

# Concurrency

# Concurrency Utilities: JSR-166

- Enables development of simple yet powerful multi-threaded applications
  - > Like Collection provides rich data structure handling capability
- Beat C performance in high-end server applications
- Provide richer set of concurrency building blocks
  - > wait(), notify() and synchronized are too primitive
- Enhance scalability, performance, readability and thread safety of Java applications

# Why Use Concurrency Utilities?

- Reduced programming effort
- Increased performance
- Increased reliability
  - > Eliminate threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated
- Improved maintainability
- Increased productivity

# Concurrency Utilities

- Task Scheduling Framework
- Callable's and Future's
- Synchronizers
- Concurrent Collections
- Atomic Variables
- Locks
- Nanosecond-granularity timing

# Concurrency: Task Scheduling Framework

# Task Scheduling Framework

- **Executor/ExecuteService/Executors** framework supports
  - > standardizing invocation
  - > scheduling
  - > execution
  - > control of asynchronous tasks according to a set of execution policies
- **Executor** is an interface
- **ExecutorService** extends **Executor**
- **Executors** is factory class for creating various kinds of **ExecutorService** implementations

# Executor Interface

- **Executor** interface provides a way of de-coupling task **submission** from the **execution**
  - > execution: mechanics of how each task will be run, including details of thread use, scheduling

- Example

```
Executor executor = getSomeKindofExecutor();  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());
```

- Many **Executor** implementations impose some sort of limitation on how and when tasks are scheduled

# Executor and ExecutorService

## ExecutorService adds lifecycle management

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

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout,
                              TimeUnit unit);

    // other convenience methods for submitting tasks
}
```

# Creating ExecutorService From Executors

```
public class Executors {
    static ExecutorService
        newSingleThreadedExecutor();

    static ExecutorService
        newFixedThreadPool(int n);

    static ExecutorService
        newCachedThreadPool(int n);

    static ScheduledExecutorService
        newScheduledThreadPool(int n);

    // additional versions specifying ThreadFactory
    // additional utility methods
}
```

# pre-J2SE 5.0 Code

## Web Server—poor resource management

```

class WebServer {

    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);

        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            // Don't do this!
            new Thread(r).start();
        }
    }
}

```

# Executors Example

## Web Server—better resource management

```
class WebServer {
    Executor pool =
        Executors.newFixedThreadPool(7);

    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);

        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            pool.execute(r);
        }
    }
}
```

# Concurrency: Callables and Futures

# Callable's and Future's: Problem (pre-J2SE 5.0)

- If a new thread (callable thread) is started in an application, there is currently no way to return a result from that thread to the thread (calling thread) that started it without the use of a shared variable and appropriate synchronization
  - > This is complex and makes code harder to understand and maintain

# Callables and Futures

- Callable thread (Callee) implements **Callable** interface
  - > Implement **call()** method rather than **run()**
- Calling thread (Caller) submits **Callable** object to Executor and then moves on
  - > Through **submit()** not **execute()**
  - > The **submit()** returns a **Future** object
- Calling thread (Caller) then retrieves the result using **get()** method of **Future** object
  - > If result is ready, it is returned
  - > If result is not ready, calling thread will block

# Build CallableExample (This is Callee)

```
class CallableExample
    implements Callable<String> {

    public String call() {
        String result = "The work is ended";

        /* Do some work and create a result */

        return result;
    }
}
```

# Future Example (Caller)

```
ExecutorService es =  
    Executors.newSingleThreadExecutor();  
  
Future<String> f =  
    es.submit(new CallableExample());  
  
/* Do some work in parallel */  
  
try {  
    String callableResult = f.get();  
} catch (InterruptedException ie) {  
    /* Handle */  
} catch (ExecutionException ee) {  
    /* Handle */  
}
```

# Concurrency: Synchronizers

# Semaphores

- Typically used to restrict access to fixed size pool of resources
- New Semaphore object is created with same count as number of resources
- Thread trying to access resource calls **acquire()**
  - > Returns immediately if semaphore count  $> 0$
  - > Blocks if count is zero until **release()** is called by different thread
  - > **acquire()** and **release()** are thread safe atomic operations

# Semaphore Example

```
private Semaphore available;  
private Resource[] resources;  
private boolean[] used;  
  
public Resource(int poolSize) {  
    available = new Semaphore(poolSize);  
    /* Initialise resource pool */  
}  
public Resource getResource() {  
    try { available.aquire() } catch (IE) {}  
    /* Acquire resource */  
}  
public void returnResource(Resource r) {  
    /* Return resource to pool */  
    available.release();  
}
```

# Concurrency: Concurrent Collections

# BlockingQueue Interface

- Provides thread safe way for multiple threads to manipulate collection
- **ArrayBlockingQueue** is simplest concrete implementation
- Full set of methods
  - > **put ( )**
  - > **offer ( )** [non-blocking]
  - > **peek ( )**
  - > **take ( )**
  - > **poll ( )** [non-blocking and fixed time blocking]

# Blocking Queue Example 1

```
private BlockingQueue<String> msgQueue;  
  
public Logger(BlockingQueue<String> mq) {  
    msgQueue = mq;  
}  
  
public void run() {  
    try {  
        while (true) {  
            String message = msgQueue.take();  
            /* Log message */  
        }  
    } catch (InterruptedException ie) {  
        /* Handle */  
    }  
}
```

# Blocking Queue Example 2

```
private ArrayBlockingQueue messageQueue =
    new ArrayBlockingQueue<String>(10);

Logger logger = new Logger(messageQueue);

public void run() {
    String someMessage;
    try {
        while (true) {
            /* Do some processing */

            /* Blocks if no space available */
            messageQueue.put(someMessage);
        }
    } catch (InterruptedException ie) { }
}
```

# Concurrency: **Atomic Variables**

# Atomics

- **java.util.concurrent.atomic**
  - > Small toolkit of classes that support lock-free thread-safe programming on single variables

```
AtomicInteger balance = new AtomicInteger(0);
```

```
public int deposit(integer amount) {  
    return balance.addAndGet(amount);  
}
```

# Concurrency: Locks

# Locks

- Lock interface
  - > More extensive locking operations than synchronized block
  - > No automatic unlocking – use try/finally to unlock
  - > Non-blocking access using **tryLock()**
- ReentrantLock
  - > Concrete implementation of Lock
  - > Holding thread can call **lock()** multiple times and not block
  - > Useful for recursive code

# ReadWriteLock

- Has two locks controlling read and write access
  - > Multiple threads can acquire the read lock if no threads have a write lock
  - > If a thread has a read lock, others can acquire read lock but nobody can acquire write lock
  - > If a thread has a write lock, nobody can have read/write lock
  - > Methods to access locks

```
rwl.readLock().lock();  
rwl.writeLock().lock();
```

# ReadWrite Lock Example

```
class ReadWriteMap {
    final Map<String, Data> m = new TreeMap<String, Data>();
    final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();
    final Lock r = rwl.readLock();
    final Lock w = rwl.writeLock();
    public Data get(String key) {
        r.lock();
        try { return m.get(key) }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
    public void clear() {
        w.lock();
        try { m.clear(); }
        finally { w.unlock(); }
    }
}
```

# Concurrency