**Artigo publicado na revista:** Easy Java Magazine
**Edição de número:** 26
**Publicada em:** Março/2013

# Entendendo os modificadores Java

## Saiba como aplicar os modificadores final, native, static, synchronized e volatile

***Do que se trata o artigo:***

Este artigo apresenta os modificadores **final**, **native**, **static**, **synchronized** e **volatile** e exemplifica por meio de códigos as peculiaridades de cada um.

***Em que situação o tema é útil:***

O bom uso dos modificadores **final**, **native**, **static**, **synchronized** e **volatile** é importante para definir uma implementação coerente com as especificações de projeto, determinando como serão os acessos a classes, métodos e/ou variáveis. Desse modo, pode-se concretizar conceitos como, por exemplo, o encapsulamento e a herança do paradigma orientado a objetos.

***Entendendo os modificadores Java:***

Este artigo apresentará os modificadores **final**, **native**, **static**, **synchronized** e **volatile** da linguagem Java, os quais permitem adequar as implementações a uma diversidade de propósitos como, por exemplo, auxiliar na adequação do código a um determinado padrão de projeto e também no reaproveitamento de código legado de outras linguagens.

Os modificadores da linguagem Java têm o objetivo de tornar as implementações mais adequadas a uma diversidade de propósitos. Uma característica geral dos modificadores é que são definidos sempre em palavras minúsculas, tais como **final**, **native**, **static**, **synchronized** e **volatile**, os quais serão tratados neste artigo.

Por exemplo, o uso do especificador **final** pode conferir segurança ao código e maior velocidade de execução. O especificador **static** confere economia de memória e a possibilidade de troca de informações entre os objetos da classe, entre outras funcionalidades que serão apresentadas mais adiante neste artigo.

Além disso, é importante ter em mente que o uso de um modificador pode ocorrer em conjunto com outro e, o uso em conjunto de dois modificadores pode gerar características próprias, as quais não eram presentes no uso individual dos modificadores.

O uso do modificador **static** em um atributo de classe, por exemplo, torna esse atributo possível de ser inicializado em qualquer parte da classe, mas quando recebe um valor, ele é compartilhado com todos os objetos da classe. Isso significa que os atributos estáticos de uma classe são compartilhados por todas as instâncias dessa classe. O modificador **final** por sua vez, torna o atributo da classe uma constante e, só pode ser inicializado uma única vez, ou diretamente na declaração da classe ou no método construtor. Quando um atributo recebe os modificadores **static** e **final**, sua inicialização apresenta uma alteração distinta, o atributo não poderá mais ser inicializado no método construtor, sendo possível sua inicialização apenas diretamente na declaração do atributo ou no bloco de inicialização estático.

Logo, é importante conhecer as características e particularidades da aplicação dos modificadores de forma que eles sejam utilizados corretamente e as classes tenham um comportamento esperado. Com base nisso, este artigo trata dos modificadores **final**, **native**, **static**, **synchronized** e **volatile**, suas características, formas de uso e exemplos.

## Modificador final

Inicialmente, iremos tratar do modificador **final**, o qual pode ser aplicado em classes, métodos e atributos. Este modificador atribui uma propriedade específica ao membro (atributo ou método) que o recebe, ou mesmo à própria classe.

No caso das classes, o modificador **final** confere terminalidade ao processo de herança, isto é, uma classe ao receber o modificador **final** representa que chegou ao nível máximo de especialização e não poderá mais ser especializada. Um exemplo clássico de classe **final** é a **java.lang.String**. A classe **String**, por se tratar de um tipo de dado, foi definida como uma classe **final**. Outro motivo para o uso do modificador **final** é a segurança, dado que quando a classe for identificada como sendo a **String**, se trata da classe **java.lang.String** propriamente dita e não de uma possível classe herdeira, a qual pode ter sofrido alterações e apresentar funcionalidades alteradas.

Por sua vez, quando aplicado aos métodos, o modificador **final** garante que este não será sobrescrito, o que implica que o método declarado como **final** terá o seu protótipo mantido tal qual foi definido e, quando for chamado por um dos objetos da classe (ou de classes herdeiras), seu código é que será executado. Dessa forma, nenhuma classe herdeira será capaz de sobrescrever um método definido como **final**.

Essa propriedade confere segurança ao código implementado, evitando a programação de subclasses que poderiam ter o mesmo protótipo do método, mas com implementações diferentes. Mais especificamente, imagine que estamos tratando de um método que confere a autenticação de usuários do sistema. Se este método for sobrescrito por um código malicioso em alguma classe herdeira, poderia ser criada uma quebra de segurança na autenticação, conferindo acesso a usuários não autorizados. Outra vantagem é o desempenho de execução, dado que as chamadas a métodos **final** são substituídas por suas definições, isto é, pelo código contido na definição do método (técnica de inclusão de código *inline*). Neste caso, o compilador pode substituir a chamada a um método **final** pelo código (corpo) do método, evitando o desvio de fluxo em sua execução. Portanto, se um método possuir uma especificação bem definida e não for sofrer especializações/redefinições pelas classes herdeiras, é aconselhável que o mesmo receba o modificador **final** por razões de segurança e desempenho.

Finalmente, quando este modificador é aplicado aos atributos, os mesmos passam a ser constantes, ou seja, uma vez o atributo inicializado, não poderá ter o seu valor alterado. Qualquer tentativa de modificá-lo gera um erro de compilação. Embora aparentemente simples, o modificador **final**, quando aplicado aos atributos, possui algumas propriedades que valem a pena destacar. A atribuição de valor a um atributo **final** pode ocorrer diretamente em sua declaração, mas também é possível ter atributos finais que recebem valores nos construtores da classe e, quando isso ocorre, eles são chamados "*blank final variable*". Essa propriedade confere maior flexibilidade na definição das constantes de uma classe. Uma restrição é que a definição deve obrigatoriamente ocorrer em uma das duas formas possíveis: na declaração ou no método construtor. Neste caso, se uma classe possuir vários métodos construtores, o atributo **final** deverá ser inicializado em todos os métodos construtores.

Outra particularidade do operador **final**, quando aplicado aos atributos, é que apenas os tipos primitivos (**byte**, **short**, **int**, **long**, **char**, **float**, **double** e **boolean**) permanecem com seus valores constantes. Sua aplicação aos atributos que sejam objetos ou vetores também é permitida, no entanto, nesses casos, apenas a referência ao objeto ou ao vetor é fixa, ou seja, os valores dos atributos do objeto **final** ou os valores contidos nas posições do vetor **final** podem ser alterados, mas impede que sejam instanciados novamente.

O modificador **final** também pode ser aplicado aos parâmetros de um método, os quais não poderão ser modificados no escopo do método, protegendo e garantindo que os valores/objetos recebidos como parâmetro terão seus valores/referências mantidas no interior do método durante a execução.

### *Utilização do modificador final*

Com o objetivo de exemplificar as características apresentadas anteriormente, o código da **Listagem 1** exibe o uso do modificador **final** em classes, métodos, atributos e em parâmetros de um método.

Na linha 1 é declarada a classe **ClasseFinal** com o uso do modificador **final**. Nas linhas 2, 3, 4 e 5 são declarados os atributos finais de classe. Na linha 4 temos também a inicialização/instanciação do objeto **botao**. Logo, não poderá haver nenhuma outra instanciação desse objeto na classe. Assim, qualquer instrução como **botao = new JButton("rotulo do botao")** será entendida pelo compilador como um erro.

Os atributos finais que não foram inicializados em sua declaração devem ser inicializados no método construtor, como descrito nas linhas 8, 9, 10, 14, 15 e 16. Neste exemplo existem dois métodos construtores, nos quais deverão ser inicializados todos os atributos ainda não inicializados. Isto se faz necessário dado que não é conhecido *a priori* qual será o método construtor que será utilizado para a criação do objeto da classe. Se houver qualquer tentativa de uma nova atribuição de valores (tipos primitivos) ou reinstanciação (objetos ou vetores) de atributos finais em qualquer outra parte do código, é gerado um erro de compilação.

Na linha 27 temos um exemplo de que é possível alterar os valores dos objetos, mesmo que eles sejam declarados como atributos finais. A instrução **botao.setText("novo rotulo do botao")** causa a alteração do rótulo do objeto e, consequentemente, do valor armazenado por ele. Na linha 25 temos outro exemplo, dessa vez aplicado ao vetor, o qual tem seus valores alterados.

Já na linha 19 temos um exemplo de uso do modificador **final** no parâmetro do método. Neste caso, a posição recebida não poderá ser alterada dentro do escopo do método.

**Listagem 1**. Código da classe ClasseFinal.

```
1.   public final class ClasseFinal{
2.       protected final String nome;
3.       protected final int idade;
4.       protected final JButton botao = new JButton("rotulo do botao");
5.       protected final int[] vetor;
6.
7.       public ClasseFinal(){
8.           idade = 20;
9.           nome = "UTFPR-CP";
10.          vetor = new int[100];
11.      }
12.
13.      public ClasseFinal(int id, String nm, int tamanho) {
14.          i = id;
15.          nome = nm;
16.          vetor = new int[tamanho];
17.      }
18.
19.      public final void Imprime(final int posicao) {
20.          System.out.println(vetor[posicao]);
21.      }
22.
23.      public final void inicializa() {
24.          for (int w = 0; w < vetor.length; w++) {
25.              vetor[w] = w * 9;
26.          }
27.          botao.setText("novo rotulo do botao");
28.      }
29.  }
```

## Modificador native

O Java é uma tecnologia multiplataforma em que *bytecodes* gerados durante o processo de compilação de uma classe podem ser executados em qualquer implementação de máquina virtual Java (JVM). Esse recurso torna a tecnologia portável para qualquer sistema operacional que possua uma JVM. Entretanto, existem casos em que o desenvolvimento de novos sistemas necessita da migração de código legado ou de aplicações com regras de negócio complexas implementadas em outras linguagens de programação. Porém, tal atividade pode levar muito tempo, além do árduo

trabalho e riscos que uma recodificação incorreta pode impactar negativamente na execução de uma aplicação.

Diante desse panorama, o Java oferece o operador **native**, capaz de combinar códigos de outras linguagens como, por exemplo, C/C++, com uma aplicação Java. A utilização desse operador é exclusiva para declaração de protótipos de métodos em classes Java, os quais reutilizarão códigos implementados em outra linguagem. Apesar de passar despercebido por muitos programadores, o modificador **native** torna-se útil quando há a necessidade de reaproveitar programas desenvolvidos em outras linguagens que possuem códigos extensos e/ou de regra de negócio complexa.

Para usar o operador **native** é preciso seguir o padrão de programação *Java Native Interface* (JNI), que permite chamar funções de códigos nativos a partir de uma JVM. É importante destacar que o uso do JNI requer alguns cuidados para que a linguagem Java não perca duas de suas principais vantagens: portabilidade e segurança. Aplicações que fazem uso de métodos nativos ficam dependentes do sistema operacional e do hardware onde o código nativo foi compilado, impedindo a portabilidade da aplicação, pois diferente do Java, muitos compiladores de outras linguagens de programação como C/C++ geram código executável de acordo com o sistema operacional e plataforma de hardware. Além disso, a JVM permite apenas o acesso restrito à memória na qual um objeto tenha direito, enquanto linguagens como C/C++ conferem acesso irrestrito à memória, o que pode causar sérios riscos de segurança durante a execução de uma aplicação.

### *Utilização do modificador native*

Para exemplificar o uso do modificador **native** e do JNI, é apresentado um exemplo para exibir uma frase no dispositivo de saída padrão a partir de uma função **imprimir()**, já implementada anteriormente em C, a partir de um código Java. A **Listagem 2** apresenta a classe **OlaMundo** contendo a declaração de um método nativo implementado na linguagem C.

**Listagem 2**. Código da classe OlaMundo.

```
1.   public class OlaMundo {
2.
3.      private native void imprimir();
4.
5.      public static void main(String args[]) {
6.        new OlaMundo().imprimir();
7.      }
8.
9.      static {
10.       System.loadLibrary("OlaMundo");
11.     }
12.
13.  }
```

Observe que na declaração do método **imprimir()** – linha 3 – foi utilizado o operador **native** para indicar que o método será provido por um código nativo, o qual é executado a partir do método **main()** – linha 5. O bloco estático inicializa o carregamento da biblioteca nativa contendo a implementação do método **imprimir()**. Assim que concluída a implementação da classe **OlaMundo**, podemos compilá-la por meio do comando:

```
javac OlaMundo.java
```

Em seguida, utiliza-se a ferramenta *javah* para gerar o arquivo de cabeçalho baseado em JNI para implementação do método nativo em C:

```
javah –jni OlaMundo
```

O *javah* cria um arquivo de extensão ".h" com o mesmo nome da classe. Tal arquivo contém o protótipo do método nativo, cuja sintaxe é "Java_<nome_classe>_<nome_método>()". Nesse sentido, ao compilar a classe **OlaMundo** utilizando o *javah*, temos o método **imprimir()** da classe Java **OlaMundo** convertido para a função **Java_OlaMundo_imprimir()** no arquivo **OlaMundo.h**, o qual deve ser implementado em C, de acordo com a **Listagem 3**.

**Listagem 3**. Conteúdo do arquivo OlaMundo.c.

```
1. #include <stdio.h>
2. #include <jni.h>
3. #include "OlaMundo.h"
4.
5. JNIEXPORT void JNICALL
6. Java_OlaMundo_imprimir(JNIEnv *env, jobject obj) {
7.   printf("Olá! Invocando um método nativo!");
8.   return;
9. }
```

Pode-se notar que a implementação da função nativa em C, nas linhas 5 a 9 da **Listagem 3**, segue exatamente a assinatura do protótipo do método presente no arquivo *OlaMundo.h* gerado pelo *javah* a partir da classe **OlaMundo**. A inclusão do cabeçalho *jni.h* na linha 2 oferece diferentes tipos de dados não disponíveis na linguagem C/C++ tratados pelo JNI para a passagem de parâmetros às funções de código nativo. Depois de implementado, pode-se compilar o código nativo a partir do diretório onde as classes Java estão presentes da seguinte forma:

```
gcc –I/usr/local/jvm/java-7-openjdk-i386/include –o libOlaMundo.so –shared OlaMundo.c
```

Na instrução acima é passado como parâmetro o diretório do arquivo de cabeçalho *jni*. A opção **–o** determina o nome da biblioteca nativa a ser gerada e deve seguir o formato *lib<nome_argumento_do_método_loadLibrary >.so*. Já a opção **–shared** indica a criação de uma biblioteca compartilhada.

Depois de todos esses passos, é preciso verificar a existência dos arquivos na raiz do diretório do projeto Java: *OlaMundo.java*, *OlaMundo.class*, *OlaMundo.h*, *OlaMundo.c* e *libOlaMundo.so*. Caso todos os arquivos estejam presentes, basta executar a aplicação através do comando ***java –Djava.library.path=. OlaMundo*** e observar o resultado:

```
Olá! Invocando um método nativo!
```

Apesar de esse exemplo ser aparentemente simples, a invocação de código nativo pode envolver diversos aspectos como o mapeamento de tipos de dados entre o Java e o código nativo, conversão de **String** nativa para Java, passagem de parâmetros e retorno de funções de código nativo, acesso a arrays de objetos, entre outros.

## Modificador static

O modificador **static**, em geral, está associado com a definição de atributos nas classes, de forma que os objetos da classe consigam compartilhar informações entre si. No entanto, este modificador também pode ser aplicado a métodos e inicializadores, podendo existir tantos quantos forem necessários, ou seja, não existe uma relação de dependência entre os membros estáticos ou não estáticos da classe.

Quando o operador **static** é aplicado a um atributo da classe, o mesmo passa a ser compartilhado com todos os objetos dessa classe, mas diferente de como ocorre normalmente (com os atributos não estáticos), o atributo passa a ser conhecido como "atributo de classe", sendo que todos os objetos instanciados dessa classe passam a compartilhar o mesmo atributo, similarmente ao que ocorre com variáveis globais em linguagens de programação estruturadas. Entre os exemplos de atributos estáticos definidos na linguagem Java estão o **java.lang.Math.E** (2.71828...) e o famoso **java.lang.Math.PI** (3.14159...), os quais recebem a especificação **public static final**, representando que os atributos têm visibilidade pública, são acessíveis sem a necessidade de existir um objeto da classe e não podem ter seus valores alterados. Por exemplo, a chamada **java.lang.Math.PI** acessa diretamente o valor do atributo estático **PI**.

Outro exemplo muito popular de atributo que recebe a especificação **public static final** é o **out** da classe **System (java.lang.System**), comumente utilizado para imprimir informações no console. O atributo **out** está associado à apresentação de caracteres na saída padrão, ou seja, na tela do monitor,

sendo declarado na classe **System** como **public static final PrintStream out**. A classe **PrintStream** é definida no pacote **java.io.Printstream** e confere a funcionalidade de imprimir diversos valores dados convenientemente, como valores Strings, inteiros, flutuantes, caracteres, entre outros. Por ser um atributo estático de classe, ele pode ser acessado em sua forma direta **System.out**.

Os atributos estáticos normalmente são utilizados na padronização de valores (constantes) dentro do projeto/sistema, visando compartilhamento de informações entre os objetos e para acesso direto aos atributos da classe sem a necessidade de existir algum objeto instanciado dessa classe.

Considerando o uso do modificador **static** em um método de uma classe, esse método é visto como pertencente à classe e não pode ser usado para chamar/usar métodos ou atributos da classe que não sejam estáticos. Como os métodos estáticos não funcionam com uma instância (objeto) da classe, eles só podem acessar membros estáticos (atributos e métodos) da classe ou de outras classes que forem visíveis.

Neste contexto, os métodos estáticos são muito úteis na definição de rotinas utilitárias dentro de um sistema. Um exemplo clássico é o método **public static double random()** da classe **java.lang.Math**, o qual implementa a geração de números pseudoaleatórios que retorna um valor **double** positivo maior ou igual a 0 e menor do que 1. Os valores pseudoaleatórios devolvidos são definidos de acordo com um número, chamado semente, de 48 bits, o qual é aplicado em uma equação para a geração dos números pseudoaleatórios (veja Donald Knuth, The Art of Computer Programming, Volume 2, seção 3.2.1.).

Os métodos estáticos, quando declarados públicos, podem ser acessados diretamente a partir da classe, sem a necessidade de declaração de objetos. Isso quer dizer que um método, para ser acessível sem a necessidade de existir um objeto da classe, além de ser estático, também deve ser público. Se um método for definido como estático e privado, ele será acessível diretamente apenas a partir da própria classe e, se for definido como estático e protegido, será acessível diretamente a partir da classe, suas herdeiras e outras classes do mesmo pacote. A sintaxe para o acesso direto (sem a necessidade de existir um objeto) é simples: **NomedaClasse.nomedoMétodo()**. O método **java.lang.Math.random()**, por exemplo, devolve um valor pseudoaleatório na forma de um tipo **double**.

*Métodos estáticos, por pertencerem à classe não podem ser abstratos, ou seja, o uso dos especificadores **static** e **abstract** são antagônicos.*

O modificador **static** ainda pode ser usado na declaração de blocos de inicialização, que se tratam de trechos de código que serão executados automaticamente quando a classe for carregada em memória. Um dos objetivos desse recurso é quando a inicialização de um atributo estático não pode ocorrer no momento de sua declaração, ou seja, em uma única expressão. Quando existe essa necessidade, recorremos ao uso de um bloco de inicialização estático. Outra aplicação dos blocos de inicialização estáticos é o desenvolvimento de rotinas de pré-configurações e validações, como a verificação da versão da JVM, da versão do sistema operacional, verificação se o acesso a algum dispositivo de rede está disponível, entre outras.

### *Utilização do modificador static*

Com o objetivo de exemplificar as características e funcionalidades apresentadas sobre o modificador **static**, a **Listagem 4** apresenta alguns trechos de código com a utilização deste operador nas diferentes situações supracitadas, como na definição de métodos, atributos e blocos de inicialização.

No código da **Listagem 4**, inicialmente são definidos quatro atributos estáticos nas linhas 2, 3, 4 e 5, os quais são acessíveis diretamente, ou seja, sem a necessidade de existir um objeto da classe. O acesso ao atributo estático é realizado por meio da instrução **Estatica.versaojava**, ou, de uma forma mais geral, **NomedaClasse.nomedoAtributo**. Uma observação importante é que o atributo definido na linha 4 não é estático, dessa forma ele não poderá ser acessado em um contexto estático, como

em um método ou bloco de inicialização estáticos. Nestes casos, para acessar um atributo, é necessária a instanciação de um objeto da classe.

Na linha 7 existe um bloco de inicialização estático, o qual exemplifica uma rotina de pré-configuração da classe. Neste bloco está a verificação do nome do sistema operacional e a versão da máquina virtual disponível. Caso a versão seja mais antiga que a 1.5, a aplicação é finalizada. Como dica, a execução da linha 17 exibe de forma textual as propriedades da classe **System** que podem ser utilizadas em rotinas de pré-configuração, por meio da impressão do método **System.getProperties()**.

Na linha 20 existe outro bloco de inicialização estático que exemplifica a inicialização do atributo estático por meio do sorteio de números pseudoaleatórios. Trata-se de uma rotina que não seria possível realizar diretamente na declaração/instanciação do objeto, dado que envolve a atribuição dos valores para cada elemento do vetor. Logo, a utilização do bloco de inicialização estático é muito adequada.

A linha 25 apresenta a declaração de um método estático. A exemplo dos atributos estáticos, o método também poderá ser acessado diretamente sem a necessidade de um objeto da classe, isto é, por meio da instrução **Estatica.imprime()**.

Por outro lado, os blocos de inicialização estáticos são executados automaticamente, na ordem em que aparecem a partir da declaração da classe, quando a classe é carregada em memória pela JVM. Logo, os blocos de inicialização estáticos são especialmente importantes para inicializar códigos como pré-configurações, verificações de compatibilidade e em itens de segurança como permissões de acesso à aplicação por usuários. Deste modo, se um usuário não estiver habilitado a utilizar a aplicação, isso pode ser codificado dentro de um bloco de inicialização estático em uma classe.

**Listagem 4.** Código da classe Estatica.

```
1.   public class Estatica{
2.      public static String SO;
3.      public static String versaojava;
4.      public int num = 0;
5.      public static double[] aleatorios = new double[10];
6.
7.      static {
8.          SO = System.getProperty("os.name");
9.          System.out.println(SO);
10.         versaojava = System.getProperty("java.version");
11.         System.out.println(versaojava);
12.         float versao = Float.valueOf(versaojava.substring(0, 3));
13.         if (versao < 1.5){
14.             System.out.println ("Instale uma versão 1.5 ou mais recente da JVM.");
15.             System.exit(1);
16.         }
17.         System.out.println(System.getProperties());
18.     }
19.
20.     static {
21.         for (int i = 0; i < aleatorios.length; i++)
22.             aleatorios[i] = Math.random();
23.     }
24.
25.     public static void Imprime() {
26.         System.out.println("metodo estatico Imprime sendo executado");
27.         for (int i = 0; i < aleatorios.length; i++)
28.             System.out.println(aleatorios[i]);
29.     }
30.  }
```

## Modificador synchronized

É conhecido que a linguagem Java disponibiliza a execução multitarefa por meio da classe **java.lang.Thread**, ou simplesmente pela implementação da interface **java.lang.Runnable**. Uma classe destinada à execução multitarefa deve implementar a interface **Runnable**. A própria classe **Thread** implementa essa interface, a qual garantirá, por sua vez, a implementação obrigatória do método **public void run()**.

A possibilidade de várias tarefas serem disparadas simultaneamente em um mesmo aplicativo pode exigir um pouco mais de cuidado na programação. Esse cuidado em geral é devido à existência de

partes do código que exijam atomicidade. Em outras palavras, algumas tarefas podem possuir uma dependência de execução, levando a um cenário que é desejável que todas as tarefas sejam executadas ou nenhuma delas. Em se tratando de sistemas de informação, um exemplo clássico é a transferência, seja ela de valores ou arquivos. Tomaremos, por exemplo, uma transferência bancária de valores, na qual uma das contas deverá ser debitada e uma outra deverá ser creditada. Logo, as duas ações devem ser atômicas. A forma de garantir que uma tarefa execute as duas ações sem ser interrompida por outra tarefa e, além disso, que nenhuma outra tarefa execute a mesma ação enquanto outra tarefa estiver executando essa ação é utilizando o modificador **synchronized**.

Este modificador pode ser utilizado em trechos de código para garantir justamente que apenas uma tarefa irá executar esse trecho a cada vez, sem que haja interrupção. Logo, se houver várias tarefas (threads) em execução em um determinado instante, pode ser que uma das tarefas inicie a execução de um trecho de código que exija atomicidade; logo, o mesmo deve ser executado até o fim sem que a tarefa seja interrompida. Além disso, deve garantir que só uma tarefa execute esse trecho de código por vez. Um exemplo de uso do modificador **synchronized** pode ser visto na **Listagem 5**.

**Listagem 5.** Exemplo do modificador synchronized.

```
synchronized(object)
{
     ...trecho de código
}
```

No exemplo da **Listagem 5**, **object** representa o objeto que o monitor irá aplicar o bloqueio. O monitor está presente em todos os objetos Java, e garante que o bloqueio de um objeto será realizado para uma única tarefa a cada momento. Então a chamada do especificador **synchronized** por uma tarefa em execução irá utilizar o monitor do objeto para bloquear o trecho de código. Sendo que o **object** do exemplo normalmente é substituído pela palavra reservada **this** para se referir ao objeto em execução, isto é, o uso do **this** normalmente irá representar o objeto que está em execução multitarefa naquele instante em que for executar o trecho de código sincronizado. Se for aplicado o modificador **synchronized** em um trecho de código e uma tarefa conseguir o bloqueio do objeto, então todas as demais tarefas que tentarem executar o mesmo trecho de código (já bloqueado) serão colocadas no estado bloqueado.

Outra possibilidade é utilizar o especificador **synchronized** em um método, sinalizando que todo o código contido neste método terá sua execução sincronizada. Métodos sincronizados são equivalentes a um trecho de código sincronizado envolvendo todo o corpo desse método e utilizando o objeto corrente (**this**) da execução multitarefa, como explicado no parágrafo anterior. No entanto, quando é utilizado o especificador **synchronized** nos métodos, o objeto corrente (**this**) é definido automaticamente de forma implícita, ou seja, sem a existência explícita de código para essa definição. Um exemplo pode ser visto na **Listagem 6**.

**Listagem 6.** Exemplo do modificador synchronized aplicado em um método, classe Banco.

```
1.    public class Banco {
2.
3.       public synchronized boolean transfere(Conta c1, Conta c2, float valor) {
4.          if (saque(c1, valor)){
5.              if (deposito(c2, valor)){
6.                 return(true);
7.              }else{
8.                 deposito(c1,valor);
9               }
10.         }
11.      }
12.
13.      public synchronized boolean saque(Conta conta, float valor) {
14.         if (valor <= conta.getSaldo() && conta.valida()) {
15.               conta.debito(valor);
16.               return (true);
17.          }
18.          return (false);
19.      }
20.
```

```
21.      public synchronized boolean deposito(Conta conta, float valor) {
22.          if (conta.valida()) {
23.               conta.credito(valor);
24.               return (true);
25.          }
26.          return (false);
27.      }
28.  }
```

O uso do especificador **synchronized** em um método garante que se duas ou mais tarefas (threads) tentarem executar esse método, como, por exemplo, os métodos **transfere()**, **saque()** e **deposito()** da **Listagem 6**, a primeira tarefa que iniciar a execução terá o bloqueio e executará o seu conteúdo do início ao fim, sem ser interrompida, enquanto as outras tarefas permanecem esperando sua vez de executar esse método. Quando a tarefa que obteve o bloqueio completar a execução do método sincronizado, o método será desbloqueado e outra tarefa que está aguardando iniciará sua execução, bloqueando novamente esse trecho de código.

Esse comportamento é particularmente útil no exemplo do método **transfere()**. Neste caso, quando uma tarefa iniciar a execução, terá o bloqueio e executará os métodos **saque()** e **depósito()** sem ser interrompida, enquanto as outras tarefas permanecem esperando sua vez de executar o método **transfere()**.

## Modificador volatile

A multiprogramação é um tópico bastante discutido e ganhou mais destaque com a vinda dos processadores com tecnologia multicore. Por exemplo, um processador de dois núcleos é capaz de executar dois fluxos de instruções (*threads*) de um processo simultaneamente, oferecendo melhor desempenho computacional. Porém, é preciso estar atento a alguns detalhes do modo como a tecnologia trata o acesso aos recursos do sistema, como a memória.

No Java, quando há instâncias de duas ou mais *threads*, é criada uma memória *cache* local para cada *thread*. Quando múltiplas *threads* compartilham a mesma variável, cada uma copia o valor da variável em sua própria *cache* local. Em qualquer mudança de valor dessa variável, a atualização é realizada no *cache* local, ao invés de ser na variável alocada na memória principal. Por exemplo, em um momento M1, a *thread* A realiza uma alteração na variável x. Em um momento T2, a *thread* B também acessa essa variável, porém a *thread* B não terá conhecimento de qualquer mudança de valor realizado pela *thread* A, causando uma possível inconsistência de dados, pois cada *thread* realizou a alteração apenas em sua *cache* local.

Para evitar esse tipo de situação, o operador **volatile** evita que qualquer alteração de variável compartilhada entre múltiplas threads seja realizada no *cache* local de uma thread. O **volatile** é aplicado exclusivamente em variáveis e, seu uso só terá sentido quando essas forem compartilhadas entre *threads*. Tal operador é pouco difundido entre os programadores e raramente detalhado na literatura. Os desenvolvedores usualmente utilizam o **synchronized** para bloquear o acesso a objetos e métodos compartilhados, enquanto o operador **volatile** permite o bloqueio de variáveis de tipos primitivos e, também, o de objetos.

Quando uma variável **volatile** sofre qualquer mudança de valor por alguma thread, tal alteração é realizada diretamente na memória principal. Em um bloco ou método sincronizado, qualquer alteração de valor de uma variável é realizada na memória cache da JVM. Esta ação, em algumas situações, pode levar a uma inconsistência de dados quando uma thread realiza a leitura em uma variável compartilhada. Para exemplificar o efeito do uso do operador **volatile**, a **Listagem 7** apresenta um exemplo da ausência desse operador em uma variável **num**, a qual é compartilhada entre duas threads. Por outro lado, a **Listagem 8** exibe o mesmo código, porém fazendo uso desse modificador.

**Listagem 7**. Código da classe NonVolatileTest.

```
1.   public class NonVolatileTest extends Thread {
2.      private static int num = 0;
3.
4.      public void run() {
5.         for(int i=0; i<4; i++) {
```

```
6.            if(getName().equals("Thread1")) {
7.               System.out.println("Valor de num (T1): " + num);
8.               num=10;
9.            } else {
10.              System.out.println("Valor do num (T2):" + num);
11.              num=20;
12.           }
13.       }
14.    }
15.
16.    public static void main(String args[]) throws InterruptedException {
17.       NonVolatileTest t = new NonVolatileTest();
18.       t.start();
19.    }
20. }
```

Por se tratar de uma aplicação com múltiplas threads cujo escalonamento é de responsabilidade do sistema operacional, uma possível saída da execução do código da **Listagem 7** é:

```
Valor de num (Thread1): 0
Valor de num (Thread2): 0
Valor de num (Thread2): 20
Valor de num (Thread1): 10
Valor de num (Thread2): 20
Valor de num (Thread1): 10
Valor de num (Thread2): 20
Valor de num (Thread1): 10
```

**Listagem 8**. Código da classe VolatileTest.

```
1.   public class VolatileTest extends Thread {
2.      private volatile static int num = 0;
3.
4.      public void run() {
5.         for(int i=0; i<4; i++) {
6.            if(getName().equals("Thread1")) {
7.               System.out.println("Valor de num (T1): " + num);
8.               num=10;
9.            } else {
10.              System.out.println("Valor do num (T2):" + num);
11.              num=20;
12.           }
13.       }
14.    }
15.
16.    public static void main(String args[]) throws InterruptedException {
17.       VolatileTest t = new VolatileTest();
18.       t.start();
19.    }
20. }
```

Pode-se notar que os valores exibidos pela *Thread1* e pela *Thread2* são 10 e 20, respectivamente. Isto ocorre porque a modificação é realizada apenas no *cache* local das *threads*.

Para que ambas as threads leiam e escrevam na mesma variável **num** alocada na memória principal, é preciso adicionar o modificador **volatile** na declaração da variável. Deste modo, ao executar o código da **Listagem 8**, obtém-se um resultado semelhante a:

```
Valor de num (Thread1): 0
Valor de num (Thread2): 10
Valor de num (Thread2): 20
Valor de num (Thread1): 20
Valor de num (Thread2): 10
Valor de num (Thread1): 20
Valor de num (Thread2): 10
Valor de num (Thread1): 20
```

Com a adição do operador **volatile** à variável **num**, a *Thread2*, em algumas execuções, passa a exibir o valor 10 atribuído pela *Thread1*, pois ambas as threads passam a acessar a mesma variável diretamente na memória principal da JVM.

## Conclusão

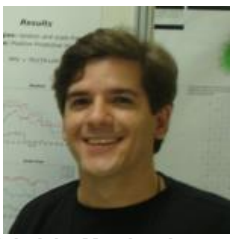
Quando aplicados de modo correto, os modificadores **final**, **native**, **static**, **synchronized** e **volatile** podem otimizar o uso de recursos como, por exemplo, o consumo de memória da JVM. Para evitar problemas de consistência em identificadores cujo valor deve ser constante, o modificador **final** pode suprir tal necessidade, além de evitar que métodos finais sejam sobrescritos ou que classes finais sejam especializadas.

O modificador **native**, por sua vez, oferece a oportunidade de reaproveitamento de código implementado em outras linguagens, porém reduz a capacidade de portabilidade de aplicações Java, pois a execução do código nativo depende do sistema operacional e da arquitetura do processador onde o código nativo foi compilado. Além disso, perde-se no quesito segurança, uma vez que códigos implementados em linguagens como C/C++ permitem o acesso a diferentes regiões da memória principal da máquina que executa a JVM.

O modificador **synchronized** permite o controle de acesso a determinados métodos ou blocos de código, garantindo a integridade dos dados, a sincronização de rotinas e a atomicidade na execução de trechos de código dentro de uma programação multitarefa.

Por fim, **volatile** garante que quaisquer modificações de valor de variáveis e objetos sejam realizadas diretamente na memória principal da JVM, ao invés de ocorrer na cache local da thread, evitando possíveis inconsistências de dados.

Apesar dos modificadores apresentados nesse artigo estarem presentes no cotidiano dos desenvolvedores, conhecer os detalhes de uso e o funcionamento dos mesmos confere ao profissional uma nova perspectiva de programação, tornando o desenvolvimento mais claro e eficiente.



**Fabrício Martins Lopes** (*fabricio@utfpr.edu.br*) é mestre em Informática e doutor em Bioinformática, atuando principalmente nos temas de pesquisa reconhecimento de padrões, bioinformática, processamento de imagens e visão computacional. Atualmente, é docente da Universidade Tecnológica Federal do Paraná atuando nos cursos de Engenharia da Computação e Tecnologia em Análise e Desenvolvimento de Sistemas.

**Henrique Yoshikazu Shishido** (*shishido@utfpr.edu.br*) é mestre em Ciência da Computação tendo como linha de pesquisa a Computação Paralela de Alto Desempenho. Atualmente, é docente da Universidade Tecnológica Federal do Paraná atuando nos cursos de Engenharia da Computação e Tecnologia em Análise e Desenvolvimento de Sistemas.