Thread objects

java.lang.Thread

- Each thread is associated with an instance of the class Thread

Two strategies for using Thread objects

- To *directly control* thread creation and management
  - Instantiate Thread each time for an asynchronous task
  - Abstract *thread management* from the rest of the application
  - Pass the tasks to an Executor
Today

Low-level: basic building blocks
  ‣ Thread API
  ‣ Wait and notify mechanism

High-level: concurrency API
  ‣ Executor framework
Thread API
How to create a thread

1. Declare a class that implements the Runnable interface

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
    }
}
```

- Separates Runnable task from the Thread object that executes the task
- Applicable to high-level thread management APIs (Executor)
How to create a thread

2. Declare a class to be a subclass of Thread

```java
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread t = new HelloThread();
        t.start();
    }
}
```
java.lang.Thread: Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Getter</th>
<th>Setter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long id</td>
<td>✓</td>
<td></td>
<td>Identifier</td>
</tr>
<tr>
<td>int priority</td>
<td>✓</td>
<td>✓</td>
<td>Priority</td>
</tr>
<tr>
<td>String name</td>
<td>✓</td>
<td>✓</td>
<td>Name</td>
</tr>
<tr>
<td>boolean isDaemon</td>
<td>✓</td>
<td>✓</td>
<td>User or daemon thread</td>
</tr>
</tbody>
</table>
### java.lang.Thread: Queries

<table>
<thead>
<tr>
<th>Instance methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isAlive()</td>
<td>Is the current thread alive?</td>
</tr>
<tr>
<td>boolean isInterrupted()</td>
<td>Has the current thread been interrupted?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread currentThread()</td>
<td>Reference to the currently executing thread</td>
</tr>
<tr>
<td>boolean interrupted()</td>
<td>Has the current thread been interrupted?</td>
</tr>
</tbody>
</table>
# java.lang.Thread: Commands

<table>
<thead>
<tr>
<th>Instance methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void run()</code></td>
<td>Default: returns ⇒ override</td>
</tr>
<tr>
<td><code>void start()</code></td>
<td>Start a Thread instance and execute its <code>run()</code> method</td>
</tr>
<tr>
<td><code>void interrupt()</code></td>
<td>Interrupt the current thread</td>
</tr>
<tr>
<td><code>void join([long])</code></td>
<td>Block until the other thread exits [for at most the given milliseconds]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void sleep(long)</code></td>
<td>Stop temporarily (for the given milliseconds) the execution of the current thread</td>
</tr>
</tbody>
</table>
JMM: Happens-before rules for threads

Thread start rule

- \( T_1.\text{start()} \) happens-before every action in \( T_1 \)

Thread termination rule

- Any action in \( T_1 \) happens-before any action in \( T_2 \) that detects that \( T_1 \) has terminated
- Detection in \( T_2 \): \( T_1.\text{join()} \) returns or \( T_1.\text{isAlive()} == \text{false} \)

 Interruption rule

- In \( T_1 \): \( T_2.\text{interrupt()} \) happens-before interrupt detection (by any thread including \( T_2 \))
- Detection: throw \( \text{InterruptedException} \), invoke \( T_2.\text{isInterrupted()} \), \( \text{Thread.interrupted()} \)
public class SimpleThreads {
  public static void main(String args[]) throws InterruptedException {

    long patience = 1000 * 60 * 60; // 1 hour delay
    long startTime = System.currentTimeMillis();

    Thread t = new Thread(new MessageLoop()).start();

    while (t.isAlive()) {
      t.join(1000); // wait for t to finish (max. 1 second)
      if (((System.currentTimeMillis() - startTime) > patience) &&
          t.isAlive()) {
        t.interrupt(); // tired of waiting -> interrupt t
        t.join(); // wait indefinitely for t to finish
      }
    }
  }
}

See example at http://java.sun.com/docs/books/tutorial/essential/concurrency/simple.html
public class MessageLoop implements Runnable {
    public void run() {
        String importantInfo[] = { "A", "B", "C", "D" ];
        try {
            for (int i = 0; i < importantInfo.length; i++) {
                try {
                    Thread.sleep(4000); // pause for 4 seconds
                } catch (InterruptedException e) {
                    printMessage("I wasn't done!");
                }
                printMessage(importantInfo[i]);
            }
        }
    }
}

Thread control example: MessageLoop
Wait and notify
Wait sets and notification

Each Object has an associated lock and wait set

Wait set

- Set of threads
- Holds threads blocked by Object.wait() until notifications/wait done
- Used by wait(), notify(), notifyAll() and thread scheduling

Wait sets interact with locks

- t.wait(), t.notify(), t.notifyAll() must be called only when synchronization lock is hold on t
  - Otherwise IllegalMonitorStateException is thrown
Object.wait() and Object.wait(long)

If current thread T has been interrupted by another thread

- return

else T is blocked

- T is placed in wait set of obj
- T releases any locks for obj (keeps other locks)
  - Lock status is restored upon later resumption

```java
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    // Perform action appropriate to condition
}
```

Object.wait(long) waits for a maximum time given
Object.notify() and Object.notifyAll()

A thread T is arbitrarily chosen from wait set of obj

- No guarantees which thread

T re-obtains lock on obj

- T blocks until notify() releases the lock
- T may block if some other thread obtains lock first

T resumes after wait()

- wait() returns

notifyAll()

- Similar as notify() but for all threads in wait set of obj
Example with useless class

To illustrate the underlying mechanisms

```java
class X {
    synchronized void w() throws InterruptedException {
        before(); wait(); after();
    }
    synchronized void n() {
        notifyAll();
    }
    void before() {}
    void after() {}
}
```

Attention! Broken program: liveness failure ⇒ missed signal
T1: x.w()
acquire lock
before();
wait:
   release lock
   enter wait set

T2: x.w()
acquire lock
before();
wait:
   release lock
   enter wait set

T3: x.n()
wait for lock
acquire lock
notifyAll();
   release lock

exit wait set
wait for lock
acquire lock
after();
   release lock

Wednesday, January 20, 2010
Remarks

Place checks for *condition variables* in while loops

- Thread only knows that it has been waken up, must re-check

Methods with *guarded waits* are not completely atomic

- On `wait()` lock is released $\Rightarrow$ other thread can be scheduled
- Objects must be in consistent state before calling `wait()`
Typical usage

```java
public class PatientWaiter {
    @GuardedBy("this") private volatile boolean flag = false;
    public synchronized void waitTillChange() {
        while (!flag) {
            try {
                this.wait();
            } catch (InterruptedException e) {}
        }
        // whatever needs to be done after condition is true
    }
    public synchronized void change() {
        flag = true;
        this.notifyAll();
    }
}
```

Slipped condition if two synchronized blocks
Executor framework
Threaded web server

```java
public class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
    private static void handleRequest(Socket connection) { ... }
}
```

See example at http://www.javaconcurrencyinpractice.com/listings/ThreadPerTaskWebServer.java
Problems of the threaded solution

Discussion
- Up to a certain point: more threads improve throughput
- Beyond that: slow down, crash

Poor resource management
- Thread lifecycle overhead
  - Thread creation and teardown
- Resource consumption
  - More runnable threads than processors ⇒ may hurt performance
  - Memory, garbage collection
- Stability
  - Number of threads limited ⇒ OutOfMemoryError
Tasks versus threads

Task
- Logical unit of work

Thread
- Mechanism by which tasks can run asynchronously

Web server example
- Each task is executed in its thread
- Poor resource management
Need: High-level abstraction for task execution

Low-level constructs

- `wait()/notify()`

High-level concurrency API

- Prefer executors and tasks to threads
- Prefer concurrency utilities to `wait()/notify()`
public class TaskExecutionWebServer {
    private static final Executor exec = ...; // see later

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
Different Executor implementations

Behavior like ThreadPerTaskWebServer

```java
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

Behavior like a single threaded web server

```java
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```
public interface Executor {
    // Execute the given command at some time in the future
    void execute(Runnable command);
}
Executor implementations

Tasks may execute in

- a newly created thread
- an existing task-execution thread
- or the thread calling `execute()`

Tasks may execute sequentially or concurrently
Execution policies

Executor decouples submission from execution

Resource management tool

‣ What resources are available?
‣ Which QOS requirements?

Policies decide

‣ In what threads will tasks execute
‣ In what order? -- FIFO, LIFO, priority queue?
‣ How many concurrent tasks?
‣ How many tasks may be queued pending execution?
‣ If system overloaded: choose victim task? notify application?
‣ Actions before/after executing a task?
Thread pool

Advantages

- Amortize thread creation/teardown costs over multiple requests
- No latency of thread creation ⇒ better responsiveness
- Tuning parameter: size

worker thread
- request task
- execute task
- wait for next task

task
work queue
thread pool
Factory methods to create thread pools

```java
public class Executors {
    // maintain n threads, unbounded queue
    public static ExecutorService newFixedThreadPool(int n)
    // create threads as needed (reused), unbounded queue
    public static ExecutorService newCachedThreadPool()
    // create one thread, unbounded queue
    public static ExecutorService newSingleThreadExecutor()
    // delayed and periodic task execution
    public static ExecutorService newScheduledThreadPool(int size)

    // ... more methods... consider also overloaded variants
}
```
public class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
Executor lifecycle

Executor processes task asynchronously
  ▸ State of tasks may not be obvious

Executor provides service to applications: must be able to
  ▸ Shutdown
  ▸ Report status of tasks
  ▸ Also: executor implementation must shut down
    ▸ JVM can exit only after all threads have terminated
States of the ExecutorService

- running
- shutting down
- terminated

Tasks submitted and executed: if task submitted, rejected execution handler.

Possible to wait for termination:
- shutdown initiated
  - graceful
  - abrupt

All tasks have been completed.
Shutdown

Lifecycle of a task

- **created**
- **submitted**
- **started**
- **completed**

- can always be cancelled
- may be cancelled (if “open” for interruption)
- cancel: no effect

**abrupt shutdown**

- cancel
cancel

**graceful shutdown**

- submitted and started tasks can complete
java.util.concurrent.ExecutorService

```java
public interface ExecutorService extends Executor {
    // graceful shutdown
    void shutdown();
    // abrupt shutdown
    // -> return list of tasks awaiting execution
    List<Runnable> shutdownNow();

    // query about state change
    boolean isShutdown();
    // ... more methods... discussed later
}
```
public interface ExecutorService extends Executor {
    // block until one event happens
    // (1) all tasks have completed
    // (2) the timeout occurs
    // (3) the current thread is interrupted
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedExecution;

    // Have all tasks been completed? following shut-down
    boolean isTerminated();

    // ... more methods... discussed later
}
public class LifecycleWebServer {
    private final ExecutorService exec = Executors.newCachedThreadPool();
    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() {
                        handleRequest(conn);
                    }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }
    public void stop() { exec.shutdown(); }
}

See complete code at http://www.javaconcurrencyinpractice.com/listings/LifecycleWebServer.java
Executor revisited

```
public interface Executor {
    // Execute the given command at some time in the future
    void execute(Runnable command);
}
```

Runnable as basic task representation

- Cannot return a value
- Cannot throw checked exceptions

Other task abstractions necessary

- Callable: task
- Future: result
Task abstraction Callable

```java
public interface Callable<V> {
    // Task that returns a result and may throw an exception
    V call() throws Exception;
}
```

See Executors for utility factory methods

- Example: wrap a Runnable in a Callable
Lifecycle abstraction with Future

- created
- submitted
- started
- completed

- can always be cancelled
- may be cancelled (if “open” for interruption)
- cancel: no effect

- get()
- blocks
- get()
- result

- cancel(boolean)
- true
isCancelled()
- true
isDone()

Wednesday, January 20, 2010
Future

public interface Future<V> {
    V get()
        throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
    boolean isDone();
    boolean cancel();
    boolean isCancelled();
}

Create a future
- Interface ExecutorService: Future<V> submit([Callable|Runnable])
- Class FutureTask<V>: base implementation of Future<V>