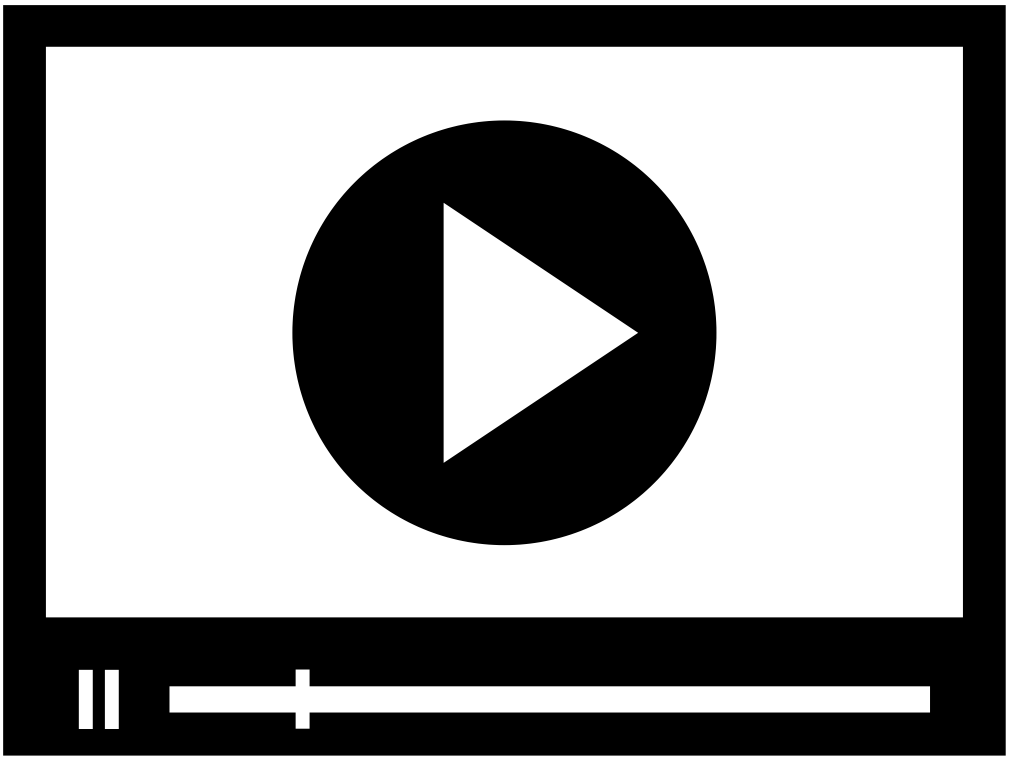
QUEUE

Trata-se de uma coleção que implementa algo mais genérico do que uma fila. 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.

Nas hierarquias de coleções mostradas, as interfaces e as classes abstratas definem o contrato que deve ser seguido pelas classes que lhes são derivadas. Essa abordagem atende ao **Princípio de Substituição de Liskov**, garantindo que as derivações constituam subtipos. As classes que implementam os comportamentos desejados são vistas em roxo nas figuras.



HERANÇA EM JAVA

No vídeo a seguir, será apresentado o conceito de herança na linguagem de programação Java.

TIPOS ESTÁTICOS E DINÂMICOS

Esse é um assunto propenso à confusão. Basta uma rápida busca pela Internet para encontrar o emprego equivocado desses conceitos. Muitos associam a vinculação dinâmica na invocação de métodos dentro de hierarquias de classe à tipagem dinâmica. Não é. Trata-se, nesse caso, de vinculação dinâmica (*dynamic binding*).

Nesta seção, vamos elucidar essa questão.

Primeiramente, precisamos entender o que é tipagem dinâmica e estática.

Tipagem é a atribuição a uma variável de um tipo.

Um tipo de dado é um conjunto fechado de elementos e propriedades que afeta os elementos. Por exemplo, quando definimos uma variável como sendo do tipo inteiro, o compilador gera comandos para alocação de memória suficiente para guardar o maior valor possível para esse tipo.

Você se lembra de que o tipo inteiro em Java (e em qualquer linguagem) é uma abstração limitada do conjunto dos números inteiros, que é infinito. Além disso, o compilador é capaz de operar com os elementos do tipo "***int***". Ou seja, adições, subtrações e outras operações são realizadas sem a necessidade de o comportamento ser implementado. O mesmo vale para os demais tipos primitivos, como "***String***", "***float***" ou outros.

Há duas formas de se tipar uma variável: estática ou dinâmica.

A tipagem estática ocorre quando o tipo é determinado em tempo de compilação.

A tipagem dinâmica é determinada em tempo de execução.

Observe que não importa se o programador determinou o tipo. Se o compilador for capaz de inferir esse tipo durante a compilação, então, trata-se de tipagem estática.

📢 ATENÇÃO

A tipagem dinâmica ocorre quando o tipo só pode ser identificado durante a execução do programa.

Um exemplo de linguagem que emprega tipagem dinâmica é a JavaScript. O Código 8 mostra um trecho de programa em JavaScript que exemplifica a tipagem dinâmica.

CÓDIGO 8: TIPAGEM DINÂMICA EM JAVASCRIPT

```
1  var a;
2  console.log('O tipo de a eh: '+ typeof(a));
3  a = 1;
4  console.log('O tipo de a eh: '+ typeof(a));
```

```
5  a = "Palavra";
6  console.log('O tipo de a eh: '+ typeof(a));
7  a = false;
8  console.log('O tipo de a eh: '+ typeof(a));
9  a = [];
10 console.log('O tipo de a eh: '+ typeof(a));
```

A execução desse código tem como saída:

```
1 O tipo de a eh: undefined
2 O tipo de a eh: number
3 O tipo de a eh: string
4 O tipo de a eh: boolean
5 O tipo de a eh: object
```

Podemos ver que a mesma variável (“a”) assumiu, ao longo da execução, diferentes tipos. Observando a linha 1 do Código 8, podemos ver que, na declaração, o compilador não tinha qualquer informação que lhe permitisse inferir o tipo de dado de “a”. Esse tipo só pôde ser determinado na execução, nas linhas 3, 5, 7 e 9. Todas essas linhas alteram sucessivamente o tipo atribuído a “a”.

Você deve ter claro em sua mente que a **linguagem de programação Java é estaticamente tipada**, o que significa que todas as variáveis devem ser primeiro declaradas (tipo e nome) e depois utilizadas [2]. Uma das possíveis fontes de confusão é a introdução da instrução “var” a partir da versão 10 da Java [3]. Esse comando, aplicável apenas às variáveis locais, instrui o compilador a inferir o tipo de variável. Todavia, o tipo é determinado ainda em tempo de compilação e, uma vez determinado, o mesmo não pode ser modificado. A Java exige que, no uso de “var”, a variável seja inicializada, o que permite ao compilador determinar o seu tipo. O Código 9 mostra o emprego correto de “var”, enquanto o Código 10 gera erro de compilação. Ou seja, “var”, em Java, não possui a mesma semântica que “var” em JavaScript. São situações distintas e, em Java, trata-se de tipagem estática.

CÓDIGO 9: USO CORRETO DE "VAR" EM JAVA - TIPAGEM ESTÁTICA

```
1 ...
2 var a = 0;
3 a = a + 1;
4 ...
```

CÓDIGO 10: USO ERRADO DE "VAR" EM JAVA

```
...
var a = 0;
a = “Palavra”;
...
```

Uma outra possível fonte de confusão diz respeito ao *dynamic binding* ou vinculação dinâmica de método. Para ilustrar essa situação, vamos adicionar o método “retornaTipo” às classes “Pessoa”, “Fisica” e “Juridica”, cujas respectivas implementações são vistas no Código 11, no Código 12 e no Código 13.

CÓDIGO 11: "RETORNATIPO" DE "PESSOA"

```
public String retornaTipo ( ) {
    return null;
}
```

CÓDIGO 12: "RETORNATIPO" DE “FISICA”

```
public String retornaTipo ( ) {
    return “Fisica”;
}
```

CÓDIGO 13: "RETORNATIPO" DE “JURIDICA”

```
public String retornaTipo ( ) {
    return “Juridica”;
}
```

Se lembrarmos que as classes “Fisica” e “Juridica” especializam “Pessoa”, ficará óbvio que ambas substituem o método “retornaTipo” de “Pessoa” por versões especializadas. O Código 14 mostra um trecho de código da classe principal do programa.

CÓDIGO 14: CLASSE PRINCIPAL - EXEMPLO DE VINCULAÇÃO DINÂMICA

```

public class Principal {
    private static Pessoa grupo [];
    ...
    grupo = new Pessoa [3];
    grupo [0] = new Fisica ( "Marco Antônio" , data_nasc , null , null , "Brasil"
    grupo [1] = new Juridica ("Escola Novo Mundo Ltda" , data_nasc , null , nu
    grupo [2] = new Pessoa ( null , null , null , null , "Brasil" , "Rio de Janeiro'

```

```
9       for ( int i = 0 ; i <= 2 ; i++ )
10          System.out.println( grupo [i].retornaTipo ( ) );
11  ...
    }
```

A execução desse código produz como saída:

```
1  Física
2  Jurídica
3  null
```

O motivo disso é a vinculação dinâmica de método. O vetor “grupo [ ]” é um arranjo de

Como “Fisica” e “Juridica” são subclasses de “Pessoa”, os objetos dessas classes podem ser referenciados por uma variável que guarda referência para a superclasse. Isso é o que ocorre nas linhas 5 e 6 do Código 14.

A vinculação dinâmica ocorre na execução da linha 9. O compilador não sabe, *a priori*, qual objeto será referenciado nas posições do vetor, mas, durante a execução, ele identifica o tipo do objeto e vincula o método apropriado. Essa situação, entretanto, não se configura como tipagem dinâmica, pois o tipo do objeto é usado para se determinar o método a ser invocado.

Nesse caso, devemos lembrar, também, que Java oculta o mecanismo de ponteiros, e uma classe definida pelo programador não é um dos tipos primitivos. Então “grupo []” é, como dito, um vetor de referências para objetos do tipo “Pessoa”, e isso se mantém mesmo nas linhas 5 e 6 (as subclasses são um tipo da superclasse). Essa é uma diferença sutil, mas significativa, com relação ao mostrado no Código 8.

Iremos explorar esse mecanismo em outra oportunidade. No entanto, é preciso ficar claro que essa situação não é tipagem dinâmica, pois o vetor “grupo []” não alterou seu tipo, qual seja, referência para objetos do tipo “Pessoa”.

VERIFICANDO O APRENDIZADO

MÓDULO 2

O MÉTODO "*TOSTRING*"

Já vimos que em Java todas as classes descendem direta ou indiretamente da classe "*Object*". Essa classe provê um conjunto de métodos que, pela herança, estão disponíveis para as subclasses. Tais métodos oferecem ao programador a facilidade de contar com comportamentos comuns fornecidos pela própria biblioteca da linguagem, além de poder especializar esses métodos para atenderem às necessidades particulares de sua implementação.

Neste módulo, abordaremos alguns dos métodos da classe "***Object***" – "***toString***", "***equals***" e "***hashCode***" – e aproveitaremos para discutir as nuances do acesso protegido e do operador "***Instanceof***".

O método "*toString*" permite identificar o objeto retornando uma representação "*String*" do próprio. Ou seja, trata-se de uma representação textual que assume a forma:

<nome completamente qualificado da classe à qual o objeto pertence>@<código hash do objeto>.

O *<nome completamente qualificado da classe à qual o objeto pertence>* é a qualificação completa da classe, incluindo o pacote ao qual ela pertence. A qualificação é seguida do símbolo de "@" e, depois dele, do código *hash* do objeto.

Veja um exemplo de saída do método "*toString*":

```
1 com.mycompany.GereEscola.Juridica@72ea2f77
```

Nessa saída, notamos o pacote (com.mycompany.GereEscola), a classe (Jurídica) e o *hash* do objeto (72ea2f77) formando o texto representativo daquela instância.

A implementação do método "*toString*" é mostrada no Código 15. Nela, identificamos o uso de outros métodos da classe "*Object*", como "*getClass*" e "*hashCode*". Não vamos nos aprofundar no primeiro, mas convém dizer que ele retorna um objeto do tipo "*Class*" que é a representação da classe em tempo de execução do objeto. O segundo será visto mais tarde neste módulo.

CÓDIGO 15: MÉTODO "*TOSTRING*"

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Essa, porém, é a implementação padrão fornecida pela biblioteca Java, que pode ser prontamente usada pelo programador. Todavia, prover uma especialização é algo interessante, pois permite dar um formato mais útil à saída do método.

Nesse caso, os mecanismos de polimorfismo e herança permitirão que o método mais especializado seja empregado. Remetendo à saída anteriormente mostrada, vemos que há pouca informação, prejudicando sua utilidade.

Além de saber o nome completamente qualificado da classe, temos somente o seu código *hash*. Assim, vamos modificar o método "*toString*" para que ele apresente outras informações, conforme a linha 18 da implementação da classe "Pessoa" parcialmente mostrada no Código 16.

A classe "Pessoa" tem "Fisica" como subclasse. A implementação de "Fisica" é mostrada no Código 17.

CÓDIGO 16: CLASSE "PESSOA" - CÓDIGO PARCIAL COM MÉTODO "*TOSTRING*" ESPECIALIZADO

```

public class Pessoa {
    //Atributos
    protected String nome , naturalidade , nacionalidade , identificador;
    private Calendar data_inicio_existencia;
    private int idade;
    private Endereco endereco;

    //Métodos
    public Pessoa ( String nome , Calendar data_inicio_existencia, String id
        this.nome = nome;
        this.data_inicio_existencia = data_inicio_existencia;
        this.identificador = identificador;
        this.endereco = endereco;
        this.nacionalidade = nacionalidade;
        this.naturalidade = naturalidade;
    }
...
public String toString (){
    return "Objeto:" + "\n\t- Classe: " + getClass().getName() + "\n\t- Hash: " + Int

```

```
        }
    }
```

CODIGO 17: CLASSE "FISICA" (DERIVADA DE "PESSOA")

```
1  public class Fisica extends Pessoa {
2      //Atributos
3
4      //Métodos
5      public Fisica ( String nome , Calendar data_nascimento , String CPF , Ende
6          super ( nome , data_nascimento, CPF , endereco , nacionalidade , r
7      atualizarIdade ();
8      }
9  }
```

O Código 18 faz a invocação do método "*toString*" para dois objetos criados, um do tipo "Pessoa" e outro do tipo "Fisica".

CÓDIGO 18: INVOCAÇÃO DA VERSÃO ESPECIALIZADA DE "*TOSTRING*"

```
1  public class Principal {
2  private static Pessoa grupo [];
3  public static void main (String args[]) {
4      grupo = new Pessoa [2];
5      grupo [0] = new Fisica ( "Marco Antônio" , data_nasc , "365.586.875-45", r
6      grupo [1] = new Pessoa ("Escola Novo Mundo Ltda" , data_nasc , "43.186.6
7      for ( int i = 0 ; i <= 1 ; i++ )
8          System.out.println( "grupo[" + i + "]: " + grupo[i].toString() );
9  }
```

A execução do Código 18 tem como saída:

```
1  grupo[0]: Objeto:
2      - Classe: com.mycompany.GereEscola.Fisica
3      - Hash: 77459877
4      - Nome: Marco Antônio
5      - Identificador: 365.586.875-45
6  grupo[1]: Objeto:
7      - Classe: com.mycompany.GereEscola.Pessoa
8      - Hash: 378bf509
9
```

```
10          - Nome: Escola Novo Mundo Ltda
            - Identificador: 43.186.666/0026-32
```

Veja que o método “*toString*” agora apresenta outra composição de saída, trazendo informações como o nome e o identificador. Além disso, formatamos a saída para apresentar as informações em múltiplas linhas identadas.

Se olharmos novamente o Código 18, veremos que, na linha 5, o objeto criado é do tipo “Fisica”. Apesar de “Fisica” não reescrever o método “*toString*”, ela herda a versão especializada de sua superclasse, que é invocada no lugar da versão padrão. Olhando também para a saída, percebemos que o objeto foi corretamente identificado como sendo da classe “Fisica”.

Ao modificar a implementação do método, tornamos seu retorno mais útil. Por esse motivo, ele é frequentemente sobrescrito nas diversas classes providas pela biblioteca Java.

OS MÉTODOS “*EQUALS*” E “*HASHCODE*”

Outros dois métodos herdados da classe “*Object*” são “*equals*” e “*hashCode*”. Da mesma maneira que o método “*toString*”, eles têm uma implementação provida por “*Object*”. Mas também são métodos que, usualmente, iremos especializar, a fim de que tenham semântica útil para as classes criadas.

O método “*equals*” é utilizado para avaliar se outro objeto é igual ao objeto que invoca o método. Se forem iguais, ele retorna “*true*”; caso contrário, ele retorna “*false*”. A sua assinatura é “*boolean equals (Object obj)*”, e sua implementação é mostrada no Código 19. Já que ele recebe como parâmetro uma referência para um objeto da classe “*Object*”, da qual todas as classes descendem em Java, ele aceita referência para qualquer objeto das classes derivadas.

CÓDIGO 19: MÉTODO "*EQUALS*" DA CLASSE "*OBJECT*"

```
1  public boolean equals(Object obj) {
2          return (this == obj);
3  }
```

A implementação de “*equals*” busca ser o mais precisa possível em termos de comparação, retornando verdadeiro (“*true*”), e somente se os dois objetos comparados são o mesmo objeto. Ou seja, “*equals*” implementa uma relação de equivalência, referências não nulas de objetos tal que as seguintes propriedades são válidas:

Reflexividade: qualquer que seja uma referência não nula R, *R.equals*(R) retorna sempre "*true*".

Simetria: quaisquer que sejam as referências não nulas R e S, *R.equals*(S) retorna "*true*" se, e somente se, *S.equals*(R) retorna "*true*".

Transitividade: quaisquer que sejam as referências não nulas R, S e T, se *R.equals*(S) retorna "*true*" e *S.equals*(T) retorna "*true*" então *R.equals*(T) deve retornar "*true*".

Consistência: quaisquer que sejam as referências não nulas R e S, múltiplas invocações de *R.equals*(S) devem consistentemente retornar "*true*" ou consistentemente retornar "*false*", dado que nenhuma informação do objeto usada pela comparação em "*equals*" tenha mudado.

📢 ATENÇÃO

Um caso excepcional do *equals* é a seguinte propriedade: qualquer que seja uma referência não nula R, *R.equals(null)* retorna sempre "*false*".

Dissemos que usualmente a implementação de "*equals*" é sobrescrita na classe derivada. Quando isso ocorre, é dever do programador garantir que essa relação de equivalência seja mantida. Obviamente, nada impede que ela seja quebrada, dando uma nova semântica ao método, mas se isso ocorrer, então deve ser o comportamento desejado e não por consequência de erro. Garantir a equivalência pode ser relativamente trivial para tipos de dados simples, mas se torna elaborado quando se trata de classes.

A seguir, veja uma aplicação simples de "*equals*". Para isso, vamos trabalhar com 9 objetos, dos quais 3 são do tipo "*int*" e 3, do tipo "*String*" – ambos tipos primitivos; e 3 são referências para objetos das classes "Pessoa" e "Fisica", conforme consta no Código 20.

CÓDIGO 20: EXEMPLO DE USO DE "*EQUALS*"

```
1
2  public class Principal {
3      //Atributos
4      private static int I1 , I2 , I3;
5      private static String S1 , S2 , S3;
6      private static Fisica p1 , p2;
7      private static Pessoa p3;
8
9      //Métodos
10     public static void main (String args[]) {
11         I1 = 1;
```

```
12            I2 = 2;
13            I3 = 1;
14            S1 = "a";
15            S2 = "b";
16            S3 = "a";
17            Calendar data_nasc = Calendar.getInstance();
18            data_nasc.set(1980, 10, 23);
19            p1 = new Fisica ( "Marco Antônio" , data_nasc , "365.586.875-45" , nu
20            p2 = new Fisica ( "Marco Antônio" , data_nasc , "365.586.875-45" , nu
21            p3 = new Pessoa ( "Classe Pessoa" , null , null , null , "Brasil" , "Rio
22            compararEquals ( p1 , p2 , p3 );
23            compararEquals ( I1, I2, I3 );
24            compararEquals ( S1, S2, S3 );
25        }
26
27        private static void compararEquals ( Object o1 , Object o2 , Object o3 ){
28            System.out.println("Uso de EQUALS para comparar " + o1.getClass().getNa
29                if ( o1.equals( o2 ) )
30                    System.out.println("o1 == o2");
31                else
32                    System.out.println("o1 != o2");
33                if ( o1.equals(o3) )
34                    System.out.println("o1 == o3");
35                else
36                    System.out.println("o1 != o3");
37        }
    }
```

Nas linhas 21, 22 e 23 do Código 20, o que fazemos é comparar os diversos tipos de objeto utilizando o método “*equals*”. O resultado é mostrado a seguir:

```
1  Uso de EQUALS para comparar com.mycompany.GereEscola.Fisica
2  o1 != o2
3  o1 != o3
4  Uso de EQUALS para comparar java.lang.Integer
5  o1 != o2
6  o1 == o3
7  Uso de EQUALS para comparar java.lang.String
8  o1 != o2
9  o1 == o3
```

Podemos ver que o objeto “I1” foi considerado igual ao “I3”, assim como “S1” e “S3”. Todavia, “p1” e “p2” foram considerados diferentes, embora sejam instâncias da mesma classe (“Fisica”), e seus atributos, rigorosamente iguais. Por quê?

A resposta dessa pergunta mobiliza vários conceitos, a começar pelo entendimento do que ocorre nas linhas 5 e 6. As variáveis “p1”, “p2” e “p3” são referências para objetos das classes “Fisica” e “Pessoa”.

Em contrapartida, as variáveis “I1”, “I2”, “I3”, “S1”, “S2” e “S3” são todas de tipos primitivos. O operador de comparação “==” atua verificando o conteúdo das variáveis que, para os tipos primitivos, são os valores neles acumulados. Mesmo o tipo “*String*”, que é um objeto, tem seu valor acumulado avaliado. O mesmo ocorre para os tipos de dados de usuário (classes), só que nesse caso “p1”, “p2” e “p3” armazenam o endereço de memória (referência) onde os objetos estão. Como esses objetos ocupam diferentes endereços de memória, a comparação entre “p1” e “p2” retorna “false” (são objetos iguais, mas não são o mesmo objeto). Uma comparação “p1.equals(p1)” retornaria “*true*” obviamente, pois, nesse caso, trata-se do mesmo objeto.

A resposta anterior também explica por que precisamos sobrescrever “*equals*” se quisermos comparar objetos. No Código 21, mostramos a reimplementação desse método para a classe “Pessoa”.

CÓDIGO 21: MÉTODO "*EQUALS*" ESPECIALIZADO

```
1  public boolean equals (Object obj) {
2      if ( ( nome.equals( ( ( Pessoa ) obj ).nome ) ) && ( this instanceof Pessoa
3          return true;
4      else
5          return false;
6  }
```

Com essa especialização, a saída agora é:

```
1  Uso de EQUALS para comparar com.mycompany.GereEscola.Fisica
2  o1 == o2
3  o1 != o3
```

Contudo, da forma que implementamos “*equals*”, “p1” e “p2” serão considerados iguais mesmo que os demais atributos sejam diferentes. Esse caso mostra a complexidade que mencionamos anteriormente, em se estabelecer a relação de equivalência entre objetos complexos. Cabe ao programador determinar quais características devem ser consideradas na comparação.

A reimplementação de “*equals*” impacta indiretamente o método “*hash*”. A própria documentação de “*equals*” aponta isso quando diz “que geralmente é necessário substituir o método “*hashCode*” sempre que esse método (“*equals*”) for substituído, de modo a manter o

contrato geral para o método “*hashCode*”, que afirma que objetos iguais devem ter códigos *hash* iguais”.

Isso faz todo sentido, já que um código hash é uma impressão digital de uma entidade. Pense no caso de arquivos. Quando baixamos um arquivo da Internet, como uma imagem de um sistema operacional, usualmente calculamos o hash para verificar se houve erro no download. Isto é, mesmo sendo duas cópias distintas, esperamos que, se forem iguais, os arquivos tenham o mesmo *hash*. Esse princípio é o mesmo por trás da necessidade de se reimplementar “*hashCode*”.

Nós já vimos o uso desse método quando estudamos “*toString*”, e a saída do Código 18 mostra o resultado do seu emprego. A sua assinatura é “*int hashCode()*”, e ele tem como retorno o código *hash* do objeto.

Esse método possui as seguintes propriedades:

Invocações sucessivas sobre o mesmo objeto devem consistentemente retornar o mesmo valor, dado que nenhuma informação do objeto usada pela comparação em “*equals*” tenha mudado.

Se dois objetos são iguais segundo “*equals*”, então a invocação de “*hashCode*” deve retornar o mesmo valor para ambos.

Caso dois objetos sejam desiguais segundo “*equals*”, não é obrigatório que a invocação de “*hashCode*” em cada um dos dois objetos produza resultados distintos. Mas produzir resultados distintos para objetos desiguais pode melhorar o desempenho das tabelas *hash*.

A invocação de “*hashCode*” nos objetos “p1” e “p2” do Código 20 nos dá a seguinte saída:

```
p1 = 2001049719
p2 = 250421012
```

Esse resultado contraria as propriedades de “*hashCode*”, uma vez que nossa nova implementação de “*equals*” estabeleceu a igualdade entre “p1” e “p2” instanciados no Código 18. Para restaurar o comportamento correto, fornecemos a especialização de “*hashCode*” vista no Código 22.

CÓDIGO 22: REIMPLEMENTAÇÃO DE "*HASHCODE*"

```
public int hashCode () {
      if ( this instanceof Pessoa )
            return this.nome.hashCode();
      else

```

```
6           return super.hashCode();
    }
```

Veja que agora nossa implementação está consistente com a de “*equals*”. Na linha 2 do Código 22, asseguramo-nos de que se trata de um objeto da classe “Pessoa”. Se não for, chamamos a implementação padrão de “*hashCode*”. Caso seja, retornamos o valor de hash da “*String*” que forma o atributo “nome”, o mesmo usado em “*equals*” para comparação. A saída é:

```
1  p1 = -456782095
2  p2 = -456782095
```

Agora, conforme esperado, “p1” e “p2” possuem os mesmos valores.

📢 COMENTÁRIO

Em ambos os casos, oferecemos uma reimplementação simples para corrigir o comportamento. Mas você deve estar preparado para necessidades bem mais complexas. O programador precisa, no fundo, compreender o comportamento modelado e as propriedades desses métodos, para que sua modificação seja consistente.

O OPERADOR “*INSTANCEOF*”

O operador “*instanceof*” é utilizado para comparar um objeto com um tipo específico. É o único operador de comparação de tipo fornecido pela linguagem.

Sua sintaxe é simples: “op1 *instanceof* op2”, onde o operando “op2” é o tipo com o qual se deseja comparar; e “op1” é o objeto ou a expressão a ser comparada. Um exemplo do uso desse operador é mostrado na linha 2 do Código 22. O operador retorna verdadeiro, se “op1” for uma instância do tipo “op2”. Então, devemos esperar que se “op1” for uma subclasse de “op2”, o retorno será “*true*”. Vejamos se isso é válido executando o Código 23.

CÓDIGO 23: OPERADOR "*INSTANCEOF*"

```
1
2  public class Principal {
3  //Atributos
4  private static Pessoa p1, p3;
```

```
5  private static Fisica p2;
6  //Métodos
7  public static void main (String args[]) {
8       Calendar data_nasc = Calendar.getInstance();
9       data_nasc.set(1980, 10, 23);
10      p1 = new Fisica ( "Marco Antônio" , data_nasc , "365.586.875-45" , null , "
11      p2 = new Fisica ( "Marco Antônio" , data_nasc , "365.586.875-45" , null , "
12      p3 = new Pessoa ( "Classe Pessoa" , null , null , null , "Brasil" , "Rio de Ja
13      if ( p1 instanceof Pessoa )
14           System.out.println("p1 é instância do tipo Pessoa.");
15      else
16           System.out.println("p1 não é instância do tipo Pessoa.");
17      if ( p2 instanceof Pessoa )
18      System.out.println("p2 é instância do tipo Pessoa.");
19      else
20           System.out.println("p2 não é instância do tipo Pessoa.");
21      if ( p3 instanceof Pessoa )
22           System.out.println("p3 é instância do tipo Pessoa.");
23      else
24           System.out.println("p3 não é instância do tipo Pessoa.");
25      if ( p3 instanceof Fisica )
26           System.out.println("p3 é instância do tipo Física.");
27      else
28           System.out.println("p3 não é instância do tipo Física.");
29           }
          }
```

Nesse código, fazemos quatro comparações, cujos resultados são:

```
1  p1 é instância do tipo Pessoa.
2  p2 é instância do tipo Pessoa.
3  p3 é instância do tipo Pessoa.
4  p3 não é instância do tipo Física.
```

Assim, o código valida nossa expectativa: "*instanceof*" retorna verdadeiro para uma instância da subclasse quando comparada ao tipo da superclasse. Tal resultado independe se a variável é uma referência para a superclasse (linha 9) ou a própria subclasse (linha 10).

ENTENDENDO O ACESSO PROTEGIDO

Já abordamos o acesso protegido em outras partes. De maneira geral, ele restringe o acesso aos atributos de uma classe ao pacote dessa classe e às suas subclasses. Portanto, uma subclasse acessará todos os métodos e atributos públicos ou protegidos da superclasse, mesmo se pertencerem a pacotes distintos. Claro que se forem pacotes diferentes, a subclasse deverá importar primeiro a superclasse.

Agora vamos explorar algumas situações particulares. Para isso, vamos criar o pacote “Matemática" e, doravante, as classes que temos utilizado (“Principal”, “Pessoa”, “Fisica”, “Juridica”, “Aluno” etc.) estarão inseridas no pacote “GereEscola”.

No pacote “Matemática", vamos criar a classe “Nota”, que implementa alguns métodos de cálculos relativos às notas dos alunos. Essa classe pode ser parcialmente vista no Código 24.

CÓDIGO 24: CÓDIGO PARCIAL DA CLASSE "NOTA"

```
1  package com.mycompany.Matematica;
2  ...
3  public class Nota {
4  ...
5      public float calcularMedia () {
6  ...
7      }
8      protected float calcularCoeficienteRendimento () {
9  ...
10     }
11 ...
12 }
```

No pacote “GereEscola”, vamos criar a classe “Desempenho”, conforme mostrado, parcialmente, no Código 25.

CÓDIGO 25: CLASSE "DESEMPENHO"

```
1  //Pacote
2  package com.mycompany.GereEscola;
3
4  //Importações
5  import com.mycompany.Matematica.Nota;
6
7  //Classe
8  public class Desempenho extends Nota {
9      //Atributos
10     private float media , CR;
11     private Nota nota;
```

```
12
13        //Métodos
14        public Desempenho () {
15              nota = new Nota ();
16              media = calcularMedia();
17              CR = calcularCoeficienteRendimento();
18              //media = nota.calculaMedia();
19              //CR = nota.caulculaCoeficienteRendimento();
20        }
21  }
```

A classe “Desempenho” é filha da classe “Nota”. Por isso, ela tem acesso aos métodos “calcularMedia” (público) e “calcularCoeficienteRendimento” (protegido) de “Nota” mesmo estando em outro pacote.

Observando a linha 11, vemos que ela também possui um objeto do tipo “Nota”, instanciado na linha 15.

PERGUNTA

Sendo assim, será que poderíamos comentar as linhas 15 e 16, substituindo-as pelas linhas 18 e 19? A resposta é **não**.

Se descomentarmos a linha 18, não haverá problema, mas a linha 19 irá causar erro de compilação.

Isso ocorre porque “Desempenho” tem acesso ao método protegido “calcularCoeficienteRendimento” por meio da herança. Por esse motivo, ele pode ser invocado diretamente na classe (linha 17). Mas a invocação feita na linha 19 se dá por meio de uma mensagem enviada para o objeto “nota”, violando a restrição de acesso imposta por “*protected*” para objetos de pacotes distintos.

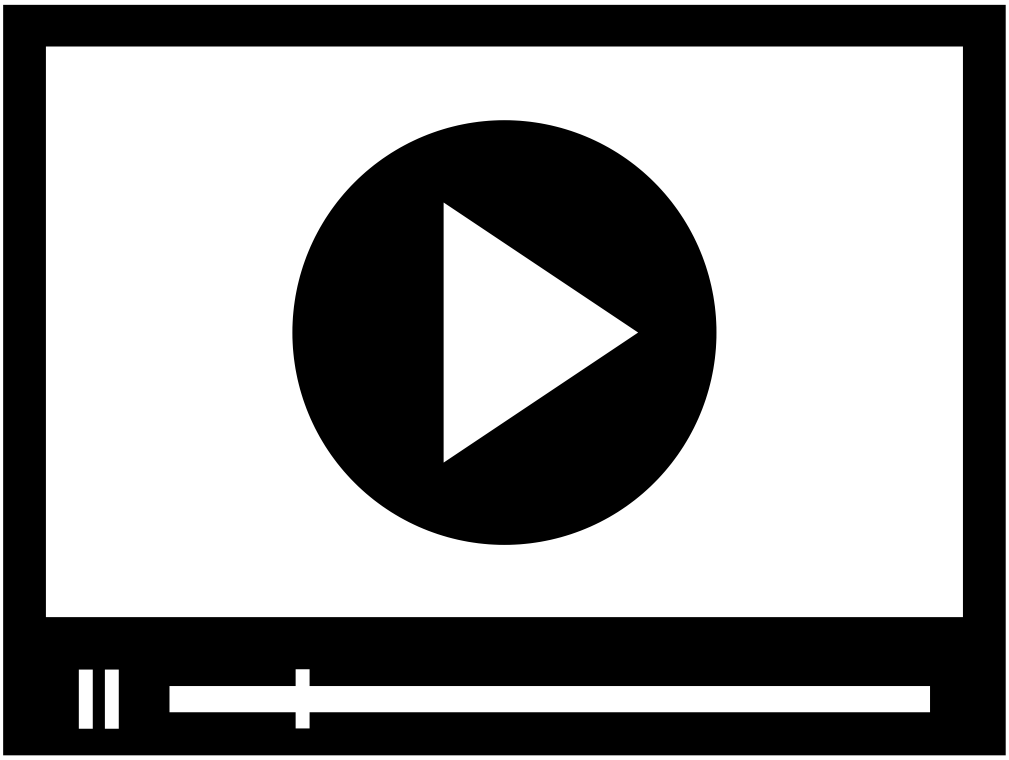
Outra violação de acesso ocorrerá se instanciarmos a classe “Desempenho” em outra parte do pacote “GereEscola” e tentarmos invocar o método “calcularCoeficienteRendimento”, conforme o trecho mostrado no Código 26.

CÓDIGO 26: EXEMPLO DE INVOCAÇÃO ILEGAL DE “CALCULARCOEFICIENTERENDIMENTO”

```
1  //Pacote
2  package com.mycompany.GereEscola;
```

```
3  ...
4  public class NovaClasse {
5  private static Desempenho des;
6  ...
7  public static void main (String args[]) {
8       des = new Desempenho ();
9       CR = des. calcularCoeficienteRendimento();
10 ...
11      }
12 ...
13 }
```

Na linha 9, a tentativa de invocar “calcularCoeficienteRendimento” irá gerar erro de compilação, pois apesar de a classe “Desempenho” ser do mesmo pacote que “NovaClasse”, o método foi recebido por “Desempenho” por meio de herança de superclasse de outro pacote. Logo, como impõe a restrição de acesso protegida, ele não é acessível por classe de outro pacote que não seja uma descendente da superclasse.



PRINCIPAIS MÉTODOS EM JAVA

Os principais métodos em Java serão apresentados no vídeo a seguir.

VERIFICANDO O APRENDIZADO

MÓDULO 3

- **Descrever polimorfismo em Java**

CLASSES E MÉTODOS ABSTRATOS

Neste módulo, vamos avançar o estudo de polimorfismo. Introduziremos o conceito de classes abstratas e veremos como a abstração de classes funciona em Java; como esse conceito afeta o polimorfismo e como ele é aplicado numa hierarquia de herança.

O módulo contempla, também, o estudo do modificador “final” e seu impacto, assim como as interações com as variáveis da super e subclasse.

No nosso estudo de hierarquia de herança e seus desdobramentos, constantemente voltamos aos conceitos de especialização e generalização. Quanto mais subimos na hierarquia, tanto mais esperamos encontrar modelos genéricos.

A generalização de modelos corresponde à generalização dos comportamentos implementados pelas superclasses. Vimos isso ocorrer no modelo mostrado na Figura 1, no qual a classe “Pessoa” representa a generalização de mais alto nível dessa hierarquia.

Se você revir os métodos da classe “Pessoa”, notará que sempre buscamos dar um comportamento genérico. Veja, como exemplo, o caso do método “atualizarID”, mostrado no Código 4. Para a classe “Pessoa” nenhuma crítica é feita ao se atualizar o atributo

“identificador”, ao contrário de suas subclasses “Fisica” e “Juridica”, que aplicam teste para verificar se o identificador é válido.

Apesar de “atualizarID” ser uma generalização de comportamento, se você refletir bem, notará que a implementação é provavelmente inútil. Qualquer caso de instanciação num caso real deverá fazer uso das versões especializadas.

Mesmo no caso de um aluno que ainda não possua CPF, é mais razoável prever isso no comportamento modelado pela versão especializada de “atualizarID” da subclasse “Pessoa”, pois esse aluno é “pessoa física”. Podemos esperar que, em algum momento, ele venha a ter um CPF e, se usar essa abordagem, manteremos a coerência, evitando instanciar objetos da classe “Pessoa”.

📢 ATENÇÃO

Pensando em termos de flexibilidade, o que esperamos de “Pessoa” é que ela defina o contrato comum a todas as classes derivadas e nos forneça o comportamento comum que não necessite de especialização.

Vejamos o exemplo a seguir:

O comportamento de “atualizarNome” (vide Código 1) é um comportamento que não esperamos que precise ser especializado, afinal pessoas físicas e jurídicas podem ser nominadas da mesma forma. Se conseguirmos isso, teremos uma classe extremamente útil e poderemos tirar o máximo proveito da OO.

Possivelmente, a primeira sugestão em que você pensou é, simplesmente, fornecer o método “atualizarID” vazio em “Pessoa”. Porém, existe um problema com essa abordagem. Imagine que desejamos que o método “atualizarID” retorne “true” quando a atualização for bem-sucedida e “false”, no inverso. Nesse caso, Java obriga que seja implementado o retorno (“return”) do método.

Além disso, há outra consideração importante. Se concebemos nosso modelo esperando que a superclasse nunca seja instanciada, seria útil se houvesse uma maneira de se impedir que isso aconteça, evitando assim um uso não previsto de nosso código.

A solução para o problema apontado é a classe abstrata, ou seja, aquela que não admite a instanciação de objetos. Em oposição, classes que podem ser instanciadas são chamadas concretas.

O propósito de uma classe abstrata é, justamente, o que mencionamos antes: fornecer uma interface e comportamentos (implementações) comuns para as subclasses. A linguagem Java implementa esse conceito por meio da instrução “*abstract*”.

Em Java, uma classe é declarada abstrata pela aplicação do modificador “*abstract*” na declaração. Isto é, podemos declarar a classe “Pessoa” como abstrata simplesmente fazendo “*public abstract class* Pessoa {...}”.

Já se a instrução “*abstract*” for aplicada a um método, este passa a ser abstrato. Isso quer dizer que ele não pode possuir implementação, pois faz parte do contrato (estrutura) fornecido para as subclasses.

Deste modo, a classe deverá, obrigatoriamente, ser declarada abstrata também. No exemplo em questão, transformamos “atualizarID” em abstrato, declarando-o “*protected abstract void* atualizarID (*String* identificador)”.

Observe que, agora, o método não pode possuir corpo.

Uma nova versão da classe “Pessoa” é mostrada no Código 27. Nessa versão, o método “atualizarID” é declarado abstrato, forçando a declaração da classe como abstrata. Entretanto, o método “recuperarID” é concreto, uma vez que não há necessidade para especializar o comportamento que ele implementa.

CÓDIGO 27: CLASSE "PESSOA" DECLARADA COMO ABSTRATA

```
1  //Classe
2  public abstract class Pessoa {
3        ...
4        private String identificador;
5        ...
6        protected abstract boolean atualizarID ( String identificador );
7        protected String recuperarID ( ) {
8              return this.identificador;
9        }
10       ...
11 }
```

Em Java, o efeito de declarar uma classe como abstrata é impedir sua instanciação. Quando um método é declarado abstrato, sua implementação é postergada. Esse método permanecerá sendo herdado como abstrato até que alguma subclasse realize sua implementação. Isso quer dizer que a abstração de um método se propaga pela hierarquia de classes. Por extensão, uma

subclasse de uma classe abstrata será também abstrata a menos que implemente o método abstrato da superclasse.

Apesar de mencionarmos que uma classe abstrata fornece, também, o comportamento comum, isso não é uma obrigação. **Nada impede que uma classe abstrata apresente apenas a interface ou apenas a implementação.** Aliás, uma classe abstrata pode ter dados de instância e construtores.

Vejamos o uso da classe abstrata “Pessoa” no seguinte trecho de código:

CÓDIGO 28: POLIMORFISMO COM CLASSE ABSTRATA

```
1  //Pacotes
2  package com.mycompany.GereEscola;
3  //Importações
4  import java.util.Calendar;
5  public class Principal {
6        //Atributos
7        private static Pessoa ref [];
8        //Método main
9        public static void main (String args[]) {
10             ref = new Pessoa [2];
11             Calendar data_nasc = Calendar.getInstance();
12             Calendar data_criacao = Calendar.getInstance();
13             data_nasc.set(1980 , 10 , 23);
14             ref [ 0 ] = new Fisica ( "Marco Antônio" , data_nasc , null , null , "B
15             data_criacao.set(1913 , 02 , 01 );
16             ref [ 1 ] = new Juridica ( "Escola Novo Mundo Ltda" , data_criacao ,
17             ref [ 0 ].atualizarID("365.586.875-45");
18             ref [ 1 ].atualizarID("43.186.666/0026-32");
19       }
20 }
```

A linha 7 do Código 28 cria um vetor de referências para objetos do tipo “Pessoa”. Nas linhas 14 e 16, são instanciados objetos do tipo “Fisica” e “Juridica”, respectivamente. Esses objetos são referenciados pelo vetor “ref”. Nas linhas 17 e 18, é invocado o método “atualizarID”, que é abstrato na superclasse, mas concreto nas subclasses.

Assim, o sistema determinará em tempo de execução o objeto, procedendo à vinculação dinâmica do método adequado. Esse é tipicamente um comportamento polimórfico. Na linha 17, o método invocado pertence à classe “Fisica” e na linha 18, à classe “Juridica”.

MÉTODOS E CLASSES “FINAL”

Vamos analisar outra hipótese agora. Considere o método “recuperarID” mostrado no Código 4. Esse método retorna o identificador na forma de uma “*String*”. Esse comportamento serve tanto para o caso de “Fisica” quanto de “Juridica”. Em verdade, é muito pouco provável que seja necessário especializá-lo, mesmo se, futuramente, uma nova subclasse for adicionada.

Esse é um dos comportamentos comuns que mencionamos antes e, por isso, ele está na superclasse. Mas se não há motivo para que esse método possa ser rescrito por uma subclasse, é desejável que possamos impedir que isso ocorra inadvertidamente. Felizmente, a linguagem Java fornece um mecanismo para isso. Trata-se do modificador “final”.

Esse modificador pode ser aplicado à classe e aos membros da classe. Diferentemente de “*abstract*”, declarar um método “final” não obriga que a classe seja declarada “final”. Porém, se uma classe for declarada “final”, todos os seus métodos são, implicitamente, “final” (isso não se aplica aos seus atributos).

Métodos “final” não podem ser redefinidos nas subclasses. Dessa forma, se tornarmos “recuperarID” “final”, impediremos que ele seja modificado, mesmo por futuras inclusões de subclasses. Esse método permanecerá imutável ao longo de toda a hierarquia de classes. Métodos “*static*” e “*private*” são, implicitamente, “final”, pois não poderiam ser redefinidos de qualquer forma.

📢 COMENTÁRIO

O uso de “final” também permite ao compilador realizar otimizações no código em prol do desempenho. Como os métodos desse tipo nunca podem ser alterados, o compilador pode substituir as invocações pela cópia do código do método, evitando desvios na execução do programa.

Vimos que, quando aplicado à classe, todos os seus métodos se tornam “final”. Isso quer dizer que nenhum deles poderá ser redefinido. Logo, não faz sentido permitir que essa classe seja estendida, e a linguagem Java proíbe que uma classe “final” possua subclasses. Contudo, ela pode possuir uma superclasse.

Quando aplicada a uma variável, “final” irá impedir que essa variável seja modificada e exigirá sua inicialização. Esta pode ser feita junto da declaração ou no construtor da classe. Quando

inicializada, qualquer tentativa de modificar seu valor irá gerar erro de compilação.

Como podemos ver no Código 29, uma classe abstrata pode ter membros “final”. A variável “dias letivos” foi declarada “final” (linha 9) e, por isso, precisa ser inicializada, o que é feito na linha 14, no construtor da classe. A não inicialização dessa variável assim como a tentativa de alterar seu valor irão gerar erro em tempo de compilação.

CÓDIGO 29: USO DE "FINAL"

```
1  //Pacote
2  package com.mycompany.GereEscola;
3  //Importações
4  public abstract class Auxiliar {
5        //Atributos
6        private float freq;
7        private final int dias_letivos;
8        private int presenca;
9        //Métodos
10       public Auxiliar ( int dias_letivos ) {
11            this.dias_letivos = dias_letivos;
12       }
13       protected final void calcularFrequencia ( ) {
14            freq = 100 * ( presenca / dias_letivos );
15       }
16       protected float recuperarFrequencia ( ) {
17            return freq;
18       }
19       protected abstract float calcularMedia ( );
20 }
```

O método “calcularFrequencia” (linha 13) foi declarado “final”, não podendo ser redefinido. Mas, como dito, a classe não é “final” e pode, então, ter classes derivadas.

ATRIBUIÇÕES PERMITIDAS ENTRE VARIÁVEIS DE SUPERCLASSE E SUBCLASSE

Já que estamos falando de como modificadores afetam a herança, é útil, também, entender como as variáveis da superclasse e da subclasse se relacionam para evitar erros conceituais,

difíceis de identificar e que levam o software a se comportar de maneira imprevisível.

Vamos começar lembrando que, quando uma variável é declarada “*private*”, ela se torna diretamente inacessível para as classes derivadas. Como vimos, nesse caso, elas são implicitamente “final”.

Elas ainda podem ter seu valor alterado, mas isso só pode ocorrer por métodos públicos ou privados providos pela superclasse. Pode soar contraditório, mas não é. As atualizações, feitas pelos métodos da superclasse, ocorrem no contexto desta, no qual a variável não foi declarada “final” e nem é, implicitamente, tomada como tal.

Podemos acessar membros públicos ou protegidos da classe derivada a partir da superclasse ou de outra parte do programa, fazendo *downcasting* da referência.

O *downcasting* leva o compilador a reinterpretar a referência empregada como sendo do tipo da subclasse. É evidente que isso é feito quando a variável de referência é do tipo da superclasse. Simetricamente, o *upcasting* causa a reinterpretação de uma variável de referência da subclasse como se fosse do tipo da superclasse. A Figura 5 ilustra o que acabamos de dizer.

O caso de variáveis privadas é óbvio. Vejamos o que ocorre com as variáveis públicas e protegidas. Como, no escopo da herança, as duas se comportam da mesma maneira, quer dizer, como ambas são plenamente acessíveis entre as classes pai e filha, trataremos, apenas, de variáveis protegidas. O mesmo raciocínio se aplicará às públicas.

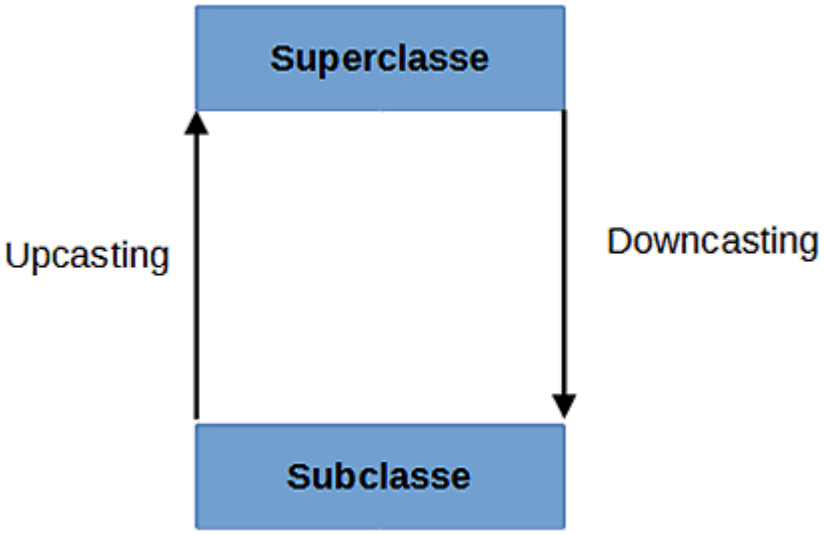


Figura 5: Reinterpretação de referência.

Para esclarecer o comportamento, usaremos as classes mostradas no Código 30 e no Código 31. O Código 32 implementa o programa principal.

CÓDIGO 30: CLASSE BASE

```
public class Base {
    protected int var_base;
    public Base ( ){
        var_base = -1;
    }
    protected void atualizarVarBase ( int valor ) {
        this.var_base = valor;
    }
    protected void imprimirVarBase ( ) {
        System.out.println ("O valor de var_base eh " + this.var_base );
    }
    protected void atualizarVarSub ( int valor ) {
        ((Derivada)this).var_der = valor;
    }
    protected void imprimirVarSub ( ) {
        System.out.println ("O valor de var_der na subclasse eh " + ((Derivada)th
    }
}
```

CÓDIGO 31: CLASSE DERIVADA

```
public class Derivada extends Base {
    protected int var_der;
    public Derivada ( ){
        System.out.println ("O valor de var_base antes da instanciacao da classe
        this.var_base = this.var_der = -2;
    }
    public void atualizarVarDer ( int valor ) {
        this.var_der = valor;
    }
    public void imprimirVarDer ( ) {
        System.out.println ("O valor de var_der eh " + this.var_der );
    }
}
```

CÓDIGO 32: VARIÁVEIS PROTEGIDAS NA HIERARQUIA DE CLASSES

```

public class Principal {
    //Atributos
    private static Derivada ref;
    //Métodos
    public static void main (String args[]) {
```

```
6           ref = new Derivada ( ); //instancia objeto do tipo Derivada
7           System.out.println ( "=> Imprime o valor de var_base");
8           ref.imprimirVarBase();
9           System.out.println ( "=> Atualiza o valor de var_der com downcasting (var
10          ref.atualizarVarSub(1000);
11          System.out.println ( "=> Imprime o valor de var_der com downcasting");
12          ref.imprimirVarSub();
13          System.out.println ( "=> Imprime o valor de var_der");
14          ref.imprimirVarDer();
15      }
16  }
```

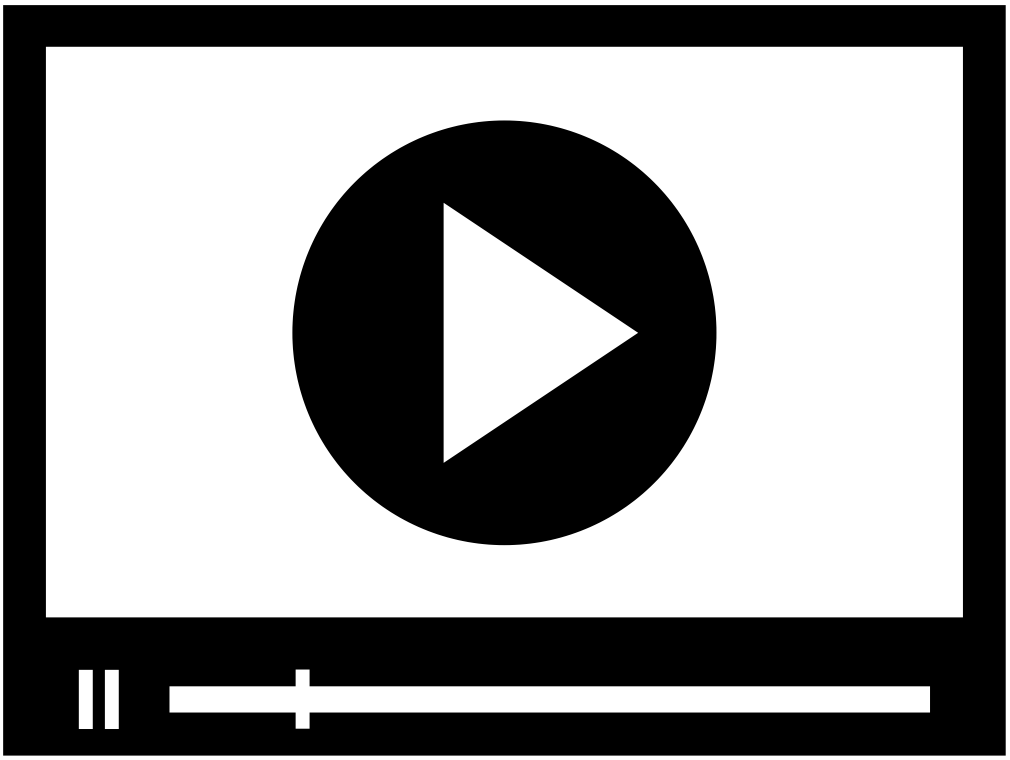
Esse código tem como saída:

```
1  O valor de var_base antes da instanciacao da classe Derivada eh -1
2  => Imprime o valor de var_base
3  O valor de var_base eh -2
4  => Atualiza o valor de var_der com downcasting (var_der = 1000)
5  => Imprime o valor de var_der com downcasting
6  O valor de var_der na subclasse eh 1000
7  => Imprime o valor de var_der
8  O valor de var_der eh 1000
```

Em primeiro lugar, o código mostra que “var_base” representa uma variável compartilhada por ambas as classes da hierarquia. Então, a mudança através de uma referência para a subclasse afeta o valor dessa variável na superclasse. Isso é esperado, pois é o mesmo espaço de memória.

A linha 6 do Código 32 deixa isso claro. Ao instanciar o objeto do tipo “Derivada”, seu construtor é chamado. A primeira ação é imprimir o valor da variável “var_base” da superclasse. Como a classe pai é instanciada primeiro, “var_base” assume valor -1. Em seguida, é sobrescrita na classe derivada com o valor -2, e uma nova impressão do seu conteúdo confirma isso.

A linha 10 do Código 32 chama o método “atualizarVarSub”, que atualiza o valor de “var_der” da subclasse, fazendo o downcasting do atributo (linha 13 do Código 30). As impressões de “var der” confirmam que seu valor foi modificado.



POLIMORFISMO EM JAVA

O especialista abordará os conceitos de polimorfismo no vídeo a seguir.

VERIFICANDO O APRENDIZADO

MÓDULO 4

- **Aplicar a criação e o uso de interfaces em Java**

ENTENDENDO A ENTIDADE “INTERFACE”

Neste módulo, estudaremos um artefato da linguagem Java chamado de Interfaces. Esse é outro conceito de OO suportado pela linguagem. Primeiramente, vamos entender o conceito e a entidade “Interface” da linguagem Java. Prosseguiremos estudando, com maior detalhamento, seus aspectos e terminaremos explorando seu uso.

Uma interface é um elemento que permite a conexão entre dois sistemas de natureza distintos, que não se conectam diretamente.

★ EXEMPLO

Um teclado fornece uma interface para conexão homem-máquina, bem como as telas gráficas de um programa, chamadas *Graphic User Interface* (GUI) ou Interface Gráfica de Usuário, que permitem a ligação entre o usuário e o *software*.

O paradigma de programação orientada a objetos busca modelar o mundo real por meio de entidades virtuais. Já discutimos as classes e os objetos como entidades que fazem tal coisa. A interface tem o mesmo propósito.

★ EXEMPLO

Podemos movê-lo no plano, e o sensor de movimento transformará esse movimento em deslocamentos do ponteiro na tela. Podemos pressionar cada um dos três botões individualmente ou de maneira combinada; e podemos girar a roda para frente e para trás. Todas essas ações físicas são transformadas pelo mouse em comandos e enviados ao computador.

Esta é toda a possibilidade de interação física com o mouse. A interface de interação com o mouse é exatamente o que acabamos de descrever. Ela é uma especificação das interações possíveis com o dispositivo, que, no entanto, não define o comportamento. Sabemos que podemos pressionar o botão direito para interagir, mas o comportamento que isso desencadeará dependerá do programa em uso.

Trazendo nosso exemplo para o *software*, podemos criar um mouse virtual com o intuito de fazer simulações por exemplo. Nesse caso, para que nosso mouse represente o modelo físico, devemos impor que sua interface de interação com o usuário seja a mesma.

Dessa forma, independentemente do tipo de mouse modelado (ótico, laser, mecânico), todos terão de oferecer a mesma interface e versão ao usuário: detecção de movimento, detecção de pressionamento dos botões 1, 2 e 3 e detecção de movimento da roda. Ainda que cada mouse simule sua própria versão do mecanismo.

Podemos, então, dizer que uma interface no paradigma OO é uma estrutura que permite garantir certas propriedades de um objeto. Ela permite definir um contrato padrão para a interação, que todos deverão seguir, isolando do mundo exterior os detalhes de implementação.

Uma definição de interface pode ser dada pelo conjunto de métodos a seguir. Qualquer programador que deseje estender nosso *mouse* deverá obedecer a esse contrato.

```
1 void onMouseMove ();
2 void onMouseButton1Click ();
3 void onMouseButton2Click ();
```

```
4  void onMouseButton3Click ();
5  void onMouseWheelSpin ();
```

PARTICULARIDADES DA “INTERFACE”

Em C++, o conceito de interface é suportado por meio de classes virtuais puras, também chamadas **classes abstratas**. A linguagem Java possui uma entidade específica chamada “Interface”. A sintaxe para se declarar uma interface em Java é muito parecida com a declaração de classe: “*public interface* Nome { ...}”. No entanto, uma interface não admite atributos e não pode ser instanciada diretamente.

Ela é um tipo de referência e somente pode conter constantes, assinaturas de métodos, tipos aninhados, métodos estáticos e default. **Apenas os métodos *default* e estático podem possuir implementação.** Tipicamente, uma interface declara um ou mais métodos abstratos que são implementados pelas classes.

Interfaces também permitem herança, mas a hierarquia estabelecida se restringe às interfaces. Uma interface não pode estender uma classe e vice-versa. Contudo, diferentemente das classes, aquelas admitem herança múltipla, o que significa que uma interface pode derivar de duas superinterfaces (interfaces pai). Nesse caso, a derivada herdará todos os métodos, as constantes e os outros tipos membros da superinterface, exceto os que ela sobrescreva ou oculte.

Uma classe que implemente uma interface deve declará-la explicitamente pelo uso do modificador “*implements*”. É possível a uma classe implementar mais de uma interface. Nesse caso, as implementadas devem ser separadas por vírgula.

Veja, a seguir, um exemplo de implementação de interfaces por uma classe:

CÓDIGO 33: INTERFACE "IDENTIFICADOR"

```
1  public interface Identificador {
2      final int tamanho_max = 21;
3      void validarID (String id);
4      void formatarID (int tipo);
5      void atualizarID (String id);
6      String recuperarID ();
7  }
```

CÓDIGO 34: INTERFACE "IPESSOA"

```
public interface iPessoa {
    void atualizarNome ( String nome );
    String recuperarNome ( );
    String recuperarNacionalidade ( );
    String recuperarNaturalidade ( );
    void atualizarIdade ( Calendar data_inicio_existencia );
    int recuperarIdade ( );
    int retornaTipo ( );
    int calcularIdade ( Calendar data_inicio_existencia );
}
```

CÓDIGO 35: IMPLEMENTAÇÃO DAS INTERFACES "IDENTIFICADOR" E "IPESSOA" PELA CLASSE "PESSOA"

```

public class Pessoa implements iPessoa , Identificador {
    //Atributos
    int idade;
    String nome = "" , naturalidade = "" , nacionalidade = "" , identificad
    //Métodos
    public void atualizarNome ( String nome ) {
        if ( !nome.isBlank() )
        this.nome = nome;
        else
        System.out.println ( "ERRO: nome em branco!" );
    }
    public String recuperarNome ( ) {
        return this.nome;
    }
    public void atualizarID ( String identificador ) {
        this.identificador = identificador;
    }
    public String recuperarID ( ) {
        return this.identificador;
    }
    public void formatarID ( int id ){
        this.identificador = String.valueOf (id);
    }
    public boolean validarID ( String id ) {
        if ( id.isBlank() || id.isEmpty() )
        return false;
        else
        return true;
    }
```

```
30      public String recuperarNacionalidade ( ) {
31          return this.nacionalidade;
32      }
33      public String recuperarNaturalidade ( ) {
34          return this.naturalidade;
35      }
36      public void atualizarIdade ( Calendar data_inicio_existencia ) {
37          this.idade = calcularIdade ( data_inicio_existencia );
38      }
39      public int recuperarIdade ( ) {
40          return this.idade;
41      }
42      public int retornaTipo ( ) {
43          return 0;
44      }
45      public int calcularIdade ( Calendar data_inicio_existencia ){
46          int lapso;
47          Calendar hoje = Calendar.getInstance();
48          lapso = hoje.get(YEAR) - data_inicio_existencia.get(YEAR);
49          if ( ( data_inicio_existencia.get(MONTH) > hoje.get(MONTH) ) ||     ( d
50          lapso--;
51          return lapso;
52      }
53  }
```

Veja que na linha 1 do Código 35 foi declarado que a classe “Pessoa” implementará as interfaces “iPessoa” e “Identificador”.

Quando uma classe implementa uma ou mais interfaces, ela deve implementar todos os métodos abstratos das interfaces.

E de fato é o que ocorre. A classe, contudo, não pode alterar a visibilidade dos métodos da interface. Assim, métodos públicos não podem ser tornados protegidos por ocasião da implementação.

Lembre-se de que uma interface define um contrato. Sua finalidade é proporcionar a interação com o mundo exterior. Por isso, não faz sentido a restrição de acesso aos métodos da interface.

Ou seja, numa interface, todos os métodos são públicos, mesmo se omitirmos o modificador “*public*”, como no Código 33 e Código 34. Mais ainda, em uma interface, todas as declarações de membro são implicitamente estáticas e públicas.

A declaração de um membro em uma interface oculta toda e qualquer declaração desse membro nas superinterfaces.

★ EXEMPLO

correspondente declaração desse método no Código 33 tornar-se-á oculta. Assim, uma interface herda de sua superinterface imediata todos os membros não privados que não sejam escondidos.

Podemos fazer declarações aninhadas, isto é, declarar uma interface no corpo de outra. Nesse caso, temos uma interface aninhada. Quando uma interface não é declarada no corpo de outra, ela é uma interface do nível mais alto (*top level* interface). Uma interface também pode conter uma classe declarada no corpo.

Observando o Código 33, vemos, na linha 2, que a interface possui um atributo estático (“tamanho_max”). Esse é um caso de atributo permitido em uma interface. Também não é possível declarar os métodos com o corpo vazio em uma interface. Imediatamente após a assinatura, a cláusula deve ser fechada com “**;**”. Java também exige que sejam utilizados identificadores nos parâmetros dos métodos, não sendo suficiente informar apenas o tipo.

Além do tipo normal de interface, existe um especial, o *Annotation*, que permite a criação de tipos de anotações pelo programador. Ele é declarado precedendo-se o identificador “interface” com o símbolo “**@**”, por exemplo: “**@interface Preambulo {...}**”. Uma vez definido, esse novo tipo de anotação torna-se disponível para uso juntamente com os tipos *bult-in*.

Fica claro, pelo que estudamos até aqui, que as interfaces oferecem um mecanismo para ocultar as implementações. São, portanto, mecanismos que nos permitem construir as **API** (*Application Programming* Interface) de *softwares* e bibliotecas.

? VOCÊ SABIA

As API possibilitam que o código desenvolvido seja disponibilizado para uso por terceiros, mantendo-se oculta a implementação, tendo muita utilidade quando não se deseja compartilhar o código que realmente implementa as funcionalidades, seja por motivos comerciais, de segurança, de proteção de conhecimento ou outros.

A pergunta que você deve estar se fazendo é:

Qual a diferença entre classes abstratas e interfaces?

Essa é uma ótima pergunta. Uma classe abstrata define um tipo abstrato de dado. Define-se um padrão de comportamento segundo o qual todas as classes que se valham dos métodos da classe abstrata herdarão da classe que os implementar.

Se você se recordar, uma classe abstrata pode possuir estado (atributos) e membros privados, protegidos e públicos. Isso é consistente com o que acabamos de dizer e muito mais do que uma interface pode possuir, significa que as classes abstratas podem representar ou realizar coisas que as interfaces não podem.

É claro que uma classe puramente abstrata e sem atributos terminará assumindo o comportamento de uma interface. Mas elas não serão equivalentes, já que uma interface admite herança múltipla, e as classes, não.

Uma interface é uma maneira de se definir um contrato. Se uma classe abstrata define um tipo (especifica o que um objeto é), uma interface especifica uma ou mais capacidades. Veja o caso do Código 33. Essa interface define as capacidades necessárias para se manipular um identificador. Não se trata propriamente de um tipo abstrato, pois tudo que ela oferece é o contrato.

Não há estado e nem mesmo comportamentos ocultos. Então, uma classe que implementar essa interface proverá o comportamento especificado pela interface.

📢 ATENÇÃO

Cuidado para não confundir a linha 2 com um estado da instância, pois não é. Além de essa variável ser declarada como final, não há métodos que atuam sobre ela para modificar seu estado.

Esclarecidas as diferenças entre classes abstratas e interfaces, resta saber quando usá-las. Essa discussão é aberta, e não há uma regra. Isso dependerá muito de cada caso, da experiência do programador e da familiaridade com o uso de ambas. Mas, com base nas diferenças apontadas, podemos estabelecer uma regra geral.

Quando seu objetivo for **especificar** as capacidades que devem ser disponibilizadas, a interface é a escolha mais indicada.

Quando estiver buscando **generalização** de comportamento e compartilhamento de código e atributos comuns (um tipo abstrato de dado), classes abstratas surgem como a opção mais adequada.

Veja a Figura 3 e a Figura 4. Ambas mesclam o uso de interfaces e classes abstratas. Na Figura 3, as interfaces "*Set*", "*List*" e "*Queue*", para citar apenas alguns, definem as capacidades que estão ligadas ao contrato que deverá ser estabelecido para a estrutura de dados com que cada uma se relaciona.

As classes abstratas, como "*AbstractList*", vão oferecer implementações compartilhadas. No caso de "*AbstractList*", podemos ver que pelo menos duas classes, "*ArrayList*" e "*Vector*", fazem uso dela. Essa figura, que mostra o modelo das coleções da API Java, é um ótimo exemplo de que os conceitos de classes abstratas e de interfaces são complementares e, portanto, não se trata de escolher entre uma e outra.

USO DE INTERFACES

Vimos a declaração das interfaces "Identificador" e "iPessoa" e da classe "Pessoa", que as implementa, respectivamente no Código 33, no Código 34 e no Código 35. Nota-se que todos os métodos abstratos de ambas as interfaces foram implementados em "Pessoa". O método "*main*" do programa, mostrado no Código 36, permite explorar seu uso.

CÓDIGO 36: TRABALHANDO COM INTERFACES

```
1  public class Principal {
2      //Atributos
3      private static Identificador refIdt;
4      private static iPessoa refiPessoa;
5      //Métodos
6      public static void main (String args[]) {
7          refIdt = new Pessoa ( );
8          refIdt.atualizarID("M-1020/001");
9          System.out.println ( refIdt.recuperarID() );
10         //refIdt.atualizarNome ("João Batista");
11         refiPessoa = (Pessoa) refIdt;
```

```
12              refiPessoa.atualizarNome("João Batista");
13              System.out.println(refiPessoa.recuperarNome());
14          }
15  }
```

Nas linhas 3 e 4 do Código 36, declaramos duas variáveis, uma que referencia “Identificador” e outra “iPessoa”. Recorde-se de que dissemos que uma interface não admite instanciação direta, mas nenhuma restrição fizemos quanto a se ter uma variável declarada como referência para a interface. E, de fato, na linha 8, não estamos instanciando a interface, mas a classe “Pessoa”. O objeto instanciado, contudo, será referenciado pela variável “refIdt”.

📢 ATENÇÃO

Podemos usar uma variável que guarda referências do tipo da interface para referenciar objetos da classe que a implementa. Entretanto, não é suficiente que a classe forneça o comportamento dos métodos abstratos, ela precisa explicitamente declarar implementar a interface.

Uma vez instanciado o objeto, podemos invocar o comportamento dos métodos especificados por “Identificador”. Mas apenas esses métodos e os que a interface “Identificador” herda estarão disponíveis nesse caso.

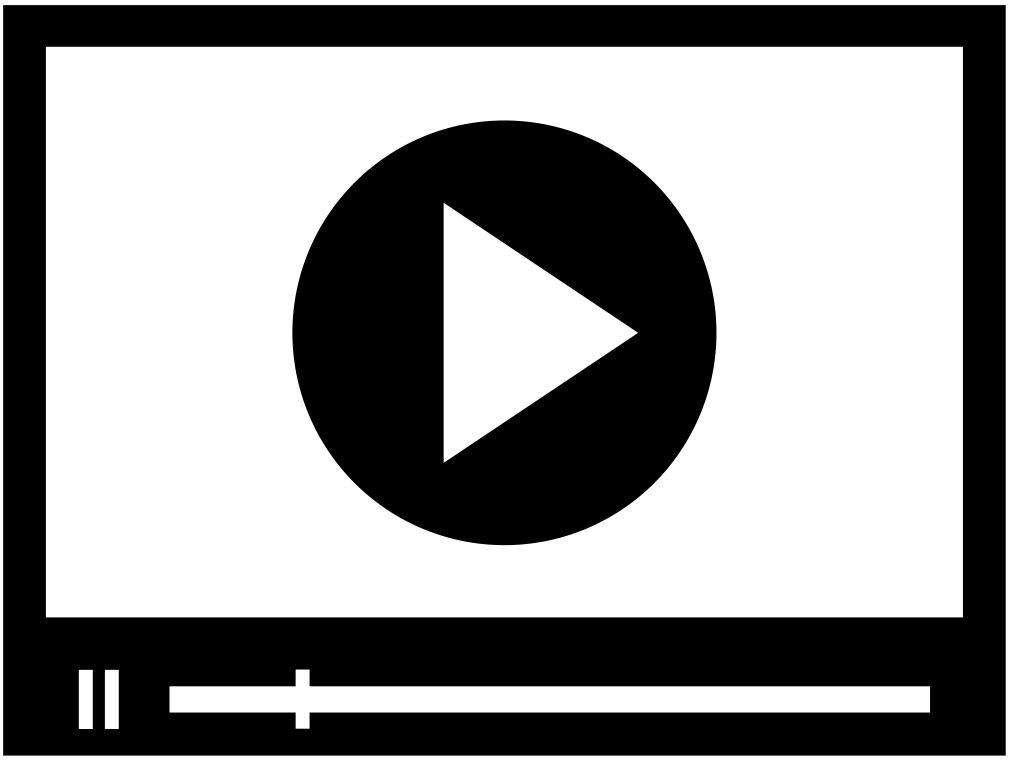
Nenhum dos métodos especificados em “iPessoa”, apesar de implementados em “Pessoa”, ou mesmo outros métodos implementados nela, estarão acessíveis por dessa referência (“refIdt”). Por isso, “descomentar” a linha 10 irá gerar erro de compilação.

Não é possível fazermos a atribuição de “refIdt” para “refiPessoa”, pois os tipos não são compatíveis. Todavia, se fizermos um *typecasting*, como o mostrado na linha 12, a atribuição se tornará possível. Agora, podemos usar a variável “refiPessoa” e acessar os métodos dessa interface que foram implementados em “Pessoa”.

É claro que, se tivéssemos usado uma variável do tipo “Pessoa” para referenciar o objeto instanciado, todos os métodos estariam prontamente acessíveis.

A saída do Código 36 é:

```
1  M-1020/001
2  João  Batista
```



CLASSES E OBJETOS EM JAVA

No vídeo a seguir, serão apresentados os conceitos de interfaces e exemplos do emprego de herança em Java.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste conteúdo, tivemos a oportunidade de aprofundar o estudo da linguagem Java. Fizemos isso investigando, em maiores detalhes, a hierarquia de herança. Compreendemos como ocorre a subtipagem e sua diferença para a subclasse, assim como vimos a vinculação dinâmica de métodos. Esses tópicos são essenciais para a melhor compreensão do polimorfismo e da programação orientada a objetos.

Nossa exploração nos mostrou os desdobramentos da herança e as potencialidades do polimorfismo. Vimos isso tanto por exemplos, como observando modelos da própria API Java. Também aprendemos sobre coleções, um tipo de interface da API que oferece importantes ferramentas de estrutura de dados para o programador.

Concluímos conhecendo a entidade “Interface” e sua relação com os conceitos vistos anteriormente. Tudo isso nos trouxe um importante entendimento sobre as potencialidades da linguagem Java e da programação OO.

Esses conceitos não são apenas uma base importante, eles são indispensáveis para qualquer programador que deseje tirar proveito de tudo o que a linguagem Java tem a oferecer.

PODCAST

No áudio a seguir, o especialista apresentará um resumo do conteúdo estudado.

AVALIAÇÃO DO TEMA:

REFERÊNCIAS

ORACLE AMERICA INC. **Java 10 Local Variable Type Inference**. Oracle Developer Resource Center, 2020.

ORACLE AMERICA INC. **Lesson**: Introduction to Collections (The JavaTM Tutorials > Collections). Java Documentation, 2020.

ORACLE AMERICA INC. **Object (Java SE 15 & JDK 15)**. Java Documentation, 2020.

ORACLE AMERICA INC. **Primitive Data Types (The JavaTM Tutorials > Learning the Java Language > Language Basics)**. The Java Tutorial, 2020.

EXPLORE+

Os princípios Solid formam um importante conjunto que todo bom programador deve dominar. Nesse módulo vimos apenas um, o princípio da substituição. Sugerimos que você procure e domine, também, os demais. Para isso, uma busca como os termos princípios Solid ou com cada um deles deve lhe trazer bons resultados.

As coleções, em Java, são outras ferramentas que merecem ser dominadas por quem quer melhorar suas habilidades. Uma boa fonte de pesquisa sobre cada tipo de coleção é a documentação Java disponibilizada pela Oracle.

Por fim, estude os códigos da API das coleções para aumentar o entendimento de como interfaces funcionam e se integram com as classes concretas e abstratas. A documentação de Java da Oracle é um bom ponto de partida!

CONTEUDISTA

Marlos de Mendonça Corrêa

🔗 CURRÍCULO LATTES