TEMA 1: INTRODUÇÃO À POO EM JAVA

Uma classe é um modelo a partir do qual se produzem os objetos. Ela define o comportamento e os atributos que todos os objetos produzidos segundo esse modelo terão.

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

Formalmente falando, uma classe é uma estrutura que define:

Os dados, Os métodos que operam sobre esses dados e formam o comportamento do objeto, O mecanismo de instanciação dos objetos.

Segundo Gosling *et. al*. (2020), pela especificação da linguagem há duas formas de declaração de classes: normal e enum. Vamos nos deter apenas na forma normal, mostrada a seguir:

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

Os modificadores podem ser qualquer elemento do seguinte conjunto:

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

Na sintaxe mostrada, os colchetes indicam elementos opcionais.

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

ANNOTATION

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

PUBLIC, PROTECTED E PRIVATE

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

STATIC

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

ABSTRACT E FINAL

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

STRICTFP

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

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

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

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

Objetos: os produtos das classes

Nem toda classe permite a criação de um objeto.

Uma classe definida como abstrata não permite a instanciação direta. É oportuno dizer que ela fornece um modelo de comportamento (e/ou estado) comum a uma coleção de classes. A tentativa de criar um objeto diretamente a partir de uma classe abstrata irá gerar erro de compilação.

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

A criação de um objeto se dá em duas etapas:

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

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

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

```
1. - Aluno objetoAluno = new Aluno ( );
```

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

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

O processo de criação do objeto começa com a alocação do espaço em memória e prossegue com a execução do código de construção do objeto.

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

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

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

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

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

Essa, porém, é apenas uma solicitação, não é uma ordem de execução, o que significa que a JVM tentará executar a coleta de lixo tão logo quanto possível, mas não necessariamente quando o método foi invocado. Apesar de a Java não implementar o destrutor, ela fornece um mecanismo para esse fim: o método "*finalize ( )*". Este, quando implementado por uma classe, é invocado no momento em que a JVM for reciclar o objeto. Todavia, o método "*finalize ( )*" é solicitado quando o objeto estiver para ser reciclado, e não quando ele sair do escopo. Isso implica, segundo Schildt (2014), que o programador não sabe quando – ou mesmo se − o método será executado.

Classes e o encapsulamento

O encapsulamento está intimamente ligado ao conceito de classes e objetos. Do ponto de vista da POO, o encapsulamento visa ocultar do mundo exterior os atributos e o funcionamento da classe. Dessa maneira, os detalhes de implementação e variáveis ficam isolados do resto do código.

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

O conceito de encapsulamento também se liga ao de visibilidade.

A visibilidade de um método ou atributo define quem pode ou não ter acesso a estes. Ou seja, ela afeta a forma como o encapsulamento funciona. Há três tipos de visibilidade, representados pelos modificadores "*private"*, *"protected"* e "*public"*.

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

Em OO há diferentes tipos de relações entre objetos.

Associação

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

Agregação

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

Composição

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

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

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

HERANÇA: ASPECTOS ELEMENTARES

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

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

Uma classe situada hierarquicamente acima é chamada de superclasse, enquanto aquelas situadas abaixo se chamam subclasses. Essas classes podem ser, também, referidas como classe base ou pai (superclasse) e classe derivada ou filha (subclasse).

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

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

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

Herança em classes.

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

Herança em interfaces.

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

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.

HERANÇA E VISIBILIDADE

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

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

PADRÃO

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

PRIVADO

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

PROTEGIDO

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

PUBLICO

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

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

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

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.

POLIMORFISMO

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

IMPLEMENTANDO O AGRUPAMENTO DE OBJETOS

No agrupamento, o estado final desejado é ter os objetos agrupados, e cada agrupamento deve estar mapeado para a chave usada como critério. Em outras palavras, buscamos construir uma função tal que, a partir de um universo de objetos de entrada, tenha como saída "n" pares ordenados formados pela chave de particionamento e a lista de objetos agrupados.

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

Para mantermos os pares de particionamento, usaremos o container "Map", que faz o mapeamento entre uma chave e um valor.

```
27. public void agruparAlunos ( ) {
28.     Map < String , List < Aluno > > agrupamento = new HashMap <> ();
29.     for ( Aluno a : discentes ) {
30.             if(!agrupamento.containsKey ( a.recuperarNaturalidade ( ) )) {
31.                     agrupamento.put( a.recuperarNaturalidade ( ) , new ArrayList<>( ) );
32.             }
33.     agrupamento.get(a.recuperarNaturalidade ( ) ).add(a);
34.     }
35.     System.out.println ("Resultado do agrupamento por naturalidade: " + agrupamento );
36. }
```

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

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

AGRUPANDO OBJETOS COM A CLASSE "COLLECTORS" DA API JAVA

A classe "*Collectors*" é uma classe da API Java que implementa vários métodos úteis de operações de redução, como, por exemplo, o agrupamento dos objetos em coleções, de acordo com a Oracle America Inc. (2021). Com o auxílio da API Java, não precisaremos varrer a lista de objetos. Vamos transformar esses objetos em um fluxo ("*stream*") que será lido pela classe "*Collectors*", a qual realizará todo o trabalho de classificação dos objetos, nos devolvendo um mapeamento.

A operação de agrupamento é feita pelo método "*groupingBy*".

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

```
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. Isto é, são os rótulos ou as chaves de cada grupo ou coleção de objetos formados. Conforme o agrupador da classe "*Collectors*" vai lendo os objetos do fluxo de entrada, ele vai criando coleções de objetos correspondentes a cada classificador. O resultado é um par ordenado (Chave, Coleção), que é armazenado em uma estrutura de mapeamento "*Map*".

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

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

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

```
27. public void agruparAlunos ( ) {
28.     Map < String , List < Aluno > > agrupamento = discentes.stream().collect(Collectors.groupingBy(
Aluno::recuperarNaturalidade));
```

```
29.     System.out.println ("Resultado do agrupamento por naturalidade: ");
30.     agrupamento.forEach (( String chave , List < Aluno > lista) -> System.out.println (chave + "="
+ lista ));
31. }
```

Analisemos a linha 28:

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

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

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

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

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

```
1. public void agruparAlunos ( int a ) {
2.      Map < String , Set < Aluno > > agrupamento = discentes.stream().collect(Collectors.groupingBy(Aluno::recuperarNaturalidade
, Collectors.toSet()));
3.      System.out.println ("Resultado do agrupamento por naturalidade: ");
4.      agrupamento.forEach (( String chave , Set < Aluno > conjunto) -> System.out.println (chave + " = " + conjunto ));
5. }
```

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

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

```
1. public void agruparAlunos ( double a ) {
2.      Map < String , Set < Aluno > > agrupamento = discentes.stream().collect(Collectors.groupingBy(Aluno::recuperarNaturalidade,
TreeMap::new , Collectors.toSet()));
3.      System.out.println ("Resultado do agrupamento por naturalidade: ");
4.      agrupamento.forEach (( String chave , Set < Aluno > conjunto) -> System.out.println (chave + " = " + conjunto ));
5. }
```

*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*").*

COLEÇÕES

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

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

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

*As interfaces providas são genéricas, o que significa que precisarão ser especializadas (vimos isso ocorrer na linha 2 da assinatura 2). Por exemplo, quando fizemos "*Set < Aluno >*". Nesse caso, instruímos o compilador a especializar a coleção "*Set*" para a classe "Aluno".*

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

SET

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

LIST

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

QUEUE

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

DEQUE

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

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

Se observarmos a linha 2 da assinatura 3 por exemplo, veremos que a chave é mapeada em um único objeto que, no caso, é uma lista de objetos "Aluno". Uma tabela hash fornece uma boa noção do funcionamento de "Map".

JAVA VERSUS C/C++: UM BREVE COMPARATIVO

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

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

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

Java e C++ são linguagens OO.

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

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

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

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

AMBIENTES DE DESENVOLVIMENTO JAVA

*Comecemos pela **JVM**, que já dissemos ser uma abstração da plataforma.*

A abstração procura ocultar do software Java detalhes específicos da plataforma, como o tamanho dos registradores da CPU ou sua arquitetura – RISC ou CISC.

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

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

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

Estrutura de desvio − if.

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

Estrutura de desvio − "switch".

```
1. switch ( < expressão> ) {
2. case < valor 1>:
3.          bloco;
4.          break;
5.          .
6.          .
7.          .
8. case :
9.          bloco;
10.         break;
11. default:
12.         bloco;
13.         break;
14. }
```

Estrutura de repetição − "while".

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

Estruturas de repetição − "do-while".

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

Estruturas de repetição − "for".

```
1. for ( inicialização ; expressão ; iteração ) {
2.          bloco;
3. }
```

TEMA *2 – HERANÇA E POLIMORFISMO*

Herança e a instanciação de objetos em Java

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.

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.

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.

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.

HERANÇA, SUBTIPOS E O PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV

Código Pessoa

```
public class Pessoa {
        ...
        private String identificador;
        ...
        protected void atualizarID ( String identificador ) {
                this.identificador = identificador;
        }
}
```

Código Física

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

Código Jurídica

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

Código Aluno

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

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.

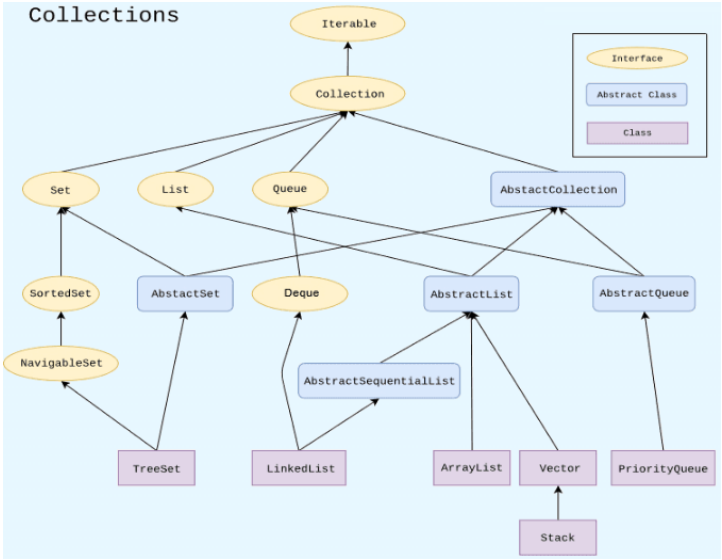
Assim, em todo ponto do programa no qual um objeto de “Pessoa” for usado, objetos de “Fisica” e “Juridica” podem ser aplicados sem quebrar o programa, mas o mesmo não se dá com objetos do tipo “Aluno”.

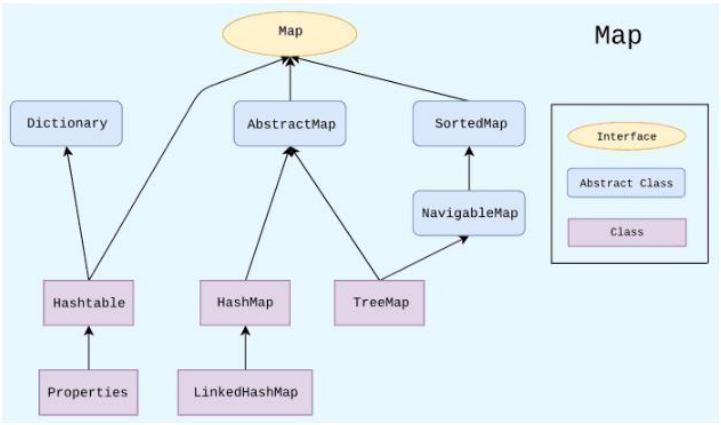
HIERARQUIA DE COLEÇÃO

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” e a outra, formada pela “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.

“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.
)bjetct [ ] toArray (Object e)	Retorna um vetor que contém todos os elementos da coleção que o invoca.

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.

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.

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.

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.

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.

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.

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*".

```
1 public class Principal {
2  //Atributos
3  private static int I1 , I2 , I3;
4  private static String S1 , S2 , S3;
5  private static Fisica p1 , p2;
6  private static Pessoa p3;
7
8  //Métodos
9  public static void main (String args[]) {
10                    I1 = 1;
11                    I2 = 2;
12                    I3 = 1;
13                    S1 = "a";
14                    S2 = "b";
15                    S3 = "a";
16                    Calendar data_nasc = Calendar.getInstance();
17                    data_nasc.set(1980, 10, 23);
18                    p1 = new Fisica ( "Marco Antônio" , data_nasc , "365.586.875-45" , null , "Brasil" , "Rio de Janeiro" );
19                    p2 = new Fisica ( "Marco Antônio" , data_nasc , "365.586.875-45" , null , "Brasil" , "Rio de Janeiro" );
20                    p3 = new Pessoa ( "Classe Pessoa" , null , null , null , "Brasil" , "Rio de Janeiro");
21                    compararEquals ( p1 , p2 , p3 );
22                    compararEquals ( I1, I2, I3 );
23                    compararEquals ( S1, S2, S3 );
24            }
25
```

```
26          private static void compararEquals ( Object o1 , Object o2 , Object o3 ){
27                  System.out.println("Uso de EQUALS para comparar " + o1.getClass().getName());
28                          if ( o1.equals( o2 ) )
29                                  System.out.println("o1 == o2");
30                          else
31                                  System.out.println("o1 != o2");
32                          if ( o1.equals(o3) )
33                                  System.out.println("o1 == o3");
34                          else
35                                  System.out.println("o1 != o3");
36          }
37 }
```

Nas linhas 21, 22 e 23 do Código, 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".

método "equals" especializado

```
1 public boolean equals (Object obj) {
2  if ( ( nome.equals( ( ( Pessoa ) obj ).nome ) ) && ( this instanceof 3 Pessoa ) )
4           return true;
5  else
6           return false;
7 }
```

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:

```
1 p1 = 2001049719
2 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 a baixo.

```
1 public int hashCode () {
2  if ( this instanceof Pessoa )
3          return this.nome.hashCode();
4  else
5          return super.hashCode();
6 }
```

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
```

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*".

Assim, "*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 ou a própria subclasse.

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 a seguir:

Código parcial da classe "Nota"

```
package com.mycompany.Matematica;
...
public class Nota {
...
   public float calcularMedia () {
...
   }
   protected float calcularCoeficienteRendimento () {
...
   }
...
}
```

No pacote "GereEscola", vamos criar a classe "Desempenho", conforme mostrado, parcialmente:

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.

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 a seguir:

Exemplo de invocação ilegal de “calcularCoeficienteRendimento”

```
1 //Pacote
2 package com.mycompany.GereEscola;
3 ...
4 public class NovaClasse {
5 pri’’vate 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.

CLASSES E MÉTODOS ABSTRATOS

O comportamento de "atualizarNome" 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, 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.

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.

MÉTODOS E CLASSES "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.

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.

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.

ENTENDENDO A ENTIDADE “INTERFACE”

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.

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*.

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.

PARTICULARIDADES DA “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.

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

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.

Se uma interface derivada de "Identificador" declarar um método com o nome "validarID", a 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.

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.

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.

Esclarecidas as diferenças entre classes abstratas e interfaces, resta saber quando usá-las.

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

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

TEMA 3 - TRATAMENTO DE EXCEÇÕES EM JAVA

CONCEITOS

Uma exceção é uma condição causada por um erro em tempo de execução que interrompe o fluxo normal de execução. Esse tipo de erro pode ter muitas causas, como uma divisão por zero, por exemplo. Quando uma exceção ocorre em Java:

É criado um objeto chamado de exception object, que contém informações sobre o erro, seu tipo e o estado do programa quando o erro ocorreu.

Esse objeto é entregue para o sistema de execução da Máquina Virtual Java (MVJ), processo esse chamado de lançamento de exceção.

Uma vez que a exceção é lançada por um método, o sistema de execução vai procurar na pilha de chamadas (*call stack*) por um método que contenha um código para tratar essa exceção. O bloco de código que tem por finalidade tratar uma exceção é chamado de: *exception handler* (tratador de exceções).

A busca seguirá até que o sistema de execução da MVJ encontre um *exception handler* adequado para tratar a exceção, passando-a para ele.

Quando isso ocorre, é verificado se o tipo do objeto de exceção lançado é o mesmo que o tratador pode tratar. Se for, ele é considerado apropriado para aquela exceção.

Quando o tratador de exceção recebe uma exceção para tratar, diz-se que ele captura (*catch*) a exceção.

Ao fornecer um código capaz de lidar com a exceção ocorrida, o programador tem a possibilidade de evitar que o programa seja interrompido. Todavia, se nenhum tratador de exceção apropriado for localizado pelo sistema de execução da MVJ, o programa irá terminar.

A possibilidade de agrupar e diferenciar os tipos de erros. Java trata as exceções lançadas como objetos que, naturalmente, podem ser classificados com base na hierarquia de classes. Por exemplo, erros definidos mais abaixo na hierarquia são mais específicos, ao contrário dos que se encontram mais próximos do topo, seguindo o princípio da especialização/generalização da programação orientada a objetos.

TIPOS DE EXCEÇÕES

Todos os tipos de exceção em Java são subclasses da classe Throwable. A primeira separação feita agrupa as exceções em dois subtipos:

ERROR

Agrupa as exceções que, em situações normais, não precisam ser capturadas pelo programa. Em situações normais, não se espera que o programa cause esse tipo de erro, mas ele pode ocorrer e, quando ocorre, causa uma falha catastrófica.

A subclasse Error agrupa erros que ocorrem em **tempo de execução**, ela é utilizada para o sistema de execução Java indicar erros relacionados ao ambiente de execução. Um estouro de pilha (*stack overflow*) é um exemplo de exceção que pertence à subclasse Error.

EXCEPTION

Agrupa as exceções que os programas deverão capturar e permite a extensão pelo programador para criar suas próprias exceções. Uma importante subclasse de Exception é a classe RuntimeException, que corresponde às exceções como a divisão por zero ou o acesso a índice inválido de array, e que são automaticamente definidas.

Exceções implícitas

Exceções implícitas são aquelas definidas nos subtipos Error e RuntimeException e suas derivadas. Trata-se de exceções ubíquas, quer dizer, que podem ocorrer em qualquer parte do programa e normalmente não são causadas diretamente pelo programador. Por essa razão, possuem um tratamento especial e não precisam ser manualmente lançadas.

Como exemplo, imagine um código cuja finalidade seja efetuar uma divisão entre dois números, no bloco que efetua a divisão, perceberemos que a divisão por zero não está presente no código. Ela não realiza a divisão que irá produzir a exceção a menos que o usuário, durante a execução, entre com o valor zero para o divisor, situação em que o Java runtime detecta o erro e lança a exceção ArithmeticException.

Outra exceção implícita, o problema de estouro de memória (*OutOfMemoryError*) pode acontecer em qualquer momento, com qualquer instrução sendo executada, pois sua causa não é a instrução em si, mas um conjunto de circunstâncias de execução que a causaram. Também são exemplos as exceções NullPointerException e IndexOutOfBoundsException, entre outras.

As exceções implícitas são lançadas pelo próprio Java runtime, não sendo necessária a declaração de um método throw para ela. Quando uma exceção desse tipo ocorre, é gerada uma referência implícita para a instância lançada.

O Código abaixo nos mostra um bloco *try-catch*. O bloco *try* indica o bloco de código que será monitorado para ocorrência de exceção e o bloco catch captura e trata essa exceção. Mas mesmo nessa versão modificada, o programador não está lançando a exceção, apenas capturando-a. A exceção continua sendo lançada automaticamente pelo Java runtime. Cabe dizer, contudo, que não há impedimento a que o programador lance manualmente a exceção. Veja o Código 4, que modifica o Código 3, passando a lançar a exceção.

```
1 try {
2 if ( divisor ==0 )
3 throw new ArithmeticException ( "Divisor nulo." );
4 quociente = dividendo / divisor;
5 }
6 catch (Exception e)
7 {
8 System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
9 }
```

Embora seja possível ao programador lançar manualmente a exceção, os chamadores do método que a lançam não serão forçados pelo interpretador a tratar a exceção. Apenas exceções que não são implícitas, isto é, lançadas pelo Java runtime, devem obrigatoriamente ser tratadas. Por isso, também dizemos que exceções implícitas são contornáveis, pois

podemos simplesmente ignorar seu tratamento. Isso, porém, não tende a ser uma boa ideia, já que elas serão lançadas e terminarão por provocar a saída anormal do programa

Exceções explícitas

Todas as exceções que não são implícitas são consideradas explícitas. Esse tipo de exceção, de maneira oposta às implícitas, é dito incontornável.

Quando um método usa uma exceção explícita, ele obrigatoriamente deve usar uma instrução throw no corpo do método para criar e lançar a exceção.

O programador pode escolher não capturar essa exceção e tratá-la no método onde ela é lançada. Ou fazê-lo e ainda assim desejar propagar a exceção para os chamadores do método. Em ambos os casos, deve ser usada uma cláusula throws na assinatura do método que declara o tipo de exceção que o método irá lançar se um erro ocorrer. Nesse caso, os chamadores precisarão envolver a chamada em um bloco *try-catch*.

O Código 5 mostra um exemplo de exceção explícita (*IllegalAccessException*). Essa exceção é uma subclasse de ReflectiveOperationException que, por sua vez, descende da classe Exception. Logo, não pertence aos subtipos Error ou RuntimeException que correspondem às exceções explícitas.

```
1 public class Arranjo {
2 int [] vetor = { 1 , 2 , 3, 4 };
3 int getElemento ( int i ) {
4 try {
5 if ( i < 0 || i > 3 )
6 throw new IllegalAccessException ();
7 } catch ( Exception e ) {
8 System.out.println ( "ERRO: índice fora dos limites do vetor." );
9 }
10 return vetor [i];
11 }
12 }
```

Como é uma exceção explícita, ela precisa ser tratada em algum momento. No caso mostrado no Código 5, ela está abarcada por um bloco try-catch, o que livra os métodos chamadores de terem de lidar com a exceção gerada.

E quais seriam as consequências se o programador optasse por não a tratar localmente? Nesse caso, o bloco try-catch seria omitido com a supressão das linhas 4, 7, 8 e 9. A linha 6, que lança a exceção, teria de permanecer, pois, lembremos, é uma exceção explícita. Ao fazer isso, o interpretador Java obrigaria a que uma cláusula throws fosse acrescentada na assinatura do método, informando aos chamadores as exceções que o método pode lançar.

DESCREVER A CLASSE EXCEPTION DE JAVA

Conceitos

O tratamento de exceções em Java é um mecanismo flexível que agrega ao programador um importante recurso. Para extrair tudo que ele tem a oferecer, faz-se necessário conhecer mais detalhadamente o seu funcionamento. Já dissemos que as exceções provocam um desvio na lógica de execução do programa. Mas esse desvio pode trazer consequências indesejadas. A linguagem, porém, oferece uma maneira de remediar tal situação.

Três cláusulas se destacam quando se busca compreender o mecanismo de tratamento de exceções de Java:

finally

A instrução finally oferece uma maneira de lidar com certos problemas gerados pelo desvio indesejado no fluxo do programa.

throw

A instrução throw é básica, por meio dela podemos lançar as exceções explícitas e, manualmente, as implícitas.

throws

A instrução throws pode ser conjugada para alterar a abordagem no tratamento das exceções lançadas.

Comando finally

Java oferece um mecanismo para lidar com tais situações: **O bloco finally**. Esse bloco será executado independentemente de uma exceção ser lançada no bloco try ao qual ele está ligado. Aliás, mesmo que haja instruções return, break ou continue no bloco try, finally será executado. A única exceção é se a saída se der pela invocação de System.exit, o que força o término do programa.

O bloco finally é opcional. Entretanto, um bloco try exige pelo menos um bloco catch ou um bloco finally. Colocando de outra forma, se usarmos um bloco try, podemos omitir o bloco catch desde que tenhamos um bloco finally. No entanto, diferentemente de catch, um try pode ter como correspondente uma, e somente uma, cláusula finally. Assim, se suprimíssemos o bloco catch da linha 17 do Código 9, mas mantivéssemos o bloco finally, o programa compilaria sem problema.

Comando throw

Já vimos que Java lança automaticamente as exceções da classe Error e RuntimeException, as chamadas exceções implícitas. Mas para lançarmos manualmente uma exceção, precisamos nos valer do **comando throw**. Por meio dele, lançamos as chamadas exceções explícitas. A sintaxe de throw é bem simples e consiste no comando seguido da exceção que se quer lançar.

O objeto lançado por throw deve, sempre, ser um objeto da classe Throwable ou de uma de suas subclasses. Ou seja, deve ser uma exceção. Não é possível lançar tipos primitivos – int, string, char – ou mesmo objetos da classe Object. Tais elementos não podem ser usados como exceção. Uma instância throwable pode ser obtida por meio da passagem de parâmetro na cláusula catch ou com o uso do operador new.

O efeito quando o compilador encontra uma cláusula throw é de desviar a execução do programa. Isso significa que nenhuma instrução existente após a cláusula será executada.

Aplicar o mecanismo de tratamento de exceções que Java disponibiliza

Certamente os erros causados quando se tenta ler além do fim de um array, quando ocorre uma divisão por zero ou mesmo uma tentativa falha de abrir um arquivo que já não existe mais geram problemas que precisam ser enfrentados. Mas, para todos esses casos, técnicas tradicionais podem ser empregadas.

Um ponto que também deve ser considerado é o impacto que o tratamento de exceções traz ao desempenho do software. Num artigo intitulado *Efficient Java Exception Handling in Just-in-Time Compilation* (SEUNGIL *et al.*, 2000), os autores argumentam que o tratamento de exceções impede o compilador Java de fazer algumas otimizações, impactando o desempenho do programa.

Notificação de exceção

A notificação é o procedimento pelo qual um método avisa ao chamador das exceções que pode lançar. Ela é obrigatória se estivermos lançando uma exceção explícita, e sua ausência irá gerar erro de compilação. Contudo, mesmo se lançarmos uma exceção implícita manualmente, ela não é obrigatória. Fazemos a notificação utilizando a cláusula throws ao fim da assinatura do método.

A razão para isso é simples: as exceções explícitas precisam ser tratadas pelo programador, enquanto as implícitas podem ser simplesmente deixadas para o tratador de exceções padrão de Java. Quando uma exceção implícita é lançada, se nenhum tratador adequado for localizado, a exceção é passada para o tratador padrão, que força o encerramento do programa.

Lançamento de exceção

O lançamento de uma exceção pode ocorrer de maneira implícita ou explícita, caso em que é utilizada a instrução throw. Como já vimos, quando uma exceção é lançada, há um desvio indesejado no fluxo de execução do programa. Isso é importante, pois o lançamento de uma exceção pode se dar fora de um bloco try-catch. Nesse caso, o programador deve ter o cuidado de inserir o lançamento em fluxo alternativo, pois, do contrário, a instrução throw estará no fluxo principal e o compilador irá acusar erro de compilação, pois todo o código abaixo dela será inatingível.

TEMA 4 – PROGRAMAÇÃO PARALELA EM JAVA: THREADS

Na era da computação da moderna, podemos executar múltiplas tarefas concomitantemente graças aos já mencionados hyperthreading, CPU de núcleo múltiplo e sistemas operacionais multitarefa preemptiva.

O termo thread ou processo leve consiste em uma sequência de instruções, uma linha de execução dentro de um processo.

EXECUÇÃO DE SOFTWARE POR UM COMPUTADOR TEÓRICO

Configuração: CPU genérica de núcleo único

Nossa primeira configuração era muito comum há pouco mais de uma década. Imagine que você está usando o Word para fazer um trabalho e, ao mesmo tempo, está calculando a soma hash (soma utilizando algoritmo) de um arquivo.

Os sistemas operacionais multitarefa preemptiva implementam o chamado escalonador de processos. O escalonador utiliza algoritmos que gerenciam o tempo de CPU que cada processo pode utilizar. Assim, quando o tempo é atingido, uma interrupção faz com que o estado atual da CPU (registradores, contador de execução de programa e outros parâmetros) seja salvo em memória, ou seja, o processo é tirado do contexto da execução e outro processo é carregado no contexto.

Toda vez que o tempo determinado pelo escalonador é atingido, essa troca de contexto ocorre, e as operações são interrompidas mesmo que ainda não finalizadas. A sua execução é retomada quando o processo volta ao contexto.

Configuração: CPU multinúcleo

Vamos então considerar a segunda configuração. Agora, temos mais de um núcleo. Cada núcleo da CPU, diferentemente do caso da hyperthreading, é um pipeline completo e independente dos demais. Sendo assim, o núcleo 0 tem seus próprios registradores, de forma que a execução de um código pelo núcleo 1 não interferirá no outro.

Sempre que um software é executado, ele dispara um processo. Os valores de registrador, a pilha de execução, os dados e a área de memória fazem parte do processo. Quando um processo é carregado em memória para ser executado, uma área de memória é reservada e se torna exclusiva. Um processo pode criar subprocessos, chamados também de processos filhos.

O que são Threads

As threads são linhas de execução de programa contidas nos processos. Diferentemente deles, elas não possuem uma área de memória exclusiva, mas compartilham o mesmo espaço. Por serem mais simples que os processos, sua criação, finalização e trocas de contexto são mais rápidas, oferecendo a possibilidade de paralelismo com baixo custo computacional, quando comparadas aos processos. O fato de compartilharem a memória também facilita a troca de dados, reduzindo a latência envolvida nos mecanismos de comunicação interprocessos.

THREADS EM JAVA

A linguagem Java é uma linguagem de programação multithread, o que significa que Java suporta o conceito de threads. Como vimos, uma thread pode ser preemptada da execução e isso é feito pelo sistema operacional que emite comandos para o hardware. Por isso, nas primeiras versões da MVJ (Máquina Virtual Java) o uso de threads era dependente da plataforma. Logo, se o programa usasse threads, ele perdia a portabilidade oferecida pela MVJ. Com a evolução da tecnologia, a MVJ passou a abstrair essa funcionalidade, de forma que tal limitação não existe atualmente.

Uma thread é uma maneira de implementar múltiplos caminhos de execução em uma aplicação.

A nível do sistema operacional (SO), diversos programas são executados preemptivamente e/ou em paralelo, com o SO fazendo o gerenciamento do tempo de execução. Um programa, por sua vez, pode possuir uma ou mais linhas de execução capazes de realizar tarefas distintas simultaneamente (ou quase).

Toda thread possui uma prioridade. A prioridade de uma thread é utilizada pelo escalonador da MVJ para decidir o agendamento de que thread vai utilizar a CPU. Threads com maior prioridade têm preferência na execução, porém é importante notar que ter preferência não é ter controle total. Suponha que uma aplicação possua apenas duas threads, uma com prioridade máxima e a outra com prioridade mínima. Mesmo nessa situação extrema, o escalonador deverá, em algum momento, preemptar a thread de maior prioridade e permitir que a outra receba algum tempo de CPU. Na verdade, a forma como as threads e os processos são escalonados depende da política do escalonador.

Isso é necessário para que haja algum paralelismo entre as threads. Do contrário, a execução se tornaria serial, com a fila sendo estabelecida pela prioridade. Num caso extremo, em que novas threads de alta prioridade continuassem sendo criadas, threads de baixa prioridade seriam adiadas indefinidamente.

A prioridade de uma thread não garante um comportamento determinístico. Ter maior prioridade significa apenas isso. O programador não sabe quando a thread será agendada.

Em Java, há dois tipos de threads:

Daemon

As daemon threads são threads de baixa prioridade, sempre executadas em segundo plano. Essas threads provêm serviços para as threads de usuário (user threads), e sua existência depende delas, pois se todas as threads de usuário finalizarem, a MVJ forçará o encerramento da daemon thread, mesmo que suas tarefas não tenham sido concluídas. O Garbage Collector (GC) é um exemplo de daemon thread. Isso esclarece por que não temos controle sobre quando o GC será executado e nem se o método “finalize” será realizado.

User

Threads de usuário são criadas pela aplicação e finalizadas por ela. A MVJ não força sua finalização e aguardará que as threads completem suas tarefas. Esse tipo de thread executa em primeiro plano e possui prioridades mais altas que as daemon threads. Isso não permite ao usuário ter certeza de quando sua thread entrará em execução, por isso mecanismos adicionais precisam ser usados para garantir a sincronicidade entre as threads. Veremos esses mecanismos mais à frente.

CICLO DE VIDA DE THREAD EM JAVA

Quando a MVJ inicia, normalmente há apenas uma thread não daemon, que tipicamente chama o método “main” das classes designadas. A MVJ continua a executar threads até que o método “exit” da classe “Runtime” é chamado e o gerenciador de segurança permite a saída ou até que todas as threads que não são daemon estejam mortas (ORACLE AMERICA INC., s.d.).

Há duas maneiras de se criar uma thread em Java:

Declarar a classe como subclasse da classe Thread.

Declarar uma classe que implementa a interface “Runnable”.

Toda thread possui um nome, mesmo que ele não seja especificado. Nesse caso, um nome será automaticamente gerado. Veremos os detalhes de criação e uso de threads logo mais. Uma thread pode existir em 6 estados.

NEW

A thread está nesse estado quando é criada e ainda não está agendada para execução (SCHILDT, 2014).

RUNNABLE

A thread entra nesse estado quando sua execução é agendada (escalonamento) ou quando entra no contexto de execução, isto é, passa a ser processada pela CPU (SCHILDT, 2014).

BLOCKED

A thread passa para este estado quando sua execução é suspensa enquanto aguarda uma trava (lock). A thread sai desse estado quando obtém a trava (SCHILDT, 2014).

TIMED_WAITING

A thread entra nesse estado se for suspensa por um período, por exemplo, pela chamada do método “sleep ()” (dormindo), ou quando o timeout de “wait ()” (esperando) ou “join ()” (juntando) ocorre. A thread sai desse estado quando o período de suspensão é transcorrido (SCHILDT, 2014).

WAITING

A chamada aos métodos “wait ()” ou “join ()” sem timeout ou “park ()” (estacionado) leva a thread ao estado de “WAITING” (SCHILDT, 2014).

TERMINATED

O último estado ocorre quando a thread encerra sua execução (SCHILDT, 2014).

É possível que em algumas literaturas você encontre essa máquina de estados com nomes diferentes. Conceitualmente, a execução da thread pode envolver mais estados, e, sendo assim, você pode representar o ciclo de vida de uma thread de outras formas. Mas além de isso não invalidar a máquina mostrada em nossa figura, esses estados são os especificados pela enumeração “State” (ORACLE AMERICA INC., s.d.) da classe “Thread” e retornados pelo método “getState ()”. Isso significa que, na prática, esses são os estados com os quais você irá operar numa implementação de thread em Java.

Convém observar que, quando uma aplicação inicia, uma thread começa a ser executada. Essa thread é usualmente conhecida como thread principal (main thread) e existirá sempre, mesmo que você não tenha empregado threads no seu programa. Nesse caso, você terá um programa single thread, ou seja, de thread única. A thread principal criará as demais threads, caso necessário, e deverá ser a última a encerrar sua execução.

Quando uma thread cria outra, a mais recente é chamada de thread filha. Ao ser gerada, a thread receberá, inicialmente, a mesma prioridade daquela que a criou. Além disso, uma thread será criada como daemon apenas se a sua thread criadora for um daemon. Todavia, a thread pode ser transformada em daemon posteriormente, pelo uso do método “setDaemon()”.

CRIANDO UMA THREAD

Trata-se mais de oferecer alternativas em linha com os conceitos de orientação a objetos (OO). A extensão de uma classe normalmente faz sentido se a subclasse vai acrescentar comportamentos ou modificar a sua classe pai.

Mecanismo de herança

Utilizar o mecanismo de herança com o único objetivo de criar uma thread pode não ser a abordagem mais interessante. Mas, se houver a intenção de se acrescentar ou modificar métodos da classe “Thread”, então a extensão dessa classe se molda melhor, do ponto de vista conceitual.

Implementação de “Runnable”

A implementação do método “run ()” da interface “Runnable” parece se adequar melhor à criação de uma thread. Além disso, como Java não aceita herança múltipla, estender a classe Thread pode complicar desnecessariamente o modelo de classes, caso não haja a necessidade de se alterar o seu comportamento. Essa razão também está estreitamente ligada aos princípios de OO.

Identificar a sincronização entre threads em Java

Imagine que desejamos realizar uma busca textual em um documento com milhares de páginas, com a intenção de contar o número de vezes em que determinado padrão ocorre.

Podemos criar um certo número de threads e repartir as páginas do documento entre as threads, deixando a própria aplicação consolidar o resultado. Essa solução, embora tecnicamente engenhosa, é mais simples de descrever do que de fazer. Mas ao paralelizar uma aplicação com o uso de threads, duas questões importantes se colocam:

Como realizar a comunicação entre as threads?

Como coordenar as execuções de cada thread?

cada thread está varrendo em paralelo determinado trecho do documento. Como dissemos, cada uma faz a contagem do número de vezes que o padrão ocorre. Sabemos que threads compartilham o espaço de memória, então seria bem inteligente se fizéssemos com que cada thread incrementasse a mesma variável responsável pela contagem. Mas aí está nosso primeiro problema:

Cada thread pode estar sendo executada em um núcleo de CPU distinto, o que significa que elas estão, de fato, correndo em paralelo. Suponha, agora, que duas encontrem o padrão buscado ao mesmo tempo e decidam incrementar a variável de contagem também simultaneamente.

Lembre-se que no nosso exemplo as duas threads estão fazendo tudo simultaneamente e que, sendo assim, elas lerão o valor acumulado (digamos que seja X).

Ambas farão o incremento desse valor em uma unidade (X+1) e ambas tentarão escrever esse novo valor em memória. Duas coisas podem ocorrer:

A colisão na escrita pode fazer com que uma escrita seja descartada.

Diferenças de microssegundos podem fazer com que as escritas ocorram com uma defasagem infinitesimal. Nesse caso, X+1 seria escrito duas vezes.

Em ambos os casos, o resultado será incorreto (o certo é X+2).

Podemos resolver esse problema se conseguirmos coordenar as duas threads de maneira que, quando uma inicie uma operação sobre a variável, a outra aguarde até que a operação esteja finalizada. Para fazermos essa coordenação será preciso que as threads troquem mensagens, contudo elas são entidades semi-independentes rodando em núcleos distintos da CPU. Não se trata de dois objetos instanciados na mesma aplicação.

SEMÁFOROS

As técnicas para evitar os problemas já mencionados envolvem uso de travas, atomização de operações, semáforos, monitores e outras. Essencialmente, o que buscamos é evitar as causas que levam aos problemas. Por exemplo, ao usarmos uma trava sobre um recurso, evitamos o que é chamado de condição de corrida.

Conceitualmente, o semáforo é um mecanismo que controla o acesso de processos ou threads a um recurso compartilhado. Ele pode ser usado para controlar o acesso a uma região crítica (recurso) ou para sinalização entre duas threads. Por meio do semáforo podemos definir quantos acessos simultâneos podem ser feitos a um recurso. Para isso, uma variável de controle é usada e são definidos métodos para a solicitação de acesso ao recurso e de restituição do acesso após terminado o uso do recurso obtido.

Esse processo acontece da seguinte forma:

Solicitação de acesso ao recurso

Quando uma thread deseja acesso a um recurso compartilhado, ela invoca o método de solicitação de acesso. O número máximo de acessos ao recurso é dado pela variável de controle.

Controle de acessos

Quando uma solicitação de acesso é feita, se o número de acessos que já foi concedido for menor do que o valor da variável de controle, o acesso é permitido e a variável é decrementada. Se o acesso for negado, a thread é colocada em espera numa fila.

Liberação do recurso obtido

Quando uma thread termina de usar o recurso obtido, ela invoca o método que o libera e a variável de controle é incrementada. Nesse momento, a próxima thread da fila é despertada para acessar o recurso.

Desde a versão 5, Java oferece uma implementação de semáforo por meio da classe “Semaphore” (ORACLE AMERICA INC., s.d.). Os métodos para acesso e liberação de recursos dessa classe são:

Clique nas barras para ver as informações.

Acquire ()

Solicita acesso a um recurso ou uma região crítica, realizando o bloqueio até que uma permissão de acesso esteja disponível ou a thread seja interrompida.

Release ()

Método responsável pela liberação do recurso pela thread.

Em Java, o número de acessos simultâneos permitidos é definido pelo construtor na instanciação do objeto.

```
public class Exemplo
{
  // (...)
  Semaphore sem = new Semaphore ( 50 , true ); //Define até 50 acessos e o uso de FIFO
  sem.acquire ( ); //Solicita 1 acesso
  ... // Região crítica
  sem.release ( ); //Libera o acesso obtido
  ... //Código não crítico
  sem.acquire ( 4 ); //Solicita 4 acessos
  ... // Região crítica
  sem.release ( 4 ); //Libera os 4 acessos obtidos
  ... //Código não crítico
}
```

Exemplo

Imagine que temos um semáforo que permite apenas um acesso à região crítica e que essa permissão de acesso foi concedida a uma thread (thread 0). Em seguida, uma nova permissão é solicitada, mas como não há acessos disponíveis, a thread (thread 1) é posta em espera. Quando a thread 0 liberar o acesso, se uma terceira thread (thread 2) solicitar permissão de acesso antes de que a thread 1 seja capaz de fazê-lo, ela obterá a permissão e bloqueará a thread 1 novamente.

Semáforo

Não verifica se a liberação de acesso veio da mesma thread que a solicitou.

Mutex

Faz a verificação para garantir que a liberação veio da thread que a solicitou.

MONITORES

Vamos retornar ao problema hipotético apresentado no início do módulo. Nele, precisamos proceder ao incremento de uma variável, garantindo que nenhuma outra thread opere sobre ela antes de terminarmos de incrementá-la.

Em outras palavras, estamos transformando a operação de incremento em uma operação atômica, ou seja, indivisível. Uma vez iniciada, nenhum acesso à variável será possível até que a operação termine.

Para casos como esse, a linguagem Java provê um mecanismo chamado de monitor. Um monitor é uma implementação de sincronização de threads que permite:

Exclusão mútua entre threads

No monitor, a exclusão mútua é feita por meio de um mutex (lock) que garante o acesso exclusivo à região monitorada.

Cooperação entre threads

A cooperação implica que uma thread possa abrir mão temporariamente do acesso ao recurso, enquanto aguarda que alguma condição ocorra. Para isso, um sistema de sinalização entre as threads deve ser provido.

Classes, objetos ou regiões de códigos monitorados são ditos “thread-safe”, indicando que seu uso por threads é seguro.

A linguagem Java implementa o conceito de monitor por meio da palavra reservada “synchronized”. Esse termo é utilizado para marcar regiões críticas de código que, portanto, deverão ser monitoradas. Em Java, cada objeto está associado a um monitor, o qual uma thread pode travar ou destravar. O uso de “synchronized” pode ser aplicado a um método ou a uma região menor de código.

Quando um método sincronizado (synchronized) é invocado, ele automaticamente dá início ao travamento da região crítica. A execução do método não começa até que o bloqueio tenha sido garantido. Uma vez terminado, mesmo que o método tenha sido encerrado anormalmente, o travamento é liberado. É importante perceber que quando se trata de um método de instância, o travamento é feito no monitor associado àquela instância. Em oposição, métodos “static” realizam o travamento do monitor associado ao objeto “Class”, representativo da classe na qual o método foi definido (ORACLE AMERICA INC., s.d.).

Em Java, todo objeto possui um “wait-set” associado que implementa o conceito de conjunto de threads. Essa estrutura é utilizada para permitir a cooperação entre as threads, fornecendo os seguintes métodos:

wait()

Adiciona a thread ao conjunto “wait-set”, liberando a trava que aquela thread possui e suspendendo sua execução. A MVJ mantém uma estrutura de dados com as threads adormecidas que aguardam acesso à região crítica do objeto.

notify()

Acorda a próxima thread que está aguardando na fila e garante o acesso exclusivo à thread despertada. Nesse momento a thread é removida da estrutura de espera.

notifyAll()

Faz basicamente o mesmo que o método notify (), mas acordando e removendo todas as threads da estrutura de espera. Entretanto, mesmo nesse caso apenas uma única thread obterá o travamento do monitor, isto é, o acesso exclusivo à região crítica.

OBJETOS IMUTÁVEIS

Um objeto é considerado imutável quando seu estado não pode ser modificado após sua criação. Objetos podem ser construídos para ser imutáveis, mas a própria linguagem Java oferece classes de objetos com essa característica. O

tipo “String” é um caso de classe que define objetos imutáveis. Caso sejam necessários objetos string mutáveis, Java disponibiliza duas classes, StringBuffer e StringBuilder, que permitem criar objetos do tipo String mutáveis (SCHILDT, 2014).

O conceito de objeto imutável pode parecer uma restrição problemática, mas na verdade há vantagens. Uma vez que já se sabe que o objeto não pode ser modificado, o código se torna mais seguro e o processo de coleta de lixo mais simples. Já a restrição pode ser contornada ao criar um novo objeto do mesmo tipo que contenha as alterações desejadas.

Se um objeto não pode ter seu estado alterado, não há risco de que ele se apresente num estado inconsistente, ou seja, que tenha seu valor lido durante um procedimento que o modifica, por exemplo. Acessos múltiplos de threads também não poderão corrompê-lo. Assim, objetos imutáveis são “thread-safe”.

Em linhas gerais, se você deseja criar um objeto imutável, métodos que alteram o estado do objeto (set) não devem ser providos. Também deve-se evitar que alterações no estado sejam feitas de outras maneiras. Logo, todos os campos devem ser declarados privados (private) e finais (final). A própria classe deve ser declarada final ou ter seu construtor declarado privado.

Aplicar a implementação de threads em Java

CLASSE THREAD E SEUS MÉTODOS

A API Java oferece diversos mecanismos que suportam a programação paralela.

A classe é bem documentada na API e apresenta uma estrutura com duas classes aninhadas (“State” e “UncaughtExceptionHandler”), campos relativos à prioridade (MAX_PRIORITY, NORM_PRIORITY e MIN_PRIORITY) e vários métodos (ORACLE AMERICA INC., s.d.).

getPriority () e setPriority (int pri)

O método “getPriority ()” devolve a prioridade da thread, enquanto “setPriority (int pri)” é utilizado para alterar a prioridade da thread. Quando uma nova thread é criada, ela herda a prioridade da thread que a criou. Isso pode ser alterado posteriormente pelo método “setPriority (int pri)”, que recebe como parâmetro um valor inteiro correspondente à nova prioridade a ser atribuída. Observe, contudo, que esse valor deve estar entre os limites mínimo e máximo, definidos respetivamente por MIN_PRIORITY e MAX_PRIORITY.

getState ()

Outro método relevante é o “getState ()”. Esse método retorna o estado no qual a thread se encontra. Embora esse método possa ser usado para monitorar a thread, ele não serve para garantir a sincronização. Isso acontece porque o estado da thread pode se alterar entre o momento em que a leitura foi realizada e o recebimento dessa informação pelo solicitante, de maneira que a informação se torna obsoleta.

getId () e getName ()

Os métodos “getId ()” e “getName ()” são utilizados para retornar o identificador e o nome da thread. O identificador é um número do tipo “long” gerado automaticamente no momento da criação da thread, e permanece inalterado até o fim de sua vida. Apesar de o identificador ser único, ele pode ser reutilizado após a thread finalizar.

setName ()

O nome da thread pode ser definido em sua criação, por meio do construtor da classe, ou posteriormente, pelo método “setName ()”. O nome da thread é do tipo “String” e não precisa ser único. Na verdade, o sistema se vale do identificador e não do nome para controlar as threads. Da mesma forma, o nome da thread pode ser alterado durante seu ciclo de vida.

currentThread ()

Caso seja necessário obter uma referência para a thread corrente, ela pode ser obtida com o método “currentThread ()”, que retorna uma referência para um objeto Thread. A referência para o próprio objeto (“this”) não permite ao programador acessar a thread específica que está em execução.

join ()

Para situações em que o programador precise fazer com que uma thread aguarde outra finalizar para prosseguir, a class “Thread” possui o método “join ()”. Esse método ocorre em três versões, sendo sobrecarregado da seguinte forma: “join ()”, “join (long millis)” e “join (long millis, int nanos)”. Suponha que uma thread “A” precisa aguardar a thread “B” finalizar antes de prosseguir seu processamento. A invocação de “B.join ()” em “A” fará com que “A” espere (“wait”) indefinidamente até que “B” finalize. Repare que, se “B” morrer, “A” permanecerá eternamente aguardando por “B”.

Uma maneira de evitar que “A” se torne uma espécie de “zumbi” é especificar um tempo limite de espera (timeout), após o qual ela continuará seu processamento, independentemente de “B” ter finalizado. A versão “join (long millis)” permite definir o tempo de espera em milissegundos, e a outra, em milissegundos e nanossegundos. Nas duas situações, se os parâmetros forem todos zero, o efeito será o mesmo de “join ()”.

run ()

É o método principal da classe Thread. Esse método modela o comportamento que é realizado pela thread quando ela é executada e, portanto, é o que dá sentido ao emprego da thread. Os exemplos mostrados nos códigos das Threads A e B ressaltam esse método sendo definido numa classe que implementa uma interface Runnable. Mas a situação é a mesma para o caso em que se estende a classe Thread.

setDaemon ()

O método “setDaemon ()” é utilizado para tornar uma thread, um daemon ou uma thread de usuário. Para isso, ele recebe um parâmetro do tipo “boolean”. A invocação de “setDaemon (true)” marca a thread como daemon. Se o parâmetro for “false”, a thread é marcada como uma thread de usuário. Essa marcação deve ser feita, contudo, antes de a thread ser iniciada (e após ter sido criada). O tipo de thread pode ser verificado pela invocação de “isDaemon ()”, que retorna “true” se a thread for do tipo daemon.

sleep (long millis)

É possível suspender temporariamente a execução de uma thread utilizando o método “sleep (long millis)”. A invocação de “sleep (long millis)” faz com que a thread seja suspensa pelo período de tempo em milissegundos equivalente a “millis”. A versão sobrecarregada “sleep (long millis, int nanos)” define um período em milissegundos e nanossegundos. Porém, questões de resolução de temporização podem afetar o tempo que a thread permanecerá suspensa de fato. Isso depende, por exemplo, da granularidade dos temporizadores e da política do escalonador.

start () e stop ()

Talvez o método “start ()” seja o mais relevante depois de “run ()”. Esse método inicia a execução da thread, que passa a executar “run ()”. O método “start ()” deve ser invocado após a criação da thread e é ilegal invocá-lo novamente em uma thread em execução. Há um método que para a execução da thread (“stop ()”), mas conforme a documentação, esse método está depreciado desde a versão 1.2. O seu uso é inseguro devido a problemas com monitores e travas e, em consequência disso, deve ser evitado. Uma boa discussão sobre o uso de “stop ()” pode ser encontrada nas referências deste material.

yield ()

O último método que abordaremos é o “yield ()”. Esse método informa ao escalonador do sistema que a thread corrente deseja ceder seu tempo de processamento. Ao ceder tempo de processamento, busca-se otimizar o uso da CPU, melhorando a performance. Contudo, cabem algumas observações: primeiramente, quem controla o agendamento de

threads e processos é o escalonador do sistema, que pode perfeitamente ignorar “yield ()”. Além disso, é preciso bom conhecimento da dinâmica dos objetos da aplicação para se extrair algum ganho pelo seu uso. Tudo isso torna o emprego de “yield ()” questionável.

TEMA 5 – INTEGRAÇÃO COM BANCO DE DADOS EM JAVA

MIDDLEWARE

Com diferentes componentes para acesso e modelos de programação heterogêneos, a probabilidade de ocorrência de erros é simplesmente enorme, levando à necessidade de uma camada de software intermediária, responsável por promover a comunicação entre o front-end e o back-end.

Foi definido o termo middleware para a classificação desse tipo de tecnologia, que permite integração de forma transparente e mudança de fornecedor com pouca ou nenhuma alteração de código.

No ambiente Java, temos o JDBC (Java Database Connectivity) como middleware para acesso aos diferentes tipos de bancos de dados. Ele permite que utilizemos produtos de diversos fornecedores, sem modificações no código do aplicativo, sendo a consulta e manipulação de dados efetuadas por meio de comandos SQL (Structured Query Language) em meio ao código Java.

JAVA DATABASE CONNECTIVITY (JDBC)

Com o banco criado, podemos codificar o front-end em Java, utilizando os componentes do middleware **JDBC**, os quais são disponibilizados com a importação do pacote **java.sql**, tendo como elementos principais as classes DriverManager, Connection, Statement, PreparedStatement e ResultSet.

Existem quatro tipos de **drivers** JDBC, os quais são apresentados no quadro seguinte:

1 - JDBC-ODBC Bridge

Faz a conexão por meio do ODBC.

2 - JDBC-Native API

Utiliza o cliente do banco de dados para a conexão.

3 - JDBC-Net

Acessa servidores de middleware via Sockets, em uma arquitetura de três camadas.

4 - Pure Java

Com a implementação completa em Java, não precisa de cliente instalado. Também chamado de Thin Driver.

As consultas ao banco são efetuadas utilizando executeQuery, enquanto comandos para manipulação de dados são executados por meio de executeUpdate.

Para o comando de seleção existe mais um detalhe, que seria a recepção da consulta em um ResultSet. Ao efetuar a consulta, o ResultSet fica posicionado antes do primeiro registro, na posição BOF (Beginning of File), e com o uso do comando next podemos mover para as posições seguintes, até atingir o final da consulta, na posição EOF (End of File).

Um recurso muito interessante, oferecido por meio do componente PreparedStatement, é a definição de comandos SQL parametrizados, os quais são particularmente úteis no tratamento de datas, já que os bancos de dados apresentam interpretações diferentes para esse tipo de dado, e quem se torna responsável pela conversão para o formato correto é o próprio JDBC.

O uso de parâmetros facilita a escrita do comando SQL, sem a preocupação com o uso de apóstrofe ou outro delimitador, além de representar uma proteção contra os ataques do tipo SQL Injection.

Para definir os parâmetros, utilizamos pontos de interrogação, os quais assumem valores posicionais, a partir de um, o que é um pouco diferente da indexação dos vetores, que começa em zero.

Descrever o modelo de persistência baseado em mapeamento objeto-relacional

Bases de dados cadastrais seguem um modelo estrutural baseado na álgebra relacional e no cálculo relacional, áreas da matemática voltadas para a manipulação de conjuntos, apresentando ótimos resultados na implementação de consultas. Mesmo apresentando grande eficiência, a representação matricial dos dados, com registros separados em campos e apresentados sequencialmente, não é a mais adequada para um ambiente de programação orientado a objetos.

Torna-se necessário efetuar uma conversão entre os dois modelos de representação, o que é feito com a criação de uma classe de entidade para expressar cada tabela do banco de dados, à exceção das tabelas de relacionamento. Segundo a conversão efetuada, os atributos da classe serão associados aos campos da tabela, e cada registro poderá ser representado por uma instância da classe.

A conversão de tabelas, e respectivos registros, em coleções de entidades é uma técnica conhecida como Mapeamento Objeto-Relacional.

DATA ACCESS OBJECT (DAO)

Agora que sabemos lidar com as operações sobre o banco de dados e definimos uma entidade para representar um registro da tabela, seria interessante organizar a forma de programar, pois é fácil imaginar a dificuldade para efetuar manutenção em sistemas com dezenas de milhares de linhas de código Java, contendo diversos comandos SQL espalhados ao longo das linhas.

Baseado na observação anterior, foi desenvolvido um padrão de desenvolvimento com o nome DAO (Data Access Object), cujo objetivo é concentrar as instruções SQL em um único tipo de classe, permitindo o agrupamento e a reutilização dos diversos comandos relacionados ao banco de dados.

Normalmente, temos uma classe DAO para cada classe de entidade relevante para o sistema.

Como a codificação das operações sobre o banco apresenta muitos trechos comuns para as diversas entidades do sistema, podemos concentrar as similaridades em uma classe de base, e derivar as demais, segundo o princípio de herança. A utilização de elementos genéricos também será um facilitador na padronização das operações comuns sobre a tabela, como inclusão, exclusão, alteração e consultas.

JAVA PERSISTENCE ARCHITECTURE (JPA)

Devido à padronização oferecida com a utilização de mapeamento objeto-relacional e classes DAO, e considerando a grande similaridade existente nos comandos SQL mais básicos, foi simples chegar à conclusão de que seria possível criar ferramentais para a automatização de diversas tarefas referentes à persistência. Surgiram frameworks de persistência, para as mais diversas linguagens, como Hibernate, Entity Framework, Pony e Speedo, entre diversos outros.

Atualmente, no ambiente Java, concentramos os frameworks de persistência sob uma arquitetura própria, conhecida como Java Persistence Architecture, ou JPA.

O modelo de programação anotado é adotado na arquitetura, simplificando muito o mapeamento entre objetos do Java e registros do banco de dados.

As anotações utilizadas são bastante intuitivas, como Entity transformando a classe em uma entidade, Table selecionando a tabela na qual os dados serão escritos, e Column associando o atributo a um campo da tabela. As características específicas dos campos podem ser mapeadas por meio de anotações como Id, que determina a chave primária, e Basic, na qual o parâmetro optional permite definir a obrigatoriedade ou não do campo.

Também é possível definir consultas em sintaxe JPQL, uma linguagem de consulta do JPA que retorna objetos, ao invés de registros. As consultas em JPQL podem ser criadas em meio ao código do aplicativo, ou associadas à classe com anotações NamedQuery.

Toda a configuração da conexão com banco é efetuada em um arquivo no formato XML com o nome persistence. temos uma unidade de persistência com o nome ExemploJavaDB01PU, sendo a conexão com o banco de dados definida

por meio das propriedades url, user, driver e password, com valores equivalentes aos que são adotados na utilização padrão do JDBC. Também temos a escolha das entidades que serão incluídas no esquema de persistência, no caso apenas Aluno, e do provedor de persistência, em que foi escolhido Eclipse Link, mas que poderia ser trocado por Hibernate ou Oracle Top Link, entre outras opções.

Tudo que precisamos fazer é instanciar um EntityManager a partir da unidade de persistência, recuperar o objeto Query e, a partir dele, efetuar a consulta por meio do método getResultList, o qual já retorna uma lista de entidades, sem a necessidade de programar o preenchimento dos atributos.