

# Concurrency without explicit threads



- You can write concurrent applications that don't use explicit threads or synchronization
- Use built-in abstractions that support coordination and parallel execution

# Key concepts

- thread-safe collections
- concurrent collections
- blocking queues
- synchronizers
- thread locals
- executors

# Thread safe collections

- Standard collections or other abstractions that are intended to be thread safe
- Generally limited to one thread operating on them at a time
  - watch out for sequences that need to be atomic
- Can use Collections wrapped methods

# Concurrent collections

- Designed to allow multiple simultaneous accesses and updates
  - blocking only when they “conflict”
- Higher space overhead
  - not much time overhead
- Many of the concurrent collections do not allow null keys or values

# ConcurrentHashMap

- Allows simultaneous reads, and by default up to 16 simultaneous writers
- can increase the number of simultaneous writers
- Use `Collections.newSetFromMap` to construct concurrent set

# Special methods

- `V putIfAbsent(K key, V value)`
  - store the value only if the key has no mapping
  - return old value (null if none)
- `boolean remove(K key, V oldValue)`
  - remove mapping only if it has the specified value
- `boolean replace(K key, V oldValue, V newValue)`
  - update the mapping only if it has the specified value

# ConcurrentSkipLists

- Skip Lists are a probabilistic alternative to balanced trees
  - something I invented back in 1988
- ConcurrentSkipLists provide a concurrent sorted set implementation
  - and lots of other API improvements over TreeMap
- Java 6 only

# CopyOnWriteArrayList

- Using locking to ensure only one thread can update it at a time
- any update copies the backing array
- thus, read only operations don't need any locks
- iteration uses a snapshot of the array
  - allows concurrent modification and update
- Suitable only if updates rare

# Important use case

- Keeping track of listeners to an Observable
- while iterating through list of listeners, one of them might ask to be unsubscribed
- a “concurrent update”, even though we only have one thread

# Waiting for something to happen

- We briefly talked about `join()`
  - slides on web have been updated to discuss them
  - wait for another thread to terminate
- There are lots of ways to have a thread wait until things are right for it to do something
  - `wait/notify` were the way to do this before Java 5
  - but now we have new ways that are often better: blocking queues and synchronizers

# Blocking Queues and Dequeues

- A Queue is a first-in, first-out queue
- A dequeue is a Double-Ended Queue
  - allows addition and removal at both ends
  - a dequeue can also serve as a stack

# What happens when it can't immediately succeed?

	throws exception	returns special value	blocks
insert	add(e)	offer(e)	put(e)
remove	remove()	poll()	take()
examine	element	peek()	

# Queue notes

- Blocking queues also offer timed offer and poll methods
- Several different implementations, each with its own advantages
- `ConcurrentLinkedQueue`
  - doesn't support blocking, but allows for simultaneous addition/deletion
- `Array/Linked Blocking Dequeue/Queue`

# Synchronizers

- Other ways to wait for some condition to be true
- `CountDownLatch`
- `Semaphore`

# CountDownLatch

- A variable that can be decremented
  - never incremented
- You can wait for it to get to zero
- You can also find out the current value
  - most of the time, you won't need to find out the current value

# Semaphore

- Contains a count of the number of permits available
- You can acquire or release permits
  - no checking that you are releasing permits you have
  - really, just a counter
- Acquire blocks if not enough permits are available

# Fairness

- Consider a Blocking queue where you atomically remove multiple elements
- What happens if one person wants to atomically remove 10 elements from a queue that can contain up to 20 elements
- but there is a constant stream of other threads that want to remove smaller number of elements?

Starvation!

# Some abstractions have fair variants

- For example, fair semaphores and fair reentrant locks
- Generally, fair guarantees first-come, first-served
- But fair almost always reduces throughput
  - over and above implementation cost
  - letting running threads run improves throughput

**java.util.concurrent  
.atomic**

# AtomicInteger

- Encapsulates an integer
- Sort of like a volatile int
- but supports additional atomic operations:
  - `int getAndIncrement()`
  - `int decrementAndGet()`
  - `boolean compareAndSet(int expect, int update)`

# Atomic operations

- The atomic operations are very efficient
- Most processors provide some kind of atomic compare and swap instruction
- needed to efficiently implement locking

# Lots of Atomic classes

- There is an `AtomicX` class for every primitive type, and for references
- There are also classes that let you atomically update volatile fields, and ones that encapsulate arrays and allow you to perform atomic operations on array elements

# Executors

# Executor

- An object that executes submitted Runnable tasks
- Rather than starting a thread for each task

```
new Thread(new RunnableTask()).start()
```

- You ask an executor to do it

```
Executor executor = anExecutor;
```

```
executor.execute(new RunnableTask1());
```

```
executor.execute(new RunnableTask2());
```

# Executors can be simple

- The execute method might just run the task
- or create and start thread
- or do something more complicated

# java.util.concurrent.Executors

- provides many factory and utility methods for executors
- `newFixedThreadPool(int nThreads)`
- `newCachedThreadPool()`
  - creates threads as needed, reuses them

# Why thread pools?

- Some overhead to starting a thread
  - If your task takes a second to run, the overhead to negligible
- Running 100,000 threads is a bad idea
  - unless you have a monster machine