



IBM Software Group

# *Concurrency Features in Java*

Rajini Sivaram  
IBM Hursley

rsivaram@uk.ibm.com

A horizontal bar containing a series of small, square icons. From left to right, the icons include: a green square, a yellow square, a red square, a purple square, a cyan square, a grayscale image of a building, a circular arrow icon, a grayscale image of a woman's face, a grayscale image of hands, a grayscale image of a person's head, and several grayscale squares of varying shades.

@business on demand software



# IBM Hursley Labs



- One of Europe's largest software labs
- Over the past 50 years, IBM Hursley has moved through hardware design to focus primarily on developing software.
- Strong links with around 20 leading UK Universities through initiatives such as the ThinkPad Challenge and collaborative research projects.





# IBM Hursley Labs



- The CICS development group came to Hursley in 1974.
  - \$1 Trillion revenue
  - Over 30 billion transactions involving more than \$1 trillion dollars are processed every day
- WebSphere MQ was developed at Hursley and first released in 1994.
  - MQ and related products have generated over \$2 billion revenue
- Java Technology Centre
  - Delivers IBM Java SDK on 12 platforms
- Emerging Technology Services, Storage Subsystems Group, Platform Technology Centre ...





# Agenda

- Introduction
- Java VM concurrency
- Application concurrency
- Concurrent applications in Java
- Lock-free concurrent data structures
- Summary





# Concurrency

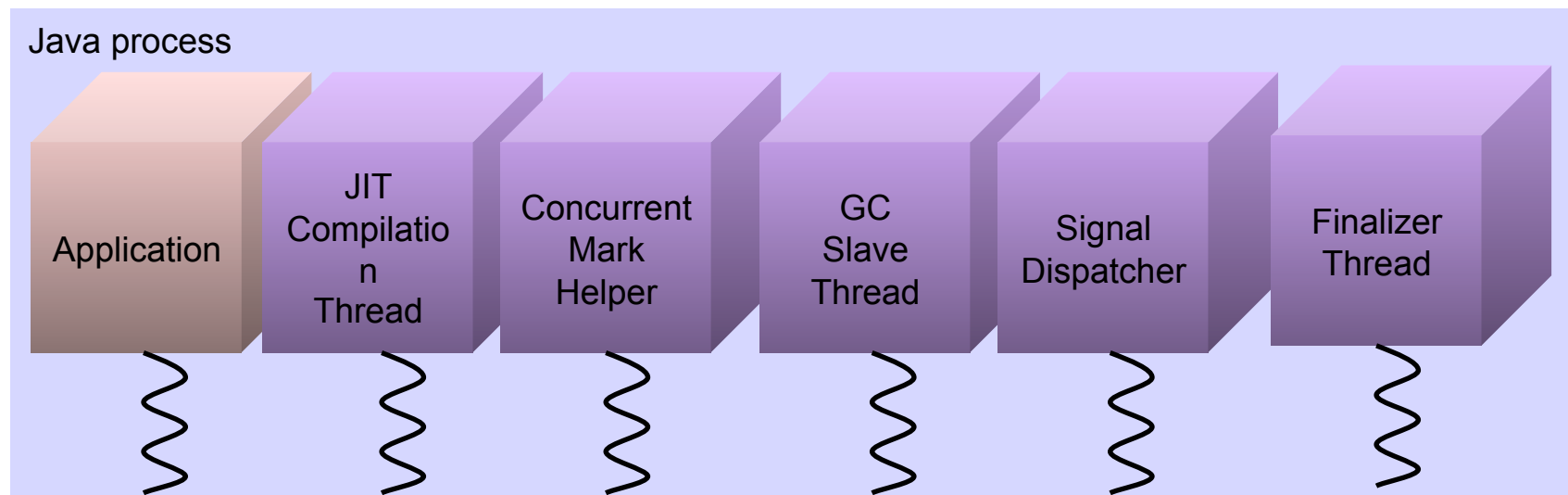
- Supercomputers
- SMPs
- Multicores





# Java VM

- Simple application – Hello World





# Application concurrency

- Simple example – C

```
void test() {  
    ....  
    pthread_create(&producerTid, attr, producer, queue);  
    pthread_create(&consumerTid, attr, consumer, queue);  
    ....  
    pthread_join(producerTid, NULL);  
    pthread_join(consumerTid, NULL);  
}
```

```
void producer(Queue *queue) {  
    ....  
    pthread_mutex_lock(queue->lock);  
    // Add to queue  
    pthread_mutex_unlock(queue->lock);  
}
```

```
void consumer(Queue *queue) {  
    ....  
    pthread_mutex_lock(queue->lock);  
    // Remove from queue  
    pthread_mutex_unlock(queue->lock);  
}
```





## Application concurrency - issues

- Difficult to program
- Difficult to understand
- Memory model - depends on processor architecture
- Threading and synchronization libraries - Non-portable
- Standard libraries now available, but....

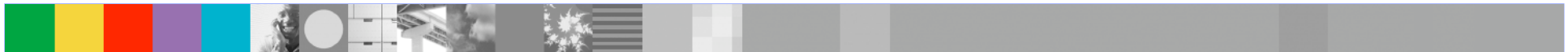




# Java application concurrency

- Simple Example

```
public class Producer extends Thread {
    public void run() {
        ....
        synchronized(queue) {
            queue.add(element);
        }
    }
}
public class Consumer extends Thread {
    public void run() {
        ....
        synchronized(queue) {
            element = queue.remove();
        }
    }
}
```





# Java application concurrency

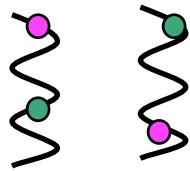
- Synchronization - First class language feature
  - ▶ Exclusive access
  - ▶ Changes visible to other threads
- Memory model - cross-platform
- Volatile variables - writes are visible to other threads
- Thread libraries - standard
- `Object.wait()`, `notify()`, `notifyAll()`





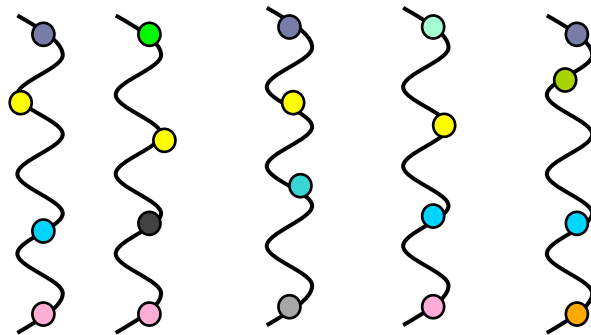
# Synchronization

- Deadlocks



- Scalability

- ▶ performance, correctness, debugging



```

void addWork(Queue queue, Object a) {
    synchronized (a) {
        ....
        synchronized (queue) {
            ....
        }
    }
}

void doWork(Queue queue) {
    synchronized (queue) {
        Object a = queue.remove();
        synchronized (a) {
            ....
        }
    }
}
    
```





# Java concurrency - pitfalls

- Deadlock
- Scalability
- Priority inversion
- Performance
  - ▶ Uncontented locks – inexpensive on standard Java
  - ▶ Contented locks
- Volatiles
  - ▶ Writes are visible to other threads but volatile does not imply atomic
  - ▶ eg. Atomic counters cannot be implemented using volatiles – read-modify-write





# Lock-free concurrent data structures

- Atomic
- Compare-and-swap
  - ▶ Hardware instruction
- Lock-free
  - ▶ Every step contributes to global progress
- Wait-free
  - ▶ Number of steps to complete an operation is bounded
- `java.util.concurrent` package
  - ▶ Fine grained synchronization
  - ▶ Scalable





# Atomic variables

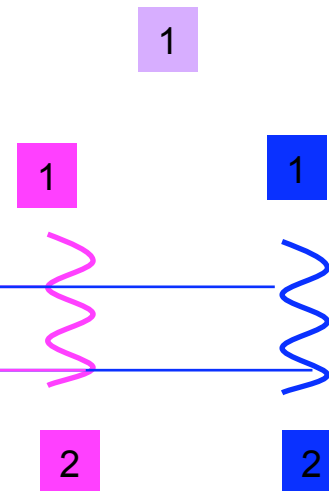
- Common simple use case – counter

- ▶ Atomic increment/decrement

```
synchronized void increment() {  
    counter++;  
}
```

- java.util.concurrent.AtomicInteger

```
public final int getAndIncrement() {  
    for (;;) {  
        int current = get();  
        int next = current + 1;  
        if (compareAndSet(current, next))  
            return current;  
    }  
}
```





# ReadWriteLock

- Producer-Consumer
  - ▶ Mutual-exclusion can lead to heavy contention when loaded
- Pair of locks
  - ▶ Exclusive write lock
  - ▶ Read lock can be held by multiple readers if there are no writers
- Reduced contention
  - But performance benefit only in certain cases, and only on multiprocessors





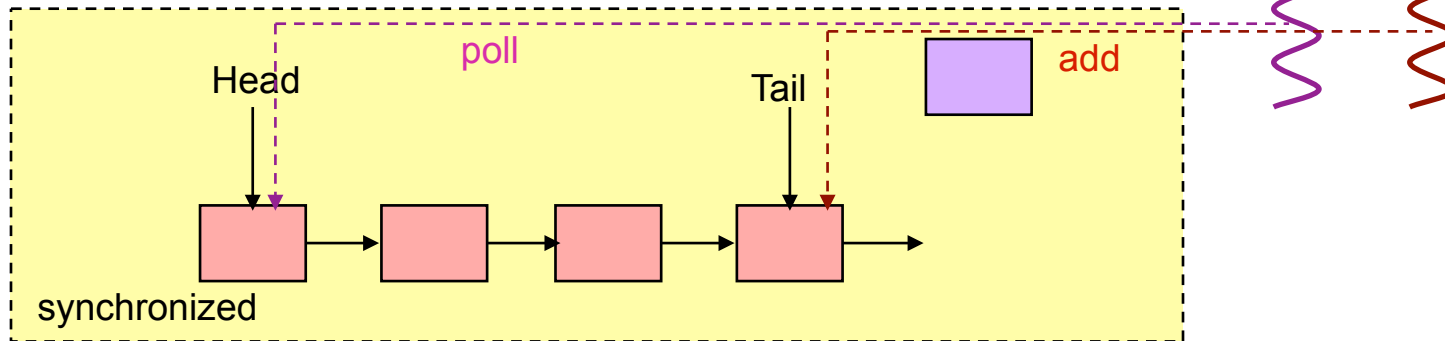
# ReentrantLock

- Same semantics as implicit monitor lock
- Additional capabilities
- Fairness setting
  - ▶ Fair locks are expensive but guarantee lack of starvation
- No implicit unlocking

```
lock.lock();  
try {  
    return queue.poll();  
} finally {  
    lock.unlock();  
}
```



# ConcurrentLinkedQueue



```
public E poll() {
    for (;;) {
        Node<E> h = head;
        Node<E> t = tail;
        Node<E> first = h.getNext();
        if (h == head) {
            if (h == t) {
                if (first == null) return null;
                else casTail(t, first);
            } else if (casHead(h, first)) {
                E item = first.getItem();
                if (item != null) {
                    first.setItem(null);
                    return item;
                }
            }
        }
    }
}
```

```
public boolean add(E o) {
    Node<E> n = new Node<E>(o, null);
    for(;;) {
        Node<E> t = tail;
        Node<E> s = t.getNext();
        if (t == tail) {
            if (s == null) {
                if (t.casNext(s, n)) {
                    casTail(t, n);
                    return true;
                }
            } else {
                casTail(t, s);
            }
        }
    }
}
```





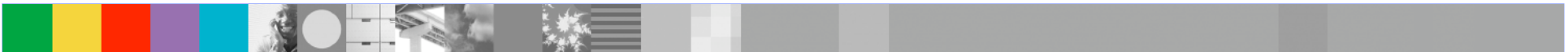
# ConcurrentSkipListMap - Javadoc

A **scalable concurrent** [ConcurrentNavigableMap](#) implementation. The map is sorted according to the [natural ordering](#) of its keys, or by a [Comparator](#) provided at map creation time, depending on which constructor is used.

This class implements a **concurrent variant** of [SkipLists](#) providing **expected average log(n) time cost** for the containsKey, get, put and remove operations and their variants. Insertion, removal, update, and access **operations safely execute concurrently** by multiple threads. **Iterators are weakly consistent**, returning elements reflecting the state of the map at some point at or since the creation of the iterator. They do *not* throw [ConcurrentModificationException](#), and **may proceed concurrently with other operations**. Ascending key ordered views and their iterators **are faster than descending ones**.

All Map.Entry pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do *not* support the Entry.setValue method. (Note however that it is possible to change mappings in the associated map using put, putIfAbsent, or replace, depending on exactly which effect you need.)

**Beware that, unlike in most collections, the size method is not a constant-time operation.** Because of the asynchronous nature of these maps, determining the current number of elements requires a traversal of the elements. Additionally, the bulk operations putAll, equals, and clear **are not guaranteed to be performed atomically**. For example, an iterator operating concurrently with a putAll operation might view only some of the added elements.





# Standard concurrency libraries

- `java.util.concurrent`
  - ▶ `ConcurrentHashMap`
  - ▶ `CopyOnWriteArraySet`
  - ▶ `ConcurrentSkipListMap`
  - ▶ `Semaphore`
  - ▶ `CyclicBarrier`
- `java.util.concurrent.locks`
  - ▶ `ReentrantLock`
  - ▶ `Condition`
- `java.util.concurrent.atomic`
  - ▶ `AtomicReference`
  - ▶ `AtomicStampedReference`
  - ▶ `AtomicIntegerFieldUpdater`





## Non-blocking data structures - summary

- Used in VMs, garbage collection
- Many standard data structures have non-blocking algorithms
- Advantages
  - ▶ Avoids locking hazards
    - Deadlocks
    - Priority inversion
  - ▶ Finer granularity – more parallelism
  - ▶ Can potentially improve performance on multicores
- Issues
  - ▶ Complex algorithms
  - ▶ Use with caution





# Transactional memory

- Transaction
  - ▶ **A**tomicity
  - ▶ **C**onsistency
  - ▶ **I**solation
  - ▶ **D**urability
- Transactional memory provides database-like transactional semantics for memory





# The Petaflop era

*The **sound barrier**. The **four minute mile**. The **moon**. So many milestones that once seemed insurmountable are now written in the record books of human and technological achievement. Now IBM has added another to that list: the **petaflop**.*





# Questions?

