

Programação em Java utilizando elementos para sincronização em Java

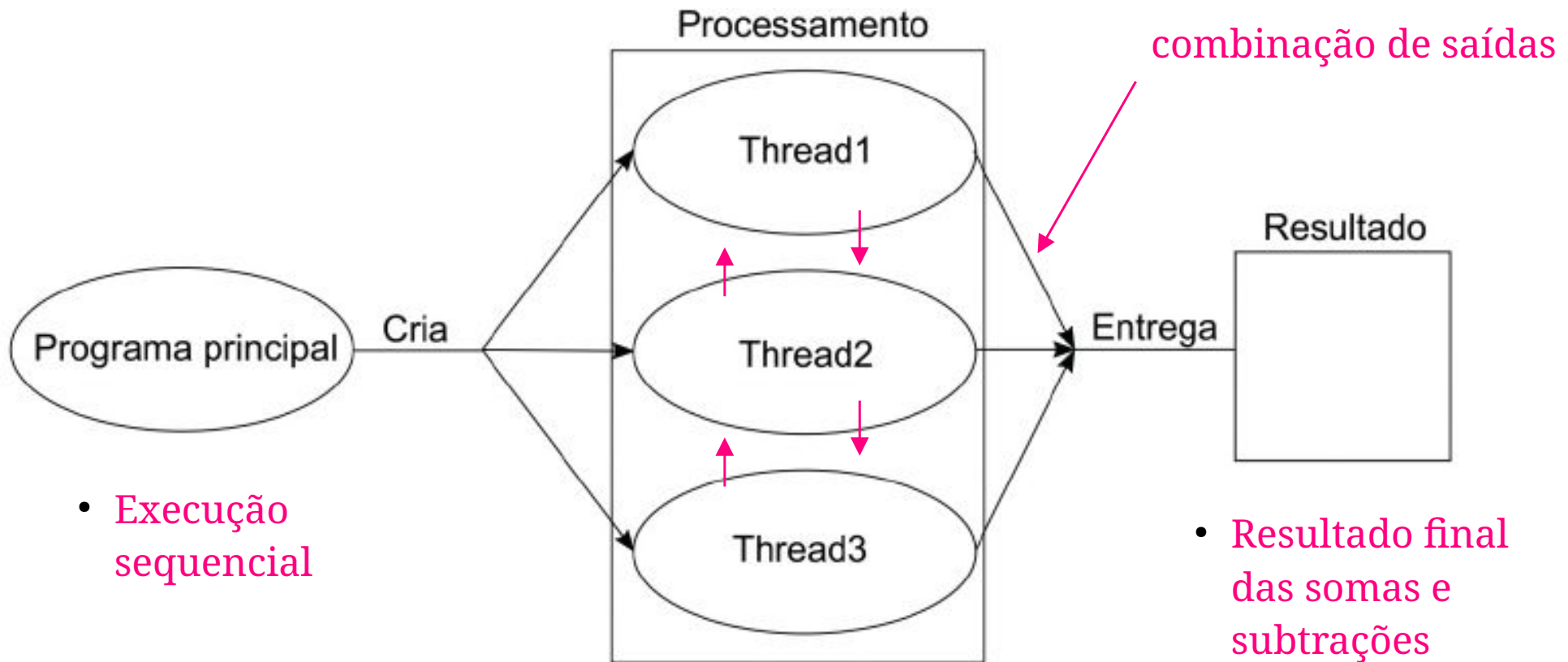
Programação Orientada a Objetos

- Recursos computacionais podem ser comparados, por exemplo, a um supermercado onde existem várias coisas acontecendo em paralelo e de forma sequencial
 - Se 2 clientes forem comprar o mesmo item ao mesmo tempo, será necessário um controle de conflito
- As *threads* podem ser, de certa forma, comparadas a clientes
- Os itens do supermercado podem ser comparados aos recursos computacionais (memória, dispositivos, etc.)

- Usar *threads* acaba sendo um pouco mais complicado
- Ex.: gestão de conflitos
 - Pode ser feita por outros programas, ou
 - Pode ser garantida pelo próprio programador
 - Que cria o código verificando se há conflitos ou não
- Esse *overhead* pode gerar custos computacionais a mais e pode não valer a pena em alguns casos
 - Necessário verificar o que vale a pena paralelizar

overhead é qualquer combinação de tempo de computação excessivo ou indireto, memória, largura de banda ou outros recursos necessários para executar uma tarefa específica

Exemplo



- Somas ou subtrações
- Execução paralela

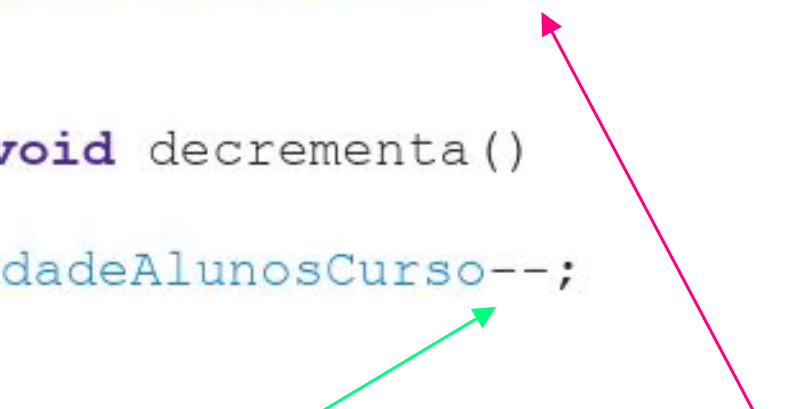
Quadro 2.18 | Código para controle da quantidade de alunos utilizado por diversas threads

```
package U2S3;

public class Contador {
    private int quantidadeAlunosCurso;

    public void incrementa()
    {
        quantidadeAlunosCurso++;
    }

    public void decrementa()
    {
        quantidadeAlunosCurso--;
    }
}
```



- Problema: enquanto uma *thread* soma, outra pode tentar subtrair ao mesmo tempo a mesma variável

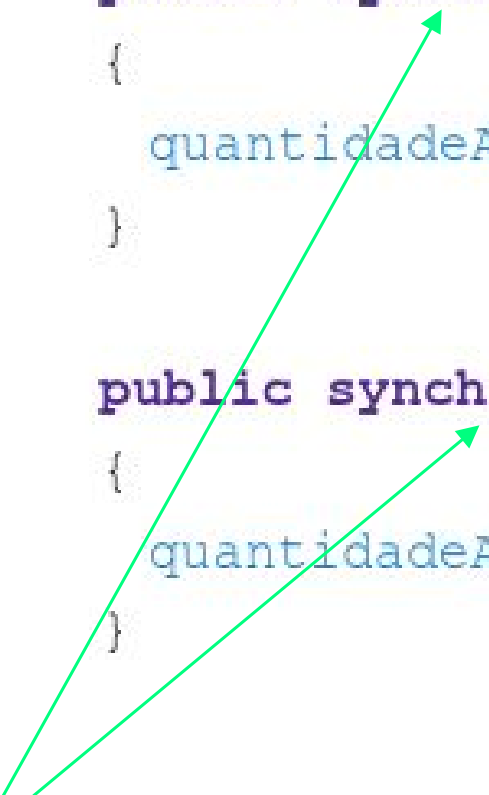
```
package U2S3;
```

```
public class ContadorSync {
```

```
1.     private int quantidadeAlunosCurso;
```

```
2.     public synchronized void incrementa()  
        {  
            quantidadeAlunosCurso++;  
        }
```

```
3.     public synchronized void decrementa()  
        {  
            quantidadeAlunosCurso--;  
        }  
    }
```



- **synchronized** garante que um único recurso seja usado por vez, evitando conflito

Example

```
public class RepetidorThread implements Runnable {
```

```
    private int pausa;
```

```
    private boolean executa;
```

```
    private Thread th;
```

← tipo

```
    public RepetidorThread() {
```

```
        pausa = 10;
```

```
        executa = true;
```

← instância

```
        th = new Thread(this);
```

← início

```
        th.start(); }
```

executa em uma *thread*

```
    public void run() {
```

```
        try {
```

```
            Thread.sleep(pausa);
```

```
        } catch (InterruptedException e) {
```

```
            e.printStackTrace(); }
```

```
        while (executa) {
```

```
            // faz o processamento
```

```
            try {
```

```
                Thread.sleep(pausa);
```

```
            } catch (InterruptedException e) {
```


```
                e.printStackTrace(); } } }
```

← pausa e libera para outros processamentos

```
    public static void main(String[] args) {} }
```

← não faz nada

Classes `Timer` e `TimerTask`

- Classe `Timer` controla a *thread*
 - É executada de forma periódica, em um intervalo
 - O método que inicia o processamento espera uma instância da classe `TimerTask`
 - Classe `TimerTask` faz o processamento
 - A classe é abstrata e implementa a interface `Runnable`
 - É necessário implementar o método `run()`
- 


```
import java.io.*;
import java.util.TimerTask;
import java.util.Timer;
```

- especialização da `TimerTask`
- faz o processamento

```
class RepetidorTimeTarefa extends TimerTask {
```

```
    private String arquivo;
```

```
    public RepetidorTimeTarefa(String parquivo) {
```

construtor

```
        this.arquivo = parquivo;
```

executa em uma *thread*

```
    }
```

```
    public void run() {
```

```
        System.out.println("Buscando arquivo.");
```

```
        try {
```

```
            File f = new File(arquivo);
```

```
            if (f.exists() == true) {
```

```
                System.out.println("Arquivo encontrado.");
```

```
                String line = null;
```

```
                FileReader fileReader = new FileReader(f);
```

```
                BufferedReader bufferedReader = new BufferedReader(fileReader);
```

```
                while ((line = bufferedReader.readLine()) != null) {
```

```
                    System.out.println(line);
```

```
                }
```

```
            }
```

```
        } catch (Exception e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

(continua)

```
import java.util.Timer;
public class RepetidorTimer {
    private Timer timer;
    private RepetidorTimeTarefa tarefa;
    private int pausa;
    public RepetidorTimer() {
        timer = new Timer();
        pausa = 1000;
        tarefa = new RepetidorTimeTarefa("arquivoDados.txt");
        timer.schedule(tarefa, 0, pausa);
    }
    public static void main(String[] args) {
        RepetidorTimer rt = new RepetidorTimer();
    }
}
```

- declara qual timer utilizar
- já importado no slide anterior

declara o objeto `timer`, controla a thread

especialização da `TimerTask`
(vide slide anterior)

0 = atraso para iniciar a tarefa

pausa = intervalo da *thread*

```
$ javac RepetidorTimer.java
```

```
$ java RepetidorTimer
```

```
Buscando arquivo.
```

```
Arquivo encontrado.
```

```
A
```

```
Buscando arquivo.
```

```
Arquivo encontrado.
```

```
A
```

```
Buscando arquivo.
```

```
Arquivo encontrado.
```

```
A
```

```
Buscando arquivo.
```

```
Arquivo encontrado.
```

```
A
```

```
Buscando arquivo.
```

```
Arquivo encontrado.
```

```
A
```

```
^C$
```

considerar que já existe um
arquivo contendo o caracter “A”