Multithreading

- Thread basics
  - Thread and process
  - Create threads
    - Implementing the Runnable interface
    - Extending the Thread class
    - Extending the Thread Class vs Implementing the Runnable Interface
  - Deadlock
    - Def
    - Conditions
  - Java concurrency APIs
- Counters
- Singleton
- BoundedBlockingQueue
- [Readers/writers lock](#To be finished)
- Thread-safe producer and consumer

Thread basics

Thread and process

- Similar goals: Split up workload into multiple parts and partition tasks into different, multiple tasks for these multiple actors. Two common ways of doing this are multi-threaded programs and multi-process systems.
- Differences

Criteria	Thread	Process
Def	A thread exists within a process and has less resource consumption	A running instance of a program
Resources	Multiple threads within the same process will share the same heap space but each thread still has its own registers and its own stack.	Each process has independent system resources. Inter process mechanism such as pipes, sockets, sockets need to be used to share resources.
Overhead for creation/termination/task switching	Faster due to very little memory copying (just thread stack). Faster because CPU caches and program context can be	Slower because whole process area needs to be copied. Slower because all process area needs to be reloaded

	maintained	
Synchronization overhead	Shared data that is modified requires special handling in the form of locks, mutexes and primitives	No synchronization needed
Use cases	Threads are a useful choice when you have a workload that consists of lightweight tasks (in terms of processing effort or memory size) that come in, for example with a web server servicing page requests. There, each request is small in scope and in memory usage. Threads are also useful in situations where multi-part information is being processed – for example, separating a multi-page TIFF image into separate TIFF files for separate pages. In that situation, being able to load the TIFF into memory once and have multiple threads access the same memory buffer	Processes are a useful choice for parallel programming with workloads where tasks take significant computing power, memory or both. For example, rendering or printing complicated file formats (such as PDF) can sometimes take significant amounts of time – many milliseconds per page – and involve significant memory and I/O requirements. In this situation, using a single-threaded process and using one process per file to process allows for better throughput due to increased independence and isolation between

	leads to performance benefits.	the tasks vs. using one process with multiple threads.

Create threads

Implementing the Runnable interface

- The runnable interface has the following very simple structure

```
public interface Runnable
{
        void run();
}
```

- Steps
  - Create a class which implements the Runnable interface. An object of this class is a Runnable object
  - Create an object of type Thread by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
  - The start() method is invoked on the Thread object created in the previous step.

```
public class RunnableThreadExample implements Runnable
{
        public int count = 0;

        public void run()
        {
                System.out.println( "RunnableThread starting.");

                try
                {
                        while ( count < 5 )
                        {
                                Thread.sleep( 500 );
                                count++;
                        }
```

```
                }
                catch ( InterruptedException exc )
                {
                        System.out.println( "RunnableThread interr
                }

                System.out.println( "RunnableThread terminating" )
        }
}

public static void main( String[] args )
{
        RunnableThreadExample instance = new RunnableThreadExample
        Thread thread = new Thread( instance );
        thread.start();

        /* waits until above thread counts to 5 (slowly) */
        while ( instance.count != 5 )
        {
                try
                {
                        Thread.sleep( 250 );
                }
                catch ( InterruptedException exc )
                {
                        exc.printStackTrace();
                }
        }
}
```

Extending the Thread class

- We can create a thread by extending the Thread class. This will almost always mean that we override the run() method, and the subclass may also call the thread constructor explicitly in its constructor.

```
public class ThreadExample extends Thread
{
        int count = 0;

        public void run()
        {
                System.out.println( "Thread starting" );
```

```
                try
                {
                        while ( count < 5 )
                        {
                                Thread.sleep( 500 );
                                System.out.println( "In thread, co
                                count++;
                        }
                }
                catch ( InterruptedException exc )
                {
                        System.out.println( "Thread interrupted" )
                }

                System.out.println( "Thread terminating" );
        }
}

public static void main( String[] args )
{
        ThreadExample instance = new ThreadExample();
        instance.start();

        while ( instance.count != 5 )
        {
                try
                {
                        Thread.sleep( 250 );
                }
                catch ( InterruptedException exc )
                {
                        exc.printStackTrace();
                }
        }
}
```

Extending the Thread Class vs Implementing the Runnable Interface

- Implementing runnable is the preferrable way.
    - Java does not support multiple inheritance. Therefore, after extending the Thread class, you can't extend any other class which you required. A class implementing the Runnable interface will be able to extend

another class.
    - A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.
- Thread and Runnable are complement to each other for multithreading not competitor or replacement. Because we need both of them for multi-threading.
    - For Multi-threading we need two things:
        - Something that can run inside a Thread (Runnable).
        - Something That can start a new Thread (Thread).
    - So technically and theoretically both of them is necessary to start a thread, one will run and one will make it run (Like Wheel and Engine of motor vehicle).

Deadlock

Def

- A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds. Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever.

Conditions

- **Mutal Exclusion**: Only one process can access a resource at a given time. (Or more accurately, there is limited access to a resource. A deadlock could also occur if a resource has limited quantity.)
- **Hold and Wait**: Processes already holding a resource can request additional resources, without relinquishing their current resources.
- **No Preemption**: One process cannot forcibly remove another process' resource.
- **Circular Wait**: Two or more processes form a circular chain where each process is waiting on another resource in the chain.

Java concurrency APIs

- Thread basics - join, yield, future
- Executor services
- Semaphore/Mutex - locks, synchronized keyword
- Condition variables - wait, notify, condition
- Concurrency collections - CountDownLatch, ConcurrentHashMap, CopyOnWriteArrayList

Counters

- See src dir for details

Singleton

- See src dir for details

BoundedBlockingQueue

- See src dir for details

Readers/writers lock [To be finished]

Thread-safe producer and consumer

- See src dir for details

Delayed scheduler

Interfaces to be implemented

```
public interface Scheduler
{
```

```
        void schedule( Task t, long delayMs );
}

public interface Task
{
        void run();
}
```

Single thread

- Main thread is in Timedwaiting state for delayMs for each call of schedule()
- Only one thread, very low CPU utilization
- Also, this is not working as later call
- How about sleeping in other threads

```
public class SchedulerImpl implements Scheduler
{
        public void schedule( Task t, long delayMs )
        {
                try
                {
                        // sleep for delayMs, and then execute the
                        Thread.sleep( delayMs );
                        t.run();
                }
                catch ( InterruptedException e )
                {
                        // ignore
                }
        }

        public static void main( String[] args )
        {
                Scheduler scheduler = new SchedulerImpl();
                Task t1 = new TaskImpl( 1 );
                Task t2 = new TaskImpl( 2 );

                // main thread in timedwaiting state for 10000 ms
                scheduler.schedule( t1, 10000 );
                scheduler.schedule( t2, 1 );
        }
}
```

One thread for each task

- No blocking when calling schedule
- What happens if we call schedule many times
  - A lot of thread creation overhead
- Call be alleviated by using a thread pool, but still not ideal

```
public class SchedulerImpl implements Scheduler
{
        public void schedule( Task t, long delayMs )
        {
                Thread t = new Thread( new Runnable() {
                        public void run()
                        {
                                try
                                {
                                        Thread.sleep( delayMs );
                                        t.run();
                                }
                                catch ( InterruptedException e )
                                {
                                        // ignore;
                                }
                        }
                } );
                t.start();
        }
}
```

PriorityQueue + A background thread

```
package designThreadSafeEntity.delayedTaskScheduler;

import java.util.PriorityQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class Scheduler
{
```

```
        // order task by time to run
        private PriorityQueue<Task> tasks;

        //
        private final Thread taskRunnerThread;

        // State indicating the scheduler is running
        // Why volatile? As long as main thread stops, runner needs
        private volatile boolean running;

        // Task id to assign to submitted tasks
        // AtomicInteger: Threadsafe. Do not need to add locks when
        // Final: Reference of atomicInteger could not be changed
        private final AtomicInteger taskId;

        public Scheduler()
        {
                tasks = new PriorityQueue<>();
                taskRunnerThread = new Thread( new TaskRunner() );
                running = true;
                taskId = new AtomicInteger( 0 );

                // start task runner thread
                taskRunnerThread.start();
        }

        public void schedule( Task task, long delayMs )
        {
                // Set time to run and assign task id
                long timeToRun = System.currentTimeMillis() + delay
                task.setTimeToRun( timeToRun );
                task.setId( taskId.incrementAndGet() );

                // Put the task in queue
                synchronized ( this )
                {
                        tasks.offer( task );
                        this.notify(); // only a single background
                }
        }

        public void stop( ) throws InterruptedException
        {
                // Notify the task runner as it may be in wait()
                synchronized ( this )
```

```
                {
                        running = false;
                        this.notify();
                }

                // Wait for the task runner to terminate
                taskRunnerThread.join();
        }

        private class TaskRunner implements Runnable
        {
                @Override
                public void run()
                {
                        while ( running )
                        {
                                // Need to synchronize with main t
                                synchronized( Scheduler.this )
                                {
                                        try
                                        {
                                                // task runner is
                                                while ( running &&
                                                {
                                                        Scheduler.
                                                }

                                                // check the first
                                                long now = System.
                                                Task t = tasks.peel

                                                // delay exhausted
                                                if ( t.getTimeToRu
                                                {
                                                        tasks.poll
                                                        t.run();
                                                }
                                                else
                                                {
                                                        // no task
                                                        Scheduler.
                                                }
                                        }
                                        catch ( InterruptedExceptio
                                        {
```

```
                                                Thread.currentThrea
                                        }
                                }
                        }
                }
        }

        public static void main( String[] args ) throws Interrupte
        {
                Scheduler scheduler = new Scheduler();
                scheduler.schedule( new Task(), 1000000 );
                scheduler.schedule( new Task(), 1000 );
                Thread.sleep( 7000 );
                scheduler.stop();
        }
}

class Task implements Comparable<Task>
{
        // When the task will be run
        private long timeToRun;
        private int id;

        public void setId( int id )
        {
                this.id = id;
        }

        public void setTimeToRun( long timeToRun )
        {
                this.timeToRun = timeToRun;
        }

        public void run()
        {
                System.out.println( "Running task " + id );
        }

        public int compareTo( Task other )
        {
                return (int) ( timeToRun - other.getTimeToRun() );
        }

        public long getTimeToRun()
        {
```

```
                return timeToRun;
        }
}
```