

Unidades concorrentes em Java

Paradigmas de Linguagens de Programação

Heloisa de Arruda Camargo

Unidades concorrentes em JAVA

- Em programação concorrente existem duas unidades básicas de execução: processo e threads.
- A programação concorrente em Java está relacionada principalmente com threads
- Um processo tem um ambiente de execução auto contido e geralmente é visto como sinônimo de programa ou aplicação
- Threads existem dentro de processos – todo processo tem pelo menos uma thread
- Threads compartilham os recursos de um processo, como memória e arquivos.
- Do ponto de vista do programador, começa-se com apenas uma thread, que é a thread main. Essa thread pode criar threads adicionais.

Unidades concorrentes em JAVA

- JAVA disponibiliza classes especiais para criação e manipulação de threads (linhas de execução)
- Podemos criar múltiplas linhas de execução em JAVA usando a classe *Thread* ou a interface *Runnable*.
- Criando um objeto de uma classe que é subclasse de Thread, inicia-se uma nova thread.
- O mesmo acontece criando um objeto de uma classe que implementa Runnable.
- Os métodos dessas classes podem ser sincronizados.

Exemplo

class Bola extends Thread

```
{....  
...}
```

class Bola implements Runnable

```
{....  
...}
```

Métodos da classe Thread (java.lang.Thread)

`void run ()`

- sempre sobreposto pelas subclasses de Thread. Contém o código que define o que a linha de execução faz.

`void start ()`

- inicia seu objeto como uma unidade concorrente chamando seu método run.
- Após a chamada para start o controle retorna imediatamente para o chamador que continua sua execução em paralelo com o método run que foi iniciado

Definindo uma Thread como subclasse de Thread:

```
class Bola extends Thread {  
    long campo;  
    Bola(long campo) {  
        this.campo = campo;  
    }  
    // sobrepõe o método run da classe Thread  
    public void run ( ) {  
        // aqui vai o que a thread vai fazer...  
        ...  
    }  
}
```

Criando e iniciando uma thread:

```
Bola b = new Bola (321);  
b.start ( );
```

Definindo uma Thread que implementa a interface Runnable:

```
class BolaRun implements Runnable {  
    long campo;  
    BolaRun(long campo) {  
        this.campo = campo;  
    }  
    // implementa o método run da classe Thread  
    public void run ( ) {  
        // aqui vai o que a thread vai fazer...  
        ...  
    }  
}
```

Criando e iniciando uma thread:

```
BolaRun b = new BolaRun (321);  
new Thread(b).start ( );
```

- Considerando o conceito de herança de classes, a classe Thread não é o “pai” natural de outras classes, no sentido compreendido pela POO. Essa classe apenas oferece serviços para as subclasses implementarem a concorrência.
- Em geral o escalonador é implementado atribuindo fatias de tempo de igual duração às diferentes linhas de execução, de forma circular, desde que elas tenham igual prioridade.

Construtores da classe Thread

Thread ()

- instancia um objeto da classe Thread

Thread (String nome)

- instancia um objeto da classe Thread, passando um nome

Thread (Runnable target)

- instancia um objeto da classe Thread passando o parâmetro target que é o objeto cujo método run será invocado quando essa thread for iniciada. Se for null, o método run dessa classe não faz nada.

Thread (Runnable target, String name)

- instancia um objeto da classe Thread, passando o parâmetro target e o nome da thread.

Exemplo

Classe Imprime implementa Runnable e imprime 1000 números inteiros

```
class Imprime implements Runnable {  
    private int id;  
    // colocar getter e setter para o atributo id  
    public void run () {  
        for (int i = 0; i < 10000; i++) {  
            System.out.println("Linha " + id + " i: " + i);  
        }  
    }  
}
```

Exemplo

Classe Imprime implementa Runnable e imprime 1000 números inteiros

```
public class Teste {  
    public static void main(String[] args) {  
        Imprime linha1 = new Imprime();  
        linha1.setId(1);  
        Thread t1 = new Thread(linha1);  
        t1.start();  
        Imprime linha2 = new Imprime();  
        linha2.setId(2);  
        Thread t2 = new Thread(linha2);  
        t2.start();  
    }  
}
```

Qual é a sequência de saída?

- Não sabemos exatamente qual é a sequência de saída. Rodando o programa várias vezes, em cada execução a saída pode ser diferente.
- No computador, em geral existem menos processadores que threads.
- O tempo dos processadores deve ser dividido entre as threads que são executadas de forma alternada, controlada pelo **escalonador de tarefas**
- A ideia é executar um pouco de cada thread e fazer essa troca tão rapidamente que a impressão que fica é que as coisas estão sendo feitas ao mesmo tempo

Ações do escalonador

- O escalonador decide quando fazer a **troca de contexto**, por quanto tempo a thread vai rodar e qual vai ser a próxima thread a ser executada.
 - *Troca de contexto*: operações de salvar o contexto da thread atual e restaurar o da thread que vai ser executada em seguida.
- Salva o estado da execução da thread atual para depois poder retomar a execução da mesma.
- Restaura o estado da thread que vai ser executada e faz o processador continuar a execução desta.
- Depois de um certo tempo, esta thread é tirada do processador, seu estado (o contexto) é salvo e outra thread é colocada em execução.

Definindo nomes para Threads

- Toda thread tem um nome para identificação.
- O nome da thread pode ser definido pelo construtor ou pelo método `setName()`.
- Se não for definido um nome na criação da thread, um nome é gerado automaticamente.

`void setName (String name)`

- Muda o nome da thread para name

`String getName ()`

- Retorna o nome da Thread que pode ser definido por `setName()` ou no construtor.

Definir e recuperar nome de uma thread

```
public class ThreadNameDemo extends Thread {  
    public ThreadNameDemo() {  
    }
```

```
    public ThreadNameDemo(String name) {  
        super(name);  
    }
```

```
@Override
```

```
    public void run() {  
        //  
        // Call getName() method to get the thread name of this  
        // thread object.  
        //  
        System.out.println("Running [" + this.getName() + "]);  
    }
```

```
public static void main(String[] args) {  
    Thread thread1 = new ThreadNameDemo();  
    thread1.setName("FOX");  
    thread1.start();  
  
    Thread thread2 = new ThreadNameDemo("DOG");  
    thread2.start();  
}  
}
```


Métodos da classe Thread

`static void sleep (long ms)`

- requisição para bloquear uma thread por um número de milisegundos especificados pelo parâmetro de sleep.
- depois que o tempo especificado passou, a thread é colocada na fila de tarefas prontas.

Métodos da classe Thread

`void join () throws InterruptedException`

- A linha que chamou `join` espera que a linha chamada termine sua execução antes de continuar.
- Lança `InterruptedException` se alguma thread interromper a thread corrente. O status “interrompida” é eliminado quando a exceção é lançada

`void join (long ms)`

- A linha que chamou `join` espera que a linha chamada termine sua execução até `ms` milisegundos antes de continuar.

Exemplo – concorrência com a classe Thread

```
class Carro extends Thread {  
    public Carro (String nome) {  
        super (nome);  
    }  
    public void run ( ) {  
        for (int i=0; i<10; i++) {  
            try { sleep((int) (Math.random( ) * 1000));  
            }  
            catch (Exception e) {  
            };  
            System.out.print(getName( ));  
            for (int j=0; j<i; j++)  
                System.out.print("- ");  
            System.out.println(">");  
        }  
        System.out.println(getName( ) + " completou a prova.");  
    }  
}
```

```
publica class Corrida {  
    public static void main(String args [ ])  
        Carro carroA = new Carro("Barrichelo");  
        Carro carroB = new Carro("Schumacher");  
        carroA.start( );  
        carroB.start( );  
        try {  
            carroA.join( );  
        } catch (Exception e) { }  
        try {  
            carroB.join( );  
        } catch (Exception e) { }  
    }  
}
```

Possível Resultado

```
Barrichelo>
Schumacher>
Schumacher-->
Barrichelo-->
Schumacher---->
Barrichelo---->
Schumacher----->
Barrichelo----->
Barrichelo----->
Schumacher----->
Barrichelo----->
Schumacher----->
Barrichelo----->
Schumacher----->
Barrichelo----->
Schumacher----->
Barrichelo----->
Barrichelo----->
Barrichelo completou a prova.
Schumacher----->
Schumacher----->
Schumacher completou a prova.
```

Exemplo – concorrência com a interface Runnable

```
class Carro2 implements Runnable {  
    private String nome;  
    public Carro2 (String nome) {  
        this.nome = nome;  
    }  
    public void run ( ) {  
        for (int i=0; i<10; i++) {  
            try { Thread.sleep((int) (Math.random( ) * 1000));  
            }  
            catch (Exception e) { };  
            System.out.print(nome);  
            for (int j=0; j<i; j++)  
                System.out.print("- ");  
            System.out.println(">");  
        }  
        System.out.println(nome( ) + " completou a prova.");  
    }  
}
```

```

public class Corrida2 {
    public static void main(String args [ ])
        Carro2 carroA = new Carro2("Barrichelo");
        Carro2 carroB = new Carro2("Schumacher");
        Thread threadA = new Thread(carroA);
        Thread threadB = new Thread(carroB);
        threadA.start( );
        threadB.start( );
        try {
            threadA.join( );
        } catch (Exception e) {
        }
        try {
            threadB.join( );
        } catch (Exception e) {
        }
    }
}

```

- A classe Carro2 não é herdeira de Thread
- A criação de um objeto dessa classe não cria uma nova linha de execução
- JAVA permite a criação de um objeto da classe Thread a partir de um objeto que implementa a interface Runnable
- Para chamar sleep é preciso explicitar a classe Thread

Estados de Linha de Execução

- Novas – a linha foi criada com o operador new mas não foi iniciada ainda.
- Passíveis de execução – quando é chamado o método start. Uma linha passível de execução pode não estar sendo executada. Isso é gerenciado pelo sistema operacional.
- Em execução – quando o código dentro da linha de execução passa a ser executado.
 - A documentação da plataforma Java não distingue entre esses dois estados.

- Bloqueadas – a execução foi interrompida e a linha de execução está aguardando a liberação de um recurso ou a passagem de um intervalo de tempo
- Mortas – o método run () foi concluído porque acabou sua execução ou por uma exceção não capturada (que encerra o método run)

Condições para bloquear uma linha de execução

- Alguém chama o método sleep() da linha de execução
- A linha de execução chama o método wait()
- Alguém chama o método suspend () da linha de execução
- A linha de execução chama uma operação que está bloqueando entrada/saída (operação que não retorna ao chamador até que esteja concluída)
- A linha de execução tenta bloquear um objeto que está bloqueado por outra execução.

Condições para sair do bloqueio

- Terminou o tempo especificado pelo método `sleep()`.
- Terminou a execução de uma operação de entrada e saída que a linha de execução estava esperando.
- A linha de execução chamou `wait ()` e outra linha de execução chamou `notify` ou `notifyAll`.
- A linha de execução pediu o bloqueio de um objeto que estava bloqueado por outra linha e essa outra liberou o objeto.
- A linha de execução foi suspensa e alguém chamou o método `resume` (não recomendado).

Métodos da classe Thread

static void yield ()

- requisição de uma thread em execução para abrir mão do tempo de execução restante. É colocada na pilha de tarefas prontas no estado executável e o escalonador escolhe outra thread para iniciar o processamento
 - é um método estático

public final boolean isAlive ()

- retorna true se a linha de execução for passível de execução ou bloqueada e false se for nova ou encerrada.

static Thread currentThread()

- método estático da classe Thread que volta uma referência para a thread que está sendo executada.

Métodos da classe Thread (**depreciados**)

`void stop ()`

- termina a execução de uma thread, alterando seu estado para “morta” ou “encerrada”.

`void suspend ()`

- suspende a execução de uma thread temporariamente.

`void resume ()`

- move uma thread do estado bloqueado para a fila de tarefas prontas, válido depois do uso de `suspend ()`.

Uso de join - Exemplo

O método join é usado quando uma thread precisa esperar que outra termine:

```
final void join( ) throws InterruptedException
```

É usado para garantir que o método main seja o último a parar.

**Exemplo
sem join**



```
public class ExemploSemJoin {  
    public static void main(String args[]) {  
        Contador c1 = new Contador();  
        c1.setQtde(10);  
        c1.setName("t001");  
        c1.start();  
        Contador c2 = new Contador();  
        c2.setQtde(15);  
        c2.setName("t002");  
        c2.start();  
    }  
}
```

```

class Contador extends Thread {
    private int qtde = 0;
    public void run() {
        for (int i=0; i <= 100; i++) {
            if ((i%qtde) == 0) {
                System.out.println(Thread.currentThread().getName()+"> "+i);
            }
            try {
                sleep(500);
            }
            catch (InterruptedException ex) {
            }
        }
    }

    public void setQtde(int value) {
        this.qtde = value;
        if (this.qtde == 0) this.qtde = 10;
    }
}

```



```
D:\Projetos JDK>javac ExemploSemJoin.java
```

```
D:\Projetos JDK>java ExemploSemJoin
```

```
t001> 0
```

```
t002> 0
```

```
t001> 10
```

```
t002> 15
```

```
t001> 20
```

```
t001> 30
```

```
t002> 30
```

```
t001> 40
```

```
t002> 45
```

```
t001> 50
```

```
t002> 60
```

```
t001> 60
```

```
t001> 70
```

```
t002> 75
```

```
t001> 80
```


```
t002> 90
```

```
t001> 90
```

```
t001> 100
```

```
D:\Projetos JDK>
```

**Exemplo
com join**



```
public class ExemploComJoin {
    public static void main(String args[]) {
        try {
            Contador c1 = new Contador();
            c1.setQtde(10);
            c1.setName("t001");
            c1.start();
            c1.join();
            for (int i=0; i <= 100; i++) {
                if ((i%5) == 0) {
                    System.out.println(Thread.currentThread().getName()+"> "+i);
                }
            }
        }
        catch (InterruptedException e) {
        }
    }
}
```

```
D:\Projetos JDK>javac ExemploComJoin.java
```

```
D:\Projetos JDK>java ExemploComJoin
```

```
t001> 0  
t001> 10  
t001> 20  
t001> 30  
t001> 40  
t001> 50  
t001> 60  
t001> 70  
t001> 80  
t001> 90  
t001> 100  
main> 0  
main> 5  
main> 10  
main> 15  
main> 20  
main> 25  
main> 30  
main> 35  
main> 40  
main> 45  
main> 50  
main> 55  
main> 60  
main> 65  
main> 70  
main> 75  
main> 80  
main> 85  
main> 90  
main> 95  
main> 100
```

```
D:\Projetos JDK>
```

Exemplo – join , isAlive

```
import java.lang.*;

public class ThreadDemo implements Runnable {
    public void run() {

        Thread t = Thread.currentThread();
        System.out.print(t.getName());
        //checks if this thread is alive
        System.out.println(", status = " + t.isAlive());
    }

    public static void main(String args[]) throws
Exception {

        Thread t = new Thread(new ThreadDemo());
        // this will call run() function
        t.start();
        // waits for this thread to die
        t.join();
        System.out.print(t.getName());
        //checks if this thread is alive
        System.out.println(", status = " + t.isAlive());
    }
}
```

```
D:\Projetos JDK>javac TestIsAlive.java
```

```
D:\Projetos JDK>java TestIsAlive
```

```
Thread-0, status = true
```

```
Thread-0, status = false
```

```
D:\Projetos JDK>
```

Métodos sincronizados

- São declarados com `synchronized`
- Threads que chamem esses métodos são executados com exclusão mútua
- Quando uma thread chama um método declarado como **synchronized** a partir de uma instância de objeto, é verificado se alguma outra thread está de posse do bloco de operações daquele objeto.
- Se sim, o requisitante é bloqueado e colocado no conjunto de threads de entrada para aquele objeto.
- Se não, ele toma posse do bloco de operações e inicia a execução do método.
- quando o bloco de operações é liberado, e há threads no conjunto de entrada um thread é selecionada arbitrariamente

```
public class Counter{  
    long count = 0;  
    // o método add é sincronizado nas instâncias de Counter  
    public synchronized void add(long value){  
        this.count += value;  
    }  
}
```

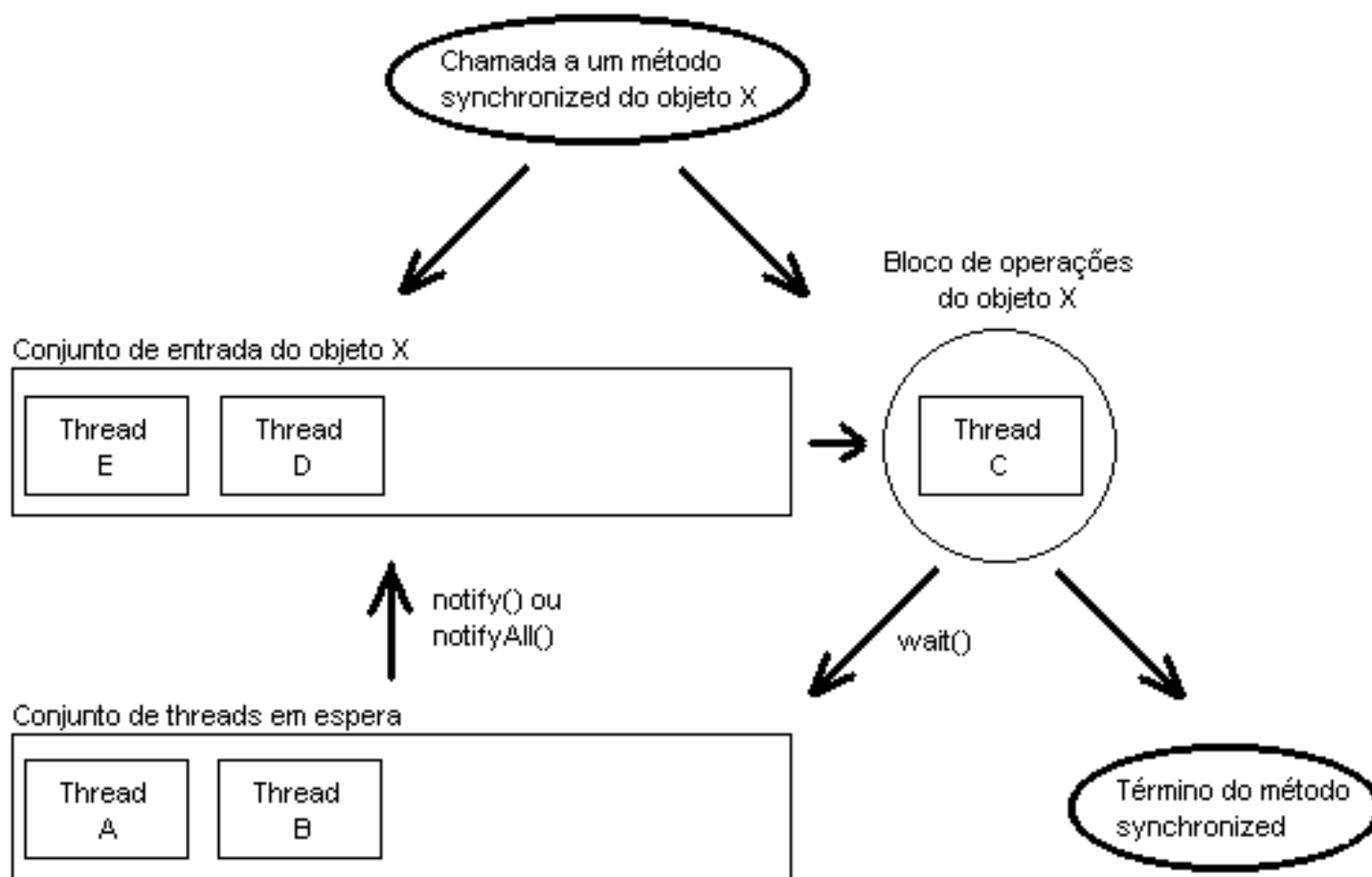
```
public class CounterThread extends Thread{  
    protected Counter counter = null;  
    public CounterThread(Counter counter){  
        this.counter = counter;  
    }  
    public void run() {  
        for(int i=0; i<10; i++){  
            counter.add(i);  
        }  
    }  
}
```



```
public class Example {  
    public static void main(String[] args){  
        Counter counter = new Counter();  
        // Duas threads são criadas. A mesma instância counter  
        // é passada para os construtores.  
        Thread threadA = new CounterThread(counter);  
        Thread threadB = new CounterThread(counter);  
        threadA.start();  
        threadB.start();  
    }  
}
```

- O método counter.add é sincronizado na instância
- Somente uma thread pode chamar o método add de cada vez
- A outra thread vai esperar até que a primeira deixe o método add para poder executar

Métodos Sincronizados



Liberação do bloco de operações

- Pelo término do método `synchronized`
- Pela chamada ao método `wait ()`.

(`wait ()` – método de `Object`)

- Quando uma thread chama o método `wait()`:
 - libera o bloco de operações
 - a thread é bloqueada e colocada no conjunto em espera;
 - outra thread do conjunto de entrada é selecionada.

Método notify ()

- a JVM seleciona uma thread do conjunto em espera e transfere para o conjunto de entrada para competir pelo bloco de operações;

Método notifyAll()

- todas as threads do conjunto em espera são transferidas para o conjunto de entrada

Produtor e Consumidor com Métodos Sincronizados

```
class BufferLimitado {  
    private int capacidade;  
    private int n;  
    private int buffer[];  
    private int fim;  
    private int ini;  
    public BufferLimitado(int capacidade) {  
        this.capacidade = capacidade;  
        n = 0;  
        fim = 0;  
        ini = 0;  
        buffer = new int[capacidade];  
    }
```

Produtor e Consumidor com Métodos Sincronizados

```
public synchronized void inserir(int elemento) throws  
    InterruptedException  
{  
    while ( n == capacidade) {  
        wait();  
    }  
    buffer[fim] = elemento;  
    fim = (fim + 1) % capacidade;  
    n = n + 1;  
    notify();  
}
```

Produtor e Consumidor com Métodos Sincronizados

```
public synchronized int retirar() throws  
InterruptedException {  
    while ( n == 0) {  
        wait();  
    }  
    int elem = buffer[ini];  
    ini = (ini + 1) % capacidade;  
    n = n - 1;  
    notify();  
    return elem;  
}  
}
```

Produtor e Consumidor com Métodos Sincronizados

```
class Produtor extends Thread {  
    private BufferLimitado buffer;  
    public Produtor(BufferLimitado buffer) {  
        this.buffer = buffer;  
    }  
}
```


Produtor e Consumidor com Métodos Sincronizados

```
public void run() {  
    int elem;  
    while (true) {  
        elem = (int)(Math.random() * 10000);  
        try {  
            buffer.inserir(elem);  
            System.out.println("produzido: " + elem);  
            Thread.sleep((int)(Math.random() * 1000));  
        } catch (InterruptedException e) {  
            System.out.println("Erro Produtor: " + e.getMessage());  
        }  
    }  
}
```

Produtor e Consumidor com Métodos Sincronizados

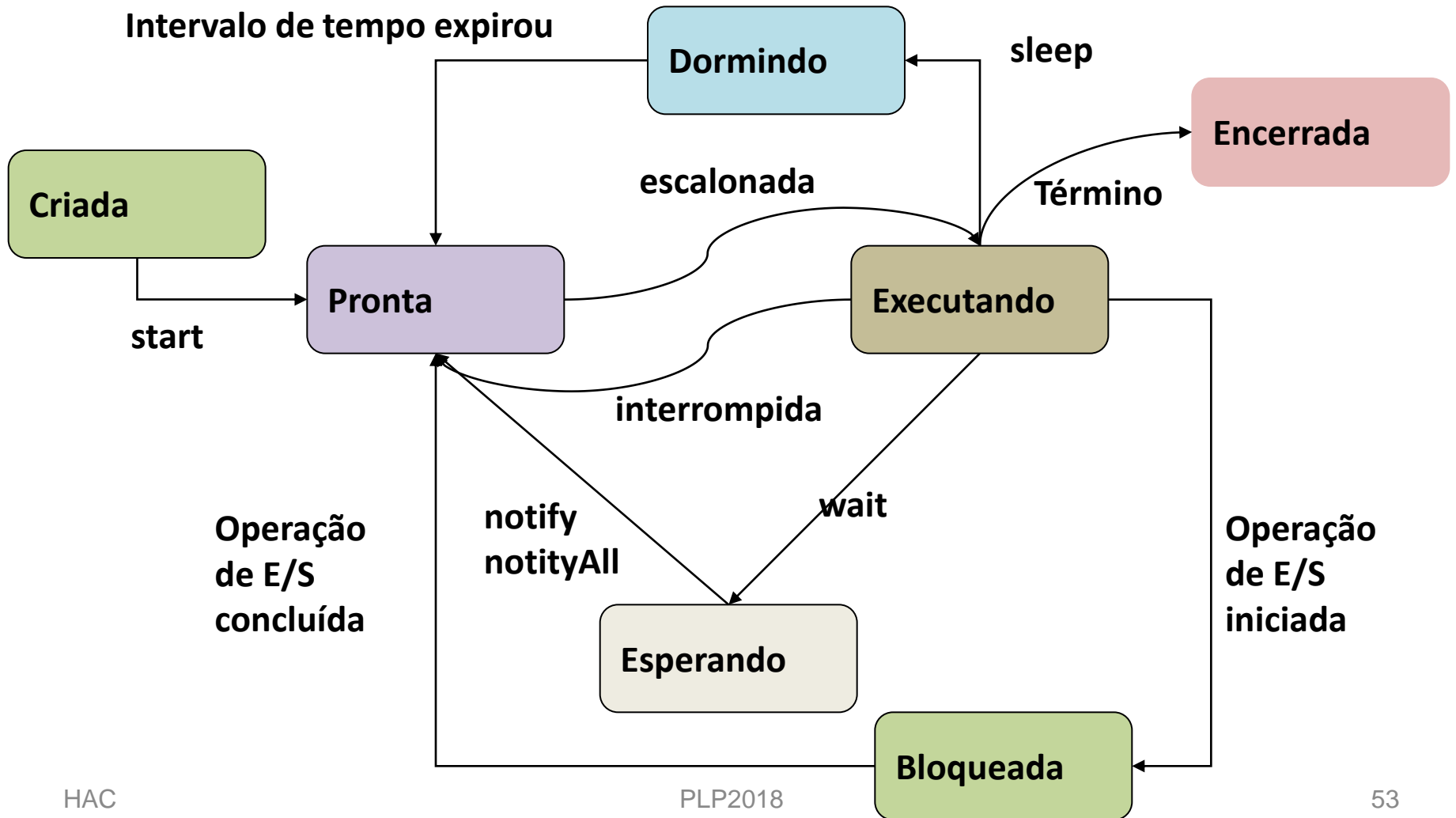
```
class Consumidor extends Thread {  
    private BufferLimitado buffer;  
    public Consumidor(BufferLimitado buffer) {  
        this.buffer = buffer;  
    }  
}
```

Produtor e Consumidor com Métodos Sincronizados

```
public void run() {  
    int elem;  
    while (true) {  
        try {  
            elem = buffer.retirar();  
            System.out.println("consumido: " + elem);  
            Thread.sleep((int)(Math.random() * 1000));  
        } catch (InterruptedException e) {  
            System.out.println("Erro Consumidor: " + e.getMessage());  
        }  
    }  
}
```

Produtor e Consumidor com Métodos Sincronizados

```
public class Fabrica {  
    public static void main(String args[]) throws  
        InterruptedException {  
        BufferLimitado buffer = new BufferLimitado(10);  
        Produtor produtor = new Produtor(buffer);  
        Consumidor consumidor = new Consumidor(buffer);  
        produtor.start();  
        consumidor.start();  
        produtor.join();  
        consumidor.join();  
    }  
}
```



Uso de join – Outro Exemplo

```
// Usando join() para esperar que threads terminem.
class NewThread implements Runnable {
    String name; // nome da thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Inicia a thread
    }
}
```

```
// Este é o ponto de entrada para a thread.  
public void run() {  
    try {  
        for(int i = 5; i > 0; i--) {  
            System.out.println(name + ": " + i);  
            Thread.sleep(1000);  
        }  
        } catch (InterruptedException e) {  
            System.out.println(name + " interrompida.");}  
            System.out.println(name + " finalizando.");  
        }  
    }
```

```
class TesteJoin {  
    public static void main(String args[]) {  
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
        NewThread ob3 = new NewThread("Three");  
  
        System.out.println("Thread One esta ativa: "+ ob1.t.isAlive());  
        System.out.println("Thread Two esta ativa: "+ ob2.t.isAlive());  
        System.out.println("Thread Three esta ativa : "+ ob3.t.isAlive());  
    }  
}
```



```
// espera as threads terminarem
try {
System.out.println("Esperando as threads terminarem.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One esta ativa: "+ ob1.t.isAlive());
System.out.println("Thread Two esta ativa : "+ ob2.t.isAlive());
System.out.println("Thread Three esta ativa : "+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Linha 1 1: 125

```
D:\Projetos JDK>javac TesteJoin.java
javac: file not found: TesteJoin.java
Usage: javac <options> <source files>
use -help for a list of possible options
```

```
D:\Projetos JDK>javac TesteJoin.java
```

```
D:\Projetos JDK>java TesteJoin
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One esta ativa: true
Thread Two esta ativa: true
Thread Three esta ativa : true
Esperando as threads terminarem.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
One finalizando.
Two finalizando.
Three finalizando.
Thread One esta ativa: false
Thread Two esta ativa : false
Thread Three esta ativa : false
Main thread exiting.
```

```
D:\Projetos JDK>
```

Sincronizando blocos

```
public class Conta {  
    private double saldo;  
    // outros métodos e atributos...  
    public void atualiza(double taxa) {  
        synchronized (this) {  
            double saldoAtualizado = this.saldo * (1 + taxa);  
            this.saldo = saldoAtualizado;  
        }  
    }  
    public void deposita(double valor) {  
        synchronized (this) {  
            double novoSaldo = this.saldo + valor;  
            this.saldo = novoSaldo;  
        }  
    }  
}
```