Task Execution

Tasks

- Natural boundaries for concurrency
- Improve responsiveness & throughput
- Enable parallelizing workloads
- Simplify structure and error containment
- Map well to client–request models

Example: Simple Web Server

- Seqential: Listing 6-1 link
- Multi-threaded: Listing 6-2 link

Disadvantages of Unbounded Thread Creation

- Thread lifecycle overhead
- Resource consumption
- Stability

The Executor Framework

```
public interface Executor {
    void execute(Runnable command);
}
```

Using Executors

```
private static final Executor exec =
    Executors.newFixedThreadPool(100);

exec.execute(task);
```

Executor Policies

- Ordering (FIFO/LIFO/Priority)
- Max concurrency
- Queue size
- Rejection policies
- Pre/post execution hooks

Thread Pools

- `FixedThreadPool`
- `CachedThreadPool`
- `SingleThreadExecutor`
- `ScheduledThreadPool`
- Example: web server with shutdown Listing 6-8 link

Callable and Future

```
public interface Callable<V> {
    V call() throws Exception;
}

public interface Future<V> {
    V get() throws InterruptedException, ExecutionExceptio
}
```

Future Lifecycle

1. Created
2. Submitted
3. Running
4. Cancelled or Completed

Example Code

- Future Listing 6-13 link
- Completion Service Listing 6-15
- Time budget task Listing 6-16

Identifying Task Boundaries

- Many small independent tasks
- Minimal coordination overhead
- Balanced workload distribution

Cancellation

- Threads are easy to start but **hard to stop safely**
- Java **cannot safely preempt** a thread
- Only cooperative cancellation is safe
- Shared data must not be left inconsistent
- Well-designed tasks require a **cancellation policy**

Summary

- Tasks are natural units for concurrency
- Executors decouple submission from execution
- Thread pools improve responsiveness and stability
- Callable/Future support values and cancellation