1. BloomFilter.java:
    - Public void add(String s) – Add an element for BloomFilterMurmur and BloomFilterFNV
    - Public boolean appears(String s) – Check if an element appears in BloomFilterMurmur and BloomFilterFNV.
    - Public int filterSize() – Returns the logical size of the filter.
    - Public int dataSize() – Returns the number of elements added to the filter.
    - Public int numHashes – Returns the number of hash functions in use by the filter.

   BloomFilterFNV.java:
    - Private class HashFunctionFNV
        - Public int hash(String s) – Hash a string s using hash FNV.
        - Private int hash(byte[] k) – Hash a byte array using hash FNV.

   BloomFilterMurmur.java:
    - Private class HashFunctionMurmur (Code taken from Murmur hash website)

   BloomFilterRan.java:
    - Public void add(String s) – Add an element to BloomFilterRan with many hash functions.
    - Public void appears(String s) – Check if an element appears in BloomFilterRan

   BloomJoin.java:
    - Public void join(String r3) – Join relations r1 and r2 and write the result to r3.

   DynamicFilter.java:
    - Public void add(String s) – Add a string s to the dynamic filter, potentially adds s to multiple filters.
    - Public Boolean appears(String s) – Check if s appears in the dynamic filter, potentially checks multiple filters for a string s.
    - Public int filterSize() – Returns the cumulative size of all created Bloom Filters.
    - Public int dataSize() – Returns the number of elements added to the dynamic filter.
    - Public int numHashes() – Returns the number of hash functions in use by the filter.

   FalsePositives.java
    - Public static void main(String[] args) – Runs the false positive experiment and prints out the results.
    - Private static void initialize() – Creates the four Bloom Filters required for the experiment and adds a list of random strings to them.
    - Private static void randomString() – Generates a random string of random length between 3 - bitsPerElement+1.

   HashFunction.java (abstract)
    - Public abstract int hash(String s) – Returns the hash for the specified string s.
    - Public static int mod(long x, long y) – Mods the two long values and returns the result.

   HashFunctionRan.java
    - Public int hash(String s) – Hash the string s and return the result.
    - Public int hash(int x) – Hash the int x and return the result.
    - Private int getPrime(int n) – Returns the first positive prime larger than n.
    - Private Boolean isPrime(int num) – Returns true if num is a prime number, false otherwise.

2. For generating the k hash values, I created a for-loop over k iterations (from 0 to k-1) using the loop iterator $i.

   Then, before hashing string S and setting a bit to 1 in the Bloom Filter, I add $i to the end of the string S. We will generate k different hash values because we will run our hash function with k different unique strings.

   A key component of this is to do the same thing when checking if a string S exists in our bloom filter. Before checking if a bit in the filter is set to 1, add our loop iterator $i to the end of S.

   Example (k=2):
   Add string “ball” to filter:
   Hash “ball1”, set result bit to 1
   Hash “ball2”, set result bit to 1
   Check if string “ball” is in filter:
   Hash “ball1”, if 0 return false
   Hash “ball2”, if 0 return false, else return true

3. For generating k hash values with BloomFilterRan, I created a for-loop over k iterations creating k random hash function objects. Each random hash function generates its own random values for a and b.

   This way, each random hash function object can be treated as a unique random function. So, when generating k hash values, we just need to hash string S with each of our k random hash functions.

4. I created a false positive experiment by first creating one of each Bloom Filter, BloomFilterRan, BloomFilterFNV, BloomFilterMurmur, and DynamicFilter. For each filter, I add k random strings to it (each filter should now contain same set of random strings). I also store all the random strings in a set L.

   Now, we take a disjoint set of L, called A. A is also composed of random strings. We would like for A to be a large enough set to give use accurate false positive readings.

   For every string ‘a’ in A, check each Bloom Filter for if ‘a’ appears in the filter. If ‘a’ appears in the filter, increment the false positives counter for the filter. This works because A is disjoint from set L, so any string in A surely should not be in any Bloom Filter.

   Finally we output the results of the experiment for each of our Bloom Filters.

5. Note: Add/search time is in seconds for adding/searching 100000 elements

```
4 bits:            FP Rate:  Theoretical:  Add Time:   Search Time:

BloomFilterFNV:     12.53 %    14.58 %       0.027          0.007
BloomFilterMurmur: 12.64 %    14.58 %       0.022          0.011
BloomFilterRan:     28.22 %    14.58 %       0.013          0.009

8 bits:

BloomFilterFNV:     2.03 %      2.12 %
BloomFilterMurmur: 1.99 %      2.12 %
BloomFilterRan:     5.11 %      2.12 %

16 bits:

BloomFilterFNV:     0.05 %      .045 %
BloomFilterMurmur: 0.04 %      .045 %
BloomFilterRan:     0.69 %      .045 %
```

The reason BloomFilterRan has so many more false positives is because the other hash functions are deterministic. My algorithm to generate random strings probably isn't random enough. When generating random strings to test, they are a lot like the strings generated for the filters. This results in similar hash codes and thus, more false positives.

Deterministic hash functions expect this phenomenon because they explicitly hash strings so that the keys are far apart from each other.

We see that search time is always faster than add time, this is especially the case when a string is not in our filter. We only return true in the appears function if every index for every hash value is set to 1.

6. After running my false positives experiment with a set of 500000 strings, I retrieved the following output:

```
BloomFilterRan:     9.33 %
DynamicFilter:      4.57 %
```

The dynamic filter will have a better false positive rate as more elements are added to the filter. This is because Dynamic Filter is the same as BloomFilterRan, except Dynamic Filter allocates extra space for elements when it needs too.

By allocating the extra space, we also increase the number of hash functions used to add an element to the filter. By increasing the number of hash functions and the space, we are ensuring that we will use the new bloom filters when they are created.

Example:
It could be the case that we add too many elements to BloomFilterRan, thus setting every bit to 1. In this case, the Bloom Filter is no longer useful, since each time we check for an element, we will always get a false positive.

With Dynamic Filter, we solve this problem by allocating new space for a filter whenever we need it. Thus, Dynamic Filter is better when we need to store many elements in a filter.