

# Previously on PP: Shared Resources

# Synchronized incrementing and decrementing

```
public class Counter implements Runnable {
    public int ticks = -1;

    private Cell cell;
    private int delta;
    private int maxTicks;

    Counter(Cell cell, int delta, int maxTicks) {
        this.cell = cell;
        this.delta = delta;
        this.maxTicks = maxTicks;
    }

    @Override
    public void run() {
        ticks = 0;

        while (ticks < maxTicks) {
            cell.inc(delta);
            ++ticks;
        }
    }
}
```

```
Cell value: -799
Cell value: 667088
Cell value: -281765
Cell value: 147854
...
```

```
public class Main {
    public static void main(String[] args) {
        ...

        Counter up = new Counter(cell, 1, MAX_TICKS);
        Counter down = new Counter(cell, -1, MAX_TICKS);

        Thread upWorker = new Thread(up);
        Thread downWorker = new Thread(down);

        upWorker.start(); downWorker.start();
        upWorker.join(); downWorker.join();

        System.out.printf("Cell value: %d\n", cell.get());
    }
}
```

```
public class Cell {
    private long value;

    ...

    public void inc(long delta) {
        this.value += delta;
    }
}
```

# Updating shared state in parallel

Single statement in LongCell.inc

```
this.value += delta;
```

is executed in several small steps

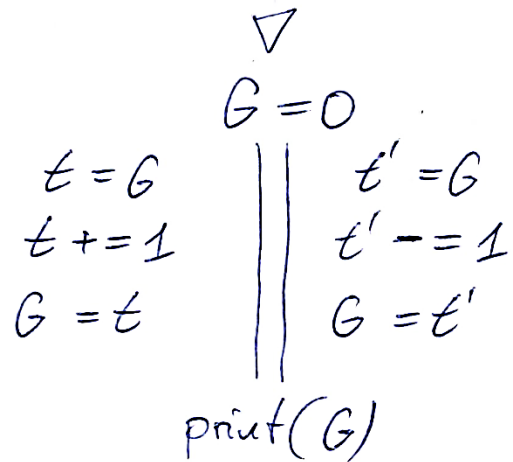
```
// relevant bytecode  
ALOAD 0  
DUP  
GETFIELD LongCell.value  
LLOAD 1  
LADD  
PUTFIELD LongCell.value
```

Many different interleavings possible, including **bad interleavings** in which state data is used

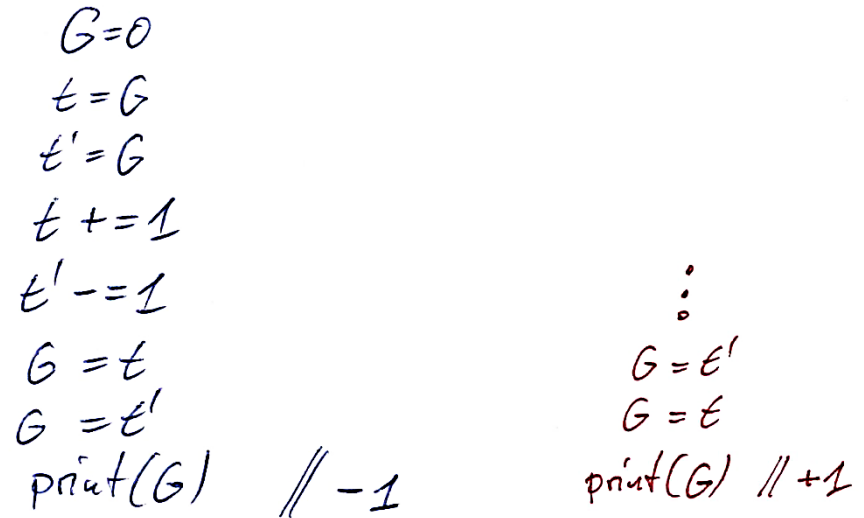
## Inc & Dec Example: Bad Interleavings

Main.global Value += delta      $T_1: \text{delta} \equiv +1$   
     $T_2: \quad \quad \equiv -1$

thus | essentially (on bytecode level)



One possible bad interleaving:



# synchronized

```
public class Cell {  
    private long value;  
  
    ...  
  
    public synchronized void inc(long delta) {  
        this.value += delta;  
    }  
}
```

body of method `inc()`  
is a **critical section**

synchronized enforces  
**mutual exclusion**  
and thereby prevents  
**bad interleavings**

# Shared memory interaction between threads

- In Java, all objects have an internal lock, called **intrinsic lock** or **monitor lock**
- **Synchronized operations** lock the object: while locked, no other thread can successfully lock the object
- Generally, if you access shared memory, **with a least one writing thread**, make sure it is **done under a lock**
- If not, your code is prone to a **data race**