

Paradigmas de Linguagens de Programação

Prof. Sergio D. Zorzo

Departamento de Computação - UFSCar

1º semestre / 2013

Aula 10

Material adaptado do Prof. Daniel Lucredio

Programação concorrente

MS-DOS version 1.25
Copyright 1981,82 Microsoft, Inc.

Command v. 1.18
Current date is Tue 1-01-1980
Enter new date:
Current time is 1:01:56.20
Enter new time:

A: 

Concorrência

- Sistema Operacional DOS (Disk Operating System)
 - Baseado em linha de comando
 - Tela preta
 - Monotarefa
- Windows 3.1
 - Multitarefa
 - Várias janelas “abertas” ao mesmo tempo
 - Usuário podia alternar entre as tarefas

Concorrência

- Mais de um fluxo de controle ativo
 - Em contraste com execução sequencial
- Unidades concorrentes podem ser executadas em:
 - Um único processador
 - Vários processadores que compartilham uma memória
 - Vários processadores independentes, sem compartilhamento de memória

Concorrência

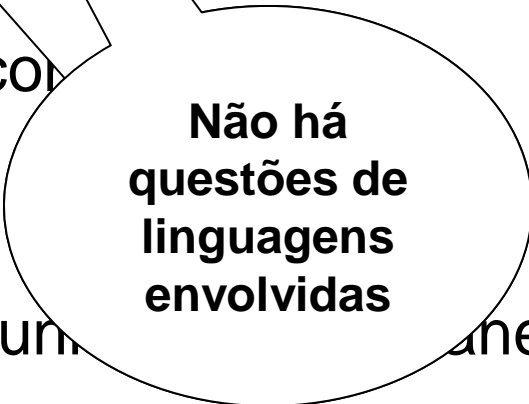
- Vantagens:
 - Quando tarefas podem ser executadas de forma independente
 - Sem que uma precise aguardar algum evento gerado por outra
 - Pode fazer uso de múltiplos processadores
 - Maior interatividade
 - Execução em segundo plano
 - Execução periódica (timer)
- Desvantagens:
 - Contra-intuitivo
 - Não-determinístico
 - Modelo conceitualmente complexo
 - Problemas difíceis de detectar

Programação concorrente

- Em nível de programa
 - Execução de dois ou mais programas simultaneamente
- Em nível de instrução
 - Execução de duas ou mais instruções de máquina simultaneamente
- Em nível de comando
 - Execução de dois ou mais comandos simultaneamente
- Em nível de unidade
 - Execução de duas ou mais unidades simultaneamente

Programação concorrente

- Em nível de programa
 - Execução de dois ou mais programas simultaneamente
- Em nível de instrução
 - Execução de duas ou mais instruções de máquina simultaneamente
- Em nível de comando
 - Execução de dois ou mais comandos simultaneamente
- Em nível de unidade
 - Execução de duas ou mais unidades simultaneamente



**Não há
questões de
linguagens
envolvidas**

Programação concorrente

- É necessário fazer uma distinção
 - Concorrência física
 - Quando há realmente duas execuções simultâneas
 - Concorrência lógica
 - Quando, para todos os efeitos, há duas execuções simultâneas
 - Mesmo que na realidade haja uma execução real preemptiva
- Para o programador tanto faz
 - Devido ao não-determinismo da execução
Programador dificilmente consegue assumir algum tipo de lógica no controle da execução intercalada

Programação concorrente

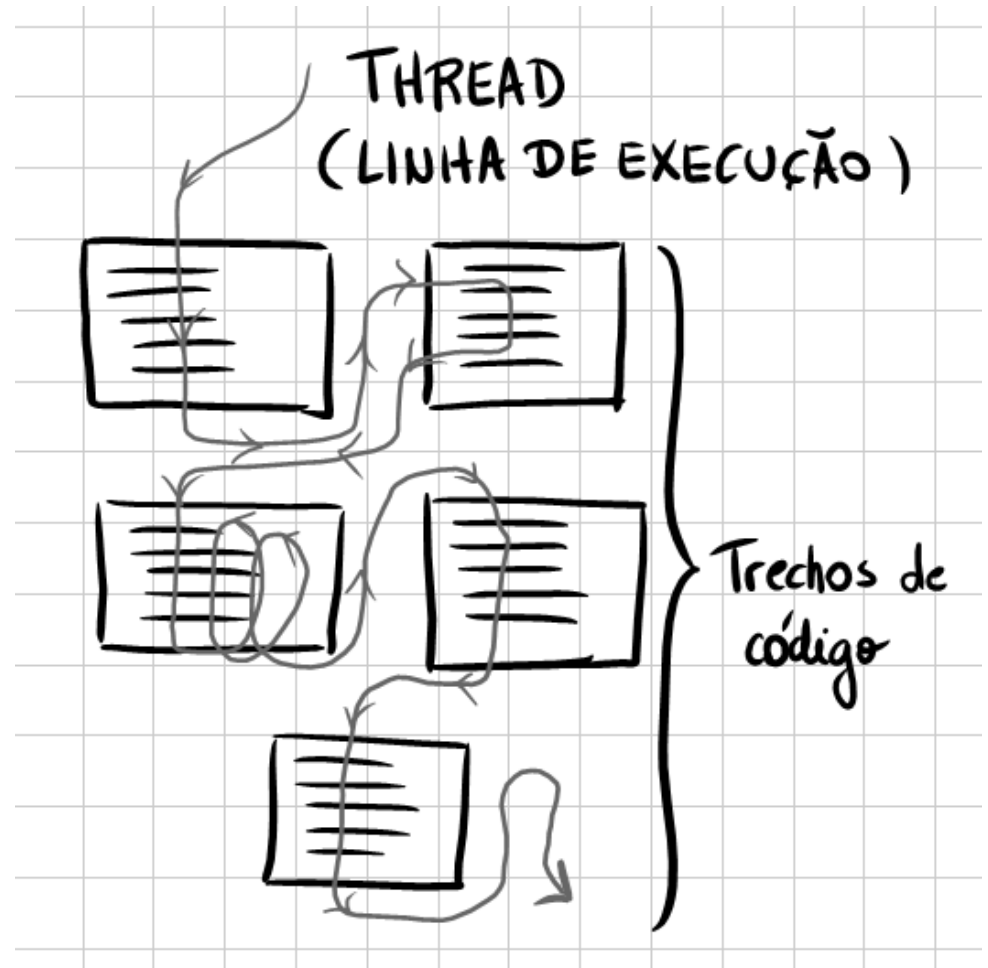
- O modelo (ou paradigma) de programação concorrente diz respeito a:
 - Formas com as quais um programador especifica execução “em paralelo”
 - Conceito de tarefas / programação multilinhas
 - Técnicas para lidar com cooperação/competição
- Não trataremos aqui dos efeitos reais dos programas na arquitetura do computador
 - Para todos os efeitos, não sabemos como as tarefas serão distribuídas

Programação concorrente

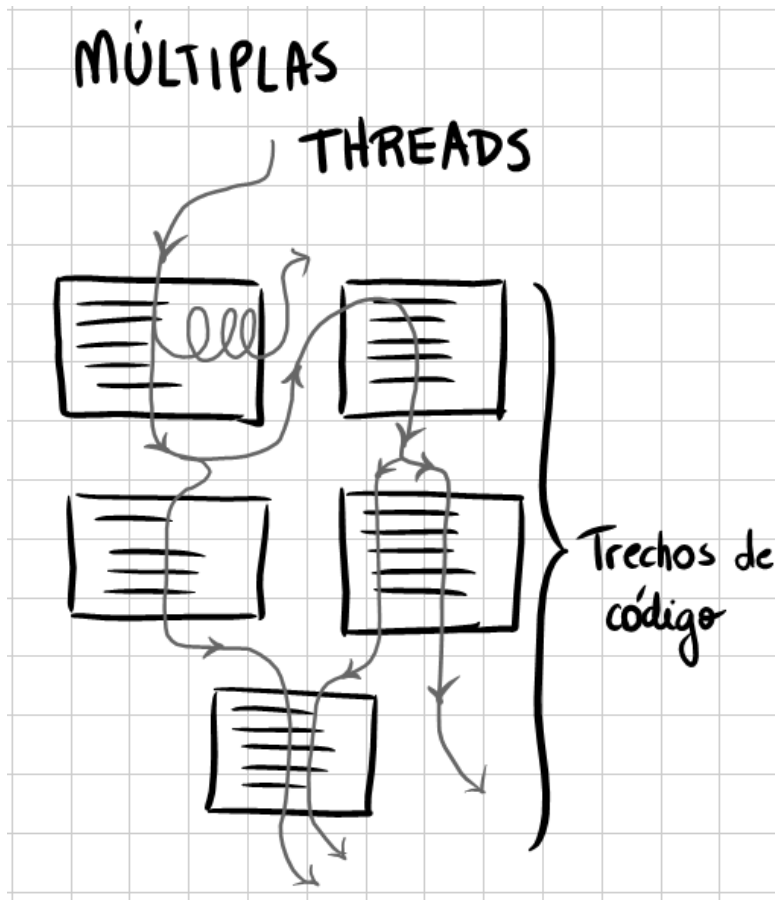
- Conceitos
 - Thread
 - Literalmente: “linha” de execução
 - Execução sequencial possui apenas uma “linha”
 - Execução concorrente pode possuir mais de uma “linha”
 - Tarefa
 - Unidade de programa que pode estar em execução concorrente com outras unidades
 - Pode se comunicar com outras tarefas através de variáveis não-locais, compartilhadas, mensagens ou parâmetros
 - Cada tarefa pode originar uma thread

Threads

- Uma thread é uma linha de execução
- Em uma thread, uma instrução executa apenas após a outra



Multi-threads



- É possível “bifurcar” uma thread, criando uma nova linha de execução
- Múltiplas threads executam em “paralelo”

Multi-threads

- A idéia do multi-threading é similar à idéia de multiprocessamento nos SOs modernos
- Enquanto com processos o SO fica responsável por “paralelizar” a execução
- Com threads esse trabalho fica por conta de um único processo
 - No caso do Java, por exemplo, é a JVM que gerencia a execução paralela

Processos vs Threads

- Processos podem fazer uso explícito da existência de múltiplos processadores
- Threads normalmente rodam em um único processo e – portanto – um único processador
- A bifurcação de uma thread é mais rápida, pois não é necessário repartir todos os recursos do SO
 - Também conhecida como processo leve (lightweight process)

Programação concorrente

- Conceitos
 - Sincronização
 - Mecanismo de controle da ordem de execução das tarefas
 - Tipos de sincronização
 - Cooperação
 - Quando uma tarefa A precisa esperar o término de outra tarefa B para prosseguir
 - Competição
 - Quando duas tarefas A e B requerem o uso de algum recurso que não pode ser usado simultaneamente

Exemplo: cooperação

- Problema produtor-consumidor:
 - Uma unidade produz um dado ou recurso usado por outra unidade
 - Dados são depositados em um buffer pelo produtor e removidos pelo consumidor
 - A sequência de retiradas e depósitos deve ser sincronizada
 - Os usuários do dado compartilhado devem cooperar para que o buffer seja usado corretamente

Exemplo: competição

- Cenário:
 - Tarefa A: adicionar 1 a TOTAL
 - A1. Ler valor de TOTAL
 - A2. Somar $TOTAL + 1$
 - A3. Atribuir o resultado a TOTAL
 - Tarefa B: multiplicar TOTAL por 2
 - B1. Ler valor de TOTAL
 - B2. Multiplicar $TOTAL * 2$
 - B3. Atribuir o resultado a TOTAL
- A vale inicialmente 3
 - Sem sincronização de competição, 3 resultados são possíveis

Exemplo: competição

- Resultado 1: Tarefa A completa antes que B comece
 - $TOTAL = 8$ (correto)
- Resultado 2: A lê TOTAL primeiro, mas B lê TOTAL antes que A atribua o resultado da soma
 - $TOTAL = 6$
- Resultado 3: B lê TOTAL primeiro, mas A lê TOTAL antes que B atribua o resultado da multiplicação
 - $TOTAL = 4$

Programação concorrente

- Métodos para sincronização
 - Semáforos
 - Monitores
 - Mensagens

Semáforos

- Pode ser usado para sincronização de cooperação ou competição
 - Consiste basicamente em uma fila
 - Nas quais as tarefas são “enfileiradas”, no aguardo de algum evento que liberem sua execução
- Exemplo:
 - Produtor-consumidor (cooperação)
 - Variáveis (booleanas) `bufferVazio` e `bufferCheio`
 - Tarefas produtoras aguardam `bufferCheio` se tornar “falso”. Enquanto isso, não executam. Ao executar, modificam `bufferVazio` para “verdadeiro” (caso necessário) e podem modificar `bufferCheio` para “verdadeiro”
 - Tarefas consumidoras aguardam `bufferVazio` se tornar “falso”. Enquanto isso, não executam. Ao executar, modificam `bufferCheio` para “falso” (caso necessário) e podem modificar `bufferVazio` para “verdadeiro”

Monitores

- Mecanismo que encapsula unidades de código, prevenindo acesso simultâneo ao trecho encapsulado
 - Somente uma thread pode estar ativa dentro do monitor em um determinado momento
 - Obs: lembrando que em ambientes monotarefas isso já acontece fisicamente! Mas aqui imaginamos que estão realmente estão em paralelo
- Um monitor permite a exclusão mútua seletiva
 - Entre grupos de tarefas, e não entre todas as tarefas

Mensagens

- Consiste no envio de dados entre tarefas
 - Quando não existe acesso compartilhado a uma área comum
 - Ex: processadores diferentes ou sistemas distribuídos
- Mecanismos simples, como send / receive
 - Problema: a comunicação deve ser síncrona, ou seja, o receptor precisa estar aguardando o envio
 - Técnicas de “conversa” para sincronizar
 - Requisição-confirmação-resposta-confirmação

Principais Razões para o Uso de Programação Concorrente

- Possibilitar a resolução de problemas que são intrinsecamente paralelos (no contexto de sistemas operacionais- arquivo sendo compartilhado por vários processos, que pode ser lido por um número ilimitado de processos)
- Explorar o paralelismo presente no Sistema Computacional para executar programas mais rápidos (útil quando a velocidade da execução sequencial não é suficiente para realizar os cálculos necessários)
ex: animação, simulação de efeitos físicos

Formas de Concorrência:

- A) Método de paralelização
 - - homogênea - todos os processadores executam o mesmo código
 - - heterogênea - cada processador executa um código diferente
- B) Granularidade (quantidade de processamento envolvido na tarefa)
 - - fina (“fine”)
 - - grossa (“coarse”)

Paralelismo Homogêneo

- de granularidade fina

```
forall i=1 to 100 {  
    a[i] = b[i] + 1;  
}
```

- de granularidade grossa

```
forall i=1 to 100 {  
    a = res_parcial (b,c,d) ; // grande processamento  
    b[i] = b[i] + 1;  
    c[i] = c[i] + 1  
}
```

Paralelismo Heterogêneo

- de granularidade fina

parbegin

 a = a + c; /* bloco 1 */

also

 e = e + c; /* bloco 2 */

parend

- de granularidade grossa

parbegin

 a = a + sub(c); /* bloco 1 */

also

 b = b + sub(d); /* bloco 2 */

also

 e = e + sub©; /* bloco 3 */

parend

Obtenção de Paralelismo / Concorrência

- Pela utilização de um compilador paralelizante
- Pela programação explícita do paralelismo / concorrência

→ o que é melhor ????

- Programa fonte -> paralelização automática -> análise de desempenho -> modificação do código ou de parâmetros do paralelizador -> fonte modificado -> paralelização automática
- Código para execução paralela / Concorrente

Programação Paralela Explícita

- Por chamadas de rotinas de uma biblioteca que trata dos recursos de paralelismo / concorrência
- Através de diretivas para o compilador
- Através de construções paralelas oferecidas pela linguagem de programação

Paralelização Automática

Análise do programa sequencial

+

Detecção de fontes de paralelismo

+

Reestruturação do código

Há 3 níveis de paralelismo em um programa:

- Nivel de subrotina / unidade
- Nivel de bloco básico
- Nivel de " loops"

Paralelismo a nível de Loop

- Não pode ocorrer “dependência de dados”

(1) $a = b + 1;$

(2) $c = a + 1;$

O Comando (2) só pode ocorrer após a execução do comando (1)

Tipos de dependência de dados

- Dependência de fluxo (dependência verdadeira)

(1) $A = B + C$

(2) $D = A - E$

- Anti - Dependência

(1) $A = B + 1$

(2) $B = C + D$

- Dependência de Saída

(1) $A = B + 1$

(2) $A = C - E$

Transformações de Código para Eliminar dependência de dados

- Reordenação de Loops

```
for (j=1; j<N; j++)  
  for (i=1; i<N; i++)  
    a[ j ][ i ] = a [j-1] [ i ] + b [ j ] [ i ];
```

Troca- se por:

```
forall (i=1; i<N; i++)  
  for (j=1; j<N; j++)  
    a[ j ][ i ] = a [j-1] [ i ] + b [ j ] [ i ];
```

Transformações de Código para Eliminar dependência de dados

- Distribuição de Loops

```
for ( i=1; i<N; i++) {  
    a[ i +1] = b [i-1] + c [ i ]; // (1)  
    b [ i ] = a [ i ] * k;          // (2)  
    c [ i ] = b [ i ] -1 ;          // (3)  
}
```

Troca- se por:

```
    for ( i=1; i<N; i++) {  
        a[ i +1] = b [i-1] + c [ i ]; // (1)  
        b [ i ] = a [ i ] * k;          // (2)  
    }  
    forall ( i=1; i<N; i++) {  
        c [ i ] = b [ i ] -1 ;          // (3)  
    }
```

Transformações de Código para Eliminar dependência de dados

- “Note Splitting”

```
for ( i=1; i<N; i++) {  
    a[ i ] = b [ i ] + c [ i ];           // (1)  
    d [ i ] = a [ i - 1 ] * a [ i + 1 ]; // (2)  
}
```

Troca- se por:

```
forall ( i=1; i<N; i++) {  
    temp [ i ] = a [ i +1 ]  
}  
forall ( i=1; i<N; i++) {  
    a[ i ] = b [ i ] + c [ i ];           // (1)  
}  
forall ( i=1; i<N; i++) {  
    d[ i ] = a [ i - 1 ] + temp [ i ];    // (2)  
}
```

Programação concorrente em Java

Threads em Java

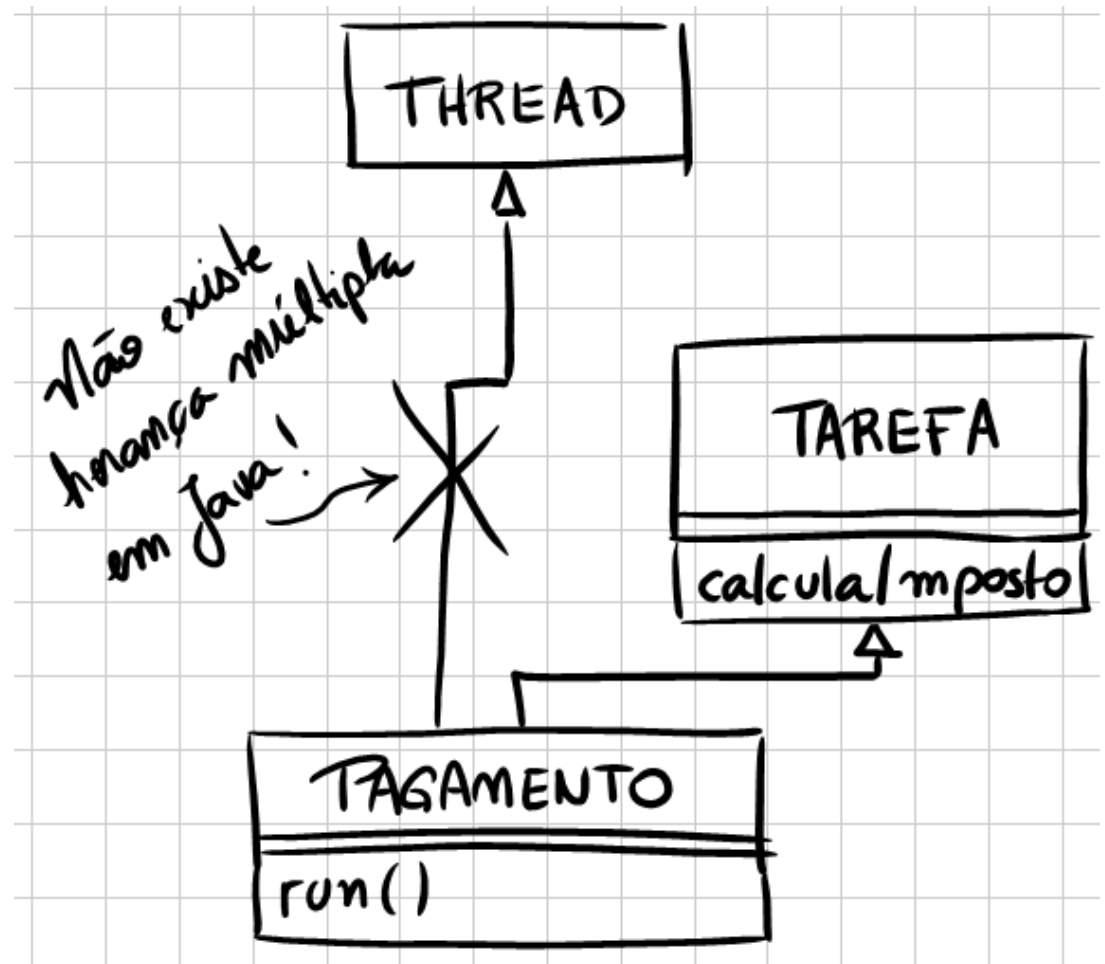
- Cada programa possui ao menos uma thread
 - Principal – criada pela JVM
 - Pode ser dividida em duas programaticamente
- Existe uma API para que o programador possa explicitamente criar essa divisão
 - Concorrência básica: `java.lang.Thread`
 - Concorrência alto nível: `java.util.concurrent`

Overview das classes

- `java.lang.Thread`
 - Implementa as funcionalidades básicas de uma thread
 - Permite iniciar uma nova linha de execução em paralelo à thread principal
 - Seu uso é através de herança
 - O programador cria uma subclasse de `java.lang.Thread`

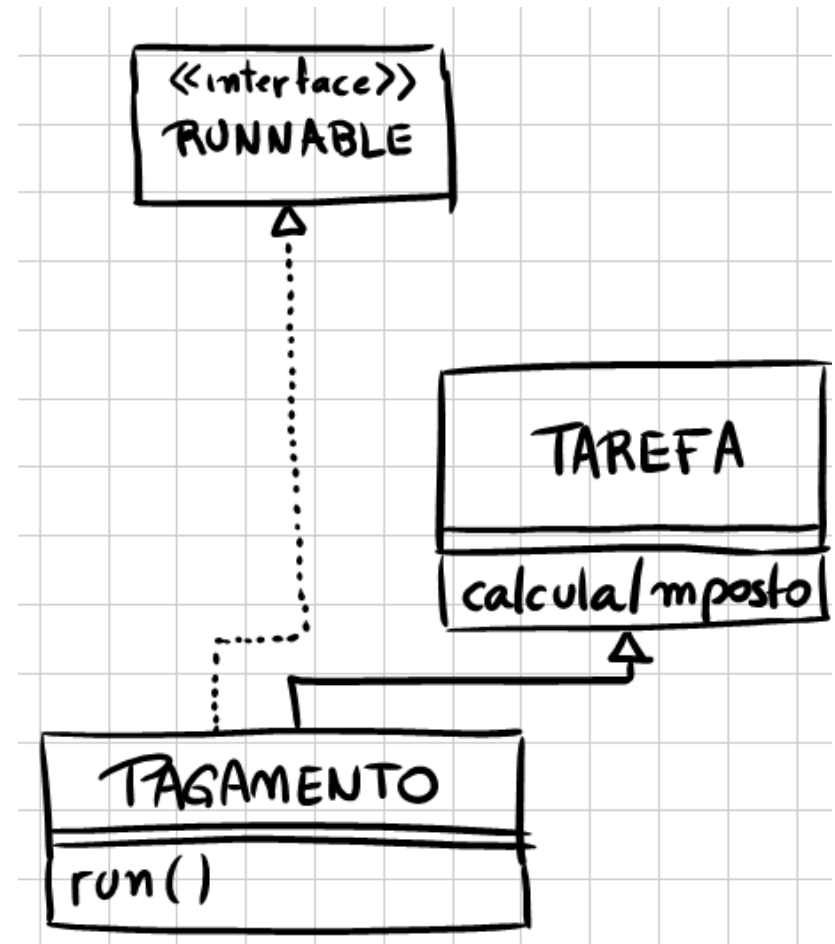
java.lang.Thread

- O problema é que dessa forma a herança é “gasta”, não sendo mais possível utilizá-la nesta classe

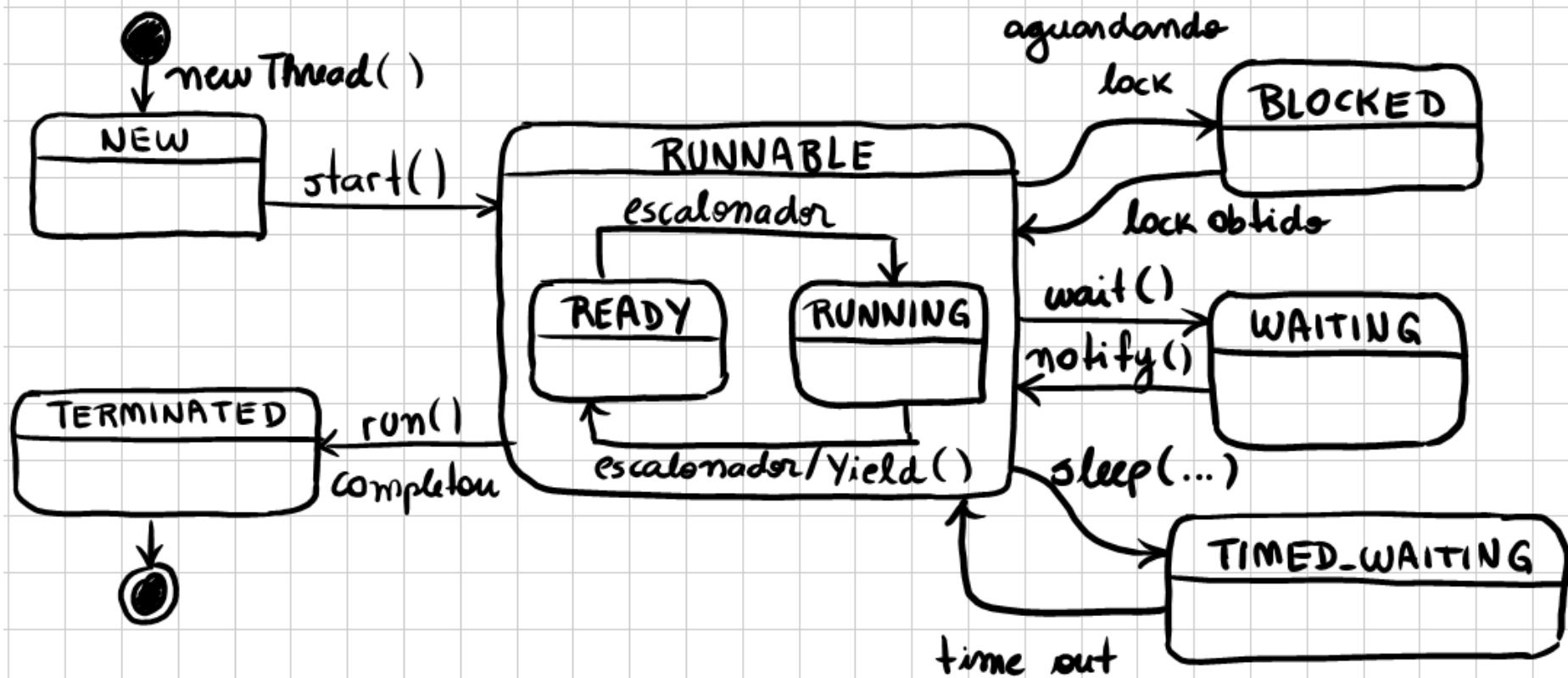


java.lang.Runnable

- Permite dar a uma classe a característica de “rodável” em paralelo
- Não “gasta” a herança, uma vez que uma classe pode implementar mais de uma interface ao mesmo tempo
- Usada em conjunto com a classe Thread



Estados

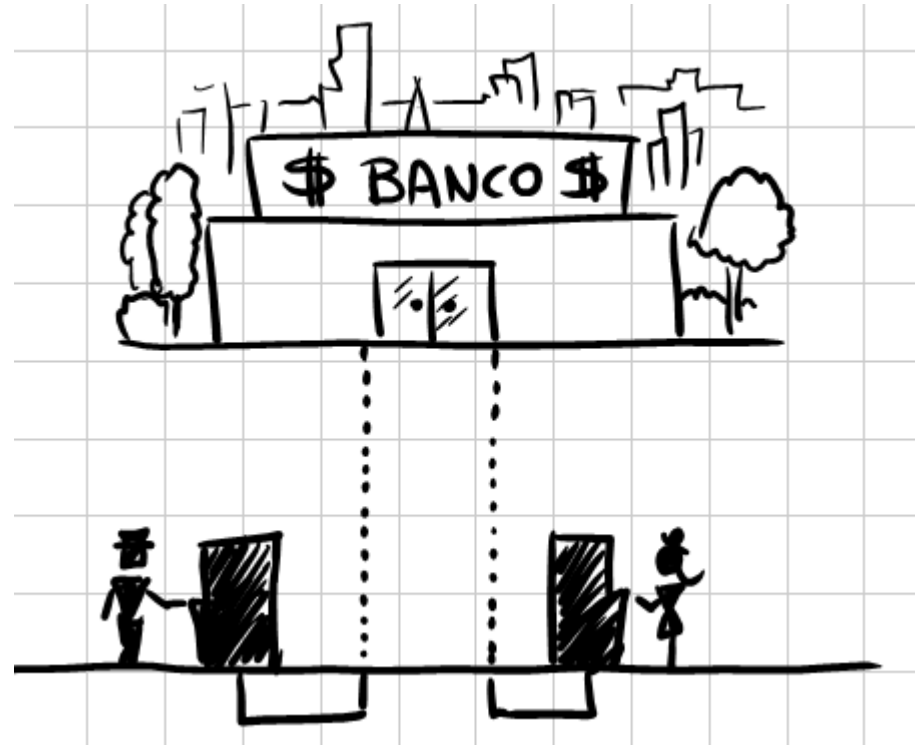


Sincronismo

- Threads compartilham recursos
- Acesso concorrente pode causar problemas
 - Objetos danificados
 - Dados inconsistentes
- Conflitos leitura/escrita
- Conflitos escrita/escrita

Exemplo

- Paulo e Lucélia, casados, possuem uma conta conjunta no banco.
- Saldo = R\$ 1000,00
- Em um belo dia, ambos chegam a diferentes caixas eletrônicos ao mesmo tempo
- Ocorre uma série inusitada de eventos de visualização de saldo e saques. Veja o que acontece:



Exemplo

10h28m13s: Paulo tira o saldo = R\$ 1000,00

10h28m28s: Lucélia tira o saldo = R\$ 1000,00

10h29m43s: Paulo solicita saque de R\$ 600,00

10h29m49s: Lucélia solicita saque de R\$ 600,00

10h30m02s: A máquina entrega o dinheiro para Paulo

10h30m03s: A máquina informa a Lucélia que não há saldo suficiente

10h30m04s: Lucélia pensa “mas acabei de olhar e vi que tinham R\$ 1000,00”

Exemplo

10h30m35s: Lucélia decide tirar um extrato para verificar

10h31m01s: O extrato mostra um saque de R\$ 600,00, naquele EXATO MOMENTO (pelo menos na concepção de um ser humano)

10h31m40s: Lucélia decide ir pessoalmente ao banco para reclamar que o dinheiro não foi entregue pela máquina

10h52m11s: O atendente confere a conta e explica a Lucélia que foram solicitados saques em terminais diferentes, Lucélia compreende, liga para o marido, e confirma o motivo da confusão

Concorrência

- Problema está nos detalhes
- Ocorrências inusitadas de eventos
- É preciso avaliar os pontos de concorrência e as possibilidades de erro

Sincronização

- Problema está no método transferir()
 - `contas[contaSaque].saldo -= valor;`
 - `contas[contaDeposito].saldo += valor;`
- O que acontece se duas threads modificarem a mesma conta EXATAMENTE entre o saque e o depósito?
 - Contas com saldo negativo
 - Algo que não deveria ocorrer

Sincronização

- É possível evitar acesso concorrente a trechos de código
 - Método transferir, por exemplo
- Modificador synchronized
 - Marca um determinado bloco
 - Estabelece uma chave de acesso (lock)
 - Toda thread deverá obter exclusividade sobre a chave de acesso
 - Apenas a thread que detiver a chave de acesso poderá executar o bloco
 - Quando uma thread encerra a execução, a chave é liberada para outras threads
- Implementa o modelo “monitor”

Sincronização

- É igual banheiro de lanchonete
- Somente quem tem a chave fica lá dentro
- Os outros esperam lá fora
- Assim que um indivíduo sai, ele devolve a chave, e um próximo pode pegar e entrar
 - Qualquer um pode ser o próximo
 - É o escalonador Java quem decide
 - Nem sempre é o primeiro da fila!!

Chave de acesso sincronizado

- Qualquer objeto Java serve

```
public void metodo() {  
    // instruções quaisquer, sem  
    // acesso concorrente  
    synchronized( objetoQualquer ) {  
        // trecho crítico  
        // com acesso concorrente  
    }  
    // instruções não críticas  
}
```

Chave de acesso sincronizado

- É possível sincronizar o método todo

```
public synchronized void metodo() {  
    // trecho crítico  
    // com acesso concorrente  
}
```

- Equivale a sincronizar todo o código do método utilizando this como chave de acesso:

```
public void metodo() {  
    synchronized(this) {  
        // código sincronizado  
    }  
}
```

Sincronização

- Para resolver o problema do banco, basta sincronizar o método transferir()

Sincronização

- Imagine outra situação:

```
class Counter {  
    private int c = 0;  
    public void increment() { c++; }  
    public void decrement() { c--; }  
    public int value() { return c; }  
}
```

- Há três métodos que acessam o mesmo recurso!

Sincronização

- No caso do banheiro, é como se houvessem três banheiros
 - Mas com um defeito no encanamento: se houverem duas descargas ao mesmo tempo ocorre um refluxo geral
 - Ou seja, mesmo existindo três banheiros, só um pode ser usado a cada momento
- Para resolver: use uma chave única para as três portas

Sincronização

- Em Java, é a mesma coisa:

```
...  
synchronized (obj1) {  
    // acesso a recurso compartilhado r1  
}  
// outras instruções  
synchronized (obj1) {  
    // acesso a recurso compartilhado r1  
}  
...
```

Sincronização

- No exemplo anterior

```
class Counter {  
    private int c = 0;  
    public synchronized void increment() { c++; }  
    public synchronized void decrement() { c--; }  
    public synchronized int value() { return c; }  
}
```


Sincronização

- Imagine que você está na mesma lanchonete dos exemplos anteriores
- Você pega a chave, entra e... está quebrado!!
- O que precisa fazer?
 - Devolver a chave
 - Esperar alguém arrumar
 - Arrumar = pegar a chave, entrar, arrumar, e devolver a chave
 - Tentar pegar a chave e entrar novamente

Sincronização

```
repetir {  
    verifica se está quebrado  
}  
até que não quebrado  
pega a chave e entra normalmente
```

Sincronização

- 2 problemas:
 - Ficar toda hora verificando se está quebrado é cansativo
 - E se, assim que acabar o conserto, quebrar logo em seguida, mas antes de eu entrar novamente?

Sincronização

- Solução:
 - Esperar sentado
 - Ser avisado do concerto concluído
 - Perguntar se está funcionando antes de tentar entrar novamente
- Modelo de “semáforo”

Sincronização

- Em Java
 - wait()
 - notify()
 - notifyAll()

Sincronização

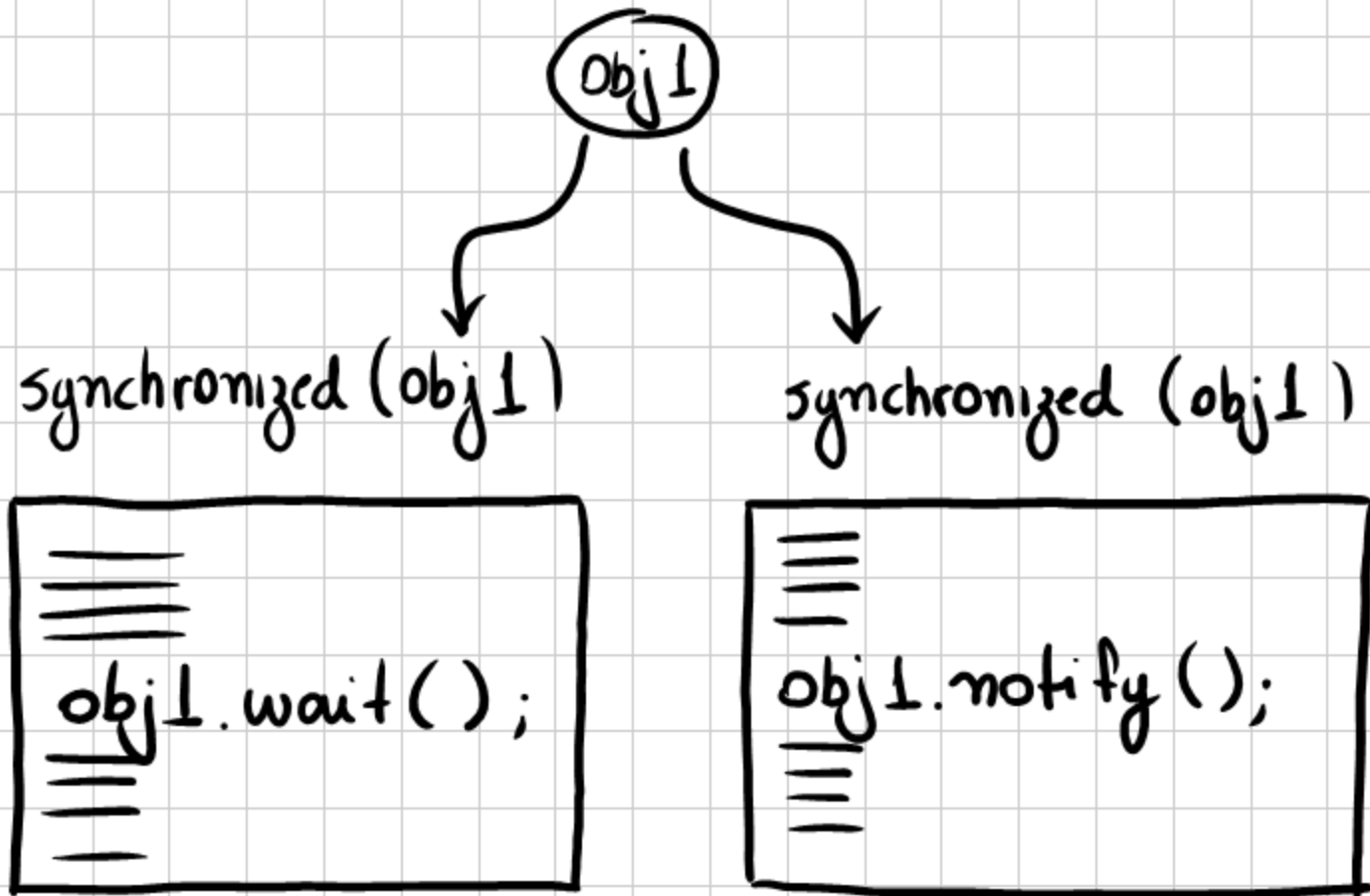
```
while(quebrado) {  
    wait() // sentado, ou seja,  
           // não vai ficar checando  
           // o valor de quebrado  
    // Só vai chegar aqui se alguém  
    // me notificar  
}
```

pega a chave e entra normalmente

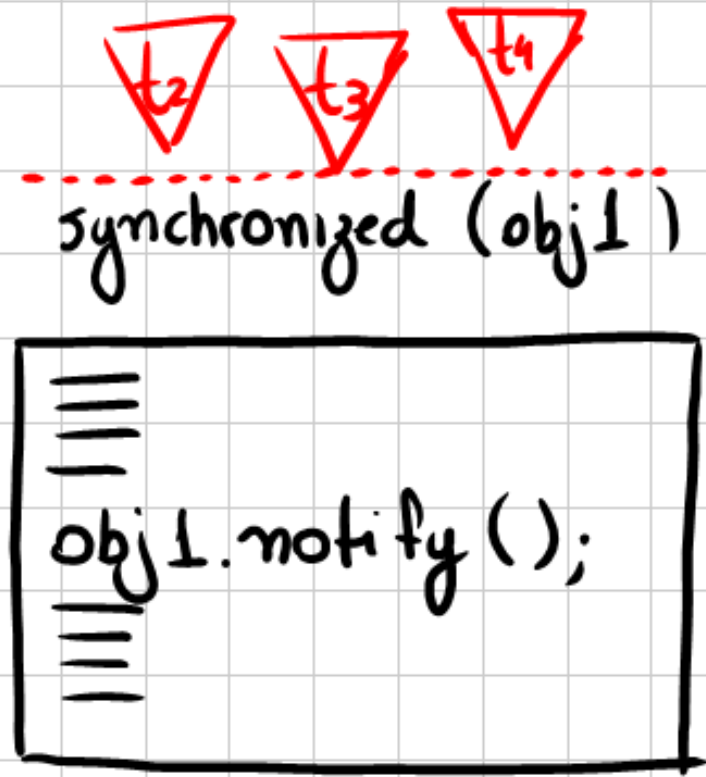
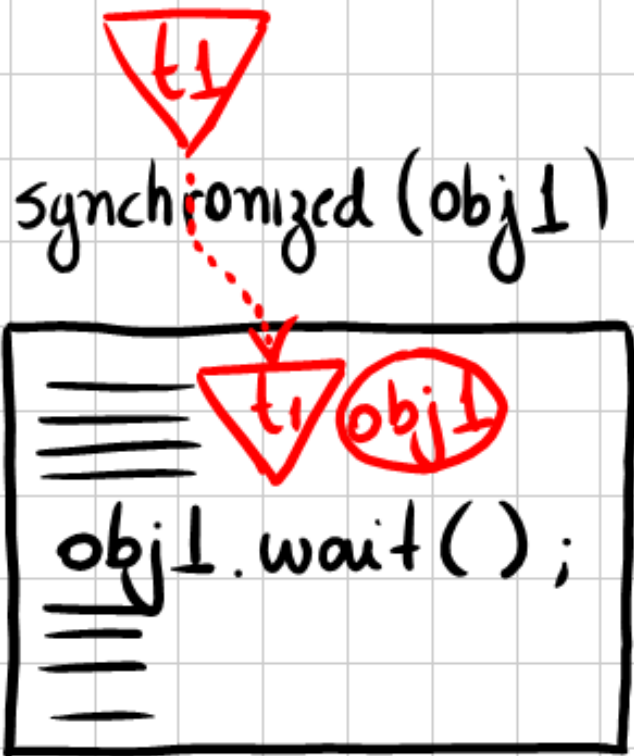
wait e notify/notifyAll

- Como funciona?
- Todo objeto Java possui uma lista para manter threads em estado de espera
 - É usada quando o objeto é uma chave de acesso
- Ao chamar o método wait(), a thread que executa a instrução entra em modo de espera
 - Devolve a chave e vai esperar sentada
- Nesse momento, outra thread pode pegar a MESMA chave
 - Quando chamar notifyAll(), as threads que estão esperando sentadas levantam e podem concorrer pela chave novamente

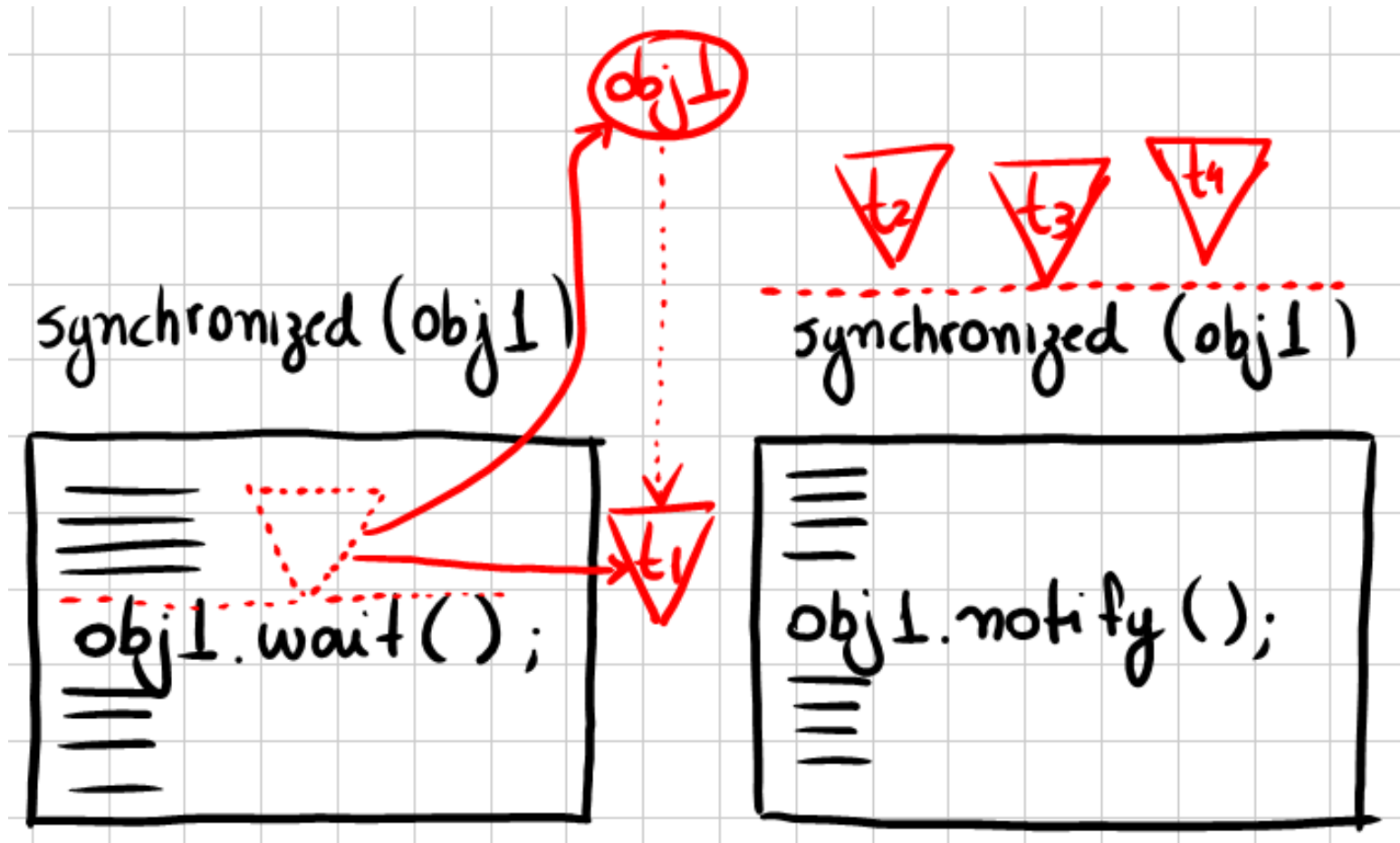
wait e notify/notifyAll



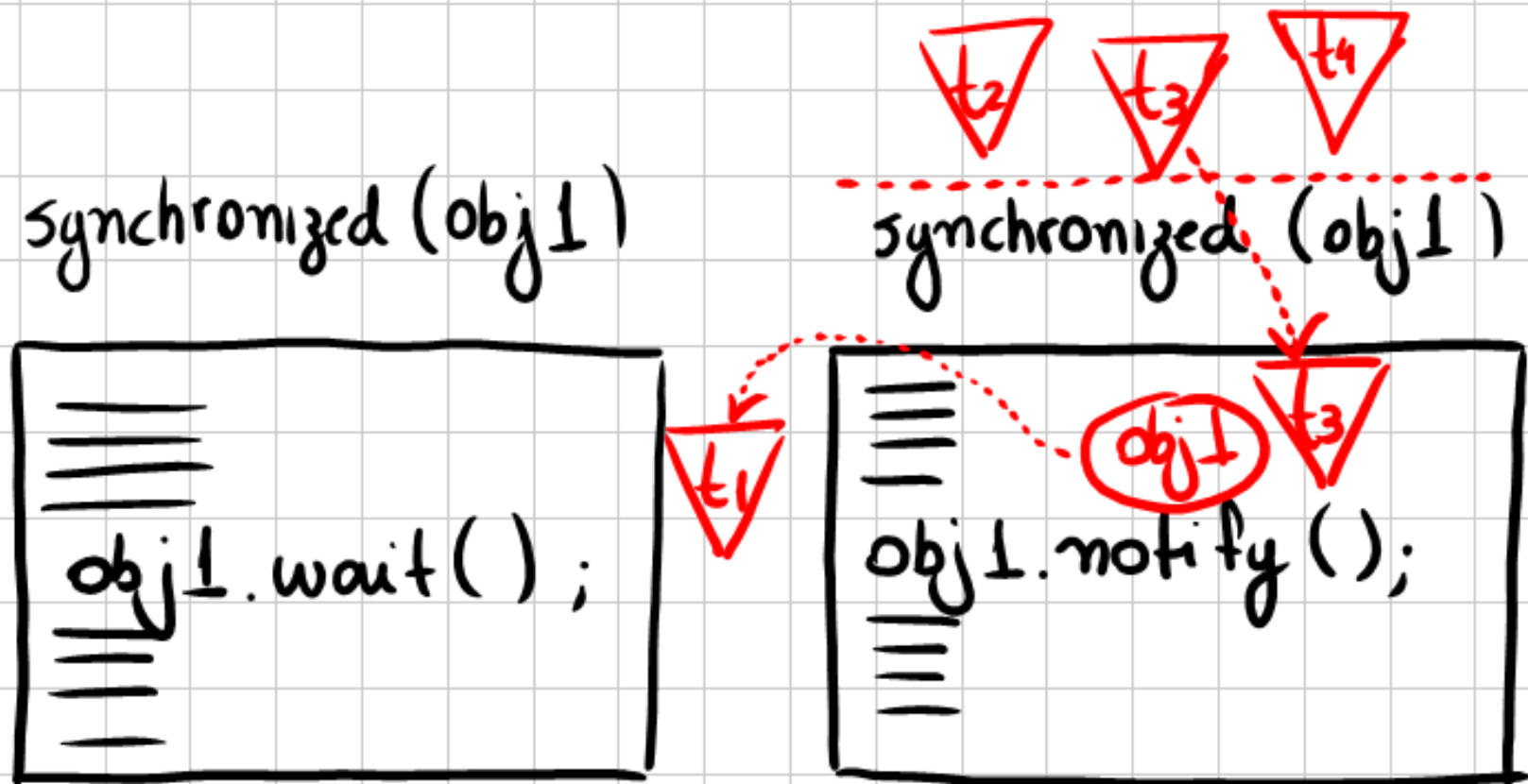
wait e notify/notifyAll



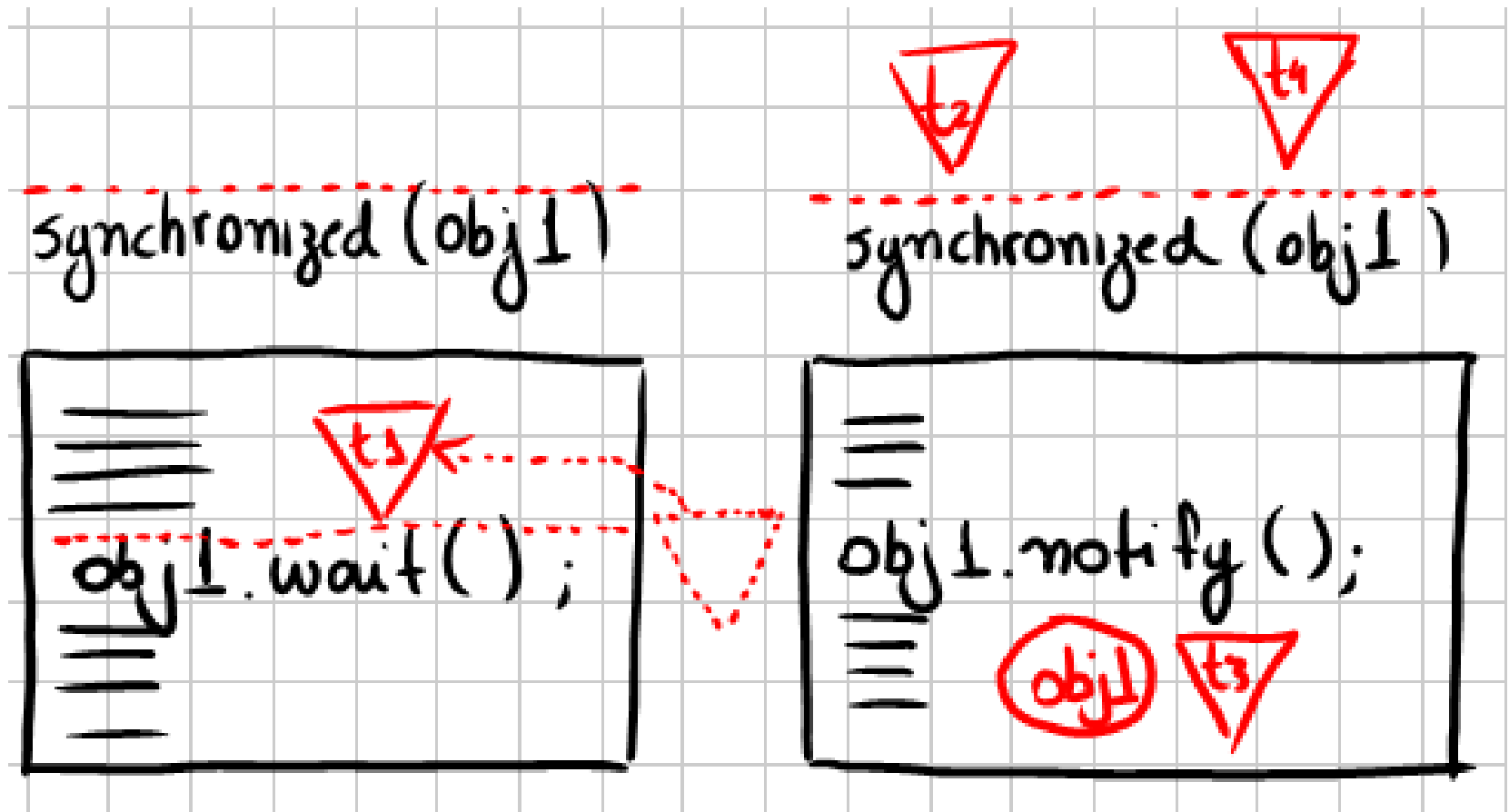
wait e notify/notifyAll



wait e notify/notifyAll

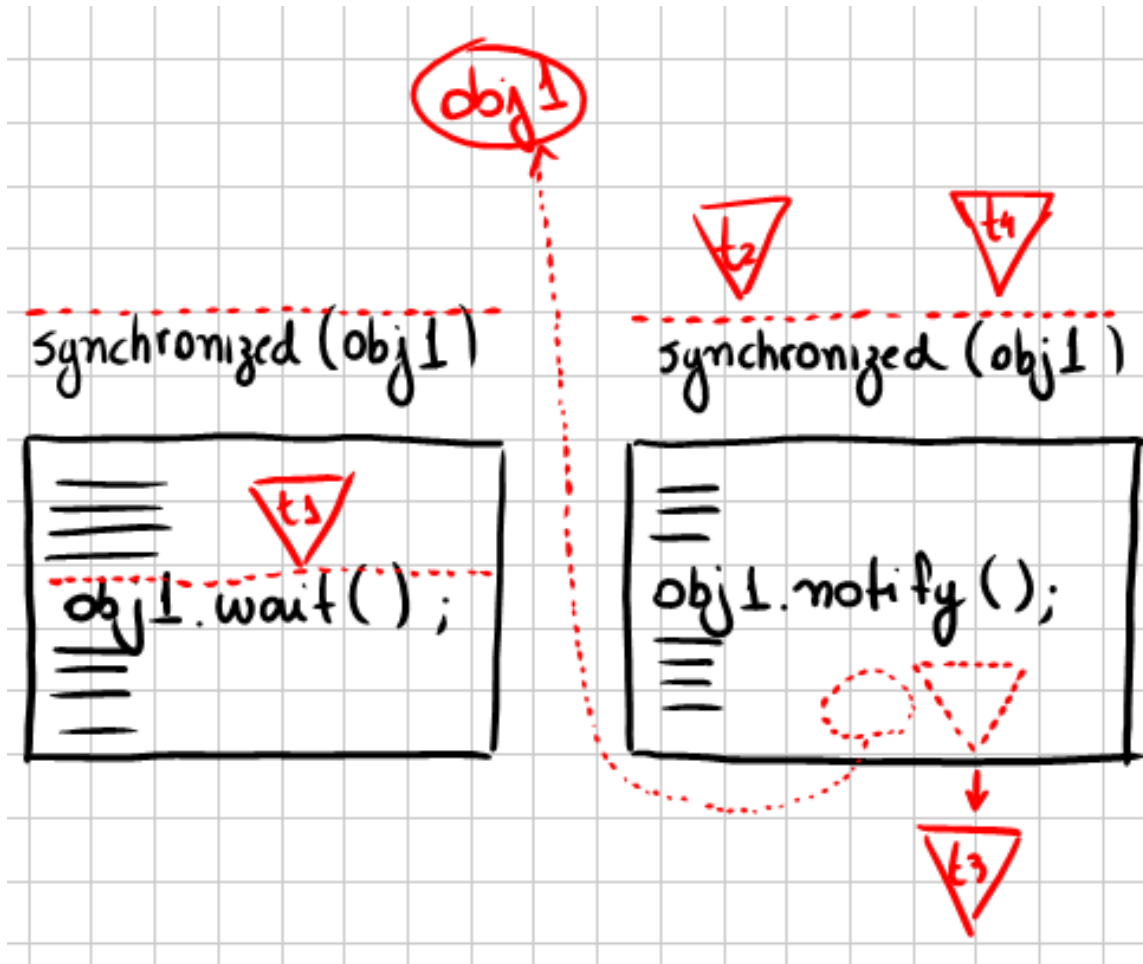


wait e notify/notifyAll



wait e notify/notifyAll

- Nesse momento, t1, t2 e t4 concorrem por obj1 em igualdade de condições



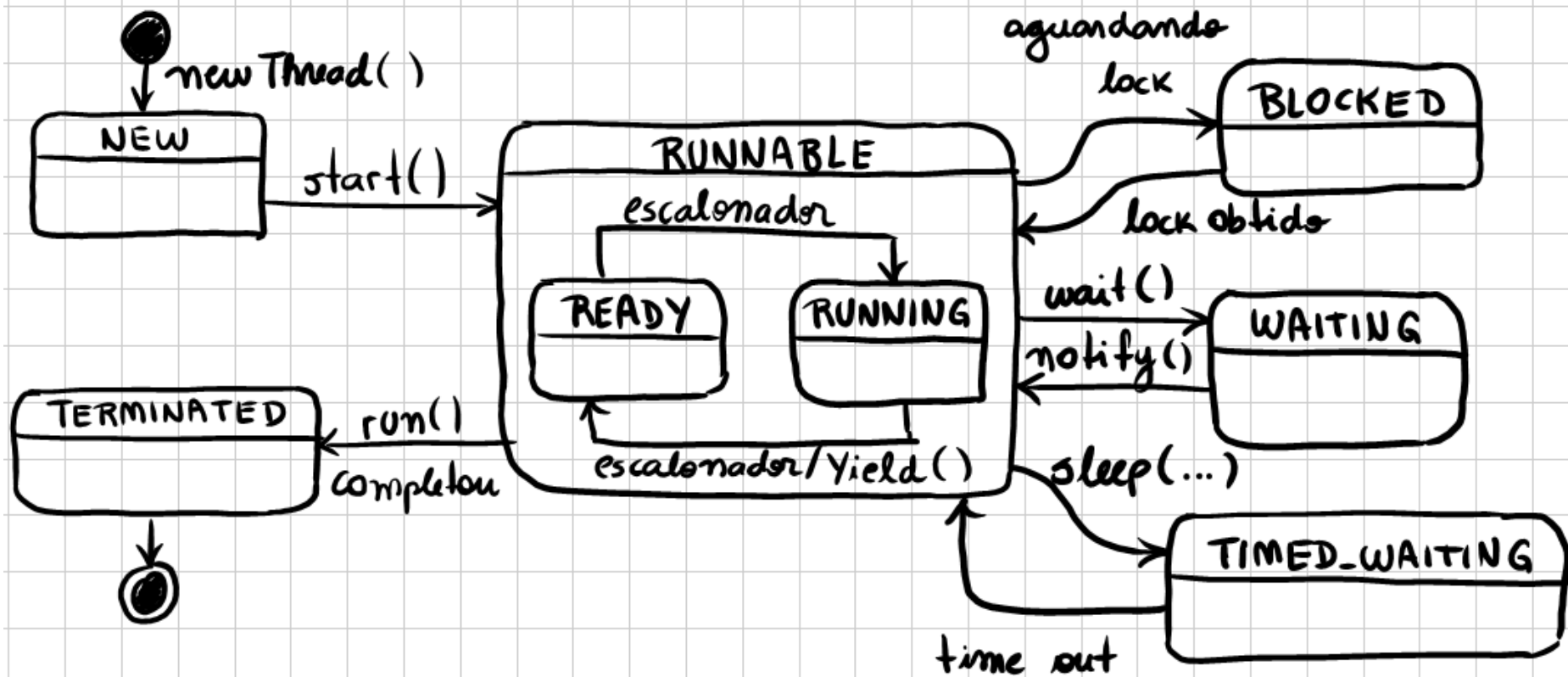
wait e notify/notifyAll

- Isso faz sentido???
- e se t2 ou t4 “pularem” a vez de t1, que estava esperando primeiro???
- Não é possível garantir quem irá executar primeiro
 - A escolha do “sortudo” fica a cargo da JVM
- Mas o importante é que agora t1 pode tentar novamente, na esperança de que t3 tenha resolvido o problema que o impedia de prosseguir

wait e notify/notifyAll

- Alguns fatos
 - Uma thread só consegue chamar `obj1.wait()` ou `obj1.notify()/notifyAll()` se ela estiver dentro de um bloco `synchronized(obj1) { ... }`
 - Senão uma exceção é lançada
 - `obj1.notify()` notifica apenas uma (qualquer uma) das threads em espera “penduradas” sob `obj1`
 - `obj1.notifyAll()` notifica todas as threads em espera “penduradas” sob `obj1`

Estados



Threads e programação Web

- Multi-threading é útil para o modelo requisição-resposta
- Imagine-se como um balconista de cartório
- Imagine se fosse possível, a cada novo cliente que pede uma cópia autenticada, dividir-se em dois, e responder a cada requisição individualmente
- É exatamente isso que é possível fazer com multithreading

Fim