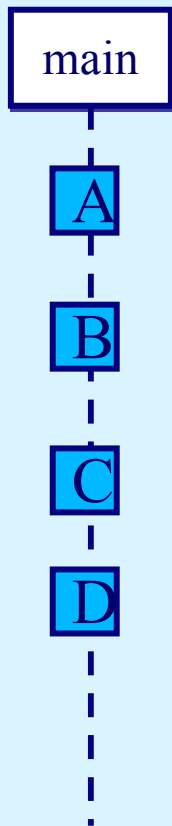


Programação Concorrente em Java

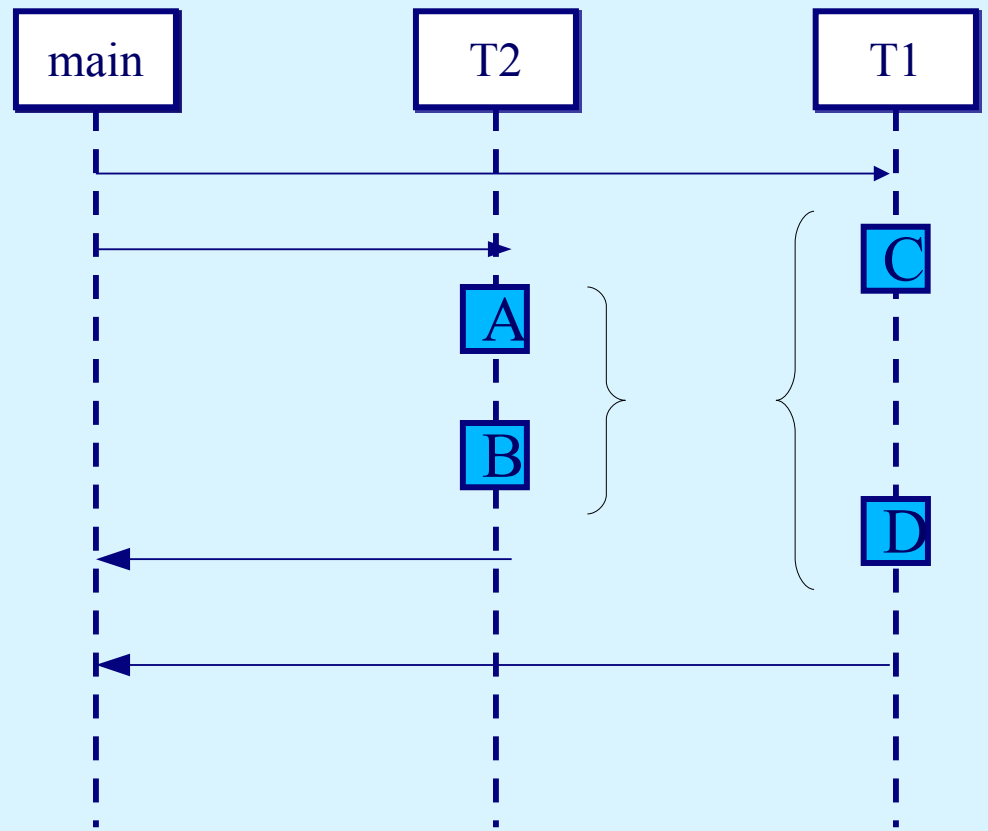
Profa Andréa Schwertner Charão
DELC/CT/UFSM

Execução concorrente

Execução sequencial

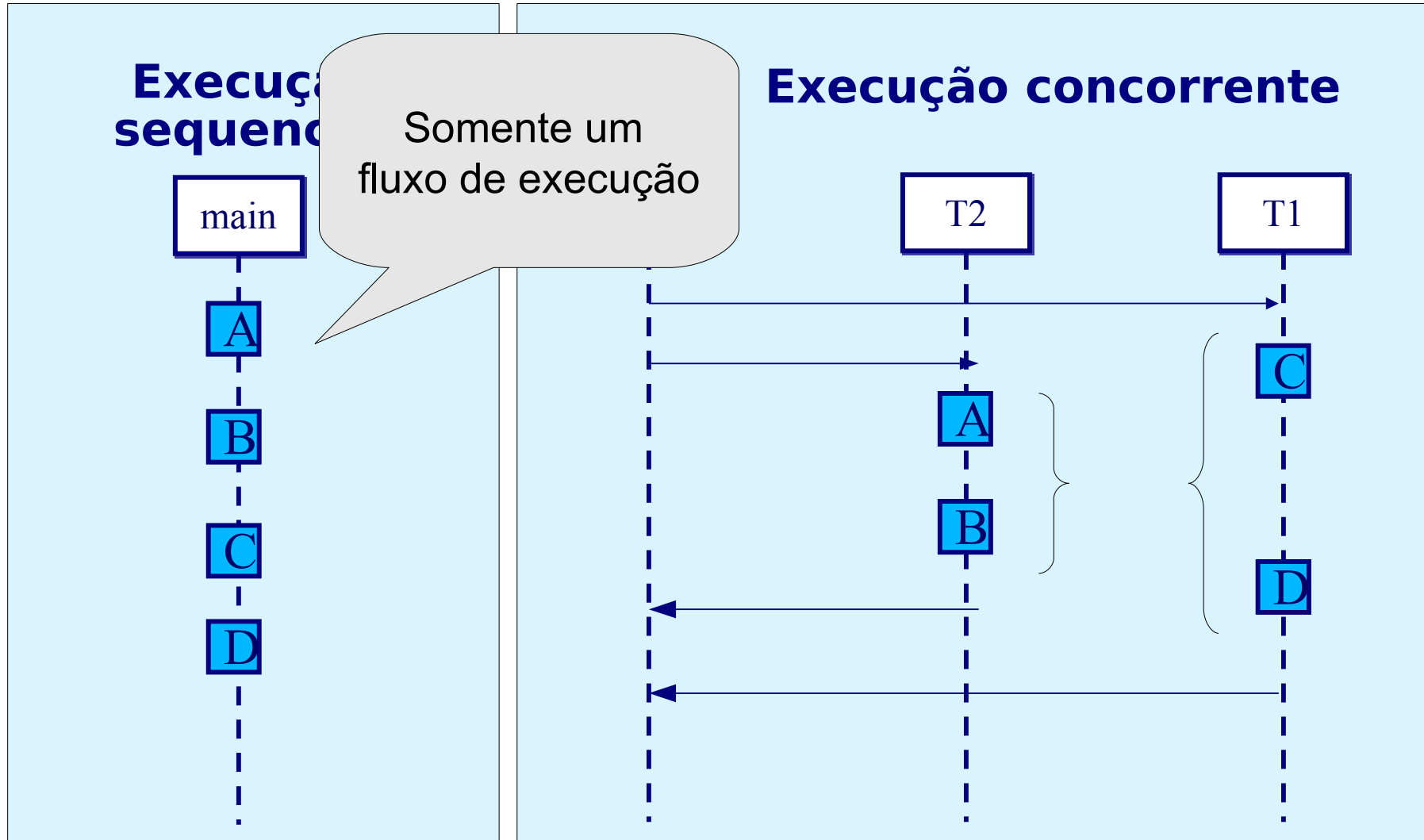


Execução concorrente

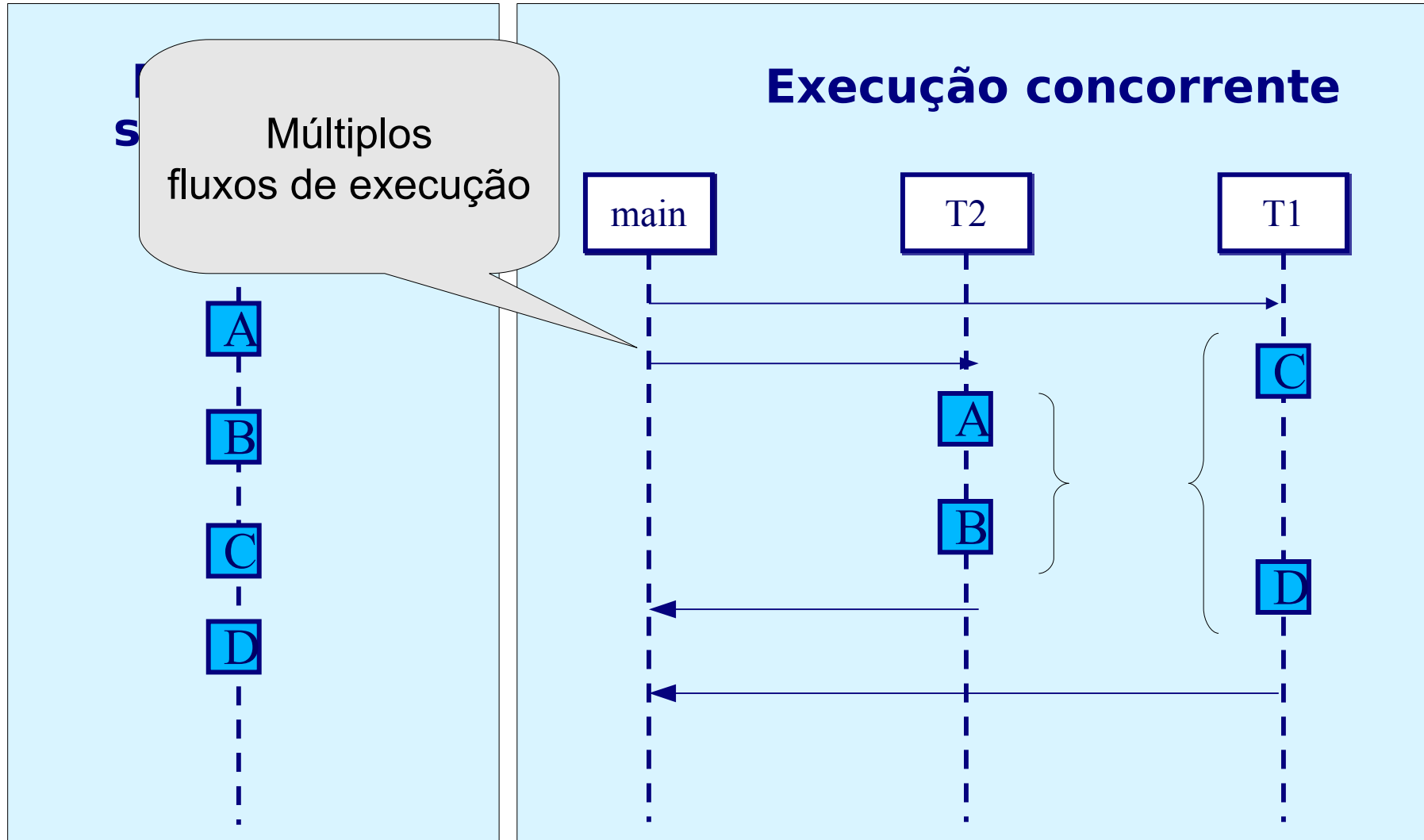


Evolução da execução no tempo

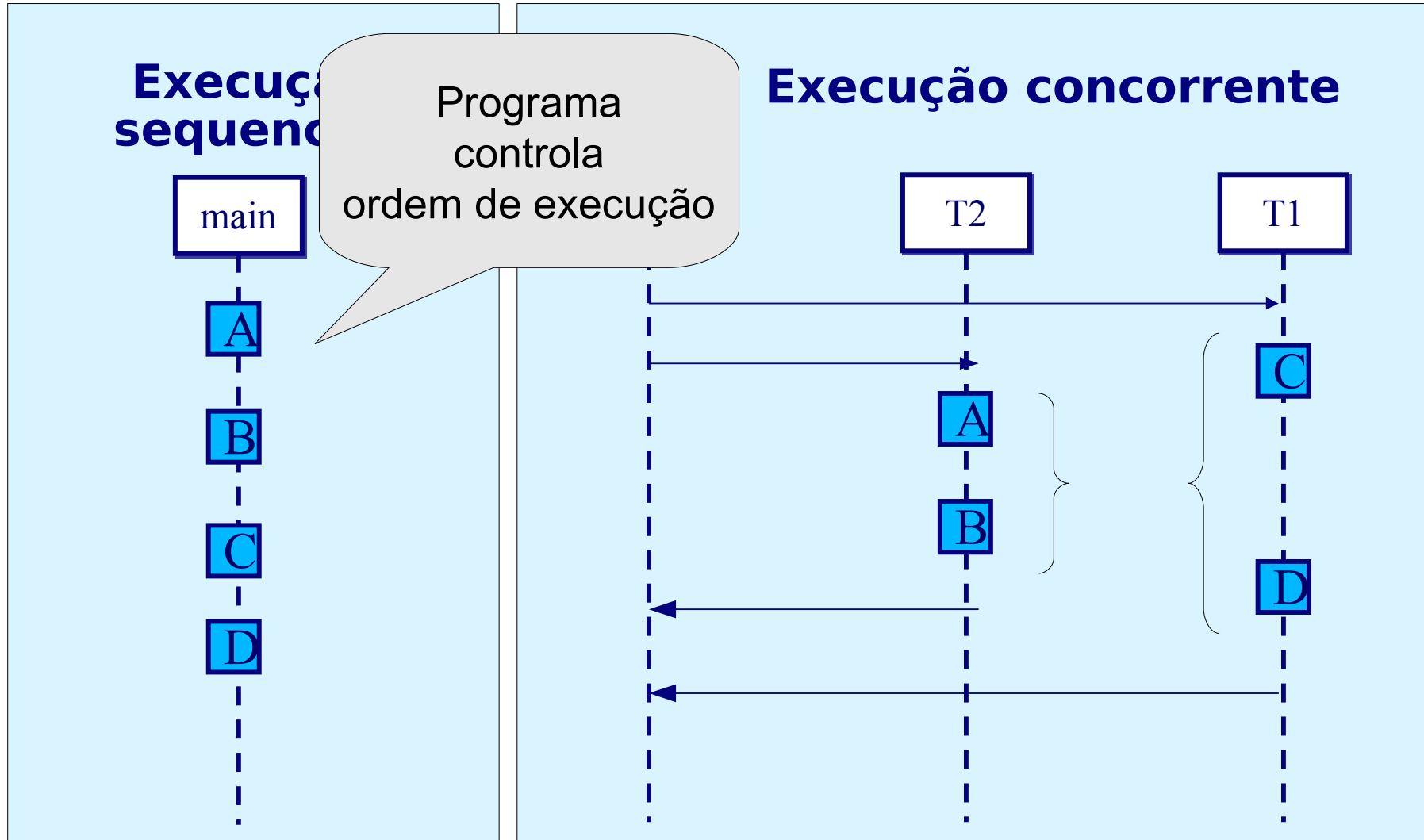
Execução concorrente



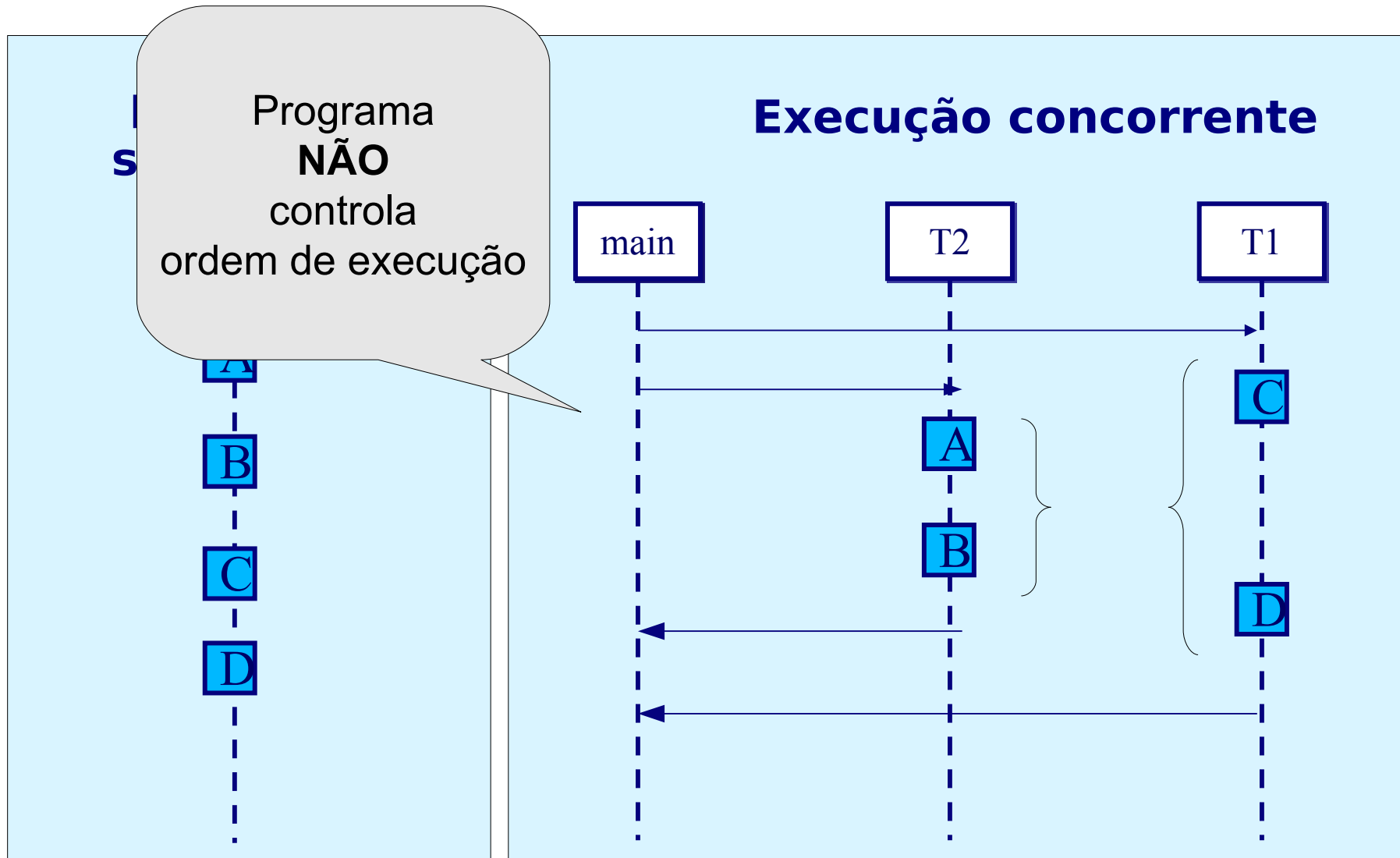
Execução concorrente



Execução concorrente



Execução concorrente



Ex.: C pode ser executado antes ou depois de A.

Execução concorrente

**Execução
sequencial**

main

A

B

C

D

Programação:
estruturas de
sequência, seleção,
repetição

Execução concorrente

T2

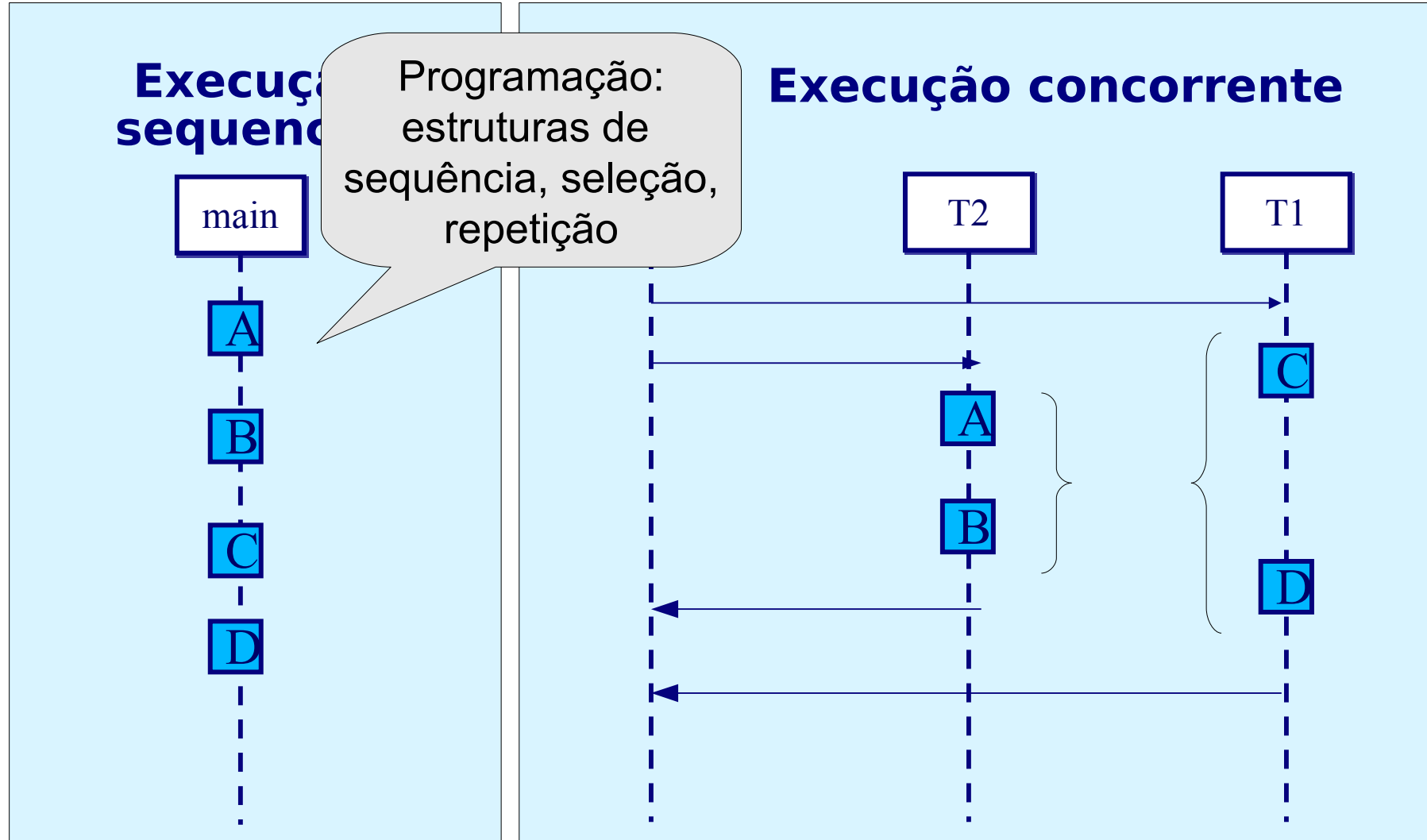
A

B

T1

C

D

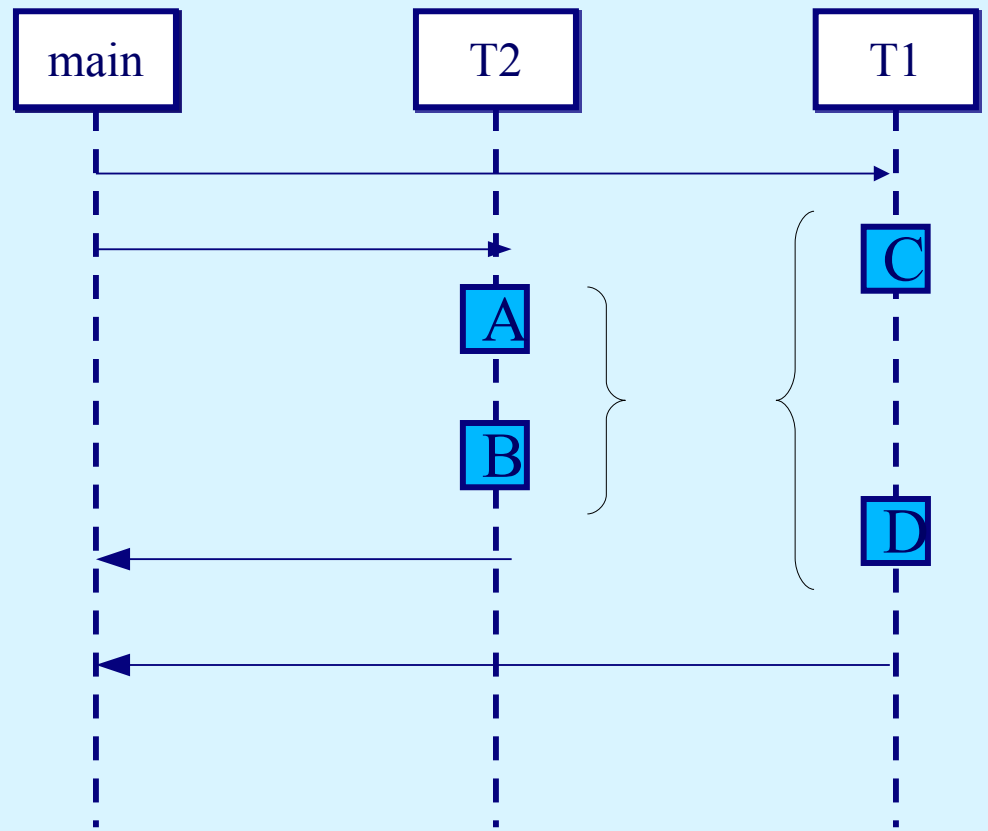


Execução concorrente

Programação:
usa **threads**



Execução concorrente

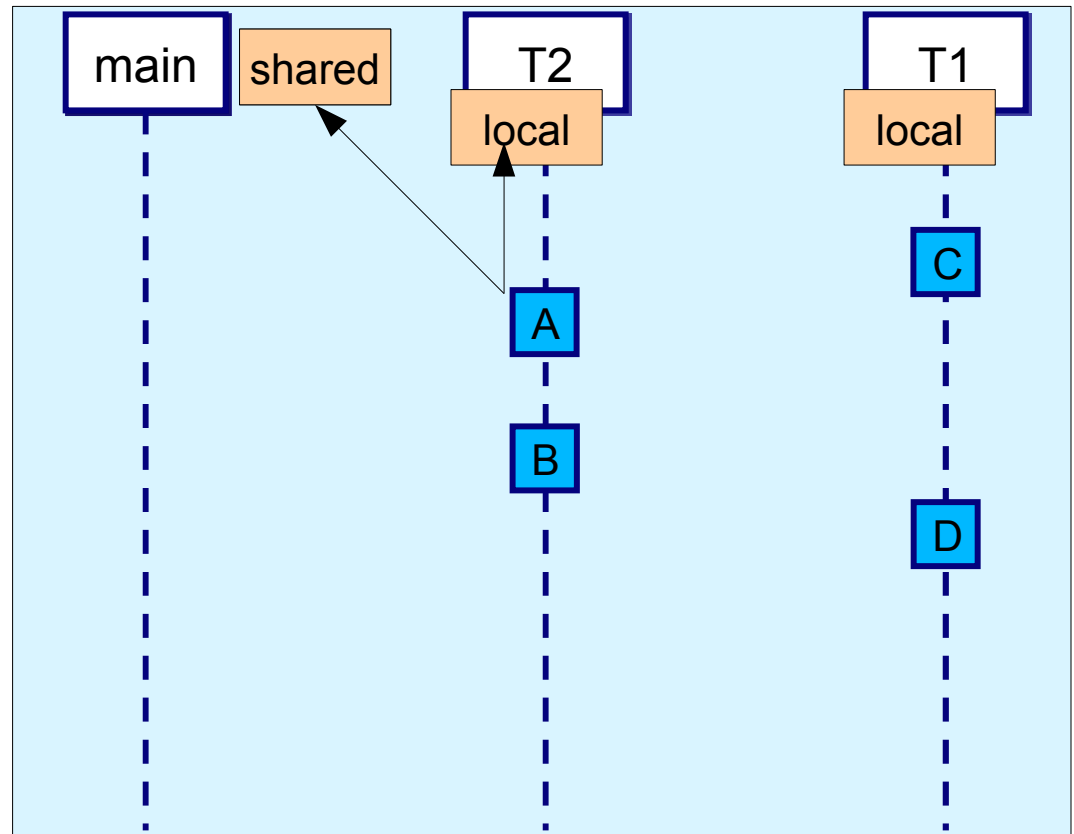


Quando usar programação concorrente?

- Desempenho
 - Ex.: game engines
- Aproveitamento de arquitetura
 - Ex.: multicore
- Disponibilidade/reatividade
 - Ex.: servidores Web, interfaces gráficas
- Organização de código
 - Tarefas independentes em threads independentes

Threads

- Fluxos de execução independentes em um **único programa**
 - Contadores de programa independentes
- Possuem área de memória própria
 - Variáveis locais
- Compartilham memória comum a outras threads



Como programar com threads?

- Especificação do código que deve ser executado
- Especificação de dados próprios ou compartilhados
- Controle: ativação, término, suspensão, etc.
- Sincronização (competição ou cooperação) entre threads, quando necessário

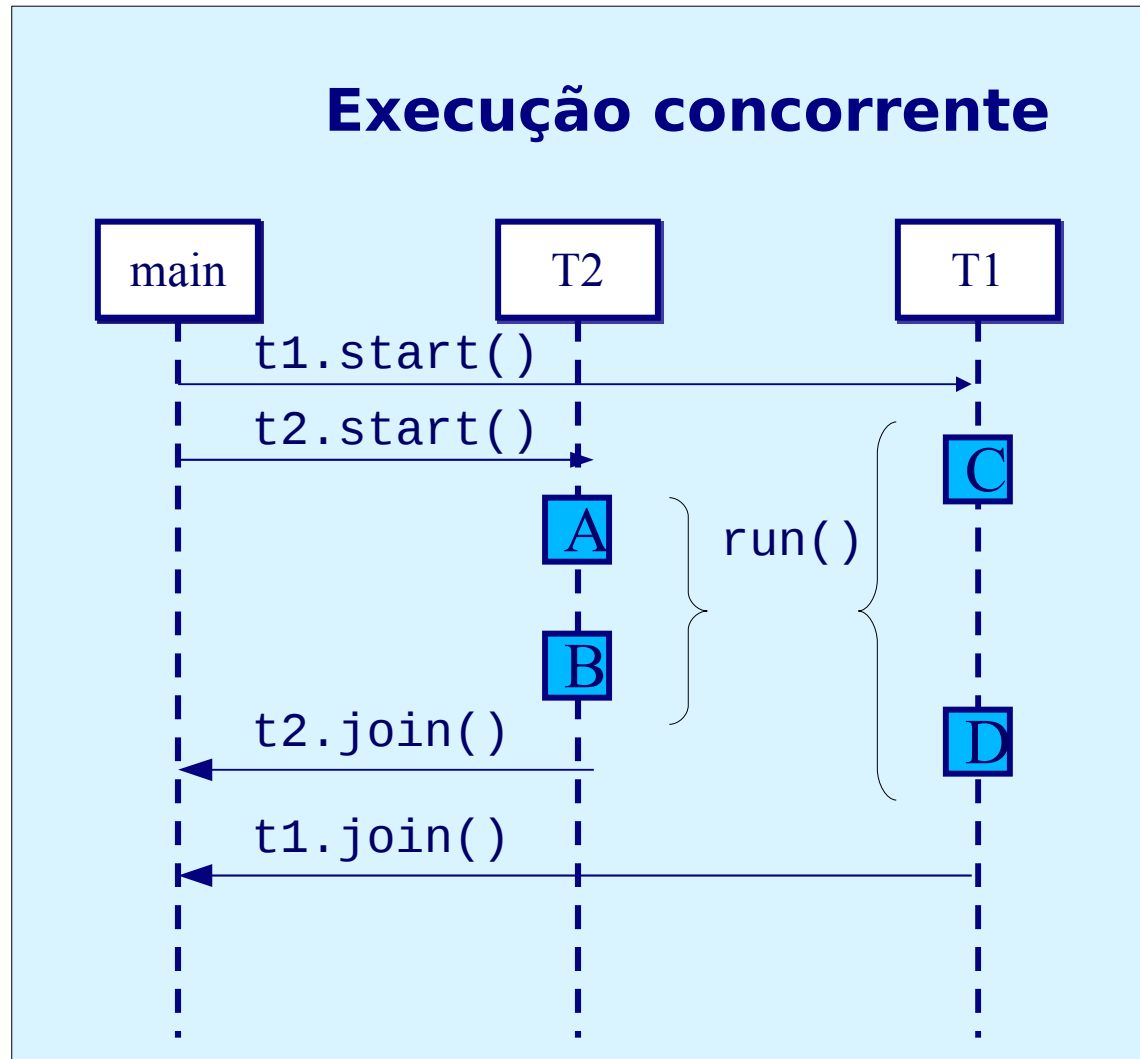
Programação com threads em Java

- Classe Thread
 - Métodos start() e join()
- Interface Runnable e método run()
- Palavra-chave synchronized
- Etc.

Veja mais em:

<http://download.oracle.com/javase/tutorial/essential/concurrency/>

Programação com threads em Java



Programação com threads em Java

- Duas opções:

- 1) Declarar classe que estende Thread

- 2) Declarar classe que implementa interface Runnable

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Trabalhando");  
    }  
}
```

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Trabalhando");  
    }  
}
```

Programação com threads em Java

- Se estender Thread, o uso da classe fica assim:

```
class ThreadApp {  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread();  
        Thread t2 = new MyThread();  
        t1.start();  
        t2.start();  
    }  
}
```

Programação com threads em Java

- Se implementar Runnable, o uso da classe fica assim:

```
class ThreadApp {  
    public static void main(String[] args) {  
        MyRunnable r = new MyRunnable();  
        Thread t1 = new Thread(r);  
        Thread t2 = new Thread(r);  
        t1.start();  
        t2.start();  
    }  
}
```

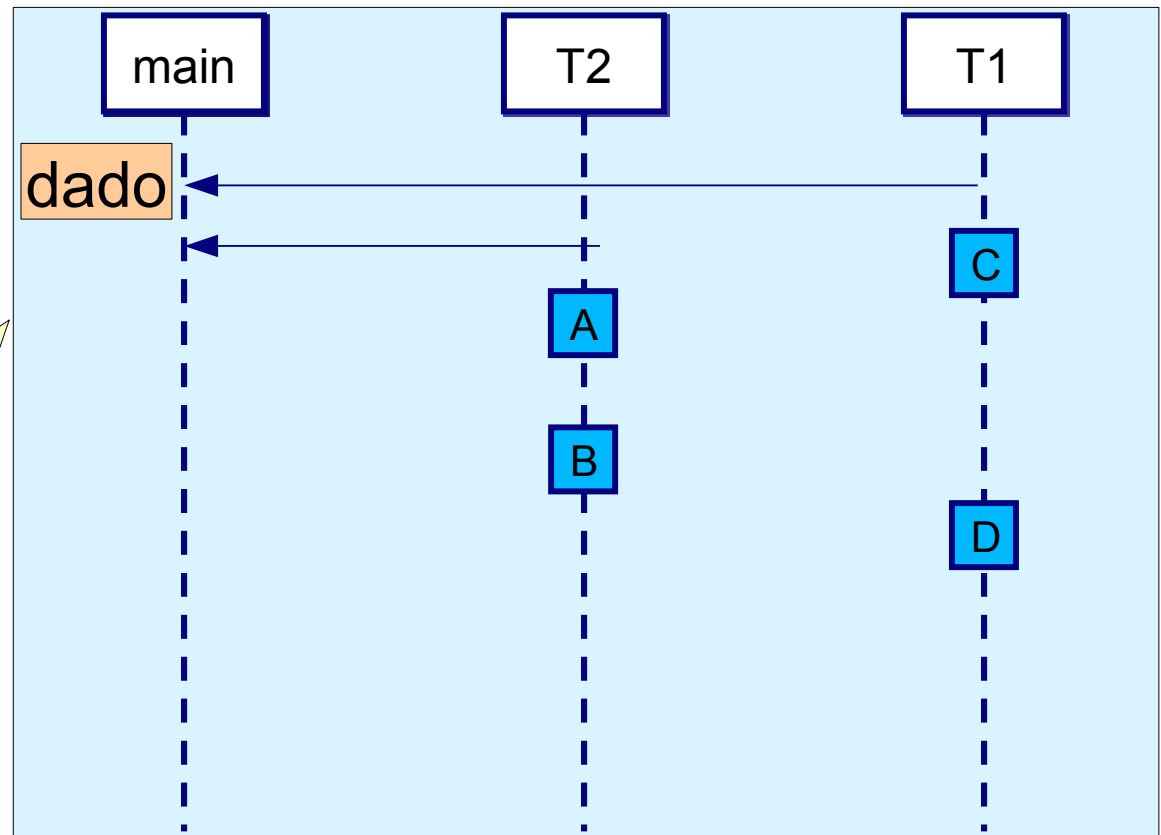
Sincronização entre threads em Java

- **Competição** no acesso a dados compartilhados

Variável de classe
(static)

E/OU

Referência para o
mesmo objeto



Sincronização entre threads

- Situação típica: 2 ou mais threads fazem operação sobre dado compartilhado

ContaBancaria

float saldo;

void deposita(float)
void retira(float)

main

ContaBancaria c;

c = new
ContaBancaria(1000);

T2

T1

c

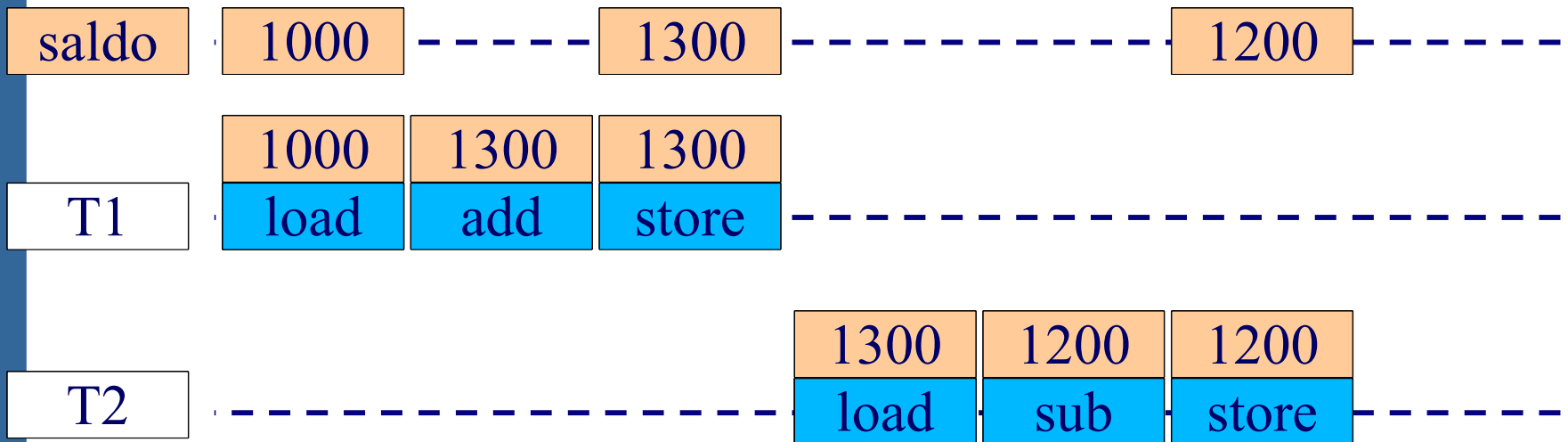
c

c.deposita(300);

c.retira(100);

Sincronização entre threads

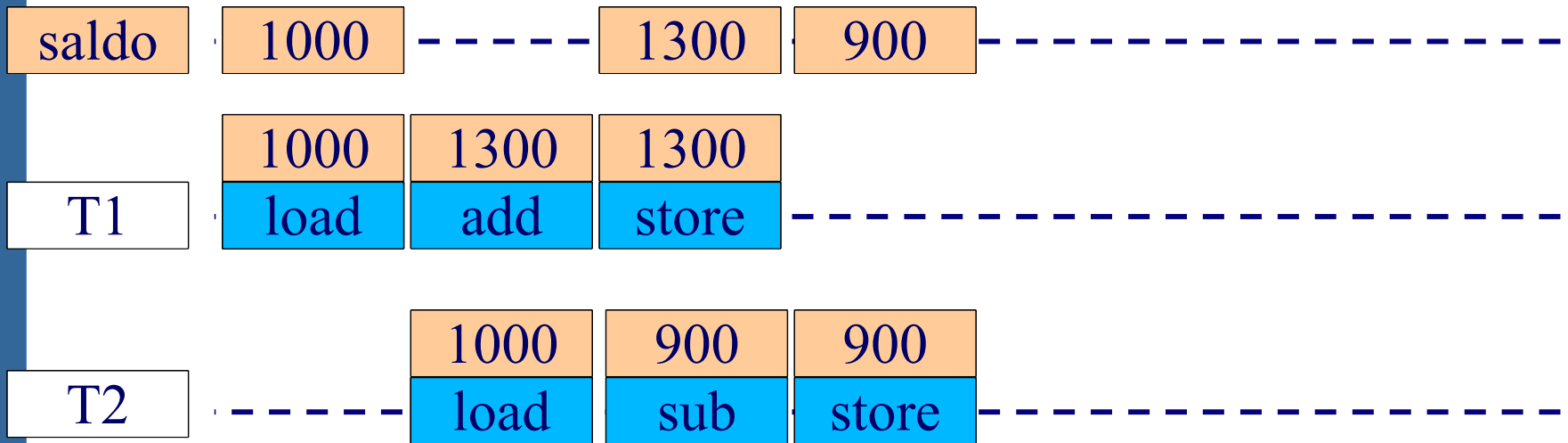
- Situação 1: T1 primeiro, T2 depois



saldo = 1200

Sincronização entre threads

- Situação 2: T1 e T2 concorrentemente



saldo = 900 ?!

Sincronização entre threads

- Problema: **inconsistência de dados**
- Em outras palavras...
 - **Condição de corrida**: operações sobre recursos compartilhados que podem levar a inconsistências dependendo da ordem de execução
 - **Seção crítica**: trecho do programa que contém operações que podem levar a inconsistências
- Solução: **exclusão mútua**
- Em outras palavras...
 - Garantir atomicidade e acesso exclusivo ao recurso na seção crítica

Exclusão mútua em Java

■ Compartilhamento do objeto

```
public static void main(String[] args) {  
  
    Conta c = new Conta(100f);  
  
    ThreadDeposita td = new ThreadDeposita(c);  
    ThreadRetira tr = new ThreadRetira(c);  
  
    td.start();  
    tr.start();  
  
}
```

Exclusão mútua em Java

■ Thread faz acesso ao objeto

```
class ThreadDeposita extends Thread {  
    private Conta c;  
  
    ThreadDeposita(Conta c) {  
        this.c = c;  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++)  
            c.deposita(100f);  
    }  
}
```

Exclusão mútua em Java

■ Uso de **synchronized**

```
class ContaBancaria {  
    private float saldo;  
    ContaBancaria(float v) {  
        saldo = v;  
    }  
    void synchronized deposita(float v) {  
        saldo += v;  
    }  
    void synchronized retira(float v) {  
        saldo -= v;  
    }  
}
```

Métodos
synchronized
executam
em exclusão mútua
sobre o mesmo objeto
compartilhado

Exclusão mútua em Java

- Métodos synchronized em um mesmo objeto são executados em exclusão mútua
- Só têm efeito em objetos compartilhados (mais de 1 thread referenciando mesmo objeto)
- Limitação à concorrência
- Usar com cuidado
- Em S.O. serão vistos outros mecanismos de exclusão mútua

Threads e exceções em Java

- Alguns métodos da classe Thread exigem manipulação de exceções
- Por exemplo:
 - `join()` throws `InterruptedException`:
método para esperar pelo término da thread
 - `sleep(long millis)` throws `InterruptedException`:
método para interromper execução da thread durante um certo tempo
- Com esses métodos:
 - OU capturar e tratar a exceção usando `try/catch`
 - OU passar a exceção adiante usando `throws`