Bloom Filter Class using MD5 as Hash Function

```java
package com.skjegstad.utils;

import java.io.Serializable;
import java.nio.charset.Charset;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.BitSet;
import java.util.Collection;

// @author Magnus Skjegstad <magnus@skjegstad.com>
public class BloomFilter<E> implements Serializable {
    private BitSet bitset;
    private int bitSetSize;
    private double bitsPerElement;
    private int expectedNumberOfFilterElements; // expected (maximum) number of elements to be added
    private int numberOfAddedElements; // number of elements actually added to the Bloom filter
    private int k; // number of hash functions

    static final Charset charset = Charset.forName("UTF-8"); // encoding used for storing hash values as strings

    static final String hashName = "MD5"; // MD5 gives good enough accuracy in most circumstances. Change to SHA1 if it's needed
    static final MessageDigest digestFunction;
    static { // The digest method is reused between instances
        MessageDigest tmp;
        try {
            tmp = java.security.MessageDigest.getInstance(hashName);
        } catch (NoSuchAlgorithmException e) {
            tmp = null;
        }
        digestFunction = tmp;
    }

    /**
      * Constructs an empty Bloom filter. The total length of the Bloom filter will be
      * c*n.
      *
      * @param c is the number of bits used per element.
      * @param n is the expected number of elements the filter will contain.
      * @param k is the number of hash functions used.
      */
    public BloomFilter(double c, int n, int k) {
      this.expectedNumberOfFilterElements = n;
      this.k = k;
      this.bitsPerElement = c;
      this.bitSetSize = (int)Math.ceil(c * n);
      numberOfAddedElements = 0;
      this.bitset = new BitSet(bitSetSize);
    }

    /**
```

```
     * Constructs an empty Bloom filter. The optimal number of hash functions (k) is estimated from the total size of the Bloom
     * and the number of expected elements.
     *
     * @param bitSetSize defines how many bits should be used in total for the filter.
     * @param expectedNumberOElements defines the maximum number of elements the filter is expected to contain.
     */
    public BloomFilter(int bitSetSize, int expectedNumberOElements) {
        this(bitSetSize / (double)expectedNumberOElements,
             expectedNumberOElements,
             (int) Math.round((bitSetSize / (double)expectedNumberOElements) * Math.log(2.0)));
    }

    /**
     * Constructs an empty Bloom filter with a given false positive probability. The number of bits per
     * element and the number of hash functions is estimated
     * to match the false positive probability.
     *
     * @param falsePositiveProbability is the desired false positive probability.
     * @param expectedNumberOfElements is the expected number of elements in the Bloom filter.
     */
    public BloomFilter(double falsePositiveProbability, int expectedNumberOfElements) {
        this(Math.ceil(-(Math.log(falsePositiveProbability) / Math.log(2))) / Math.log(2), // c = k / ln(2)
             expectedNumberOfElements,
             (int)Math.ceil(-(Math.log(falsePositiveProbability) / Math.log(2)))); // k = ceil(-log_2(false prob.))
    }

    /**
     * Construct a new Bloom filter based on existing Bloom filter data.
     *
     * @param bitSetSize defines how many bits should be used for the filter.
     * @param expectedNumberOfFilterElements defines the maximum number of elements the filter is expected to contain.
     * @param actualNumberOfFilterElements specifies how many elements have been inserted into the <code>filterData</code> BitSet.
     * @param filterData a BitSet representing an existing Bloom filter.
     */
    public BloomFilter(int bitSetSize, int expectedNumberOfFilterElements, int actualNumberOfFilterElements, BitSet filterData) {
        this(bitSetSize, expectedNumberOfFilterElements);
        this.bitset = filterData;
        this.numberOfAddedElements = actualNumberOfFilterElements;
    }

    /**
     * Generates a digest based on the contents of a String.
     *
     * @param val specifies the input data.
```

```
     * @param charset specifies the encoding of the input data.
     * @return digest as long.
     */
    public static int createHash(String val, Charset charset) {
        return createHash(val.getBytes(charset));
    }

    /**
     * Generates a digest based on the contents of a String.
     *
     * @param val specifies the input data. The encoding is expected to be UTF-8.
     * @return digest as long.
     */
    public static int createHash(String val) {
        return createHash(val, charset);
    }

    /**
     * Generates a digest based on the contents of an array of bytes.
     *
     * @param data specifies input data.
     * @return digest as long.
     */
    public static int createHash(byte[] data) {
        return createHashes(data, 1)[0];
    }

    /**
     * Generates digests based on the contents of an array of bytes and splits the result into 4-byte int's and store them in an array. The
     * digest function is called until the required number of int's are produced. For each call to digest a salt
     * is prepended to the data. The salt is increased by 1 for each call.
     *
     * @param data specifies input data.
     * @param hashes number of hashes/int's to produce.
     * @return array of int-sized hashes
     */
    public static int[] createHashes(byte[] data, int hashes) {
        int[] result = new int[hashes];

        int k = 0;
        byte salt = 0;
        while (k < hashes) {
            byte[] digest;
            synchronized (digestFunction) {
                digestFunction.update(salt);
                salt++;
                digest = digestFunction.digest(data);
            }

            for (int i = 0; i < digest.length/4 && k < hashes; i++) {
                int h = 0;
                for (int j = (i*4); j < (i*4)+4; j++) {
                    h <<= 8;
                    h |= ((int) digest[j]) & 0xFF;
```

```
                }
                result[k] = h;
                k++;
            }
        }
        return result;
    }

    /**
     * Compares the contents of two instances to see if they are equal.
     *
     * @param obj is the object to compare to.
     * @return True if the contents of the objects are equal.
     */
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final BloomFilter<E> other = (BloomFilter<E>) obj;
        if (this.expectedNumberOfFilterElements != other.expectedNumberOfFilterElements) {
            return false;
        }
        if (this.k != other.k) {
            return false;
        }
        if (this.bitSetSize != other.bitSetSize) {
            return false;
        }
        if (this.bitset != other.bitset && (this.bitset == null || !this.bitset.equals(other.bitset))) {
            return false;
        }
        return true;
    }

    /**
     * Calculates a hash code for this class.
     * @return hash code representing the contents of an instance of this class.
     */
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 61 * hash + (this.bitset != null ? this.bitset.hashCode() : 0);
        hash = 61 * hash + this.expectedNumberOfFilterElements;
        hash = 61 * hash + this.bitSetSize;
        hash = 61 * hash + this.k;
        return hash;
    }


```

```
        /**
         * Calculates the expected probability of false positives based on
         * the number of expected filter elements and the size of the Bloom filter.
         * <br /><br />
         * The value returned by this method is the <i>expected</i> rate of false
         * positives, assuming the number of inserted elements equals the number of
         * expected elements. If the number of elements in the Bloom filter is less
         * than the expected value, the true probability of false positives will be lower.
         *
         * @return expected probability of false positives.
         */
        public double expectedFalsePositiveProbability() {
            return getFalsePositiveProbability(expectedNumberOfFilterElements);
        }

        /**
         * Calculate the probability of a false positive given the specified
         * number of inserted elements.
         *
         * @param numberOfElements number of inserted elements.
         * @return probability of a false positive.
         */
        public double getFalsePositiveProbability(double numberOfElements) {
            // (1 - e^(-k * n / m)) ^ k
            return Math.pow((1 - Math.exp(-k * (double) numberOfElements
                            / (double) bitSetSize)), k);

        }

        /**
         * Get the current probability of a false positive. The probability is calculated from
         * the size of the Bloom filter and the current number of elements added to it.
         *
         * @return probability of false positives.
         */
        public double getFalsePositiveProbability() {
            return getFalsePositiveProbability(numberOfAddedElements);
        }


        /**
         * Returns the value chosen for K.<br />
         * <br />
         * K is the optimal number of hash functions based on the size
         * of the Bloom filter and the expected number of inserted elements.
         *
         * @return optimal k.
         */
        public int getK() {
            return k;
```

```java
    }

    /**
     * Sets all bits to false in the Bloom filter.
     */
    public void clear() {
        bitset.clear();
        numberOfAddedElements = 0;
    }

    /**
     * Adds an object to the Bloom filter. The output from the object's
     * toString() method is used as input to the hash functions.
     *
     * @param element is an element to register in the Bloom filter.
     */
    public void add(E element) {
       add(element.toString().getBytes(charset));
    }

    /**
     * Adds an array of bytes to the Bloom filter.
     *
     * @param bytes array of bytes to add to the Bloom filter.
     */
    public void add(byte[] bytes) {
       int[] hashes = createHashes(bytes, k);
       for (int hash : hashes)
           bitset.set(Math.abs(hash % bitSetSize), true);
       numberOfAddedElements ++;
    }

    /**
     * Adds all elements from a Collection to the Bloom filter.
     * @param c Collection of elements.
     */
    public void addAll(Collection<? extends E> c) {
        for (E element : c)
            add(element);
    }

    /**
     * Returns true if the element could have been inserted into the Bloom filter.
     * Use getFalsePositiveProbability() to calculate the probability of this
     * being correct.
     *
     * @param element element to check.
     * @return true if the element could have been inserted into the Bloom filter.
     */
    public boolean contains(E element) {
        return contains(element.toString().getBytes(charset));
    }

    /**
```

```
     * Returns true if the array of bytes could have been inserted into the Bloom filter.
     * Use getFalsePositiveProbability() to calculate the probability of this
     * being correct.
     *
     * @param bytes array of bytes to check.
     * @return true if the array could have been inserted into the Bloom filter.
     */
    public boolean contains(byte[] bytes) {
        int[] hashes = createHashes(bytes, k);
        for (int hash : hashes) {
            if (!bitset.get(Math.abs(hash % bitSetSize))) {
                return false;
            }
        }
        return true;
    }

    /**
     * Returns true if all the elements of a Collection could have been inserted
     * into the Bloom filter. Use getFalsePositiveProbability() to calculate the
     * probability of this being correct.
     * @param c elements to check.
     * @return true if all the elements in c could have been inserted into the Bloom filter.
     */
    public boolean containsAll(Collection<? extends E> c) {
        for (E element : c)
            if (!contains(element))
                return false;
        return true;
    }

    /**
     * Read a single bit from the Bloom filter.
     * @param bit the bit to read.
     * @return true if the bit is set, false if it is not.
     */
    public boolean getBit(int bit) {
        return bitset.get(bit);
    }

    /**
     * Set a single bit in the Bloom filter.
     * @param bit is the bit to set.
     * @param value If true, the bit is set. If false, the bit is cleared.
     */
    public void setBit(int bit, boolean value) {
        bitset.set(bit, value);
    }

    /**
     * Return the bit set used to store the Bloom filter.
```

```
     * @return bit set representing the Bloom filter.
     */
    public BitSet getBitSet() {
        return bitset;
    }

    /**
     * Returns the number of bits in the Bloom filter. Use count() to retrieve
     * the number of inserted elements.
     *
     * @return the size of the bitset used by the Bloom filter.
     */
    public int size() {
        return this.bitSetSize;
    }

    /**
     * Returns the number of elements added to the Bloom filter after it
     * was constructed or after clear() was called.
     *
     * @return number of elements added to the Bloom filter.
     */
    public int count() {
        return this.numberOfAddedElements;
    }

    /**
     * Returns the expected number of elements to be inserted into the filter.
     * This value is the same value as the one passed to the constructor.
     *
     * @return expected number of elements.
     */
    public int getExpectedNumberOfElements() {
        return expectedNumberOfFilterElements;
    }

    /**
     * Get expected number of bits per element when the Bloom filter is full. This value is set by the constructor
     * when the Bloom filter is created. See also getBitsPerElement().
     *
     * @return expected number of bits per element.
     */
    public double getExpectedBitsPerElement() {
        return this.bitsPerElement;
    }

    /**
     * Get actual number of bits per element based on the number of elements that have currently been inserted and the length
     * of the Bloom filter. See also getExpectedBitsPerElement().
     *
     * @return number of bits per element.
     */
    public double getBitsPerElement() {
```

```
            return this.bitSetSize / (double)numberOfAddedElements;
        }
    }
```

How to Use Bloom Filter Class

```java
package com.skjegstad.utils;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

// @author Magnus Skjegstad

public class BloomfilterBenchmark {
    static int elementCount = 50000; // Number of elements to test

    public static void printStat(long start, long end) {
        double diff = (end - start) / 1000.0;
        System.out.println(diff + "s, " + (elementCount / diff) + " elements/s");
    }

    public static void main(String[] argv) {


        final Random r = new Random();

        // Generate elements first
        List<String> existingElements = new ArrayList(elementCount);
        for (int i = 0; i < elementCount; i++) {
            byte[] b = new byte[200];
            r.nextBytes(b);
            existingElements.add(new String(b));
        }

        List<String> nonExistingElements = new ArrayList(elementCount);
        for (int i = 0; i < elementCount; i++) {
            byte[] b = new byte[200];
            r.nextBytes(b);
            nonExistingElements.add(new String(b));
        }

        BloomFilter<String> bf = new BloomFilter<String>(0.001, elementCount);

        System.out.println("Testing " + elementCount + " elements");
        System.out.println("k is " + bf.getK());

        // Add elements
        System.out.print("add(): ");
        long start_add = System.currentTimeMillis();
        for (int i = 0; i < elementCount; i++) {
            bf.add(existingElements.get(i));
        }
        long end_add = System.currentTimeMillis();
        printStat(start_add, end_add);

        // Check for existing elements with contains()
        System.out.print("contains(), existing: ");
        long start_contains = System.currentTimeMillis();
        for (int i = 0; i < elementCount; i++) {
```

```
            bf.contains(existingElements.get(i));
        }
        long end_contains = System.currentTimeMillis();
        printStat(start_contains, end_contains);

        // Check for existing elements with containsAll()
        System.out.print("containsAll(), existing: ");
        long start_containsAll = System.currentTimeMillis();
        for (int i = 0; i < elementCount; i++) {
            bf.contains(existingElements.get(i));
        }
        long end_containsAll = System.currentTimeMillis();
        printStat(start_containsAll, end_containsAll);

        // Check for nonexisting elements with contains()
        System.out.print("contains(), nonexisting: ");
        long start_ncontains = System.currentTimeMillis();
        for (int i = 0; i < elementCount; i++) {
            bf.contains(nonExistingElements.get(i));
        }
        long end_ncontains = System.currentTimeMillis();
        printStat(start_ncontains, end_ncontains);

        // Check for nonexisting elements with containsAll()
        System.out.print("containsAll(), nonexisting: ");
        long start_ncontainsAll = System.currentTimeMillis();
        for (int i = 0; i < elementCount; i++) {
            bf.contains(nonExistingElements.get(i));
        }
        long end_ncontainsAll = System.currentTimeMillis();
        printStat(start_ncontainsAll, end_ncontainsAll);

    }
}
```