```
/*
 Copyright 2012 Robert C. Ilardi

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

 http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
 */

/**
 * Created Aug 3, 2012
 */
package com.roguelogic.util;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

/**
 * @author Robert C. Ilardi
 *
 *         This is a Sample Class for a Standalone Process that would run as
 *         part of a Batch. Implementations that use this template may be run
 *         from a scheduler such as Cron or Autosys or as Manual Utility
```

```
 *         Processes.
 *
 *         I have released this code under the Apache 2.0 Open Source License.
 *         Please feel free to use this as a template for your own Batch Job or
 *         Utility Process Implementations.
 *
 *         Finally, as you will notice I used STDOUT AND STDERR for all logging.
 *         This is for simplicity of the template. You can use Log4J or Java
 *         Logging or any other log library you prefer. In my professional
 *         experience, I also include an Exception or "Throwable" emailer
 *         mechanism so that our development team receives all exceptions from
 *         any process even front-ends in real time.
 *
 */

public class DoNothingStandaloneProcess {

  /*
   * I personally like having a single property file for the configuration of
   * all my batch jobs and utilities. In my professional projects, I actually
   * have a more complex method of properties management, where all properties
   * are stored in a database table, and I have something called a Resource
   * Bundle and Resource Helper facility to manage it.
   *
   * My blog at EnterpriseProgrammer.com has more information on properties and
   * connection management using this concept.
   *
   * However for demonstration purposes I am using a simple Properties object to
   * manage all configuration data for the Standalone Process Template. Feel
   * free to replace this field with a more advanced configuration management
   * mechanism that means your needs.
```

```
 */
private Properties appProps;

/*
 * This flag ensures that the Cleanup method only runs once. This is because I
 * wanted to have a shutdown hook, in case the process receives an interrupt
 * signal and in the main method, I explicitly call cleanup() from the finally
 * block. Technically the shutdown hook based on my implementation is only a
 * backup so it actually will never run unless there's a situation like an
 * interrupt signal.
 */
private boolean ranCleanup = false;

/*
 * If this variable is set to true, any exception caused in the cleanup
 * routine will cause the entire process to exit non-zero.
 *
 * However in my professional experience, we usually just want to log these
 * exceptions, perhaps even email them to the team for investigation later,
 * and allow the process to exit ZERO, so that the batch job scheduler can
 * continue onto the next job, especially is the real execution is completed.
 */
private boolean treatCleanupExceptionsAsFatal = false;

/**
 * I'm not really using the constructor here. I purpose more explicit init
 * methods. It's a good practice especially if you work with a lot of
 * reflection, however feel free to add some base initialization here if you
 * prefer.
 */
public DoNothingStandaloneProcess() {}
```

```
// Start public methods that shouldn't be customized by the user
// ---------------------------------------------------------------->

/**
 * The init method wraps two user customizable methods: 1. readProperties(); -
 * Use this to add reads from the appProps object. 2. customProcessInit() -
 * Use this to customize your process before the execution logic runs.
 *
 * As stated previously, so not touch these methods, they are simple wrappers
 * around the methods you should customize instead and provide what in my
 * professional experience are good log messages for batch jobs or utilities
 * to print out, such as the execution timing information. This is especially
 * useful for long running jobs. You can eventually take average over the
 * course of many runs of the batch job, and then you will know when your
 * batch job is behaving badly, when it's taking too long to finish execution.
 */
public synchronized void init() {
  long start, end, total;

  System.out.println("Initialization at: " + GetTimeStamp());
  start = System.currentTimeMillis();

  readProperties(); // Hook to the user's read properties method.

  customProcessInit(); // Hook to the user's custom process init method!

  end = System.currentTimeMillis();

  total = end - start;
```

```
  System.out.println("Initialization Completed at: " + GetTimeStamp());
  System.out.println("Total Init Execution Time: "
      + CompactHumanReadableTimeWithMs(total));
}

/**
 * Because we aren't using a more advanced mechanism for properties
 * management, I have included this method to allow the main() method to set
 * the path to the main properties file used by the batch jobs.
 *
 * In my professional versions of this template, this method is embedded in
 * the init() method which basically will initialize the Resource Helper
 * component and obtain the properties from the configuration tables instead.
 *
 * Again you shouldn't touch this method's implementation, instead use
 * readProperties() to customize what you do with the properties after the
 * properties load.
 */
public synchronized void loadProperties(String appPropsPath)
    throws IOException {
  FileInputStream fis = null;

  try {
    fis = new FileInputStream(appPropsPath);
    appProps = new Properties();
    appProps.load(fis);
  } // End try block
  finally {
    if (fis != null) {
      try {
        fis.close();
```

```
      }
      catch (Exception e) {}
    }
  }
}

/**
 * This method performs the cleanup of any JDBC connections, files, sockets,
 * and other resources that your execution process or your initialization
 * process may have opened or created.
 *
 * Once again do not touch this method directly, instead put your cleanup code
 * in the customProcessCleanup() method.
 *
 * This method is called automatically in the last finally block of the main
 * method, and if there's an interrupt signal or other fatal issue where
 * somehow the finally block didn't get called the Runtime shutdown hook will
 * invoke this method on System.exit...
 *
 * @throws Exception
 */
public synchronized void cleanup() throws Exception {
  long start, end, total;

  // This prevents cleanup from running more than onces.
  if (ranCleanup) {
    return;
  }

  try {
    System.out.println("Starting Cleanup at: " + GetTimeStamp());
```

```
    start = System.currentTimeMillis();

    customProcessCleanup(); // Hook to the users Process Cleanup Method

    end = System.currentTimeMillis();

    total = end - start;

    System.out.println("Cleanup Completed at: " + GetTimeStamp());
    System.out.println("Total Cleanup Execution Time: "
        + CompactHumanReadableTimeWithMs(total));

    ranCleanup = true;
  } // End try block
  catch (Exception e) {
    /*
     * It is in my experience that the Operating System will cleanup anything
     * we have "forgotten" to clean up. Therefore I do not want to waste my
     * production support team members time at 3AM in the morning to handle
     * "why did a database connection not close" It will close eventually,
     * since it is just a socket, and even if it doesn't we'll catch this in
     * other jobs which may fail due to the database running out of
     * connections.
     *
     * However I usually have these exceptions emailed to our development team
     * for investigation the next day. For demo purposes I did not include my
     * Exception/Stacktrace Emailing utility, however I encourage you to add
     * your own.
     *
     * If you really need the process to exit non-ZERO because of the cleanup
     * failing, set the treatCleanupExceptionsAsFatal to true.
```

```
     */
    e.printStackTrace();

    if (treatCleanupExceptionsAsFatal) {
      throw e;
    }
  }
}

/**
 * This method wraps the customExecuteProcessing() method where you should add
 * your customize process execution logic to.
 *
 * Again like the other methods in this section of the class, do not modify
 * this method directly.
 *
 * For demo purposes I made it throw the generic Exception object so that your
 * customExecuteProcessing() method can throw any Exception it likes.
 *
 * @throws Exception
 */
public synchronized void executeProcessing() throws Exception {
  long start, end, total;

  ranCleanup = false;

  System.out.println("Start Processing at: " + GetTimeStamp());
  start = System.currentTimeMillis();

  customExecuteProcessing(); // Hook to the User's Custom Execute Processing
                             // Method! - Where the magic happens!
```

```
  end = System.currentTimeMillis();

  total = end - start;

  System.out.println("Processing Completed at: " + GetTimeStamp());
  System.out.println("Total Processing Execution Time: "
      + CompactHumanReadableTimeWithMs(total));
}

/**
 * This is the method that adds the shutdown hook.
 *
 * All this method does it property invokes the
 * Runtime.getRuntime().addShutdownHook(Thread t); method by adding an
 * anonymous class implementation of a thread.
 *
 * This thread's run method simply calls the Process's cleanup method.
 *
 * Whenever I create a class like this, I envision it being ran two ways,
 * either directly from the main() method or as part of a larger component,
 * which may wrap this entire class (A HAS_A OOP relationship).
 *
 * In the case of the wrapper, adding the shutdown hook might be optional
 * since the wrapper may want to handle shutdown on it's own.
 *
 */
public synchronized void addShutdownHook() {
  Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
      try {
```

```
        cleanup();
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  });
}

/**
 * This method is only provided in case you are loading properties from an
 * input stream or other non-standard source that is not a File.
 *
 * It becomes very useful in the wrapper class situation I described in the
 * comments about the addShutdownHook method.
 *
 * Perhaps the wrapping process reads properties from a Database or a URL?
 *
 * @param appProps
 */
public synchronized void setAppProperties(Properties appProps) {
  this.appProps = appProps;
}

/**
 * Used to detect which mode the cleanup exceptions are handled in.
 *
 * @return
 */
public boolean isTreatCleanupExceptionsAsFatal() {
  return treatCleanupExceptionsAsFatal;
```

```
}

/**
 * Use this method to set if you want to treat cleanup exception as fatal. The
 * default, and my personal preference is not to make these exception fatal.
 * But I added the flexibility into the template for your usage.
 *
 * @param treatCleanupExceptionsAsFatal
 */
public void setTreatCleanupExceptionsAsFatal(
    boolean treatCleanupExceptionsAsFatal) {
  this.treatCleanupExceptionsAsFatal = treatCleanupExceptionsAsFatal;
}

// ------------------------------------------------------------------->

// Start methods that need to be customized by the user
// ------------------------------------------------------------------->

/**
 * In general for performance reasons and for clarity even above performance,
 * I like pre-caching the properties as Strings or parsed Integers, etc,
 * before running any real business logic.
 *
 * This is why I provide the hook to readProperties which should read
 * properties from the appProps field (member variable).
 *
 * If you don't want to pre-cache your property values you can leave this
 * method blank. However I believe it's a good practice especially if your
 * batch process is a high speed ETL Loader process where every millisecond
 * counts when loading millions of records.
```

```
 */
private synchronized void readProperties() {
  System.out.println("Add Your Property Reads Here!");
}

/**
 * After the properties are read from the readProperties() method this method
 * is called.
 *
 * It is provided for the user to add custom initialization processing.
 *
 * Let's say you want to open all JDBC connections at the start of a process,
 * this is probably the right place to do so.
 *
 * For more complex implementations, this is the best place to create and
 * initialize all your sub-components of your process.
 *
 * Let's say you have a DbConnectionPool, a Country Code Mapping utility, an
 * Address Fuzzy Logic Matching library.
 *
 * This is where I would initialize these components.
 *
 * The idea is to fail-fast in your batch processes, you don't want to wait
 * until you processed 10,000 records before some logic statement is triggered
 * to lazy instantiate these components, and because of a network issue or a
 * configuration mistake you get a fatal exception and your process exists,
 * and your data is only partially loaded and you or your production support
 * team members have to debug not only the process but did the portion of the
 * data already loaded make it in ok. This is extremely important if your
 * batch process interacts is real-time system components such as message
 * publishers, maybe you started publishing the updated records to downstream
```

```
 * consumers?
 *
 * Fail-Fast my friends... And as soon as the process starts if possible!
 */
private synchronized void customProcessInit() {
  System.out.println("Add Custom Initialization Logic Here!");
}

/**
 * This is where you would add your custom cleanup processing. If you open and
 * connections, files, sockets, etc and keep references to these
 * objects/resources opened as fields in your class which is a good idea in
 * some cases especially long running batch processes you need a hook to be
 * able to close these resources before the process exits.
 *
 * This is where that type of logic should be placed.
 *
 * Now you can throw any exception you like, however the cleanup wrapper
 * method will simply log these exceptions, the idea here is that, even though
 * cleanup is extremely important, the next step of the process is a
 * System.exit and the operating system will most-likely reclaim any resources
 * such as files and sockets which have been left opened, after some bit of
 * time.
 *
 * Now my preference is usually not to wake my production support guys up
 * because a database connection (on the extremely rare occasion) didn't close
 * correctly. The process still ran successfully at this point, so just exit
 * and log it.
 *
 * However if you really need to make the cleanup be truly fatal to the
 * process you will have to set treatCleanupExceptionsAsFatal to true.
```

```
 *
 * @throws Exception
 */
private synchronized void customProcessCleanup() throws Exception {
  System.out.println("Add Custom Cleanup Logic Here!");
}

private synchronized void customExecuteProcessing() throws Exception {
  System.out.println("Add Custom Processing Logic Here!");
}

// ---------------------------------------------------------------->

/*
 * Start String Utility Methods These are methods I have in my custom
 * "StringUtils.java" class I extracted them and embedded them in this class
 * for demonstration purposes.
 *
 * I encourage everyone to build up their own set of useful String Utility
 * Functions please feel free to add these to your own set if you need them.
 */
// ---------------------------------------------------------------->

/**
 * This will return a string that is a human readable time sentence. It is the
 * "compact" version because instead of having leading ZERO Days, Hours,
 * Minutes, Seconds, it will only start the sentence with the first non-zero
 * time unit.
 *
 * In my string utils I have a non-compact version as well that prints the
 * leading zero time units.
```

```
 *
 * All depends on how you need to presented in your logs.
 */
public static String CompactHumanReadableTimeWithMs(long milliSeconds) {
  long days, hours, inpSecs, leftOverMs;
  int minutes, seconds;
  StringBuffer sb = new StringBuffer();

  inpSecs = milliSeconds / 1000; // Convert Milliseconds into Seconds
  days = inpSecs / 86400;
  hours = (inpSecs - (days * 86400)) / 3600;
  minutes = (int) (((inpSecs - (days * 86400)) - (hours * 3600)) / 60);
  seconds = (int) (((inpSecs - (days * 86400)) - (hours * 3600)) - (minutes * 60));
  leftOverMs = milliSeconds - (inpSecs * 1000);

  if (days > 0) {
    sb.append(days);
    sb.append((days != 1 ? " Days" : " Day"));
  }

  if (sb.length() > 0) {
    sb.append(", ");
  }

  if (hours > 0 || sb.length() > 0) {
    sb.append(hours);
    sb.append((hours != 1 ? " Hours" : " Hour"));
  }

  if (sb.length() > 0) {
    sb.append(", ");
```

```
  }

  if (minutes > 0 || sb.length() > 0) {
    sb.append(minutes);
    sb.append((minutes != 1 ? " Minutes" : " Minute"));
  }

  if (sb.length() > 0) {
    sb.append(", ");
  }

  if (seconds > 0 || sb.length() > 0) {
    sb.append(seconds);
    sb.append((seconds != 1 ? " Seconds" : " Second"));
  }

  if (sb.length() > 0) {
    sb.append(", ");
  }

  sb.append(leftOverMs);
  sb.append((seconds != 1 ? " Milliseconds" : " Millisecond"));

  return sb.toString();
}

/**
 * NVL = Null Value, in my experience, most times, we want to treat empty or
 * whitespace only strings are NULLs
 *
 * So this method is here to avoid a lot of if (s == null || s.trim().length()
```

```
 * == 0) all over the place, instead you will find if(IsNVL(s)) instead.
 */
public static boolean IsNVL(String s) {
  return s == null || s.trim().length() == 0;
}

/**
 * Simply returns a timestamp as a String.
 *
 * @return
 */
public static String GetTimeStamp() {
  return (new java.util.Date()).toString();
}

// ------------------------------------------------------------------->

// Start Main() Helper Static Methods
// ------------------------------------------------------------------->

/**
 * This method returns true if the command line arguments are valid, and false
 * otherwise.
 *
 * Please change this method to meet your implementation's requirements.
 */
private static boolean CheckCommandLineArguments(String[] args) {
  boolean ok = false;

  /*
   * This is configured to make sure we only have one parameter which is the
```

```
   * app properties. We could have made it more advanced and actually checked
   * if the file exists, but just checking is the parameter exists for demo
   * purposes is good enough.
   */
  ok = args.length == 1 && !IsNVL(args[0]);

  return ok;
}

/**
 * This prints to STDERR (a common practice), the command line usage of the
 * program.
 *
 * Please change this to meet your implementation's command line arguments.
 */
private static void PrintUsage() {
  StringBuffer sb = new StringBuffer();

  sb.append("\nUsage: java ");

  sb.append(DoNothingStandaloneProcess.class.getName());

  /*
   * Modify this append call to have each command line argument name example:
   * sb.append(
   * " [APP_PROPERTIES_FILE] [SOURCE_INPUT_FILE] [WSDL_URL] [TARGET_OUTPUT_FILE]"
   * );
   *
   * For demo purposes we will only use [APP_PROPERTIES_FILE]
   */
  sb.append(" [APP_PROPERTIES_FILE]");
```

```
  sb.append("\n\n");

  System.err.print(sb.toString());
}

/**
 * I usually like the Batch and Daemon Processes or Utilities to print a small
 * Banner at the top of their output.
 *
 * Please change this to suit your needs.
 */
private static void PrintWelcome() {
  StringBuffer sb = new StringBuffer();

  sb.append("\n*********************************************\n");
  sb.append("*       Do Nothing Standalone Process       *\n");
  sb.append("*********************************************\n\n");

  System.out.print(sb.toString());
}

/**
 * This method simple prints the process startup time. I found this to be very
 * useful in batch job logs. I probably wouldn't change it, but you can if you
 * really need to.
 */
private static void PrintStartupTime() {
  StringBuffer sb = new StringBuffer();

  sb.append("Startup Time: ");
```

```
  sb.append(GetTimeStamp());
  sb.append("\n\n");

  System.out.print(sb.toString());
}

// Start Main() Method
// ------------------------------------------------------------------->

/**
 * Here's your standard main() method which allows you to start a Java program
 * from the command line. You can probably use this as is, once you rename the
 * DoNothingStandaloneProcess class name to a proper name to represent your
 * implementation correctly.
 *
 * MAKE SURE: To change the data type of the process object reference to the
 * name of your process implementation class. Other than that, you are good to
 * go with this main method!
 */
public static void main(String[] args) {
  int exitCode;
  DoNothingStandaloneProcess process = null;

  if (!CheckCommandLineArguments(args)) {
    PrintUsage();
    exitCode = 1;
  }
  else {
    try {
      PrintWelcome();
```

```
  PrintStartupTime();

  process = new DoNothingStandaloneProcess();

  process.setTreatCleanupExceptionsAsFatal(false); // I don't believe
                                                   // cleanup exceptions
                                                   // are really fatal,
                                                   // but that's up to
                                                   // you...

  process.loadProperties(args[0]); // Load properties using the file way.

  process.init(); // Performance Process Initialization, again I don't
                  // like over use of the constructor.

  process.addShutdownHook(); // Just in case we get an interrupt signal...

  process.executeProcessing(); // Do the actually business logic
                               // execution!

  // If we made it to this point without an exception, that means
  // we are successful, the process exit code should be ZERO for SUCCESS!
  exitCode = 0;
} // End try block
catch (Exception e) {
  exitCode = 1; // If there was an exception, the process exit code should
                // be NON-ZERO for FAILURE!
  e.printStackTrace(); // Log the exception, if you have an Exception
                       // email utility like I do, use that instead.
}
finally {
```

```
    if (process != null) {
      try {
        process.cleanup(); // Technically we don't need to do this because
                           // of the shutdown hook
        // But I like to be explicit here to show when during a
        // normal execution, when the call
        // to cleanup should happen.
      }
      catch (Exception e) {
        // We shouldn't receive an exception
        // But in case there is a runtime exception
        // Just print it, but treat it as non-fatal.
        // Technically most if not all resources
        // will be reclaimed by the operating system as an
        // absolute last resort
        // so we did our best attempt at cleaning things up,
        // but we don't want to wake our developers or our
        // production services team
        // up at 3 in the morning because something weird
        // happened during cleanup.
        e.printStackTrace();

        // If we set the process to treat cleanup exception as fatal
        // the exit code will be set to 1...
        if (process != null && process.isTreatCleanupExceptionsAsFatal()) {
          exitCode = 1;
        }
      }
    }
  } // End finally block
} // End else block
```

```
    // Make sure our standard streams are flushed
    // so we don't miss anything in the logs.
    System.out.flush();
    System.err.flush();

    System.out.println("Process Exit Code = " + exitCode);

    System.out.flush();

    // Make sure to return the exit code to the parent process
    System.exit(exitCode);
  }

  // ------------------------------------------------------------------>

}
```