Technical assignment 1: refactoring task - Nikolaos Christidis (nick.christidis@yahoo.com)

- Paper Topics
  - refactored code
  - test cases
  - visual vm screenshots
  - comments

Refactored code:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.nio.channels.FileLock;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.StampedLock;
import java.util.function.BiConsumer;
import java.util.function.Supplier;

@ParserFacade.ThreadSafe
@ParserFacade.SingletonScope
public class ParserFacade {

 @GuardedBy(threadAccess = "ParserFacade class monitor")
  private static boolean ACQUIRE_GLOBAL_FILE_LOCK = false;

  private static final int READ_LOCK_TIMEOUT_SEC = 1;
  private static final int WRITE_LOCK_TIMEOUT_SEC = 1;

  private static final int OP_READ_RETRY_COUNT = 5;
  private static final int BUFFER_SIZE = 1024;

  @GuardedBy(threadAccess = "ParserFacade class monitor")
  private static ParserFacade INSTANCE;
```

```
  private final StampedLock stampedLock = new StampedLock();

  /*
     Note: From multiple thread access protection we could have used also
AtomicReference, AtomicReferenceFieldUpdater,
          sun.misc.Unsafe, or in Java9 VarHandle, based on contention (mid
to low --> Atomic, high --> Lock).


          From protection from different JVM processes, or other processes
(for example this file might modified
          from other applications-programs, etc.) it will be good
          to use FileChannel-->FileLock (lock globally)
          on top of our multithreading application management.

   */
  @GuardedBy(threadAccess = "stampedLock", processAccess =
"java.nio.channels.FileLock")
  private File file;

  /*
     Note: We use this mutex, in order to eliminate
OverlappingFileLockException errors occurred inside the same JVM when
          we want to acquire global file lock.
   */
  private final Object MUTEX = new Object();

  private ParserFacade() {
    // Note: no instantiation
  }

  public static ParserFacade getInstance() {
    synchronized (ParserFacade.class) {
      if (INSTANCE == null) {
        INSTANCE = new ParserFacade();
      }
      return INSTANCE;
    }
  }

  public void setAcquireGlobalFileLock(boolean b) {
    synchronized (ParserFacade.class) {
      ACQUIRE_GLOBAL_FILE_LOCK = b;
    }
```

```
  }

  public Pair<Optional<File>, Optional<Long> /*optimistic read stamp*/>
getFile() {

    // Note: first try optimistic read (non-pessimistic read lock). 90% of the
cases.
    for (int i = 0; i < OP_READ_RETRY_COUNT; i++) {
      long optimisticRead = stampedLock.tryOptimisticRead();
      Optional<File> current = Optional.ofNullable(file);
      if (stampedLock.validate(optimisticRead)) {
        return Pair.of(current, Optional.of(optimisticRead));
      }
    }

    // Note: if we are here, we should downgrade to pessimistic read lock, it
should be 10% of the cases.
    final long stamp = acquireReadLock();
    if (stamp != 0) {
      try {
        Optional<File> current = Optional.ofNullable(this.file);
        return Pair.of(current, Optional.empty());
      } finally {
        stampedLock.unlockRead(stamp);
      }
    } else {
      throw new CouldNotAcquireReadLock();
    }

  }

  public void setFile(File f) {
    setFile(f, null);
  }

  public void setFile(File f, @Nullable Long optimisticReadStamp) {

    // Note: try acquire write lock from optimistic read if provided
    if (optimisticReadStamp != null) {
      long stamp =
stampedLock.tryConvertToWriteLock(optimisticReadStamp);
      if (stamp != 0) { // Note: acquired write lock
        try {
          file = f;
```

```
            return;
          } finally {
            stampedLock.unlockWrite(stamp);
          }
        }
      }

      // Note: otherwise, acquire write lock
      final long stamp = acquireWriteLock();
      if (stamp != 0) {
        try {
          file = f;
        } finally {
          stampedLock.unlockWrite(stamp);
        }
      } else {
        throw new CouldNotAcquireWriteLock();
      }
    }

    public Pair<String, Optional<Long> /*optimistic read stamp*/>
getContent() {
      return getContentOperation(() ->
__getContent(StandardCharsets.UTF_8,
ACQUIRE_GLOBAL_FILE_LOCK));
    }

    public Pair<String, Optional<Long> /*optimistic read stamp*/>
getContentWithoutUnicode() {
      return getContentOperation(this::__getContentWithoutUnicode);
    }

    public void saveContent(String content) {
      saveContent(content, StandardCharsets.UTF_8, null, false);
    }

    public void saveContent(String content, boolean append) {
      saveContent(content, StandardCharsets.UTF_8, null, append);
    }

    public void saveContent(String content, boolean append, long
optimisticReadStamp) {
      saveContent(content, StandardCharsets.UTF_8, optimisticReadStamp,
append);
```

```
  }

  public void clearContent() {
    saveContent("");
  }

  private void saveContent(String content,
      Charset encoding,
      @Nullable Long optimisticReadStamp,
      boolean append) {

    saveContentOperation(content,
        encoding,
        optimisticReadStamp,
        (_content, _encoding) -> __saveContent(_content, _encoding,
append, ACQUIRE_GLOBAL_FILE_LOCK)
    );
  }

  private void saveContentOperation(String content,
      Charset encoding,
      @Nullable Long optimisticReadStamp,
      BiConsumer<String, Charset> contentStorage) {

    // Note: try acquire write lock from optimistic read if provided
    if (optimisticReadStamp != null) {
      long stamp =
stampedLock.tryConvertToWriteLock(optimisticReadStamp);
      if (stamp != 0) { // Note: acquired write lock
        try {
          contentStorage.accept(content, encoding);
          return;
        } finally {
          stampedLock.unlockWrite(stamp);
        }
      }
    }

    // Note: otherwise, acquire write lock
    final long stamp = acquireWriteLock();
    if (stamp != 0) {
      try {
        contentStorage.accept(content, encoding);
      } finally {
```

```
            stampedLock.unlockWrite(stamp);
        }
    } else {
        throw new CouldNotAcquireWriteLock();
    }
  }

  private Pair<String, Optional<Long> /*optimistic read stamp*/>
getContentOperation(Supplier<String> contentProvider) {
    // Note: first try optimistic read (non-pessimistic read lock). 90% of the
cases.
    for (int i = 0; i < OP_READ_RETRY_COUNT; i++) {
        long optimisticRead = stampedLock.tryOptimisticRead();
        String current = contentProvider.get();
        if (stampedLock.validate(optimisticRead)) {
            return Pair.of(current, Optional.of(optimisticRead));
        }
    }

    // Note: if we are here, we should downgrade to pessimistic read lock, it
should be 10% of the cases.
    final long stamp = acquireReadLock();
    if (stamp != 0) {
        try {
            return Pair.of(contentProvider.get(), Optional.empty());
        } finally {
            stampedLock.unlockRead(stamp);
        }
    } else {
        throw new CouldNotAcquireReadLock();
    }
  }

  private void __saveContent(String content, Charset encoding, boolean
append, boolean acquireGlobalFileLock) {
    try (FileOutputStream o = new FileOutputStream(file, append)) {

        if (acquireGlobalFileLock) {
            synchronized (MUTEX) {
                FileLock exclusiveLock = o.getChannel().tryLock(0L,
Long.MAX_VALUE, false);
                if (exclusiveLock == null) {
                    throw new CouldNotAcquireGlobalFileLock(false);
                }
```

```
            try {
              writeToFile(content, encoding, o);
            } finally {
              exclusiveLock.release();
            }
          }
        } else {
          writeToFile(content, encoding, o);
        }

      } catch (IOException error) {
        throw new SaveOperationFailure("could not save content", error);
      }
    }

    private String __getContent(Charset charset, boolean
acquireGlobalFileLock) {

      try (FileInputStream i = new FileInputStream(file)) {
        StringBuilder output = new StringBuilder();

        if (acquireGlobalFileLock) {
          synchronized (MUTEX) {
            FileLock sharedLock = i.getChannel().tryLock(0,
Long.MAX_VALUE, true);
            if (sharedLock == null) {
              throw new CouldNotAcquireGlobalFileLock(true);
            }
            try {
              readFromFile(charset, i, output);
            } finally {
              sharedLock.release();
            }
          }
        } else {
          readFromFile(charset, i, output);
        }

        return output.toString();

      } catch (IOException error) {
        throw new ReadOperationFailure("could not read content", error);
      }
```

```
  }

  // Note: from old code --> 0x80 == 128 in decimal which is exclusive
upper limit of ascii, so we read in ascii encoding.
  private String __getContentWithoutUnicode() {
    return __getContent(StandardCharsets.US_ASCII,
ACQUIRE_GLOBAL_FILE_LOCK);
  }

  private void readFromFile(Charset charset, FileInputStream in,
StringBuilder output) throws IOException {
    int data;
    byte[] buf = new byte[BUFFER_SIZE];
    while ((data = in.read(buf, 0, buf.length)) > 0) {
      output.append(new String(buf, 0, data, charset));
    }
  }

  private void writeToFile(String content, Charset encoding,
FileOutputStream out) throws IOException {
    byte[] contentBytes = content.getBytes(encoding);
    out.write(contentBytes, 0, contentBytes.length);
  }

  private long acquireReadLock() {
    try {
      return stampedLock.tryReadLock(READ_LOCK_TIMEOUT_SEC,
TimeUnit.SECONDS);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      throw new RuntimeException(e);
    }
  }

  private long acquireWriteLock() {
    try {
      return stampedLock.tryWriteLock(WRITE_LOCK_TIMEOUT_SEC,
TimeUnit.SECONDS);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      throw new RuntimeException(e);
    }
  }
```

```java
@Documented
@Target({ ElementType.PARAMETER })
@Retention(RetentionPolicy.SOURCE) // Note: documentation purpose
@interface Nullable {
}

@Documented
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.SOURCE) // Note: documentation purpose
@interface GuardedBy {

  String threadAccess();

  String processAccess() default "";
}

@Documented
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.SOURCE) // Note: documentation purpose
@interface ThreadSafe {
}

@Documented
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.SOURCE) // Note: documentation purpose
@interface SingletonScope {
}

// ---- code infrastructure(ds, errors, etc.) ---
static class Pair<E1, E2> {

  private final E1 first;
  private final E2 second;

  private Pair(E1 first, E2 second) {
    this.first = first;
    this.second = second;
  }

  public static <E1, E2> Pair<E1, E2> of(E1 e1, E2 e2) {
    return new Pair<>(e1, e2);
  }

  public E1 getFirst() {
```

```
      return first;
    }

    public E2 getSecond() {
      return second;
    }
  }

  static class SaveOperationFailure extends RuntimeException {

    public SaveOperationFailure(String message, Throwable e) {
      super(message, e);
    }
  }

  static class ReadOperationFailure extends RuntimeException {

    public ReadOperationFailure(String message, Throwable e) {
      super(message, e);
    }
  }

  static class CouldNotAcquireReadLock extends RuntimeException {
    public CouldNotAcquireReadLock() {
      super("could not acquire read lock");
    }
  }

  static class CouldNotAcquireWriteLock extends RuntimeException {
    public CouldNotAcquireWriteLock() {
      super("could not acquire write lock");
    }
  }

  static class CouldNotAcquireGlobalFileLock extends RuntimeException {

    public CouldNotAcquireGlobalFileLock(boolean shared) {
      super("could not acquire global file lock, shared: " + shared);
    }
  }
}
```

Test cases

```
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import java.io.File;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Arrays;
import java.util.Optional;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Phaser;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.stream.IntStream;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

public class ParserFacadeTest {

  private ParserFacade parserFacade;

  @Before
  public void init() {
    parserFacade = ParserFacade.getInstance();
  }

  @Test
  public void setFileGetFileWorksAsExpected() throws Exception {
```

```
    // given
    int idx = ThreadLocalRandom.current().nextInt(10);
    Path testFile = Files.createTempFile("testFile__", idx + "");

    // when
    parserFacade.setFile(testFile.toFile());

    // then
    ParserFacade.Pair<Optional<File>, Optional<Long>> result =
parserFacade.getFile();
    assertTrue(result.getFirst().isPresent());
    // Note: test runs from one thread so always we have optimistic read
stamp.
    assertTrue(result.getSecond().isPresent());

    result.getSecond().ifPresent(optimisticReadStamp -> {

      try {

        // given
        Path newTestFile = Files.createTempFile("testFile__", idx + "");

        // when
        parserFacade.setFile(newTestFile.toFile(),
optimisticReadStamp);

        // then
        ParserFacade.Pair<Optional<File>, Optional<Long>> r =
parserFacade.getFile();
        assertTrue(r.getFirst().isPresent());
        // Note: test runs from one thread so always we have optimistic
read stamp.
        assertTrue(r.getSecond().isPresent());

      } catch (Exception e) {
        fail(e.getMessage());
        e.printStackTrace(System.err);
      }

    });
```

```
  }

  @Test
  public void saveContentAndGetContentWorksAsExpected() throws
Exception {

    // given
    String toWrite = "hello world, {} this is ascii, ôą this is utf-8 ";

    int idx = ThreadLocalRandom.current().nextInt(10);

    Path testFile = Files.createTempFile("testFile__", idx + "");
    parserFacade.setFile(testFile.toFile());

    // when
    parserFacade.saveContent(toWrite);

    ParserFacade.Pair<String, Optional<Long>> result =
parserFacade.getContent();
    ParserFacade.Pair<String, Optional<Long>> resultNoUnicode =
parserFacade.getContentWithoutUnicode();

    // then
    assertEquals(toWrite, result.getFirst());
    assertTrue(result.getSecond().isPresent());

    // Note: we can see utf8 characters not displayed properly, because
we read with ascii encoding.
    assertEquals(
        "hello world, {} this is ascii, ���� this is utf-8 ",
        resultNoUnicode.getFirst()
    );
    assertTrue(resultNoUnicode.getSecond().isPresent());

  }

  @Test
  public void saveContentAppendModeWorksAsExpected() throws
Exception {

    // given
    int idx = ThreadLocalRandom.current().nextInt(10);
```

```
    Path testFile = Files.createTempFile("testFile__", idx + "");
    parserFacade.setFile(testFile.toFile());

    // when
    String contents = "a,b,c,d,e,f,g";
    for (String elem : contents.split(",")) {
      parserFacade.saveContent(elem, true);
    }

    // then
    assertEquals("abcdefg", parserFacade.getContent().getFirst());

  }

  @Test
  public void
checkMultiThreadOperationsSafety_PessimisticRead_AcquireGlobalLo
ck() throws Exception {

    // given
    parserFacade.setAcquireGlobalFileLock(true);

    int runs = 100;

    int idx = ThreadLocalRandom.current().nextInt(10);
    Path testFile = Files.createTempFile("testFile__", idx + "");
    parserFacade.setFile(testFile.toFile());


    for (int i = 0; i < runs; i++) {
      checkMultiThreadOperationsSafety(false);
    }
  }

  @Test
  public void
checkMultiThreadOperationsSafety_PessimisticRead_NoAcquireGloba
lLock() throws Exception {

    // given
    int runs = 100;
```

```
    int idx = ThreadLocalRandom.current().nextInt(10);
    Path testFile = Files.createTempFile("testFile__", idx + "");
    parserFacade.setFile(testFile.toFile());

    for (int i = 0; i < runs; i++) {
      checkMultiThreadOperationsSafety(false);
    }
  }

  @Test
  public void
checkMultiThreadOperationsSafety_OptimisticRead_AcquireGlobalLoc
k() throws Exception {

    // given
    parserFacade.setAcquireGlobalFileLock(true);

    int runs = 100;

    int idx = ThreadLocalRandom.current().nextInt(10);
    Path testFile = Files.createTempFile("testFile__", idx + "");
    parserFacade.setFile(testFile.toFile());


    for (int i = 0; i < runs; i++) {
      checkMultiThreadOperationsSafety(true);
    }

  }

  @Test
  public void
checkMultiThreadOperationsSafety_OptimisticRead_NoAcquireGlobal
Lock() throws Exception {

    // given
    int runs = 100;

    int idx = ThreadLocalRandom.current().nextInt(10);
    Path testFile = Files.createTempFile("testFile__", idx + "");
    parserFacade.setFile(testFile.toFile());
```

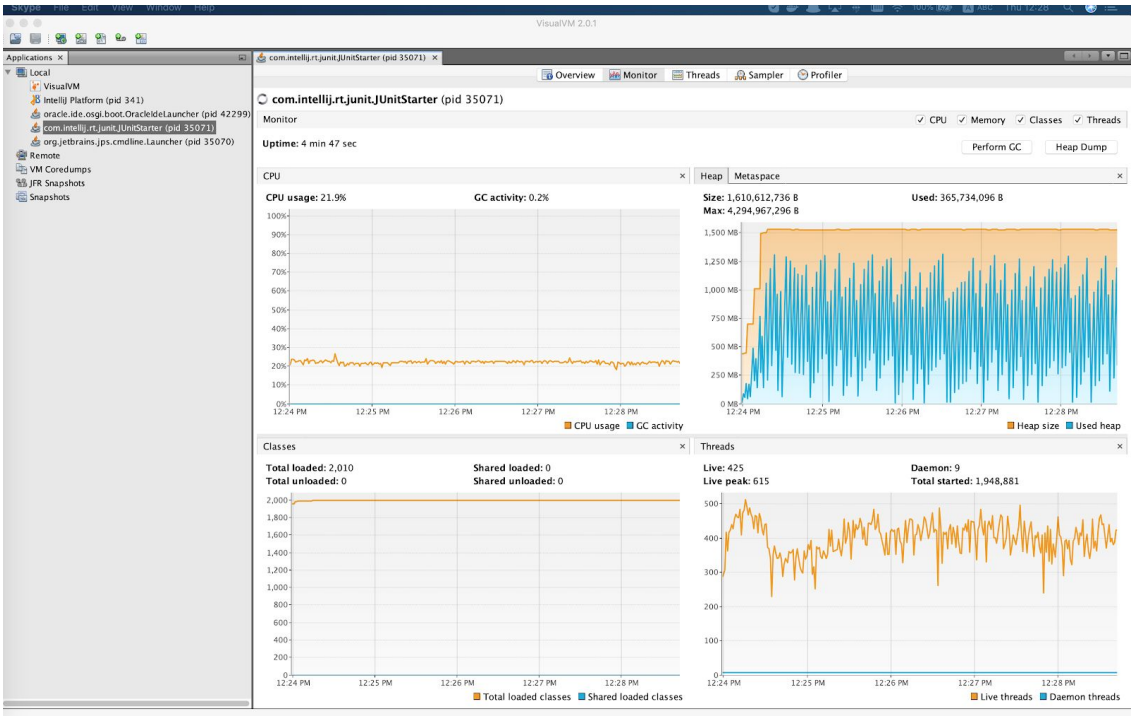
```
    for (int i = 0; i < runs; i++) {
      checkMultiThreadOperationsSafety(true);
    }

  }

  // --- utils ---

  private void checkMultiThreadOperationsSafety(boolean
tryOptimisticReads) {

    // create some readers to create traffic for read lock.
    int readersSize = 70;
    Phaser readersOnReadyState = new Phaser(readersSize + 1  /*
plus one for junit test thread / waiter role */);

    ExecutorService readers =
Executors.newFixedThreadPool(readersSize, new ThreadFactory() {

      private final AtomicInteger idx = new AtomicInteger(0);

      @Override public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setName("reader--" + idx.getAndIncrement());
        return t;
      }
    });
    AtomicBoolean readersRunning = new AtomicBoolean(true);

    IntStream.rangeClosed(1, readersSize).forEach(readerIdx -> {
      readers.submit(() -> {
        try {
          readersOnReadyState.arriveAndDeregister();

          while (readersRunning.get()) {
            parserFacade.getContent();
            Thread.sleep(70);
            Thread.yield();
            parserFacade.getContentWithoutUnicode();
            Thread.sleep(40);
```

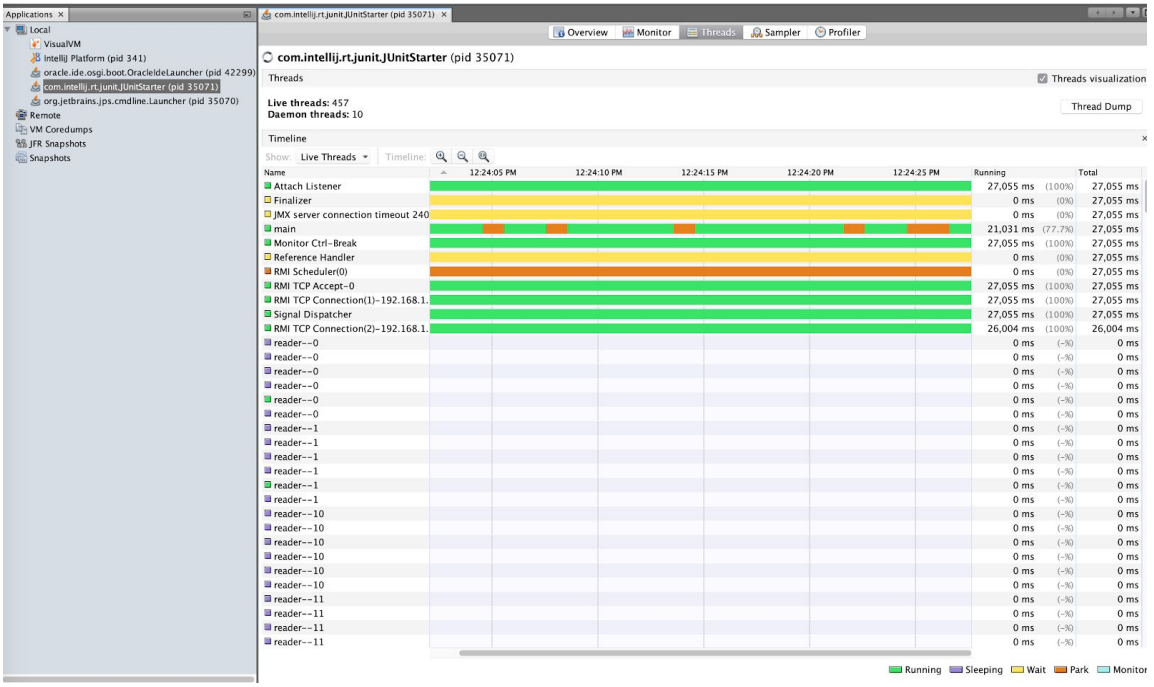
```
            Thread.yield();

          }
        } catch (Exception e) {
          System.err.println(Thread.currentThread().getName() + " --
reader error: " + e.getMessage());
          // let the reader exit.
        }
      });
    });

    readersOnReadyState.arriveAndAwaitAdvance();

    // create writers to create traffic for write lock.
    int workersSize = 40;

    Phaser writersFinishedWork = new Phaser(workersSize + 1 /* plus
one for junit test thread / waiter role */);

    CyclicBarrier writersRendezvousBeforeWork = new
CyclicBarrier(workersSize,
        () -> System.out.println("all workers (size = " + workersSize + ")
at 'fair' position to access/test save contents method")
    );

    ExecutorService writers =
Executors.newFixedThreadPool(workersSize, new ThreadFactory() {

      private final AtomicInteger idx = new AtomicInteger(0);

      @Override public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setName("writer--" + idx.getAndIncrement());
        return t;
      }
    });

    // when
    int timesEachWorkerWillSave = 10;

    for (int w = 0; w < workersSize; w++) {
```

```
        writers.submit(() -> {
          // ~~ rendezvous point ~~
          try {
            writersRendezvousBeforeWork.await();
          } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
          } catch (BrokenBarrierException ignored) {
            Assert.fail();
          }

          // ~~ actual work ~~
          for (int i=0; i<timesEachWorkerWillSave; i++) {
            if (tryOptimisticReads) {

              Optional<Long> optimisticReadStampH = Optional.empty();
              while (!optimisticReadStampH.isPresent()) {
                ParserFacade.Pair<String, Optional<Long>> result =
parserFacade.getContent();
                optimisticReadStampH = result.getSecond();
              }
              parserFacade.saveContent("1,", true,
optimisticReadStampH.get());

            } else {
              parserFacade.saveContent("1,", true);
            }
          }
          writersFinishedWork.arriveAndDeregister();
        });

      }

      // then
      writersFinishedWork.arriveAndAwaitAdvance();
      readersRunning.set(false);

      String contents = parserFacade.getContent().getFirst();

      int sum = Arrays.stream(contents.split(","))
          .map(Integer::parseInt)
          .reduce(0, Integer::sum);
```

```
        assertEquals(workersSize * timesEachWorkerWillSave, sum);

        //cleanup
        parserFacade.clearContent();

        writers.shutdown();
        readers.shutdown();
    }

}
```

Visual VM Screenshots





Note: In order to perform a quick n dirty benchmark (instead of jmh, JMeter, etc) you can increase the runs of the tests in the test cases provided and examine how it `rolls` via VisualVM or something similar.

Comments.

- The old code was not doing file writing and reading with buffering, instead it read/write one byte at a time, so this changed, in order to use a buffer of size 1024. 1024 has extracted as a constant so that can be changed more easily.

- Also getContent(), getContentWithoutUnicode() and saveContent() methods are not thread safe, meaning that in an environment with multiple readers/writers, a reader can read stall data when a writer performs a saveContent() operation, so for these reasons I have leveraged the usage of StampedLock (optimistic reads usage).

- setFile(..) getFile() have used synchronization blocks, these do not scale well (contention), so with StampedLock we have separate readers (getFile()) from writers (setFile(...)). Moreover setFile(..) and getFile() were synchronized on object(this) monitor but getContent(), saveContent() and getContentWithoutUnicode() were not synchronized on object(this) monitor, so we were having race condition problems there. For exampe, we could have read the contents of and old file during a thread performed a setFile(...) operation, etc.

- For file operations (getContent(), getContentWithoutUnicode(), saveContent()) we have leverage OS native file locks (it is selectable as a constant) with the help of java.nio.channels.FileLock

- I have created some annotations for documentation purposes (retention policy == source).

- I have leverage Java 8 functional interfaces in order to model templates and change behaviour where needed (reuse).

- I have created my unchecked exceptions in order to deal with exceptions. This piece of code I have guessed that will be used from another `internal` service so no need for checked exceptions. However If we were creating a library, we could use checked exceptions in order to force the client/user of our library to handle these errors.

  Note: New JVM languages (Kotlin, Scala) does not have the notion of checked exceptions.

- I created a Pair in order to hold info (Tuple2 like in vavr.io)

- Moreover the getInstance() method, is doing a lazy initialization of the object (we want to use ParserFacade as a singleton) and is not thread safe, so I have leverage class monitor synchronization.
  Double checked locking is considered an antipattern (Java Concurrency in Practice, Brian Goetz, 16.2.4)

- I have written test cases for testing behaviour and multi-thread access (pessimistic read / optimistic read). Also these tests, if we increase the runs, can play the role of a quick n dirty torture tests.

Kind regards,
Nikolaos Christidis (nick.christidis@yahoo.com)