Quartz Scheduler Example Programs and Sample Code

Version 2.2.2

October 2015

This document applies to Quartz Scheduler Version 2.2.2 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: QS-EX-222-20151015

Table of Contents

1 The Quartz Example Programs

About the Quartz Example Programs

Welcome to the documentation for the Quartz example programs. Quartz ships with 13 out-of-the-box examples that demonstrate various features of Quartz and the Quartz API.

Where to Find the Examples

All of the examples listed below are part of the Quartz distribution.

The quartz examples reside in the "examples" directory under the main Quartz directory. In the examples directory, you will find a sub-directory for each example.

Each sub-directory includes UNIX/Linux shell scripts for executing the examples as well at Windows batch files. Additionally, every example has a readme.txt file. Please consult the readme file before running the examples.

The source code for the examples is located in the org.quartz.examples package. You will find a sub-package for each example.

The Examples

The following provides an overview of the example programs. Documentation is provided for certain examples, as noted below.

- Example 1 - First Quartz Program: Provides a "Hello World" example for Quartz. See "Example 1 - Your First Quartz Program" on page 7.
- Example 2 - Simple Triggers: Shows a dozen different ways of using Simple Triggers to schedule your jobs.
- Example 3 - Cron Triggers: Shows how Cron Triggers can be used to schedule your job. See "Example 3 - Cron-based Triggers" on page 9.
- Example 4 - Job State and Parameters: Demonstrates how parameters can be passed into jobs and how jobs maintain state. See "Example 4 - Job Parameters and Job State" on page 11.
- Example 5 - Handling Job Misfires: Show how to handle misfires when a job does not execute when it is supposed to. See "Example 5 - Job Misfires" on page 14.
- Example 6 - Dealing with Job Exceptions: Shows how to deal with exceptions that are thrown by your job. See "Example 6 - Dealing with Job Exceptions" on page 17.
- Example 7 - Interrupting Jobs: Shows how the scheduler can interrupt your jobs and how to code your jobs to deal with interruptions.
- Example 8 - Fun with Calendars: Demonstrates how a Holiday calendar can be used to exclude execution of jobs on a holiday.

- Example 9 - Job Listeners: Shows how to use job listeners to have one job trigger another job, building a simple workflow.
- Example 10 - Using Quartz Plug-Ins: Demonstrates the use of the XML Job Initialization plug-in as well as the History Logging plug-ins.
- Example 11 - Quartz Under High Load: Shows how to use thread pools to limit the number of jobs that can execute simultaneously.
- Example 12 - Remote Job Scheduling using RMI: Shows how to use the Remote Method Invocation to have Quartz Scheduler remotely scheduled by a client.
- Example 13 - Clustered Quartz: Demonstrates how Quartz can be used in a clustered environment and how Quartz can use the database to persist scheduling information.
- Example 14 - Trigger Priorities: Demonstrates how Trigger priorities can be used to manage firing order for Triggers with the same fire time. See "Example 14 - Trigger Priorities" on page 19.
- Example 15 - TC Clustered Quartz: Demonstrates how to cluster Quartz using a Terracotta Server Array, rather than with a database.

Example 1 - Your First Quartz Program

This example is designed to demonstrate how to get up and running with Quartz. This example will fire off a simple job that says "Hello World."

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule a job to run at the next even minute
- Wait for 90 seconds to give Quartz a chance to run the job
- Shut down the scheduler

Running the Example

This example can be executed from the examples/example1 directory. There are two ways to run this example

- example1.sh - A UNIX/Linux shell script
- example1.bat - A Windows Batch file

The Code

The code for this example resides in the package org.quartz.examples.example1.

The code in this example is made up of the following classes:

Class Name	Description
SimpleExample	The main program.
HelloJob	A simple job that says Hello World.

HelloJob

HelloJob is a simple job that implements the Job interface and logs a message to the log (by default, this will simply go to the screen). The current date and time is printed in the job so that you can see exactly when the job was execcuted.

```
public void execute(JobExecutionContext context) throws JobExecutionException {
    // Say Hello to the World and display the date/time
    _log.info("Hello World! - " + new Date());
}
```

SimpleExample

The program starts by getting an instance of the Scheduler. This is done by creating a StdSchedulerFactory and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

HelloJob is defined as a job to Quartz using the JobDetail class:

```
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("job1", "group1")
    .build();
```

We create a SimpleTrigger that will fire off at the next round minute:

```
// compute a time that is on the next round minute
Date runTime = evenMinuteDate(new Date());
// Trigger the job to run on the next round minute
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(runTime)
    .build();
```

We now will associate the job to the trigger in the scheduler:

```
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

At this point, the job is scheduled to run when its trigger fires. However, the scheduler is not yet running. So, we must tell the scheduler to start up.

```
sched.start();
```

To let the scheduler have an opportunity to run the job, our program sleeps for 90 seconds. The scheduler is running in the background and should fire off the job during those 90 seconds.

```
Thread.sleep(90L * 1000L);
```

Finally, the program gracefully shuts down the scheduler:

```
sched.shutdown(true);
```

Note: Passing "true" to the shutdown method tells Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Example 3 - Cron-based Triggers

This example is designed to demonstrate how you can use CronTriggers to schedule jobs. This example will fire off several simple jobs that say "Hello World" and display the date and time that the job was executed.

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule several jobs using various features of CronTrigger
- Wait for 300 seconds (5 minutes) to give Quartz a chance to run the jobs
- Shut down the scheduler

Note: Refer to the Quartz Javadoc for a thorough explanation of CronTrigger.

Running the Example

This example can be executed from the examples/example3 directory. There are two ways to run this example

- example3.sh - A UNIX/Linux shell script
- example3.bat - A Windows batch file

The Code

The code for this example resides in the package org.quartz.examples.example3.

The code in this example is made up of the following classes:

Class Name	Description
CronTriggerExample	The main program.
SimpleJob	A simple job that says Hello World and displays the date/time.

SimpleJob

SimpleJob is a simple job that implements the Job interface and logs a message to the log (by default, this will simply go to the screen). The current date and time is printed in the job so that you can see exactly when the job was executed.

```
public void execute(JobExecutionContext context) throws JobExecutionException {
    JobKey jobKey = context.getJobDetail().getKey();
    _log.info("SimpleJob says: " + jobKey + " executing at " + new Date());
}
```

CronTriggerExample

The program starts by getting an instance of the Scheduler. This is done by creating a StdSchedulerFactory and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 20 seconds

```
JobDetail job = newJob(SimpleJob.class)
    .withIdentity("job1", "group1")
    .build();
CronTrigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .withSchedule(cronSchedule("0/20 * * * * ?"))
    .build();
sched.scheduleJob(job, trigger);
```

Job #2 is scheduled to run every other minute starting at 15 seconds past the minute.

```
job = newJob(SimpleJob.class)
    .withIdentity("job2", "group1")
    .build();
trigger = newTrigger()
    .withIdentity("trigger2", "group1")
    .withSchedule(cronSchedule("15 0/2 * * * ?"))
    .build();
sched.scheduleJob(job, trigger);
```

Job #3 is scheduled to run every other minute between 8am and 5pm (17 :00).

```
job = newJob(SimpleJob.class)
    .withIdentity("job3", "group1")
    .build();
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 0/2 8-17 * * ?"))
    .build();
sched.scheduleJob(job, trigger);
```

Job #4 is scheduled to run every three minutes, but only between 5pm and 11pm

```
job = newJob(SimpleJob.class)
    .withIdentity("job4", "group1")
    .build();
trigger = newTrigger()
    .withIdentity("trigger4", "group1")
    .withSchedule(cronSchedule("0 0/3 17-23 * * ?"))
    .build();
sched.scheduleJob(job, trigger);
```

Job #5 is scheduled to run at 10am on the 1st and 15th days of the month

```
job = newJob(SimpleJob.class)
    .withIdentity("job5", "group1")
    .build();
trigger = newTrigger()
    .withIdentity("trigger5", "group1")
```

```
    .withSchedule(cronSchedule("0 0 10am 1,15 * ?"))
    .build();
sched.scheduleJob(job, trigger);
```

Job #6 is scheduled to run every 30 seconds on weekdays (Monday through Friday)

```
job = newJob(SimpleJob.class)
    .withIdentity("job6", "group1")
    .build();
trigger = newTrigger()
    .withIdentity("trigger6", "group1")
    .withSchedule(cronSchedule("0,30 * * ? * MON-FRI"))
    .build();
sched.scheduleJob(job, trigger);
```

Job #7 is scheduled to run every 30 seconds on weekends (Saturday and Sunday)

```
job = newJob(SimpleJob.class)
    .withIdentity("job7", "group1")
    .build();
trigger = newTrigger()
    .withIdentity("trigger7", "group1")
    .withSchedule(cronSchedule("0,30 * * ? * SAT,SUN"))
    .build();
sched.scheduleJob(job, trigger);
```

The scheduler is then started (it also would have been fine to start it before scheduling the jobs).

```
sched.start();
```

To let the scheduler have an opportunity to run the job, our program sleeps for five minutes (300 seconds). The scheduler is running in the background and should fire off several jobs during that time.

Note: Because many of the jobs have hourly and daily restrictions on them, not all of the jobs will run in this example. For example: Job #6 only runs on weekdays while Job #7 only runs on weekends.

```
Thread.sleep(300L * 1000L);
```

Finally, the program gracefully shuts down the scheduler:

```
sched.shutdown(true);
```

Note: Passing "true" into the shutdown() method tells the Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Example 4 - Job Parameters and Job State

This example is designed to demonstrate how you can pass run-time parameters into Quartz jobs and how you can maintain state in a job.

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule two jobs, each job will execute every ten seconds for a specified total of times

- The scheduler will pass a run-time job parameter of "Green" to the first job instance
- The scheduler will pass a run-time job parameter of "Red" to the second job instance
- The program will wait 60 seconds so that the two jobs have plenty of time to run
- Shut down the scheduler

Running the Example

This example can be executed from the examples/example4 directory. There are two ways to run this example

- example4.sh - A UNIX/Linux shell script
- example4.bat - A Windows Batch file

The Code

The code for this example resides in the package org.quartz.examples.example4.

The code in this example is made up of the following classes:

Class Name	Description
JobStateExample	The main program.
ColorJob	A simple job that prints a favorite color (passed in as a run-time parameter) and displays its execution count..

ColorJob

ColorJob is a simple class that implement the Job interface and is annotated as shown below:

```
@PersistJobDataAfterExecution
@DisallowConcurrentExecution
public class ColorJob implements Job {
```

The annotations cause behavior as their names describe. Multiple instances of the job will not be allowed to run concurrently (consider a case where a job has code in its execute() method that takes 34 seconds to run, but it is scheduled with a trigger that repeats every 30 seconds), and will have its JobDataMap contents re-persisted in the scheduler's JobStore after each execution. For the purposes of this example, only `@PersistJobDataAfterExecution` annotation is truly relevant, but it's always wise to use the `@DisallowConcurrentExecution` annotation with it, to prevent race-conditions on saved data.

ColorJob logs the following information when the job is executed:

- The job's identification key (name and group) and time/date of execution
- The job's favorite color (which is passed in as a run-time parameter)

- The job's execution count calculated from a member variable
- The job's execution count maintained as a job map parameter

```
_log.info("ColorJob: " + jobKey + " executing at " + new Date() + "\n" +
    "  favorite color is " + favoriteColor + "\n" +
    "  execution count (from job map) is " + count + "\n" +
    "  execution count (from job member variable) is " + _counter);
```

The variable *favoriteColor* is passed in as a job parameter. It is retrieved as follows from the JobDataMap:

```
JobDataMap data = context.getJobDetail().getJobDataMap();
String favoriteColor = data.getString(FAVORITE_COLOR);
```

The variable *count* is stored in the job data map as well:

```
JobDataMap data = context.getJobDetail().getJobDataMap();
int count = data.getInt(EXECUTION_COUNT);
```

The variable is later incremented and stored back into the job data map so that job state can be preserved:

```
count++;
data.put(EXECUTION_COUNT, count);
```

There is also a member variable named *counter* . This variable is defined as a member variable to the class:

```
private int _counter = 1;
```

This variable is also incremented and displayed. However, its count will always be displayed as "1" because Quartz will always instantiate a new instance of the class during each execution. This prevents member variables from being used to maintain state.

JobStateExample

The program starts by getting an instance of the Scheduler. This is done by creating a StdSchedulerFactory and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 10 seconds, for a total of five times:

```
JobDetail job1 = newJob(ColorJob.class)
    .withIdentity("job1", "group1")
    .build();
SimpleTrigger trigger1 = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(10)
            .withRepeatCount(4))
    .build();
```

Job #1 is passed in two job parameters. One is a favorite color, with a value of "Green". The other is an execution count, which is initialized with a value of 1.

```
job1.getJobDataMap().put(ColorJob.FAVORITE_COLOR, "Green");
job1.getJobDataMap().put(ColorJob.EXECUTION_COUNT, 1);
```

Job #2 is also scheduled to run every 10 seconds, for a total of five times:

```
JobDetail job2 = newJob(ColorJob.class)
    .withIdentity("job2", "group1")
    .build();
SimpleTrigger trigger2 = newTrigger()
    .withIdentity("trigger2", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(10)
            .withRepeatCount(4))
    .build();
```

Job #2 is also passed in two job parameters. One is a favorite color, with a value of "Red". The other is an execution count, which is initialized with a value of 1.

```
job2.getJobDataMap().put(ColorJob.FAVORITE_COLOR, "Red");
job2.getJobDataMap().put(ColorJob.EXECUTION_COUNT, 1);
```

The scheduler is then started.

```
sched.start();
```

To let the scheduler have an opportunity to run the job, our program will sleep for one minute (60 seconds)

```
Thread.sleep(60L * 1000L);
```

Finally, the program gracefully shuts down the scheduler:

```
sched.shutdown(true);
```

Note: Passing "true" to the shutdown method tells Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Example 5 - Job Misfires

This example is designed to demonstrate concepts related to trigger misfires.

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule two jobs, each job will execute every three seconds for an indefinite length of time
- The jobs will take ten seconds to run (preventing the execution trigger from firing every three seconds)
- Each job has different misfire instructions
- The program will wait 10 minutes so that the two jobs have plenty of time to run
- Shut down the scheduler

Running the Example

This example can be executed from the examples/example5 directory. There are two ways to run this example

- example5.sh - A UNIX/Linux shell script
- example5.bat - A Windows Batch file

The Code

The code for this example resides in the package org.quartz.examples.example5.

The code in this example is made up of the following classes:

Class Name	Description
MisfireExample	The main program.
StatefulDumbJob	A simple job class whose execute method takes 10 seconds to run.

StatefulDumbJob

StatefulDumbJob is a simple job that prints its execution time and then waits for a period of time before completing. The amount of wait time is defined by the job parameter `EXECUTION_DELAY`. If this job parameter is not passed in, the job will default to a wait time of 5 seconds. The job also keeps its own count of how many times it has executed using a value in its JobDataMap called `NUM_EXECUTIONS`. Because the class has the `PersistJobDataAfterExecution` annotation, the execution count is preserved between each execution.

```
@PersistJobDataAfterExecution
@DisallowConcurrentExecution
public class StatefulDumbJob implements Job {
    public static final String NUM_EXECUTIONS = "NumExecutions";
    public static final String EXECUTION_DELAY = "ExecutionDelay";
    public StatefulDumbJob() {
    }
    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        System.err.println("---" + context.getJobDetail().getKey()
                + " executing.[" + new Date() + "]");
        JobDataMap map = context.getJobDetail().getJobDataMap();
        int executeCount = 0;
        if (map.containsKey(NUM_EXECUTIONS)) {
            executeCount = map.getInt(NUM_EXECUTIONS);
        }
        executeCount++;
        map.put(NUM_EXECUTIONS, executeCount);
        long delay = 5000l;
        if (map.containsKey(EXECUTION_DELAY)) {
            delay = map.getLong(EXECUTION_DELAY);
        }
        try {
```

```
            Thread.sleep(delay);
        } catch (Exception ignore) {
        }
        System.err.println("  -" + context.getJobDetail().getKey()
                + " complete (" + executeCount + ").");
    }
}
```

MisfireExample

The program starts by getting an instance of the Scheduler. This is done by creating a StdSchedulerFactory and then using it to create a scheduler. This will create a simple, RAM-based scheduler, because no quartz.properties telling it to do otherwise are provided.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 3 seconds indefinitely. An execution delay of 10 seconds is passed into the job:

```
JobDetail job = newJob(StatefulDumbJob.class)
    .withIdentity("statefulJob1", "group1")
    .usingJobData(StatefulDumbJob.EXECUTION_DELAY, 10000L)
    .build();
SimpleTrigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(3)
            .repeatForever())
    .build();
sched.scheduleJob(job, trigger);
```

Job #2 is scheduled to run every 3 seconds indefinitely. An execution delay of 10 seconds is passed into the job:

```
job = newJob(StatefulDumbJob.class)
            .withIdentity("statefulJob2", "group1")
            .usingJobData(StatefulDumbJob.EXECUTION_DELAY, 10000L)
            .build();
        trigger = newTrigger()
            .withIdentity("trigger2", "group1")
            .startAt(startTime)
            .withSchedule(simpleSchedule()
                    .withIntervalInSeconds(3)
                    .repeatForever()
                    .withMisfireHandlingInstructionNowWithExistingCount()) // set
                      // misfire instruction
            .build();
```

Note: The trigger for job #2 is set with a misfire instruction that will cause it to reschedule with the existing repeat count. This policy forces Quartz to refire the trigger as soon as possible. Job #1 uses the default "smart" misfire policy for simple triggers, which causes the trigger to fire at it's next normal execution time.

The scheduler is then started.

```
sched.start();
```

To let the scheduler have an opportunity to run the job, our program sleeps for ten minutes (600 seconds)

```
Thread.sleep(600L * 1000L);
```

Finally, the program gracefully shuts down the scheduler:

```
sched.shutdown(true);
```

Note: Passing "true" to the shutdown method tells Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Example 6 - Dealing with Job Exceptions

This example is designed to demonstrate how to deal with job execution exceptions. Jobs in Quartz are permitted to throw a JobExecutionExceptions. When this exception is thrown, you can direct Quartz to take a specified action.

The program will perform the following actions:

- Start up the Quartz Scheduler.
- Schedule two jobs. Each job will execute every three seconds for an indefinite period of time.
- The jobs will throw exceptions and Quartz will take appropriate action.
- The program will wait 60 seconds so that the two jobs have plenty of time to run.
- Shut down the scheduler.

Running the Example

This example can be executed from the examples/example6 directory. There are two ways to run this example

- example6.sh - A UNIX/Linux shell script
- example6.bat - A Windows Batch file

The Code

The code for this example resides in the package org.quartz.examples.example6.

The code in this example is made up of the following classes:

Class Name	Description
JobExceptionExample	The main program.
BadJob1	A simple job that will throw an exception and instruct Quartz to refire its trigger immediately.

Class Name	Description
BadJob2	A simple job that will throw an exception and instruct Quartz to never schedule the job again.

BadJob1

BadJob1 is a simple job that creates an artificial exception (divide by zero). When this exception occurs, a JobExecutionException is thrown and set to refire the job immediately.

```
try {
    int zero = 0;
    int calculation = 4815 / zero;
}
catch (Exception e) {
    _log.info("--- Error in job!");
    JobExecutionException e2 =
        new JobExecutionException(e);
    // this job will refire immediately
    e2.refireImmediately();
    throw e2;
}
```

This will force Quartz to run this job over and over again.

BadJob2

BadJob2 is a simple job that creates an artificial exception (divide by zero). When this exception occurs, a JobExecutionException is thrown and set to ensure that Quartz never runs the job again.

```
try {
    int zero = 0;
    int calculation = 4815 / zero;
}
catch (Exception e) {
    _log.info("--- Error in job!");
    JobExecutionException e2 =
        new JobExecutionException(e);
    // Quartz will automatically unschedule
    // all triggers associated with this job
    // so that it does not run again
    e2.setUnscheduleAllTriggers(true);
    throw e2;
}
```

This will force Quartz to shutdown this job so it does not run again.

JobExceptionExample

The program starts by getting an instance of the Scheduler. This is done by creating a StdSchedulerFactory and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 3 seconds indefinitely. This job will fire BadJob1.

```
JobDetail job = newJob(BadJob1.class)
    .withIdentity("badJob1", "group1")
    .build();
SimpleTrigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(3)
            .repeatForever())
    .build();
Date ft = sched.scheduleJob(job, trigger);
```

Job #2 is scheduled to run every 3 seconds indefinitely. This job will fire BadJob2.

```
job = newJob(BadJob2.class)
    .withIdentity("badJob2", "group1")
    .build();
trigger = newTrigger()
    .withIdentity("trigger2", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(3)
            .repeatForever())
    .build();
ft = sched.scheduleJob(job, trigger);
```

The scheduler is then started.

```
sched.start();
```

To let the scheduler have an opportunity to run the job, our program will sleep for 1 minute (60 seconds)

```
Thread.sleep(60L * 1000L);
```

The scheduler will run both jobs (BadJob1 and BadJob2). Each job will throw an exception. Job 1 will attempt to refire immediately. Job 2 will never run again.

Finally, the program gracefully shuts down the scheduler:

```
sched.shutdown(true);
```

Note: Passing "true" to the shutdown method tells Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Example 14 - Trigger Priorities

This example will demonstrate how you can use trigger priorities to manage firing order for triggers with the same fire time.

The program will perform the following actions:

- Create a scheduler with a single worker thread
- Schedule three triggers with different priorities that fire the first time at the same time, and a second time at staggered intervals

- Start up the Quartz Scheduler
- Wait for 30 seconds to give Quartz a chance to fire the triggers
- Shut down the scheduler

Running the Example

This example can be executed from the examples/example14 directory. There are two ways to run this example

- example14.sh - A UNIX/Linux shell script
- example14.bat - A Windows Batch file

Expected Results

Each of the three triggers should fire twice. Once in order of priority, as they all start at the same time, and a second time in order of their staggered firing times. You should see something like this in the log or on the console:

```
INFO 15 Aug 12:15:51.345 PM PriorityExampleScheduler_Worker-0
     org.quartz.examples.example14.TriggerEchoJob
TRIGGER: Priority10Trigger15SecondRepeat
INFO 15 Aug 12:15:51.345 PM PriorityExampleScheduler_Worker-0
     org.quartz.examples.example14.TriggerEchoJob
TRIGGER: Priority5Trigger10SecondRepeat
INFO 15 Aug 12:15:51.345 PM PriorityExampleScheduler_Worker-0
     org.quartz.examples.example14.TriggerEchoJob
TRIGGER: PriorityNeg5Trigger5SecondRepeat
INFO 15 Aug 12:15:56.220 PM PriorityExampleScheduler_Worker-0
     org.quartz.examples.example14.TriggerEchoJob
TRIGGER: PriorityNeg5Trigger5SecondRepeat
INFO 15 Aug 12:16:01.220 PM PriorityExampleScheduler_Worker-0
     org.quartz.examples.example14.TriggerEchoJob
TRIGGER: Priority5Trigger10SecondRepeat
INFO 15 Aug 12:16:06.220 PM PriorityExampleScheduler_Worker-0
     org.quartz.examples.example14.TriggerEchoJob
TRIGGER: Priority10Trigger15SecondRepeat
```

The Code

The code for this example resides in the package org.quartz.examples.example14.

The code in this example is made up of the following classes:

Class Name	Description
PriorityExample	The main program.
TriggerEchoJob	A simple job that echoes the name if the Trigger that fired it.

TriggerEchoJob

TriggerEchoJob is a simple job that implements the Job interface and writes the name of the trigger that fired it to the log (by default, this will simply go to the screen):

```
public void execute(JobExecutionContext context) throws JobExecutionException {
    LOG.info("TRIGGER: " + context.getTrigger().getKey());
}
```

PriorityExample

The program starts by getting an instance of the Scheduler. This is done by creating a StdSchedulerFactory and then using it to create a scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory(
    "org/quartz/examples/example14/quartz_priority.properties");
Scheduler sched = sf.getScheduler();
```

We pass a specific Quartz properties file to the StdSchedulerFactory to configure our new Scheduler instance. These properties will create a RAM-based scheduler with only one worker thread so that we can see priorities act as the tie breaker when triggers compete for the single thread, quartz_priority.properties:

```
org.quartz.scheduler.instanceName=PriorityExampleScheduler
# Set thread count to 1 to force Triggers scheduled for the same time to
# to be ordered by priority.
org.quartz.threadPool.threadCount=1
org.quartz.threadPool.class=org.quartz.simpl.SimpleThreadPool
org.quartz.jobStore.class=org.quartz.simpl.RAMJobStore
```

The TriggerEchoJob is defined as a job to Quartz using the JobDetail class. As shown below, it passes "null" for its group, so it will use the default group:

```
JobDetail job = new JobDetail("TriggerEchoJob", null, TriggerEchoJob.class);
```

We create three SimpleTriggers that will all fire the first time five seconds from now but with different priorities, and then fire a second time at staggered five second intervals:

```
// Calculate the start time of all triggers as 5 seconds from now
Date startTime = futureDate(5, IntervalUnit.SECOND);
// First trigger has priority of 1, and will repeat after 5 seconds
Trigger trigger1 = newTrigger()
    .withIdentity("PriorityNeg5Trigger5SecondRepeat")
    .startAt(startTime)
    .withSchedule(simpleSchedule().withRepeatCount(1).withIntervalInSeconds(5))
    .withPriority(1)
    .forJob(job)
    .build();
// Second trigger has default priority of 5 (default), and will repeat after 10 seconds
Trigger trigger2 = newTrigger()
    .withIdentity("Priority5Trigger10SecondRepeat")
    .startAt(startTime)
    .withSchedule(simpleSchedule().withRepeatCount(1).withIntervalInSeconds(10))
    .forJob(job)
    .build();
// Third trigger has priority 10, and will repeat after 15 seconds
Trigger trigger3 = newTrigger()
    .withIdentity("Priority10Trigger15SecondRepeat")
    .startAt(startTime)
    .withSchedule(simpleSchedule().withRepeatCount(1).withIntervalInSeconds(15))
    .withPriority(10)
    .forJob(job)
```

```
    .build();
```

We now associate the three triggers with our job in the scheduler. The first time we need to also add the job itself to the scheduler:

```
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger1);
sched.scheduleJob(trigger2);
sched.scheduleJob(trigger3);
```

At this point, the triggers have been scheduled to run. However, the scheduler is not yet running, so we must tell it to start up.

```
sched.start();
```

To let the scheduler have an opportunity to run the job, our program sleeps for 30 seconds. The scheduler is running in the background and should fire the job six times during those 30 seconds.

```
Thread.sleep(30L * 1000L);
```

Finally, the program gracefully shuts down the scheduler:

```
sched.shutdown(true);
```

Note: Passing "true" to the shutdown method tells Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

2 Quartz Code Snippets

About the Code Snippets

The Quartz code snippets are a collection of code examples that show you how to do specific things with Quartz.

The snippets assume you have used static imports of Quartz's DSL classes such as these:

```
import static org.quartz.JobBuilder.*;
import static org.quartz.TriggerBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.CalendarIntervalScheduleBuilder.*;
import static org.quartz.JobKey.*;
import static org.quartz.TriggerKey.*;
import static org.quartz.DateBuilder.*;
import static org.quartz.impl.matchers.KeyMatcher.*;
import static org.quartz.impl.matchers.GroupMatcher.*;
import static org.quartz.impl.matchers.AndMatcher.*;
import static org.quartz.impl.matchers.OrMatcher.*;
import static org.quartz.impl.matchers.EverythingMatcher.*;
```

How-To: Use Multiple (Non-Clustered) Schedulers

Reasons you might want to do this:

- For managing resources, for example, if you have a mix of light-weight and heavy-weight jobs, then you might want to have a scheduler with many threads to service the lightweight jobs and one with few threads to service the heavy-weight jobs. Doing this would keep your machine’s resources from being overwhelmed by running too many heavy-weight jobs concurrently.
- To schedule jobs in one application and have them execute within another (when using JDBC-JobStore).

Note that you can create as many schedulers as you like within any application, but they must have unique scheduler names (typically defined in the quartz.properties file). This means that you'll need to have multiple properties files, which means that you'll need to specify them as you initialize the StdSchedulerFactory (as it only defaults to finding "quartz.properties").

If you run multiple schedulers they can, of course, each have distinct characteristics. For example, one might use RAMJobStore and have 100 worker threads, and another might use JDBC-JobStore and have 20 worker threads.

Important: Never start (scheduler.start()) a non-clustered instance against the same set of database tables that any other instance with the same scheduler name is running (start()ed) against. Doing so will cause data corruption and erratic behavior.

Scheduling Jobs from One Application to be Executed in Another Application

Note: The following information applies to JDBC-JobStore. You might also want to look at RMI or JMX features to control a scheduler in a remote process, which works for any JobStore. Additionally, you might want to look at the Terracotta Quartz Where features.

Currently, if you want particular jobs to run in a particular scheduler, then it needs to be a distinct scheduler, unless you use the Terracotta Quartz Where features.

Suppose you have an application "App A" that needs to schedule jobs (based on user input) that need to run either on the local process/machine "Machine A" (for simple jobs) or on a remote machine "Machine B" (for complex jobs).

It is possible within an application to instantiate two (or more) schedulers, and schedule jobs into both (or more) schedulers, and have only the jobs placed into one scheduler run on the local machine. This is achieved by calling scheduler.start() on the scheduler(s) within the process where you want the jobs to execute. Scheduler.start() causes the scheduler instance to start processing the jobs (that is, start waiting for trigger-fire times to arrive, and then executing the jobs). However a non-started scheduler instance can still be used to schedule (and retrieve) jobs.

For example:

- In "App A" create "Scheduler A" (with config that points it at database tables prefixed with "A"), and invoke start() on "Scheduler A". Now "Scheduler A" in "App A" will execute jobs scheduled by "Scheduler A" in "App A."
- In "App A" create "Scheduler B" (with config that points it at database tables prefixed with "B"), and DO NOT invoke start() on "Scheduler B". Now "Scheduler B" in "App A" can schedule jobs to run where "Scheduler B" is started.
- In "App B" create "Scheduler B" (with config that points it at database tables prefixed with "B"), and invoke start() on "Scheduler B". Now "Scheduler B" in "App B" will execute jobs scheduled by "Scheduler B" in "App A".

How-To: Define a Job (with input data)

A Job Class

```
public class PrintPropsJob implements Job {
    public PrintPropsJob() {
        // Instances of Job must have a public no-argument constructor.
    }
    public void execute(JobExecutionContext context)
            throws JobExecutionException {
        JobDataMap data = context.getMergedJobDataMap();
        System.out.println("someProp = " + data.getString("someProp"));
    }
}
```

Defining a Job Instance

```
// Define job instance
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .usingJobData("someProp", "someValue")
    .build();
```

Note that if your Job class contains setter methods that match your JobDataMap keys (for example. "setSomeProp" in the example above), and you use the default JobFactory implementation, Quartz will automatically call the setter method with the JobDataMap value. Thus, there is no need to include code in the Job's execute method to retrieve the value from the JobDataMap.

How-To: Schedule a Job

Scheduling a Job

```
// Define job instance
JobDetail job1 = newJob(ColorJob.class)
    .withIdentity("job1", "group1")
    .build();
// Define a Trigger that will fire "now", and not repeat
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startNow()
    .build();
// Schedule the job with the trigger
sched.scheduleJob(job, trigger);
```

How-To: Unschedule a Job

Unscheduling a Particular Trigger from a Job

```
// Unschedule a particular trigger from the job (a job may have more than one trigger)
scheduler.unscheduleJob(triggerKey("trigger1", "group1"));
```

Deleting a Job and Unscheduling all of its Triggers

```
// Schedule the job with the trigger
scheduler.deleteJob(jobKey("job1", "group1"));
```

How-To: Store a Job for Later Use

Storing a Job

```
// Define a durable job instance (durable jobs can exist without triggers)
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .storeDurably()
    .build();
// Add the the job to the scheduler's store
sched.addJob(job, false);
```

How-To: Schedule an Already Stored Job

Scheduling an Already Stored Job

```
// Define a Trigger that will fire "now" and associate it with the existing job
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startNow()
    .forJob(jobKey("job1", "group1"))
    .build();
// Schedule the trigger
sched.scheduleJob(trigger);
```

How-To: Update an Existing Job

Update an Existing Job

```
// Add the new job to the scheduler, instructing it to "replace"
//  the existing job with the given name and group (if any)
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .build();
// store, and set overwrite flag to 'true'
scheduler.addJob(job1, true);
```

How-To: Update a Trigger

Replacing a Trigger

```
// Define a new Trigger
Trigger trigger = newTrigger()
    .withIdentity("newTrigger", "group1")
    .startNow()
    .build();
// tell the scheduler to remove the old trigger with the given key, and
// put the new one in its place
sched.rescheduleJob(triggerKey("oldTrigger", "group1"), trigger);
```

Updating an Existing Trigger

```
// retrieve the trigger
Trigger oldTrigger = sched.getTrigger(triggerKey("oldTrigger", "group1");
// obtain a builder that would produce the trigger
TriggerBuilder tb = oldTrigger.getTriggerBuilder();
// update the schedule associated with the builder, and build the new trigger
// (other builder methods could be called, to change the trigger in any
// desired way)
Trigger newTrigger = tb.withSchedule(simpleSchedule()
    .withIntervalInSeconds(10)
    .withRepeatCount(10)
    .build();
sched.rescheduleJob(oldTrigger.getKey(), newTrigger);
```

How-To: Initialize Job Data with Scheduler Initialization

You can initialize the Scheduler with predefined jobs and triggers using the XMLSchedulingDataProcessorPlugin. An example is provided in the Quartz distribution in the directory examples/example10. However, the following is a short description of how the plugin works.

First of all, you must explicitly specify that you want to use the XMLSchedulingDataProcessorPlugin. This is an excerpt from an example quartz.properties:

```
#============================================================
# Configure the Job Initialization Plugin
#============================================================
org.quartz.plugin.jobInitializer.class =
org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin
org.quartz.plugin.jobInitializer.fileNames = jobs.xml
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.scanInterval = 10
org.quartz.plugin.jobInitializer.wrapInUserTransaction = false
```

The following describes what each property does:

- `fileNames` is a comma separated list of filenames (with paths). These files contain the XML definition of jobs and associated triggers. An example jobs.xml definition file is shown later in this topic.
- `failOnFileNotFound` specifies whether the plugin should thrown an exception if the XML definition files are not found, thus preventing it (the plugin) from initializing?
- `scanInterval` specifies whether the XML definition files are to be reloaded if a change is detected in one of the files. This property specifies the interval (in seconds) at which the files are checked for changes. Set to 0 to disable reloading.
- `wrapInUserTransaction` must be set to true when using the XMLSchedulingDataProcessorPlugin with JobStoreCMT. Otherwise, you might experience erratic behavior.

The jobs.xml file (or any other name you assign to this file in the `fileNames` property) declaratively defines jobs and triggers. It can also contain directives to delete existing data. Here's a self-explanatory example:

```
<?xml version='1.0' encoding='utf-8'?>
<job-scheduling-data
  xmlns="http://www.quartz-scheduler.org/xml/JobSchedulingData"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.quartz-scheduler.org/xml/JobSchedulingData
  http://www.quartz-scheduler.org/xml/job_scheduling_data_1_8.xsd"
  version="1.8">
    <schedule>
        <job>
            <name>my-very-clever-job</name>
            <group>MYJOB_GROUP</group>
            <description>The job description</description>
            <job-class>com.acme.scheduler.job.CleverJob</job-class>
```

```
            <job-data-map allows-transient-data="false">
                <entry>
                    <key>burger-type</key>
                    <value>hotdog</value>
                </entry>
                <entry>
                    <key>dressing-list</key>
                    <value>ketchup,mayo</value>
                </entry>
            </job-data-map>
        </job>
        <trigger>
            <cron>
                <name>my-trigger</name>
                <group>MYTRIGGER_GROUP</group>
                <job-name>my-very-clever-job</job-name>
                <job-group>MYJOB_GROUP</job-group>
                <!-- trigger every night at 4:30 am -->
                <!-- do not forget to light the kitchen's light -->
                <cron-expression>0 30 4 * * ?</cron-expression>
            </cron>
        </trigger>
    </schedule>
</job-scheduling-data>
```

You can find another example of the jobs.xml file in the examples/example10 directory of your Quartz distribution.

Checkout the XML schema at http://www.quartz-scheduler.org/xml/job_scheduling_data_2_0.xsd for full details of what is possible.

How-To: List Jobs in the Scheduler

Listing all Jobs in the scheduler

```
// enumerate each job group
for(String group: sched.getJobGroupNames()) {
    // enumerate each job in group
    for(JobKey jobKey : sched.getJobKeys(groupEquals(group))) {
        System.out.println("Found job identified by: " + jobKey);
    }
}
```

How-To: List Triggers in Scheduler

Listing all Triggers in the Scheduler

```
// enumerate each trigger group
for(String group: sched.getTriggerGroupNames()) {
    // enumerate each trigger in group
    for(TriggerKey triggerKey : sched.getTriggerKeys(groupEquals(group))) {
        System.out.println("Found trigger identified by: " + triggerKey);
    }
}
```

How-To: Find Triggers of a Job

Finding Triggers of a Job

```
List<Trigger> jobTriggers = sched.getTriggersOfJob(jobKey("jobName", "jobGroup"));
```

How-To: Use Job Listeners

Creating a JobListener

Implement the JobListener interface.

```
package foo;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.JobListener;
public class MyJobListener implements JobListener {
    private String name;
    public MyJobListener(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void jobToBeExecuted(JobExecutionContext context) {
        // do something with the event
    }
    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {
        // do something with the event
    }
    public void jobExecutionVetoed(JobExecutionContext context) {
        // do something with the event
    }
}
```

- OR -

Extend JobListenerSupport.

```
package foo;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.listeners.JobListenerSupport;
public class MyOtherJobListener extends JobListenerSupport {
    private String name;
    public MyOtherJobListener(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
        @Override
    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {
        // do something with the event
    }
}
```

Registering a JobListener with the Scheduler to Listen to All Jobs

```
scheduler.getListenerManager().addJobListener(myJobListener,
  allJobs());
```

Registering a JobListener with the Scheduler to Listen to A Specific Job

```
scheduler.getListenerManager().addJobListener(myJobListener,
  jobKeyEquals(jobKey("myJobName", "myJobGroup")));
```

Registering a JobListener with the Scheduler to Listen to all Jobs in a Group

```
scheduler.getListenerManager().addJobListener(myJobListener,
  jobGroupEquals("myJobGroup"));
```

How-To: Use Trigger Listeners

Creating a TriggerListener

Implement the TriggerListener interface.

```
package foo;
import org.quartz.JobExecutionContext;
import org.quartz.Trigger;
import org.quartz.TriggerListener;
import org.quartz.Trigger.CompletedExecutionInstruction;
public class MyTriggerListener implements TriggerListener {
    private String name;
    public MyTriggerListener(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void triggerComplete(Trigger trigger, JobExecutionContext context,
            CompletedExecutionInstruction triggerInstructionCode) {
        // do something with the event
    }
    public void triggerFired(Trigger trigger, JobExecutionContext context) {
        // do something with the event
    }
    public void triggerMisfired(Trigger trigger) {
        // do something with the event
    }
    public boolean vetoJobExecution(Trigger trigger, JobExecutionContext context) {
        // do something with the event
        return false;
    }
}
```

- OR -

Extend TriggerListenerSupport.

```
package foo;
import org.quartz.JobExecutionContext;
import org.quartz.Trigger;
import org.quartz.listeners.TriggerListenerSupport;
public class MyOtherTriggerListener extends TriggerListenerSupport {
    private String name;
```

```
    public MyOtherTriggerListener(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    @Override
    public void triggerFired(Trigger trigger, JobExecutionContext context) {
        // do something with the event
    }
}
```

Registering a TriggerListener with the Scheduler to Listen to all Triggers

```
scheduler.getListenerManager().addTriggerListener(
  myTriggerListener, allTriggers());
```

Registering a TriggerListener with the Scheduler to Listen to a Specific Trigger

```
scheduler.getListenerManager().addTriggerListener(
  myTriggerListener, triggerKeyEquals(triggerKey("myTriggerName",
  "myTriggerGroup")));
```

Registering a TriggerListener with the Scheduler to Listen to all Triggers In a Group

```
scheduler.getListenerManager().addTriggerListener(
  myTriggerListener, triggerGroupEquals("myTriggerGroup"));
```

How-To: Use Scheduler Listeners

Creating a SchedulerListener

Extend TriggerListenerSupport and override methods for the events in which you are interested.

```
package foo;
import org.quartz.Trigger;
import org.quartz.listeners.SchedulerListenerSupport;
public class MyOtherSchedulerListener extends SchedulerListenerSupport {
    @Override
    public void schedulerStarted() {
        // do something with the event
    }
    @Override
    public void schedulerShutdown() {
        // do something with the event
    }
    @Override
    public void jobScheduled(Trigger trigger) {
        // do something with the event
    }
}
```

Registering a SchedulerListener with the Scheduler

```
scheduler.getListenerManager().addSchedulerListener(mySchedListener);
```

How-To: Create a Trigger that Executes Every Ten Seconds

Using SimpleTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(10)
            .repeatForever())
    .build();
```

How-To: Create a Trigger That Executes Every 90 minutes

Using SimpleTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
            .withIntervalInMinutes(90)
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInMinutes(90))
    .build();
```

How-To: Create a Trigger that Executes Every Day

If you want a trigger that always fires at a certain time of day, use CronTrigger or CalendarIntervalTrigger, because these triggers will preserve the firing time across daylight savings time changes.

Using CronTrigger

Create a CronTrigger. that executes every day at 3:00 PM:

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(dailyAtHourAndMinute(15, 0))
    // fire every day at 15:00
    .build();
```

Using SimpleTrigger

Create a SimpleTrigger that executes 3:00 PM tomorrow, and then every 24 hours after that.

Note: Be aware that this type of trigger might not always fire at 3:00 PM, because adding 24 hours on days when daylight savings time shifts can result in an execution time of 2:00 PM or 4:00 PM, depending upon whether the 3:00 PM time was started during DST or standard time.

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(24)
            // interval is actually set at 24 hours' worth of milliseconds
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInDays(1))
           // interval is set in calendar days
    .build();
```

How-To: Create a Trigger that Executes Every 2 Days

For this kind of schedule, you might be tempted to use a CronTrigger. However, if this is truly to be every two days, a CronTrigger will not work. To illustrate this, simply think of how many days are in a typical month (28-31). A cron expression like "0 0 5 2/2 * ?" would give us a trigger that would restart its count at the beginning of every month. This means that we would get subsequent firings on July 30 and August 2, which is an interval of three days, not two.

Likewise, an expression like "0 0 5 1/2 * ?" would end up firing on July 31 and August 1, just one day apart.

Therefore, for this schedule, using a SimpleTrigger or CalendarIntervalTrigger makes sense:

Using SimpleTrigger

Create a SimpleTrigger that executes 3:00 PM tomorrow, and then every 48 hours after that.

Note: Be aware that this trigger might not always fire at 3:00 PM, because adding (2 * 24) hours on days when daylight savings time shifts can result in an

execution time of 2:00 PM or 4:00 PM, depending upon whether the 3:00 PM time was started during DST or standard time.

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
           .withIntervalInHours(2 * 24)
           // interval is actually set at 48 hours' worth of milliseconds
           .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInDays(2)) // interval is set in calendar days
    .build();
```

How-To: Create a Trigger that Executes Every Week

If you want a trigger that always fires at a certain time of day each week, use CronTrigger or CalendarIntervalTrigger because they can preserve the firing time across daylight savings time changes.

Using CronTrigger

Create a CronTrigger. that executes every Wednesday at 3:00 PM:

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(weeklyOnDayAndHourAndMinute(DateBuilder.WEDNESDAY, 15, 0))
    // fire every wednesday at 15:00
    .build();
```

- OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 ? * WED"))
    // fire every wednesday at 15:00
    .build();
```

Using SimpleTrigger

Create a SimpleTrigger that executes at 3:00 PM tomorrow, and then every 7 * 24 hours after that.

Note: Be aware that this trigger might not always fire at 3:00 PM, because adding 7 * 24 hour on days when daylight savings time shifts can result in an execution time of 2:00 PM or 4:00 PM, depending upon whether the 3:00 PM time was started during DST or standard time.

```
trigger = newTrigger()
```

```
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(7 * 24)
            // interval is actually set at 7 * 24 hours worth of milliseconds
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInWeeks(1)) // interval is set in calendar weeks
    .build();
```

How-To: Create a Trigger that Executes Every 2 Weeks

As with a trigger meant to fire every two days, CronTrigger will not work for this schedule. (For more information about this issue, see "How-To: Create a Trigger that Executes Every 2 Days" on page 34.) You need to use a SimpleTrigger or CalendarIntervalTrigger:

Using SimpleTrigger

Create a SimpleTrigger that executes 3:00 PM tomorrow, and then every two weeks (14 * 24 hours) after that.

Note: Be aware that this trigger might not always fire at 3:00 PM, because adding (14 * 24) hours on days when daylight savings time shifts can result in an execution time of 2:00 PM or 4:00 PM, depending upon whether the 3:00 PM time was started during DST or standard time.

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
         .withIntervalInHours(14 * 24)
         // interval is actually set at 14 * 24 hours' worth of milliseconds
         .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
         .withIntervalInWeeks(2)) // interval is set in calendar weeks
    .build();
```

How-To: Create a Trigger that Executes Every Month

Using CronTrigger

Create a CronTrigger that executes on a specified day each month at 3:00 PM.

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(monthlyOnDayAndHourAndMinute(5, 15, 0))
       // fire on the 5th day of every month at 15:00
    .build();
```

- OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 5 * ?"))
        // fire on the 5th day of every month at 15:00
    .build();
```

- OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 L * ?"))
         // fire on the last day of every month at 15:00
    .build();
```

- OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 LW * ?"))
        // fire on the last weekday day of every month at 15:00
    .build();
```

- OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 L-3 * ?"))
        // fire on the third to last day of every month at 15:00
    .build();
```

The triggers shown above were created by changing the day-of-month field. There are other possible combinations as well, which are more fully covered in the API documentation.

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInMonths(1)) // interval is set in calendar months
    .build();
```