

Soquetes Java

Prof. Tiago Garcia de Senna Carneiro

2007

Duas Abordagens para Usar Threads

Implementar a interface Runnable

```
public class MyRunnable
    implements Runnable
{
    public void run() { ... };
}

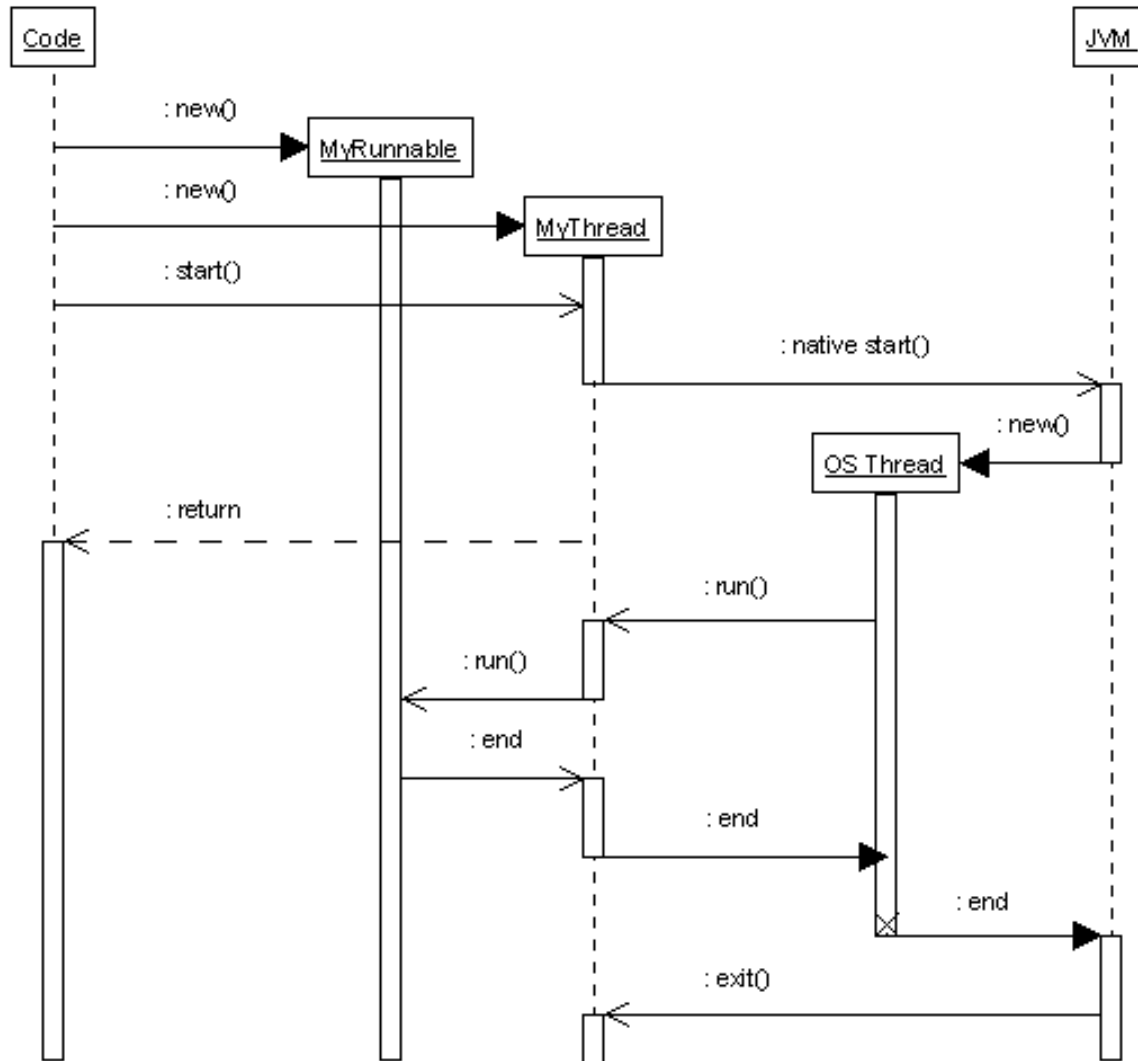
(new Thread(new
    MyRunnable(...))).start();
```

Extender a classe Thread

```
public class MyThread
    extends Thread
{
    public void run() { ... };
}

(new MyThread(...)).start();
```

Gerenciamento de Threads



Thread - Fundamentos

- Uma *thread* possui um contador de programa separado
... uma pilha, variáveis locais, etc.
- Classes, objetos, métodos, etc. não pertencem a uma *thread*
- Qualquer método pode ser executado por uma *thread*, inclusive simultaneamente

O método Sleep

```
public void run() {  
    for(;;) {  
        try {  
            sleep(1000); // Pause for 1 second  
        } catch (InterruptedException e) {  
            return;      // caused by thread.interrupt()  
        }  
        System.out.println("Tick");  
    }  
}
```

O método Sleep

Este código imprime Tick uma vez por segundo? Não.

sleep() ← é um limite inferior

O resto do loop leva um período indeterminado de tempo

```
public void run() {
    for(;;) {
        try {
            sleep(1000); // Pause for 1 second
        } catch (InterruptedException e) {
            return;      // caused by thread.interrupt()
        }
        System.out.println("Tick");
    }
}
```

Race Conditions

- In a concurrent world, always assume someone else is accessing your objects
- Other threads are your adversary
- Consider what can happen when simultaneously reading and writing:



Thread 1

f1 = a.field1

f2 = a.field2

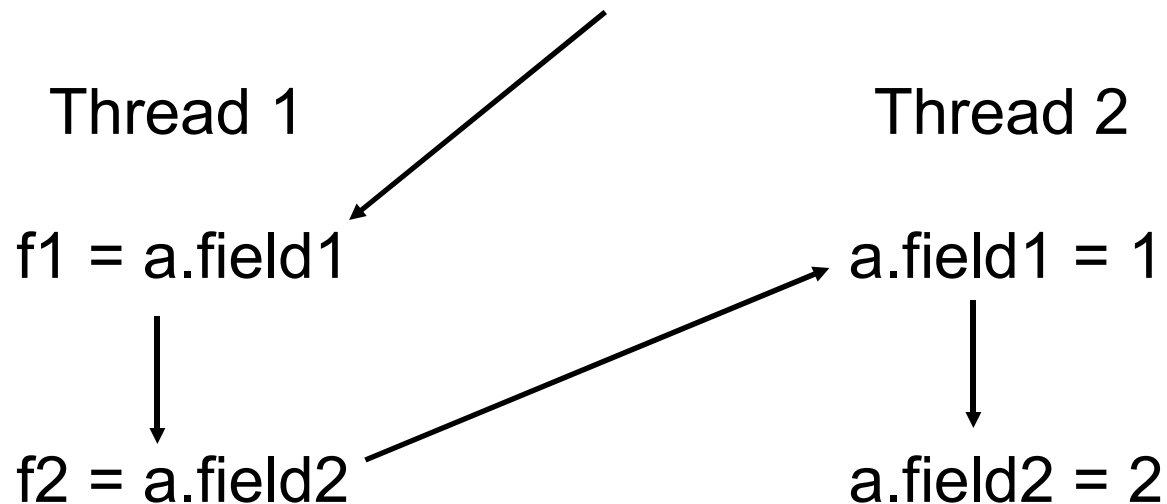
Thread 2

a.field1 = 1

a.field2 = 2

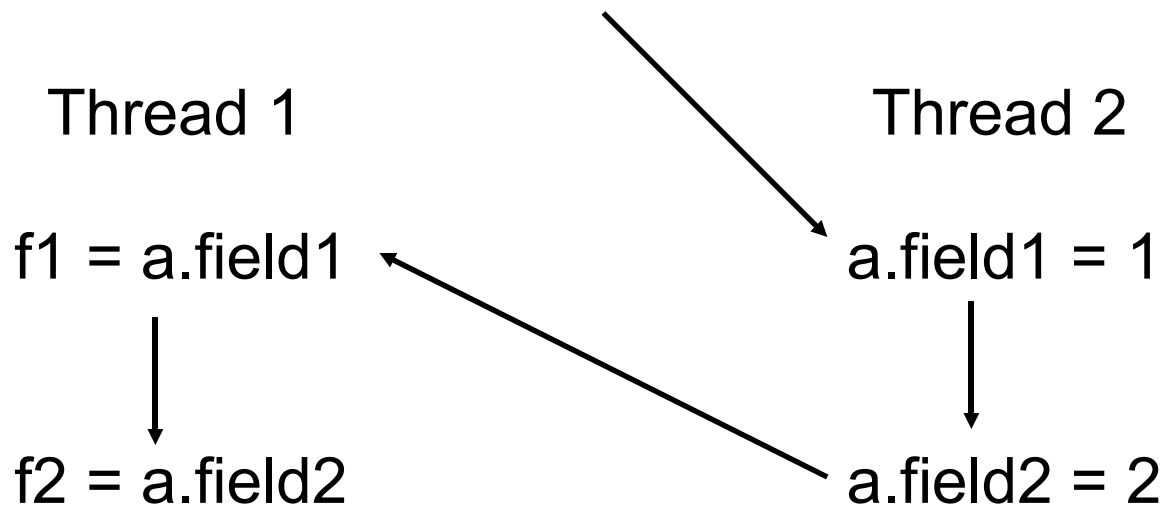
Race Conditions

- Thread 1 goes first
- Thread 1 reads original values



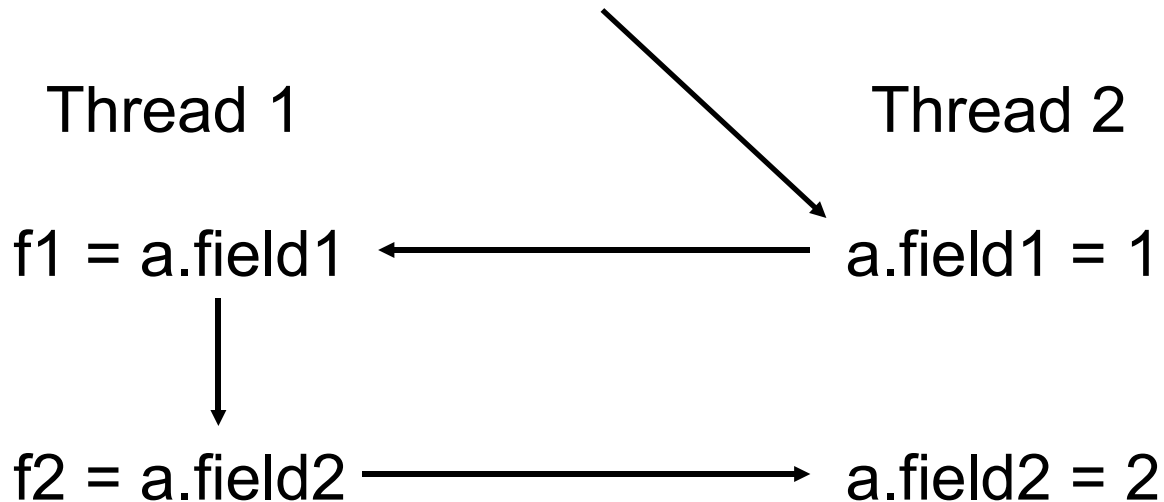
Race Conditions

- Thread 2 goes first
- Thread 1 reads new values



Race Conditions

- Interleaved execution
- Thread 1 sees one new value, one old value



Non-atomic Operations

- 32-bit reads and writes are guaranteed atomic
- 64-bit operations may not be
- Therefore,

```
int i; double d;
```

```
Thread 1      Thread 2
```

```
i = 10;
```

```
i = 20;
```

i will contain 10 or 20

```
d = 10.0;
```

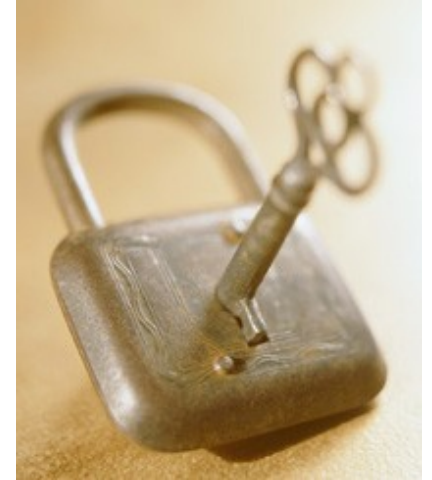
```
d = 20.0;
```

i might contain garbage



Per-Object Locks

- Each Java object has a lock that may be owned by at least one thread
- A thread waits if it attempts to obtain an already-obtained lock
- The lock is a counter: one thread may lock an object more than once



The Synchronized Statement

- A synchronized statement gets an object's lock before running its body

```
Counter mycount = new Counter;  
synchronized(mycount) {  
    mycount.count();  
}
```

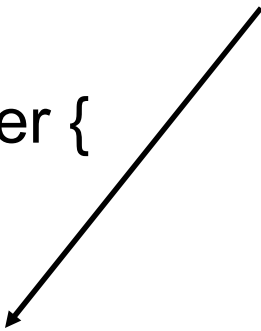
← “get the lock for mycount
before calling count()”

- Releases the lock when the body terminates

Synchronized Methods

```
class AtomicCounter {  
    private int _count;
```

“get the lock for the AtomicCounter object before running this method”



```
    public synchronized void count() {  
        _count++;  
    }  
}
```

This implementation guarantees at most one thread can increment the counter at any time

Deadlock

```
synchronized(Foo) {  
    synchronized(Bar) {  
        /* Deadlocked */  
    }  
}
```

```
synchronized(Bar) {  
    synchronized(Foo) {  
        /* Deadlocked */  
    }  
}
```

- **Rule: always acquire locks in the same order**

Priorities

- Each thread has a priority from 1 to 10 (5 typical)
- Scheduler's job is to keep highest-priority threads running
- `thread.setPriority(5)`

What the Language Spec. Says

- From *The Java Language Specification*

Every thread has a *priority*. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

- Vague enough for you?

Multiple threads at same priority?

- Language gives implementer freedom
- Calling `yield()` suspends current thread to allow other at same priority to run ... maybe
- Solaris implementation runs threads until they stop themselves (`wait()`, `yield()`, etc.)
- Windows implementation timeslices

Starvation

- Not a fair scheduler
- Higher-priority threads can consume all resources, prevent lower-priority threads from running
- This is called starvation

Waiting for a Condition

- Say you want a thread to wait for a condition before proceeding
- An infinite loop may deadlock the system

```
while (!condition) {}
```

- Yielding avoids deadlock, but is very inefficient

```
while (!condition) yield();
```



Java's Solution: wait() and notify()

- wait() like yield(), but requires other thread to reawaken it

```
while (!condition) wait();
```

- Thread that might affect this condition calls() notify to resume the thread
- Programmer responsible for ensuring each wait() has a matching notify()

wait() and notify()

- Each object has a set of threads that are waiting for its lock (its wait set)

```
synchronized (obj) { // Acquire lock on obj
    obj.wait();           // suspend
                        // add thread to obj's wait set
                        // relinquish locks on obj
}
```

In other thread:

```
obj.notify();           // enable some waiting thread
```

wait() and notify()

1. Thread 1 acquires lock on object
2. Thread 1 calls wait() on object
3. Thread 1 releases lock on object, adds itself to object's wait set
4. Thread 2 acquires lock on object
5. Thread 2 calls notify() on object (must own lock)
6. Thread 2 releases lock on object
7. Thread 1 is reawakened: it was in object's wait set
8. Thread 1 reacquires lock on object
9. Thread 1 continues from the wait()

wait() and notify()

- Confusing enough?
- notify() nondeterministically chooses one thread to reawaken (may be many waiting on same object)
 - What happens when there's more than one?
- notifyAll() enables all waiting threads
 - Much safer?

Building a Blocking Buffer

```
class OnePlace {  
    El value;  
  
    public synchronized void write(El e) { ... }  
    public synchronized El read() { ... }  
}
```

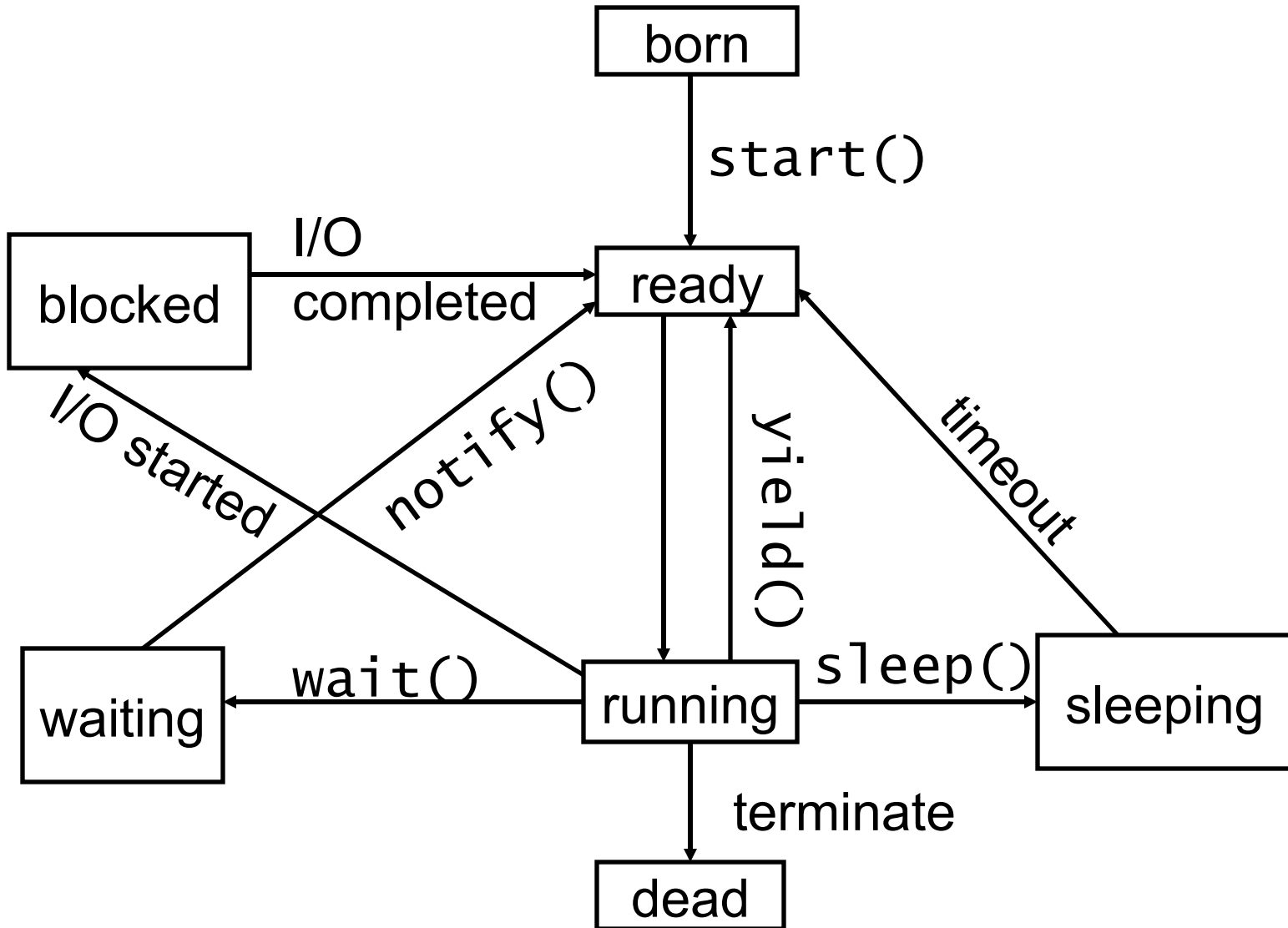
- Idea: One thread at a time can write to or read from the buffer
- Thread will block on read if no data is available
- Thread will block on write if data has not been read

Building a Blocking Buffer

```
synchronized void write(EI e) throws InterruptedException
{
    while (value != null) wait();           // Block while full
    value = e;
    notifyAll();                           // Awaken any waiting read
}
```

```
public synchronized EI read() throws InterruptedException
{
    while (value == null) wait();          // Block while empty
    EI e = value; value = null;
    notifyAll();                          // Awaken any waiting write
    return e;
}
```

Thread States



Deprecated Methods

- Do not use the ***stop()*** method to terminate a thread, it is deprecated
- Testes:
 - Olá mundo: `System.out.flush();`
 - Comunicação: `sleep()` x `yield()`;
 - Sincronização: `sleep()` x `yield()` na classe Pessoa; Suporte sem `wait()` ; Suporte com `yield()` no lugar de `wait()`;