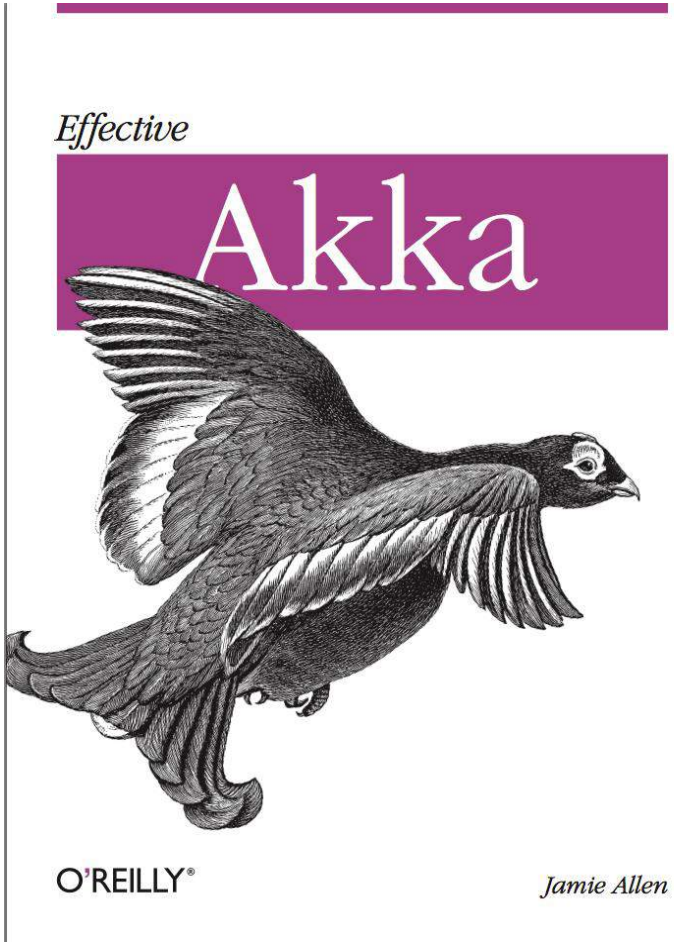
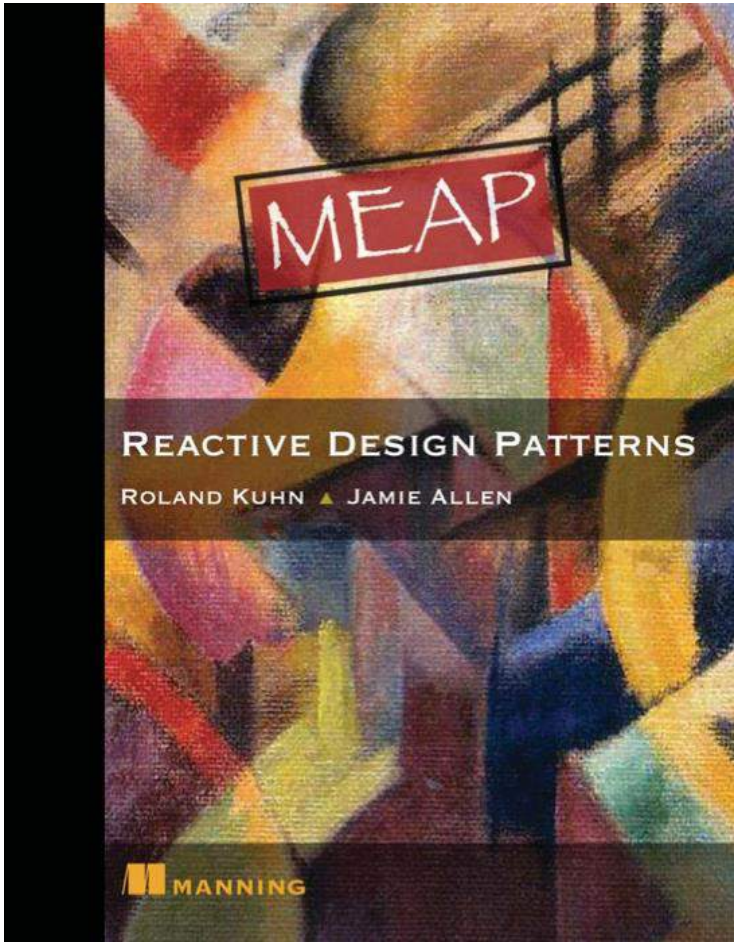

Typesafe

@jamie_allen


Effective
Akka
O'REILLY
Jamie Allen

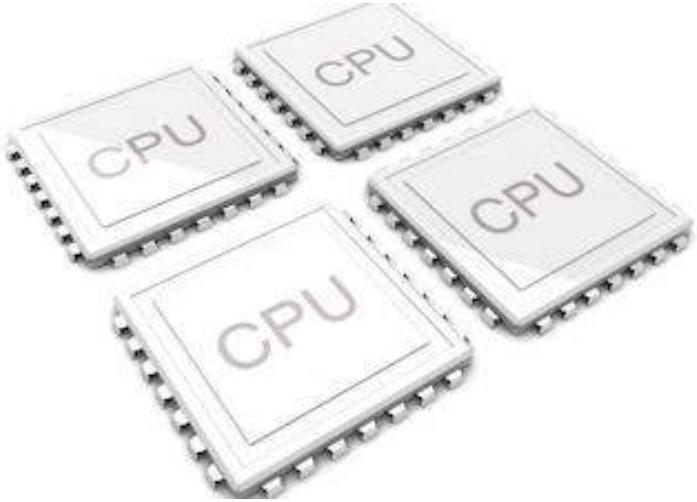

MEAP
REACTIVE DESIGN PATTERNS
ROLAND KUHN ▲ JAMIE ALLEN
MANNING

Agenda

- The Java Memory Model
- What do concurrent and parallel mean?
- How does the heap work?
- How do we protect shared state on the heap?

Scaling in the Multicore Era

- We have more more cores than ever (Intel Xeon E5-2699-v3 has 18 cores!)
- Shared mutable state, and the contention between threads of execution vying to update them simultaneously, is a performance killer
  - Blocking
  - Locking



The JVM is awesome!

- The power to leverage all cores on a box through a single, unified process
- The first formal memory model, with well-defined semantics for visibility

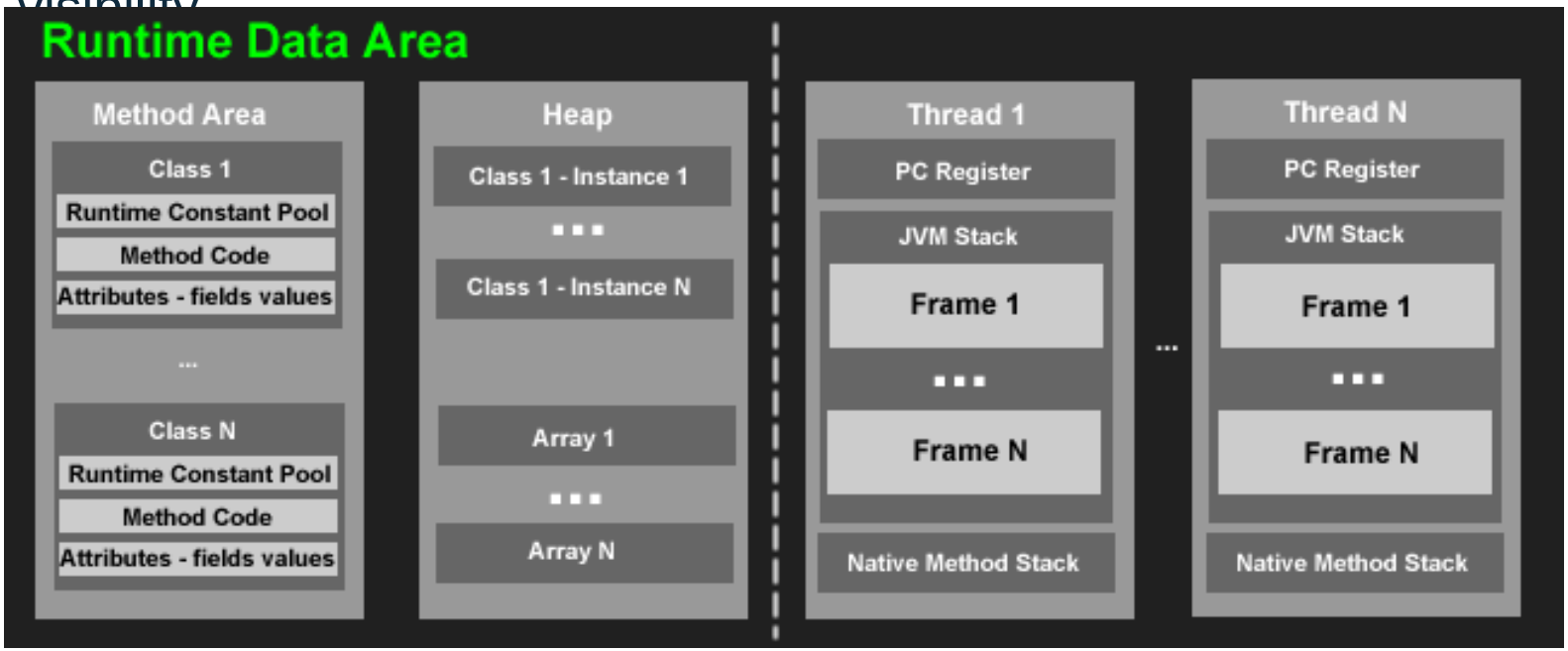


Diagram by Juan Pablo Francisconi
http://www.researchbeta.com/?p=334

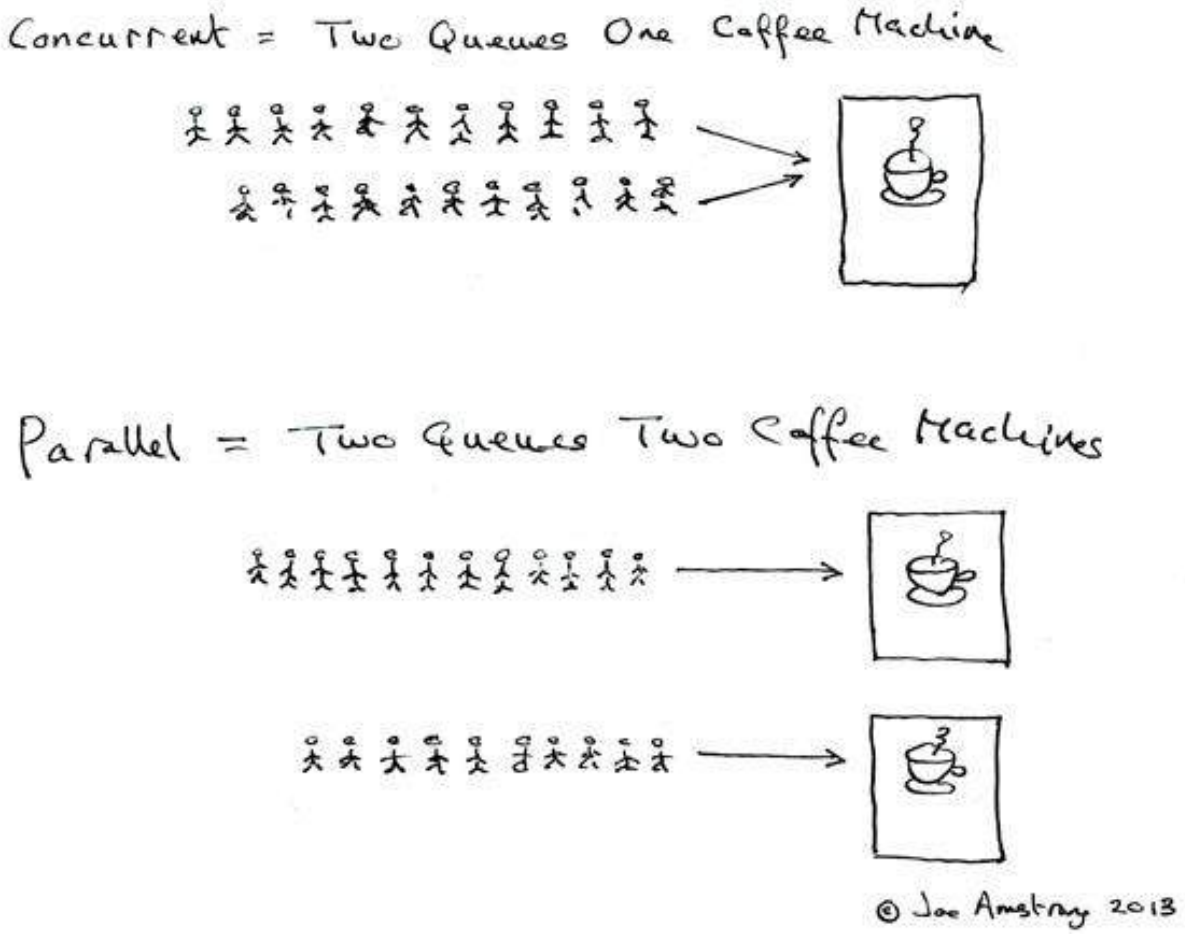
The Java Memory Model

- Two primary concepts:
  - **Happens-Before** – if A happens before B, the A's result should be visible to B
  - **Synchronize-With** – action results in write-through to RAM before next action

The Java Memory Model

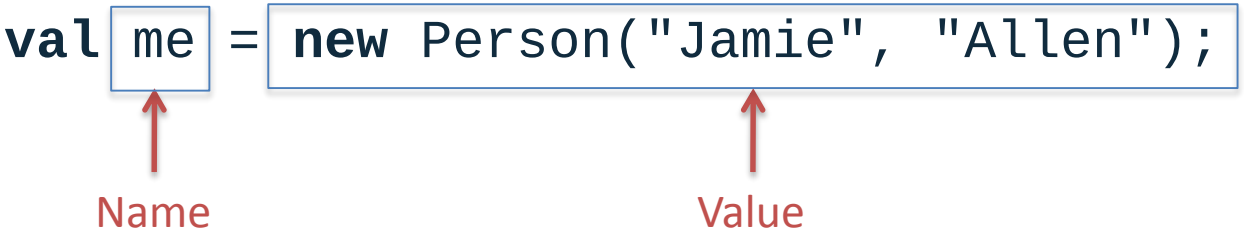
- There are no visibility guarantees between threads in a JVM without locks or memory barriers (**`volatile`**)
- Something may end up being visible to another thread by pure happenstance
- Contended locks are arbitrated at the kernel level, leading to expensive context switches

Concurrent and Parallel



Name Versus Value

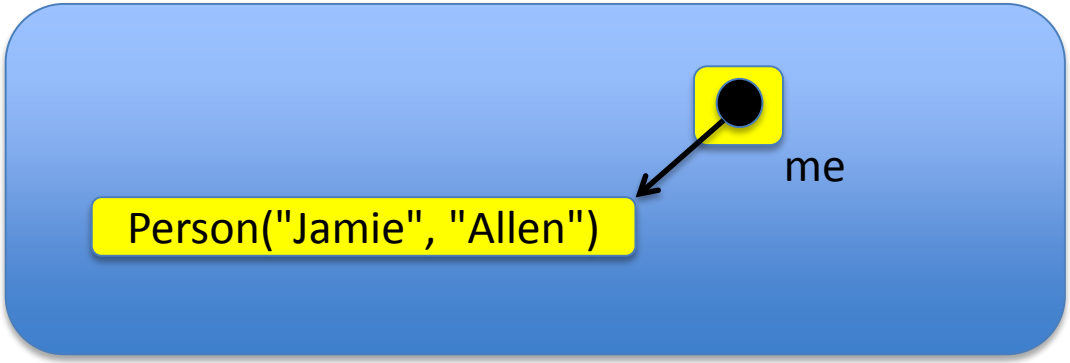
- The name is the handle you (or the compiler) bind to a reference
- The value is the actual state stored in the heap
- Therefore, the name is a reference to a location on the heap



Name Versus Value

- For instance variables, a heap location for the class instance holds the reference via the name field
- The value itself is located elsewhere (possibly contiguous, no guarantee)

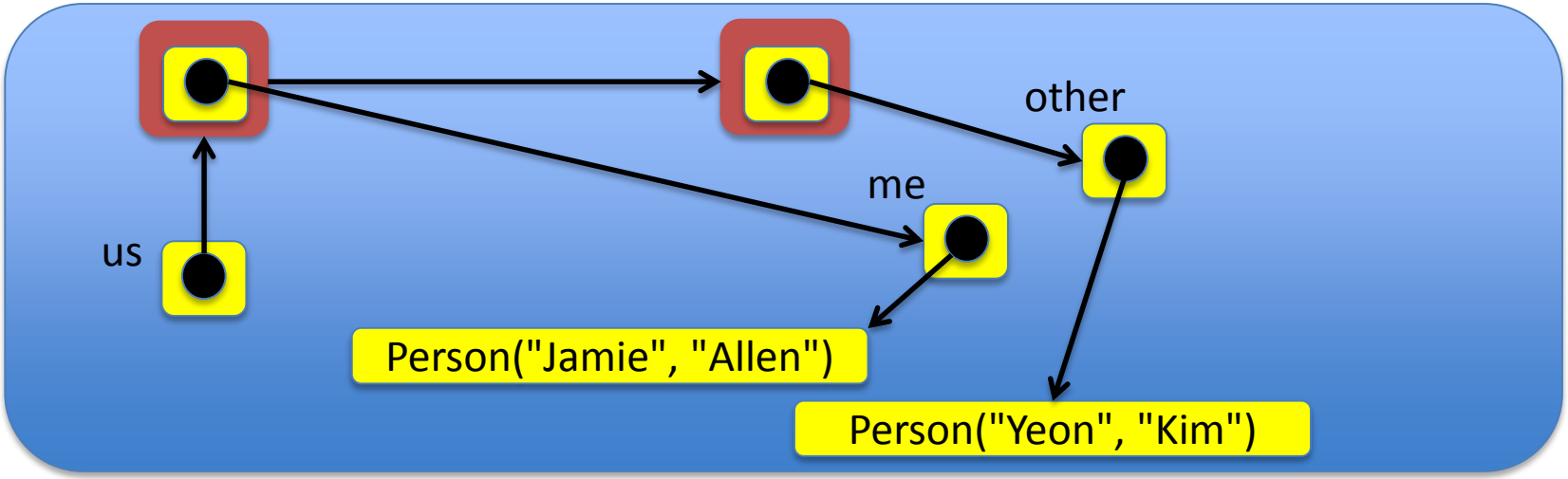
```
class Customer {
    val me = new Person("Jamie", "Allen");
    ...
}
```



Collections

- Collections are an aggregation of references to values
- Java collections are all mutable by default, developers have to use Guava to get immutable collections

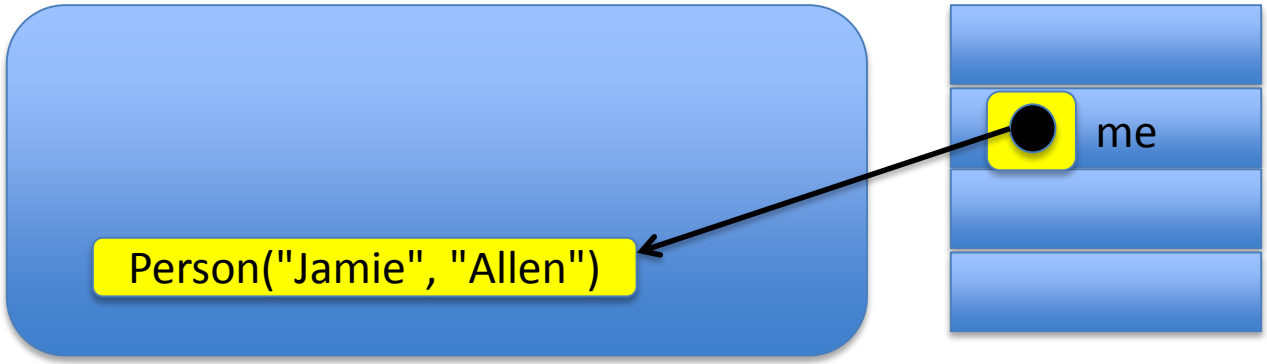
```
val me = new Person("Jamie", "Allen");
val other = new Person("Yeon", "Kim");
val us = List(me, other);
```



Method-Scoped Name Versus Value

- For variables defined in a method, the reference to the name is on the stack frame
- The value itself is still located elsewhere (possibly contiguous, no guarantee)

```
def myMethod() = {
    val me = new Person("Jamie", "Allen");
    ...
}
```



Final

- If we make a reference final in Java, it cannot be reassigned to a new value
- Easy to forget to use the keyword
- A mutable value remains mutable despite the usage of the **final** keyword

```
final Person me = new Person("Jamie", "Allen");
```

```
val me: Person = Person("Jamie", "Allen")
```

The Mutability Matrix of Pain

NAME \ VALUE	Mutable	Immutable
var		
val		

Implications of Immutability

- Uses more Eden/NewGen heap space
- Resizing Eden/NewGen larger means longer stop the world pauses, may cause Survivor spillover directly into Tenured/OldGen
- Measure the effects of attempts to tune GC at varying loads
- Only optimize where you must, over 99% of my customers don't even try and are still wildly successful

Volatile is Your Friend

- Use snapshots when mutability must be attained, and publish updates via **`volatile`** with the last write
- Readers should first read the one **`volatile`** field of multiple writes (reverse your order of access)

Caveat

- Just because you're using **`volatile`** doesn't mean that publishing cannot happen before you expect
- For greater consistency guarantees, **`volatile`** may not be appropriate

Use `volatile`!

- Don't be mutable, but if you have to, stay in the snapshot world with **`volatile`** and avoid locks
- Understand the visibility guarantees you need for your application, and consider the impact when you leave visibility to chance
- You can be more specific with read and write barriers if you go with `sun.misc.Unsafe`