Programação Concorrente em Java

Prof. Ilo Rivero

ilo.rivero@anhanguera.com

Programação Concorrente em Java

- Na linguagem de programação Java, programação concorrente está principalmente preocupada com threads. No entanto, os processos também são importantes.

- Um sistema de computador normalmente tem muitos processos ativos e threads.

- Ter processadores com múltiplos núcleos de execução aumenta muito a capacidade de um sistema para a execução simultânea de processos e threads - mas a concorrência é possível, mesmo em sistemas simples, sem vários processadores ou núcleos de execução

Programação Concorrente em Java

- Cada thread é associada a uma instância da classe thread . Há duas estratégias básicas para utilizar objetos thread para criar uma aplicação concorrente:

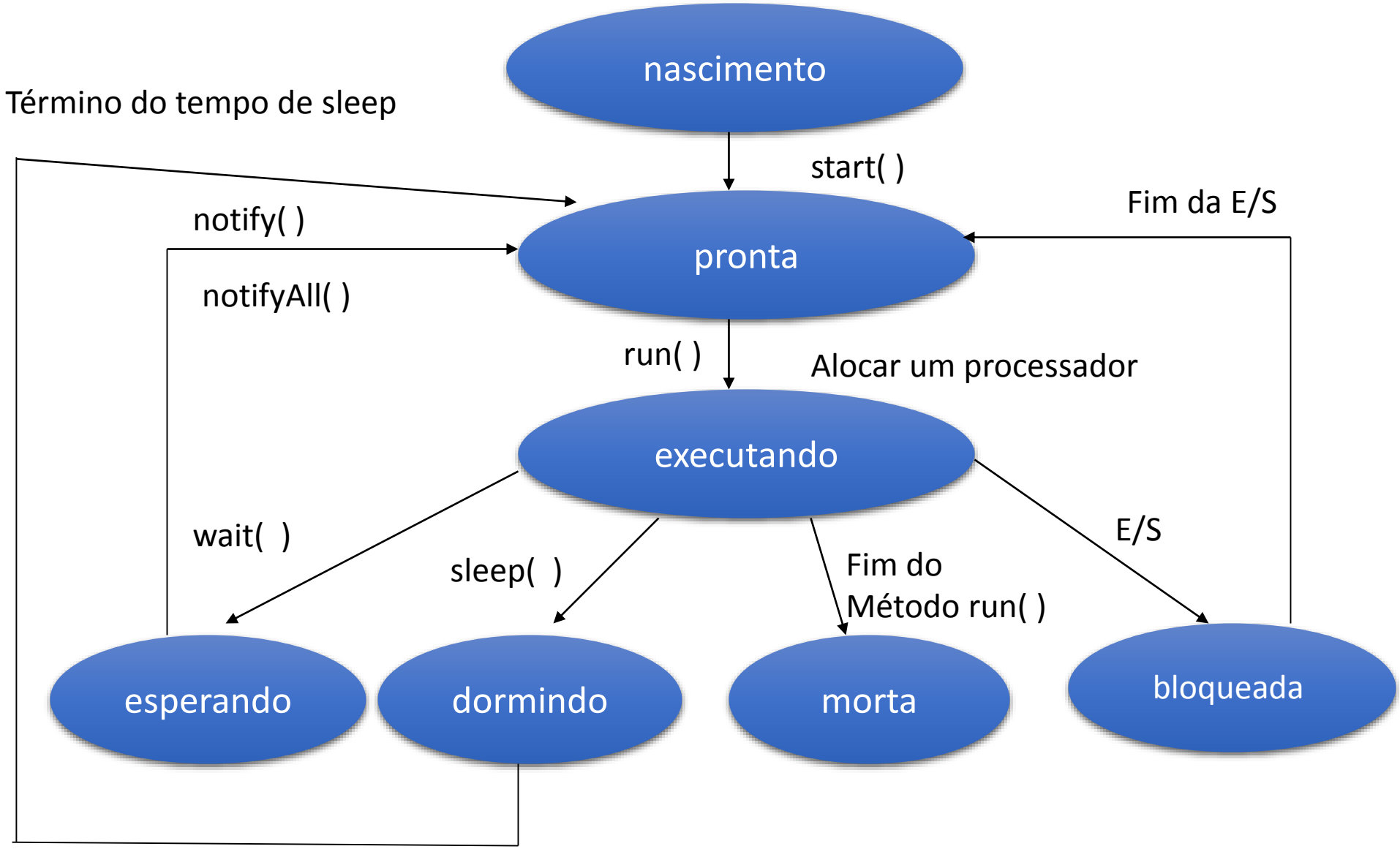
  - Controlar diretamente a criação e gerenciamento de threads, basta instanciar Thread cada vez que o aplicativo precisar iniciar uma tarefa assíncrona

  - Abstrair a thread do resto da sua aplicação, passando as tarefas do aplicativo para um executor

Programação Concorrente em Java

- Threads Java são implementadas pela classe Thread do pacote java.lang.thread:
  - Esta classe implementa um encapsulamento independente de sistema, isto é, a implementação real de threads é oferecida pelo sistema operacional.
  - A classe Thread oferece uma interface unificada para todos os sistemas.
  - Uma mesma implementação do Java Thread pode fazer com que a aplicação proceda de forma diferente em cada sistema, uma vez que a implementação real é feita pelo S.O.

Programação Concorrente em Java



Programação Concorrente em Java

- Alguns Métodos:
  - **run():** é o método que executa as atividades de uma THREAD. Quando este método finaliza, a THREAD também termina.
  - **start():** método que dispara a execução de uma THREAD. Este método chama o método run( ) antes de terminar.
  - **sleep(int x):** método que coloca a THREAD para dormir por x milisegundos.

Programação Concorrente em Java

- Alguns Métodos:
  - **join( ):** método que espera o término da THREAD para qual foi enviada a mensagem para ser liberada
  - **interrupt( ):** método que interrompe a execução de uma THREAD
  - **interrupted( ):** método que testa se uma THREAD está ou não interrompida.
  - **yield( ):** cede o processamento para outra thread

Programação Concorrente em Java

- Alguns Métodos:
  - **Wait() :** utilizado para aguardar até que uma outra thread termine. É uma forma de sincronizar threads
  - **notify( ) e notifyAll( ):** utilizados para acordar a thread e verificar as alterações enquanto ela estava em wait()

Programação Concorrente em Java

- Uma aplicação que cria uma instância de uma Thread deve executar o código que vai rodar naquela thread. Existem duas maneiras de fazer isso:
    - Implementar Runnable: A interface Runnable define um único método, run, que contém o código executado na thread. Runnable é passado para o constructor da Thread , como no exemplo abaixo:

```
public class OlaRunnable implements Runnable {
        public void run() {
                System.out.println("Olá Thread Runnable!");
        }
        public static void main(String args[]) {
                (new Thread(new OlaRunnable())).start();
        }
}
```

Programação Concorrente em Java

- A própria classe Thread implementa Runnable , mas o método RUN não faz nada. Isso permite realizar uma implementação própria do método RUN.

```
public class OlaThread extends Thread{
	public void run() {
		System.out.println("Olá thread!");
	}
	public static void main(String args[]) {
		(new OlaThread()).start();
	}
}
```

Programação Concorrente em Java

- Qual deles implementar?
  - Implementar RUNNABLE é mais geral, pois pode instanciar uma subclasse diferente da Thread
  - Usar Thread é mais fácil implementar, mas não permite muita flexibilidade para controlar as threads
- RUNNABLE permite maior controle e flexibilidade para controlar o funcionamento das threads

Programação Concorrente em Java

- THREAD.SLEEP
  - Permite a interrupção da thread. Faz a thread atual parar durante um tempo específico.
  - Isso é útil para permitir ao processador executar outras threads da mesma aplicação ou de outras que estiverem sendo executadas ao mesmo tempo
  - Contudo, o tempo especificado da thread (em milissegundos) não é preciso, pois a thread pode ser interrompida pelo S.O.
  - O exemplo a seguir é utilizado em uma aplicação com thread simples

Programação Concorrente em Java

```
public class Contagem  {
    public static void main(String args[])
        throws InterruptedException {
        String contagem[] = {
            "Um", "Dois", "Três", “Quatro", "Cinco", "Seis", "Sete", "Oito","Nove", "Dez!"
        };

        for (int i = 0;
             i < contagem.length;
             i++) {
            //Pausa a thread por 1 segundo
            Thread.sleep(1000);
            //Imprime uma mensagem
            System.out.println(contagem[i]);
        }
    }
}
```

Programação Concorrente em Java

- Nomeando Threads

```
public class NomeandoThreads {
  public static void main(String[] args){
    System.out.println(Thread.currentThread().getName());
    for(int i=0; i<100; i++){
      new Thread("" + i){
        public void run(){
          System.out.println("Thread: " + getName() + " rodando");
        }
      }.start();
    }
  }
}
```

Programação Concorrente em Java

- Threads Concorrentes

```
class ImprimirThread_1 implements Runnable {
String str;
        public ImprimirThread_1(String str) {
                this.str = str;
        }
        public void run() {
                for(int i=0;i<100;i++)
                System.out.print(str);
                }
        }
class TesteConcorrente {
        public static void main(String Args[]) {
                new Thread(new ImprimirThread_1("A")).start();
                new Thread(new ImprimirThread_1("B")).start();
        }
}
```

Programação Concorrente em Java

- Liberando uma thread para outra: Yield()

```
public class MetodoYield {
        public static void main(String Args[]) {
                new Thread(new ImprimirThread_2("A")).start();
                new Thread(new ImprimirThread_2("B")).start();
                }
        }
class ImprimirThread_2 implements Runnable {
        String str;
        public ImprimirThread_2(String str) {
                this.str = str;
        }
        public void run() {
        for(;; ) {
                for(;;) {
                System.out.print(str);
                Thread.currentThread();
                Thread.yield();
} } } }
```

Programação Concorrente em Java

- Prioridade de Threads

- Cada thread possui uma prioridade de execução que vai de Thread. MIN_PRIORITY (igual a 1) a Thread.MAX_ PRIORITY (igual a 10).

  - Importante: uma thread herda a prioridade da thread que a criou.

- O algoritmo de escalonamento sempre deixa a thread (runnable) de maior prioridade executar.

  - A thread de maior prioridade preempta as outras threads de menor prioridade.

  - Se todas as threads tiverem a mesma prioridade, a CPU é alocada para todos, um de cada vez, em modo round-Robin

    - getPriority(): obtém a prioridade corrente da thread;
    - setPriority(): define uma nova prioridade.

Programação Concorrente em Java

- Prioridade de Threads

- Cada thread possui uma prioridade de execução que vai de Thread. MIN_PRIORITY (igual a 1) a Thread.MAX_ PRIORITY (igual a 10).

  - Importante: uma thread herda a prioridade da thread que a criou.

- O algoritmo de escalonamento sempre deixa a thread (runnable) de maior prioridade executar.

  - A thread de maior prioridade preempta as outras threads de menor prioridade.

  - Se todas as threads tiverem a mesma prioridade, a CPU é alocada para todos, um de cada vez, em modo round-Robin

    - getPriority(): obtém a prioridade corrente da thread;
    - setPriority(): define uma nova prioridade.

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Um código é seguro quando chamado por múltiplas threads quando não possui condição de corrida.
  - A condição de corrida ocorre somente quando múltiplas threads tentam atualizar um recurso compartilhado
  - É importante conhecer quais os recursos as threads compartilham quando em execução

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Variáveis locais:
    - As variáveis locais são armazenadas na própria pilha de cada thread
    - Isso significa que essas variáveis nunca são compartilhadas entre as threads
    - Todas as variáveis locais primitivas são “thread safe”:. Exemplo:

```
public void algumMetodo() {

  long threadSafeInt = 0;

  threadSafeInt++;
}
```

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Referências a objetos locais:
    - As referências a objetos locais se comportam de maneira um pouco diferente de variáveis locais
    - A referência propriamente dita não é compartilhada, mas a referência ao objeto não é armazenada na pilha local de cada thread. Todos os objetos são armazenados no heap compartilhado
    - Se o objeto criado localmente nunca “foge” do método que o criou, ele é “thread safe”
    - É possível passar esse objeto para outros métodos e objetos, desde que nenhum deles torne esse objeto disponível para outras threads

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Exemplo:

```
public void algumMetodo(){

  ObjetoLocal objetoLocal = new objetoLocal();

objetoLocal.callMethod();
  metodo2(objetoLocal);
}

public void metodo2(ObjetoLocal objetoLocal){
  objetoLocal.setValue(“Alguma Coisa");
}
```

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - A instância de ObjetoLocal nesse exemplo não é retornada pelo método nem é passada para quaisquer outros objetos que são acessíveis de fora de algumMetodo()
  - Cada thread executando algumMetodo() irá criar sua própria instância de ObjetoLocal e atribuir a ele a referência a objetoLocal
  - Mesmo que a instância de ObjetoLocal seja passada como parâmetro para outros métodos da mesma classe ou de outras classes, o uso dessa forma é “thread safe”
  - A exceção é se um dos métodos chamados pelo ObjetoLocal como parâmetro armazena o ObjetoLocal de forma a permitir o acesso a outras threads

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Membros de Objetos:
    - Os membros de um objeto são armazenados no heap junto com esse objeto
    - Entretanto, se duas threads chamam um método dentro da mesma instância desse objeto e este método atualiza os membros desse objeto, o método não é seguro para threads. Exemplo:

```
public class ThreadNaoSegura{
   StringBuilder builder = new StringBuilder();

   public adicionar(String texto){
      this.builder.append(texto);
   }
}
```

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados

    - Se duas threads chamam o objeto adicionar() simultaneamente na mesma instância de ThreadNaoSegura, leva a uma condição de corrida

```
ThreadNaoSegura instanciaCompartilhada = new ThreadNaoSegura();

new Thread(new MinhaRunnable(instanciaCompartilhada)).start();
new Thread(new MinhaRunnable(instanciaCompartilhada)).start();

public class MinhaRunnable implements Runnable{
  ThreadNaoSegura instancia = null;

  public MinhaRunnable(ThreadNaoSegura instancia){
    this.instancia = instancia;
  }

  public void run(){
    this.instancia.adicionar(“algum texto");
  }   }
```

```
ThreadNaoSegura instanciaCompartilhada = new ThreadNaoSegura();

new Thread(new MinhaRunnable(instanciaCompartilhada)).start();
new Thread(new MinhaRunnable(instanciaCompartilhada)).start();

public class MinhaRunnable implements Runnable{
 ThreadNaoSegura instancia = null;

 public MinhaRunnable(ThreadNaoSegura instancia){
  this.instancia = instancia;
 }

 public void run(){
  this.instancia.adicionar(“algum texto");
 }  }
```

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - No exemplo anterior, as duas instâncias de MinhaRunnable compartilham a mesma instância de ThreadNaoSegura.
  - Mas quando elas chamam o método adicionar() simultaneamente leva a uma condição de corrida
  - Entrentanto, se duas threads chamam o método adicionar() simultaneamente em instâncias diferentes, não leva a uma condição de corrida

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Modificando levemente o exemplo ThreadNaoSegura:

```
new Thread(new MinhaRunnable(new ThreadNaoSegura())).start();
new Thread(new MinhaRunnable(new ThreadNaoSegura())).start();
```

  - Faz com que cada thread tenha sua própria instância de ThreadNaoSegura
  - Quando ela chama o método adicionar(), uma não interfere na outra e elimina a condição de corrida
    - Mesmo que um objeto não seja thread safe, ele pode ser utilizado de forma a não causar condição de corrida

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Recursos podem ser compartilhados: conexão ao banco de dados, objetos, arrays, arquivos
  - Mas em Java, esses recursos não são finalizados explicitamente (disposed). A referência a eles não é explicitamente anulada por default
  - Por este motivo, mesmo que um objeto seja “thread safe”, se esse objeto aponta para um recurso compatilhado, como um banco de dados, a aplicação como um todo pode não ser thread safe

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados

  - Se, por um exemplo, as threads 1 e 2 criam, cada uma, sua própria conexão à um banco de dados (conexao1 e conexao2) , cada conexão é segura

  - Mas se cada conexão aponta para um mesmo registro no BD, pode não ser thread safe. Algoritmo de Exemplo:

    Verifica se o registro X existe
    Se não existe, insere registro X

    Thread1: Verifica se registro X existe. Resultado: não
    Thread2: Verifica se registro X existe. Resultado: não
    Thread1: Insere registro X
    Thread2: Insere registro X

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Condições de corrida ocorrem somente se múltiplas threads estão acessando o mesmo recurso e uma ou mais threads tentam escrever no recurso
  - Se múltiplas threads executam somente leitura do mesmo recurso, a condição de corrida não ocorre
  - Para ter certeza que objetos compartilhados entre threads nunca serão atualizados por nenhuma thread, é necessário fazer com que esse recurso seja imutável (e thread safe, por consequência)

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Exemplo:

```
public class ValorImutavel{

  private int valor = 0;

  public ValorImutavel(int valor){
    this.valor = valor;
  }

  public int pegaValor(){
    return this.valor;
  }
}
```

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Veja que o valor da instância de ValorImutavel é passada no construtor e não existe método set, somente get
  - Uma vez que a instância de ValorImutavel é criada, você não pode alterá-la. É imutável
  - Mas é possível lê-la, usando o método pegaValor

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Se for necessário efetuar operações na instância de ValorImutavel, você deve criar uma nova instância com o valor resultante da operação:

```
public ValorImutavel adicionar ( int valorAdicionar) {
return  new ValorImutavel(this.valor + valorAdicionar);
}
```

  - O método adicionar() retorna uma nova instância de ValorImutavel com o resultado da operação adicionar, mas não o valor original de ValorImutavel propriamente dito

Programação Concorrente em Java

```
public class ValorImutavel{

 private int valor = 0;

 public ValorImutavel(int valor){
  this.valor = valor;
 }

 public int pegaValor(){
  return this.valor;
 }
      public ValorImutavel adicionar ( int valorAdicionar) {
      return  new ValorImutavel(this.valor + valorAdicionar);
      }
}
```

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - Vale lembrar que mesmo que um objeto seja imutável, a referência a ele pode não ser. Exemplo:

```
public class Calculadora{
  private ValorImutavel valorAtual= null;

  public ValorImutavel pegaValor (){
    return valorAtual;
  }

  public void gravaValor(ValorImutavel novoValor){
    this.valorAtual = novoValor;
  }

  public void adicionard(int novoValor){
    this.valorAtual = this.valorAtual.adicionar(novoValor);
  }
}
```

Programação Concorrente em Java

- Segurança em Threads Java e Recursos Compartilhados
  - A classe Calculadora mantém uma referência para uma instância de ValorImutavel
  - Mas é possível alterar a referência através dos métodos gravaValor() e de adicionar()
  - Entretanto, mesmo se a classe Calculadora usa um objeto imutável internamente, a própria classe não é imutável e, por consequência, não é segura para threads
  - Em outras palavras: a classe ValorImutavel é thread safe, mas seu uso não é...
  - Para fazer a classe Calculadora ok para threads, é necessário fazer os métodos gravaValor, pegaValor e adicionar SINCRONIZADOS

Programação Concorrente em Java

- Java Synchronized
  - Os métodos sincronizados são utilizados para permitir que somente uma thread seja executada dentro de um bloco sincronizado
  - Um bloco sincronizado em Java é marcado pela palavra chave synchronized e um bloco é sincronizado dentro de algum objeto
  - Toda e qualquer tentativa de entrar no bloco sincronizado é bloqueada até que a thread dentro desse bloco sincronizado saia dele

Programação Concorrente em Java

- Java Synchronized
  - Podem ser usadas em quatro diferentes tipos de blocos:
    - Instâncias de métodos
    - Métodos estáticos
    - Blocos de código dentro de instâncias de métodos
    - Blocos de código dentro de métodos estáticos

Programação Concorrente em Java

- Java Synchronized
  - Instâncias de métodos
    - A instância do método sincronizado em Java é sincronizada na instância que é proprietária do método, sendo executada apenas uma thread por instância
    - Se mais de uma instância existe, então cada thread é executada em um momento, por instância. Uma thread por instância somente.

```
public synchronized void adicionar(int valor){
    this.somar += valor;
 }
```

Programação Concorrente em Java

- Java Synchronized
  - Métodos Estáticos Sincronizados
    - Os métodos estáticos sincronizados são sincronizados na classe do objeto da classe ao qual o método sincronizado estático pertence. Uma vez que somente uma classe de objeto existe na JVM por classe, somente uma thread pode executar dentro desse método estático na mesma classe
    - Se o método estático sincronizado está localizado em uma classe diferente, então apenas uma thread pode ser executada dentro de cada método estático sincronizado dentro de cada classe.
    - Usa uma thread por classe independentemente de qual método estático sincronizado ela chama

```
public  static synchronized void adicionar(int valor){
somar += valor;
 }
```

Programação Concorrente em Java

- Java Synchronized
  - Blocos Sincronizados em Instâncias de Métodos
    - Não é necessário sincronizar todo um método, muitas vezes é preferível sincronizar somente parte de um método. O código abaixo será executado como se estivesse em um método sincronizado
    - O (this) é chamado de objeto monitor, que é a instância do método adicionar, e o código entre chaves é o código a ser sincronizado no objeto monitor

```
public  void adicionar(int valor){
            synchronized(this)  {
                    somar += valor;
             }
}
```

Programação Concorrente em Java

- Java Synchronized
  - Blocos Sincronizados em Instâncias de Métodos
    - Exemplo:

```
public class MinhaClasse {

   public synchronized void log1(String msg1, String msg2){
     log.writeln(msg1);
     log.writeln(msg2);
   }


   public void log2(String msg1, String msg2){
     synchronized(this){
       log.writeln(msg1);
       log.writeln(msg2);
     }
   }
 }
```

Programação Concorrente em Java

- Java Synchronized
  - Blocos Sincronizados em Instâncias de Métodos
    - Somente uma thread pode ser executada dentro do bloco sincronizado no mesmo objeto monitor.
    - No exemplo, temos dois blocos sincronizados, mas somente uma thread é executada dentro de ambos os blocos
    - Caso o segundo bloco sincronizado seja sincronizado em um objeto diferente do (this), então somente uma thread será executada por vez dentro de cada método

Programação Concorrente em Java

- Java Synchronized
  - Blocos Sincronizados em Métodos Estáticos
    - Somente uma thread pode ser executada dentro de cada método por vez
    - Se um segundo bloco sincronizado for sincronizado em um objeto diferente, então somente uma thread poderá executar dentro de cada método por vez

```
public class MinhaClasse {

  public static synchronized void log1(String msg1, String msg2){
    log.writeln(msg1);
    log.writeln(msg2);
  }
  public static void log2(String msg1, String msg2){
    synchronized(MinhaClasse.class){
      log.writeln(msg1);
      log.writeln(msg2);
    }    }   }
```

Programação Concorrente em Java

- Exemplo
  - Contador Sincronizado
    - Duas threads são criadas, utilizando o método adicionar de uma mesma instância da classe Contador

```
public class Contador {

  long contar = 0;

  public synchronized void adicionar(long valor){
    this.contar += valor;
    System.out.println(Thread.currentThread().getName() + " : "+ this.contar);
  }
}
```

Programação Concorrente em Java

- Contador Sincronizado

  - Gera os valores para o contador

```
public class ContaThread extends Thread{

  protected Contador contador = null;

  public ContaThread(Contador contador){
    this.contador = contador;
  }

  public void run() {
          for(int i=0; i<10; i++){
      contador.adicionar(i);
    } } }
```

Programação Concorrente em Java

- Exemplo
  - Contador Sincronizado
    - Nesse exemplo, são criadas duas threads e a mesma instância de contador é passada para ambas as threads
    - Somente uma das threads pode chamar o método adicionar a cada momento
    - A outra thread deverá aguardar a finalização da primeira para poder utilizar o método adicionar

Programação Concorrente em Java

- Exemplo
  - Contador Sincronizado

```
public class Exemplo {

        public static void main(String[] args){
            Contador contador = new Contador();
            Thread  threadA = new ContaThread(contador);
            Thread  threadB = new ContaThread(contador);

            threadA.start();
            threadB.start();
          }     }
```

Programação Concorrente em Java

- Exemplo
  - Contador Sincronizado
    - Se as duas threads referenciam instâncias separadas do contador, não há problemas em chamar contador simultaneamente
    - As chamadas são realizadas para objetos diferentes e as chamadas de uma thread não bloqueiam as de outra thread

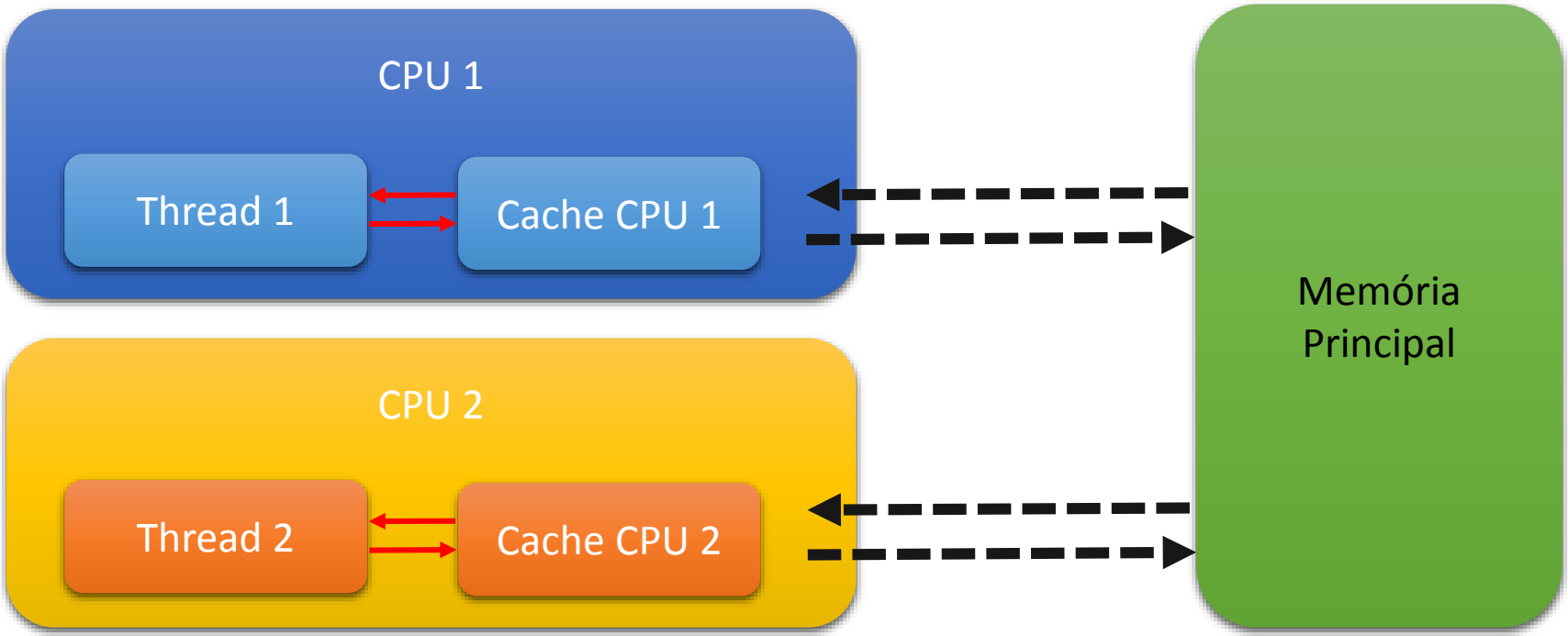
```
public class Exemplo {

        public static void main(String[] args){

            Contador contadorA = new Contador();
            Contador contadorB = new Contador();
            Thread  threadA = new ContaThread(contadorA);
            Thread  threadB = new ContaThread(contadorB);

            threadA.start();
            threadB.start();
          }
        }
```

Programação Concorrente em Java

- Java Volatile
  - Synchronized é um dos mecanismos para permitir o acesso sincronizado de um objeto por várias threads, mas não é o único
  - O tipo volatile do Java é utilizado para determinar variáveis que ficam armazenadas na memória principal e não no cache da CPU
  - Ela garante a **visibilidade de mudanças nas variáveis** através de várias threads
  - Em programas que não utilizam variáveis volatile, cada thread pode copiar essas variáveis da memória principal para o cache da CPU, para melhorar o desempenho

Programação Concorrente em Java

- Java Volatile
  - Se o computador possui várias CPUs, cada thread pode utilizar uma CPU diferente, ou seja, cada variável pode ser copiada pelo cache de CPUs diferentes



Programação Concorrente em Java

- Java Volatile
  - Não existe garantia de que a JVM irá escrever os dados diretamente da memória principal no cache da CPU e vice-versa
  - Imagine a situação em que duas ou mais threads precisam acessar um objeto compartilhado (contador, por exemplo)

```
public class ObjetoCompartilhado {

    public int contador = 0;

}
```

Programação Concorrente em Java

- Java Volatile
  - Nesse caso, a thread 1 pode ler uma variável compartilhada com o valor 0 no cache da CPU, incrementar para 1 e não gravar na memória principal
  - Enquanto isso, uma thread 2 pode ler a mesma variável diretamente da memória principal e também incrementá-la, sem escrever de volta na memória principal
  - As duas threads estariam fora de sincronia. O valor real da variável deveria ser 2, no cache de cada CPU estaria em 1 e em 0 na memória principal...

Programação Concorrente em Java

- Java Volatile

  - Ao declarar a variável como Volatile, a JVM garante que todas leituras das variáveis serão feitas diretamente da memória principal e que todas as escritas também irão diretamente para a memória principal:

```
public class ObjetoCompartilhado {

    public volatile int counter = 0;

}
```

Programação Concorrente em Java

- Java Volatile
  - Todas as variáveis volatile são visíveis por todas as threads. Isso é necessário para manter a consistência dos dados na memória
  - Por questões de desempenho, a JVM pode reordenar a execução das instruções que utilizam variáveis non-volatile, mas o mesmo não acontece com variáveis declaradas volatile
  - As instruções antes e depois das variáveis volatile podem ser reordenadas, mas somente após uma operação de leitura ou escrita dessa variável
  - É necessário, porém, a sincronização para garantir a atomicidade das operações de leitura e escrita

Programação Concorrente em Java

- Java Volatile
  - O problema de utilizar volatile é o desempenho. Ler e gravar diretamente na memória principal demora mais tempo do que utilizar o cache da CPU
  - O problema de não ser possível reordenar as instruções pode causar stall no pipeline com maior frequência
  - Volatile só deverá ser utilizado quando for necessário deixar a variável visível para outras threads

Programação Concorrente em Java

- ThreadLocal
  - Permite que sejam criadas variáveis que são somente lidas e escritas pela mesma thread
  - Se duas threads executam o mesmo código, e ambas são threadlocal, uma não pode enxergar as variáveis da outra

```
private ThreadLocal minhaThreadLocal = new ThreadLocal();
```

Programação Concorrente em Java

- ThreadLocal
  - Permite que sejam criadas variáveis que são somente lidas e escritas pela mesma thread
  - Se duas threads executam o mesmo código, e ambas são threadlocal, uma não pode enxergar as variáveis da outra

```
private ThreadLocal minhaThreadLocal = new ThreadLocal();
```

  - Uma vez que threadlocal foi utilizada, todas as modificações feitas por um método set() serão visíveis somente para essa thread

Programação Concorrente em Java

- ThreadLocal
  - Uma vez que threadlocal foi criada, você pode ler o valor da variável usando

```
String valorThreadLocal = (String) minhaThreadLocal.get();
```

  - E pode gravar nessa variável usando

```
minhaThreadLocal.set(“Algum valor");
```

Programação Concorrente em Java

- ThreadLocal
  - Uma vez que os valores de threadlocal são visíveis somente pela thread que as utiliza, sem visibilidade para outras threads, nenhuma thread pode inicializar valores nela utilizando set(), que a tornaria visível para todas as outras threads
  - Para fazer isso, é necessário chamar o método initialValue():

```
private ThreadLocal minhaThreadLocal = new ThreadLocal<String>() {
    @Override protected String initialValue() {
        return “Este é o valor inicial!";
    }
};
```

Programação Concorrente em Java

- ThreadLocal
  - O exemplo a seguir cria uma instância simples de MeuRunnable, que é passada para duas threads diferentes.
  - Ambas executam o método run() e setam valores diferentes da instância threadlocal.
  - Se o set() fosse sincronizado e não fosse um objeto threadlocal, a segunda thread sobrescreveria o valor da variável

Programação Concorrente em Java

- Sinalização através de objetos compartilhados
  - O objetivo geral da sinalização é..... Sinalizar!
  - Permite que uma thread envie sinais de umas para as outras, bem como receber sinais de outras threads
  - Uma thread B pode aguardar uma thread A terminar um processamento e preparar dados para que a thread B possa continuar sua execução

Programação Concorrente em Java

- Sinalização através de objetos compartilhados
  - Uma forma de sinalizar (ou enviar sinais) é atribuir algum valor para uma variável compartilhada
  - Uma thread A pode setar uma variável booleana dentro de um bloco sincronizado para verdadeiro, permitindo que uma outra thread B leia essa variável, também dentro do bloco sincronizado
  - Para tanto, ambas deve referenciar uma variável compartilhada para essa sinalização funcionar

Programação Concorrente em Java

- Sinalização através de objetos compartilhados
  - Um objeto que pode manter uma variável compartilhada é apresentado abaixo:

```
public class MeuSinal  {

 protected boolean temDadoParaProcessar = false;

 public synchronized boolean temDadoParaProcessar(){
  return this. temDadoParaProcessar;
 }

 public synchronized void setaTemDadoParaProcessar(boolean temDado){
  this. temDadoParaProcessar = temDado;
 }
}
```

Programação Concorrente em Java

- Sinalização através de objetos compartilhados
  - Espera Ocupada:
    - Uma thread B pode ficar aguardando uma thread A terminar seu processamento para continuar. Nessa atividade, ela pode entrar em loop esperando a finalização de A

```
while(!sinalCompartilhado.temDadoParaProcessar()){
  //não faz nada.... Espera ocupada……
}
```

    - O loop fica em true enquanto aguarda o término de A. É a espera ocupada.

Programação Concorrente em Java

- Sinalização: usando wait, notify e notifyAll
  - A espera ocupada não é eficiente do ponto de vista do processamento da CPU
  - Java possui um mecanismo para permitir que threads possam se tornar inativas ou “dormir”, enquanto esperam uma sinalização
  - Uma thread que chama wait() em qualquer objeto se torna inativa até que alguma outra thread chame um notify() naquele objeto.
  - Tanto wait() quanto notify() necessitam estar dentro de um bloco sincronizado

Programação Concorrente em Java

- Sinalização: usando wait, notify e notifyAll

```
public class MonitoraObjeto{ }

public class EsperaNotificacao{
 MonitoraObjeto meuMonitoraObjeto = new MonitoraObjeto();

 public void executaEspera(){
  synchronized(meuMonitoraObjeto){
   try{
    meuMonitoraObjeto.wait();
   } catch(InterruptedException e){...}
  } }

 public void executaNotificacao(){
  synchronized(meuMonitoraObjeto){
   meuMonitoraObjeto.notify();
  } }  }
```

Programação Concorrente em Java

- Sinalização: usando wait, notify e notifyAll
  - A thread que deve esperar executa executaEspera() e a que deve notificar deve chamar o executaNotificacao()
  - Quando o notify() é executado para um objeto, uma das threads esperando aquele objeto é acordada e é permitida sua execução.
  - O método notifyAll() é utilizado para acordar todas as threads que esperam um determinado objeto
  - É obrigatória a utilização do wait() e notify() dentro de um método ou bloco sincronizado. Se isso não acontecer, é gerada a exceção illegalMonitorStateException

Programação Concorrente em Java

- Sinalização: usando wait, notify e notifyAll
  - A thread que deve esperar executa executaEspera() e a que deve notificar deve chamar o executaNotificacao()
  - Quando o notify() é executado para um objeto, uma das threads esperando aquele objeto é acordada e é permitida sua execução.
  - O método notifyAll() é utilizado para acordar todas as threads que esperam um determinado objeto
  - É obrigatória a utilização do wait() e notify() dentro de um método ou bloco sincronizado. Se isso não acontecer, é gerada a exceção illegalMonitorStateException