

LUCENE - FIELD OPTIONS

http://www.tutorialspoint.com/lucene/lucene_fieldoptions.htm

Copyright © tutorialspoint.com

Field is the most important and the foundation unit of indexing process. It is the actual object containing the contents to be indexed. When we add a field, lucene provides numerous controls on the field using Field Options which states how much a field is to be searchable.

We add *Documents* containing *Fields* to *IndexWriter* where *IndexWriter* is used to update or create indexes.

Now we'll show you a step by step process to get a kick start in understanding of various Field options using a basic example.

Various Field Options

- **Index.ANALYZED** - First analyze then do indexing. Used for normal text indexing. Analyzer will break the field's value into stream of tokens and each token is searchable separately.
- **Index.NOT_ANALYZED** - Don't analyze but do indexing. Used for complete text indexing for example person's names, URL etc.
- **Index.ANALYZED_NO_NORMS** - Variant of Index.ANALYZED. Analyzer will break the field's value into stream of tokens and each token is searchable separately but NORMs are not stored in the indexes. NORMs are used to boost searching but are sometime memory consuming.
- **Index.Index.NOT_ANALYZED_NO_NORMS** - Variant of Index.NOT_ANALYZED. Indexing is done but NORMs are not stored in the indexes.
- **Index.NO** - Field value is not searchable.

Use of Field Options

- Create a method to get a lucene document from a text file.
- Create various types of fields which are key value pairs containing keys as names and values as contents to be indexed.
-
- Set field to be analyzed or not. In our case, only contents is to be analyzed as it can contain data such as a, am, are, an etc. which are not required in search operations.
-
- Add the newly created fields to the document object and return it to the caller method.

```
private Document getDocument(File file) throws IOException{
    Document document = new Document();

    //index file contents
    Field contentField = new Field(LuceneConstants.CONTENT,
        new FileReader(file));
    //index file name
    Field fileNameField = new Field(LuceneConstants.FILE_NAME,
        file.getName(),
        Field.Store.YES, Field.Index.NOT_ANALYZED);
    //index file path
    Field filePathField = new Field(LuceneConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES, Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);
}
```

```
    return document;
}
```

Example Application

Let us create a test Lucene application to test indexing process.

Step	Description
1	Create a project with a name <i>LuceneFirstApplication</i> under a package <i>com.tutorialspoint.lucene</i> as explained in the <i>Lucene - First Application</i> chapter. You can also use the project created in <i>EJB - First Application</i> chapter as such for this chapter to understand indexing process.
2	Create <i>LuceneConstants.java</i> , <i>TextFileFilter.java</i> and <i>Indexer.java</i> as explained in the <i>Lucene - First Application</i> chapter. Keep rest of the files unchanged.
3	Create <i>LuceneTester.java</i> as mentioned below.
4	Clean and Build the application to make sure business logic is working as per the requirements.

LuceneConstants.java

This class is used to provide various constants to be used across the sample application.

```
package com.tutorialspoint.lucene;

public class LuceneConstants {
    public static final String CONTENTS="contents";
    public static final String FILE_NAME="filename";
    public static final String FILE_PATH="filepath";
    public static final int MAX_SEARCH = 10;
}
```

TextFileFilter.java

This class is used as a .txt file filter.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;

public class TextFileFilter implements FileFilter {

    @Override
    public boolean accept(File pathname) {
        return pathname.getName().toLowerCase().endsWith(".txt");
    }
}
```

Indexer.java

This class is used to index the raw data so that we can make it searchable using lucene library.

```
package com.tutorialspoint.lucene;

import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;
```

```

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.Version;

public class Indexer {

    private IndexWriter writer;

    public Indexer(String indexDirectoryPath) throws IOException{
        //this directory will contain the indexes
        Directory indexDirectory =
            FSDirectory.open(new File(indexDirectoryPath));

        //create the indexer
        writer = new IndexWriter(indexDirectory,
            new StandardAnalyzer(Version.LUCENE_36), true,
            IndexWriter.MaxFieldLength.UNLIMITED);
    }

    public void close() throws CorruptIndexException, IOException{
        writer.close();
    }

    private Document getDocument(File file) throws IOException{
        Document document = new Document();

        //index file contents
        Field contentField = new Field(LuceneConstants.CONTENT,
            new FileReader(file));
        //index file name
        Field fileNameField = new Field(LuceneConstants.FILE_NAME,
            file.getName(),
            Field.Store.YES, Field.Index.NOT_ANALYZED);
        //index file path
        Field filePathField = new Field(LuceneConstants.FILE_PATH,
            file.getCanonicalPath(),
            Field.Store.YES, Field.Index.NOT_ANALYZED);

        document.add(contentField);
        document.add(fileNameField);
        document.add(filePathField);

        return document;
    }

    private void indexFile(File file) throws IOException{
        System.out.println("Indexing "+file.getCanonicalPath());
        Document document = getDocument(file);
        writer.addDocument(document);
    }

    public int createIndex(String dataDirPath, FileFilter filter)
        throws IOException{
        //get all files in the data directory
        File[] files = new File(dataDirPath).listFiles();

        for (File file : files) {
            if(!file.isDirectory()
                && !file.isHidden()
                && file.exists()
                && file.canRead()
                && filter.accept(file)
            ){
                indexFile(file);
            }
        }
    }
}

```

```

    }
    }
    return writer.numDocs();
}
}

```

LuceneTester.java

This class is used to test the indexing capability of lucene library.

```

package com.tutorialspoint.lucene;

import java.io.IOException;

public class LuceneTester {

    String indexDir = "E:\\Lucene\\Index";
    String dataDir = "E:\\Lucene\\Data";
    Indexer indexer;

    public static void main(String[] args) {
        LuceneTester tester;
        try {
            tester = new LuceneTester();
            tester.createIndex();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void createIndex() throws IOException{
        indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
        long endTime = System.currentTimeMillis();
        indexer.close();
        System.out.println(numIndexed+" File indexed, time taken: "
            +(endTime-startTime)+" ms");
    }
}

```

Data & Index directory creation

I've used 10 text files named from record1.txt to record10.txt containing simply names and other details of the students and put them in the directory **E:\Lucene\Data**. [Test Data](#). An index directory path should be created as **E:\Lucene\Index**. After running this program, you can see the list of index files created in that folder.

Running the Program:

Once you are done with creating source, creating the raw data, data directory and index directory, you are ready for this step which is compiling and running your program. To do this, Keep LuceneTester.java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **LuceneTester** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console:

```

Indexing E:\Lucene\Data\record1.txt
Indexing E:\Lucene\Data\record10.txt
Indexing E:\Lucene\Data\record2.txt
Indexing E:\Lucene\Data\record3.txt
Indexing E:\Lucene\Data\record4.txt
Indexing E:\Lucene\Data\record5.txt
Indexing E:\Lucene\Data\record6.txt
Indexing E:\Lucene\Data\record7.txt
Indexing E:\Lucene\Data\record8.txt
Indexing E:\Lucene\Data\record9.txt

```

10 File indexed, time taken: 109 ms

Once you've run the program successfully, you will have following content in your **index directory**:

Name	Date modified	Type	Size
 _0.fdt	5/25/2014 3:15 PM	FDT File	1 KB
 _0.fdx	5/25/2014 3:15 PM	FDX File	1 KB
 _0.fnm	5/25/2014 3:15 PM	FNM File	1 KB
 _0.frq	5/25/2014 3:15 PM	FRQ File	1 KB
 _0	5/25/2014 3:15 PM	Mixed Mode CD C...	1 KB
 _0.prx	5/25/2014 3:15 PM	PRX File	1 KB
 _0.tii	5/25/2014 3:15 PM	TII File	1 KB
 _0.tis	5/25/2014 3:15 PM	TIS File	1 KB
 segments.gen	5/25/2014 3:15 PM	GEN File	1 KB
 segments_1	5/25/2014 3:15 PM	File	1 KB