



Linguagem de Programação III (LP35A)

Profª Eliana Santos

Linguagem de Programação III (LP35A)

Thread safety
Coleções threads

Resumo última aula: Controle de Threads

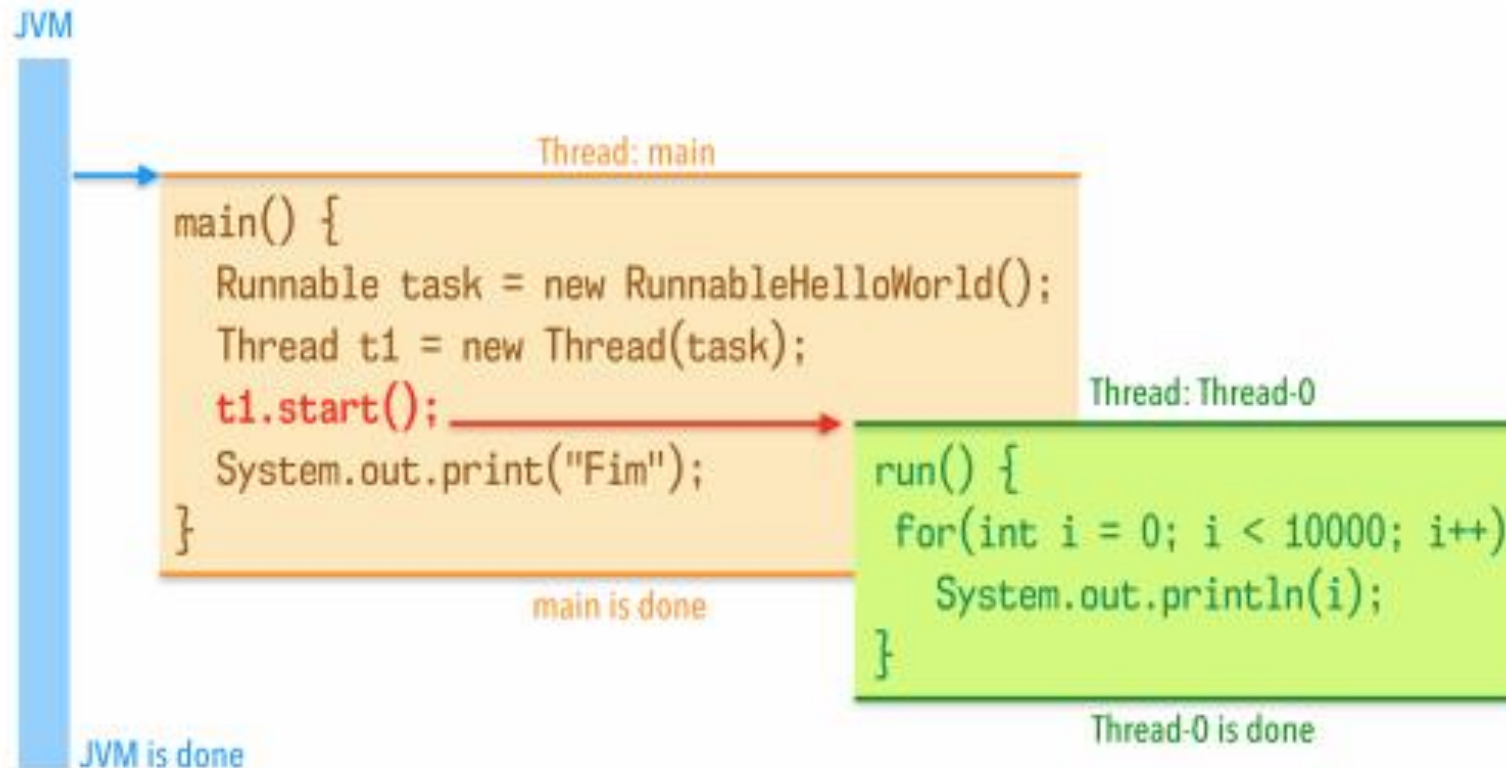
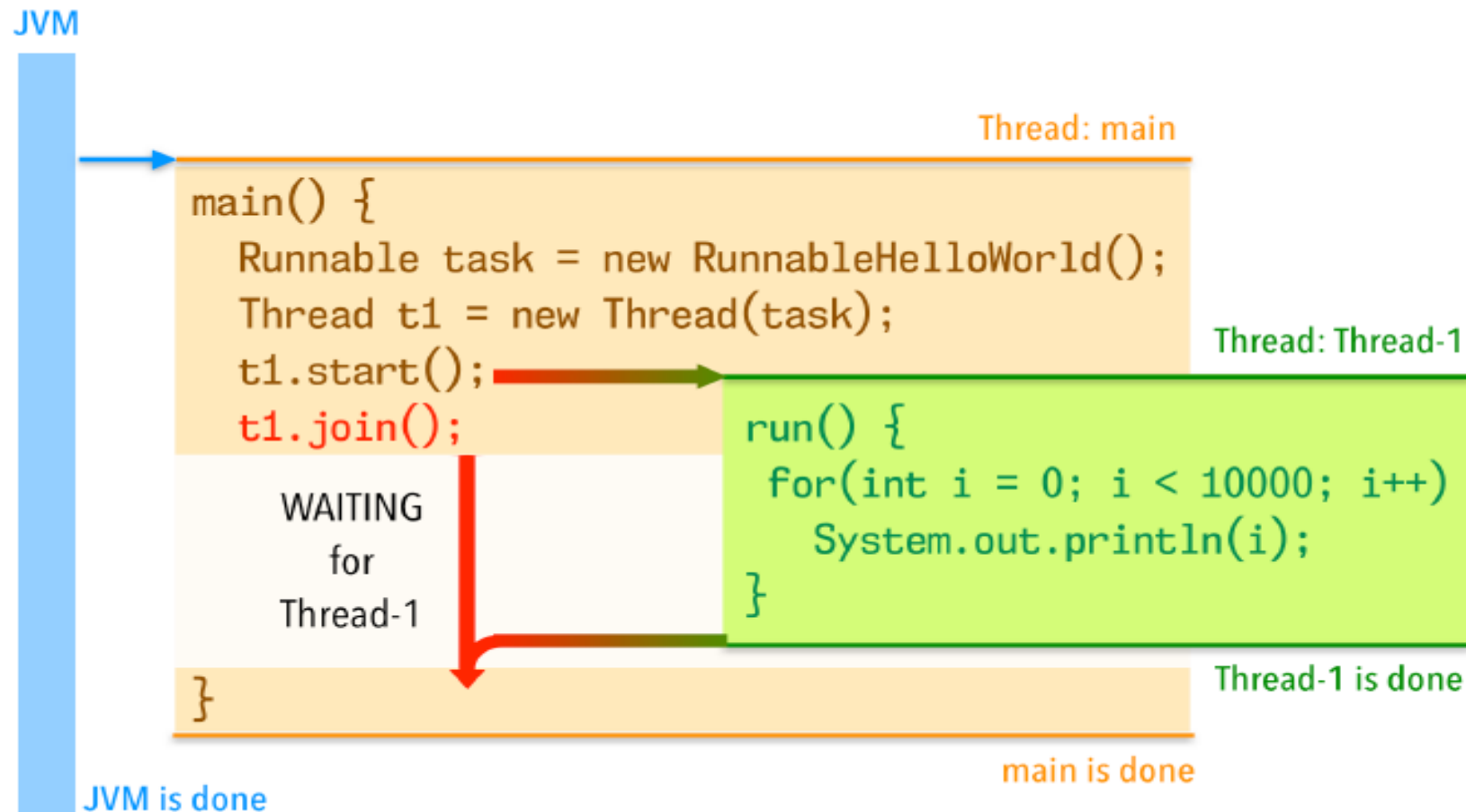


Figura 2.1 Execução de um novo thread a partir do thread "main". A palavra "Fim" será provavelmente impressa antes que o loop comece a imprimir números. O thread main provavelmente terminará antes do Thread-0.

Resumo última aula: Controle de Threads



Linguagem de Programação III (LP35A)

Thread safety

Threads safety

- Um requisito muito importante para trabalhar com Threads, é **conhecer e saber utilizar estes como Thread-Safe**, garantindo assim uma **aplicação robusta e sem inconsistências**.
- O conceito de Thread-Safe **surgiu quando há a necessidade de trabalhar-se com programação concorrente**.
- **Seu principal objetivo é garantir que 2 ou mais threads** que estejam em **“condição de corrida”** **não obtenham informações erradas**
- Condição de corrida ou **race condition** ocorre quando **várias threads desejam acessar o mesmo recurso**.
- A **solução para evitar o race condition**, quando houver necessidade, é **utilizar a exclusão mútua**, ou seja, **certificar-se que para aquele determinado recurso apenas 1 thread poderá utilizá-lo por vez**.

Thread safety

- Imagine que temos o Recurso A, e que a thread 1 e 2 desejam acessar esse recurso **de forma concorrente**.
- Garantindo a **exclusão mútua ao Recurso A**, se a Thread 1 começar o acesso ao Recurso A, **ela só perderá o acesso ao mesmo quando terminar toda a tarefa** que ela deseja,

Enquanto isso, a Thread 2 fica esperando...

- Os conceitos “race condition” e “exclusão mútua” são essenciais **para entender o conceito de Thread-Safe**

Thread safety

Exemplo prático:

- Um Sistema possui dois procedimentos concorrentes (procedimento A e procedimento B),
- Ambos fazem uso de um Arquivo chamado “registroDeVariavel.txt” e sempre antes de usá-lo ele é limpo pelo procedimento que vai escrever nele o que desejar.
- Esse arquivo **serve para escrita e leitura de variáveis por apenas um procedimento**, por isso sempre que um novo procedimento vai utilizá-lo, ele deve ser limpo para evitar inconsistências.

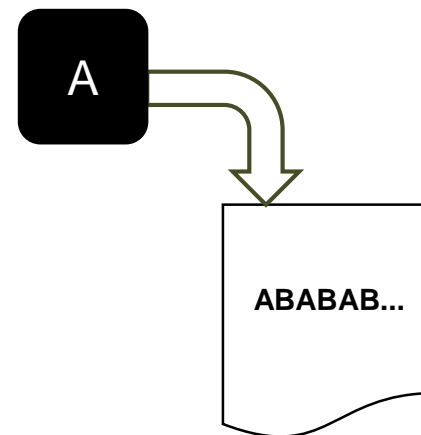


registroDeVariavel.txt

Thread safety

Exemplo prático:

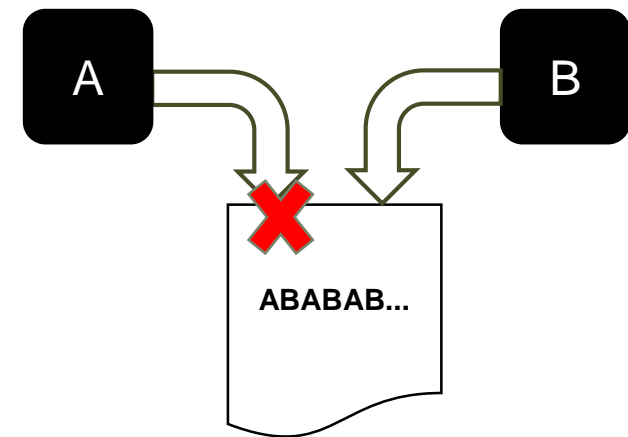
- O procedimento A acessa o arquivo e começa a realizar uma escrita...



Thread safety

Exemplo prático:

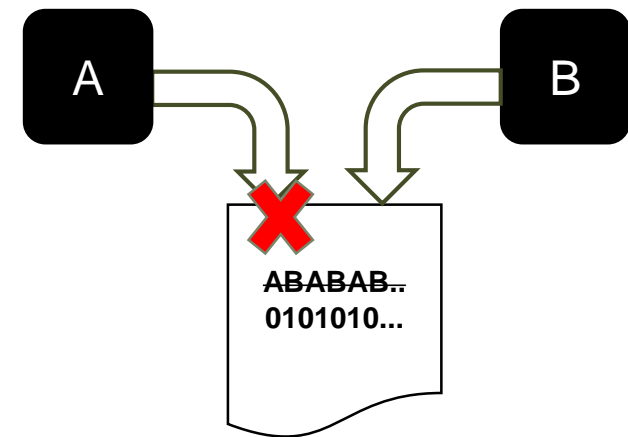
- porém no meio do processo o escalonador do processador interrompe sua execução e dá prioridade ao procedimento B.



Thread safety

Exemplo prático:

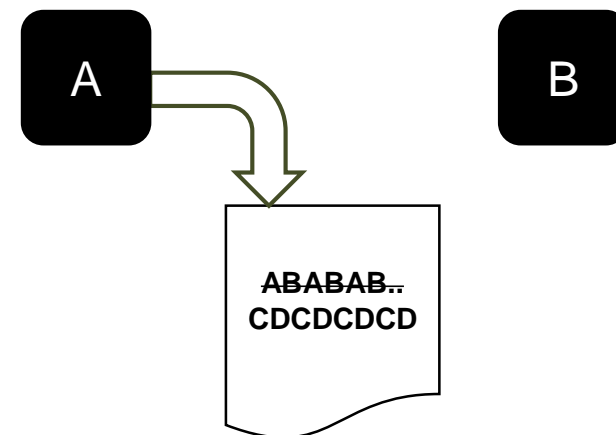
- Neste momento o procedimento B limpa o arquivo e começa a escrever o que ele precisa e termina toda sua execução.



Thread safety

Exemplo prático:

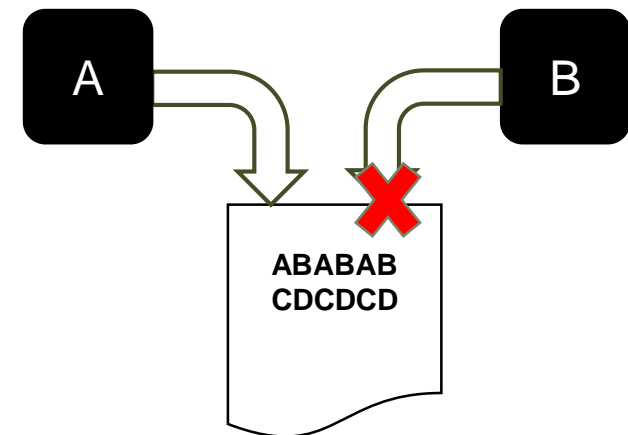
- Após o término da escrita no arquivo pelo procedimento B, o procedimento A volta a ser executado e continua sua escrita normalmente.
- Perceba aqui uma **coisa super importante**: **O procedimento B apagou tudo que o A fez**, então **quando o A continuar a escrita o arquivo apresentará inconsistências**



Thread safety

Exemplo prático:

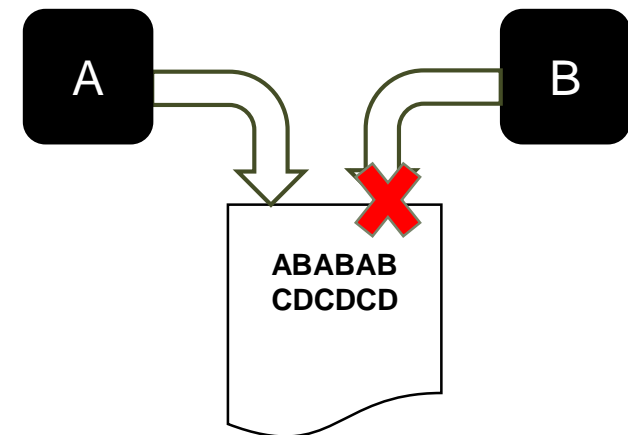
- Qual a solução para isso ? Usar Thread-Safe.
- Em Java podemos garantir a exclusão mútua através da palavra reservada “synchronized”, assim o procedimento B não poderá ser executado até o término do procedimento A.



Thread safety

Exemplo prático:

- Qual a solução para isso ? Usar Thread-Safe.
- Em Java podemos garantir a exclusão mútua através da palavra reservada “synchronized”, assim o procedimento B não poderá ser executado até o término do procedimento A.



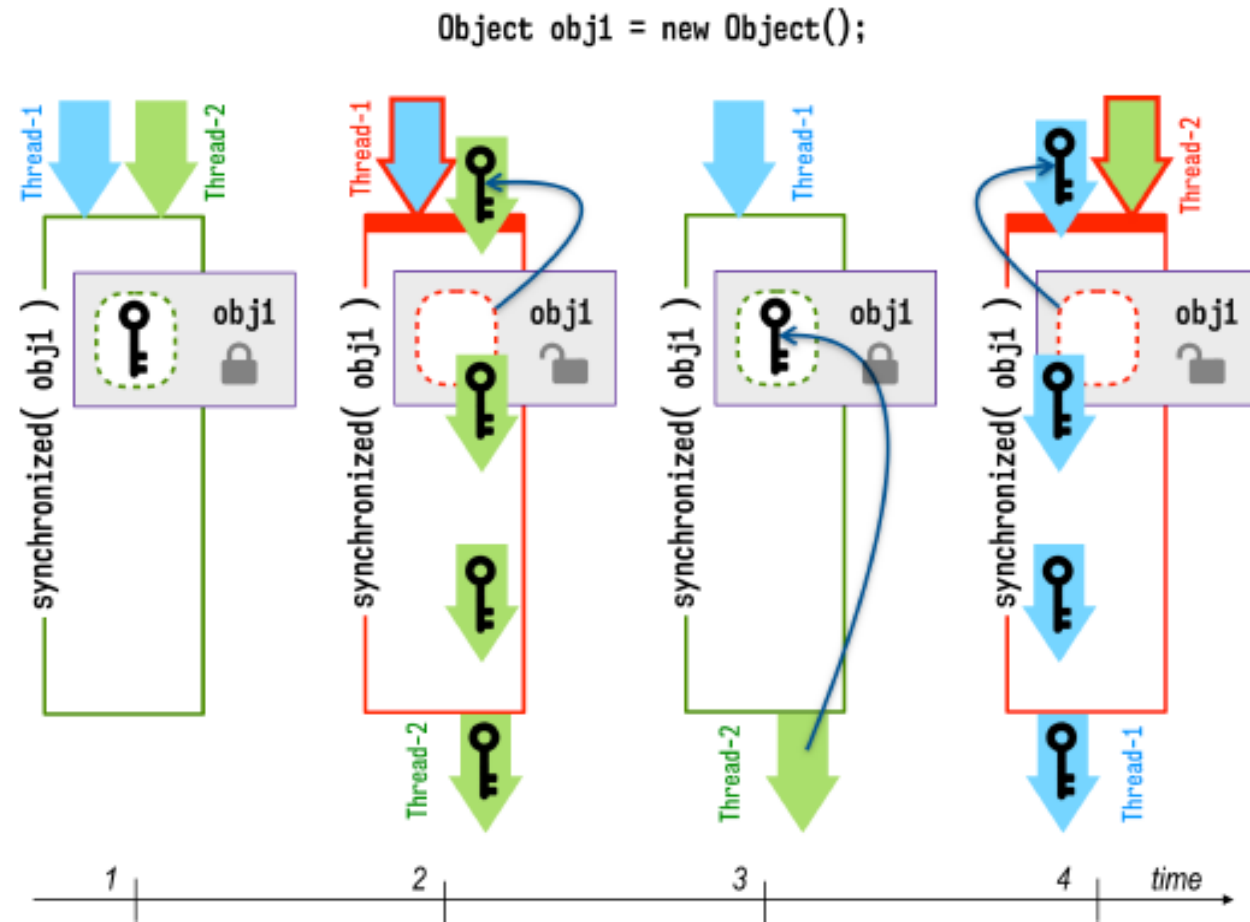
Thread safety

Em Java a exclusão mútua pode ser obtida usando um bloco **synchronized**: um mecanismo que obtém uma trava exclusiva para um objeto, impedindo que ele seja acessível por outros threads enquanto um thread estiver executando o conteúdo do bloco. O bloco `synchronized` recebe como argumento a referência do objeto:

```
Object obj = new Object() {  
    synchronized( obj ) {  
        // apenas um thread poderá entrar aqui de cada vez  
    }  
}
```

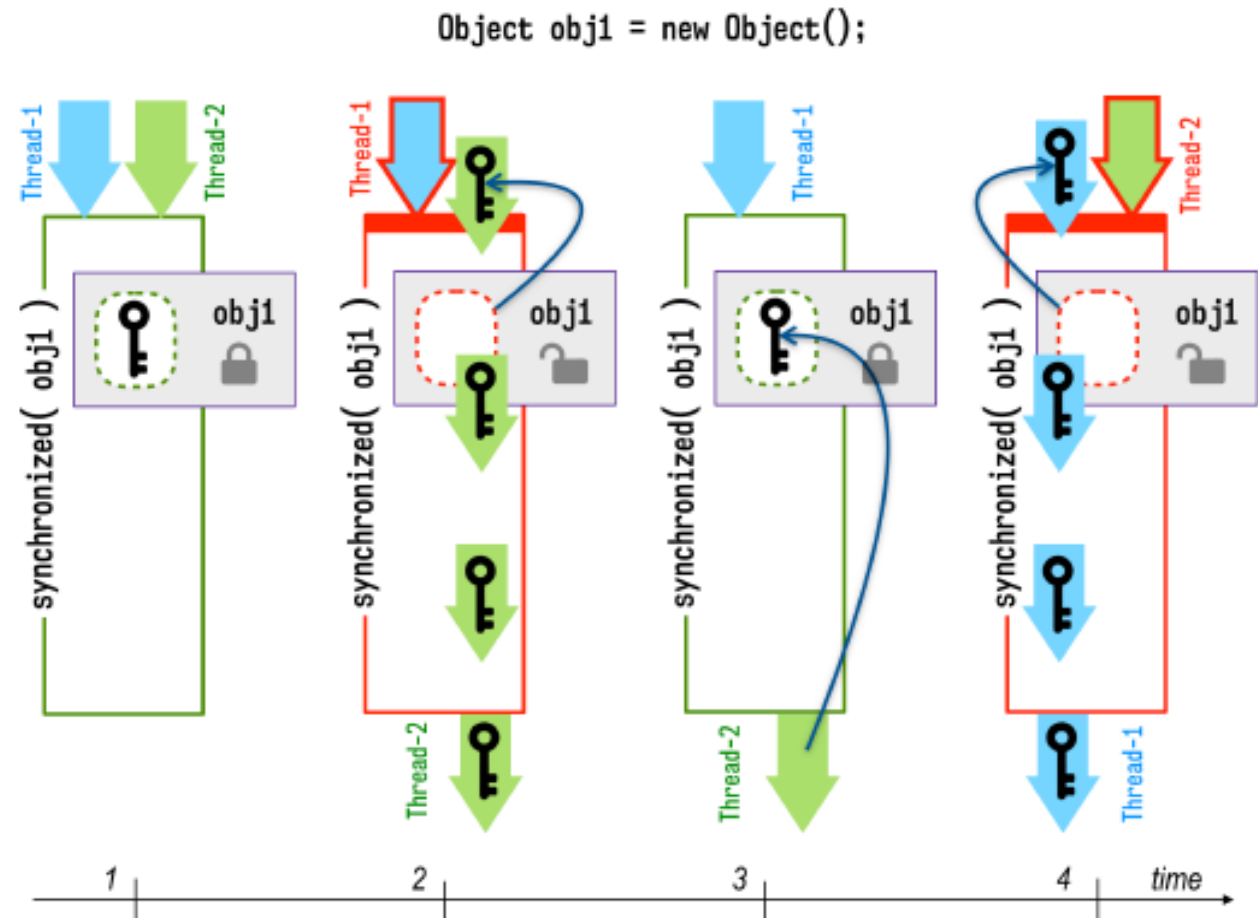
Thread safety

A Figura 4.1 mostra dois threads tentando obter acesso a um bloco synchronized de uma mesma instância. Os dois chegam no bloco ao mesmo tempo, mas o Thread-2 consegue obter a trava do objeto primeiro. A trava está representada por uma chave (com o qual poderá abrir o bloco synchronized e entrar.) Quando o thread sai do bloco, a trava é devolvida para o objeto, e o Thread-1, que estava esperando, agora pode obtê-la e ganhar acesso ao bloco. Se o Thread-2 tentar entrar novamente, ele será bloqueado e terá que esperar que o Thread-1 termine e libere a trava.



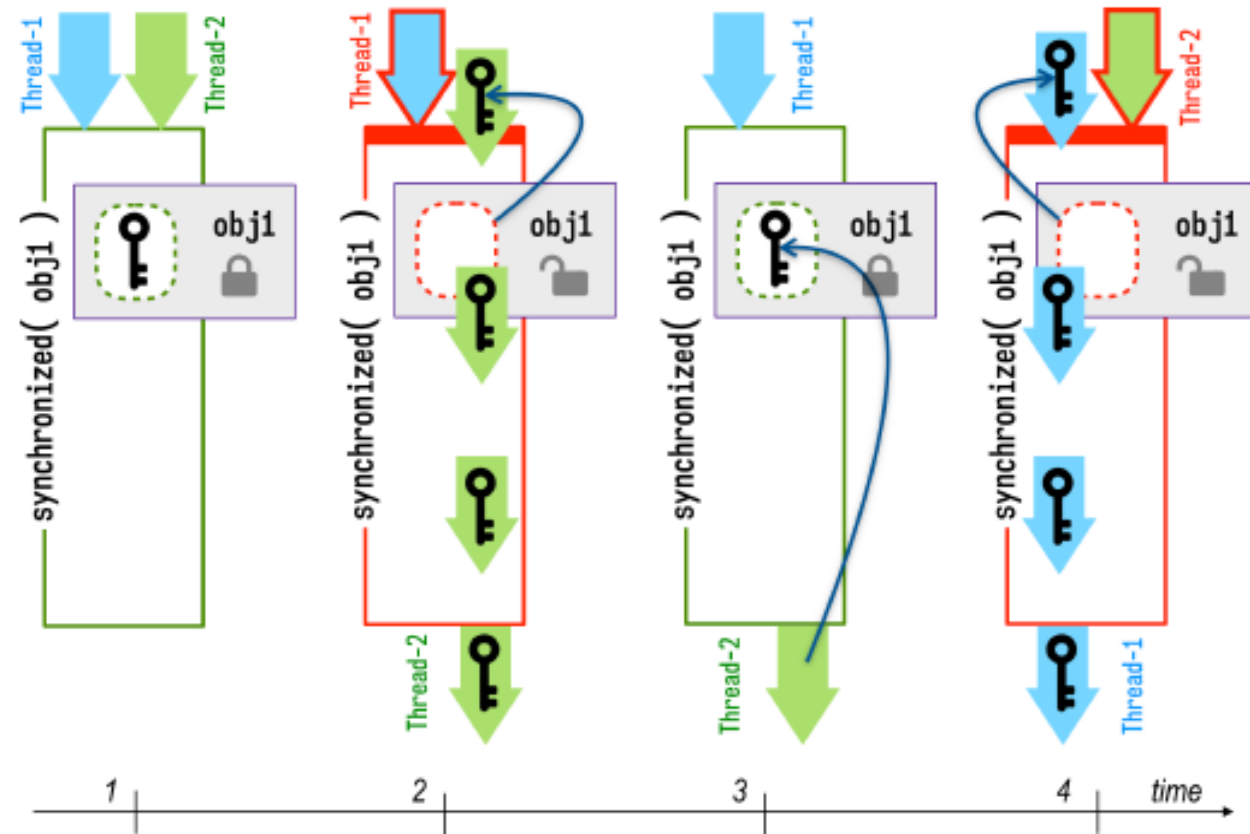
Thread safety

- **Todo objeto Java possui uma única trava.**
- Para entrar em um bloco ou método **synchronized**, **o thread precisa dessa trava.**
- Enquanto o thread estiver executando instruções do bloco, ele retém a trava do objeto e *nenhum* outro thread consegue acessar outro bloco ou método **synchronized** desse objeto,
- Ainda será possível acessar partes do objeto que não estejam dentro de um bloco **synchronized**, **com ou sem a trava do objeto.**



Thread safety

Quando uma trava é liberada, havendo mais de um thread esperando, não há como saber qual ganhará acesso à CPU, pois a ordem é indefinida e depende de regras de agendamento de threads que são dependentes de plataforma. Travas que respeitam ordem de chegada são chamadas de *travas justas* (*fair locks*). Essa característica (*fairness*) não é suportada pelos blocos `synchronized`, mas é implementada nos sincronizadores de `java.util.concurrent`, como `ReentrantLock` e `Semaphore`.



Thread safety

Operação de incremento variável++ : 1) ler valor anterior, 2) adicionar 1 e 3) gravar novo valor

- Como operações executadas em threads diferentes podem ser intercaladas, **é possível que dois threads leiam o mesmo valor e somem um a ele**, fazendo com que o mesmo número seja retornado em cada thread.
- Para que o resultado **funcione como esperado**, é necessário que **a operação seja realizada dentro de um bloco ou método** synchronized.

Thread safety

Operação de incremento variável++ : 1) ler valor anterior, 2) adicionar 1 e 3) gravar novo valor

- Por exemplo, é possível prever o resultado de uma sequência de operações de `append()` em um *`StringBuilder()` – (pacote **java.lang**), mas essa ordem não será garantida se o objeto for simultaneamente modificado por threads diferentes.
- Pode ser que, por pura sorte, o problema nunca apareça, e as concatenações sempre produzam um resultado correto.
- Essa impossibilidade de prever o resultado da operação é chamada de condição de corrida (race condition) e pode ser evitada através do uso de uma trava de acesso exclusivo como o bloco `synchronized`.

*Essa classe permite criar e manipular dados de Strings dinamicamente, ou seja, podem criar variáveis de String modificáveis.

Thread safety

- Condições de corrida nem sempre causam falhas. É necessário também que aconteça um timing desfavorável. Para garantir thread-safety operações do tipo *check-then-act* (ex: *lazy initialization) e *read-modify-write* (ex: incremento) devem ser sempre atômicas (commit; rollback). Observe os exemplos abaixo:


```
public class Repository {  
    private StringBuilder buffer = new StringBuilder();  
    private String prefix;  
    ...  
    public void writeData(String text) {  
        synchronized(buffer) { // a trava é obtida para o objeto buffer  
            buffer.append(this.prefix);  
            buffer.append(text);  
            buffer.append(this.suffix);  
        }  
    }  
}
```

```
public class Repository {  
    private StringBuilder buffer = new StringBuilder(); ...  
    public void writeData(String text) {  
        synchronized(this) { // a trava é obtida para um objeto Repositorio  
            buffer.append(this.prefix);  
            buffer.append(text);  
            buffer.append(this.suffix);  
        }  
    }  
}
```

* A inicialização lenta é a tática de atrasar a criação de um objeto, o cálculo de um valor ou algum outro processo caro até a primeira vez que for necessário


Thread safety

```
public class Repository {  
    private StringBuilder buffer = new StringBuilder();  
    private String prefix;  
    ...  
    public void writeData(String text) {  
        synchronized(buffer) { // a trava é obtida para o objeto buffer  
            buffer.append(this.prefix);  
            buffer.append(text);  
            buffer.append(this.suffix);  
        }  
    }  
}
```



A classe acima protege apenas StringBuilder mas não protege outras variáveis de Repository que poderiam ser alteradas em outros métodos.

```
public class Repository {  
    private StringBuilder buffer = new StringBuilder(); ...  
    public void writeData(String text) {  
        synchronized(this) { // a trava é obtida para um objeto Repositorio  
            buffer.append(this.prefix);  
            buffer.append(text);  
            buffer.append(this.suffix);  
        }  
    }  
}
```



Usando a trava do próprio objeto Repository no bloco synchronized garante acesso exclusivo ao *objeto inteiro* enquanto o bloco estiver sendo executado. Neste caso, o argumento é **this**.

Thread safety

Se o bloco envolve todo o conteúdo do método, podemos usar o *modificador* `synchronized` para **obter o mesmo efeito**.

- Na classe `Repository` **ambos os métodos** `writeData()` e `readData()` são `synchronized`.
- **Se um thread estiver acessando** o método `writeData()` em um objeto, **outro thread não poderá acessar** nem `writeData()`, nem `readData()`.

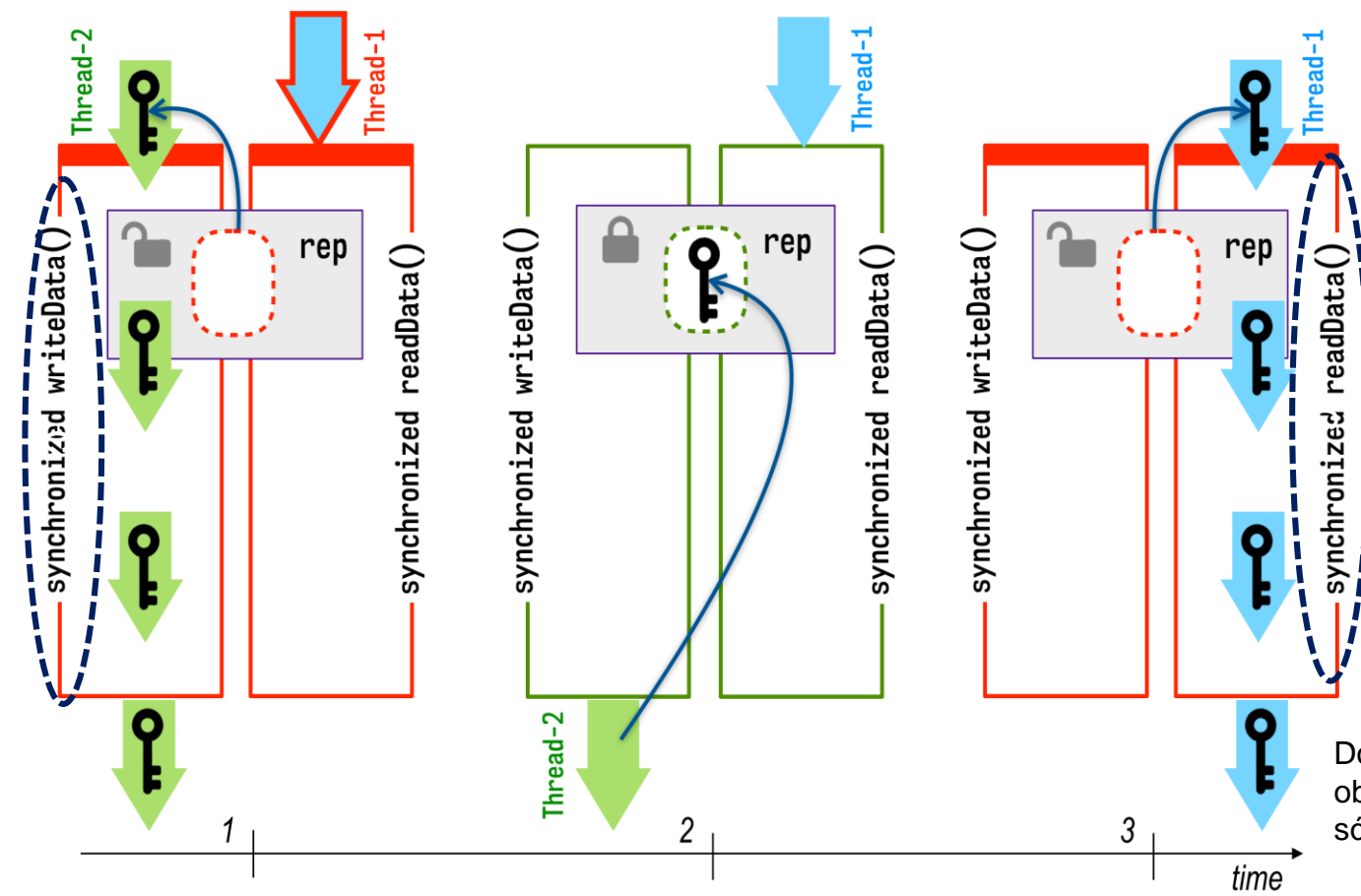
```
public class Repository {  
    private StringBuilder buffer = new StringBuilder();  
    ...  
    public synchronized void writeData(String prefix, String text, String suffix) {  
        buffer.append(prefix);  
        buffer.append(text);  
        buffer.append(suffix);  
    }  
    public synchronized String readData() {  
        return buffer.toString();  
    }  
}
```


Thread safety

Na classe Repository ambos os métodos `writeData()` e `readData()` são `synchronized`. Se um thread estiver acessando o método `writeData()` em um objeto, outro thread não poderá acessar nem `writeData()`, nem `readData()`.

Each thread has exclusive access to the object

`Repository rep = new Repository();`
`Thread-2: rep.writeData("A"); Thread-1: rep.readData();`



```
public class Repository {  
    private StringBuilder buffer = new StringBuilder();  
  
    ...  
    public synchronized void writeData(String prefix, String text, String suffix) {  
        buffer.append(prefix);  
        buffer.append(text);  
        buffer.append(suffix);  
    }  
    public synchronized String readData() {  
        return buffer.toString();  
    }  
}
```

Dois threads tentando acessar métodos sincronizados diferentes de um mesmo objeto. O Thread-2 adquiriu a trava do objeto primeiro, bloqueando o Thread-1 que só terá acesso depois que o Thread-2 deixar o método e liberar a trava do objeto.

Thread safety

IMPORTANTE!!

- **Todo objeto Java possui *uma única trava*.**

Para entrar em um bloco ou método synchronized, o thread precisa dessa trava.

- Enquanto o thread estiver executando instruções do bloco, ele retém a trava do objeto e *nenhum* outro thread consegue acessar outro bloco ou método synchronized desse objeto,
- **Mas ainda será possível acessar partes do objeto que não estejam dentro de um bloco synchronized, com ou sem a trava do objeto.**

Thread safety

Na Figura 4.3, dois threads acessam o mesmo objeto ao mesmo tempo, apesar de um dos threads ter obtido a trava do objeto e ela não estar disponível para o outro. Threads sempre podem acessar objetos sem precisar de suas travas se estiverem fora de blocos ou métodos synchronized.

Exemplo:

Observe a forma de acesso das Threads

LEMBRE-SE:

- Uma trava está sempre associada a uma *instância*.
- Se uma mesma classe é implementada por mais de um objeto, threads diferentes podem acessar um bloco synchronized ao mesmo tempo **quando o código executa no contexto de objetos diferentes.**

```
Repository rep = new Repository();  
Thread-2: rep.writeData("A");  
Thread-1: rep.readData();
```

Both threads have non-exclusive access to the object (they can both use it at the same time)

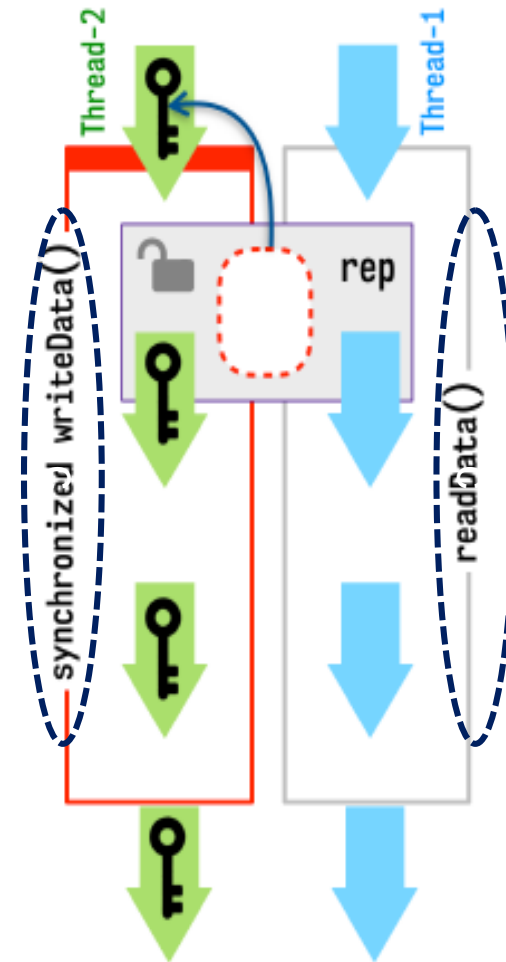


Figura 4.3 Thread-2 acessa obtém a trava de um objeto ao acessar um método sincronizado. Apesar da trava do objeto não estar disponível, Thread-1 também consegue acesso a ele através de método que não foi marcado como synchronized.

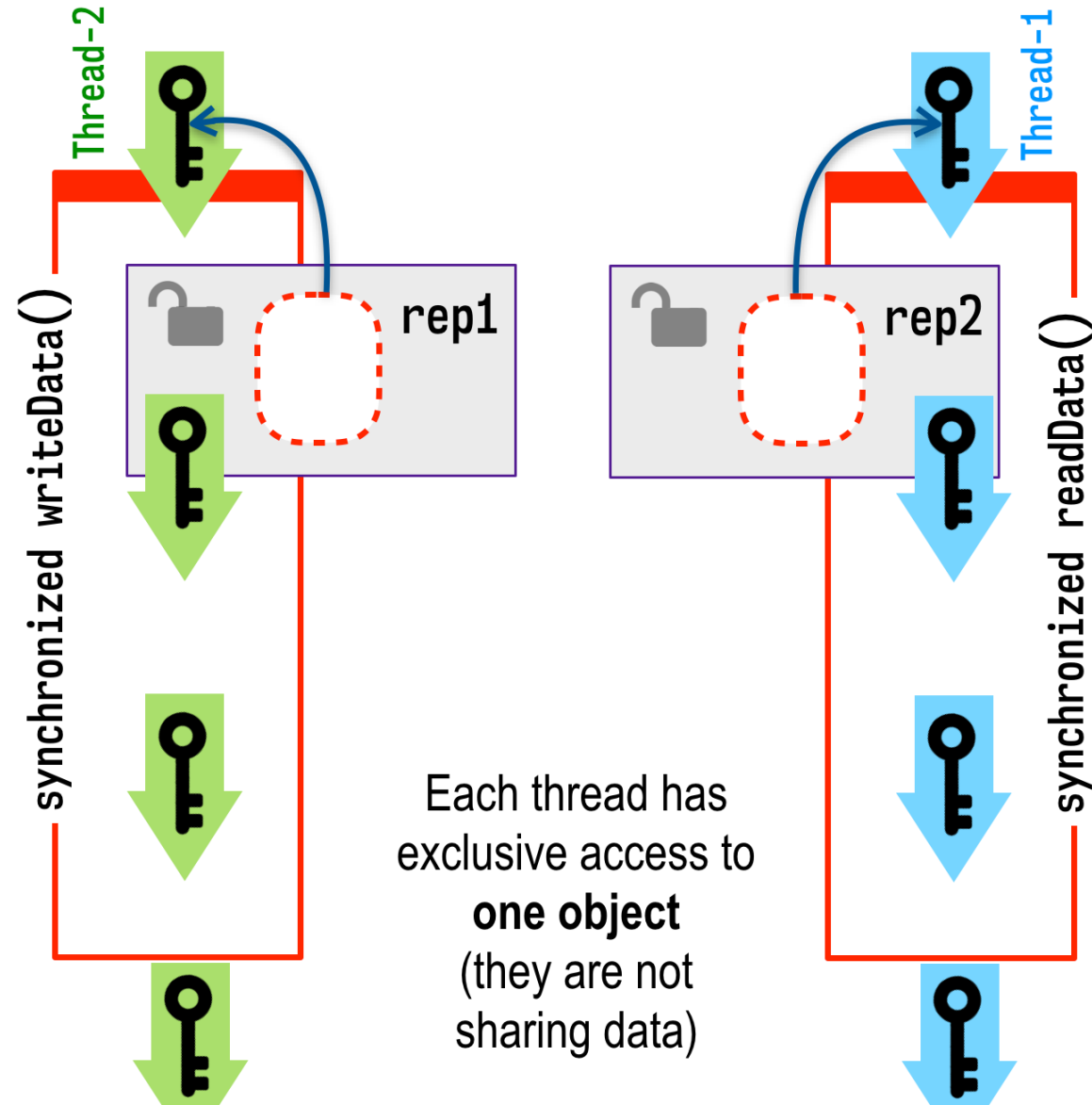
Thread safety

```
Repository rep1 = new Repository();  
Thread-2: rep1.writeData("A");
```

```
Repository rep2 = new Repository();  
Thread-1: rep2.readData();
```

Dois threads acessam métodos sincronizados de instâncias da mesma classe ao mesmo tempo.

Ambos podem executar ao mesmo tempo pois as instâncias são diferentes, e cada uma possui uma trava distinta.



Thread safety

IMPORTANTE!!

- **Sempre** que um objeto compartilhar dados mutáveis entre threads diferentes, **é fundamental** que o acesso e a gravação desses dados ocorra em um bloco ou método synchronized.
- Fique atento aos problemas causados por excesso de sincronização. Um impasse (deadlock) é um problema que pode ocorrer quando métodos/blocos synchronized chamam outros métodos/blocos synchronized (travas reentrantes).

No impasse, threads esperam que uma condição mude para liberar uma trava, mas os threads que poderiam mudar essa condição não conseguem acesso por não possuírem a trava.

- É possível evitar impasses evitando travas reentrantes para fora do objeto (**nunca chame métodos de objetos externos em blocos synchronized**) ou controlando a ordem de acesso/liberação dessas travas.

Mantenha sempre o mínimo de código necessário dentro de um bloco synchronized.

- Para que a sincronização funcione de forma esperada, também é importante que referências estejam confinadas ao objeto, evitando que *vazem* para fora do objeto e possam ser alteradas sem sincronização. Isto é possível se os métodos nunca retornarem referências, mas cópias dos dados.

Thread safety

- O método estático `holdsLock(Object)` retorna `true` se o thread corrente possui a trava para um objeto.
- Este método é útil para escrever asserções e testes unitários envolvendo threads.
- Pode ser usado por um método para saber se foi chamado dentro de um contexto `synchronized` de um determinado objeto.

Thread safety

- Nesta classe, dentro do método `synchronized`, `Thread.holdsLock()` sempre será `true` para a referência `this` (e pode ou não ser `true` para outra referência).

```
class SharedResource {  
    public synchronized void synchronizedMethod() {  
        System.out.println("synchronizedMethod(): " + Thread.holdsLock(this));  
    }  
    public void method() {  
        System.out.println("method(): " + Thread.holdsLock(this));  
    }  
}
```

```
SharedResource obj = new SharedResource();  
synchronized (obj) {  
    obj.method(); // holdsLock = true  
}  
obj.method(); // holdsLock = false
```



Nas duas chamadas ao método comum `method()`, na primeira `holdsLock()` retornará `true`, e na segunda `false`:

Linguagem de Programação III (LP35A)

Variáveis voláteis

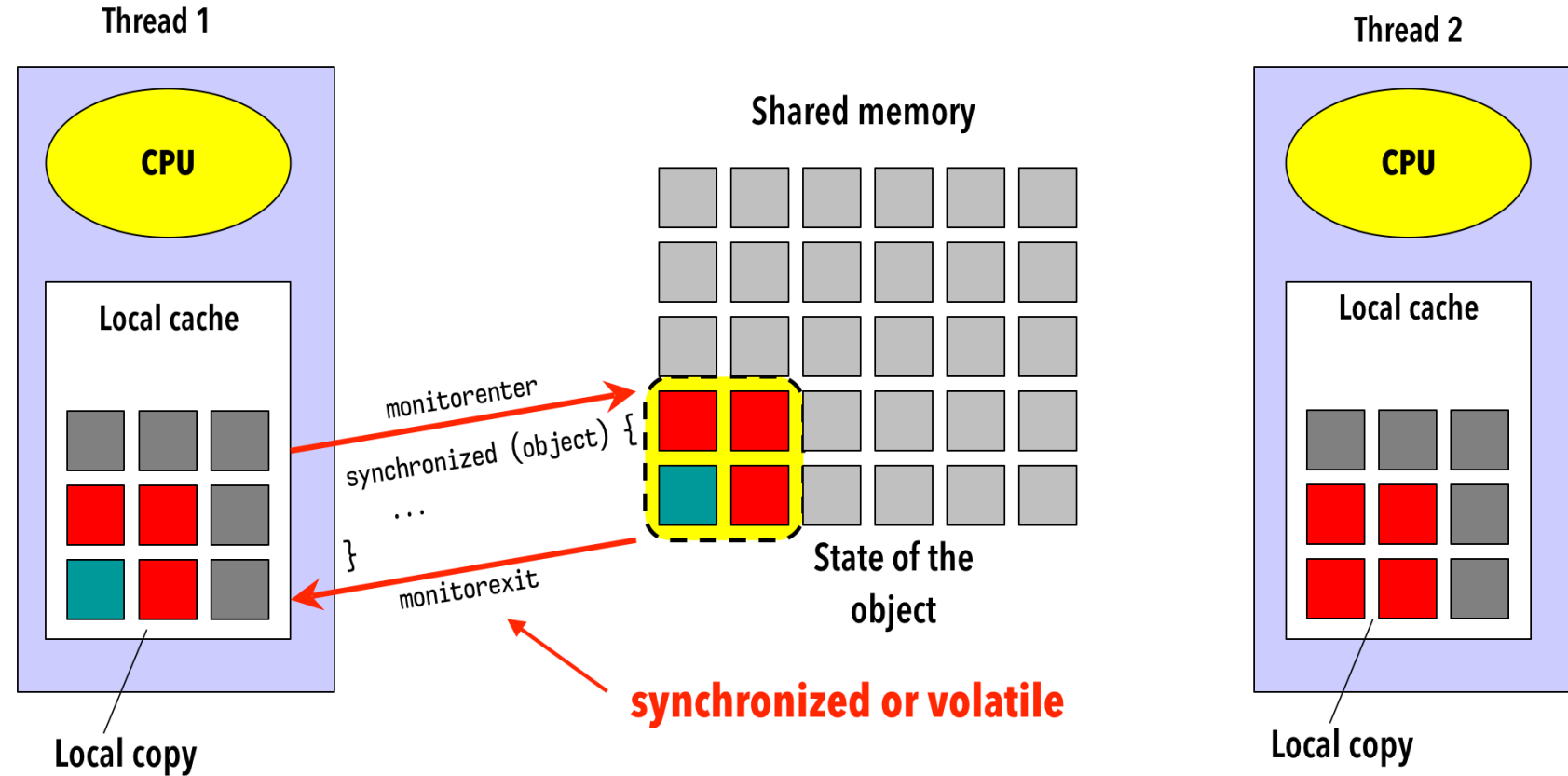
Thread safety

Variáveis voláteis possuem as seguintes características:

- operações atômicas;
- mudanças visíveis globalmente;
- são instáveis.

synchronized (objeto)

1. { Obtém trava
2. Atualiza cache local com dados da memória compartilhada
3. Manipula dados localmente (interior do bloco)
4. } Persiste dados locais na memória compartilhada
5. Libera trava



Threads podem armazenar variáveis em um cache local, que não garante sincronização com a memória compartilhada. Desta forma, duas CPUs podem ver dados diferentes quando acessam a mesma variável

Thread safety

- Para garantir a sincronização dos dados, variáveis que precisam ser compartilhadas entre threads devem ser marcadas com o modificador volatile. Assim garante-se que ela seja acessível em todos os threads paralelos e concorrentes.
- No exemplo, um thread **testa a propriedade de um objeto acessível através de uma referência e outro thread modifica a variável done** que é usada pelo primeiro thread para decidir se continua ou não uma tarefa infinita.
- Estas variáveis devem ser declaradas como volatile para garantir que seu estado seja sincronizado entre threads.

```
public class WordFinder implements Runnable {  
    public volatile boolean done;  
    public volatile int count;  
    public List<String> words = new Collections.concurrentList(ArrayList<>());  
  
    public void run() {  
        while(!done) { // leitura feita por thread secundário  
            words.add(lookForWord("thread")); // alteração feita por thread secundário  
            count = words.size();  
        }  
    }  
}
```


Thread safety

- , um thread **testa a propriedade de um objeto acessível através de uma referência** e outro thread **modifica a variável done** que é usada pelo primeiro thread para decidir se continua ou não uma tarefa infinita.
- Estas variáveis devem ser declaradas como volatile para garantir que seu estado seja sincronizado entre threads.

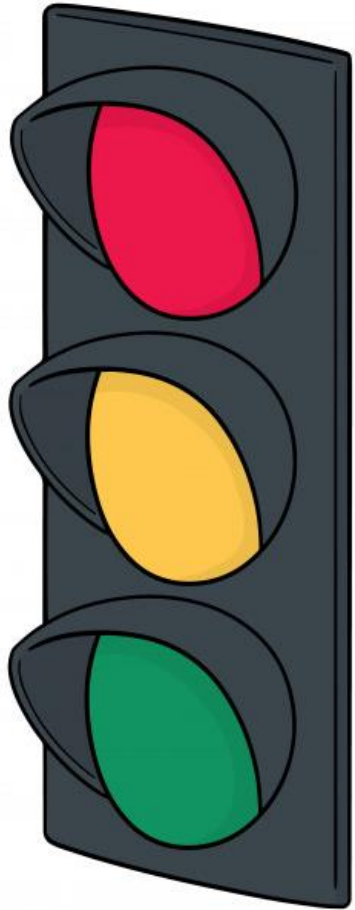
```
public class TestWordFinder {  
    public static void main(String[] args) {  
        WordFinder runnable = new WordFinder(...);  
        new Thread(runnable).start();  
        while(runnable.count <= 5) { // leitura feita por thread main  
            runnable.done = false; // alteração feita por thread main  
        }  
        runnable.done = true; // alteração feita por thread main  
    }  
}
```

```
public class WordFinder implements Runnable {  
    public volatile boolean done;  
    public volatile int count;  
    public List<String> words = new Collections.concurrentList(ArrayList<>());  
  
    public void run() {  
        while(!done) { // leitura feita por thread secundário  
            words.add(lookForWord("thread")); // alteração feita por thread secundário  
            count = words.size();  
        }  
    }  
}
```

Linguagem de Programação III (LP35A)

Comunicação entre threads

Thread safety

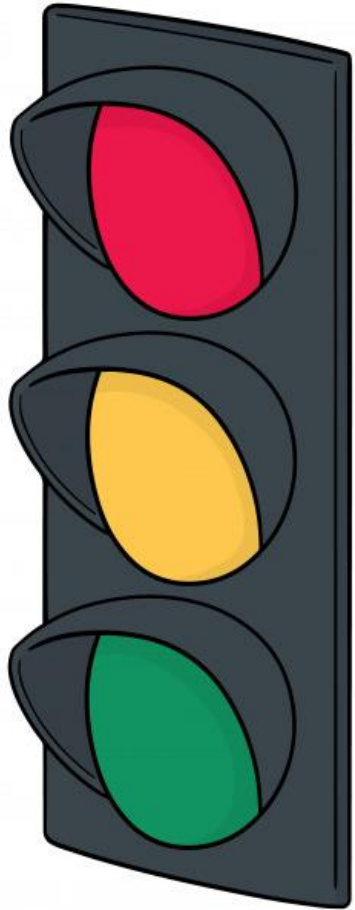


`wait()`, `notify()` e `notifyAll()`

Estes métodos da classe **Object** implementam o conceito de monitores sem utilizar espera ativa. Ao invés disso, notificam as *threads* indicando se estas devem ser suspensas ou se devem voltar a ficar em execução.

O *lock* do objeto chamado pela *thread* para realizar as notificações será utilizado. Por isso, antes de chamar um dos três métodos o *lock* deve ser obtido utilizando-se o comando **synchronized**.

Thread safety



Se um thread chama o método `wait()` de um objeto, ele libera sua trava de acesso exclusivo e entra em estado de espera. Ele só acordará, readquirindo a trava, quando receber uma notificação gerada por outro thread que chame `notify()` no mesmo objeto.

Chamar `notify()` em um objeto notifica um thread que está esperando a trava. Não há como escolher qual thread será notificado. Se não houver nenhum thread esperando, `notify()` simplesmente retorna (a notificação é perdida).

O método `notifyAll()` deve ser usado se há threads que esperam condições *diferentes* disputando acesso a um objeto. A trava do objeto sabe quais threads esperam, mas não sabe que condição que esperam. Como `notify()` *notifica apenas um thread aleatório*, ele poderá notificar um thread espera uma condição que não está presente. O método `notifyAll()` evita esse problema enviando notificações para *todos* os threads que estão esperando.

Thread safety

A chamada do método `wait()` deve sempre estar associada a uma *condição*, que deve ser testada pelo thread logo que a trava for adquirida. Essa condição deve ser testada dentro de um bloco `while`. (Um `if` não é suficiente, pois quando o thread acorda após esperar no `wait()`, a condição precisa ser testada novamente para saber se *ainda* é válida.) Como `notifyAll()` acorda vários métodos ao mesmo tempo, o `while` é fundamental para que o thread teste sua condição após acordar e decida se a condição é a esperada, liberando a trava se não for. Portanto, o padrão para usar `wait()` é:

```
synchronized(trava) { // ou método
    while(!condição) {
        trava.wait();
    }
    // Linhas de código para realizar a tarefa quando a condição for verdadeira
    notifyAll(); // ou notify(), se todos os threads esperam a mesma condição
}
```

Thread safety

- A melhor maneira de entender o funcionamento desses métodos é através de um exemplo.
- A classe SharedObject representa um objeto que compartilha um campo inteiro value.
- O método set() grava um valor nesse campo, e o método reset grava -1.
- Observe que ambos são synchronized.

```
public class SharedObject {
    private volatile int value = -1;
    public boolean isSet() { return value != -1; }
    public synchronized boolean set(int v) {
        try {
            while(isSet()) { // enquanto houver valor definido, espere
                wait();
            }
            value = v;
            System.out.println(Thread.currentThread().getName() + ": PRODUCED: " + value);
            notifyAll(); // avisa a produtores e consumidores (notify() avisa a um deles)
            return true;
        } catch (InterruptedException e) { return false; }
    }

    public synchronized boolean reset() {
        try {
            while (!isSet()) { // enquanto não houver valor definido, espere
                wait();
            }
            System.out.println(Thread.currentThread().getName() + ": CONSUMED: " + value);
            value = -1;
            notifyAll(); // avisa a todos os threads
            return true;
        } catch (InterruptedException e) { return false; }
    }
}
```

Thread safety

1. Quando o método `set()` é chamado, ele verifica se a variável `value` está livre (se o valor armazenado for -1).
2. Se já houver um valor armazenado, ele chama `wait()` e é suspenso, entrando em estado de espera e liberando a trava do objeto.
3. Se estiver livre, ele copia o valor recebido para `value` e
4. em seguida notifica todos os threads, também liberando a trava do objeto.

```
public class SharedObject {
    private volatile int value = -1;
    public boolean isSet() { return value != -1; }
    public synchronized boolean set(int v) {
        try {
            1 while(isSet()) { // enquanto houver valor definido, espere
                2 wait();
            }
            3 value = v;
            System.out.println(Thread.currentThread().getName() + ": PRODUCED: " + value);
            4 notifyAll(); // avisa a produtores e consumidores (notify() avisa a um deles)
            return true;
        } catch (InterruptedException e) { return false; }
    }

    public synchronized boolean reset() {
        try {
            while (!isSet()) { // enquanto não houver valor definido, espere
                wait();
            }
            System.out.println(Thread.currentThread().getName() + ": CONSUMED: " + value);
            value = -1;
            notifyAll(); // avisa a todos os threads
            return true;
        } catch (InterruptedException e) { return false; }
    }
}
```

Thread safety

1. Quando o método `reset()` é chamado, ele verifica se a variável `value` tem um valor diferente de `-1`.
2. Se o valor for já `-1`, o thread chama `wait()` para esperar que um produtor grave um valor positivo.
3. Se tiver, imprime o valor e depois muda para `-1`.

Os métodos de `SharedObject` podem ser chamados por threads separados.

```
public class SharedObject {
    private volatile int value = -1;
    public boolean isSet() { return value != -1; }
    public synchronized boolean set(int v) {
        try {
            while(isSet()) { // enquanto houver valor definido, espere
                wait();
            }
            value = v;
            System.out.println(Thread.currentThread().getName() + ": PRODUCED: " + value);
            notifyAll(); // avisa a produtores e consumidores (notify() avisa a um deles)
            return true;
        } catch (InterruptedException e) { return false; }
    }

    public synchronized boolean reset() {
        try {
            1 while (!isSet()) { // enquanto não houver valor definido, espere
                2 wait();
            }
            3 System.out.println(Thread.currentThread().getName() + ": CONSUMED: " + value);
            value = -1;
            notifyAll(); // avisa a todos os threads
            return true;
        } catch (InterruptedException e) { return false; }
    }
}
```


Thread safety

1. O método `run()` da **classe `Producer`** chama várias vezes apenas o método `set()`, passando um valor aleatório como argumento.
2. O valor retornado por `set()` (`true` ou `false`) define se o loop continua ou não. Se o loop for interrompido, o thread termina:

```
public class Producer implements Runnable {  
    private SharedObject shared;  
    private static final int TENTATIVAS = 3;  
  
    Producer(SharedObject shared) { this.shared = shared; }  
  
    @Override public void run() {  
        1 for (int i = 0; i < TENTATIVAS; i++) {  
            if( !shared.set(new Random().nextInt(1000)) )  
                2 break; // termina o thread se set() retornar false (foi interrompido)  
        }  
        System.out.println(Thread.currentThread().getName() + ": Producer DONE.");  
    }  
}
```

Thread safety

- A classe Consumer é similar, mas chama o método reset():

```
public class Consumer implements Runnable {  
    private SharedObject shared;  
    private static final int TENTATIVAS = 3;  
  
    Consumer(SharedObject shared) { this.shared = shared; }  
  
    @Override public void run() {  
  
        for (int i = 0; i < TENTATIVAS; i++) {  
            if(!shared.reset())  
                break; // termina thread se retornar false (foi interrompido)  
        }  
        System.out.println(Thread.currentThread().getName() + ": Consumer DONE.");  
    }  
}
```

Thread safety

1. Finalmente, a **classe ProducerConsumer** cria quatro threads (dois produtores e dois consumidores) que irão rodar concorrentemente.
2. No final o thread main irá esperar por todos os threads, mas se algum não terminar no timeout de 15 segundos, ele será interrompido.

```
public class ProducerConsumerExample {  
    public static void main(String[] args) {  
        SharedObject o = new SharedObject();  
        String[] names = {"C1", "C2", "P1", "P2"};  
        1 Thread[] threads = { new Thread(new Consumer(o)), new Thread(new Consumer(o)),  
                                new Thread(new Producer(o)), new Thread(new Producer(o)) };  
        for(int i = 0; i < threads.length; i++) {  
            threads[i].setName(names[i]);  
            threads[i].start();  
        }  
  
        try {  
            for(Thread t: threads) {  
                2 t.join(15000); // will wait up to 15 seconds for each thread to finish  
                if(t.isAlive()) { t.interrupt(); }  
            }  
        } catch (InterruptedException ignored) {}  
        System.out.println("Main DONE.");  
    }  
}
```

Thread safety

- **Os produtores e consumidores tem 3 oportunidades de produção e consumo** cada (pode ter havido mais tentativas que foram perdidas porque threads acordaram com a condição errada).
- Os dois produtores produzem seis valores, que são consumidos pelos dois threads consumidores.
- A listagem ilustra uma execução do programa:

```
P1: PRODUCED: 616.  
C1: CONSUMED: 616.  
P1: PRODUCED: 768.  
C2: CONSUMED: 768.  
P2: PRODUCED: 773.  
C2: CONSUMED: 773.  
  
P1: PRODUCED: 835.  
P1: Producer DONE.  
C1: CONSUMED: 835.  
P2: PRODUCED: 933.  
C2: CONSUMED: 933.  
C2: Consumer DONE.  
P2: PRODUCED: 877.  
P2: Producer DONE.  
Main DONE.  
C1: CONSUMED: 877.  
C1: Consumer DONE.
```


Linguagem de Programação III (LP35A)

Exercícios

Orientações Gerais

Atividades aula 03

- 1) Monte todos os códigos-exemplos explorados em aula e execute para fixar os conceitos abordados;
- 2) Faça os exercícios propostos e poste sua solução no GITHUB. Coloque seu nome e matricula nos comentários dos códigos desenvolvidos;
- 3) Os códigos devem ser postados em formato doc ou txt para facilitar o teste em qualquer IDE;
- 4) Você poderá desenvolver os exercícios propostos na IDE que julgar mais confortável;
- 5) Você precisa ter testado anteriormente os exemplos para poder aproveitar o código para o exercício 1.

▪ **Bom trabalho!!!**

Exemplos de codigos java

Edit

[Manage topics](#)

🕒 7 commits

🔗 1 branch

📦 0 packages

🏷 0 releases

👤 1 contributor

Branch: master ▾

New pull request

Create new file

Upload files

Find file

Clone or download ▾



elianasantos Add files via upload ...

Latest commit 1752fe3 now

📄 Aula_01.pdf	Add files via upload	now
📄 Aula_02.pdf	Add files via upload	8 minutes ago
📄 Exemplo Interrupt.docx	Add files via upload	12 hours ago
📄 Exemplo Join.docx	Add files via upload	1 hour ago
📄 Exemplo Sleep.docx	Add files via upload	2 hours ago
📄 Exercicios aula 02.pdf	Add files via upload	14 minutes ago

Obrigada!

