

Java Tools for Developers

2nd edition



Apache Commons
iText
POI
JFreeChart
EasyMock, JUnit &
PowerMock
JMeter
JavaCC
Solr
James
Jackson
Hibernate Validator

Cheng-Hung Chou

JAVA TOOLS
FOR DEVELOPERS

2ND EDITION

by Cheng-Hung Chou

To
My Wife

7 years

2557 days

61,368 hours

3,682,080 minutes

2 Cute Daughters

TABLE OF CONTENTS

Preface

Apache Commons

Commons Lang

StringUtils

ArrayUtils

ExceptionUtils

WordUtils

Commons IO

IOUtils

FileUtils

FilenameUtils

Commons CSV

CSVParser

Commons Codec

DigestUtils

Commons Net

FTPClient

Commons Compress

Zip

Commons CLI

Commons FileUpload

iText

Creating a PDF document

Basic text elements

Paragraph

Phrase

Chunk

Fonts

Creating a table

[PdfPTable](#)

[PdfPCell](#)

[Encryption](#)

[Adding watermarks](#)

[Creating multiple columns](#)

[Merging documents](#)

[Filling forms](#)

[Servlet](#)

Apache POI

[Creating an Excel document](#)

[Adding formulas](#)

[Formula evaluation](#)

[Plotting a chart](#)

[Creating a Word document](#)

JFreeChart

[Creating a simple chart](#)

[Creating a bar chart](#)

[Creating a scatter plot](#)

[Creating a combined chart](#)

[Making a chart clickable](#)

[Displaying a chart in a web page](#)

EasyMock, JUnit, and PowerMock

[An introduction to JUnit](#)

[Annotations](#)

[Assertions](#)

[Test Runners](#)

[A sample JUnit test case](#)

[An introduction to EasyMock](#)

[Verifying behavior](#)

[Expectations](#)

[EasyMockSupport](#)

[Class mocking](#)

[Partial mocking](#)

[Using EasyMock with JUnit](#)

[Using Annotations](#)

[Using expectations](#)

[PowerMock](#)

[Test coverage analysis](#)

[JMeter](#)

[Building a JMeter test plan](#)

[Starting JMeter](#)

[Adding test elements](#)

[Running a test plan](#)

[Saving a test plan](#)

[Debugging a test plan](#)

[Remote testing](#)

[Test elements](#)

[Samplers](#)

[Logic controllers](#)

[Listeners](#)

[Configuration elements](#)

[Assertions](#)

[Timers](#)

[Pre-processors](#)

[Post-processors](#)

[Testing a web application](#)

[Testing with authentication](#)

[Using regular expressions](#)

[Using HTTP proxy server](#)

[Testing web applications using GWT](#)

[Adding Java classes](#)

[JUnit Request](#)

[Java Request](#)

[JAVACC](#)

[Structure of a grammar file](#)

[Options](#)

[Class declaration](#)

[Token manager declarations](#)

[Lexical specifications](#)

[Production rules](#)

[A simple calculator](#)

[EBNF grammars](#)

[JavaCC grammars](#)

[Generating code](#)

[A formula calculator](#)

[A text processor](#)

[Apache Solr](#)

[Getting started](#)

[Basic directory structure](#)

[Solr home directory](#)

[Solr core directory](#)

[Creating a new core](#)

[SolrCloud](#)

[Document schema](#)

[Document fields](#)

[Field types](#)

[Field analysis](#)

[Updating data](#)

[XML messages](#)

[add](#)

[commit](#)

[delete](#)

[CSV](#)

Querying data

Request handler

Query parameters

Fields

Boolean operators

Grouping searches

Wildcard searches

Regular expression searches

Proximity searches

Range searches

Boosting terms

Special characters

Sorting

Hit highlighting

Faceted search

Field faceting

Query faceting

Range faceting

Pivot Faceting

Result Grouping

Indexing rich documents

Accessing Solr programmatically

Updating data

Querying data

Query

Field faceting

Result grouping

Apache James

Installing James Server

Configuring James Server

config.xml

[environment.xml](#)

[james-fetchmail.xml](#)

[A quick test](#)

[Mailables and matchers](#)

[Creating a custom matcher](#)

[GenericMatcher](#)

[GenericRecipientMatcher](#)

[Creating a custom mailable](#)

[Reading an object store file](#)

Jackson

[POJO model](#)

[JSON Properties](#)

[Property naming and inclusion](#)

[Property exclusion](#)

[Ordering](#)

[Map](#)

[Enum](#)

[Formatting](#)

[Polymorphic types](#)

[Filtering](#)

[Views](#)

[Filters](#)

[Custom Serializers and Deserializers](#)

[Custom Serializers](#)

[Custom Deserializers](#)

[Configurations](#)

Hibernate Validator

[Applying Constraints](#)

[Field-level constraints](#)

[Error messages](#)

[Property-level constraints](#)

[Validating Parameters](#)

[Inheritance](#)

[Grouping Constraints](#)

[Programmatic constraints](#)

[Creating a Custom Constraint](#)

[Creating a constraint annotation](#)

[Creating an validator](#)

Preface

When working on a project, sometimes you face the following questions during the design phase: Should we include this task in this phase? It seems risky. Should we adopt this technology and do code refactoring on existing programs? Should we use this framework instead of writing our own?

Incorporating right development tools into your products can increase productivity of developers and expand functionality on products. You do not need to reinvent everything from scratch. You are not an expert on everything. Leave the job to the expert. But, searching and evaluating right tools can be time consuming. Especially those not well-written or maintained can put your products at risk. Finding well-proven 3rd party tools is the key since they have been using in many products and still improving regularly. A good development tool even can be part of coding practice for developers.

To learn a new technology or tool, the most difficult part is to find where to start with. Sometimes, it is not even well-documented. Or, you do not want to spend time to read the whole book. This book is not trying to teach you how to write Java programs. It assumes you already have basic idea about Java programming language. The main purpose of this book is to broaden your knowledge as a professional Java developer and save you time in finding and learning useful development tools.

Topics in this book cover a variety of development tools in Java. They include APIs, testing tools and servers. In the second edition, it includes the following updates:

Adding to the chapter of **Apache Commons**: Commons CSV and Commons Codec

Adding PowerMock to the chapter of **EasyMock, JUnit, and PowerMock**

Rewriting the chapter of **Apache Solr** to cover Solr 5

Adding two new chapters: **Jackson, Hibernate Validator**

The following are quick summaries of topics in this book:

Apache Commons

The purpose of Apache Commons is to focus on all aspects of reusable Java components. Component interfaces will keep as stable as possible and try to be independent from other components. There are over 40 active components in the Apache Commons so far. Some implementations in the Apache Commons even are included in the core Java API.

iText

Usually, PDF documents are created or updated manually through some kind of end-user applications such as Adobe Acrobat. But, that is only suitable for static documents. If you need to generate PDF documents dynamically based on requests or to generate personalized contents, you need to use different approaches. iText is an API that helps developers to generate or update PDF documents on the fly programmatically without manual process.

Apache POI

Apache POI is the Java API for Microsoft Documents, which allows you to read and write MS Office documents such as Excel, Word and PowerPoint using Java. Through POI, you can generate MS Office documents dynamically based on requests or to generate personalized reports on the fly. POI supports OLE2 files such as XLS, DOC and PPT and new XML based standard Office OpenXML files such as XLSX, DOCX and PPTX.

JFreeChart

JFreeChart is a chart library that helps you to create a variety of chart types such as pie charts, bar charts, line charts or scatter plots in your Swing applications. Many output types such as images (JPEG or PNG) are supported. JFreeChart is not just limited to desktop applications. It can be used on the server side such as servlets or JSPs too.

EasyMock, JUnit, and PowerMock

Unit tests are written by programmers to test classes or methods internally in programmer's perspective. Each test should be independent from each other and should be tested without any dependencies. How do we do unit testing in isolation without any dependencies? Mock objects are objects that mimic the real objects in controlled ways for different scenarios. They can help to decide if a test is either failed or passed. EasyMock is a framework that can save you time in hand wiring mock objects and can create mock objects at runtime. JUnit is a unit testing framework. JUnit and EasyMock can work together easily. PowerMock is a mock framework that extends other mock frameworks. PowerMock extends EasyMock with features such as mocking on private, static, or final methods. Currently, it supports EasyMock and Mockito.

Apache JMeter

Apache JMeter is a Java-based desktop application, which can be used for load testing to measure the performance of a system or used for stress testing to see if a system is crashed gracefully. JMeter provides a variety of test elements. They are quite handy and can save you time in writing your own Java programs for testing. Using JMeter is quite intuitive because it provides a nice GUI to create and run tests. You also can run JMeter tests in non-GUI mode. Tests can be run either locally or remotely.

JavaCC

JavaCC is an open source lexical analyzer generator and a parser generator for use with the Java applications, which takes a grammar specification (e.g., EBNF) and generates the Java source code of a lexical analyzer and a parser. A lexical analyzer breaks a sequence of characters into tokens and identifies the type of each token. A parser takes a sequence of tokens from a lexical analyzer and then analyzes them to determine the structure and generates output depending on your need.

Apache Solr

Apache Solr is an open source search platform based on Apache Lucene running as a standalone server. Solr provides features like full-text indexing, hit highlighting, faceted search, rich documents (e.g., PDF, MS Word) indexing and database integration. Solr

provides REST-like APIs which can be called over HTTP to make it easy to use. Solr allows customization through configuration and plugin architecture.

Apache James

Apache James Server is a Java-based mail server, which supports SMTP and POP3 protocols. Also, it can serve as an NNTP news server. Something special about James Server is that it provides a maillet container. Just like servlets are used to process HTTP requests for a servlet container. Maillets are used to process emails for a maillet container. Through configurations, you can use maillets to do complex email processing tasks. That is what makes James Server flexible and powerful. There are standard maillets provided by James Sever. Also, you can build your own maillets by using the Maillet API.

Jackson

JSON (JavaScript Object Notation) is based on the object notation from the JavaScript programming language. Just like XML, JSON is a format that is used for data storage and data exchange. But, the advantage of JSON is that you can use it in the JavaScript programs easily because a JSON string can be converted to a JavaScript object. A common use case is to use JSON data between back end and front end in web-based applications. Modern browsers have native support on JSON. In Java, you can use Jackson API to convert Java objects to and from JSON. The original purpose of Jackson API was for data binding on JSON data. Now, it also contains packages that can support formats such as XML, CSV.

Hibernate Validator

Input validations can happen at different places in applications. Custom and possible duplicate code can be anywhere in the applications. Not to mention they are usually part of logic in the applications. Hibernate Validator is a reference implementation of Bean Validation. Bean Validation (added as part of Java EE 6) is a framework that defines a metadata model and API for JavaBeans validation. Constraints on JavaBeans can be expressed via annotations (the default metadata model) and can be extended through XML constraint mappings. Bean Validation 1.1 allows put constraints to the parameters or return values on methods or constructors.

Apache Commons

The purpose of Apache Commons is to focus on all aspects of reusable Java components. Component interfaces will keep as stable as possible and try to be independent from other components. There are over 40 active components in the Apache Commons so far. The following are some of components grouped by categories:

Command line options: CLI

Core Java API: BeanUtils, Collections, IO, Lang, Math, Primitives

Database: DBCP, DbUtils

Encoding/decoding: Codec

Execution: Daemon, Exec, Launcher

File formats: Compress, Configuration, CSV

Logging: Logging

Network: Email, FileUpload, Net, VFS

XML: Digester, Jelly, XPath

Since there are so many components in the Apache Commons, only some of the popular components will be introduced here. You can check the home page of Apache Commons <http://commons.apache.org> to see a complete list of components.

COMMONS LANG

The Apache Commons Lang, which contains Java utility classes, is just like a standard Java API. It is being used in lots of projects. The Commons Lang provides additions to the `java.lang` package in the core Java API. Some implementations in the Commons Lang even are included in the core Java API now. One example is the Enum introduced in Java 5.

Starting from Lang 3.0 (Java 5 and above is required), the package name is different from previous versions (`org.apache.commons.lang3` vs. `org.apache.commons.lang`). The current stable version is 3.4 at the time of writing. You can download it from <http://commons.apache.org/proper/commons-lang>. The JAR file you need to include in the classpath is `commons-lang3-3.4.jar`.

StringUtils

Probably you have been using the following code snippet or something similar to check if a string is empty or not:

```
if(str == null || str.trim().length() == 0) {  
    ...  
} else {  
    ...  
}
```

It is just a few lines of code. But, it can be very trivial if the same code appears again and again in the application. Of course, a better way is to create a method and put it in a utility class. If you are working on a project with others or some legacy code, you need to find out if someone has created it or not and where it is. `StringUtils` is a utility class for string manipulations. It provides utility methods to check if a string contains any text or not.

They are quite handy:

`static boolean isEmpty(CharSequence cs)`: checks if a `CharSequence` is empty or null. It does not trim whitespace characters at both ends.

`static boolean.isBlank(CharSequence cs)`: checks if a `CharSequence` is empty or null. It does trim whitespace characters at both ends.

Starting from 3.0, the parameter is changed from `String` to `CharSequence`. `CharSequence` is an interface. `String`, `StringBuffer` and `StringBuilder` implement `CharSequence`.

Different from the `String` class, `StringUtils` handles the case of null. For a `String`, you need to check if it is null first or a `NullPointerException` might be thrown. Starting from Java 6, the `isEmpty()` method is added to the `String` class.

`StringUtils` has methods that are not available in the `String` class but only in the `Character` class. For example,

`static boolean isAlpha(CharSequence cs)`

`static boolean isAlphanumeric(CharSequence cs)`

`static boolean isNumeric(CharSequence cs)`

`String` has very limited support on ignoring case differences. `StringUtils` allows you to search or match a substring in case-insensitive way:

`static int indexOfIgnoreCase(CharSequence str, CharSequence searchStr)`: -1 is returned if no match or null in any one of input strings. If the search string is empty (“”), it is always matched and 0 is returned.

`static int indexOfIgnoreCase(CharSequence str, CharSequence searchStr, int startPos)`

`static int lastIndexOfIgnoreCase(CharSequence str, CharSequence searchStr)`

static int lastIndexOfIgnoreCase(CharSequence str, CharSequence searchStr, int startPos)

static boolean endsWithIgnoreCase(CharSequence str, CharSequence suffix): true is returned if both inputs are null. false is returned if one of them is null. If the suffix is empty (“”), it is always matched and true is returned

static boolean startsWithIgnoreCase(CharSequence str, CharSequence prefix)

Both String and StringUtils have split methods. In StringUtils, split method treats adjacent separators as one separator. For example,

```
StringUtils.split("a:b::c:d", ':') -> ["a", "b", "c", "d"]
```

```
"a:b::c:d".split(":") -> ["a", "b", "", "c", "d"]
```

If you need to join an array of objects into a String with separators between them, you can use:

static String join(Object[] array, char separator): null object or empty strings are treated as empty strings.

static String join(Object[] array, String separator)

Both String and StringUtils have trim methods. Both trim methods only remove unprintable control codes (ASCII code less than 32, e.g., line feed) and spaces (ASCII code 32) at both ends. If you want to remove all whitespace characters (including those with code > 32) at both ends, you can use StringUtils.strip(String str). If you want to have control on what should be removed at both ends, you can use StringUtils.strip(String str, String stripChars).

ArrayUtils

ArrayUtils is a utility class that provides operations on arrays including arrays of primitive data types and arrays of objects.

To add all elements in one array to another array or to add variable number of elements to one array, you can use:

`static int[] addAll(int[] array1, int... array2)`: When one of arrays is null, a new array with all elements of the non-null array is returned. When both arrays are null, null is returned. This method is overloaded to support all primitive data types. You can replace `int` with other primitive data types.

`static <T> T[] addAll(T[] array1, T... array2)`: Since `array2` is `varargs`, it can be either an array or a sequence of arguments. If `array1` is null and `array2` is a null array (e.g., `String[] array2 = null`), null is returned. If `array1` is null and `array2` is a null object (e.g., `String array2 = null`), `[null]` is returned.

To convert an array containing elements of certain primitive data type to an array of objects, you can use:

`static Integer[] toObject(int[] array)`: null is returned if the input is null. This method is overloaded to support all primitive data types.

To convert an array of objects to the corresponding primitive data type, you can use:

`static int[] toPrimitive(Integer[] array)`: null is returned if the input is null. If the array contains any null element, a `NullPointerException` is thrown. This method is overloaded to support all primitive data types.

The following example demonstrates how to use methods mentioned above:

```
import java.util.Arrays;
```

```
import java.util.List;
```

```
import org.apache.commons.lang3.ArrayUtils;
```

```
public class ArrayUtilsExample {  
    public static void main(String[] args) {  
        // add one array to another array  
        int[] array1 = new int[] {1, 2, 3};  
        int[] array2 = new int[] {4, 5, 6};  
        int[] newArray = ArrayUtils.addAll(array1, array2);  
        System.out.println(Arrays.toString(newArray));  
    }  
}
```

```
// add elements to an array
String[] strArray1 = {"a", "b", "c"};
String[] newStrArray = ArrayUtils.<String>addAll(strArray1, "d", "e", "f");
System.out.println(Arrays.toString(newStrArray));
// convert int[] to Integer[]
int[] ids = new int[]{1001, 1002, 1003};
Integer[] intObjs = ArrayUtils.toObject(ids);
System.out.println(Arrays.toString(intObjs));
// convert Integer[] to int[]
List<Integer> idList = Arrays.asList(1001, 1002, 1003);
ids = ArrayUtils.toPrimitive(idList.toArray(new Integer[idList.size()]));
System.out.println(Arrays.toString(ids));
}
}
```

The following is the output:

```
[1, 2, 3, 4, 5, 6]
```

```
[a, b, c, d, e, f]
```

```
[1001, 1002, 1003]
```

```
[1001, 1002, 1003]
```

ExceptionUtils

The core Java API does not provide a method to get the stack trace as a String. If you need to log such information, you need to use the following code to write the stack trace to a String:

```
Writer stringWriter = new StringWriter();  
PrintWriter printWriter = new PrintWriter(stringWriter);  
ex.printStackTrace(printWriter);  
String stackTrace = stringWriter.toString();
```

Instead, you can use `ExceptionUtils.getStackTrace(Throwable throwable)`. It returns a String of stack trace generated by the `printStackTrace(PrintWriter s)` method.

WordUtils

In the case that you need to wrap a long line of text, you can use:

```
static String wrap(String str, int wrapLength, String newLineStr, boolean wrapLongWords): null is returned if str is null. If newLineStr is null, the default line separator is used. Leading spaces are stripped. But, trailing spaces are not.
```

For example, in an HTML page, you can wrap a long line by using `
` as the line separator. Instead of wrapping it, you can also use `StringUtils.abbreviate(String str, int maxWidth)` to abbreviate it with an ellipsis (...) in the end:

```
import org.apache.commons.lang3.StringUtils;
import org.apache.commons.lang3.text.WordUtils;
```

```
public class WordUtilsExample {
    public static void main(String[] args) {
        // wrap a single line
        String str = "Starting from Lang 3.0 (Java 5 and above is required), " +
            "the package name is different from previous versions.";
        String wrappedStr = WordUtils.wrap(str, 40, "<br/>", true);
        System.out.println(wrappedStr);
        // abbreviate a string using an ellipsis
        System.out.println(StringUtils.abbreviate(str, 20));
    }
}
```

The following is the output:

```
Starting from Lang 3.0 (Java 5 and above<br/>is required), the package name
is<br/>different from previous versions.
```

```
Starting from Lan...
```


COMMONS IO

The Apache Commons IO provides utility classes for common tasks such as reading and writing through input and output streams, and file and filename manipulations. The current stable version is 2.4 (Java 6 and above is required) at the time of writing. You can download it from <http://commons.apache.org/proper/commons-io>. The JAR file you need to include in the classpath is commons-io-2.4.jar.

IOUtils

IOUtils is a utility class that provides static methods for IO stream operations such as read, write, copy and close.

For IO stream operations, a good practice is to close opened streams at the finally block. To close a stream, you need to use another try-catch block such as:

```
try {  
    ...  
} catch(...) {  
    ...  
} finally {  
    try {  
        in.close();  
    } catch (Exception ex) {}  
}
```

Or, you can use try-with-resources statement introduced in the Java 7.

```
try (  
    resource 1;  
    resources 2;  
    ...  
) {  
    ...  
} catch(...) {  
    ...  
}
```

Any objects that implement the AutoCloseable interface can be used in the try-with-resources statement.

Instead, you can use IOUtils.closeQuietly(in) to close it without using another try-catch block. closeQuietly() closes a stream unconditionally even it is null or closed. It has overloaded methods that support InputStream, OutputStream, Reader, Writer, Socket, ServerSocket, Selector and Closeable.

For better performance, all methods in this class that read a stream using a buffered stream (BufferedInputStream or BufferedReader) internally. There is no reason to use a buffered stream again.

To copy the content from one stream to another stream, you can use `copy(InputStream input, OutputStream output)` or `copy(Reader reader, Writer writer)`. You also can copy from a binary stream to a character stream, and vice versa. For a large input stream (over 2GB), you can use `copyLarge` method.

If you need to get the content of a stream (`InputStream` or `Reader`) as a `String`, you can use `toString(InputStream input, String encoding)` or `toString(Reader reader)`. Other than a stream, you also allow to use an URI or URL to get certain resource from the network. For example, `toString(URL url, String encoding)`.

FileUtils

FileUtils is a utility class that provides static methods for file (or directory) manipulations in platform neutral way. FileUtils provides additions to the File class.

To copy a whole directory (including all subdirectories) to a new location, you can use:

```
static void copyDirectory(File srcDir, File destDir, FileFilter filter, boolean
preserveFileDate)
```

A new directory is created if it does not exist. Optionally, you can specify to preserve the original file date. Also, you can use a file filter to define files or directories should be copied. If it is null, all directories and files are copied. To copy one directory as a child of another directory, you can use `copyDirectoryToDirectory(File srcDir, File destDir)`. A new directory is created if it does not exist. If you need to move a directory, you can use `moveDirectory` or `moveDirectoryToDirectory`.

The Common IO supplies many common filter classes by implementing `IOFileFilter` interface. To filter files based on the filename, you can use `SuffixFilter`, `PrefixFileFilter`, `NameFileFilter`, or `RegexFileFilter`. To filter files based on date, you can use `AgeFileFilter`. To combine filters together for conditional operations, you can use `AndFileFilter`, `OrFileFilter`, or `NotFileFilter`. Also, you can use `FileFilterUtils`, which is a utility class that provides access to all filter implementations, to create a filter without knowing the class name of filter.

The following example demonstrates how to copy a directory by using a filter. All CVS and empty directories, and `.cvsignore` are not copied:

```
import java.io.File;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.filefilter.AndFileFilter;
import org.apache.commons.io.filefilter.DirectoryFileFilter;
import org.apache.commons.io.filefilter.EmptyFileFilter;
import org.apache.commons.io.filefilter.FileFileFilter;
import org.apache.commons.io.filefilter.FileFilterUtils;
import org.apache.commons.io.filefilter.IOFileFilter;
import org.apache.commons.io.filefilter.NameFileFilter;
import org.apache.commons.io.filefilter.NotFileFilter;

public class FileFilterExample {
    public static void main(String[] args) {
```

```

try {
File src = new File("apps");
File dest = new File("dest");
IOFileFilter notCvsFilter1 = new NotFileFilter(
new AndFileFilter(DirectoryFileFilter.INSTANCE, new NameFileFilter("CVS")));
IOFileFilter notCvsFilter2 = new NotFileFilter(
new AndFileFilter(FileFileFilter.FILE, new NameFileFilter(".cvsignore")));
// ignore CVS, .cvsignore and empty directory
IOFileFilter filter = FileFilterUtils.and(notCvsFilter1, notCvsFilter2,
EmptyFileFilter.NOT_EMPTY);
FileUtils.copyDirectory(src, dest, filter);
} catch(Exception ex) {
System.out.println(ex);
}
}

```

To copy a file to a new location, you can use:

```
static void copyFile(File srcFile, File destFile, boolean preserveFileDate)
```

Optionally, you can specify to preserve the original file date. To copy a file to a directory, you can use `copyFileToDirectory`. A new directory is created if it does not exist. If you need to move a file, you can use `moveFile` or `moveFileToDirectory`.

To read the content of a file to a String, you can use `readFileToString(File file, String encoding)`. To write a String to a file, you can use `writeStringToFile(File file, String data, String encoding, boolean append)`. Optionally, you can choose to overwrite or append an existing file. A new file is created if it does not exist.

For file or directory deletions, `File.delete()` does not delete a directory recursively and the directory needs to be empty. `FileUtils.deleteDirectory(File directory)` allows to delete a directory recursively. `FileUtils.deleteQuietly(File file)` can delete a file or a directory and all of its subdirectories. It never throws an exception if it cannot be deleted.

If you need to traverse a directory structure, you can use:

```
static Iterator<File> iterateFiles(File dir, IOFileFilter fileFilter, IOFileFilter dirFilter):
```

The returned iterator only includes the files. If `dirFilter` is null, subdirectories are not searched.

```
static Iterator<File> iterateFilesAndDirs(File dir, IOFileFilter fileFilter, IOFileFilter dirFilter):
```

The returned iterator includes the files and directories.

The following example traverses a directory to search for classes of Apache Commons


```
while(it.hasNext()) {  
File file = it.next();  
example.find(file, map);  
}
```

```
System.out.println(map);  
}  
} catch(Exception ex) {  
System.out.println(ex);  
}  
}
```

```
public void find(File file, Map<String, Integer> map) {  
FileInputStream in = null;  
try {  
in = new FileInputStream(file);  
LineIterator it = IOUtils.lineIterator(in, "UTF-8");  
while(it.hasNext()) {  
String line = it.nextLine();  
if(!StringUtils.isBlank(line)) {  
Matcher matcher = classPattern.matcher(line);  
if(matcher.find()) {  
break;  
}  
matcher = importPattern.matcher(line);  
if(matcher.find()) {  
String className = matcher.group(2);  
if(map.containsKey(className)) {  
Integer count = map.get(className);  
map.put(className, count + 1);  
} else {  
map.put(className, 1);  
}
```

```
}  
}  
}  
}  
} catch(IOException ex) {  
System.out.println(ex);  
} finally {  
IOUtils.closeQuietly(in);  
}  
}  
}
```


FilenameUtils

FilenameUtils is a utility class that provides static methods for filename and file path manipulations in platform neutral way. FilenameUtils breaks a filename into six components. You can access them from the following methods:

```
static String getFullPath(String filename)
```

```
static String getPath(String filename)
```

```
static String getPrefix(String filename)
```

```
static String getName(String filename)
```

```
static String getBaseName(String filename)
```

```
static String getExtension(String filename)
```

Take, C:\Commons\examples\commons-io\FilenameUtilsExample.java, as an example, you can get the following output:

```
Full path: C:\Commons\examples\commons-io\
```

```
Path: Commons\examples\commons-io\
```

```
Prefix: C:\
```

```
Name: FilenameUtilsExample.java
```

```
Base name: FilenameUtilsExample
```

```
Extension: java
```

COMMONS CSV

The Apache Commons CSV provides API to read (parse) and write files in CSV (comma separated value) format. The current stable version of Commons CSV is 1.2 at the time of writing. You can download it from <http://commons.apache.org/proper/commons-csv>. The JAR file you need to include in the classpath is commons-csv-1.2.jar.

CSVParser

To parse a file in CSV format, you can use CSVParser:

```
CSVParser(Reader reader, CSVFormat format)
```

CSVFormat specifies the format of a CSV file to be parsed. There are several pre-defined formats, such as CSVFormat.DEFAULT, CSVFormat.RFC4180. There is no standard for CSV format. But, a popular standard (MIME type, text/csv) is defined in RFC4180, <https://tools.ietf.org/html/rfc4180>. CSVFormat.DEFAULT is based on CSVFormat.RFC4180, but it allows empty lines. This format uses the following settings in parsing:

Adjacent fields are separated by a comma

Fields are quoted by double-quote characters

Empty lines are skipped

You can extend a format by using withXXX methods.

For a file that has a header row (the first record), you can use:

```
CSVFormat csvFormat = CSVFormat.DEFAULT.withHeader();
```

Values at the first record are used to get column names automatically. Or, you can manually define column names to use to access records, such as:

```
CSVFormat csvFormat = CSVFormat.DEFAULT.withHeader(new String[]{"id", "name", "comment"});
```

And, the first record should be skipped by calling

```
withSkipHeaderRecord(true)
```

The following example parses a file using CSVFormat.DEFAULT:

```
import java.io.File;
```

```
import java.io.FileReader;
```

```
import java.util.List;
```

```
import org.apache.commons.csv.CSVFormat;
```

```
import org.apache.commons.csv.CSVParser;
```

```
import org.apache.commons.csv.CSVRecord;
```

```
import org.apache.commons.io.IOUtils;
```

```
public class CSVParserExample1 {
```

```

private static final String COL1 = "id";
private static final String COL2 = "name";
private static final String COL3 = "comment";

public static void main(String[] args) {
    FileReader reader = null;
    CSVParser parser = null;
    try {
        File csvFile = new File("example1.csv");
        reader = new FileReader(csvFile);
        int startRow = 0;
        CSVFormat csvFormat = CSVFormat.DEFAULT.withHeader();

        parser = new CSVParser(reader, csvFormat);
        List<CSVRecord> csvRecords = parser.getRecords();
        for(int i = startRow; i < csvRecords.size(); i++) {
            CSVRecord record = csvRecords.get(i);
            System.out.println(record.get(COL1) + ": " + record.get(COL3));
        }
        } catch(Exception ex) {
        System.out.println(ex);
        } finally {
        IOUtils.closeQuietly(reader);
        IOUtils.closeQuietly(parser);
        }
    }
}

```

The following is a sample CSV file:

"id","name","comment"

"1","John","This is a simple one."

"2","Joel","This is a comment with ,"

"3","Joel","This is a comment with ""quotes"""

“4”,“Eric”,“This is a comment with multiple lines.”

The following is the output:

1: This is a simple one.

2: This is a comment with ,

3: This is a comment with “quotes”

4: This is a comment with multiple lines.

As you can see from this example, fields containing commas, double quotes, or line breaks need to be quoted.

To parse a string, you can use the following factory method:

```
CSVParser.parse(String string, CSVFormat format)
```

COMMONS CODEC

The Apache Commons Codec provides common encoders and decoders such as Base64, message digest, and GNU C library crypt() compatible hash function. The current stable version of Commons Codec is 1.10 at the time of writing. You can download it from <http://commons.apache.org/proper/commons-codec>. The JAR file you need to include in the classpath is commons-codec-1.10.jar.

DigestUtils

Message digests are created from one-way cryptographic hash functions that take arbitrary length of input (called message) and produce a fixed length hash value (called digest). Because hash values are hard to invert to the original input data and collision resistance, they can be used in verifying integrity of files or password verification. MD5 (128-bit hash value) and SHA-1 (160 bits) are considered less secure. It's recommended to use SHA-2. SHA-256 (256 bits), SHA-384 (384 bits), and SHA-512 (512 bits) are part of SHA-2 family.

In the following example, a secure key that contains user name, password, and date information is generated using SHA-256 algorithm:

```
import java.util.Calendar;
import org.apache.commons.codec.binary.Base64;
import org.apache.commons.codec.digest.DigestUtils;

public class DigestUtilsExample {
    public static void main(String[] args) {
        Calendar cal = Calendar.getInstance();
        String user = "john111";
        String pwd = "secret";
        String key = user + pwd + cal.get(Calendar.DAY_OF_MONTH) +
(cal.get(Calendar.MONTH) + 1) + cal.get(Calendar.YEAR);
        byte[] sha256 = DigestUtils.sha256(key);
        System.out.println(sha256.length);
        String sha256hex = DigestUtils.sha256Hex(key);
        System.out.println(sha256hex);
        String base64 = Base64.encodeBase64URLSafeString(sha256);
        System.out.println(base64);
    }
}
```

The following is the output:

32

be9c5a9ffa3c65b22b0b6d6064f2aaf71973e2a62e3d687f15f3eda37126f326

vpxan_o8ZbIrC21gZPKq9xlz4qYuPWh_FfPto3Em8yY

Because SHA-256 produces 32-byte hash value, the array size is 32. The second line is SHA-256 digest in hexadecimal value. Also, you can use Base64 class to encode hash values using URL-safe Base64 algorithm, as shown at the last line.

To store sensitive information such as password, hash values are not completely safe. The same type of hash function produces the same hash values for the same input data. They can be cracked by using pre-computed hash values (lookup tables) of possible passwords. To further protect hash values, a string, called a salt, is added to the data to be hashed. So, even the same password using the same hash function, a different hash value is created if using different salt. This will make it impossible to create hash values for every possible salt with a long salt. Certainly, using the same salt on every hash is not a good idea. Ideally, a random salt is generated every time. At least, a salt should come from certain unique information from each account.

COMMONS NET

The Apache Commons Net provides implementations for client side access on many internet protocols such as FTP(S), Telnet and SMTP(S)/POP3(S)/IMAP(S). HTTPclient is not included here. It was in the Commons HttpClient and has been replaced by Apache HttpComponents. The current stable version of Commons Net is 3.2 (Java 5 and above is required) at the time of writing. You can download it from <http://commons.apache.org/proper/commons-net>. The JAR file you need to include in the classpath is commons-net-3.2.jar.

FTPClient

FTPClient extends the FTP class. The FTP class provides basic functionality for an FTP client. To use it, you need to have better understanding of the FTP protocol. The FTPClient class provides higher level of functionality. It is easier to use. To use the FTPClient class to create an FTP client, the following are some basic steps:

Step 1

First, you need to create an instance of FTPClient by using the default constructor. Then, you can use the connect(String hostname) or connect(String hostname, int port) method to connect to the FTP server. Immediately after connecting, you can check the reply code by using the getReplyCode() method and use the FTPReply.isPositiveCompletion(int reply) method to verify if the reply code is a positive completion response or not. Similarly, you can always check reply code right after the command you have sent to the server.

Step 2

Use the login(String username, String password) method to login the FTP server. The return value is a boolean, false is returned if failed.

Step 3

Use the enterLocalPassiveMode() method to set the current data connection mode to local passive mode if necessary. The local passive mode is used in a situation where the client is behind a firewall and unable to accept incoming TCP connections. By default, it is local active mode.

Also, you can use the setFileType(int fileType) method to set the file type (FTP.ASCII_FILE_TYPE or FTP.BINARY_FILE_TYPE) for the file to be transferred. By default, the file type is ASCII.

Step 4

Now, you can start sending commands to the server.

Step 5

Once you are done, use the logout() method to log out and use the disconnect() method to disconnect from the server.

The following example demonstrates how to use the FTPclient class to upload a file, list files and download the same file with a new filename:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
```

```
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPFile;
import org.apache.commons.net.ftp.FTPReply;

public class FTPClientExample {

    public static void main(String[] args) {
        if(args.length != 3) {
            System.out.println("usage: FTPClientExample host username password");
            System.exit(1);
        }
        String host = args[0];
        String username = args[1];
        String password = args[2];
        FTPClientExample example = new FTPClientExample();
        FTPClient ftp = new FTPClient();
        try {
            ftp.connect(host);
            if(!example.isOk(ftp)) {
                System.exit(1);
            }
            ftp.login(username, password);
            if(example.isOk(ftp)) {
                ftp.enterLocalPassiveMode();
                // upload a file
                String filename = "FTPClientExample.java";
                InputStream input = new FileInputStream(new File(filename));
                ftp.storeFile(filename, input);
                example.isOk(ftp);
                // list files
                FTPFile[] files = ftp.listFiles();
                if(example.isOk(ftp)) {
```

```

for(FTPFile file : files) {
System.out.println(file.getRawListing());
}
}
// download a file
ftp.retrieveFile(filename, new FileOutputStream(new File(filename + ".new")));
example.isOk(ftp);
ftp.logout();
example.isOk(ftp);
}
} catch(Exception ex) {
System.out.println(ex);
} finally {
if(ftp.isConnected()) {
try {
ftp.disconnect();
} catch(Exception ex) {}
}
}
}

public boolean isOk(FTPClient ftp) {
boolean ok = true;

int replyCode = ftp.getReplyCode();
if(!FTPReply.isPositiveCompletion(replyCode))
ok = false;
System.out.println("ftp reply=" + ftp.getReplyString());

return ok;
}
}

```

The following are responses from the FTP server:

ftp reply=220 ProFTPD 1.3.3c Server (ProFTPD)

ftp reply=230 User joelchou logged in

ftp reply=226 Transfer complete

ftp reply=226 Transfer complete

-rw-r—r— 1 joelchou joelchou 2322 Apr 29 23:34 FTPClientExample.java

ftp reply=226 Transfer complete

ftp reply=221 Goodbye.

COMMONS COMPRESS

The Apache Commons Compress provides API for many archive file formats. There are three types of archive file formats: archiving only, compression only, archiving and compression. In the Commons Compress, archiving is through archivers. Archivers deal with archives containing structured contents. The base classes for archivers are `ArchiveInputStream` and `ArchiveOutputStream`. An archive contains entries which are represented by the `ArchiveEntry` interface. The file formats ar, cpio, dump, tar and zip are supported as archivers. Even though zip is treated as an archiver in the Commons Compress, it can be compressed too. Compression is through compressors. Compressors compress a single entry. The base classes for compressors are `CompressInputStream` and `CompressOutputStream`. The file formats bzip2, Pack200, XZ and gzip are supported as compressors.

The current stable version of Commons Compress is 1.5 (Java 5 and above is required) at the time of writing. You can download it from <http://commons.apache.org/proper/commons-compress>. The JAR file you need to include in the classpath is `commons-compress-1.5.jar`.

Zip

The `ZipArchiveOutputStream` class is to fix an issue that has been around for a while on `java.util.zip.ZipOutputStream`. When a file with some non-ASCII characters (e.g., Chinese) in the filename is archived as zip format, the unzipped filename is not the same as the original one. It contains some weird characters. This problem is not fixed until Java 7.

The following example demonstrates how to use `ZipArchiveOutputStream` to archive files in zip format:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Collection;

import org.apache.commons.compress.archivers.zip.ZipArchiveEntry;
import org.apache.commons.compress.archivers.zip.ZipArchiveOutputStream;
import org.apache.commons.io.FileUtils;
import org.apache.commons.io.IOUtils;

public class ZipExample {
    public static void main(String[] args) {
        ZipArchiveOutputStream zipOut = null;
        try {
            zipOut =
                new ZipArchiveOutputStream(new File("zipExample.zip"));
            Collection<File> files = FileUtils.listFiles(new File("."),
                new String[]{"java", "class", "txt"}, false);
            for(File file : files) {
                ZipArchiveEntry entry = new ZipArchiveEntry(file, file.getName());
                zipOut.putArchiveEntry(entry);
                InputStream entryIn = null;
                try {
                    entryIn = new FileInputStream(file);
                    IOUtils.copy(entryIn, zipOut);
```

```

zipOut.closeArchiveEntry();
System.out.println("file: " + entry.getName() + " size: " +
entry.getSize() + " compressed size: " + entry.getCompressedSize());
} finally {
IOUtils.closeQuietly(entryIn);
}
}
zipOut.finish();
zipOut.close();
} catch(Exception ex) {
System.out.println(ex);
}
}
}

```

The following example demonstrates how to use the ZipFile class to unzip the zip file created in the previous example:

```

import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;

import org.apache.commons.compress.archivers.zip.ZipArchiveEntry;
import org.apache.commons.compress.archivers.zip.ZipFile;
import org.apache.commons.io.IOUtils;

public class ZipFileExample {
    public static void main(String[] args) {
        ZipFile zipFile = null;
        try {
            File baseDir = new File("unzipped");
            if(!baseDir.exists())

```



```
if(!baseDir.mkdir())
System.exit(1);
zipFile =
new ZipFile(new File("zipExample.zip"));
Enumeration<ZipArchiveEntry> entries = zipFile.getEntries();
while(entries.hasMoreElements()) {
ZipArchiveEntry entry = entries.nextElement();
System.out.println("filename: " + entry.getName());
InputStream entryIn = null;
OutputStream output = null;
try {
entryIn = zipFile.getInputStream(entry);
output = new FileOutputStream(new File(baseDir, entry.getName()));
IOUtils.copy(entryIn, output);
} finally {
IOUtils.closeQuietly(entryIn);
IOUtils.closeQuietly(output);
}
} catch(Exception ex) {
System.out.println(ex);
} finally {
try {
zipFile.close();
} catch(Exception ex) {}
}
}
```

COMMONS CLI

The Apache Commons CLI provides API for creating command line options and parsing command line options passed to command line applications. It also provides means to generate help and usage information.

The current stable version of Commons CLI is 1.2 at the time of writing. You can download it from <http://commons.apache.org/proper/commons-cli>. The JAR file you need to include in the classpath is commons-cli-1.2.jar.

To use the Apache Commons CLI to add command line options to an application, basically it involves two steps:

Step 1

The first step is to create command line options. It starts with an Options object which represents a collection of options in an application. An option is represented by the Option class. An Option is created by the following constructor:

```
Option(String opt, String description)
```

```
Option(String opt, boolean hasArg, String description)
```

```
Option(String opt, String longOpt, boolean hasArg, String description)
```

For example, to create a Boolean option, which is an option without any argument representing either true or false, with a short option and a long option such as -h, —help, you can use:

```
Option helpOption = new Option("h", "help", false, "print this help");
```

To add an Option to the Options, you can use the addOption(Option option) method.

Other than using the Option class, you can use the OptionBuilder class to create options. Since the OptionBuilder class allows method chaining, you can use it to create more complicated options in descriptive way. For example, to create a required argument option such as -H,—host <host>, you can use:

```
Option hostOption =  
OptionBuilder.hasArgs().withArgName("host").withLongOpt("host").withDescription("ho
```

If you need to create a Java property option such as -D<property>=<value>, you can use:

```
Option propertyOption =  
OptionBuilder.hasArgs(2).withValueSeparator().withArgName("property=value").withDes  
property").isRequired().create("D");
```

hasArgs(2) means there are two arguments. withValueSeparator() means there are separators between arguments and the default separator is =.

If you need to create a group of mutually exclusive options, you can create an OptionGroup and use the addOption(Option option) method to add options into an option

group. To add an OptionGroup to an Options, you can use the addOptionGroup(OptionGroup group) method.

Step 2

Once options are created, you can use a command line parser to parse command line arguments. For example, to use a BasicParser to parse command line arguments:

```
CommandLineParser parser = new BasicParser();
```

```
CommandLine cmd = parser.parse(options, args);
```

Once the command line arguments are parsed, a CommandLine object representing a list of arguments parsed against options is returned. For a Boolean option, you can use the hasOption(String opt) method to see if an option is found or not. For an argument option, you can use:

```
String getOptionValue(String opt)
```

```
String getOptionValue(String opt, String defaultValue)
```

For a Java property option, you can use:

```
Properties getOptionProperties(String opt)
```

A Properties, which is a map of name-value pairs, is returned.

The following example demonstrates how to use the Commons CLI to create command line options as shown below:

```
usage: FTPClient <options>
```

Possible options:

```
-H,--host <host>      hostname
```

```
-h,--help             print this help
```

```
-P,--password <password> password
```

```
-U,--username <username> username
```

```
-v,--verbose         enable verbose output
```

In this example, there are two sets of options and one parser for each set of options. That is because one set of options precedes the other set of options. Here, -h option precedes the other options. There is no reason to parse other arguments once the help option appears in the arguments. That is why the last argument is set as true:

```
CommandLine cmd = parser.parse(options, args, true);
```

It means it will stop parsing once a non-option argument is encountered. If you need to get a list of non-option arguments, you can use getArgList() or getArgs() with a List or a String[] returned respectively.

```
import org.apache.commons.cli.BasicParser;
```

```
import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.HelpFormatter;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;

public class CLIExample {
    private static String OPT_HELP = "h";
    private static String OPT_VERBOSE = "v";
    private static String OPT_HOST = "H";
    private static String OPT_USERNAME = "U";
    private static String OPT_PASSWORD = "P";

    private boolean verbose = false;
    private String host;
    private String username;
    private String password;

    public static void main(String[] args) {
        CLIExample example = new CLIExample();
        // create command line options
        Options helpOptions = new Options();
        Options options1 = example.createOptions1(helpOptions);
        Options options2 = example.createOptions2(helpOptions);
        // parse command line arguments
        if(args == null || args.length == 0) {
            example.printHelp(helpOptions);
            System.exit(0);
        } else {
            try {
```

```
if(!example.parseArguments1(options1, helpOptions, args)) {
example.parseArguments2(options2, args);
System.out.println("host: " + example.host);
System.out.println("username: " + example.username);
System.out.println("password: " + example.password);
System.out.println("verbose: " + example.verbose);
}
} catch(ParseException parseEx) {
System.out.println(parseEx);
System.exit(1);
}
}
}
```

```
// options precede others
```

```
public Options createOptions1(Options helpOptions) {
Options options = new Options();
Option helpOption = new Option(OPT_HELP, "help", false, "print this help");

options.addOption(helpOption);
helpOptions.addOption(helpOption);

return options;
}
```

```
public Options createOptions2(Options helpOptions) {
Options options = new Options();
Option verboseOption = new Option(OPT_VERBOSE, "verbose", false,
"enable verbose output");
Option hostOption = OptionBuilder.hasArgs().withArgName("host")
.withLongOpt("host").withDescription("hostname").isRequired().create(OPT_HOST);
Option usernameOption = OptionBuilder.hasArgs().withArgName("username")
```

```

.withLongOpt("username").withDescription("username").isRequired().create(OPT_USI
Option passwordOption = OptionBuilder.hasArgs().withArgName("password")
.withLongOpt("password").withDescription("password").isRequired().create(OPT_PAS
options.addOption(verboseOption);
options.addOption(hostOption);
options.addOption(usernameOption);
options.addOption(passwordOption);
helpOptions.addOption(verboseOption);
helpOptions.addOption(hostOption);
helpOptions.addOption(usernameOption);
helpOptions.addOption(passwordOption);

return options;
}

```

```

public void printHelp(Options options) {
HelpFormatter formatter = new HelpFormatter();
formatter.printHelp("FTPClient <options>", "Possible options:",
options, null, false);
}

```

```

public boolean parseArguments1(Options options, Options helpOptions, String[] args)
throws ParseException {
CommandLineParser parser = new BasicParser();
CommandLine cmd = parser.parse(options, args, true);
boolean hasOption = false;
if(cmd.getOptions().length > 0) {
// if there is any option here, don't need to parse the 2nd set
hasOption = true;
if(cmd.hasOption(OPT_HELP))
printHelp(helpOptions);
}
}

```

```
return hasOption;  
}
```

```
public void parseArguments2(Options options, String[] args)  
throws ParseException {  
    CommandLineParser parser = new BasicParser();  
    CommandLine cmd = parser.parse(options, args);  
    verbose = cmd.hasOption(OPT_VERBOSE);  
    if(cmd.hasOption(OPT_HOST))  
        host = cmd.getOptionValue(OPT_HOST);  
    if(cmd.hasOption(OPT_USERNAME))  
        username = cmd.getOptionValue(OPT_USERNAME);  
    if(cmd.hasOption(OPT_PASSWORD))  
        password = cmd.getOptionValue(OPT_PASSWORD);  
    }  
}
```

COMMONS FILEUPLOAD

The Apache Commons FileUpload provides file upload capability to web applications using servlets. The Commons FileUpload handles HTTP requests from the form-based file upload HTML pages (with enctype="multipart/form-data" in the form).

The current stable version of Commons FileUpload is 1.3 at the time of writing. You can download it from <http://commons.apache.org/proper/commons-fileupload/>. The JAR files you need to include in the classpath are commons-fileupload-1.3.jar and commons-io-2.4.jar.

To use the Apache Commons FileUpload in a servlet of your web application, it involves the following steps:

Step 1

First, you can verify if this request contains multipart content by using:

`boolean ServletFileUpload.isMultipartContent(HttpServletRequest request)`: true if this contains multipart content.

Step 2

Next, you create a `DiskFileItemFactory`. This is a factory that maintains `FileItem` objects. The `FileItem` class represents a file or form field received from a request with multipart content. The `DiskFileItemFactory` keeps `FileItem` objects either in the memory or in a temporary file on disk. You can use the `setSizeThreshold(int threshold)` method to define a size threshold at which a file should be stored on disk. The default value is 10KB.

Also, you can use the `setRepository(File repository)` method to define the location where temporary files should be stored. For example, you can use the `ServletContext` attribute `javax.servlet.context.tempdir` to get the temporary directory of the web application. In Apache Tomcat server, the temporary directory for web application upload-example is something like `\work\Catalina\localhost\upload-example` under a Tomcat instance.

Step 3

To handle file upload requests, you need to create a file upload handler by using `ServletFileUpload(FileItemFactory factory)` to construct an instance of `ServletFileUpload` with the `DiskFileItemFactory` created at previous step. To define the maximum allowed file size, you can use the `setSizeMax(long sizeMax)` method.

To parse a request with multipart content, you can use:

`List<FileItem> parseRequest(HttpServletRequest request)`: a list of `FileItem` objects are returned in the order they were transmitted.

In the case of uploading large files, you can use `setProgressListener(ProgressListener listener)` to have a progress listener registered to the `ServletFileUpload`. You create a progress listener by implementing the `ProgressListener` interface. It has only one method:

public void update(long pBytesRead, long pContentLength, int pItems): pBytesRead indicates the total number of bytes been read so far. pContentLength is the total number of bytes to be uploaded. pItems indicates the current item being read (starting from 1). 0 indicates no item so far.

Step 4

Now, you can write uploaded files to a destination by using the write(File file) method provided by the FileItem class. It simply writes a FileItem to a file and does not care if it is in the memory or a temporary location. If you want to write the content to a stream, you can use getInputStream() to get an InputStream.

Once a FileItem is written to a destination, you can choose to use the delete() method to delete it earlier to save resources.

In the following example, there are two servlets: FileUploadServlet and UploadProgressServlet. The FileUploadServlet handle requests for uploading files. A progress handler, the UploadProgressListener, is registered and set as an attribute of the user session. After all file items are uploaded and transmitted to the destination, an HTML response is generated. The following is the FileUploadServlet.java:

```
package servlet;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Iterator;
import java.util.List;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.apache.commons.fileupload.FileItem;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;

public class FileUploadServlet extends HttpServlet {
    // size threshold to be written directly to disk
```

```

private static final int SIZE_THRESHOLD = 10*1024; // 1024K
// max file size
private static final long MAX_FILE_SIZE = 50*1024*1024; // 50M
// uploaded directory
private static final String UPLOADED_DIR = "files";

public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>An example for Apache Commons FileUpload</title>");
out.println("</head>");
out.println("<body>");

if(ServletFileUpload.isMultipartContent(request)) {
try {
// create a factory for disk-based file items
DiskFileItemFactory factory = new DiskFileItemFactory();
File repository =
(File)getContext().getAttribute("javax.servlet.context.tempdir");
factory.setSizeThreshold(SIZE_THRESHOLD);
factory.setRepository(repository); // location to store temporary files
// create a new file upload handler
ServletFileUpload upload = new ServletFileUpload(factory);
// progress listener
HttpSession session = request.getSession();
UploadProgressListener pListener = new UploadProgressListener();
upload.setProgressListener(pListener);
session.setAttribute("Progress", pListener);
// set max file size

```

```
upload.setFileSizeMax(MAX_FILE_SIZE);
// parse the request
List<FileItem> items = upload.parseRequest(request);
// process items
Iterator<FileItem> it = items.iterator();
File uploadedDir = new File(getServletContext().getRealPath(UPLOADED_DIR));
while(it.hasNext()) {
FileItem item = it.next();
if(!item.isFormField()) {
// process uploaded file
out.println("<p>Filename: " + item.getName() + "<br/>");
out.println("File Size: " + item.getSize() + " bytes</p>");
try {
item.write(new File(uploadedDir, item.getName()));
} catch(Exception ex) {
out.println(ex);
} finally {
// make sure the temporary file is deleted
item.delete();
}
} else {
// process regular form fields
}
}
} catch(Exception ex) {
out.println(ex);
}
} else {
out.println("This request does not contain multipart content.");
}

out.println("</body>");
```

```
out.println("</html>");
out.close();
}
```

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
doGet(request, response);
}
}
```

The UploadProgressListener provides current progress of uploading. The following is the ProgressListener.java:

```
package servlet;
```

```
import org.apache.commons.fileupload.ProgressListener;
```

```
public class UploadProgressListener implements ProgressListener {
    private long bytesRead;
    private long contentLength;
    private int itemNo;

    public void update(long pBytesRead, long pContentLength, int pItems) {
        bytesRead = pBytesRead;
        contentLength = pContentLength;
        itemNo = pItems;
    }

    public long getBytesRead() {
        return bytesRead;
    }

    public long getContentLength() {
        return contentLength;
    }
}
```

```

    }

    public int getItemNo() {
        return itemNo;
    }
}

```

The UploadProgressServlet handles requests for progress checking from Ajax by using the information provided by the UploadProgressListener. For each request, an XML response is generated and returned. The following is a sample message:

```

<progress>
<message>Uploading item #1... 100 percent</message>
<completed>>true</completed>
</progress>

```

The following is the UploadProgressServlet.java:

```

package servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class UploadProgressServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {

        HttpSession session = request.getSession();
        UploadProgressListener pListener =
            (UploadProgressListener)session.getAttribute("Progress");
        if(pListener != null) {

```

```

long bytesRead = pListener.getBytesRead();
long contentLength = pListener.getContentLength();
int itemNo = pListener.getItemNo();
if(itemNo > 0) {
    StringBuffer buffer = new StringBuffer();
    buffer.append("Uploading item #" + itemNo + "... ");
    if(contentLength > -1)
        buffer.append(String.format("%3d percent", (bytesRead*100)/contentLength));
    else
        buffer.append(bytesRead + " bytes");
    String message = buffer.toString();
    String completed = "false";
    if(bytesRead == contentLength) {
        completed = "true";
    }
    // response
    response.setContentType("text/xml");
    response.setHeader("Cache-Control", "no-cache");
    PrintWriter out = response.getWriter();
    buffer = new StringBuffer();
    buffer.append("<progress>");
    buffer.append("<message>").append(message).append("</message>");
    buffer.append("<completed>").append(completed).append("</completed>");
    buffer.append("</progress>");
    out.println(buffer.toString());
    out.close();
}
}
}

```

```

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {

```

```
doGet(request, response);
}
}
```

The following is the index.html for serving the main page of this example. It has one iframe which is used to show final result and prevent it from moving to a new page when the form is submitted. Once the form is submitted, an Ajax request is sent to the UploadProgressServlet to get current progress. A request is sent to the server every one second until it is completed:

```
<html>
<head>
<title>Upload File</title>
</head>
<script type="text/javascript">
function createXmlHttpRequest() {
    var xmlhttp = null;
    if(window.XMLHttpRequest) {
        // non-IE
        xmlhttp = new XMLHttpRequest();
    } else if(window.ActiveXObject) {
        // IE/Windows ActiveX version
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }

    return xmlhttp;
}

function sendRequest() {
    document.resultFrame.document.body.innerHTML = "";
    window.setTimeout("checkProgress();", 500);
}

function checkProgress() {
    xmlhttp = createXmlHttpRequest();
```

```

try {
xmlHttp.onreadystatechange = showProgress;
xmlHttp.open("GET", "UploadProgressServlet", true);
xmlHttp.send(null);
} catch(ex) {
alert(ex);
}
}

function showProgress() {
if(xmlHttp.readyState == 4) {
if(xmlHttp.status == 200) {
var xml = xmlHttp.responseXML;
if(xml != null) {
var message = xml.getElementsByTagName("message")[0].childNodes[0].nodeValue;
var completed = xml.getElementsByTagName("completed")
[0].childNodes[0].nodeValue;
document.getElementById("progress").innerHTML = message;
if(completed == "false")
window.setTimeout("checkProgress();", 1000);
}
}
}
}
</script>
<body>
<form method="post" action="FileUploadServlet" enctype="multipart/form-data"
target="resultFrame" onsubmit="sendRequest();">
<input type="file" name="fileupload">
<br>
<input type="submit" value="Upload">
</form>
<p><div id="progress"></div></p>

```



```
<iframe id="resultFrame" name="resultFrame" height="100" width="600"
  frameborder="0"></iframe>
</body>
</html>
```

The following is the web.xml for deploying this example to a servlet container:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE web-app
```

```
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
```

```
  "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app>
```

```
  <display-name>FileUpload Example</display-name>
```

```
  <servlet>
```

```
    <servlet-name>FileUploadServlet</servlet-name>
```

```
    <servlet-class>servlet.FileUploadServlet</servlet-class>
```

```
    <load-on-startup>1</load-on-startup>
```

```
  </servlet>
```

```
  <servlet>
```

```
    <servlet-name>UploadProgressServlet</servlet-name>
```

```
    <servlet-class>servlet.UploadProgressServlet</servlet-class>
```

```
    <load-on-startup>1</load-on-startup>
```

```
  </servlet>
```

```
  <servlet-mapping>
```

```
    <servlet-name>FileUploadServlet</servlet-name>
```

```
    <url-pattern>/FileUploadServlet</url-pattern>
```

```
  </servlet-mapping>
```

```
  <servlet-mapping>
```

```
    <servlet-name>UploadProgressServlet</servlet-name>
```

```
    <url-pattern>/UploadProgressServlet</url-pattern>
```

```
  </servlet-mapping>
```

```
</web-app>
```


iText

PDF (Portable Document Format) was invented by Adobe Systems. It is an open standard for portable electronic documents. Virtually, you can open PDF documents from any platforms or devices through a free Adobe Reader. Usually, PDF documents are created or updated manually through some kind of end-user applications such as Adobe Acrobat. But, that is only suitable for static documents. If you need to generate PDF documents dynamically based on requests or to generate personalized contents, you need to use different approaches. iText is an API that helps developers to generate or update PDF documents on the fly programmatically without manual process.

You can download iText from <http://itextpdf.com>. The latest version is 5.4.0 at the time of writing.

CREATING A PDF DOCUMENT

Using iText to create a PDF document involves the following five steps:

Step 1

First, a Document object needs to be created. The default page size is A4. Available pre-defined page sizes are in the PageSize class. For example, you can use PageSize.LETTER for letter size. If you need to change orientation from portrait to landscape, you can use PageSize.LETTER.rotate() or PageSize.LETTER_LANDSCAPE. You can also create a Rectangle object as a custom page size. The Rectangle class (not the Rectangle class in the core Java API) also supports background color, border width/color. To create a Document with page size or margins or both specified, you can use the following constructors:

Document(Rectangle pageSize): constructs a Document with specific page size and default margins. The default value is 36 points for each margin. Since there are 72 points per inch, that is equal to 0.5 inch.

Document(Rectangle pageSize, float marginLeft, float marginRight, float marginTop, float marginBottom)

Step 2

To generate the output of a document, you need to specify a writer (PdfWriter) that listens to certain document. Every element added to the document will be written to an output stream. Here, the document will be written to a file. Usually, it is one PdfWriter per document. But, you can have more than one PdfWriter listening to the same document to create slightly different documents. For example, a writer can be paused for writing by calling pause() method and resume writing by calling resume() method.

Step 3

Before any content can be added to the body of the document, it needs to be opened. Once a document is opened, you cannot add any metadata (document description and custom properties) to that document. You can add some metadata for this document with the following methods provided by the Document class:

boolean addAuthor(String author)

boolean addKeywords(String keywords)

boolean addSubject(String subject)

boolean addTitle(String title)

boolean addHeader(String name, String content): adds a custom property.

You can find metadata of a PDF document by opening it from the Adobe Reader. Click File -> Properties from the menu bar on top. You can find metadata under the Description and Custom (for custom properties) tabs.

Once metadata are set, you can call `open()` method to open the document for writing.

Step 4

Now, you can start adding content to the document by using the `add(Element element)` method. You can add any text elements (type of `Element`) to the document by using this method. When an element is added to a document, its content is written to the writer listening to this document too. Once it is added, you cannot change it. For example, the `Paragraph` class is one of the text elements.

Step 5

The last step is to close the document for writing by calling the `close()` method to flush and close the output stream. Once a document is closed, nothing can be written to the body anymore.

To build and run programs using the iText API, you need to include proper iText JAR file (e.g., `itextpdf-5.4.0.jar` for iText 5.4.0) in the classpath.

The following example creates a PDF document, `simple.pdf`, which only contains one paragraph in the document:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.pdf.PdfWriter;

public class SimpleExample {

    public static void main(String args[]) {
        try {
            File file = new File("simple.pdf");
            new SimpleExample().generatePdf(file);
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }
}
```

```
public void generatePdf(File file) throws DocumentException, IOException {  
    // 1. Create a document object for PDF  
    Document doc = new Document();  
    // 2. create a PDF writer that listens to this document  
    PdfWriter.getInstance(doc, new FileOutputStream(file));  
    // 3. open this document  
    doc.open();  
    // 4. add a paragraph  
    Paragraph par = new Paragraph("My first PDF document.");  
    doc.add(par);  
    // 5. close this document  
    doc.close();  
    }  
}
```

BASIC TEXT ELEMENTS

Previously, we create a PDF document that only contains a paragraph. A paragraph contains a series of chunks/phrases. A chunk is the smallest significant part of text that can be added to a document. A chunk contains a string with a certain font or an image. A phrase contains a series of chunks. Chunk is represented by the Chunk class and phrase is represented by the Phrase class. Both Chunk and Phrase classes are text elements too. A Paragraph is quite similar with a Phrase except a Paragraph has additional layout parameters: indentation and text alignment. Also, every Paragraph starts a new line when it is added to a Document. Paragraph is a subclass of Phrase.

Paragraph

To construct a Paragraph, you can use the following constructors:

Paragraph(float leading)

Paragraph(Chunk chunk)

Paragraph(Phrase phrase)

Paragraph(String string)

Paragraph(String string, Font font)

Also, you can use the add(Element element) or add(int index, Element element) method to add text elements (including a Paragraph). Paragraph provides the following methods to change layout parameters:

void setAlignment(int alignment): sets alignments of this paragraph (e.g., Element.ALIGN_CENTER).

void setFirstLineIndent(float indentation): sets indentation on the first line.

void setIndentationLeft(float indentation): sets indentation on the left side.

void setIndentationRight(float indentation): sets indentation on the right side.

void setLeading(float leading): sets the space between lines. The default value is 16 points.

void setSpacingBefore(float spacing): sets spacing before this paragraph.

void setSpacingAfter(float spacing): sets spacing after this paragraph.

Phrase

To construct a Phrase, you can use the following constructors:

Phrase(Chunk chunk)

Phrase(Phrase phrase)

Phrase(String string)

Phrase(String string, Font font)

Similarly, you can use add methods to add a Chunk or Phrase to this Phrase.

Chunk

To construct a Chunk, you can use the following constructors:

Chunk(Chunk chunk)

Chunk(String string)

Chunk(String string, Font font)

Chunk(Image image, float offsetX, float offsetY)

A Chunk can contain not just text. It can contain an image too. An Image (not the Image class in the core Java API) represents a graphic element (e.g., JPEG or PNG). You can get an Image through `Image.getInstance(URL url)` or `Image.getInstance(String filename)`. You can scale an image through `scalePercent(int percent)` or `scaleToFit(int width, int height)`. To rotate an image, you can use `setRotationDegrees(float degree)`. The following are some useful methods provided by Chunk:

Chunk `setAnchor(String url)`: sets a link. Once it is clicked, it will open the link in a browser.

Chunk `setAnchor(URL url)`

Chunk `setBackground(BaseColor color)`: sets background color.

Chunk `setHorizontalScaling(float scale)`: sets the text horizontal scaling. You can shrink the text width by using a value less than 1 or expand it by using a value greater than 1.

Chunk `setTextRenderMode(int mode, float strokeWidth, BaseColor strokeColor)`: sets the text rendering mode. Available text rendering modes are defined in the PdfContentByte class. For example, `TEXT_RENDER_MODE_STROKE` is to outline text.

Chunk `setTextRise(float rise)`: sets the text displacement relative to the baseline. Positive number has the effect of superscript and negative number has the effect of subscript.

These methods return a Chunk object. That allows method chaining. So, you can define a Chunk in descriptive way. For example, you can use `chunk.setHorizontalScaling(1.5f).setBackground(BaseColor.LIGHT_GRAY)`. The following is a sample output using these methods:



The following example shows how to create a PDF document using the Paragraph, Phrase and Chunk classes and it also demonstrates how font styles are propagated among text elements:

```
import java.io.File;
```

```
import java.io.FileOutputStream;
```

```
import java.io.IOException;

import com.itextpdf.text.Chunk;
import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Font;
import com.itextpdf.text.FontFactory;
import com.itextpdf.text.Font.FontFamily;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.Phrase;
import com.itextpdf.text.pdf.PdfWriter;

public class ParagraphExample {

    public static void main(String args[]) {
        try {
            File file = new File("paragraph.pdf");
            new ParagraphExample().generatePdf(file);
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }

    public void generatePdf(File file) throws DocumentException, IOException {
        // 1. create a document object for PDF
        Document doc = new Document();
        // 2. create a PDF writer that listens to this document
        PdfWriter.getInstance(doc, new FileOutputStream(file));
        // 3. open this document
        doc.open();
        // 4. add paragraphs
        Paragraph par1 = new Paragraph("This is paragraph 1 with TIMES_ROMAN
NORMAL.",
```

```
new Font(FontFamily.TIMES_ROMAN, 10));
par1.add(new Chunk("This is chunk 1 with COURIER BOLD.",
new Font(FontFamily.COURIER, 10, Font.BOLD)));
par1.add(new Chunk("This is chunk 2 with paragraph default."));
doc.add(par1);
// propagation on font styles
Paragraph par2 = new Paragraph("This is paragraph 2 with TIMES_ROMAN
ITALIC.",
new Font(FontFamily.TIMES_ROMAN, 10, Font.ITALIC));
par2.add(new Chunk("This is chunk 1 with COURIER BOLD.",
new Font(FontFamily.COURIER, 10, Font.BOLD)));
par2.add(new Chunk("This is chunk 2 with paragraph default."));
doc.add(par2);
// no propagation on font styles
Paragraph par3 = new Paragraph("This is paragraph 3 with TIMES_ROMAN
ITALIC.",
FontFactory.getFont(FontFactory.TIMES_ROMAN, 10, Font.ITALIC));
par3.add(new Chunk("This is chunk 1 with COURIER BOLD.",
FontFactory.getFont(FontFactory.COURIER, 10, Font.BOLD)));
par3.add(new Chunk("This is chunk 2 with paragraph default."));
doc.add(par3);
// isolate font styles by using phrases
Paragraph par4 = new Paragraph("This is paragraph 4.");
Phrase phrase1 = new Phrase("This is phrase 1 with TIMES_ROMAN ITALIC.",
new Font(FontFamily.TIMES_ROMAN, 10, Font.ITALIC));
phrase1.add(new Chunk("This is chunk 1 with COURIER BOLD.",
new Font(FontFamily.COURIER, 10, Font.BOLD)));
phrase1.add(new Chunk("This is chunk 2 with default."));
par4.add(phrase1);
Phrase phrase2 = new Phrase("This is phrase 2 with TIMES_ROMAN UNDERLINE.",
new Font(FontFamily.TIMES_ROMAN, 10, Font.UNDERLINE));
phrase2.add(new Chunk("This is chunk 1 with COURIER BOLD.",
new Font(FontFamily.COURIER, 10, Font.BOLD)));
```

```

phrase2.add(new Chunk("This is chunk 2 with default."));
par4.add(phrase2);
par4.add(new Chunk("This is end of paragraph 4."));
doc.add(par4);
// 5. close this document
doc.close();
}
}

```

The following is the output:

```

This is paragraph 1 with TIMES_ROMAN NORMAL.This is chunk 1 with COURIER BOLD.This is chunk 2 with
paragraph default.
This is paragraph 2 with TIMES_ROMAN ITALIC.This is chunk 1 with COURIER BOLD.This is chunk 2 with
paragraph default.
This is paragraph 3 with TIMES_ROMAN ITALIC.This is chunk 1 with COURIER BOLD.This is chunk 2 with
paragraph default.
This is paragraph 4.This is phrase 1 with TIMES_ROMAN ITALIC.This is chunk 1 with COURIER BOLD.This
is chunk 2 with default.This is phrase 2 with TIMES_ROMAN UNDERLINE.This is chunk 1 with COURIER BOLD.
This is chunk 2 with default.This is end of paragraph 4.

```

There are four paragraphs in this example. The font style is not specified in the first paragraph. There are two children (two chunks) in this paragraph. Since the default font style for the parent is NORMAL, nothing is propagated to child elements. Chunk 1 and chunk 2 are displayed as defined.

In the second paragraph, the font style is changed to ITALIC. It propagates to child elements. Since the font style of chunk 1 is defined as BOLD, the result font style is ITALIC + BOLD.

In the third paragraph, we use the FontFactory class to create fonts. FontFactory uses different technique to construct fonts. Font styles are not propagated.

To have better control on propagation of font style, you can use Phrase objects as the middle layer between a Paragraph (top layer) and Chunk objects (bottom layer). You do not specify font style in the paragraph. Font style is defined in a phrase if needed. In this way, it only affects child elements under it and will not affect other phrases too.

FONTS

You can define a font (including font family, size, style and color) by using the `Font` class (not the `Font` class in the core Java API). To construct a `Font`, you can use the following constructors:

```
Font(Font.FontFamily family)
```

```
Font(Font.FontFamily family, float size)
```

```
Font(Font.FontFamily family, float size, int style)
```

```
Font(Font.FontFamily family, float size, int style, BaseColor color)
```

Available fonts are defined in the `Font.FontFamily` class: `COURIER`, `HELVETICA`, `SYMBOL`, `TIMES_ROMAN`, `ZAPFDINGBATS`. You can combine multiple styles together by using vertical bars `|`. For example, to combine italic and strikethrough, you can use `Font.ITALIC | Font.STRIKETHRU`. Those fonts mentioned above are Type 1 fonts. If you need to use different types of fonts, you can use the `BaseFont` class to create base fonts. A `BaseFont` can be created by using:

```
BaseFont.createFont(String name, String encoding, boolean embedded):
```

 creates a new base font. Name is the name of the font or the location of a font file. Encoding is the encoding to be used in this font. If the font is to be embedded in the PDF document, you can set `embedded` as `true` (or `BaseFont.EMBEDDED`). If fonts are embedded, document size becomes larger.

```
For example, to create a base font HELVETICA with WINANSI encoding, you can use:  
BaseFont.createFont(BaseFont.HELVETICA, BaseFont.WINANSI,  
BaseFont.NOT_EMBEDDED)
```

Encoding `WINANSI` is the same as `CP1252` (a character encoding for Latin). Each font only supports certain encodings, you need to use supported encoding. Or, you might get the following error message during the runtime:

```
com.itextpdf.text.DocumentException: Font 'c:/windows/fonts/mingliu.ttc' with 'Cp1252'  
is not recognized.
```

The following are constructors that can be used to create a `Font` by using a `BaseFont`:

```
Font(BaseFont bf)
```

```
Font(BaseFont bf, float size)
```

```
Font(BaseFont bf, float size, int style)
```

```
Font(BaseFont bf, float size, int style, BaseColor color)
```

If you need to use a True Type font installed in your computer, you can point to the location of a font file. For example, to use Century Gothic font in MS Windows, you can

use:

```
BaseFont.createFont("c:/windows/fonts/gothic.ttf", BaseFont.WINANSI, true)
```

If you want to use CJK (Chinese, Japanese and Korean) font technology, you need an extra JAR file, `itext-asian.jar`, during the runtime. You can download it from <http://sourceforge.net/projects/itext/files/extrajars>. For example, to use a Traditional Chinese font, you can use:

```
BaseFont.createFont("MSungStd-Light", "UniCNS-UCS2-H",  
BaseFont.NOT_EMBEDDED)
```

For Traditional Chinese, three fonts are available: MHei-Medium, MSung-Light and MSungStd-Light and two encodings are available: UniCNS-UCS2-H (horizontal writing), UniCNS-UCS2-V (vertical writing). For Simplified Chinese, STSong-Light and STSongStd-Light are available with encodings UniGB-UCS2-H and UniGB-UCS2-V.

You cannot embed those fonts in the PDF document using iText because of licensing issue. You can either install the font pack on your own or install it on demand when Adobe Reader realizes it is missing. An alternative to using CJK font technology is to use fonts in your computer. For example, to use KaiTi font in MS Windows, you can use:

```
BaseFont.createFont("c:/windows/fonts/simkai.ttf", BaseFont.IDENTITY_H,  
BaseFont.EMBEDDED)
```

You do not need to include an extra JAR file. But, you need to make sure those fonts are embedded in the PDF document. The following example shows how to use BaseFont:

```
import java.io.File;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
import com.itextpdf.text.Chunk;  
import com.itextpdf.text.Document;  
import com.itextpdf.text.DocumentException;  
import com.itextpdf.text.Font;  
import com.itextpdf.text.Font.FontFamily;  
import com.itextpdf.text.PageSize;  
import com.itextpdf.text.Paragraph;  
import com.itextpdf.text.pdf.BaseFont;  
import com.itextpdf.text.pdf.PdfWriter;  
  
public class FontsExample {
```



```

public static void main(String args[]) {
try {
File file = new File("fonts.pdf");
new FontsExample().generatePdf(file);
} catch(Exception ex) {
System.out.println(ex);
}
}

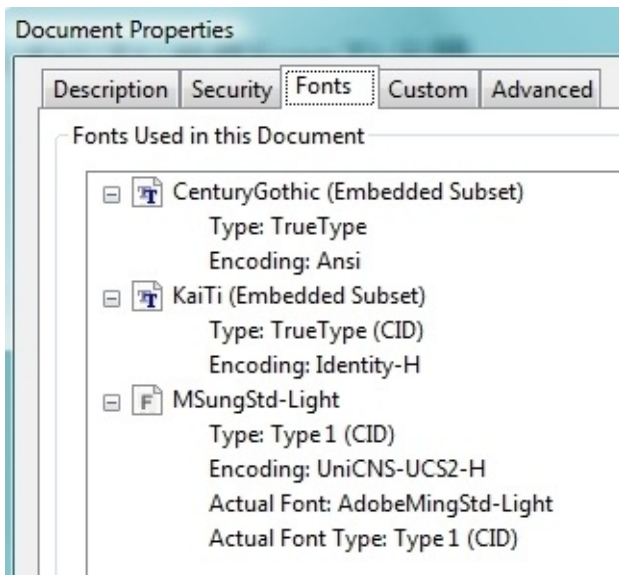
```

```

public void generatePdf(File file) throws DocumentException, IOException {
Document doc = new Document(PageSize.LETTER);
PdfWriter.getInstance(doc, new FileOutputStream(file));
doc.open();
Paragraph par = new Paragraph();
BaseFont bfGothic = BaseFont.createFont("c:/windows/fonts/gothic.ttf",
BaseFont.WINANSI, BaseFont.EMBEDDED);
Font font = new Font(bfGothic, 12);
par.add(new Chunk("Century Gothic", font));
BaseFont bfKai = BaseFont.createFont("c:/windows/fonts/simkai.ttf",
BaseFont.IDENTITY_H, BaseFont.EMBEDDED);
font = new Font(bfKai, 12);
par.add(new Chunk("Kai Ti 楷體 ", font));
BaseFont bfSung = BaseFont.createFont("MSungStd-Light", "UniCNS-UCS2-H",
BaseFont.NOT_EMBEDDED);
font = new Font(bfSung, 12);
par.add(new Chunk("Sung Ti 宋體 ", font));
doc.add(par);
doc.close();
}
}

```

When you check Document Properties of this PDF document, you can find information on fonts using in this document as shown below:



Note: When you include any multi-byte characters in your Java source, you need to save it using UTF-8 encoding and compile it with “-encoding utf8” option.

CREATING A TABLE

To create a table in the document, you can use the PdfPTable class. A table is constructed by cells. Usually, you specify number of columns in the table. Rows are added automatically.

PdfPTable

To construct a PdfPTable, you can use the following constructors:

`PdfPTable(float[] columnWidths)`: constructs a PdfPTable specifying relative column widths.

`PdfPTable(int numColumns)`: constructs a PdfPTable specifying number of columns.

Table cells are added through the `addCell` methods. The following types of parameter are supported: `String`, `Phrase`, `Image`, `PdfPCell`, `PdfPTable` (a nested table). The following are useful methods that you can use in a table:

`void setWidthPercentage(float width)`: defines table width in percentage that the table will occupy in the page. A value of 100 means 100%.

`void setHorizontalAlignment(int alignment)`: defines horizontal alignment relative to the page. It is only meaningful when the width percentage is less than 100%. For example, `Element.ALIGN_CENTER`.

PdfPCell

As we mentioned before, five types of objects can be added as table cells. It is not required to use a PdfPCell. If you need to have custom cell properties, you can use a PdfPCell. To construct a PdfPCell, you can use the following constructors:

PdfPCell(Image image)

PdfPCell(Image image, boolean fit)

PdfPCell(PdfPCell cell)

PdfPCell(PdfPTable table)

PdfPCell(Phrase phrase)

The following are useful methods that you can use in a cell:

void setHorizontalAlignment(): sets the horizontal alignment of the cell. For example, Element.ALIGN_CENTER.

void setVerticalAlignment(): sets the vertical alignment of the cell. For example, Element.ALIGN_MIDDLE.

void setColspan(int colspan): sets the column span (number of columns this cell can occupy).

void setRowspan(int rowspan): sets the row span (number of rows this cell can occupy).

void setPadding(float padding): sets the space between content and cell borders.

void setRotation(int rotation): sets the rotation of the cells (in counterclockwise direction). Rotation must be a multiple of 90 and can be negative.

Since PdfPCell is a subclass of the Rectangle class, you can use methods supported by Rectangle to change properties such as background color or borders. Now, we use the following example to demonstrate how to create a table in a document:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

import com.itextpdf.text.Anchor;
import com.itextpdf.text.Chunk;
import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Element;
import com.itextpdf.text.Font;
```

```

import com.itextpdf.text.Font.FontFamily;
import com.itextpdf.text.PageSize;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.Phrase;
import com.itextpdf.text.pdf.PdfPCell;
import com.itextpdf.text.pdf.PdfPTable;
import com.itextpdf.text.pdf.PdfWriter;

public class TableExample {
    private static Font boldFont =
        new Font(FontFamily.TIMES_ROMAN, 12, Font.BOLD);
    public static void main(String args[]) {
        try {
            File file = new File("table.pdf");
            new TableExample().generatePdf(file);
        } catch(Exception ex) {
            System.out.println(ex);
        }
    }

    public void generatePdf(File file) throws DocumentException, IOException {
        // reate a document object for PDF
        Document doc = new Document(PageSize.LETTER, 10, 10, 20, 10);
        // create a PDF writer that listens to this document
        PdfWriter.getInstance(doc, new FileOutputStream(file));
        // set file properties
        doc.addAuthor("CC");
        doc.addSubject("An example of creating a table");
        // open this document
        doc.open();
        // title
        Paragraph titlePar = new Paragraph();

```

```
titlePar.add(new Chunk("Purchase Order\n\n"));
doc.add(titlePar);
// table
float[] widths = {0.5f, 0.2f, 0.1f, 0.1f, 0.1f};
PdfPTable itemsTable = new PdfPTable(widths);
itemsTable.setWidthPercentage(100); // 100% of the available space
// headers
itemsTable.addCell(new Phrase("Name", boldFont));
itemsTable.addCell(new Phrase("Company", boldFont));
itemsTable.addCell(new Phrase("Price", boldFont));
itemsTable.addCell(new Phrase("Quantity", boldFont));
itemsTable.addCell(new Phrase("Subtotal", boldFont));
// items
itemsTable.addCell("Acer ICONIA TAB A700");
Anchor url = new Anchor("Acer");
url.setReference("http://www.acer.com");
itemsTable.addCell(url);
PdfPCell priceCell = new PdfPCell(new Phrase(String.format("%8.2f", 400.0)));
priceCell.setHorizontalAlignment(Element.ALIGN_RIGHT);
itemsTable.addCell(priceCell);
PdfPCell quantityCell = new PdfPCell(new Phrase("2"));
quantityCell.setHorizontalAlignment(Element.ALIGN_RIGHT);
itemsTable.addCell(quantityCell);
PdfPCell subtotalCell = new PdfPCell(new Phrase(String.format("%8.2f", 800.0)));
subtotalCell.setHorizontalAlignment(Element.ALIGN_RIGHT);
itemsTable.addCell(subtotalCell);
itemsTable.addCell("ASUS TF700T");
url = new Anchor("ASUS");
url.setReference("http://www.asus.com");
itemsTable.addCell(url);
priceCell = new PdfPCell(new Phrase(String.format("%8.2f", 429.0)));
priceCell.setHorizontalAlignment(Element.ALIGN_RIGHT);
```

```

itemsTable.addCell(priceCell);
quantityCell = new PdfPCell(new Phrase("1"));
quantityCell.setHorizontalAlignment(Element.ALIGN_RIGHT);
itemsTable.addCell(quantityCell);
subtotalCell = new PdfPCell(new Phrase(String.format("%8.2f", 429.0)));
subtotalCell.setHorizontalAlignment(Element.ALIGN_RIGHT);
itemsTable.addCell(subtotalCell);

PdfPCell totalCell = new PdfPCell(new Phrase(String.format("Total %8.2f", 1229.0)));
totalCell.setHorizontalAlignment(Element.ALIGN_RIGHT);
totalCell.setColspan(4);
itemsTable.addCell(totalCell);
doc.add(itemsTable);
// close this document
doc.close();
}
}

```

The following is the output:

Purchase Order

Name	Company	Price	Quantity	Subtotal
Acer ICONIA TAB A700	Acer	400.00	2	800.00
ASUS TF700T	ASUS	429.00	1	429.00

Cells in the Company column contain links to the company website. Once it is clicked, it will open the link in a browser. The Anchor class can be used to create a link. Since Anchor is a subclass of Phrase, you can add it to a cell directly.

ENCRYPTION

Adobe Acrobat allows users to add security permissions to PDF documents through Advanced -> Security. You can use either a password or a certificate to restrict users from accessing certain features. You can find security properties of a document on Security tab of Document Properties (File -> Properties). To encrypt a document programmatically, you can use the PdfEncryptor class or use setEncryption() method from the PdfWriter class:

`void setEncryption(byte[] userPassword, byte[] ownerPassword, int permissions. int encryptionType):` sets the password encryption. The user password is the password to remove security if it is set. The owner password is the password to open the document if it is set. Both passwords can be null or empty. You can combine multiple permissions together by using vertical bars |.

`void setEncryption(Certificate[] certs, int[] permissions. int encryptionType):` sets the certificate encryption. An array of public certificates is provided together with an array of permissions of the same size for each certificate.

To encrypt a PDF document, you need to read it first. To read a PDF document, you can use a PdfReader. To modify a PDF document, you can use a PdfStamper. Next, you get a PdfWriter from this PdfStamper and then call the setEncryption() method to encrypt it. For an encrypted document, to remove security, you need to do that through Adobe Acrobat (Advanced -> Security -> Remove Security).

To run the following example, you need to include the Bouncy Castle Crypto provider package (http://www.bouncycastle.org/latest_releases.html) in the classpath (e.g., bcprov-jdk15on-1.48.jar). The following example shows how to do password encryption to an existing PDF document:

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

import com.itextpdf.text.pdf.PdfReader;
import com.itextpdf.text.pdf.PdfStamper;
import com.itextpdf.text.pdf.PdfWriter;

public class EncryptionExample {
    public static void main(String[] args) {
        try {
            PdfReader reader = new PdfReader("paragraph.pdf");
```

```

BufferedOutputStream outputStream =
new BufferedOutputStream(new
FileOutputStream("paragraph.tmp.pdf"));
PdfStamper stamper = new PdfStamper(reader, outputStream);
PdfWriter writer = stamper.getWriter();
PdfSecurity security = new PdfSecurity();
security.encrypt(writer, "mypassword".getBytes());
stamper.close();
} catch(Exception ex) {
System.out.println(ex);
}
}
}

```

```
import java.io.IOException;
```

```
import com.itextpdf.text.DocumentException;
```

```
import com.itextpdf.text.pdf.PdfWriter;
```

```
/**
```

```
* PdfSecurity is responsible for changing the security settings of a PDF document.
```

```
*/
```

```
public class PdfSecurity {
```

```
    private static int LIMITED_PERMISSIONS =
```

```
    PdfWriter.ALLOW_COPY;
```

```
    private static int ALL_PERMISSIONS =
```

```
    PdfWriter.ALLOW_PRINTING |
```

```
    PdfWriter.ALLOW_MODIFY_CONTENTS |
```

```
    PdfWriter.ALLOW_COPY |
```

```
    PdfWriter.ALLOW_MODIFY_ANNOTATIONS |
```

```
    PdfWriter.ALLOW_FILL_IN |
```

```
    PdfWriter.ALLOW_SCREENREADERS |
```

```
    PdfWriter.ALLOW_ASSEMBLY |
```

```
    PdfWriter.ALLOW_DEGRADED_PRINTING;
```

```
/**
```

```
* Encrypts a PDF document to disable the editing capability.
```

```
* Printing is still allowed.
```

```
*/
```

```
public void encrypt(PdfWriter writer, byte[] password)
throws IOException, DocumentException {
    sign(writer, password, null, LIMITED_PERMISSIONS);
}
```

```
/**
```

```
* Decrypts a PDF document to restore the editing capability.
```

```
*/
```

```
public void decrypt(PdfWriter writer, byte[] password)
throws IOException, DocumentException {
    sign(writer, password, null, ALL_PERMISSIONS);
}
```

```
/**
```

```
* Signs the document and changes security settings.
```

```
*
```

```
* @param writer a PDF writer
```

```
* @param ownerPassword owner password for changing security settings
```

```
* @param userPassword user password for opening the document
```

```
* @param permissions file permissions
```

```
* @throws IOException
```

```
* @throws DocumentException
```

```
*/
```

```
private void sign(PdfWriter writer, byte[] ownerPassword, byte[] userPassword,
int permissions) throws IOException, DocumentException {
    writer.setEncryption(userPassword, ownerPassword, permissions,
PdfWriter.STANDARD_ENCRYPTION_128);
}
```

```
}
```

The following is a code snippet of using a PdfEncryptor to encrypt an existing document:

```
PdfReader reader = new PdfReader(file.toString());  
BufferedOutputStream outputStream =  
    new BufferedOutputStream(new FileOutputStream("paragraph.tmp.pdf"));  
PdfEncryptor.encrypt(reader, outputStream,  
    userPassword, ownerPassword, permissions, true);  
outputStream.close();
```

You also can encrypt a document before it is created. It is a similar process. But, the encryption only can be added before the document is opened as shown by the following code snippet:

```
Document doc = new Document();  
File file = new File("encrypted.pdf");  
PdfWriter writer = PdfWriter.getInstance(doc, new FileOutputStream(file));  
PdfSecurity security = new PdfSecurity();  
security.encrypt(writer, "mypassword".getBytes());  
doc.open();  
...  
doc.close();
```

ADDING WATERMARKS

A PDF document has a layered structure. Text elements mentioned previously are in the middle two layers. One is for text and the other is for graphics (e.g., background or borders). Text layer is on top of graphics layer. For low-level operations such as absolute positioning of text and graphics, you need to access the direct content (represented by the PdfContentByte class). You cannot access direct contents in the middle two layers. Those are managed by iText internally. But, you can access layers over or under middle layers. The PdfStamper class provides methods to access direct content at certain page of the original document. The getUnderContent() method is for the bottom layer and the getOverContent() method is for the top layer. Both methods return a PdfContentByte object. PdfContentByte allows you to draw graphics, position text and image or change the way graphical objects (including text) are rendered.

The following example uses these two layers to demonstrate how to add watermarks in a document:

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

import com.itextpdf.text.Image;
import com.itextpdf.text.Rectangle;
import com.itextpdf.text.pdf.BaseFont;
import com.itextpdf.text.pdf.PdfContentByte;
import com.itextpdf.text.pdf.PdfReader;
import com.itextpdf.text.pdf.PdfStamper;

public class WatermarkExample {
    public static void main(String[] args) {
        try {
            Image logo = Image.getInstance("logo.png");
            logo.scalePercent(50);
            BaseFont bf = BaseFont.createFont(BaseFont.HELVETICA,
            BaseFont.WINANSI, BaseFont.EMBEDDED);
            PdfReader reader = new PdfReader("table.pdf");
            BufferedOutputStream outputStream =
```

```

new BufferedOutputStream(new FileOutputStream("watermark.pdf"));
PdfStamper stamper = new PdfStamper(reader, outStream);
for(int i = 1; i <= reader.getNumberOfPages(); i++) {
Rectangle pageSize = reader.getPageSize(i);
// add image at the bottom layer
logo.setAbsolutePosition((pageSize.getWidth()-logo.getScaledWidth())/2,
pageSize.getHeight()-200);
PdfContentByte content = stamper.getUnderContent(i);
content.addImage(logo);
// add text at the top layer
content = stamper.getOverContent(i);
content.beginText();
content.setTextRenderingMode(PdfContentByte.TEXT_RENDER_MODE_FILL_STROKE);
content.setLineWidth(2.0f);
content.setRGBColorStroke(0xC0, 0xC0, 0xC0);
content.setRGBColorFill(0xFF, 0xFF, 0xFF);
content.setFontAndSize(bf, 48);
content.showTextAligned(PdfContentByte.ALIGN_CENTER, "Just a Sample",
pageSize.getWidth()/2, pageSize.getHeight()-120, 0);
content.endText();
}
stamper.close();
} catch(Exception ex) {
System.out.println(ex);
}
}
}

```

The following is the output:

Purchase Order

Name	Company	Price	Quantity	Subtotal
Acer ICONIA TAB A700	Acer	400.00	2	800.00
ASUS TF700T	ASUS	429.00	1	429.00

Just a Sample

To write text to a direct content, you start with the `beginText()` method and end with the `endText()` method. When the way graphical objects rendered are changed, it means graphic state in the content is changed. If you do not maintain graphic state carefully, it might affect following graphical objects unexpectedly. To prevent this from happening, you can use the `saveState()` method to save the graphic state before making any changes and use the `restoreState()` method to restore it after changes.

Other than getting a `PdfContentByte` through a `PdfStamper`, you also can get a `PdfContentByte` from the `getDirectContent()` method for top layer or from the `getDirectContentUnder()` method for bottom layer in a `PdfWriter`.

CREATING MULTIPLE COLUMNS

If you need to create a document with multiple columns, you can use the `ColumnText` class. It involves the following steps:

Step 1

First, you need to create a `ColumnText` object using the following constructor:

```
ColumnText(PdfContentByte content)
```

You can get a `PdfContentByte` by calling the `getDirectContent()` method from a `PdfWriter`.

Step 2

Now, you can start loading text to a `ColumnText` by using the `addText(Phrase phrase)` or `addText(Chunk chunk)` method.

Step 3

Before a `ColumnText` can start writing text to the document, you need to define the column first. The easier way is to define a rectangular column by using:

```
void setSimpleColumn(float llx, float lly, float urx, float ury, float leading, int alignment):
```

`llx` and `lly` define the lower left corner. `urx` and `ury` define the upper right corner.

You also need to define the starting position at a column by using:

```
void setYLine(float yLine)
```

Other than using the `setSimpleColumn` method above to define leading and alignment, you can use the following methods provided by the `ColumnText` class:

```
void setAlignment(int alignment)
```

```
void setLeading(float leading)
```

Step 4

To start writing text from a `ColumnText` to the column you just defined, call the `go()` method. A status of either `NO_MORE_COLUMN` or `NO_MORE_TEXT` is returned. If the status is `NO_MORE_COLUMN`, it means a new column needs to be defined. So, go back to the step 3 to define another column. It depends on the number of columns at a page. Once all columns at a page are written and there is still more text. A new page needs to be created. You continue this process until there is no more text.

In the following example, it reads a text file and writes two pages as one page by creating two columns per page in the document:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
```



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

import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Element;
import com.itextpdf.text.PageSize;
import com.itextpdf.text.Phrase;
import com.itextpdf.text.Rectangle;
import com.itextpdf.text.pdf.ColumnText;
import com.itextpdf.text.pdf.PdfContentByte;
import com.itextpdf.text.pdf.PdfWriter;

public class ColumnTextExample {
    public static void main(String[] args) {
        try {
            File pdfFile = new File("columnText.pdf");
            File file = new File("ColumnTextExample.java");
            new ColumnTextExample().generatePdf(pdfFile, file);
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }

    public void generatePdf(File pdfFile, File file) throws DocumentException,
        IOException {
        // landscape mode
        Rectangle pageSize = PageSize.LETTER.rotate();
        Document doc = new Document(pageSize);
        PdfWriter writer = PdfWriter.getInstance(doc, new FileOutputStream(pdfFile));
        doc.open();
        PdfContentByte content = writer.getDirectContent();
```

```

ColumnText columnText = new ColumnText(content);
BufferedReader reader =
new BufferedReader(new FileReader(file));
String line;
while((line = reader.readLine()) != null) {
columnText.addText(new Phrase(line + "\n"));
}
reader.close();
// the first column
boolean isFirstColumn = true;
// define a rectangular column
setColumn(columnText, isFirstColumn, pageSize);
// set starting position
columnText.setYLine(pageSize.getHeight());
int status = ColumnText.START_COLUMN;
while(ColumnText.hasMoreText(status)) {
status = columnText.go();
isFirstColumn = !isFirstColumn;
// write to the next column
setColumn(columnText, isFirstColumn, pageSize);
// set starting position for the next column
columnText.setYLine(pageSize.getHeight());
if(isFirstColumn) {
// create a new page when all columns are written
doc.newPage();
}
}
doc.close();
}

private void setColumn(ColumnText columnText, boolean isFirstColumn,
Rectangle pageSize) {

```

```

if(isFirstColumn) {
columnText.setSimpleColumn(0, 0, pageSize.getWidth() / 2 - 3,
pageSize.getHeight(), 16, Element.ALIGN_JUSTIFIED);
} else {
columnText.setSimpleColumn(pageSize.getWidth() / 2 + 3, 0,
pageSize.getWidth(), pageSize.getHeight(), 16, Element.ALIGN_JUSTIFIED);
}
}
}

```

The following is the output:

```

import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;

import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Element;
import com.itextpdf.text.PageSize;
import com.itextpdf.text.Phrase;
import com.itextpdf.text.Rectangle;
import com.itextpdf.text.pdf.ColumnText;
import com.itextpdf.text.pdf.PdfContentByte;
import com.itextpdf.text.pdf.PdfWriter;

public class ColumnTextExample {
    public static void main(String[] args) {
        try {
            File pdfFile = new File("columnText.pdf");
            File file = new File("ColumnTextExample.java");
            new ColumnTextExample().generatePdf(pdfFile, file);
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }

    public void generatePdf(File pdfFile, File file) throws
DocumentException,
IOException {
    // landscape mode
    Rectangle pageSize = PageSize.LETTER.rotate();
    Document doc = new Document(pageSize);
    PdfWriter writer = PdfWriter.getInstance(doc, new
FileOutputStream(pdfFile));
    doc.open();
    PdfContentByte content = writer.getDirectContent();
    ColumnText columnText = new ColumnText(content);

    BufferedReader reader =
new BufferedReader(new FileReader(file));
    String line;
    while((line = reader.readLine()) != null) {
        columnText.addText(new Phrase(line + "\n"));
    }
    reader.close();
    // the first column
    boolean isFirstColumn = true;
    // define a rectangular column
    setColumn(columnText, isFirstColumn, pageSize);
    // set starting position
    columnText.setYLine(pageSize.getHeight());
    int status = ColumnText.START_COLUMN;
    while(ColumnText.hasMoreText(status)) {
        status = columnText.go();
        isFirstColumn = !isFirstColumn;
        // write to the next column
        setColumn(columnText, isFirstColumn, pageSize);
        // set starting position for the next column
        columnText.setYLine(pageSize.getHeight());
        if(isFirstColumn) {
            // create a new page when all columns are written
            doc.newPage();
        }
    }
    doc.close();

    private void setColumn(ColumnText columnText, boolean
isFirstColumn,
Rectangle pageSize) {
    if(isFirstColumn) {
        columnText.setSimpleColumn(0, 0, pageSize.getWidth() / 2 - 3,
pageSize.getHeight(), 16, Element.ALIGN_JUSTIFIED);
    } else {
        columnText.setSimpleColumn(pageSize.getWidth() / 2 + 3, 0,
pageSize.getWidth(), pageSize.getHeight(), 16,

```

The setSimpleColumn method allows you to define a rectangular column. But, the ColumnText class allows a column to have any shape by using:

void setColumns(float[] leftLine, float[] rightLine): leftLine defines the left bound of the column and rightLine defines the right bound of the column. Each array contains line points such as {x1, y1, x2, y2, ...} to form a line for any shape. In each array, at least two line points (four array elements) are needed to form a straight line.

So, if you want to use the setColumns method in this example, you can use the following arrays for column bounds:

```

float[][] leftLines = new float[][]{
    {0, 0, 0, pageSize.getHeight()},

```

```
{pageSize.getWidth() / 2 + 3, 0, pageSize.getWidth() / 2 + 3, pageSize.getHeight()}}};  
float[][] rightLines = new float[][]{  
    {pageSize.getWidth() / 2 - 3, 0, pageSize.getWidth() / 2 - 3, pageSize.getHeight()},  
    {pageSize.getWidth(), 0, pageSize.getWidth(), pageSize.getHeight()}}};
```

And, replace `setSimpleColumn` with `setColumns`:

```
if(isFirstColumn) {  
    columnText.setColumns(leftLines[0], rightLines[0]);  
} else {  
    columnText.setColumns(leftLines[1], rightLines[1]);  
}
```

Since leading and alignment were defined in the `setSimpleColumn` method, you can define them through a `ColumnText`:

```
columnText.setLeading(16);  
columnText.setAlignment(Element.ALIGN_JUSTIFIED);
```

MERGING DOCUMENTS

Since the PdfStamper class allows you to add extra content to an existing document, it provides a better way to merge documents. You start with the original document by constructing a PdfStamper and read a document to be merged as extra content. By using the insertPage() method, you can either insert a blank page before certain page or append to the end of a document if the page number is bigger than the total number of pages. Next step is to use the getImportPage() method to import a page from the other document and then add this page to direct content of newly created blank page. The following example demonstrates how to merge two documents:

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

import com.itextpdf.text.pdf.PdfContentByte;
import com.itextpdf.text.pdf.PdfImportedPage;
import com.itextpdf.text.pdf.PdfReader;
import com.itextpdf.text.pdf.PdfStamper;

public class MergeExample {
    public static void main(String[] args) {
        try {
            PdfReader origReader = new PdfReader("table.pdf");
            PdfReader otherReader = new PdfReader("paragraph.pdf");
            BufferedOutputStream outStream =
                new BufferedOutputStream(new FileOutputStream("merge.pdf"));
            PdfStamper stamper = new PdfStamper(origReader, outStream);
            int pageNum = origReader.getNumberOfPages();
            for(int i = 1; i <= otherReader.getNumberOfPages(); i++) {
                stamper.insertPage(++pageNum, otherReader.getPageSize(i));
                PdfImportedPage page = stamper.getImportedPage(otherReader, i);
                PdfContentByte content = stamper.getUnderContent(pageNum);
                content.addTemplate(page, 0, 0);
            }
        }
    }
}
```

```
stamper.close();
} catch(Exception ex) {
System.out.println(ex);
}
}
}
```

If you need to insert an existing PDF document during the process of creating a new PDF document, you can use the `getDirectContent()` method from existing `PdfWriter` to get a `PdfContentByte`. The following is the code snippet:

```
PdfReader reader = new PdfReader("appendix.pdf");
PdfContentByte content = writer.getDirectContent();
for(int i = 1; i <= reader.getNumberOfPages(); i++) {
    doc.newPage();
    PdfImportedPage page = writer.getImportedPage(reader, i)
    content.addTemplate(page, 0, 0);
}
```

FILLING FORMS

Interactive PDF forms allow users to fill in form data through freely available Adobe Reader. There are two types of PDF forms: AcroForm (Acrobat form) and XFA form (Adobe XML Forms Architecture form). How do we fill in form data programmatically? The tricky part is to get field names correctly. First, we can fill in a form with dummy data with value starting from 1. If you have Adobe Acrobat, you can export form data as an XML file (Forms -> Manage Form Data -> Export Data). We use an IRS W-4 form as an example. It is an XFA form. To make it simple, we are only interested in some fields in the form:

1	Your first name and middle initial	Last name	2	Your social security number		
1	Home address (number and street or rural route)		3	<input type="checkbox"/> Single <input checked="" type="checkbox"/> Married <input type="checkbox"/> Married, but withhold at higher Single rate. Note. If married, but legally separated, or spouse is a nonresident alien, check the "Single" box.		
4	City or town, state, and ZIP code		4	If your last name differs from that shown on your social security card, check here. You must call 1-800-772-1213 for a replacement card. <input checked="" type="checkbox"/>		
5	Total number of allowances you are claiming (from line H above or from the applicable worksheet on page 2)		5	6		
6	Additional amount, if any, you want withheld from each paycheck		6	\$ 7		
7	I claim exemption from withholding for 2013, and I certify that I meet both of the following conditions for exemption. • Last year I had a right to a refund of all federal income tax withheld because I had no tax liability, and • This year I expect a refund of all federal income tax withheld because I expect to have no tax liability. If you meet both conditions, write "Exempt" here		7	8		
Under penalties of perjury, I declare that I have examined this certificate and, to the best of my knowledge and belief, it is true, correct, and complete.						
Employee's signature (This form is not valid unless you sign it.) ▶			Date ▶			
8	Employer's name and address (Employer: Complete lines 8 and 10 only if sending to the IRS.)		9	Office code (optional)	10	Employer identification number (EIN)
9			10		11	

The following is part of data XML exported from Adobe Acrobat:

```
<?xml version="1.0" encoding="UTF-8"?>
- <topmostSubform>
  <ColumnOne xfa:dataNode="dataGroup" xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <ColumnTwo xfa:dataNode="dataGroup" xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <ColumnThree xfa:dataNode="dataGroup" xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <f1_01_0_>
  <f1_02_0_>
  <f1_03_0_>
  <f1_04_0_>
  <f1_05_0_>
  <f1_06_0_>
  <f1_07_0_>
  <f1_08_0_>
  <PageOneHeader xfa:dataNode="dataGroup" xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  - <Line1>
    <f1_09_0_>1</f1_09_0_>
    <f1_10_0_>2</f1_10_0_>
    <f1_14_0_>4</f1_14_0_>
    <f1_15_0_>5</f1_15_0_>
  </Line1>
  <f1_13_0_>3</f1_13_0_>
  <c1_02>1</c1_02>
  <f1_16_0_>6</f1_16_0_>
  <f1_17_0_>7</f1_17_0_>
  <f1_18_0_>8</f1_18_0_>
  <f1_19_0_>9</f1_19_0_>
  <f1_20_0_>10</f1_20_0_>
  <f1_22_0_>11</f1_22_0_>
  <f2_01_0_>
  <f2_02_0_>
```

Now, we run the following Java program to retrieve form fields and field values:

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
```

```

import java.util.Set;

import com.itextpdf.text.pdf.AcroFields;
import com.itextpdf.text.pdf.PdfReader;
import com.itextpdf.text.pdf.PdfStamper;
import com.itextpdf.text.pdf.XfaForm;

public class ReadFormExample {
    public static void main(String[] args) {
        try {
            PdfReader reader = new PdfReader("fw4_tmp.pdf");
            AcroFields form = reader.getAcroFields();
            XfaForm xfa = form.getXfa();
            System.out.println(xfa.isXfaPresent() ? "XFA form" : "AcroForm");
            Set<String> fields = form.getFields().keySet();
            for(String key : fields) {
                System.out.println("field:" + key + " value:" + form.getField(key));
            }
            reader.close();
        } catch(Exception ex) {
            System.out.println(ex);
        }
    }
}

```

The following fields are part of the output:

```

field:topmostSubform[0].Page1[0].Line1[0].f1_09_0_[0] value:1
field:topmostSubform[0].Page1[0].Line1[0].f1_10_0_[0] value:2
field:topmostSubform[0].Page1[0].c1_01[0] value:2
field:topmostSubform[0].Page1[0].f1_16_0_[0] value:6

```

As you can see, field names are not exactly the same as those in the data XML. By comparing with the W-4 form filled with dummy numbers, you can get field name for corresponding field in the form. The following example shows how to fill in data at some

fields:

```
import java.io.BufferedOutputStream;
```

```
import java.io.FileOutputStream;
```

```
import com.itextpdf.text.pdf.AcroFields;
```

```
import com.itextpdf.text.pdf.PdfReader;
```

```
import com.itextpdf.text.pdf.PdfStamper;
```

```
public class FillFormExample {  
    public static void main(String[] args) {  
        try {  
            PdfReader reader = new PdfReader("fw4.pdf");  
            BufferedOutputStream outStream =  
            new BufferedOutputStream(new FileOutputStream("fw4_filled.pdf"));  
            // keep the original version and append mode  
            PdfStamper stamper = new PdfStamper(reader, outStream, '\0', true);  
            AcroFields form = stamper.getAcroFields();  
            // first name  
            form.setField("topmostSubform[0].Page1[0].Line1[0].f1_09_0_[0]", "Cheng-Hung");  
            // last name  
            form.setField("topmostSubform[0].Page1[0].Line1[0].f1_10_0_[0]", "Chou");  
            // checkbox  
            form.setField("topmostSubform[0].Page1[0].c1_01[0]", "2");  
            // allowance  
            form.setField("topmostSubform[0].Page1[0].f1_16_0_[0]", "10");  
            stamper.close();  
        } catch (Exception ex) {  
            System.out.println(ex);  
        }  
    }  
}
```

In this example, a PdfStamper is constructed as:

```
PdfStamper stamper = new PdfStamper(reader, outStream, '\0', true);
```

The third argument '\0' indicates keeping the same version as the original document. The fourth argument is set as true to use append mode. This is very important. If not, you will not be able to fill this form in Adobe Reader anymore.

SERVLET

If you need to provide PDF documents through a server (e.g., a servlet container), you can create a servlet that takes requests from users and generates PDF documents on the fly. Instead of writing output to a file, it writes PDF content to a temporary buffer and then sends back to the user after the PDF document is generated. In the response, it informs the browser that the content type is PDF and the filename is xxx. The user can either choose to save it as a file or open it through a PDF reader if it is available. The following is the code snippet:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    response.setContentType("application/pdf");
    response.setHeader("Content-Disposition", "attachment;filename=file.pdf");
    Document doc = new Document(PageSize.LETTER);
    // create a PDF writer that listens to this document and writes to a temporary buffer
    ByteArrayOutputStream buffer = new ByteArrayOutputStream();
    PdfWriter.getInstance(doc, buffer);
    // open this document
    doc.open();
    // write document content
    ...
    // close this document
    doc.close();
    // output the writer as bytes to the response output
    DataOutput output = new DataOutputStream(response.getOutputStream());
    byte[] bytes = buffer.toByteArray();
    response.setContentLength(bytes.length);
    output.write(bytes);
    output.flush();
    output.close();
}
```


Apache POI

Apache POI (Poor Obfuscation Implementations) is the Java API for Microsoft Documents, which allows you to read and write MS Office documents such as Excel, Word and PowerPoint using Java. Through POI, you can generate MS Office documents dynamically based on requests or to generate personalized reports on the fly. POI supports OLE2 files such as XLS, DOC and PPT and new XML based standard Office OpenXML (OOXML) files such as XLSX, DOCX and PPTX. Apache POI contains the following major components:

Excel: HSSF (Horrible Spreadsheet Format) for XLS and XSSF (XML Spreadsheet Format) for XLSX

Word: HWPFF (Horrible Word Processor Format) for DOC and XWPF (XML Word Processor Format) for DOCX

PowerPoint: HSLF (Horrible Slide Layout Format) for PPT and XSLF (XML Slide Layout Format) for PPTX

Other than document formats mentioned above, POI also supports Visio, Publisher and Outlook document formats.

You can download Apache POI from <http://poi.apache.org>. The latest version is 3.9 at the time of writing.

CREATING AN EXCEL DOCUMENT

First, we begin with creating an Excel document. To create an Excel document using POI, we break down the process into the following steps:

Step 1

First, you need to create an Excel workbook. A workbook is an Excel file that contains one or more worksheets. You can create a workbook which is represented by the Workbook interface using either the HSSFWorkbook (for HSSF) or XSSFWorkbook class (for XSSF). For creating a workbook, you can use the default constructor. For example,

```
Workbook wb = new HSSFWorkbook();
```

Note: You should always try to use interface as the return type instead of using concrete class as possible. It can minimize effort to switch implementation (HSSF or XSSF) or to support both.

Step 2

A workbook can contain more than one sheet. A sheet is represented by the Sheet interface. There are different types of sheets such as worksheets and chart sheets. Mainly, you will be working on the worksheet which contains a grid of cells. A Sheet object is created by a Workbook by using the createSheet(String name) method. For example, to create a sheet with name “my worksheet”, you can use:

```
Sheet sheet = wb.createSheet(“my worksheet”);
```

A sheet name must be unique in the workbook and contains no more than 31 characters. Also, you can use WorkbookUtil.createSafeSheetName(String nameProposal) to create a valid name that can be safely used in the createSheet() method.

Step3

Once a sheet is created, you can start creating rows using the createRow(int row) method. The row number is zero-based. For example, to create a row at the second row of a worksheet, you can use:

```
Row row = sheet.createRow(1);
```

Step 4

Now, you can add cells into a row by using the createCell(int column) method. Cells are the most important component in a worksheet. You will spend most of the time to define them while creating an Excel document. The column number is zero-based. For example, to create a cell at the second cell of a worksheet, you can use:

```
Cell cell = row.createCell(1);
```

Cells are where data are stored. There are different types of cells. A cell value can be a number, string, date, hyperlink or formula. You can set a cell value by using the following

methods:

```
void setCellValue(boolean value)
```

```
void setCellValue(Calendar value)
```

```
void setCellValue(Date value)
```

```
void setCellValue(double value)
```

```
void setCellValue(RichTextStringValue value)
```

```
void setCellValue(String value)
```

```
void setCellValue(Hyperlink value)
```

For a cell containing a formula, you can use the following method to define a formula:

```
void setCellFormula(String formula)
```

A cell style (represented by the `CellStyle` interface) defines the layout, font and format of a cell. A cell style is created through a workbook by using the `createCellStyle()` method and then using the `setCellStyle(CellStyle style)` method provided by the `Cell` class to set the style for a cell.

Step 5

The final step is to write this workbook to an `OutputStream`. For example, to write to a file you can use:

```
FileOutputStream out = new FileOutputStream("po.xls");
```

```
wb.write(out);
```

```
out.close();
```

Or, to generate an Excel document in a servlet and download it as a file, you can use the following code snippet:

```
response.setContentType("application/excel;");
```

```
response.setHeader("Content-Disposition", "attachment;filename=po.xls");
```

```
ByteArrayOutputStream buffer = new ByteArrayOutputStream();
```

```
wb.write(buffer);
```

```
DataOutput output = new DataOutputStream(response.getOutputStream());
```

```
byte[] bytes =buffer.toByteArray();
```

```
response.setContentLength(bytes.length);
```

```
output.write(bytes);
```

```
output.flush();
```

```
output.close();
```

The following is an example showing how to create an Excel document. To compile and run this example, you need to include poi-3.9-20121203.jar in the classpath for POI 3.9. For XSSF, you need poi-3.9-20121203.jar and poi-ooxml-3.9-20121203.jar to compile it. To run it, in addition to those two JARs, you also need poi-ooxml-schemas-3.9-20121203.jar, ooxml-lib /xmlbeans-2.3.0.jar and ooxml-lib/dom4j-1.6.1.jar.

```
import java.io.FileOutputStream;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;
//import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.CellStyle;
import org.apache.poi.ss.usermodel.CreationHelper;
import org.apache.poi.ss.usermodel.Font;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;

public class SimpleExcel {
    public static CellStyle numberStyle;
    public static void main(String[] args) {
        try {
            Workbook wb = new HSSFWorkbook();
            //Workbook wb = new XSSFWorkbook();
            Sheet sheet = wb.createSheet("purchase order");
            CreationHelper createHelper = wb.getCreationHelper();
            numberStyle = wb.createCellStyle();
            numberStyle.setAlignment(CellStyle.ALIGN_RIGHT);
            numberStyle.setDataFormat(
                createHelper.createDataFormat().getFormat("##,###.00"));
            int startRow = 1;
            short startCol = 1;
            SimpleExcel simple = new SimpleExcel();
            // header row
```



```

simple.createHeaderRow(wb, sheet, startRow, startCol);
// data rows
simple.createDataRow(wb, sheet, startCol,
“Acer ICONIA TAB A700”, 399.95, 2);
simple.createDataRow(wb, sheet, startCol,
“ASUS TF700T”, 425, 5);
// output
FileOutputStream out = new FileOutputStream(“po.xls”);
//FileOutputStream out = new FileOutputStream(“po1.xlsx”);
wb.write(out);
out.close();
} catch(Exception ex) {
System.out.println(ex);
}
}

```

```

public void createHeaderRow(Workbook wb, Sheet sheet, int startRow,
short startCol) {
short col = startCol;
Row headerRow = sheet.createRow(startRow);
sheet.setColumnWidth(col++, (short)5120);
sheet.setColumnWidth(col++, (short)2560);
sheet.setColumnWidth(col++, (short)2560);
sheet.setColumnWidth(col++, (short)2560);
Font headerFont = wb.createFont();
headerFont.setBoldweight(Font.BOLDWEIGHT_BOLD);
headerFont.setFontHeightInPoints((short)12);
CellStyle headerStyle = wb.createCellStyle();
headerStyle.setFont(headerFont);
col = startCol;
Cell cell = headerRow.createCell(col++);
cell.setCellValue(“Product Name”);

```

```

cell.setStyle(headerStyle);
cell = headerRow.createCell(col++);
cell.setCellValue("Price");
cell.setStyle(headerStyle);
cell = headerRow.createCell(col++);
cell.setCellValue("Quantity");
cell.setStyle(headerStyle);
cell = headerRow.createCell(col++);
cell.setCellValue("Subtotal");
cell.setStyle(headerStyle);
}

```

```

public void createDataRow(Workbook wb, Sheet sheet, short startCol,
String name, double price, int quantity) {
int rowNum = sheet.getLastRowNum() + 1;
Row dataRow = sheet.createRow(rowNum);
short col = startCol;
dataRow.createCell(col++).setCellValue(name);
Cell cell = dataRow.createCell(col++);
cell.setCellValue(price); // price
cell.setStyle(numberStyle);
cell = dataRow.createCell(col++);
cell.setCellValue(quantity); // quantity
cell.setStyle(numberStyle);
cell = dataRow.createCell(col++); // subtotal
cell.setStyle(numberStyle);
}
}

```

You can open this document in MS Excel and the following is the screen shot:

	A	B	C	D	E	F
1						
2		Product Name	Price	Quantity	Subtotal	
3		Acer ICONIA TAB A700	399.95	2.00		
4		ASUS TF700T	425.00	5.00		
5						

In this example, the first row is the header row. Column widths are defined here. You cannot set individual cell width. But, you can set column width through a Sheet using the following methods:

`void setColumnWidth(int column, int width):` sets the column width in units of 1/256th of a character width. The maximum column width is 255 characters.

`void autoSizeColumn(int column):` adjusts the column width to fit the content.

Other than setting column width, the following are useful methods that you can use in a worksheet:

`void createFreezePane(int colSplit, int rowSplit):` creates a freeze pane that keeps an area visible when you scroll in the worksheet.

`void setDisplayGridLines(boolean show):` sets it as false if not to display grid lines.

`void setDisplayRowColHeadings(boolean show):` sets it as false if not to display row and column headers.

`PrintSetup getPrintSetup():` gets the print setup. The `PrintSetup` allows you to set up page size and page layout of a worksheet. For example, to use letter page size, you can use `setPageSize(PrintSetup.LETTER_PAGESIZE)`. To set page orientation as landscape, you can use `setLandscape(true)`.

There are two cell styles in this example. One is to define font for the header. The other is for the numbers. It is a good practice to reuse cell styles in the code. You can change a cell style by using methods defined in the `CellStyle`. The following are some of them:

`void setAlignment(short align):` sets horizontal alignment for the cell. For example, `CellStyle.ALIGN_CENTER`.

`void setDataFormat(short fmt):` sets data format through a format index. For built-in formats, you can find them in the `BuiltinFormats` class. You can create a custom data format through a `CreationHelper`.

`void setFillBackgroundColor(short bg):` sets background fill color. Indexed colors are defined in the `IndexedColors` class. For example, `IndexedColors.AQUA.getIndex()`.

`void setFillForegroundColor(short fg):` sets fill foreground color. Indexed colors are defined in the `IndexedColors` class.

`void setFillPattern(short fp):` sets fill pattern. Available patterns are defined in the `CellStyle`. For example, `CellStyle.FINE_DOTS`.

`void setFont(Font font):` sets a `Font` created from the `createFont()` method in `Workbook`.

`void setRotation(short degree):` sets the degree of rotation between -90 and 90 degrees.

`void setVerticalAlignment(short alignment)`: sets vertical alignment for the cell. For example, `CellStyle.VERTICAL_CENTER`.

`void setWrapText(Boolean wrapped)`: sets it as true if the text should wrapped.

You can get a `CreationHelper` from a `Workbook`. The advantage of using the `getCreationHelper()` method to get a `CreationHelper` is that you do not need to worry about if you are dealing with `HSSF` or `XSSF`. `CreationHelper` is an interface. The `HSSFCreationHelper` and `XSSFCreationHelper` classes implement the `CreationHelper` interface. The following are some methods defined in the `CreationHelper`:

`DataFormat createDataFormat()`: creates a `DataFormat` representing a built-in or user defined data format. For user-defined data formats, format patterns follow those defined in core Java API such as the `DecimalFormat` or `SimpleDateFormat` class. You can use the `getFormat(String format)` method from the `DataFormat` to get a format index. If such format does not exist, a new format index is created.

`RichTextString createRichTextString(String text)`: creates a `RichTextString` representing a rich text Unicode string.

ADDING FORMULAS

In the previous example, a table was created and the Subtotal column was left empty. To calculate subtotal for each product, you can define a formula in the cell of Subtotal column. To define a formula, you need the string representation of cell references. You can get that kind of information by either taking a look at the Excel document in previous example or using the CellReference class. The following is the modified version of createDataRow():

```
public void createDataRow(Workbook wb, Sheet sheet, short startCol,
    String name, double price, int quantity) {
    int rowNum = sheet.getLastRowNum() + 1;
    Row dataRow = sheet.createRow(rowNum);
    short col = startCol;
    dataRow.createCell(col++).setCellValue(name);
    Cell cell = dataRow.createCell(col++);
    cell.setCellValue(price); // price
    cell.setCellStyle(numberStyle);
    CellReference priceCellRef = new CellReference(cell);
    cell = dataRow.createCell(col++);
    cell.setCellValue(quantity); // quantity
    cell.setCellStyle(numberStyle);
    CellReference quantCellRef = new CellReference(cell);
    cell = dataRow.createCell(col++); // subtotal
    cell.setCellFormula(priceCellRef.formatAsString() + "*" +
        quantCellRef.formatAsString());
    cell.setCellStyle(numberStyle);
}
```

A new method is also added to calculate the total (a summation of subtotal cells):

```
public void createTotalRow(Workbook wb, Sheet sheet, short startCol) {
    int firstDataRowNum = sheet.getFirstRowNum() + 1;
    Row row = sheet.getRow(firstDataRowNum);
    short subtotalCol = (short)(row.getLastCellNum() - 1);
```

```

int rowNum = sheet.getLastRowNum();
// use absolute cell reference with sheet name
CellReference cellRef1 = new CellReference(sheet.getSheetName(),
firstDataRowNum, subtotalCol, true, true);
CellReference cellRef2 = new CellReference(null,
rowNum, subtotalCol, true, true);
Name name = wb.createName();
name.setNameName("subtotal");
name.setRefersToFormula(cellRef1.formatAsString() + ":" +
cellRef2.formatAsString());
rowNum++;
Row totalRow = sheet.createRow(rowNum);
Cell cell = totalRow.createCell(subtotalCol); // total
cell.setCellFormula("SUM(subtotal)");
cell.setCellStyle(numberStyle);
}

```

You can open this document in MS Excel and the following is the screen shot:

	A	B	C	D	E	F
1						
2		Product Name	Price	Quantity	Subtotal	
3		Acer ICONIA TAB A700	399.95	2.00	799.90	
4		ASUS TF700T	425.00	5.00	2,125.00	
5					2,924.90	
6						

To define a formula, you need the string representation of cell references to identify the location of cells. Cell references can be either relative (such as A1) or absolute (such as \$A\$1). A relative cell reference is changed when it is copied and pasted to other cell. To retrieve a cell, you need both row and column information. You can construct a CellReference either using string representation or row and column information from the following constructors:

CellReference(Cell cell): constructs a cell reference using a Cell.

CellReference(int row, short col): constructs a cell reference using row and cell numbers.

CellReference(String cellRef): constructs a cell reference using a string representation such as A1 or sheet1!A1.

CellReference(String sheetName, int row, short col, boolean absoluteRow, boolean absoluteCol): constructs a cell reference using row and cell numbers with a sheet name.

Once a CellReference is created, you can use the following methods to get cell reference either in string representation or row and column numbers:

`int getRow():` gets row number.

`short getCol():` gets column number.

`String getSheetName():` gets sheet name. It can be null if sheet name is not specified.

`String formatAsString():` gets the string representation of a cell reference.

An alternative to (absolute) cell references is named ranges for a range of cells. Named ranges provide meaningful names in formulas. A formula is used as a reference to a name. The Cell reference in the formula needs to include sheet name. For example, to create a named range “subtotal”, you can use the following code snippet:

```
Name name = wb.createName();
```

```
name.setNameName(“subtotal”);
```

```
name.setRefersToFormula(“’sheet 1’!$E$3:$E$4”);
```

FORMULA EVALUATION

We have discussed about how to create an Excel document already. It starts with creating an empty Workbook. How about loading an existing Excel document? It is a similar process. But, a Workbook is created by loading an Excel document through an InputStream. For example,

```
Workbook wb = new XSSFWorkbook(new FileInputStream("chartdata.xlsx"));
```

To access a worksheet in the workbook, you can use either `getSheet(String name)` or `getSheetAt(int index)`. The following example shows how to load the Excel document created in the previous example and also shows how to get the cell value of a cell:

```
import java.io.FileInputStream;
```

```
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
```

```
import org.apache.poi.ss.usermodel.Cell;
```

```
import org.apache.poi.ss.usermodel.Row;
```

```
import org.apache.poi.ss.usermodel.Sheet;
```

```
import org.apache.poi.ss.usermodel.Workbook;
```

```
import org.apache.poi.ss.util.CellReference;
```

```
public class FormulaEvalExcel {
```

```
    public static void main(String[] args) {
```

```
        try {
```

```
            Workbook wb = new XSSFWorkbook(new FileInputStream("po1.xlsx"));
```

```
            Sheet sheet = wb.getSheetAt(0);
```

```
            CellReference cellRef = new CellReference("E5");
```

```
            Row row = sheet.getRow(cellRef.getRow());
```

```
            Cell cell = row.getCell(cellRef.getCol());
```

```
            FormulaEvalExcel1 formula = new FormulaEvalExcel1();
```

```
            formula.getCellValue(cell);
```

```
        } catch(Exception ex) {
```

```
            System.out.println(ex);
```

```
        }
```

```
    }
```



```

public void getCellValue(Cell cell) {
int cellType = cell.getCellType();
int cellValueType = cellType;
if(cellType == Cell.CELL_TYPE_FORMULA)
cellValueType = cell.getCachedFormulaResultType();
switch(cellValueType) {
case Cell.CELL_TYPE_NUMERIC:
System.out.println(cell.getNumericCellValue());
break;
case Cell.CELL_TYPE_STRING:
System.out.println(cell.getStringCellValue());
break;
case Cell.CELL_TYPE_BOOLEAN:
System.out.println(cell.getBooleanCellValue());
break;
case Cell.CELL_TYPE_ERROR:
System.out.println("Error code: " + cell.getErrorCellValue());
break;
}
}
}

```

Before you can get cell value of a cell, you need to use the `getCellType()` method to get its cell type. But, this is not going to work if you want to get the cell value of a formula cell. For a formula cell, the evaluated cell value is cached. You need to use the `getCachedFormulaResultType()` method to get type of the cell value. This method only can be used in formula cells since only formula cells have cached results. In this example, cell reference E5 is a formula cell.

Now, you can run this example. You will be surprised that the result is 0.0. It is supposed to be 2924.90. What is going on? It looks fine in MS Excel. That is because Workbook Calculation is set as Automatic by default in MS Excel. You can verify that by clicking on the Office Button at upper-left corner -> Excel Options -> Formulas. That is why formulas are recalculated automatically. In the example of previous section, we only created an Excel document with formulas defined and did not evaluate formulas at all. Unless it is

opened and saved in MS Excel (with calculated results), you are not going to get correct values from formula cells. To fix that in the code, you can use a `FormulaEvaluator` to evaluate formulas in the code. You can get a `FormulaEvaluator` by using the `createFormulaEvaluator()` method from a `CreationHelper`. It is a good practice to have only one `FormulaEvaluator` per workbook to take advantage of caching in `FormulaEvaluator`. You can use the following methods to evaluate formulas:

`CellValue evaluate(Cell cell)`: evaluates a given cell without affecting it. A `CellValue` with evaluated value is returned.

`int evaluateFormulaCell(Cell cell)`: evaluates a given cell and the value is saved. The type of evaluated value is returned.

`Cell evaluateInCell(Cell cell)`: evaluates a given cell and puts back the evaluated value to replace the formula.

`void evaluateAll()`: loops through all cells in all sheets in the workbook to evaluate formula cells and the results are saved.

So, you can modify the example in the previous section a little bit by adding the `evaluateFormulaCell()` method to evaluate formula cells and get values saved. Or, modify above example to use a `FormulaEvaluator` to evaluate the formula cell.

For performance reason, `FormulaEvaluator` keeps previously evaluated intermediate values in a cache. If any related cells are changed (value or formula) between `evaluatexxx` methods, you will not get the updated value unless either the `notifyUpdateCell(Cell cell)` or `clearAllCachedResultValues()` method is called. The following code snippet shows how to avoid that kind of problem:

```
// total cell
CellReference cellRef = new CellReference("E5");
Row row = sheet.getRow(cellRef.getRow());
Cell totalCell = row.getCell(cellRef.getCol());
// after this point, evaluated intermediate values are cached
CellValue value = evaluator.evaluate(totalCell);
// quantity cell
cellRef = new CellReference("D3");
row = sheet.getRow(cellRef.getRow());
Cell quantCell = row.getCell(cellRef.getCol());
quantCell.setCellValue(5);
// need to notify cache that this cell has been changed
evaluator.notifyUpdateCell(quantCell);
value = evaluator.evaluate(totalCell);
```


PLOTTING A CHART

Next, we will talk about how to create an Excel document with charts. This is only available for XSSF. To plot a chart using POI, we can break down the process into the following steps:

Step 1

First, we need data to plot a chart. You can either load data from an existing Excel document or create a new Excel document with data retrieved from some data source.

Step 2

Now, we define where to draw the chart in the worksheet by creating a drawing patriarch. A drawing patriarch is the top level container for drawing. It can be created by the `createDrawingPatriarch()` method in a Sheet. A Drawing object (representing a drawing patriarch) is returned. To define where a chart is located and how big it is, you can use the following method defined in the Drawing to create a client anchor:

`ClientAnchor createAnchor(int dx1, int dy1, int dx2, int dy2, int col1, int row1, int col2, int row2)`: creates a client anchor with (dx1, dy1) or (col1, row1) as the top-left coordinates and (dx2, dy2) or (col2, row2) as the bottom-right coordinates. dx1, dy1, dx2 and dy2 use EMU (English Metric Unit). 914400 EMUs is equal to one inch. col1, col2, row1 and row2 are zero-based.

To convert between points and EMUs, you can use the utility class, `Units`.

Step 3

Once the client anchor is defined, a chart can be created in the drawing patriarch by using the `createChart(ClientAnchor anchor)` method defined in the Drawing. A Chart is returned. Before a chart can be plotted, you can define chart properties by using the following methods defined in the Chart:

`ChartAxisFactory getChartAxisFactory()`: `ChartAxisFactory` is a factory for chart axes. Usually a chart has two axes. A chart axis is represented by `ValueAxis`. You can create a `ValueAxis` by using the `createValueAxis(AxisPosition pos)` method from `ChartAxisFactory`. Available locations are defined in `AxisPosition`. They are `BOTTOM`, `LEFT`, `RIGHT` and `TOP`.

`ChartDataFactory getChartDataFactory()`: `ChartDataFactory` is a factory for different chart types. So far, only scatter chart is supported. You can get a `ScatterChartData` from the `createScatterChartData()` method.

`ChartLegend getOrCreateLegend()`: A chart legend is represented by `ChartLegend`. You can specify the position of a chart legend using the `setPosition(LegendPosition position)` method. Available locations are defined in `LegendPosition`. They are `BOTTOM`, `LEFT`, `RIGHT`, `TOP` and `TOP_RIGHT`.

Step 4

A scatter chart needs two sets of data. One is for X-axis values and the other is for Y-axis values. Two `ChartDataSource` objects which represent X-axis values and Y-axis values respectively are added to a `ScatterChartData` using the `addSerie(ChartDataSource<?> xs, ChartDataSource<? extends Number> ys)` method.

Step 5

Once the chart data and axes are defined, you can call the `plot(ChartData data, ChartAxis... axis)` method defined in the `Chart` to plot the chart.

The following example shows how to plot a scatter chart by reading data from an Excel document with two columns of data:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.apache.poi.ss.usermodel.Chart;
import org.apache.poi.ss.usermodel.ClientAnchor;
import org.apache.poi.ss.usermodel.Drawing;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.ss.usermodel.charts.AxisCrosses;
import org.apache.poi.ss.usermodel.charts.AxisPosition;
import org.apache.poi.ss.usermodel.charts.ChartDataSource;
import org.apache.poi.ss.usermodel.charts.ChartLegend;
import org.apache.poi.ss.usermodel.charts.DataSources;
import org.apache.poi.ss.usermodel.charts.LegendPosition;
import org.apache.poi.ss.usermodel.charts.ScatterChartData;
import org.apache.poi.ss.usermodel.charts.ValueAxis;
import org.apache.poi.ss.util.CellRangeAddress;

public class ChartExcel {
    public static void main(String[] args) {
        try {
            Workbook wb = new XSSFWorkbook(new FileInputStream("chartdata.xlsx"));
```

```

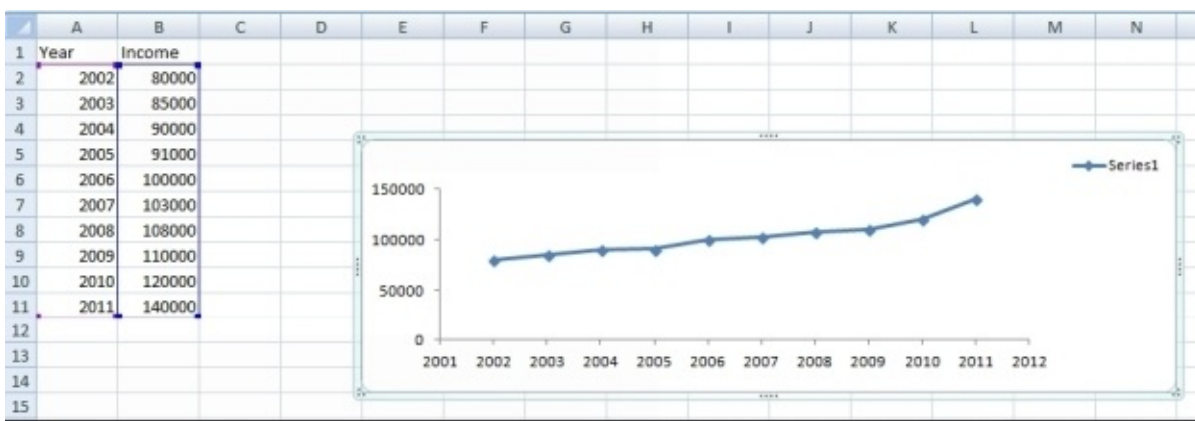
Sheet sheet = wb.getSheetAt(0);
ChartExcel chart = new ChartExcel();
chart.plot(sheet);
// output
FileOutputStream out = new FileOutputStream("chart.xlsx");
wb.write(out);
out.close();
} catch(Exception ex) {
System.out.println(ex);
}
}

public void plot(Sheet sheet) {
int startDataRowNum = sheet.getFirstRowNum() + 1;
int endDataRowNum = sheet.getLastRowNum();
int startColNum = sheet.getRow(startDataRowNum).getFirstCellNum();
// define the drawing location
Drawing drawing = sheet.createDrawingPatriarch();
ClientAnchor anchor = drawing.createAnchor(0, 0, 0, 0, 4, 4, 14, 14);
// define chart
Chart chart = drawing.createChart(anchor);
ChartLegend legend = chart.getOrCreateLegend();
legend.setPosition(LegendPosition.TOP_RIGHT);
ScatterChartData data = chart.getChartDataFactory().createScatterChartData();
ValueAxis bottomAxis =
chart.getChartAxisFactory().createValueAxis(AxisPosition.BOTTOM);
ValueAxis leftAxis =
chart.getChartAxisFactory().createValueAxis(AxisPosition.LEFT);
leftAxis.setCrosses(AxisCrosses.AUTO_ZERO);
// data series at X-Axis
ChartDataSource<Number> xs = DataSources.fromNumericCellRange(sheet,
new CellRangeAddress(startDataRowNum, endDataRowNum, startColNum,
startColNum));

```

```
// data series at Y-axis
ChartDataSource<Number> ys = DataSources.fromNumericCellRange(sheet,
    new CellRangeAddress(startDataRowNum, endDataRowNum, startColNum + 1,
startColNum + 1));
data.addSerie(xs, ys);
// plot chart
chart.plot(data, bottomAxis, leftAxis);
}
}
```

You can open this document in MS Excel and the following is the screen shot:



CREATING A WORD DOCUMENT

Now, we will learn how to create a Word document by using XWPF (.docx). We can break down the process into the following steps:

Step 1

First, we need to create a document that represents a Word document:

```
XWPFDocument doc = new XWPFDocument();
```

Step 2

A document contains many paragraphs. A paragraph is represented by the XWPFParagraph class. A paragraph can be created by using:

```
XWPFParagraph par1 = doc.createParagraph();
```

The XWPFParagraph class defines alignments, indentations, spacing and borders.

Step 3

A paragraph contains many runs. A run is represented by the XWPFRun class which can be created by using:

```
XWPFRun run1 = par1.createRun();
```

The XWPFRun class defines a region of text with common properties such as font size, font family, font styles and color. A run can contain a picture by using the following method:

```
XWPFPicture addPicture(InputStream pictureData, int pictureType, String filename, int width, int height): the available picture types are defined in the Document. For example, Document.PICTURE _TYPE_JPEG. Both width and height are in EMUs.
```

Step 4

The final step is to write this document to an OutputStream. For example, to write to a file you can use:

```
FileOutputStream out = new FileOutputStream("po.docx");
```

```
wb.write(out);
```

```
out.close();
```

The following is an example showing how to create a Word document (.docx). To compile this example, you need to include poi-3.9-20121203.jar, poi-ooxml-3.9-20121203.jar and poi-ooxml-schemas-3.9-20121203.jar in the classpath. To run it, in addition to those three JARs mentioned above, you also need ooxml-lib /xmlbeans-2.3.0.jar and ooxml-lib/dom4j-1.6.1.jar.

```
import java.io.FileOutputStream;
```



```
import org.apache.poi.xwpf.usermodel.Borders;
import org.apache.poi.xwpf.usermodel.ParagraphAlignment;
import org.apache.poi.xwpf.usermodel.UnderlinePatterns;
import org.apache.poi.xwpf.usermodel.XWPFDocument;
import org.apache.poi.xwpf.usermodel.XWPFParagraph;
import org.apache.poi.xwpf.usermodel.XWPFRun;
import org.apache.poi.xwpf.usermodel.XWPFTable;
import org.apache.poi.xwpf.usermodel.XWPFTableCell;
import org.apache.poi.xwpf.usermodel.XWPFTableRow;
```

```
public class WordExample {
    public static void main(String[] args) {
        try {
            WordExample word = new WordExample();
            XWPFDocument doc = new XWPFDocument();
            // title
            XWPFParagraph par1 = doc.createParagraph();
            XWPFRun run1 = par1.createRun();
            run1.setText("Thanks for your order,");
            run1.setBold(true);
            run1.setFontSize(12);
            run1.setFontFamily("Times New Roman");
            run1.addBreak();
            XWPFParagraph par2 = doc.createParagraph();
            par2.setBorderBottom(Borders.BASIC_WIDE_INLINE);
            XWPFRun run2 = par2.createRun();
            run2.setText("Order Summary:");
            run2.setColor("0000ff"); // RRGGBB
            // table
            XWPFTable table = doc.createTable(3, 4);
            table.setCellMargins(10, 50, 10, 50);
```

```

XWPFTableRow row = table.getRow(0);
word.getHeaderRow(row);
double total = 0;
total = total + word.getDataRow(table.getRow(1), "Acer ICONIA TAB A700",
399.95, 2);
total = total + word.getDataRow(table.getRow(2), "ASUS TF700T",
425, 5);
// total
XWPFParagraph totalPar = doc.createParagraph();
totalPar.setSpacingBefore(512); // 2 characters
XWPFRun totalRun = totalPar.createRun();
totalRun.setText(String.format("Total: %8.2f", total));
totalRun.setBold(true);
totalRun.setUnderline(UnderlinePatterns.SINGLE);
totalRun.setFontSize(12);
// output
FileOutputStream out = new FileOutputStream("po.docx");
doc.write(out);
out.close();
} catch(Exception ex) {
System.out.println(ex);
}
}

public void getHeaderRow(XWPFTableRow row) {
XWPFTableCell cell = row.getCell(0);
getCellText(cell, "Product Name", true, 12, ParagraphAlignment.LEFT);
cell = row.getCell(1);
getCellText(cell, "Price", true, 12, ParagraphAlignment.LEFT);
cell = row.getCell(2);
getCellText(cell, "Quantity", true, 12, ParagraphAlignment.LEFT);
cell = row.getCell(3);

```

```
getCellText(cell, "Subtotal", true, 12, ParagraphAlignment.LEFT);  
}
```

```
public double getDataRow(XWPFFTableRow row, String name, double price, int  
quantity) {
```

```
    XWPFFTableCell cell = row.getCell(0);
```

```
    getCellText(cell, name, false, 12, ParagraphAlignment.LEFT);
```

```
    cell = row.getCell(1);
```

```
    getCellText(cell, String.format("%8.2f", price), false, 12,  
ParagraphAlignment.RIGHT);
```

```
    cell = row.getCell(2);
```

```
    getCellText(cell, String.format("%5d", quantity), false, 12,  
ParagraphAlignment.RIGHT);
```

```
    cell = row.getCell(3);
```

```
    double subtotal = price * quantity;
```

```
    getCellText(cell, String.format("%8.2f", subtotal), false, 12,  
ParagraphAlignment.RIGHT);
```

```
    return subtotal;
```

```
}
```

```
public XWPFFParagraph getCellText(XWPFFTableCell cell, String text,  
boolean bold, int fontSize, ParagraphAlignment alignment) {
```

```
    XWPFFParagraph par = cell.addParagraph();
```

```
    par.setAlignment(alignment);
```

```
    XWPFFRun run1 = par.createRun();
```

```
    run1.setText(text);
```

```
    run1.setFontSize(fontSize);
```

```
    run1.setBold(bold);
```

```
    return par;
```

```
}
```

```
}
```

The following is the output:

Thanks for your order,

Order Summary:

Product Name	Price	Quantity	Subtotal
Acer ICONIA TAB A700	399.95	2	799.90
ASUS TF700T	425.00	5	2125.00

Total: 2924.90

Other than creating paragraphs, you can use XWPFDocument to create tables. For example,

```
XWPFTable table = doc.createTable(3, 4);
```

A table is represented by the XWPFTable class. Here, it creates an empty table with 3 rows and 4 columns. If you use the createTable() method, it creates an empty table with one row and one column. Since rows have been created already (with no data), All you need to do is to retrieve them by using the getRow(int pos) method. An XWPFTableRow is returned. If needed, you still can use the createRow() method to add additional rows. A row contains cells. A cell is represented by the XWPFTableCell class. You can retrieve a cell by using the getCell(int pos) method. pos is zero-based. Basically, a cell is constructed by paragraphs. You can add paragraphs into a cell by using the addParagraph() method. An XWPFParagraph is created and returned. So, you can set up properties of a cell just like the way you define paragraphs.

JFreeChart

JFreeChart is a chart library that helps you to create a variety of chart types such as pie charts, bar charts, line charts or scatter plots in your Swing applications. Many output types such as images (JPEG or PNG) are supported. JFreeChart is not just limited to desktop applications. It can be used on the server side such as servlets or JSPs too.

You can download JFreeChart from <http://www.jfree.org/jfreechart>. The latest version is 1.0.14 at the time of writing.

CREATING A SIMPLE CHART

We start with a simple example to demonstrate how to create a chart by using JFreeChart in a Swing application. The following are basic steps to create a chart:

Step 1

For a Swing application, it contains at least one top-level container to contain other Swing components. There are four top-level containers: JFrame, JDialog, JWindow and JApplet. Each top-level container has a content pane which is used to contain visible child components of the top-level container. Usually, the top-level container is a JFrame. If you are creating an applet application, you can use a JApplet.

Step 2

You need to provide the data set for the chart to be created. As for which dataset class to use, it depends on the chart type. The following are major dataset types (based on the Dataset interface):

PieDataset: This is an interface for a dataset that contains a list of key-value pairs. The key should be unique and cannot be null. You can use the DefaultPieDataset class to create a dataset for a pie chart or ring chart.

CategoryDataset: This is an interface for a dataset that contains one or more data series. Each data series has values associated with categories. You can use the DefaultCategoryDataset class to create a dataset for an area chart, bar chart, line chart, multiple-pie chart or waterfall chart.

XYDataset: This is an interface for a dataset that contains data in the form of (x, y). You can use the DefaultXYDataset class to create a dataset for a polar chart, scatter plot or time series chart.

Step 3

Now, you can create a chart by using the dataset from previous step. You can use the JFreeChart class to construct a chart. But, an easier way to create a chart is to use a utility class, ChartFactory, to create a chart. For example, to create a pie chart, you can use:

```
JFreeChart ChartFactory.createPieChart(String title, PieDataset dataset, boolean legend, boolean tooltips, boolean urls)
```

The JFreeChart class represents a JFreeChart chart, which coordinates objects to draw a chart based on the Java 2D API. Those objects include a Dataset mentioned above. It also includes a Plot which is responsible for drawing by using data from a Dataset. The Plot class is the parent class of all plots. Based on those Datasets mentioned above, the PiePlot class is for PieDataset, the CategoryPlot class is for CategoryDataset and the XYPlot class is for XYDataset. You can use the getPlot() method to get the Plot for the chart from a JFreeChart object.

A JFreeChart also contains a list of Title objects (the main title and subtitles including legend). The Title class is the parent class of all titles. You are allowed to define title properties such as alignments (vertically or horizontally) or location (top, bottom, left or right). You can use the TextTitle or CompositeTitle class to render text (or texts) or use the ImageTitle class to render an image. The legend is represented by the LegendTitle class. You can add a subtitle by using the addSubtitle(Title title) method.

Step 4

To display a chart in a Swing application, you can use the ChartPanel class to create a panel for a JFreechart object. The ChartPanel class inherits the JPanel class. To construct a ChartPanel, you can use the following methods:

```
ChartPanel(JFreeChart chart)
```

```
ChartPanel(JFreeChart chart, boolean properties, boolean save, boolean print, boolean zoom, boolean tooltips):
```

 This allows you to control which menu items will be available in the popup menu. All menu items are available by default.

Step 5

You add the chart panel to a container and the remaining is just the usual way you are writing a Swing application.

The following is an example that demonstrates how to create a pie chart by using Java Swing. To compile and run this example, you need to include jfreechart-1.0.14.jar and jcommon-1.0.17.jar under the lib directory of JFreeChart to the classpath:

```
import java.awt.Color;
import java.awt.Container;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.text.NumberFormat;
import javax.swing.JFrame;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.labels.StandardPieSectionLabelGenerator;
import org.jfree.chart.plot.PiePlot;
import org.jfree.chart.title.LegendTitle;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.general.PieDataset;
```



```
import org.jfree.ui.RectangleEdge;

public class PieChartExample {
    public static void main(String[] args) {
        PieChartExample chart = new PieChartExample();

        JFrame frame = new JFrame();
        frame.setTitle("Pie Chart");
        Container content = frame.getContentPane();
        PieDataset dataset = chart.createDataset();
        String title = "A Pie Chart";
        ChartPanel chartPanel = chart.createChartPanel(dataset, title);
        content.add(chartPanel);

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });

        frame.pack();
        frame.setVisible(true);
    }

    public PieDataset createDataset() {
        DefaultPieDataset dataset = new DefaultPieDataset();
        dataset.setValue("Section 1", 29);
        dataset.setValue("Section 2", 20);
        dataset.setValue("Section 3", 40);
        dataset.setValue("Section 4", 11);

        return dataset;
    }
}
```

```

}

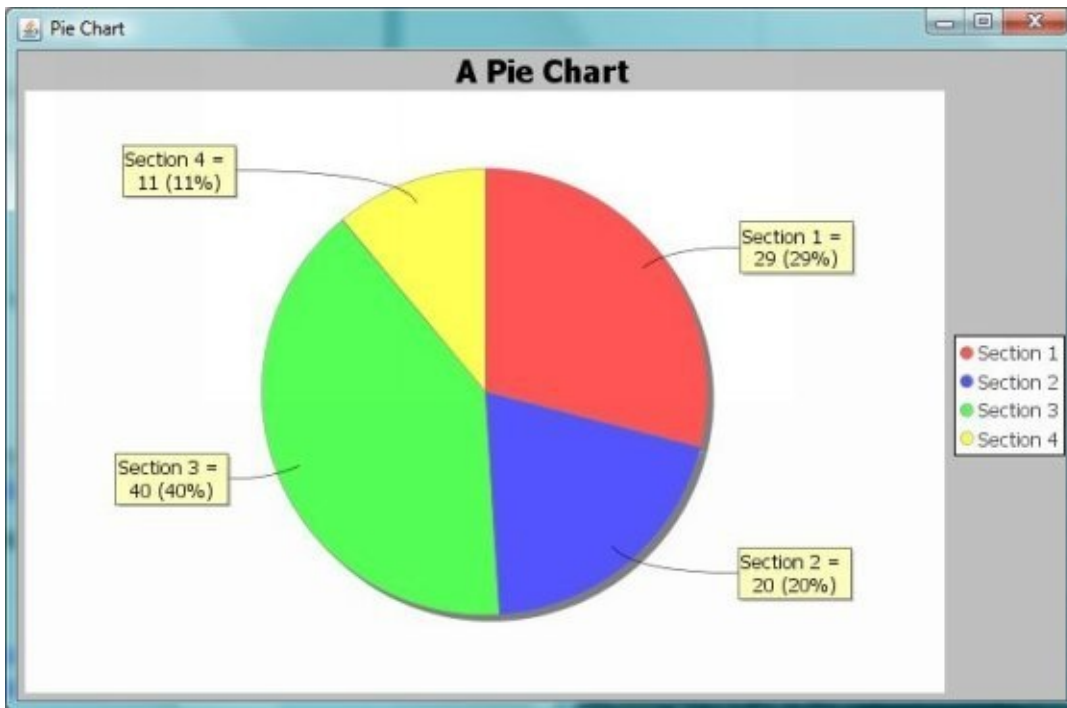
public ChartPanel createChartPanel(PieDataset dataset, String title) {
    JFreeChart chart = ChartFactory.createPieChart(title,
dataset, true, true, false);
    chart.setBackgroundPaint(Color.LIGHT_GRAY);
    LegendTitle legend = chart.getLegend();
    legend.setPosition(RectangleEdge.RIGHT);
    PiePlot plot = (PiePlot)chart.getPlot();
    plot.setBackgroundPaint(Color.WHITE);
    plot.setOutlineVisible(false);
    plot.setCircular(true);
    plot.setLabelGenerator(new StandardPieSectionLabelGenerator(
        "{0} = {1} ({2})", NumberFormat.getNumberInstance(),
        NumberFormat.getPercentInstance()));
    plot.setNoDataMessage("No data available");

    ChartPanel chartPanel = new ChartPanel(chart);

    return chartPanel;
}
}

```

The following is a screenshot of this example. You can bring up the popup menu by right-clicking on the mouse. But, you cannot zoom in or zoom out a pie chart:



By default, the chart legend is located at the bottom. You can use the `setPosition(RectangleEdge position)` method provided by the `LegendTitle` class to change its position. Available values are `RectangleEdge.BOTTOM`, `RectangleEdge.LEFT`, `RectangleEdge.RIGHT` and `RectangleEdge.TOP`.

To change section labels in a pie chart, you can use the `setLabelGenerator(PieSectionLabelGenerator generator)` method provided by the `PiePlot` class. If it is null, there are no section labels. Also, you can use a standard section label generator by providing a format string. For example,

```
setLabelGenerator(new StandardPieSectionLabelGenerator("{0} = {1} ({2})",
NumberFormat.getNumberInstance(), NumberFormat.getPercentInstance()));
```

{0} is for the section key, {1} is for the section value and {2} is for the percentage.

Or, you can create a custom label generator by implementing the `PieSectionLabelGenerator` interface.

Similarly, if you want to control the format of a tooltip, you can use the `setToolTipGenerator(PieToolTipGenerator generator)` method. If it is null, there will be no tooltip. A standard tooltip generator is provided by the `StandardPieToolTipGenerator` class. `StandardPieSectionLabelGenerator` and `StandardPieToolTipGenerator` inherit the same parent class. So, the usage is the same.

To define the starting angle of a pie chart, you can use:

`void setStartAngle(double angle)`: sets the starting angle in degrees. The default value is 90 degrees (at 12 o'clock). A value of zero is at 3 o'clock.

To define the drawing direction, you can use:

`void setDirection(Rotation rotation)`: sets the drawing direction (either `Rotation.CLOCKWISE` or `Rotation.ANTICLOCKWISE`). By default, it is `Rotation.ANTICLOCKWISE`.

To set custom color or color pattern on a section, you can use the `setSectionPaint(Comparable key, Paint paint)` method. For example, the following code snippet sets color on each section:

```
int value = 0;
for(Object key : dataset.getKeys()) {
    plot.setSectionPaint((Comparable)key, new Color(value, 255, value));
    value = value + 50;
}
```

You can create a pie chart with 3D effect by using `ChartFactory.createPieChart3D`. The chart object uses a `PiePlot3D` as the plot.

JFreeChart provides a rich set of charts. We are not able to go through all of them. A trick on using or customizing a chart is that you need to know those classes representing the three basic objects in a chart: plot, axis and renderer. If you are using `ChartFactory` to create a chart, you can find the information from the method creating the chart. For JFreeChart API documentation, you can check <http://www.jfree.org/jfreechart/api/javadoc>. For JCommon API documentation, you can check <http://www.jfree.org/jcommon/api>.

CREATING A BAR CHART

Next, we will talk about using the second type of dataset, `CategoryDataset`. One of the chart types using the `CategoryDataset` is the bar chart. To create a bar chart, you can use:

```
JFreeChart ChartFactory.createBarChart(String title, String categoryAxisLabel, String valueAxisLabel, CategoryDataset dataset, PlotOrientation orientation, boolean legend, boolean tooltips, boolean urls)
```

You use either `PlotOrientation.VERTICAL` to create a vertical bar chart or `PlotOrientation.HORIZONTAL` to create a horizontal bar chart. For a vertical bar chart, the value axis is on the y-direction and the category axis is on the x-direction. The value axis (or range axis), which is represented by the `ValueAxis` (or its subclass `NumberAxis`) class, is an axis for displaying numerical data. Through this class, you can define axis properties such as ticks or value range. The category axis (or domain axis), which is represented by the `CategoryAxis` class, is an axis for displaying categories. Through this class, you can define axis properties such as category label position.

A bar chart can contain more than one data series. You can use the `DefaultCategoryDataset` class to create a dataset for the pie chart by using the following method:

```
void addValue(double value, Comparable rowKey, Comparable columnKey): rowKey is the data series name. columnKey is the category name.
```

To add markers to the plot, you can use the `addRangeMarker(Marker marker, Layer layer)` method. You can use either `Layer.FOREGROUND` or `Layer.BACKGROUND`. To mark a range of values, you can use the `IntervalMarker` class. To mark a single value, you can use the `ValueMarker` class. If you need to mark categories, you can use the `addDomainMarker(CategoryMarker marker, Layer layer)` method.

In the following example, it creates a vertical bar chart. In the bar chart, there are three data series. Each data series has four categories:

```
import java.awt.Color;
import java.awt.Container;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.text.DecimalFormat;
import javax.swing.JFrame;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
```

```
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.axis.NumberTickUnit;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.IntervalMarker;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.data.category.CategoryDataset;
import org.jfree.ui.Layer;
import org.jfree.ui.RectangleAnchor;
import org.jfree.ui.TextAnchor;

public class BarChartExample {
    public static void main(String[] args) {
        BarChartExample chart = new BarChartExample();

        JFrame frame = new JFrame();
        frame.setTitle("Bar Chart");
        Container content = frame.getContentPane();
        CategoryDataset dataset = chart.createDataset();
        String title = "A Bar Chart";
        ChartPanel chartPanel = chart.createChartPanel(dataset, title);
        content.add(chartPanel);

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });

        frame.pack();
        frame.setVisible(true);
    }
}
```

```
}
```

```
public CategoryDataset createDataset() {  
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();  
    double[][] data = new double[][]{  
        {80, 60, 70, 90}, {85, 70, 60, 80}, {90, 55, 80, 95}  
    };  
    String[] series = new String[] {"Series 1", "Series 2", "Series 3"};  
    String[] categories = new String[] {"Category 1", "Category 2",  
        "Category 3", "Category 4"};  
    for(int ser = 0; ser <= 2; ser++) {  
        for(int cat = 0; cat < 4; cat++) {  
            dataset.addValue(data[ser][cat], series[ser], categories[cat]);  
        }  
    }  
  
    return dataset;  
}
```

```
public ChartPanel createChartPanel(CategoryDataset dataset, String title) {  
    JFreeChart chart = ChartFactory.createBarChart(title,  
        "Category", "Value", dataset, PlotOrientation.VERTICAL, true, true, false);  
    chart.setBackgroundPaint(Color.LIGHT_GRAY);  
    CategoryPlot plot = (CategoryPlot)chart.getPlot();  
    plot.setBackgroundPaint(Color.WHITE);  
    plot.setOutlineVisible(false);  
    // grid lines  
    plot.setDomainGridlinePaint(Color.BLACK); // for the category axis  
    plot.setDomainGridlinesVisible(true);  
    plot.setRangeGridlinePaint(Color.BLACK); // for the value axis  
    // ticks on the value axis  
    NumberAxis valueAxis = (NumberAxis)plot.getRangeAxis();
```

```

valueAxis.setMinorTickMarksVisible(true);
valueAxis.setTickUnit(new NumberTickUnit(20, new DecimalFormat("#,##0.00"), 4));
valueAxis.setUpperBound(100);
valueAxis.setLowerBound(0);
// interval markers
IntervalMarker target = new IntervalMarker(90, 100);
target.setLabel("Excellent");
target.setLabelAnchor(RectangleAnchor.BOTTOM_LEFT);
target.setLabelTextAnchor(TextAnchor.BASELINE_LEFT);
target.setPaint(new Color(0, 255, 0, 50));
plot.addRangeMarker(target, Layer.BACKGROUND);
target = new IntervalMarker(80, 90);
target.setLabel("Great");
target.setLabelAnchor(RectangleAnchor.BOTTOM_LEFT);
target.setLabelTextAnchor(TextAnchor.BASELINE_LEFT);
target.setPaint(new Color(0, 150, 0, 50));
plot.addRangeMarker(target, Layer.BACKGROUND);

plot.setNoDataMessage("No data available");

ChartPanel chartPanel = new ChartPanel(chart);

return chartPanel;
}
}

```

The following is a screenshot of this example. It has two highlighted intervals created by the IntervalMarker class:



JFreechart has a utility class related to datasets. It can be handy for creating a `CategoryDataset` if there are many data series. For example, to create a `CategoryDataset`, you can use:

```
CategoryDataset DatasetUtilities.createCategoryDataset(Comparable[] rowKeys,
Comparable[] columnKeys, double[][] data)
```

For the dataset in this example, you can use `DatasetUtilities.createCategoryDataset(series, categories, data)`.

To use custom color or color pattern on bars, you can do that through a `BarRenderer`. You can get a `BarRenderer` from the `getRenderer()` method provided by the `CategoryPlot` class. To set the paint for a series, you can use the `setSeriesPaint(int series, Paint paint)` method from the `BarRenderer` class. The series index is zero-based.

If the data series contains date (or time) data, you can use the `setRangeAxis(ValueAxis axis)` method from the `CategoryPlot` class to pass in a `DateAxis`. The `DateAxis` is a subclass of `ValueAxis`, which is used to display dates. The actual data stored in the dataset are still numbers (the data type is `long`). But, those numbers are formatted as dates. The subclasses (e.g., `Day`, `Hour`) of the `RegularTimePeriod` class provide convenient ways to calculate certain time periods or are used to represent a certain date or time.

You can create a bar chart with 3D effect by using `ChartFactory.createBarChart3D`. The chart object uses a `CategoryPlot` as the plot. The value axis is represented by the `NumberAxis3D` class. The category axis is represented by the `CategoryAxis3D` class.

CREATING A SCATTER PLOT

In the following example, we will show how to create a scatter plot using the third type of dataset, `XYDataset`. A scatter plot is used to plot numerical data. Both axes are `NumberAxis` and the default render is an `XYLineAndShapeRenderer` (of type `XYItemRenderer`). In this example, the x-direction axis is for date. So, it is replaced by a `DateAxis`. A `TimeSeries` is used to contain a sequence of data for a period of time. The `TimeSeriesCollection` class implements `XYDataset`. It is used to contain a collection of `TimeSeries` objects.

The renderer for a scatter plot is an `XYLineAndShapeRenderer` which is used to draw data points and connect data points together. You can use it to customize how a series is plotted. Or, you can replace it with a built-in renderer by using the `setRenderer(XYItemRenderer renderer)` method. There are many renderers that implement the `XYItemRenderer` interface. For example, the `StandardXYItemRenderer` class is for drawing points and/or lines, the `XYBarRenderer` class is for drawing bars and the `XYAreaRenderer` class is for drawing filled areas.

Just like previous example, you can mark the plot with markers. Here, we use the `ValueMarker` class to mark single value. There are four `ValueMarker` objects on the domain axis and one `ValueMarker` object on the range axis. If you need to annotate a specific data point, you can use the `addAnnotation(XYAnnotation annotation)` method.

```
import java.awt.Color;
import java.awt.Container;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.geom.Ellipse2D;
import java.text.DecimalFormat;
import java.text.SimpleDateFormat;
import javax.swing.JFrame;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.axis.NumberTickUnit;
import org.jfree.chart.plot.Marker;
```

```
import org.jfree.chart.plot.ValueMarker;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.time.Month;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
import org.jfree.data.xy.XYDataset;
import org.jfree.ui.RectangleAnchor;
import org.jfree.ui.TextAnchor;

public class ScatterPlotExample {
    public static void main(String[] args) {
        ScatterPlotExample chart = new ScatterPlotExample();

        JFrame frame = new JFrame();
        frame.setTitle("Scatter Plot");
        Container content = frame.getContentPane();
        XYDataset dataset = chart.createDataset();
        String title = "A Scatter Plot";
        ChartPanel chartPanel = chart.createChartPanel(dataset, title);
        content.add(chartPanel);

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });

        frame.pack();
        frame.setVisible(true);
    }
}
```

```

public XYDataset createDataset() {
    TimeSeries s1 = new TimeSeries("Series 1");
    s1.add(new Month(1, 2012), 70.1);
    s1.add(new Month(2, 2012), 67.5);
    s1.add(new Month(3, 2012), 65.2);
    s1.add(new Month(4, 2012), 55.9);
    s1.add(new Month(5, 2012), 60.4);
    s1.add(new Month(6, 2012), 66.0);
    s1.add(new Month(7, 2012), 75.2);
    s1.add(new Month(8, 2012), 80.6);
    s1.add(new Month(9, 2012), 81.8);
    s1.add(new Month(10, 2012), 85.1);
    s1.add(new Month(11, 2012), 88.3);
    s1.add(new Month(12, 2012), 91.7);

    TimeSeriesCollection dataset = new TimeSeriesCollection();
    dataset.addSeries(s1);

    return dataset;
}

```

```

public ChartPanel createChartPanel(XYDataset dataset, String title) {
    JFreeChart chart = ChartFactory.createScatterPlot(title,
        "Date", "Value", dataset, PlotOrientation.VERTICAL, true, true, false);
    XYPlot plot = (XYPlot)chart.getPlot();
    // grid lines
    plot.setDomainGridlinePaint(Color.BLACK); // for the month axis
    plot.setDomainGridlinesVisible(true);
    plot.setRangeGridlinePaint(Color.BLACK); // for the value axis
    // ticks on the value axis
    NumberAxis valueAxis = (NumberAxis)plot.getRangeAxis();
}

```

```

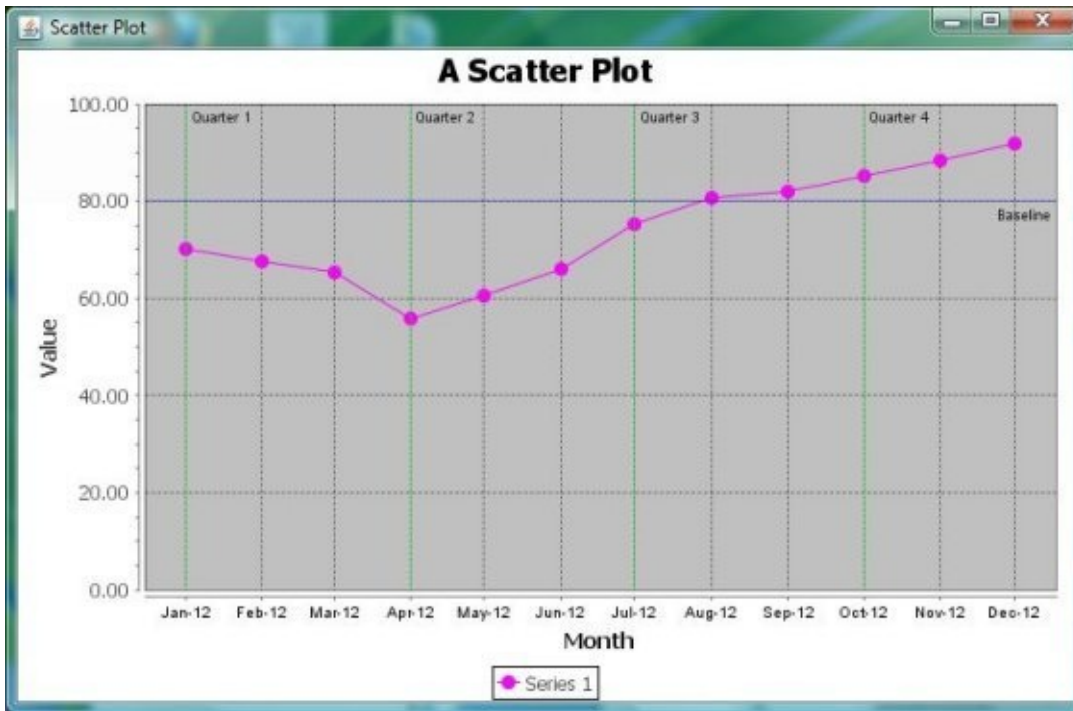
valueAxis.setMinorTickMarksVisible(true);
valueAxis.setTickUnit(new NumberTickUnit(20, new DecimalFormat("#,##0.00"), 4));
valueAxis.setUpperBound(100);
valueAxis.setLowerBound(0);
// replace original NumberAxis by DateAxis
DateAxis dateAxis = new DateAxis("Month");
dateAxis.setLabelFont(valueAxis.getLabelFont());
plot.setDomainAxis(dateAxis);
dateAxis.setDateFormatOverride(new SimpleDateFormat("MMM-yy"));
// markers
for(int i = 0; i <= 3; i++) {
Month month = new Month(i*3+1, 2012);
Marker marker = new ValueMarker(month.getFirstMillisecond());
marker.setPaint(Color.GREEN);
marker.setLabel("Quarter " + (i+1));
marker.setLabelAnchor(RectangleAnchor.TOP_RIGHT);
marker.setLabelTextAnchor(TextAnchor.TOP_LEFT);
plot.addDomainMarker(marker);
}
Marker marker = new ValueMarker(80);
marker.setPaint(Color.BLUE);
marker.setLabel("Baseline");
marker.setLabelAnchor(RectangleAnchor.BOTTOM_RIGHT);
marker.setLabelTextAnchor(TextAnchor.TOP_RIGHT);
plot.addRangeMarker(marker);
// renderer
XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer)plot.getRenderer();
renderer.setSeriesPaint(0, Color.MAGENTA);
renderer.setSeriesShape(0, new Ellipse2D.Double(-4, -4, 8, 8));
renderer.setSeriesLinesVisible(0, true);

ChartPanel chartPanel = new ChartPanel(chart);

```

```
return chartPanel;  
}  
}
```

The following is a screenshot of this example. It has four markers on the domain axis and one marker on the range axis:



CREATING A COMBINED CHART

So far, we only discuss about using the ChartFactory class to create a chart. Now, we will discuss how to use the JFreeChart class to create a chart. Also, we will learn how to combine charts together with a common range or domain axis. You can use the following constructors to create a combined plot:

`CombinedDomainCategoryPlot(CategoryAxis domainAxis)` : A combined category plot where the domain axis is shared.

`CombinedDomainXYPlot(ValueAxis valueAxis)` : A combined XY plot where the domain axis is shared.

`CombinedRangeCategoryPlot(ValueAxis valueAxis)` : A combined category plot where the range axis is shared.

`CombinedRangeXYPlot(ValueAxis valueAxis)`: A combined XY plot where the range axis is shared.

Once the combined plot is constructed, you can add plots into it. Take a `CombinedDomainXYPlot` as an example, you can use the `add(XYPlot plot)` or `add(XYPlot plot, int weight)` method. The weight determines how much space a plot is occupied relative to others. The default value is 1. To construct an `XYPlot`, you can use:

`XYPlot(XYDataset dataset, ValueAxis domainAxis, ValueAxis rangeAxis, XYItemRenderer renderer)`: If this plot is going to be added into a `CombinedDomainXYPlot`, `domainAxis` is null. If it is going to be added into a `CombinedRangeXYPlot`, `rangeAxis` is null.

To construct a `CategoryPlot`, you can use:

`CategoryPlot(CategoryDataset dataset, CategoryAxis domainAxis, ValueAxis rangeAxis, CategoryItemRenderer renderer)`: If this plot is going to be added into a `CombinedDomainCategoryPlot`, `domainAxis` is null. If it is going to be added into a `CombinedRangeCategoryPlot`, `rangeAxis` is null.

To construct a combined chart, you can use the following constructors:

`JFreeChart(String title, Font font, Plot plot, boolean createLegend)`

`JFreeChart(String title, Plot plot)`

In this example, we are using data from the previous example to draw a chart with two plots with the same data and shared domain axis. One is a scatter plot with lines connecting data points together and the other is using filled areas:

```
import java.awt.Color;
```

```
import java.awt.Container;
```

```
import java.awt.event.WindowAdapter;
```

```
import java.awt.event.WindowEvent;
import java.text.DecimalFormat;
import java.text.SimpleDateFormat;
import javax.swing.JFrame;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.axis.NumberTickUnit;
import org.jfree.chart.plot.CombinedDomainXYPlot;
import org.jfree.chart.plot.XYPlot;
import org.jfree.data.time.Month;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
import org.jfree.chart.renderer.xy.StandardXYItemRenderer;
import org.jfree.chart.renderer.xy.XYAreaRenderer;
import org.jfree.chart.renderer.xy.XYItemRenderer;
import org.jfree.data.xy.XYDataset;

public class CombinedChartExample {
    public static void main(String[] args) {
        CombinedChartExample chart = new CombinedChartExample();

        JFrame frame = new JFrame();
        frame.setTitle("Combined Chart");
        Container content = frame.getContentPane();
        String title = "A Combined Chart";
        ChartPanel chartPanel = chart.createChartPanel(title);
        content.add(chartPanel);
    }
}
```



```
frame.addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent we) {  
        System.exit(0);  
    }  
});
```

```
frame.pack();  
frame.setVisible(true);  
}
```

```
public XYDataset createDataset(String name) {  
    TimeSeries s1 = new TimeSeries(name);  
    s1.add(new Month(1, 2012), 70.1);  
    s1.add(new Month(2, 2012), 67.5);  
    s1.add(new Month(3, 2012), 65.2);  
    s1.add(new Month(4, 2012), 55.9);  
    s1.add(new Month(5, 2012), 60.4);  
    s1.add(new Month(6, 2012), 66.0);  
    s1.add(new Month(7, 2012), 75.2);  
    s1.add(new Month(8, 2012), 80.6);  
    s1.add(new Month(9, 2012), 81.8);  
    s1.add(new Month(10, 2012), 85.1);  
    s1.add(new Month(11, 2012), 88.3);  
    s1.add(new Month(12, 2012), 91.7);
```

```
    TimeSeriesCollection dataset = new TimeSeriesCollection();  
    dataset.addSeries(s1);  
  
    return dataset;  
}
```

```
public XYPlot createXYPlot(XYDataset dataset, String valueAxisLabel,
```

```

XYItemRenderer renderer) {
    XYPlot plot = new XYPlot(dataset, null, new NumberAxis(valueAxisLabel),
    renderer);
    // grid lines
    plot.setDomainGridlinePaint(Color.BLACK); // for the month axis
    plot.setDomainGridlinesVisible(true);
    plot.setRangeGridlinePaint(Color.BLACK); // for the value axis
    // ticks on the value axis
    NumberAxis valueAxis = (NumberAxis)plot.getRangeAxis();
    valueAxis.setMinorTickMarksVisible(true);
    valueAxis.setTickUnit(new NumberTickUnit(20, new DecimalFormat("#,##0.00"), 4));
    valueAxis.setUpperBound(100);
    valueAxis.setLowerBound(0);

    return plot;
}

```

```

public ChartPanel createChartPanel(String title) {
    // create a combined domain plot on DateAxis
    DateAxis dateAxis = new DateAxis("Month");
    dateAxis.setDateFormatOverride(new SimpleDateFormat("MMM-yy"));
    CombinedDomainXYPlot combinedPlot = new CombinedDomainXYPlot(dateAxis);
    combinedPlot.setGap(15);
    // plot 1
    XYDataset dataset1 = createDataset("Series 1");
    XYPlot plot1 = createXYPlot(dataset1, "Value 1",
    new StandardXYItemRenderer(StandardXYItemRenderer.SHAPES_AND_LINES));
    combinedPlot.add(plot1, 1);
    // plot 2
    XYDataset dataset2 = createDataset("Series 2");
    XYPlot plot2 = createXYPlot(dataset2, "Value 2", new XYAreaRenderer());
    combinedPlot.add(plot2, 1);
}

```

```
JFreeChart chart = new JFreeChart(title, JFreeChart.DEFAULT_TITLE_FONT,  
combinedPlot, true);
```

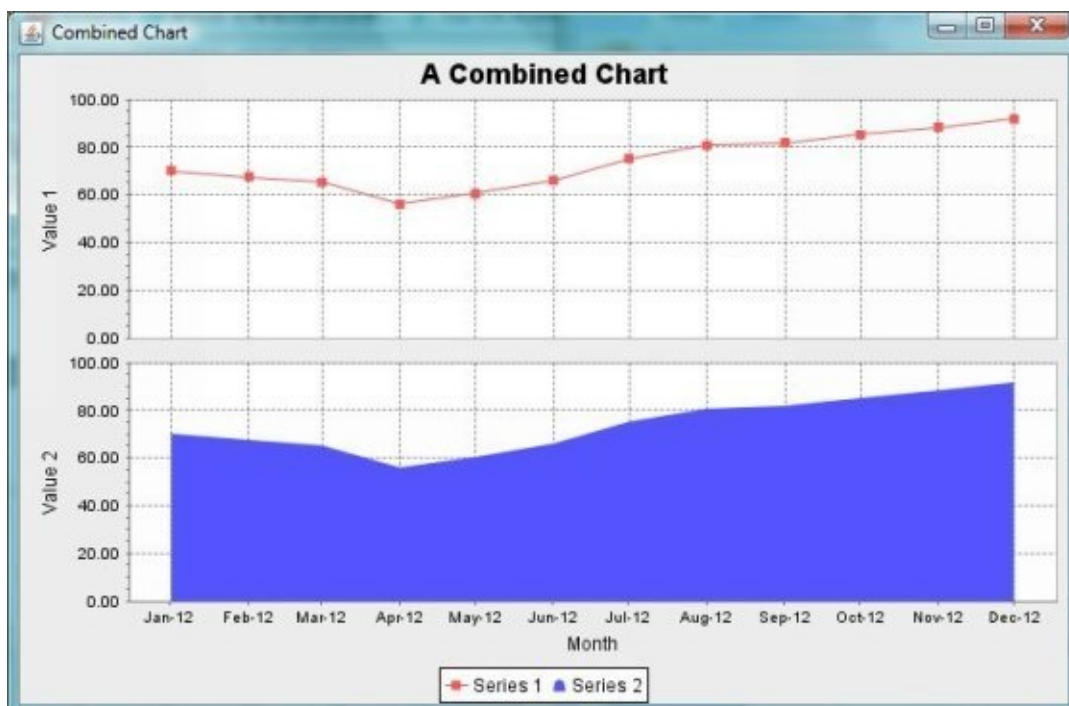
```
ChartPanel chartPanel = new ChartPanel(chart);
```

```
return chartPanel;
```

```
}
```

```
}
```

The following is the screenshot of this example:



MAKING A CHART CLICKABLE

To make a chart respond to a mouse event, you can register a `ChartMouseListener` to the `ChartPanel` by using the `addChartMouseListener(ChartMouseListener listener)` method. When the chart is clicked, the `ChartPanel` will send a `ChartMouseEvent` notification to the registered listener.

The following code snippet shows how to make a pie chart clickable. When any section is clicked, it pops up a dialog showing the section name:

```
final ChartPanel chartPanel = new ChartPanel(chart);
chartPanel.addChartMouseListener(new ChartMouseListener() {
    public void chartMouseClicked(ChartMouseEvent event) {
        PieSectionEntity entity = (PieSectionEntity)event.getEntity();
        JOptionPane.showMessageDialog(chartPanel, entity.getSectionKey());
    }

    public void chartMouseMoved(ChartMouseEvent event) {
    }
});
```

You can get the entity information related to this mouse event by using the `getEntity()` method. A `ChartEntity` is returned. The `ChartEntity` class is the parent class of chart entity classes. The actual entity class depends on the chart type. For a pie chart, it is the `PieSectionEntity` class.

Other useful mouse behavior you can enable is the mouse wheel behavior. To enable it, you can use the `setMouseWheelEnabled(boolean flag)` method. For a pie chart, you can rotate the chart once it is enabled. For a chart with axes, you can zoom in or zoom out if the chart is zoomable.

DISPLAYING A CHART IN A WEB PAGE

It is not required to have GUI to use JFreeChart. You can generate a chart and write it out directly without showing it on the screen. JFreeChart provides a utility class, ChartUtilities, which can convert charts into certain image formats (JPEG or PNG). To save a chart to an image file, you can use the following static methods:

saveChartAsJPEG(File file, float quality, JFreeChart chart, int width, int height): The quality is in the range 0.0f to 1.0f.

saveChartAsJPEG(File file, JFreeChart chart, int width, int height)

saveChartAsPNG(File file, JFreeChart chart, int width, int height)

You also can choose to write it to an OutputStream:

writeChartAsJPEG(OutputStream out, float quality, JFreeChart chart, int width, int height)

writeChartAsJPEG(OutputStream out, JFreeChart chart, int width, int height)

writeChartAsPNG(OutputStream out, JFreeChart chart, int width, int height)

To use JFreeChart in a web application, you can use those methods to display a chart on a web page by writing it as an image. The following is the code snippet of a servlet:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    ...
    response.setContentType("image/png");
    OutputStream out = response.getOutputStream();
    ChartUtilities.writeChartAsPNG(out, chart, 640, 480);
    output.flush();
    output.close();
}
```

The ChartUtilities class also can generate an image map that can be used in an HTML page. An image map contains a list of coordinates related to an image. It allows you to make a chart clickable or show tooltips from the web page. To write an image map to an output stream, you can use:

writeImageMap(PrintWriter writer, String name, ChartRenderingInfo info, boolean userOverLibForToolTips)

To generate an image map, you need to provide a ChartRenderingInfo. A ChartRenderingInfo contains information about dimensions and entities of a chart. You

can collect chart information by passing in a `ChartRenderingInfo` to some `saveXXX` or `writeXXX` method.

In the following example, an HTML page is created with an image map of a pie chart. To make each section on the pie chart clickable, you need to enable URL generation while using `ChartFactory` to create the pie chart. To define the base URL for each section, you can use the `setURLGenerator(PieURLGenerator generator)` method to assign a `PieURLGenerator`. There are two classes that implement the `PieURLGenerator` interface: the `StandardPieURLGenerator` and `CustomPieURLGenerator` classes. The `StandardPieURLGenerator` class is the default URL generator for the `PiePlot`. You can use the following constructors to construct a `StandardPieURLGenerator`:

```
StandardPieURLGenerator(String prefix)
```

```
StandardPieURLGenerator(String prefix, String categoryParamName)
```

```
StandardPieURLGenerator(String prefix, String categoryParamName, String  
indexParamName)
```

Parameter `prefix` is for the base URL. Parameter `categoryParamName` is to define the query parameter for the category name. Parameter `indexParamName` is to define the query parameter for the pie index. For example, if the prefix is `http://www.my.site/mychart.html`, the URL for one of the sections looks like the following:

```
http://www.my.site/mychart.html?category=Section+4&pieIndex=0
```

Also, you can use the `CustomPieURLGenerator` class to assign a map containing (key, URL) mapping by using the `addURLs(Map urlMap)` method. For a `MultiplePiePlot`, you can assign multiple maps.

```
import java.io.BufferedOutputStream;
```

```
import java.io.File;
```

```
import java.io.FileOutputStream;
```

```
import java.io.IOException;
```

```
import java.io.OutputStream;
```

```
import java.io.PrintWriter;
```

```
import org.jfree.chart.ChartFactory;
```

```
import org.jfree.chart.ChartRenderingInfo;
```

```
import org.jfree.chart.ChartUtilities;
```

```
import org.jfree.chart.JFreeChart;
```

```
import org.jfree.chart.entity.StandardEntityCollection;
```

```
import org.jfree.chart.labels.StandardPieSectionLabelGenerator;
```

```
import org.jfree.chart.plot.PiePlot;
```

```

import org.jfree.chart.urls.StandardPieURLGenerator;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.general.PieDataset;

public class ImageMapExample {
    public static void main(String[] args) {
        try {
            ImageMapExample chart = new ImageMapExample();
            PieDataset dataset = chart.createDataset();
            String title = "A Pie Chart";
            JFreeChart jfchart = chart.createChart(dataset, title);
            ChartRenderingInfo info = new ChartRenderingInfo(new StandardEntityCollection());
            // save the chart as an image
            File imageFile = new File("piechart.png");
            ChartUtilities.saveChartAsPNG(imageFile, jfchart, 640, 480, info);
            // create an HTML page with an image map
            File htmlFile = new File("piechart.html");
            OutputStream out = new BufferedOutputStream(new FileOutputStream(htmlFile));
            PrintWriter writer = new PrintWriter(out);
            writer.println("<HTML>");
            writer.println("<HEAD><TITLE>A Pie Chart</TITLE></HEAD>");
            writer.println("<BODY>");
            ChartUtilities.writeImageMap(writer, "chart", info, false);
            writer.println("<IMG SRC=\"piechart.png\" WIDTH=\"640\" " +
                "<HEIGHT=\"480\" BORDER=\"0\" USEMAP=\"#chart\">");
            writer.println("</BODY>");
            writer.println("</HTML>");
            writer.close();
        } catch (IOException ex){
            System.out.println(ex);
        }
    }
}

```

```
public PieDataset createDataset() {
    DefaultPieDataset dataset = new DefaultPieDataset();
    dataset.setValue("Section 1", 29);
    dataset.setValue("Section 2", 20);
    dataset.setValue("Section 3", 40);
    dataset.setValue("Section 4", 11);

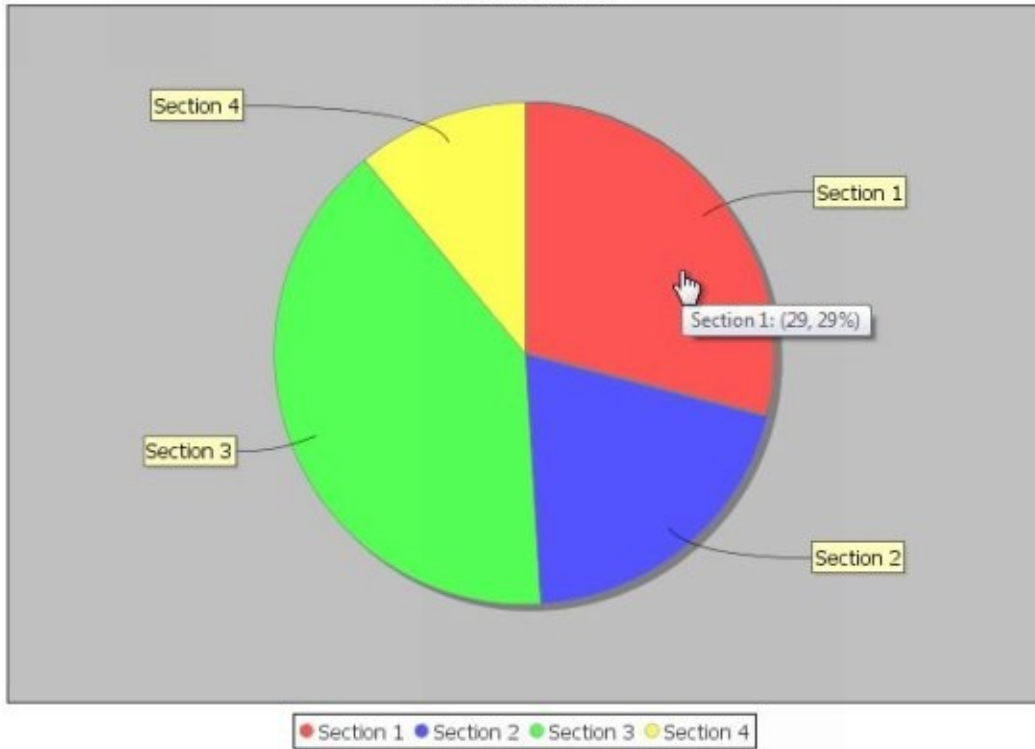
    return dataset;
}

public JFreeChart createChart(PieDataset dataset, String title) {
    JFreeChart chart = ChartFactory.createPieChart(title,
        dataset, true, true, true); // enable URLs
    PiePlot plot = (PiePlot)chart.getPlot();
    // set base URL
    plot.setURLGenerator(new
    StandardPieURLGenerator("http://www.my.site/mychart.html"));

    return chart;
}
}
```

The following is an HTML page with an image map showing on a browser:

A Pie Chart



Similarly, the `CategoryPlot` class allows you to assign a `CategoryURLGenerator` (the `StandardCategoryURLGenerator` or `CustomCategoryURLGenerator` class) and the `XYPlot` class allows you to assign a `XYURLGenerator` (the `StandardXYURLGenerator` or `CustomXYURLGenerator` class).

EasyMock, JUnit, and PowerMock

Unit tests are written by programmers to test classes or methods internally in programmer's perspective. Each test should be independent from each other and should be tested without any dependencies. But, objects do not work alone. In a real system, objects work together. And, some scenarios are difficult to create or reproduce. It may involve interaction with database or network. To set up the whole environment can be complicated. If we are not doing integration testing, how do we do unit testing in isolation without any dependencies? We need to incorporate stub or mock objects in unit testing for dependencies. Both types of objects have the same interface as the real objects. Stub objects are objects that simply return prearranged responses. Mock objects are objects that mimic the real objects in controlled ways for different scenarios. They can help to decide if a test is either failed or passed. It can be a tedious job no matter what type of objects you are going to create.

EasyMock is a framework that can save you time in hand wiring mock objects and can create mock objects at runtime. JUnit is a unit testing framework. JUnit and EasyMock can work together easily. PowerMock is a mock framework that extends other mock frameworks. PowerMock extends EasyMock with features such as mocking on private, static, or final methods. Currently, it supports EasyMock and Mockito.

You can download EasyMock from <http://easymock.org>. The latest version is 3.4 at the time of writing.

You can download JUnit from <http://junit.org>. The latest stable version is 4.12 at the time of writing.

You can download PowerMock for EasyMock and JUnit from <https://github.com/jayway/powermock>. The latest version is 1.6.4 at the time of writing.

AN INTRODUCTION TO JUNIT

EasyMock is used in conjunction with JUnit. Before we start learning how to use EasyMock, let's talk about how to use JUnit to create unit tests first. A JUnit test case (or test class) is defined through a Java class. A test case contains many tests. You also can combine test cases together as a test suite.

Annotations

Starting from JUnit 4 (JSE 5.0 and above is required), annotations are supported:

@BeforeClass: indicates that a method is executed once before the start of all tests. The annotated methods need to be public static void and no parameters. **@BeforeClass** breaks the concept that tests should be independent from each other. However, you can use this to initialize expensive resources such as database connections. You do not want to do that for each test.

@AfterClass: indicates that a method is executed once after the end of all tests. Usually, it is used to release resources created in **@BeforeClass**. The annotated methods need to be public static void and no parameters. They are guaranteed to run even if any **@BeforeClass** method throws an exception.

@Before: indicates that a method is executed before each **@Test** method. The annotated methods need to be public void and no parameters.

@After: indicates that a method is executed after each **@Test** method. The annotated methods need to be public void and no parameters. They are guaranteed to run even if any **@Before** or **@Test** method throws an exception.

@Test: indicates that a method is a test. A test method needs to be declared as public void and no parameters. There are two optional parameters available for **@Test**:

The first one is timeout parameter which is to specify timeout in milliseconds. If the time is exceeded, a `TimeoutException` is thrown. For example, to specify the timeout as 5 seconds, you can use `@Test(timeout=5000)`.

The second one is expected parameter which is to specify expected exception to be thrown. If the exception is not thrown or a different one is thrown, the test is failed. For example, to specify that `IllegalArgumentException` is expected to be thrown, you can use `@Test(expected=IllegalArgumentException.class)`.

@Ignore: To ignore a test, you can either remove **@Test** or comment it out. If you need to test runners to report number of ignored tests, you can add **@Ignore** in front of or after **@Test**. **@Test** can be used in the test class to ignore the whole test case too. If you want to add comment on the ignored test, you can use something such as `@Ignore("a string")`.

@FixMethodOrder: Tests should be independent from each other and should not assume any execution order. Test execution order is undefined in JUnit. The execution order depends on JVM. Even though order dependent tests are not encouraged. But, in version 4.11, you can use **@FixMethodOrder** to change test execution order in a test case. Available options are: `MethodSorters.DEFAULT` (a more predictable default behavior), `MethodSorters.JVM` (pre-4.11 behavior), `MethodSorters.NAME_ASCENDING`.

The following is the basic structure of a test case:

```
public class MyTestCase {
```

```

@BeforeClass
public static void setUpClass() {
...
}
@AfterClass
public static void tearDownClass() {
...
}
@Before
public void setUp() {
...
}
@After
public void tearDown() {
...
}
@Test
public void testMethod1() {
...
}
@Test
public void testMethod2() {
...
}
}

```

The following is execution order of above test case:

```

setUpClass()
setUp()
testMethod1()
tearDown()
setUp()

```

```
testMethod2()
tearDown()
tearDownClass()
```

A test fixture contains a common set of objects used by tests in a test case. It is just like the initialization of a test. You use a test fixture as a baseline in a test case to ensure tests are run in a fixed environment so that test results are repeatable. You can use fixture annotations in a test case without duplicating the same code for each test. There are four fixture annotations: `@BeforeClass`, `@AfterClass`, `@Before` and `@After`.

Assertions

Assertions are used to check that the results of tests are as expected. When an assertion fails, an `AssertionError` is thrown. Assertion methods are defined in the `Assert` class with an optional message for the `AssertionError`. Since assertion methods are declared as static void, you can use them directly such as `Assert.assertFalse(...)`. But, you can use static import (e.g., `import static org.junit.Assert.*`) to make them look better in the code without including the class name. The following are some of assertion methods:

`fail([String message])`: fails a test unconditionally.

`assertFalse([String message,]boolean condition)`: asserts that a condition is false.

`assertTrue([String message,]boolean condition)`: asserts that a condition is true.

`assertNotNull([String message,]Object object)`: asserts that an object is not null.

`assertNull([String message,]Object object)`: asserts that an object is null.

`assertNotSame([String message,]Object expected, Object actual)`: asserts that expected and actual do not refer to the same object.

`assertSame([String message,]Object expected, Object actual)`: asserts that expected and actual refer to the same object (object reference equality).

`assertEquals([String message,]Object expected, Object actual)`: asserts that expected and actual are equal (object value equality through implementing equals method).

`assertEquals([String message,]double expected, double actual, double delta)`: asserts that expected and actual are equal within a positive delta.

`assertArrayEquals([String message,]Object[] expecteds, Object[] actuals)`: asserts that expecteds and actuals are equal.

Test Runners

You can run JUnit tests through test runners. To run tests (or test suites) from a Java program, you can use:

```
org.junit.runner.Result org.junit.runner.JUnitCore.runClasses(java.lang.Class<?>...
classes)
```

To run tests from the command line, you can use:

```
org.junit.runner.JUnitCore {test case class 1}[ test case class 2...]
```

The default test runner is BlockJUnit4ClassRunner. It can be replaced by using the following annotation:

```
@RunWith
```

For example, to combine test cases in a suite, you can use the following annotations in a class before the class declaration:

```
@RunWith(Suite.class)
```

```
@Suite.SuiteClasses({TestCase1.class, TestCase2.class, ...})
```

For example,

```
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)
```

```
@Suite.SuiteClasses({MyTestCase.class, PurchaseManagerTest.class})
```

```
public class MyTestSuite {
```

```
    @BeforeClass
```

```
    public static void setUpClass() {
```

```
        ...
```

```
    }
```

```
    @AfterClass
```

```
    public static void tearDownClass() {
```

```
        ...
```

```
}  
}
```

JUnit will use the class specified in `@RunWith` to run the tests instead of the built-in runner. You still can use `@BeforeClass` or `@AfterClass` in a test suite. `@BeforeClass` will run before `@BeforeClass` in all test cases and `@AfterClass` will run after `@AfterClass` in all test cases. To start a suite runner, you do that through JUnitCore mentioned before.

To provide unit testing on objects that are constructed and initialized through Spring application context, you can use `SpringJUnit4ClassRunner`. For example, you can create a base class for test classes with the following basic structure:

```
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(locations = {  
    ...  
})  
public abstract class MyTestBase implements ApplicationContextAware {  
  
    private ApplicationContext applicationContext;  
  
    @Override  
    public void setApplicationContext(ApplicationContext applicationContext) throws  
BeansException {  
        this.applicationContext = applicationContext;  
    }  
}
```

By implementing `ApplicationContextAware`, a reference of `ApplicationContext` will be supplied to the test instance.

For additional information on JUnit, you can check JUnit API documentation from <http://junit.org/javadoc/latest/index.html>.

A SAMPLE JUNIT TEST CASE

We already have basic understanding on JUnit. Now, let's use a simple example to explain how to create a test case. The following example has a manager class which is responsible for getting product information such as price or inventory through a data provider. A data provider is represented by the `DataProvider` interface. It can be implemented based on a variety of data sources.

```
public interface DataProvider {
    public double getPrice(String productNo);
    public int getInventory(String productNo);
    public boolean exists(String productNo);
}

public class PurchaseManager {
    private DataProvider dataProvider;
    public PurchaseManager(DataProvider dataProvider) {
        this.dataProvider = dataProvider;
    }

    public double getPrice(String productNo) throws ProductNotFoundException {
        if(!existsProduct(productNo))
            throw new ProductNotFoundException("Cannot find product # " + productNo);
        return dataProvider.getPrice(productNo);
    }

    public int getInventory(String productNo) throws ProductNotFoundException {
        if(!existsProduct(productNo))
            throw new ProductNotFoundException("Cannot find product # " + productNo);
        return dataProvider.getInventory(productNo);
    }

    public double getTotal(String productNo, int quantity)
        throws ProductNotFoundException, NotEnoughInventoryException {
```

```
    if(quantity < 1)
    throw new IllegalArgumentException("quantity is less than one.");
    if(!existsProduct(productNo))
    throw new ProductNotFoundException("Cannot find product # " +
productNo);
```

```
    int inventory = dataProvider.getInventory(productNo);
    if(inventory < quantity)
    throw new NotEnoughInventoryException();
    double price = dataProvider.getPrice(productNo);
    return price*quantity;
}
```

```
public boolean existsProduct(String productNo) {
    if(productNo == null)
    throw new IllegalArgumentException("productNo is null.");
    return dataProvider.exists(productNo);
}
}
```

Since this is for unit testing, an actual data provider is not available. So, we create a dummy data provider which uses a map to store testing data as follows:

```
import java.util.HashMap;
import java.util.Map;
```

```
public class DummyDataProvider implements DataProvider {
    private Map<String, Product> data = new HashMap<String, Product>();

    public DummyDataProvider() {
        Product product = new Product("A0001", 150.0, 10);
        data.put(product.getProductNo(), product);
        product = new Product("A0002", 200.0, 5);
        data.put(product.getProductNo(), product);
```

```
}
```

```
public double getPrice(String productNo) {  
    Product product = data.get(productNo);  
    if(product != null)  
        return product.getPrice();  
    else  
        return -1;  
}
```

```
public int getInventory(String productNo) {  
    Product product = data.get(productNo);  
    if(product != null)  
        return product.getInventory();  
    else  
        return -1;  
}
```

```
public boolean exists(String productNo) {  
    return data.containsKey(productNo);  
}
```

```
}
```

```
class Product {  
    String productNo;  
    double price;  
    int inventory;  
  
    Product(String productNo, double price, int inventory) {  
        this.productNo = productNo;  
        this.price = price;  
        this.inventory = inventory;  
    }  
}
```

```
}  
  
String getProductNo() {  
    return productNo;  
}
```

```
double getPrice() {  
    return price;  
}
```

```
int getInventory() {  
    return inventory;  
}
```

```
}
```

Once the external data source is all set, we can create tests for the PurchaseManager class. To compile this example, you need junit-4.12.jar in the classpath. To run it, you need junit-4.12.jar and hamcrest-core-1.3.jar.

```
import static org.junit.Assert.*;  
  
import org.junit.After;  
import org.junit.AfterClass;  
import org.junit.Before;  
import org.junit.BeforeClass;  
import org.junit.Test;  
  
public class PurchaseManagerTest {  
    private static DataProvider provider;  
    private PurchaseManager purchaseMgr;  
  
    @BeforeClass  
    public static void setUpClass() {  
        provider = new DummyDataProvider();  
    }
```

@AfterClass

```
public static void tearDownClass() {  
    provider = null;  
}
```

@Before

```
public void setUp() {  
    purchaseMgr = new PurchaseManager(provider);  
}
```

@After

```
public void tearDown() {  
    purchaseMgr = null;  
}
```

@Test(expected=IllegalArgumentException.class)

```
public void nullProduct() {  
    purchaseMgr.existsProduct(null);  
}
```

@Test

```
public void nullProductException() {  
    try {  
        purchaseMgr.existsProduct(null);  
        fail();  
    } catch (Exception ex) {  
    }  
}
```

@Test

```
public void existsProduct() {
```

```
boolean actual = purchaseMgr.existsProduct("A0001");
assertTrue(actual);
}
```

```
@Test(expected=ProductNotFoundException.class)
public void notFound() throws ProductNotFoundException {
purchaseMgr.getPrice("B0001");
}
```

```
@Test
public void getPrice() throws ProductNotFoundException {
double expected = 150;
double price = purchaseMgr.getPrice("A0001");
assertEquals(expected, price, 0);
}
```

```
@Test
public void getInventory() throws ProductNotFoundException {
int expected = 10;
int inventory = purchaseMgr.getInventory("A0001");
assertEquals(expected, inventory);
}
```

```
@Test
public void getTotal() throws ProductNotFoundException,
NotEnoughInventoryException {
double expected = 150*8;
double total = purchaseMgr.getTotal("A0001", 8);
assertEquals(expected, total, 0);
}
```

```
@Test(expected=NotEnoughInventoryException.class)
```



```
public void notEnoughInventory() throws ProductNotFoundException,
NotEnoughInventoryException {
purchaseMgr.getTotal("A0002", 8);
}
}
```

Now, you can run

```
java org.junit.runner.JUnitCore PurchaseManagerTest
```

The following is the output:

```
JUnit version 4.12
```

```
.....
```

```
Time: 0.016
```

```
OK (8 tests)
```

If there were any failed tests, you would see similar error message such as:

There was 1 failure:

1) getTotal(PurchaseManagerTest)

```
java.lang.AssertionError: expected:<1200.0> but was:<1120.0>
```

```
at org.junit.Assert.fail(Assert.java:88)
```

```
at org.junit.Assert.failNotEquals(Assert.java:834)
```

```
at org.junit.Assert.assertEquals(Assert.java:553)
```

```
at org.junit.Assert.assertEquals(Assert.java:683)
```

```
at PurchaseManagerTest.getTotal(PurchaseManagerTest.java:77)
```

```
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
...
```

FAILURES!!!

Tests run: 8, Failures: 1

AN INTRODUCTION TO EASYMOCK

We just created our first JUnit test case using a mock object (a dummy data provider). The disadvantage to this approach is that you have additional code to create and maintain. Now, we want to get rid of that manual hand wiring step by incorporating EasyMock and JUnit. To create and use a mock object using EasyMock, it involves the following steps:

Step 1

You need to create a mock object that implements the given interface by using `EasyMock.createMock(Class<T> toMock)`.

Step 2

You record the expected behavior of a mock object. It is called record state. In record state, it records expected method calls (or mock methods).

Step 3

Switching from record state to replay state by calling `EasyMock.replay(Object... mocks)`. In this state, it checks whether the expected methods are called or not.

Note: You cannot record your own behavior on `equals()`, `toString()`, `hashCode()` and `finalize()`. EasyMock provides built-in behavior for them.

Note: If you want to reuse a mock object, you can use the `reset(Object... mocks)` method to reset it back to expectation setting mode. It can be handy for mock objects that are expensive to create. You can create them in `@BeforeClass` method and reset them in `@Before` or `@After` method.

Verifying behavior

By default, EasyMock only checks that expected methods with right arguments are called. If you need to verify behavior, you can use `EasyMock.verify(Object... mocks)` in the end of a test. What kind of checking that `verify()` will do depends on how the mock object is created:

On a mock object created by `EasyMock.createMock()`, the order of method calls is not checked. But, all expected methods with specified arguments should be called. An `AssertionError` is thrown for any unexpected calls.

On a mock object created by `EasyMock.createStrictMock()`, the order of method calls is checked. And, all expected methods with specified arguments should be called. An `AssertionError` is thrown for any unexpected calls.

On a mock object created by `EasyMock.createNiceMock()`, the order of method calls is not checked. But, all expected methods with specified arguments should be called. Unexpected method calls are allowed. Depending on data type of returned value, `0` (non-boolean primitive data types), `false` (boolean type) or `null` (Object and its subtypes) is returned for unexpected method calls.

You can switch order checking on and off by using `EasyMock.checkOrder(Object mock, boolean flag)`.

Expectations

In record state, you can record more than just expected method calls. You can wrap expected method calls in `EasyMock.expect(T value)` to set expectations for a method call or use `EasyMock.expectLastCall()` to set expectations on the last expected method call. Both method calls return an `IExpectationSetters` object. Expectation settings are defined in the `IExpectationSetters` interface.

To specify call counts, you can use the following expectations:

`times(int count)`

`times(int min, int max)`

`atLeastOnce()`

`once()`

`anyTimes()`

`once()` is the default one for expected method calls. For example, to allow the last expected method to be called three times, you can use `EasyMock.expectLastCall().times(3)`.

To specify returned values on method calls, you can use `andReturn(T value)`. For example, to expect `provider.exists("A0001")` to return `true`, you can use `expect(provider.exists("A0001")).andReturn(true)`. In some cases, we do not want `EasyMock` to verify the behavior on a method call, but still need it to respond to make a test function properly. Then, you can use stub behavior by using the `andReturn(T value)` method. Using stub behavior can be handy when a mock object has many methods and you are only interested in some of them.

To specify exception to be thrown, you can use `andThrow(Throwable throwable)`. Unchecked exceptions can be thrown from any methods. Checked exceptions only can be thrown from methods that throw them. For example, `expect(provider.exists("A0001")).andThrow(new ProductNotFoundException())`. Similarly, you can use stub behavior by using `andReturn(Throwable throwable)`.

Other than using exact match (raw value) on method arguments, you can use argument matchers to specify expectations for arguments on method calls. The following are some of the argument matchers defined in the `EasyMock` class:

`eq(X value)`: expects a certain value. This is available for all primitive types and objects.

`same(T value)`: expects the same object.

`anyXXX()`: matches any value. This is available for all primitive types and objects. For example, `anyDouble()`, `anyObject()`.

You need to pay special attention while using `anyObject`. For example,

```
Inst1.method1(EasyMock.anyObject(SomeClass.class));
```

```
EasyMock.expectLastCall();
```

Above code snippet expects this method to be called with any instance of SomeClass.

```
SomeClass some = new SomeClass();
```

```
Inst1.method1(some);
```

```
EasyMock.expectLastCall();
```

But, above code snippet expects this method to be called with the exact instance of SomeClass.

`isNull(Class<T> clazz)`: expects null.

`notNull(Class<T> clazz)`: expects not null.

`isA(Class<T> clazz)`: expects an object of a certain type.

`not(X value)`: expects a value that does not match. This is available for all primitive types and objects.

`and(X first, X second)`: expects both values are matched. This is available for all primitive types and objects.

`or(X first, X second)`: expects one of them is matched. This is available for all primitive types and objects.

`lt` (less than), `leq` (less than or equal to), `geq` (greater than or equal to), `gt` (greater than): They are available for numeric primitive types and Comparable types.

String operations: `startsWith(String prefix)`, `contains(String substring)`, `endsWith(String suffix)`.

For example, if you want to replace an exact match by any string in the argument, you can use `expect(provider.exists((String)anyObject())).andReturn(true)`. In this way, any string in the argument will return true.

Argument matchers can combine together through `and()`, `or()`, `not()`. You can define your own argument matchers by implementing the `IArgumentMatcher` interface.

If any one argument in a method is using argument matchers, all arguments in that method need to use argument matchers. If a method has matchers mixed with raw values, you can use `eq()` argument matcher on raw values to prevent an error from happening.

EasyMockSupport

EasyMockSupport is a helper class that keeps track of all mock objects automatically. If there is more than one mock object in a test case, it can extend EasyMockSupport class. You can use `replayAll()`, `verifyAll()` or `resetAll()` on all mock objects instead of using `replay()`, `verify()` or `reset()` on mock objects one by one.

When extending EasyMockSupport, you do not use the static `createMock()` method from the EasyMock class. You use `createMock()` inherited from the EasyMockSupport. It is the same for `createStrictMock()` and `createNiceMock()`.

Class mocking

EasyMock supports both interface mocking and class mocking. To perform class mocking, you need to include Objenesis (<http://objenesis.org>) in the classpath. Final and private methods cannot be mocked. If called, their normal code will be executed. For abstract classes, abstract methods are mocked automatically.

Partial mocking

To create a partial mock for an interface or class, you can use `createMockBuilder` method. A partial mock builder of type `IMockBuilder` is returned. For example,

```
createMockBuilder(SomeClass.class)
.addMockedMethod("method1")
.addMockedMethod("method2")
.createMock();
```

This creates a partial mock builder that mocks two methods: `method1` and `method2`. No overload methods are allowed. For overload methods, you can define method parameters explicitly by using `addMockMethod(String methodName, Class<?>... parameters)`. Private, static, or final methods cannot be mocked.

In EasyMock 3.4, a new version of `createMockBuilder` method, `partialMockBuilder`, can be used instead.

USING EASYMOCK WITH JUNIT

Now, we can modify previous test case by using EasyMock to create mock object for the data provider. In this example, we use interface mocking. EasyMock works on JSE 5.0 and above. Also, you need to add easymock-3.4.jar in the classpath in addition to JARs for JUnit.

```
import static org.easymock.EasyMock.*;
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PurchaseManagerTest1 {
    private DataProvider provider;
    private PurchaseManager purchaseMgr;

    @Before
    public void setUp() {
        provider = createMock(DataProvider.class);
        purchaseMgr = new PurchaseManager(provider);
    }

    @After
    public void tearDown() {
        provider = null;
        purchaseMgr = null;
    }

    @Test(expected=IllegalArgumentException.class)
    public void nullProduct() {
        purchaseMgr.existsProduct(null);
    }
}
```

```
}
```

```
@Test
```

```
public void nullProductException() {  
    try {  
        purchaseMgr.existsProduct(null);  
        fail();  
    } catch(Exception ex) {  
    }  
}
```

```
@Test
```

```
public void existsProduct() {  
    String productNo = "A0001";  
    expect(provider.exists(productNo)).andReturn(true);  
    replay(provider);  
    boolean actual = purchaseMgr.existsProduct(productNo);  
    assertTrue(actual);  
}
```

```
@Test(expected=ProductNotFoundException.class)
```

```
public void notFound() throws ProductNotFoundException {  
    String productNo = "B0001";  
    expect(provider.exists(productNo)).andReturn(false);  
    replay(provider);  
    purchaseMgr.getPrice(productNo);  
}
```

```
@Test
```

```
public void getPrice() throws ProductNotFoundException {  
    String productNo = "A0001";  
    double expected = 150;
```

```
expect(provider.exists(productNo)).andReturn(true);
expect(provider.getPrice(productNo)).andReturn(expected);
replay(provider);
double price = purchaseMgr.getPrice(productNo);
assertEquals(price, expected, 0);
}
```

```
@Test
public void getInventory() throws ProductNotFoundException {
    String productNo = "A0001";
    int expected = 10;
    expect(provider.exists(productNo)).andReturn(true);
    expect(provider.getInventory(productNo)).andReturn(expected);
    replay(provider);
    int inventory = purchaseMgr.getInventory(productNo);
    assertEquals(inventory, expected);
}
```

```
@Test
public void getTotal() throws ProductNotFoundException,
    NotEnoughInventoryException {
    String productNo = "A0001";
    int quantity = 8;
    double price = 150;
    double expected= price*quantity;
    expect(provider.exists(productNo)).andReturn(true);
    expect(provider.getInventory(productNo)).andReturn(10);
    expect(provider.getPrice(productNo)).andReturn(price);
    replay(provider);
    double total = purchaseMgr.getTotal(productNo, quantity);
    assertEquals(total, expected, 0);
    verify(provider);
}
```

```
}
```

```
@Test(expected=NotEnoughInventoryException.class)
public void notEnoughInventory() throws ProductNotFoundException,
NotEnoughInventoryException {
    String productNo = "A0002";
    int quantity = 8;
    expect(provider.exists(productNo)).andReturn(true);
    expect(provider.getInventory(productNo)).andReturn(5);
    replay(provider);
    double total = purchaseMgr.getTotal(productNo, quantity);
    verify(provider);
}
}
```

If there are any failed tests, you will see output such as:

There was 1 failure:

1) notEnoughInventory(PurchaseManagerTest1)

java.lang.Exception: Unexpected exception, expected<NotEnoughInventoryException>
but was<java.lang.AssertionError>

...

Caused by: java.lang.AssertionError:

Unexpected method call DataProvider.exists("A0002"):

 DataProvider.getInventory("A0002"): expected: 1, actual: 0

 at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHand
ler.java:44)

...

USING ANNOTATIONS

You also can use annotations to create mocks and inject them to test objects. This is added since EasyMock 3.2. `@Mock` is identical to `createMock()` method. To specify a mock type, you can use an optional element, `type`, in `@Mock` to create a nice or strict mock. `@TestSubject` is used to inject mocks created with `@Mock` to its fields. Setters are not needed in test classes for mock injections. Because the test runner takes care of initialization now, `setUp()` method is not needed. Previous example can be modified as:

```
@RunWith(EasyMockRunner.class)
public class PurchaseManagerTest2 {
    @Mock
    private DataProvider provider;
    @TestSubject
    private PurchaseManager purchaseMgr = new PurchaseManager();
    ...
}
```

What if tests are using other test runner, then you can use a JUnit rule. This is available since EasyMock 3.3. For example,

```
public class PurchaseManagerTest2 {
    @Rule
    public EasyMockRule mocks = new EasyMockRule(this);
    @Mock
    private DataProvider provider;
    @TestSubject
    private PurchaseManager purchaseMgr = new PurchaseManager();
    ....
}
```

USING EXPECTATIONS

In EasyMock, expectations play an important role in verifying behavior during testing. Here, we will spend more time talking about using expectations in EasyMock. In the following example, The MessageBuilder class is used to build messages. A message is constructed by three parts: a greeting on top, paragraph(s) in the middle and a sign off at the bottom. There are different styles of greeting and sign off.

```
public interface MessageBuilder {  
    public void setGreeting(int style, StringBuilder message);  
    public void addParagraph(String paragraph, StringBuilder message);  
    public void setSignOff(int style, StringBuilder message);  
}
```

```
public class MyMessage {  
    private MessageBuilder builder;  
    public MyMessage(MessageBuilder builder) {  
        this.builder = builder;  
    }  
  
    public String generate(int style, String[] paragraphs) {  
        if(paragraphs == null)  
            throw new IllegalArgumentException("paragraphs is null.");  
  
        StringBuilder message = new StringBuilder();  
        builder.setGreeting(style, message);  
        for(String paragraph : paragraphs)  
            builder.addParagraph(paragraph, message);  
        builder.setSignOff(style, message);  
  
        return message.toString();  
    }  
}
```

In the following test case, there is only one test. In the test, we assume there are two styles

(with values of 1 and 2) available in the MessageBuilder.

```
import static org.easymock.EasyMock.*;
```

```
import org.junit.After;
```

```
import org.junit.Before;
```

```
import org.junit.Test;
```

```
public class MyMessageTest {
```

```
    private MessageBuilder builder;
```

```
    private MyMessage message;
```

```
    @Before
```

```
    public void setUp() {
```

```
        builder = createMock(MessageBuilder.class);
```

```
        message = new MyMessage(builder);
```

```
    }
```

```
    @After
```

```
    public void tearDown() {
```

```
        builder = null;
```

```
        message = null;
```

```
    }
```

```
    @Test
```

```
    public void generate() {
```

```
        String[] paragraphs = new String[]{"p1", "p2"};
```

```
        builder.setGreeting(and(geq(1), leq(2)), isA(StringBuilder.class));
```

```
        builder.addParagraph(isA(String.class), isA(StringBuilder.class));
```

```
        expectLastCall().times(paragraphs.length);
```

```
        builder.setSignOff(and(geq(1), leq(2)), isA(StringBuilder.class));
```

```
        replay(builder);
```

```

    String result = message.generate(1, paragraphs);
    verify(builder);
}
}

```

You might notice that the above test case is missing something. It does test the behavior of `generate()` method. But, it does not verify the result. Since methods in the `MessageBuilder` class are void, certainly we cannot use the `andReturn()` method. What do we do if a method that returns void? The answer is to use expectation setter `andReturn(IAnswer<? extends T> answer)` which allows you to calculate the answer for expected method call or to throw an exception. This allows you to mimic external actions. All we need to do is to implement the `IAnswer` interface. `IAnswer` has only one method:

`T answer() throws Throwable`

Inside this method, you can use `EasyMock.getCurrentArguments()` to get an array of arguments in the expected method call. For void method, null is returned. The returned value (message to be generated) is passing among methods through a mutable class, `StringBuilder`. The following is the revised version of `MyMessageTest`:

```

import static org.easymock.EasyMock.*;
import static org.junit.Assert.*;

import org.easymock.IAnswer;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class MyMessageTest1 {
    private MessageBuilder builder;
    private MyMessage message;

    @Before
    public void setUp() {
        builder = createMock(MessageBuilder.class);
        message = new MyMessage(builder);
    }

    @After

```



```
public void tearDown() {  
    builder = null;  
    message = null;  
}
```

```
@Test
```

```
public void generate() {  
    String[] paragraphs = new String[]{"p1", "p2"};  
    String expected = "greeting1p1p2signoff1";  
    builder.setGreeting(and(geq(1), leq(2)), isA(StringBuilder.class));  
    expectLastCall().andAnswer(new GreetingAnswer<Object>());  
    builder.addParagraph(isA(String.class), isA(StringBuilder.class));  
    expectLastCall().andAnswer(new ParagraphAnswer<Object>  
   ()).times(paragraphs.length);  
    builder.setSignOff(and(geq(1), leq(2)), isA(StringBuilder.class));  
    expectLastCall().andAnswer(new SignOffAnswer<Object>());  
    replay(builder);  
    String actual = message.generate(1, paragraphs);  
    assertEquals(actual, expected);  
    verify(builder);  
}
```

```
private class GreetingAnswer<T> implements IAnswer<T> {  
    public T answer() throws Throwable {  
        Integer style = (Integer)getCurrentArguments()[0];  
        StringBuilder str = (StringBuilder)getCurrentArguments()[1];  
        str.append("greeting" + style);  
  
        return null;  
    }  
}
```

```
private class ParagraphAnswer<T> implements IAnswer<T> {
```

```

public T answer() throws Throwable {
    String para = (String)getCurrentArguments()[0];
    StringBuilder str = (StringBuilder)getCurrentArguments()[1];
    str.append(para);

    return null;
}
}

```

```

private class SignOffAnswer<T> implements IAnswer<T> {
    public T answer() throws Throwable {
        Integer style = (Integer)getCurrentArguments()[0];
        StringBuilder str = (StringBuilder)getCurrentArguments()[1];
        str.append("signoff" + style);

        return null;
    }
}
}

```

In the above example, we use the `andAnswer()` method to get returned values from void methods. Another approach is to use argument matcher `EasyMock.capture(Capture<T> captured)`. In a scenario that an object is updated in a method, you can use `EasyMock.capture` to capture it for later use outside the method. The captured object (represented by a `Capture`) contains updated value. You can use the `getValue()` or `getValues()` method in a `Capture` to get captured value or values. The following is the revised version by using capture:

```

import static org.easymock.EasyMock.*;
import static org.junit.Assert.*;

import org.easymock.Capture;
import org.easymock.IAnswer;
import org.junit.After;
import org.junit.Before;

```

```

import org.junit.Test;

public class MyMessageTest2
{
    private MessageBuilder builder;
    private MyMessage message;

    @Before
    public void setUp() {
        builder = createMock(MessageBuilder.class);
        message = new MyMessage(builder);
    }

    @After
    public void tearDown() {
        builder = null;
        message = null;
    }

    @Test
    public void generate() {
        String[] paragraphs = new String[]{"p1", "p2"};
        String expected = "greeting1p1p2signoff1";
        Capture<StringBuilder> capStr = new Capture<StringBuilder>();
        builder.setGreeting(and(geq(1), leq(2)), capture(capStr));
        expectLastCall().andAnswer(new GreetingAnswer<Object>(capStr));
        builder.addParagraph(isA(String.class), capture(capStr));
        expectLastCall().andAnswer(new ParagraphAnswer<Object>
(capStr)).times(paragraphs.length);
        builder.setSignOff(and(geq(1), leq(2)), capture(capStr));
        expectLastCall().andAnswer(new SignOffAnswer<Object>(capStr));
        replay(builder);
        String actual = message.generate(1, paragraphs);
    }
}

```

```
assertEquals(actual, expected);
verify(builder);
}
```

```
private class GreetingAnswer<T> implements IAnswer<T> {
    private Capture capture;
    public GreetingAnswer(Capture<StringBuilder> capture) {
        this.capture = capture;
    }
    public T answer() throws Throwable {
        Integer style = (Integer)getCurrentArguments()[0];
        StringBuilder str = (StringBuilder)capture.getValue();
        str.append("greeting" + style);

        return null;
    }
}
```

```
private class ParagraphAnswer<T> implements IAnswer<T> {
    private Capture capture;
    public ParagraphAnswer(Capture<StringBuilder> capture) {
        this.capture = capture;
    }
    public T answer() throws Throwable {
        String para = (String)getCurrentArguments()[0];
        StringBuilder str = (StringBuilder)capture.getValue();
        str.append(para);

        return null;
    }
}
```

```
private class SignOffAnswer<T> implements IAnswer<T> {
private Capture capture;
public SignOffAnswer(Capture<StringBuilder> capture) {
this.capture = capture;
}
public T answer() throws Throwable {
Integer style = (Integer)getCurrentArguments()[0];
StringBuilder str = (StringBuilder)capture.getValue();
str.append("signoff" + style);

return null;
}
}
}
```

POWERMOCK

As mentioned in the beginning, EasyMock cannot mock private methods. PowerMock is a mock framework that extends EasyMock with features such as mocking on private methods. In the following example, a class representing a job submitter that submits a job to either queue 1 or queue 2 driven by a count returned from the getCount() method.

```
Public class JobSubmitter {  
  
    public void submit() {  
        if(getCount() <= 10000) {  
            submitToQueue1();  
        } else {  
            submitToQueue2();  
        }  
    }  
  
    private int getCount() {  
        // assume this method has complex operations  
        return 0;  
    }  
  
    private void submitToQueue1() {  
        System.out.println("Submitting to queue 1...");  
    }  
  
    private void submitToQueue2() {  
        System.out.println("Submitting to queue 2...");  
    }  
}
```

Because PowerMock extends EasyMock, basically it's just like using EasyMock. As you can see, PowerMock also simplifies partial mocking. You cannot use an expect() method because a private method cannot be accessed outside a class. Instead, you need to use an expectPrivate() method.

```
import static org.powermock.api.easymock.PowerMock.*;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;
```

```
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
@RunWith(PowerMockRunner.class)
```

```
@PrepareForTest(JobSubmitter.class)
```

```
public class JobSubmitterTest1 {
```

```
    private JobSubmitter submitterMock;
```

```
    @Before
```

```
    public void setUp() {
```

```
        submitterMock = createPartialMock(JobSubmitter.class,
        "getCount");
```

```
    }
```

```
    @Test
```

```
    public void testToQueue1() throws Exception {
```

```
        expectPrivate(submitterMock, "getCount").andReturn(9000);
```

```
        replay(submitterMock);
```

```
        submitterMock.submit();
```

```
    }
```

```
    @Test
```

```
    public void testToQueue2() throws Exception {
```

```
        expectPrivate(submitterMock, "getCount").andReturn(11000);
```

```
        replay(submitterMock);
```

```
        submitterMock.submit();
```

```
}
```

```
}
```

To mock a class with static methods, you can either use

```
mockStatic(Class<?> type)
```

to mock the whole class, or use a partial mocking method such as

```
mockStaticPartial(Class<?> type, String... methodNames)
```

to mock some methods.

Instead of using `@RunWith(PowerMockRunner.class)`, you can bootstrap PowerMock with a JUnit rule, `PowerMockRule`. This is included in `powermock-module-junit4-rule`.

For example,

```
@PrepareForTest(JobSubmitter.class)
```

```
public class JobSubmitterTest1 {
```

```
    @Rule
```

```
    public PowerMockRule mocks = new PowerMockRule();
```

```
    ...
```

```
}
```


TEST COVERAGE ANALYSIS

Cobertura is a test coverage tool for Java programs. It can generate report to help you discover where your software is being tested. You can download Cobertura from <http://cobertura.sourceforge.net>. Cobertura can be used either from command line or from Apache Ant tasks. The following are steps to generate Cobertura coverage report through command line:

Step 1

Cobertura adds instrumentation instructions into the bytecode directly. First, you create instrumented version of classes by adding instrumentation instructions into your classes and put them in different directory. You can do that through a command line utility, `cobertura-instrument`, in the Cobertura directory:

```
cobertura-instrument [—datafile datafile] [—destination directory] classes [...]
```

By default, the coverage data file `cobertura.ser` is located at the current directory. Classes can be specified individually or as a directory. For example, to instrument classes under the `classes` directory and store them in the `instrumented` directory, you can run:

```
cobertura-instrument —destination instrumented classes
```

A coverage data file storing metadata of classes, `cobertura.ser`, will be created.

Step 2

Before you can run your tests, you need to make a few changes. You need to:

Add Cobertura JAR (e.g., `cobertura-2.1.1.jar`) and lib directory to the classpath.

In the classpath, add the directory containing instrumented classes before the directory containing original classes.

Specify location of the data file by using the system property `net.sourceforge.cobertura.datafile`. For example, -
`Dnet.sourceforge.cobertura.datafile=cobertura.ser`.

Step 3

Once you have run your tests, you can start generating report. To generate Cobertura coverage report, you can use `cobertura-report`:

```
cobertura-report [—format (html|xml)] [—datafile datafile] [—destination directory]  
source code directory [...]
```

For example,

```
cobertura-report —format html —datafile cobertura.ser —destination reports ..
```

The following is the coverage report for the `PurchaseManager` class and associated classes based on the `PurchaseManagerTest1` test case:

Coverage Report - All Packages

Package /	# Classes	Line Coverage		Branch Coverage		Complexity
(default)	4	56%	22/39	75%	9/12	1.706
All Packages	4	56%	22/39	75%	9/12	1.706
Classes in this Package /		Line Coverage		Branch Coverage		Complexity
DataProvider		N/A	N/A	N/A	N/A	1
NotEnoughInventoryException		25%	2/8	N/A	N/A	1
ProductNotFoundException		25%	2/8	N/A	N/A	1
PurchaseManager		78%	18/23	75%	9/12	3

You can further dig into a class for detailed coverage report:

Coverage Report - PurchaseManager

Classes in this File	Line Coverage	Branch Coverage	Complexity
PurchaseManager	78% 18/23	75% 9/12	3

```

1  public class PurchaseManager {
2      private DataProvider dataProvider;
3
4  0  public PurchaseManager() {
5  0  }
6
7  8  public PurchaseManager(DataProvider dataProvider) {
8  8      this.dataProvider = dataProvider;
9  8  }
10
11  public double getPrice(String productNo) throws ProductNotFoundException {
12  2  if(!existsProduct(productNo))
13  1      throw new ProductNotFoundException("Cannot find product # " + productNo);
14  1  return dataProvider.getPrice(productNo);
15  }
16
17  public int getInventory(String productNo) throws ProductNotFoundException {
18  1  if(!existsProduct(productNo))
19  0      throw new ProductNotFoundException("Cannot find product # " + productNo);
20  1  return dataProvider.getInventory(productNo);
21  }
22
23  public double getTotal(String productNo, int quantity)
24  throws ProductNotFoundException, NotEnoughInventoryException {
25  2  if(quantity < 1)
26  0      throw new IllegalArgumentException("quantity is less than one.");
27  2  if(!existsProduct(productNo))
28  0      throw new ProductNotFoundException("Cannot find product # " + productNo);
29
30  2  int inventory = dataProvider.getInventory(productNo);
31  2  if(inventory < quantity)
32  1      throw new NotEnoughInventoryException();
33  1  double price = dataProvider.getPrice(productNo);
34  1  return price*quantity;
35  }
36
37  public boolean existsProduct(String productNo) {
38  8  if(productNo == null)
39  2      throw new IllegalArgumentException("productNo is null.");
40  6  return dataProvider.exists(productNo);
41  }
42  }

```


JMeter

Apache JMeter is a Java-based desktop application, which can be used for load testing to measure the performance of a system or used for stress testing to see if a system is crashed gracefully. It can generate reports to help eliminate bottlenecks of the system or to see how it performs under heavy loads. Other than load testing, you also can use JMeter in functional testing. JMeter provides a variety of test elements. They are quite handy and can save you time in writing your own Java programs for testing. Using JMeter is quite intuitive because it provides a nice GUI to create and run tests. You also can run JMeter tests in non-GUI mode. Tests can be run either locally or remotely. JMeter is designed using plugin approach to provide flexibility in adding new features (test elements) as custom plugins.

You can download JMeter from <http://jmeter.apache.org>. The latest version is 2.9 at the time of writing.

BUILDING A JMETER TEST PLAN

To use JMeter for testing, you can either load an existing test plan or create a new one. Basically, a test plan has a tree structure, which contains test elements describing how a test will run.

Starting JMeter

First, we need to start JMeter (JSE 6.0 or above is required). You can start JMeter from the command line by running `jmeter.bat` (for Windows) or `jmeter.sh` (for Unix/Linux) in the `bin` directory.

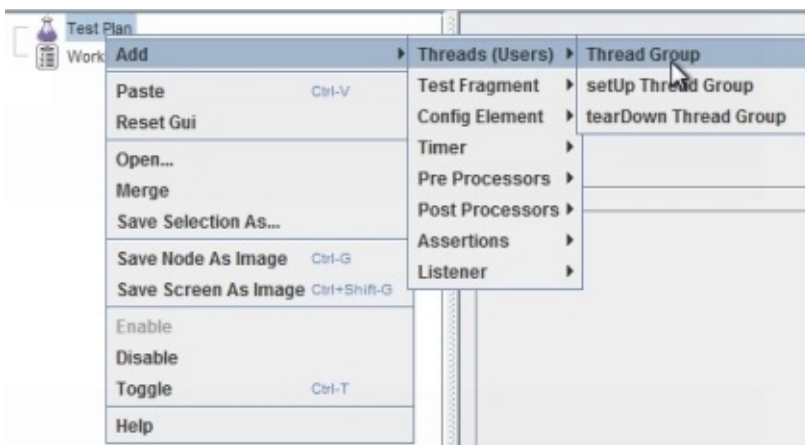
To run JMeter in non-GUI mode, you can use the following command line options:

- n non-GUI mode
- t the test plan file (with extension name `jmx`)
- l the result log file (with extension name `jtl`)
- j the log file

Adding test elements

There are two panels on the screen of JMeter GUI. The left panel is the test tree for a test plan and the right panel is the configuration panel for the selected element. A new test plan starts with two elements: Test Plan and WorkBench. The Test Plan is the root of a test tree. The WorkBench is a temporary place for storing test elements not in use or non-test elements. The WorkBench is not saved with the test plan automatically. You can save it separately.

To add a new element to the test tree, you can either right-click on an element and choose Add from the popup menu or choose Edit -> Add from the menu bar on top:



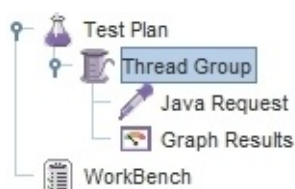
Also, you can remove or disable/enable elements from the same menu.

In this example, we choose to add a Thread Group to the test plan. A test plan contains at least one Thread Group. A test case is defined in a Thread Group. The Thread Group allows you to define the number of threads running for a test case, the number of times it is going to run and optionally a start and stop times through a scheduler.

Next, we add a Java Request sampler under this Thread Group by choosing Add -> Sampler -> Java Request. The Java Request sampler allows you to put your own Java classes (or custom samplers) as long as they implement the `JavaSamplerClient` interface or extend the `AbstractJavaSamplerClient` class. There are two Java sample classes available in the JMeter. We are using the `JavaTest` class for demonstration purpose. The `JavaTest` does not access network and is useful for testing the performance of local machine.

To show testing results, we add a Graph Results listener. The Graph Results listener plots testing results for all samples. It is good for showing the progress of testing visually. But, it does consume more resources. Usually, this is used in testing or debugging a test plan with small load.

The following is our first JMeter test plan:



Note: When you specify number of thread groups/threads running in a test plan, you need to be careful about the capability of the machine running JMeter. The more load you put on the machine, the less accurate timing information you might get. Using non-GUI mode with results written to files can help. You even can consider running your test in multiple machines (remote testing) for a test with heavy load.

Running a test plan

Once you are done on defining a test plan, you can run it. To run a test plan, you can do that through the Run menu or the toolbar. When a test plan is running, you can check the current status from the upper-right corner:



The green square icon indicates a test plan is running. The yellow triangle icon with a number is the error/fatal counter. By default, JMeter log file `jmeter.log` is located under the `bin` directory (or using `-j` option to specify it). To see logging screen from the GUI, you can toggle it on and off by clicking on the yellow triangle icon or choose the Options menu -> Log Viewer. `5/5` indicates that total number of threads is five and five threads are running.

To clear results in the whole test plan, you can choose Clear All from the Run menu. Or, you can choose Clear to clear results in a selected element.

You can stop or shutdown a running test through the Run menu or the toolbar. The difference between stop and shutdown is that threads stop immediately for stop and threads stop gracefully at the end of current work for shutdown. For the non-GUI mode, you can use `shutdown.cmd` (or `shutdown.sh`) or `stoptest.cmd` (or `stoptest.sh`).

Saving a test plan

To save a test plan, you can do that through the File menu or the toolbar. A test plan is saved with extension name jmx. You can save partial test plan by using Save Selection As from the File menu or the popup menu.

Debugging a test plan

Enabling debug logging can be helpful in creating a test plan. Most test element support debug logging. To enable debug logging on a test element, select a test element and choose the Help menu -> Enable debug. You can disable it by choosing Disable debug. To view debug logging, open jmeter.log under the bin directory or enable Log Viewer before you run it.

Remote testing

You can use JMeter to do remote (or distributed) testing. Using remote testing can save you trouble in copying the test plan to each machine and also allow you to simulate larger load with more users. In remote testing, one JMeter client is used as the master to control other JMeter instances (serving as slaves) running as remote servers. Data will be collected by the client. You can start a JMeter server by running `jmeter-server.bat` (for Windows) or `jmeter-server` (for Unix/Linux). All JMeter instances should use the same version of JMeter. In the JMeter client, you add remote servers into the property `remote_hosts` in the `jmeter.properties`. Then, you can start the JMeter client and load the test plan. To start remote servers, select the Run menu -> Remote Start or Remote Start All. If you are using a non-GUI client, you can run:

```
jmeter -n -t {test script} -R server1, server2, ...
```

TEST ELEMENTS

A JMeter test plan is constructed by test elements as a tree structure (there are few non-test elements you can add in the WorkBench). JMeter provides a rich set of test elements. Those are what make JMeter such a powerful tool. Most of test elements are processed in the order they appear in the test tree. Basically, you use samplers to test against a server and use logic controllers to control the flow of a test run, and use listeners to gather or view testing results from the server.

The following are test elements categorized by types:

Samplers

There are two types of controllers. Samplers are one of them. Samplers generate requests (or samples) and perform actual testing jobs in JMeter. Mostly, samplers are used to send requests to a server and wait for responses.

JMeter has samplers that support the following protocols: HTTP (the HTTP Request sampler), FTP (the FTP Request sampler), TCP (the TCP sampler), SMTP (the SMTP sampler), POP3 (the Mail Reader sampler) and IMAP (the Mail Reader sampler). You even can execute commands on the local machine (the OS Process sampler) or send an SQL query to a database (the JDBC Request sampler). To use your own Java classes, you can use the Java Request sampler or JUnit Request sampler (for JUnit test classes). To use a scripting language, you can use the JSR223 sampler (scripting for the Java platform) or BSF sampler (Apache Bean Scripting Framework).

There is a special sampler called Test Action sampler. It does not send any request to the server. Instead, it can pause or stop a target thread or all threads. The Test Action sampler is used with a conditional controller.

Logic controllers

Logic controllers are controllers too. Logical controllers can break up the flow of execution and allow you to control the order that samplers are processed.

For looping, you can use the Loop Controller, While Controller or ForEach Controller. For decision making, you can use If Controller. For executing a group of elements, you can use the Simple Controller or Random Order Controller. The difference is that the Simple Controller executes elements inside it in order and the Random Order Controller executes them in random order. To execute one of the elements per iteration, you can use the Interleave Controller (alternating among elements for each iteration) or Random Controller (picking one in random for each iteration). You also can use the Once Only Controller to tell JMeter to process elements inside it only once per thread.

To structure your tests better, you can break them into fragments (or modules). To include a Test Fragment (created through the Test Fragment element), you can use the Include Controller to include it through an external jmx file. Or, you can use the Module Controller to include test fragments located in any Thread Group or in the WorkBench. You can include any fragment under a logic controller even it is disabled.

Both samplers and logic controllers are processed in the order they are defined in the test tree.

Listeners

Listeners gather information from tests. They provide means to allow you to view results from GUI mode and optionally to save results to a file in XML or CSV format. You can configure items to be saved in a file by clicking on the Configure button in the listener.

To view detailed results, you can use the Graph Results, View Results in Table or View Results Tree listener. To view aggregate data, you can use the Aggregate Report, Aggregate Graph or Summary Report. If you only simply want to save results to a file without viewing it from GUI, you can use the Simple Data Writer. Similarly, if you want to save responses to a file, you can use the Save Responses to a file listener.

Listeners can be added anywhere in the test tree. But, they only gather data from elements inside the same tree branch (at or below the same level). To avoid confusion, a good practice is to put listeners at the end of the tree branch they are used for gathering data.

Configuration elements

Configuration elements are used to set up configurations used by samplers. Configuration elements are only accessible by elements inside the same tree branch (including descendant branches). They are processed at the start of the tree branch in which they are defined no matter where they are located. One exception is the User Defined Variables. It is processed at the start of a test run no matter where it is located.

For the HTTP Request sampler, you can use the HTTP Authorization Manager, HTTP Cache Manager, HTTP Cookie Manager, HTTP Request Defaults or HTTP Header Manager to work with it. For the JDBC Request sampler, you can use the JDBC Connection Configuration. If you need to use a counter, you can use the Counter configuration element.

To define user defined variables, you can use the User Defined Variables (or define them in the Test Plan). User defined variables can be referenced as `${variable_name}`. Variables are local to each thread. To share values between threads or thread groups, you can use JMeter properties since they are global. Properties are defined in the `jmeter.properties`. Or, you can use additional files to define properties: `user.properties` and `system.properties`. Properties can be referenced as `${__P(property_name)}` or `${__P(property_name,default_value)}`. Both variables and properties are case-sensitive. You can debug variables and properties by using the Debug Sampler. You can see the values from the Response data tab in the View Results Tree.

Assertions

Assertions work with samplers to check the responses from the server. The scope of an assertion is only at the current branch in which it is defined. It cannot be accessed by descendant branches unless it is specified. Assertions are processed after every sampler in the same scope. If you want an assertion only applies to a sampler, you can make it as a child of the sampler.

You can use the Response Assertion to check responses. The pattern matching is Perl5 style regular expressions. To assert that each response was received within certain amount of time, you can use the Duration Assertion. To assert the size of each response, you can use the Size Assertion. To view assertion results, you can use the Assertion Results listener.

Timers

Timers work with samplers to cause a delay between each request. Timers are processed before every sampler in the same scope.

The Constant Timer is used to pause the same amount of time between each request. If you need a random amount of time, you can use the Gaussian Random Timer, Uniform Random Timer or Poisson Random Timer.

Pre-processors

Pre-processors are executed before samplers at the same level. You can add a pre-processor as the child of a sampler to make it pre-process information before feeding it into the sampler. Usually, they are used to modify settings before samplers run. Pre-processors are executed right before timers and after configuration elements.

To process HTML responses, you can use the HTML Link Parser or HTML URL Rewriting Modifier.

Post-processors

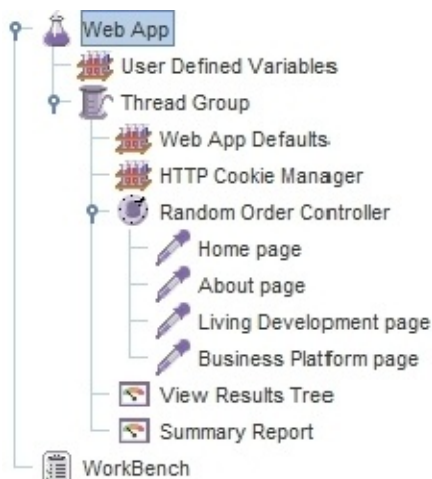
Post-processors are executed right after samplers at the same level. Usually, they are used to process responses. Post-processors are executed before assertions and listeners.

To extract values from a response and assign the result to a variable for later use, you can use the Regular Expression Extractor.

TESTING A WEB APPLICATION

Now, we will learn how to create a JMeter test plan to test a web application. To make HTTP (or HTTPS) requests, you can use the HTTP Request sampler. The HTTP Request sampler is not a browser. But, it can do most of jobs a browser can do except something like rendering pages or running scripts. Optionally, it even allows you to retrieve embedded resources from web pages. For common settings (e.g., server name or IP) shared among the HTTP Request samplers, you can define them in the HTTP Request Defaults. Other samplers will inherit values defined in the HTTP Request Defaults. For example, you can use “Server Name or IP” field in the HTTP Request Defaults to specify the server to be tested. Then, you can leave that field blank in other HTTP Request samplers. The scope of the HTTP Request Defaults is within the current branch (and its descendants). Normally, a web application needs cookie support. We can use the HTTP Cookie Manager just in case cookie support might be needed by the application.

In the following example, four pages will be visited by each thread in random order. We use the Random Order Controller to simulate this behavior. The JMeter test plan is shown below:



After finished running this test plan, you can check results from two listeners defined in it. The following screenshot is from the View Results Tree:



On the results tree, it contains a list of pages being visited. For a failed request, the color is red if there is any. You can select a page to view the request and the response. If the response data is in HTML, you can view rendered page by choosing HTML from the drop-down menu at the bottom of the tree. The rendered page only gives you some idea about a page. It is not going to be good comparing with a browser. JMeter also can handle different types of documents (e.g., PDF, MS Office). You can view text in the document by choosing Document. Since the View Results Tree records requests and responses from all visited pages, it is good for checking the behavior of a web application or to debug a web application. Also, you can use it to test your JMeter test plan. But, it is not a good idea to use it during a load testing since it is going to consume lots of resources.

Another listener in this test plan is the Summary Report. The Summary Report gives you a table view of results. It gives you a summary report arranged by pages (as table rows). Average, Min and Max are elapsed times in milliseconds.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Living Development page	50	292	141	3286	443.18	0.00%	4.7/sec	62.66	13530.0
About page	50	206	152	435	75.12	0.00%	4.6/sec	62.72	14054.0
Business Platform page	50	213	154	443	78.61	0.00%	4.6/sec	63.13	13938.0
Home page	50	128	106	302	46.99	0.00%	5.1/sec	68.07	13674.0
TOTAL	200	210	106	3286	236.58	0.00%	18.2/sec	245.73	13799.0

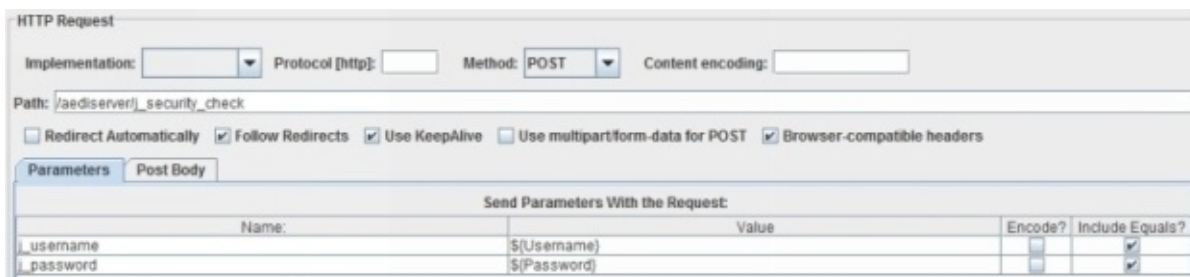
In the previous example, there are four pages that will be visited (in random order) during the test. Now, we want to include more pages in the test. Adding pages manually is kind of painful. We can use the HTML Link Parser to do that for us. The HTML Link Parser is a pre-processor that can extract links from previous HTML response. Being followed by an HTTP Request sampler and included in a loop can make the whole thing behave like a web crawler. We will add this process into previous example. Each thread will randomly select one of the four pages in previous example as the starting page and crawl through the web site. In the HTML Request sampler, the Path field uses the regular expression (?i).*\.html to match any HTML page. (?i) indicates that it is case-insensitive. For detailed information about using regular expressions in JMeter, you can check http://jmeter.apache.org/usermanual/regular_expressions.html.

To continue crawling through the web site, we add the HTTP Request sampler inside the

TESTING WITH AUTHENTICATION

Some web applications require user authentication to access certain pages. Form-based authentication is a popular method to secure resources for web applications. To do testing on web sites that require user authentication, you need to add additional steps to sign in the web site automatically in the test plan before it can do anything.

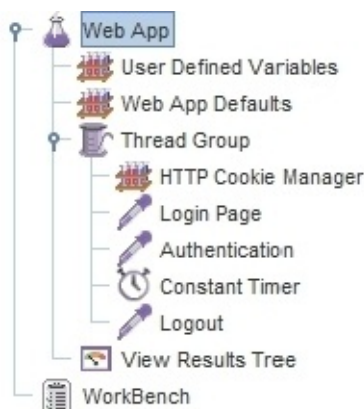
First, you need to visit a page to get cookies from the web site. Usually, it is a session ID for session tracking. The next step is to go through security checking by providing username and password. To find the link for form action and query parameters required to submit to the server for authentication, you can use “View source” function at the login page from a browser. The following is a sample HTTP Request sampler for authentication:



It is using HTTP POST method. The form action is j_security_check. The path is /aediserver/j_security_check. The context root for the web application aediserver is included. The form field for username is j_username and the form field for password is j_password. Both username and password are defined in the User Defined Variables.

Similarly, you can find form action to sign out the web site from “View source”. This depends on how it is implemented.

The following is a simple example showing how to add authentication in a test plan. It includes the following steps: visiting login page to initialize connection to the server, sign in and then sign out.



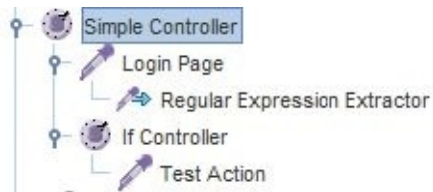
In this example, we only store one set of user account information for authentication. If you want to simulate the scenario that different users sign in at the same time, you can use the User Parameters pre-processor to store multiple user accounts. For example, to have one user account per thread for four threads in a test, you can define four sets of parameters for username and password:

Parameters				
Name:	User_1	User_2	User_3	User_4
Username	test1	test2	test3	test4
Password	test10	test20	test30	test40

If the number of threads is more than the number of sets, they will be reused.

USING REGULAR EXPRESSIONS

What if you need to stop testing in case of connection problem such as connection refused by the server? In such case, there is no reason to continue testing. The following is a fragment that replaces Login Page node in the previous example:



We add the Regular Expression Extractor post-processor to parse response from the server while connecting to the login page. If there is any connection problem, an exception will be thrown in the response code. You can check Sample result tab in the View Results Tree to see what will return when it failed. Inside the Regular Expression Extractor, field values are set as follows:

Reference Name: Exception

Regular Expression: `.[a-zA-Z0-9]*Exception)`

Template: `1`

Match No.: 1

Default Value: `NO_EXCEPTION`

We use a regular expression to search for any text related to exception (e.g., `HttpException`) in the response. Parentheses (and) indicates a group in the regular expression. It is used to get a portion of matched string. In this case, it is used to remove a dot before the exception class name. `1` in the Template field indicates group 1 and 1 in the Match No. field indicates the first match. If there is no match, default value “`NO_EXCEPTION`” is returned. The value is assign to a variable named `Exception`.

In the If Controller, the Condition field is set as

`"${Exception}" != "NO_EXCEPTION"`

The variable needs to be double-quoted. In the Test Action, the action is “Stop Now”.

One tricky part about using regular expressions against responses is how to verify the results. One easy way to validate regular expressions is through the View Results Tree. You can find a Search field at the bottom of the Response data tab. Type in a regular expression and make Regular exp. checkbox checked. Click on the Find button. You can see matched string is highlighted if there is any. Click on the Find next button for next matched string.

Sampler result	Request	Response data
<pre>org.apache.http.conn.HttpHostConnectException: Connection to http://192.168.1.12:8080 refused at org.apache.http.impl.conn.DefaultClientConnectionOperator.openConnection(DefaultClientConnectionOperator.java:190) at org.apache.http.impl.conn.ManagedClientConnectionImpl.open(ManagedClientConnectionImpl.java:294) at org.apache.http.impl.client.DefaultRequestDirector.tryConnect(DefaultRequestDirector.java:645) at org.apache.http.impl.client.DefaultRequestDirector.execute(DefaultRequestDirector.java:480) at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:906) at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:805) at org.apache.jmeter.protocol.http.sampler.HTTPHC4Impl.sample(HTTPHC4Impl.java:286) at org.apache.jmeter.protocol.http.sampler.HTTPSamplerProxy.sample(HTTPSamplerProxy.java:62) at org.apache.jmeter.protocol.http.sampler.HTTPSamplerBase.sample(HTTPSamplerBase.java:1088) at org.apache.jmeter.protocol.http.sampler.HTTPSamplerBase.sample(HTTPSamplerBase.java:1077) at org.apache.jmeter.threads.JMeterThread.process_sampler(JMeterThread.java:428) at org.apache.jmeter.threads.JMeterThread.run(JMeterThread.java:256) at java.lang.Thread.run(Unknown Source) Caused by: java.net.ConnectException: Connection refused: connect at java.net.PlainSocketImpl.socketConnect(Native Method) at java.net.PlainSocketImpl.doConnect(Unknown Source) at java.net.PlainSocketImpl.connectToAddress(Unknown Source) at java.net.PlainSocketImpl.connect(Unknown Source) at java.net.SocksSocketImpl.connect(Unknown Source) at java.net.Socket.connect(Unknown Source) at org.apache.http.conn.scheme.PlainSocketFactory.connectSocket(PlainSocketFactory.java:127) at org.apache.http.impl.conn.DefaultClientConnectionOperator.openConnection(DefaultClientConnectionOperator.java:180) ... 12 more</pre>		
Search: <input type="text" value="\{[a-zA-Z0-9]*Exception)"/> <input type="button" value="Find next"/> <input type="checkbox"/> Case sensitive <input checked="" type="checkbox"/> Regular exp.		

USING HTTP PROXY SERVER

In the case that there are too many form fields. To put them in manually is time consuming. Or, you are not able to see form field names from view source in a browser. You can use the Recording Controller to record test samples. You are doing recording through the HTTP Proxy Server. All recorded samples will be saved under the Recording Controller. You can use them in a test plan. The following are basic steps to do recording using the HTTP Proxy Server:

Step 1

First, you add a Thread Group.

Step 2

Next, you add the Recording Controller. The Recording Controller is a logic controller. Optionally, you can add an HTTP Request Defaults under the Recording Controller. That will leave those fields in the recorded elements blank if they are specified in the HTTP Request Defaults.

Step 3

Now, we can add the HTTP Proxy Server under the WorkBench. The HTTP Proxy server is a non-test element. You specify a port number for the proxy server (e.g., 8088). If you do not want URLs embedded in the page being recorded, you can set Grouping as “Store 1st sampler of each group only”. To include or exclude certain resources, you can use “URL Patterns to Include” or “URL Patterns to Exclude”. All will be recorded if nothing is specified. Usually, you can exclude image files or simply click on the “Add suggested Excludes” button to use suggested patterns. You can add the View Results Tree listener if you want to see the responses.

Step 4

We need to change the browser you will be using to set up proxy server information. For Internet Explorer 9, choose the Tools menu -> Internet Options -> Connections. Click on LAN settings at the bottom to bring up a dialog. In the dialog, make the checkbox for Proxy server checked and type in the port number of the proxy server. Make sure local addresses are not bypassed. Click on the OK button to update it. For Firefox, choose the Tools menu -> Options -> Network -> Settings -> Manual proxy configurations.

One way to verify your settings is to connect to a web site. You should not be able to connect to it unless the proxy server is started.

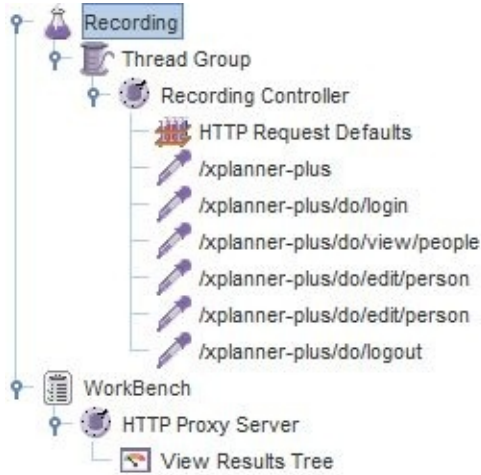
Step 5

You switch back to JMeter. Click on the Start button at the bottom of the HTTP Proxy Server to start it.

Step 6

Now, you can open the link to the server from the browser and browse around as usual. Every step you make is recorded. Stop the proxy server when you are done recording. Also, remember to change back your browser settings.

The following is a test plan with recorded elements by using Grouping as “Store 1st sampler of each group only” and also “Capture HTTP Headers” checkbox is unchecked. The elements below the HTTP Request Defaults are recorded elements.



Before you can run it, at least you need to add an HTTP Cookie Manager since cookie support is needed. Of course, a better way is to create a new test plan and copy elements from the recording test plan. You are allowed to do copy and paste between JMeter instances.

The following is an HTTP Request that contains query parameters for updating a form. You can see query parameters for the form fields from the Parameters tab:

Send Parameters With the Request:			
Name:	Value	Encode?	Include Equals?
id	0	<input type="checkbox"/>	<input checked="" type="checkbox"/>
returnto	/do/view/people	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
name		<input type="checkbox"/>	<input checked="" type="checkbox"/>
userIdentfier		<input type="checkbox"/>	<input checked="" type="checkbox"/>
initials		<input type="checkbox"/>	<input checked="" type="checkbox"/>
email		<input type="checkbox"/>	<input checked="" type="checkbox"/>
phone		<input type="checkbox"/>	<input checked="" type="checkbox"/>
hidden	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>
newPassword		<input type="checkbox"/>	<input checked="" type="checkbox"/>
newPasswordConfirm		<input type="checkbox"/>	<input checked="" type="checkbox"/>
projectId[0]	1684	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
projectRole[0]	none	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
projectId[1]	2289	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
projectRole[1]	none	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
projectId[2]	1409	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

TESTING WEB APPLICATIONS USING GWT

A challenge to test web applications using Ajax technologies is that there is no need to request new HTML pages from the server. The interaction between the client (Ajax code running in the browser) and the server is through a remote procedure call (RPC). Certainly, view source from the browser is not going to help. Using the Recording Controller certainly is very helpful for web applications using technologies related to Ajax. The following example is the recording of a web application using GWT:

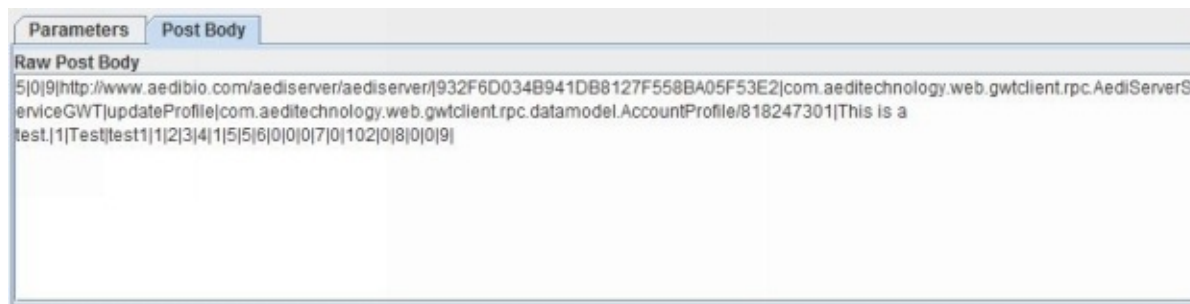


To run it, you need to add an HTTP Cookie Manager. For each GWT RPC service call, which is AediServerService in this case, you need to add an HTTP Header Manager as a child with the header:

```
Content-Type: text/x-gwt-rpc; charset=utf-8
```

Or, you can make “Capture HTTP Headers” checkbox in the HTTP Proxy Server checked. Then, an HTTP Header Manager will be added to every sampler automatically.

Comparing with the previous example, you are not able to see query parameters for the form fields from the Parameters tab. Instead, you can see the GWT RPC request (a serialized stream in plain text) from the Post Body tab. The following is the GWT RPC HTTP Request from one of the service calls:



As shown in the post body, the GWT RPC request is a sequence of fields using the vertical bar as the delimiter. For detailed information about the GWT RPC wire protocol, you can check <https://docs.google.com/document/d/1eG0YocsYYbNAtivkLtcaiEE5IOF5u4LUol8-LL0TIKU/edit?pli=1>. Here, we are only interested in some of the fields. The third field,

which is 9, indicates there are nine strings in the string table as shown below:

http://www.aedibio.com/aediserver/aediserver/

932F6D034B941DB8127F558BA05F53E2

com.aeditechnology.web.gwtclient.rpc.AediServerServiceGWT

updateProfile

com.aeditechnology.web.gwtclient.rpc.datamodel.AccountProfile/818247301

This is a test.

1

Test

test1

The third field is the interface name of the GWT RPC service. The fourth field is the name of the method to be called. The fifth field is the type of the first parameter in the method. The remaining fields are values of AccountProfile. All the remaining numeric fields are used to reconstruct the method call and parameters based on fields in the string table.

ADDING JAVA CLASSES

JMeter allows you to add Java classes in the test plan through the following samplers:

JUnit Request

The JUnit Request sampler allows you to include JUnit test classes to a test plan. To include JUnit test classes, there are two options:

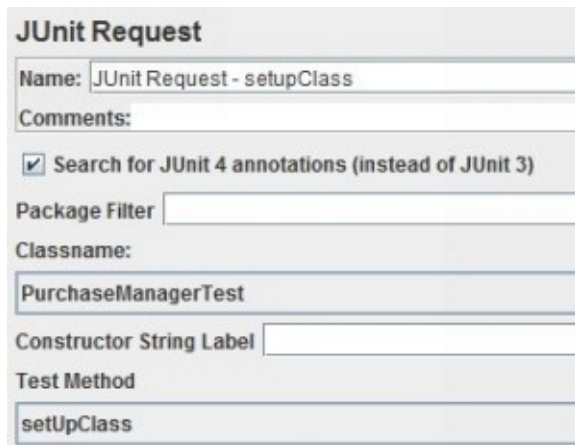
Option1

Pack classes to a JAR file and place it in the lib/junit directory

Option 2

Add the directory of JUnit classes to the user.classpath property in the user.properties under the bin directory

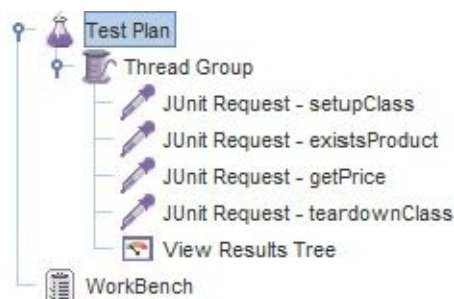
Now, in the JUnit Request sampler, you should be able to see those test classes from the Classname drop-down. If JUnit test classes are using JUnit 4 annotations, you need to make “Search for JUnit 4 annotations” checked in order to see them. You can select a test method from the Test Method dropdown. One JUnit Request sampler is for one test method.



The screenshot shows the 'JUnit Request' configuration dialog box. It has the following fields and options:

- Name:** JUnit Request - setupClass
- Comments:** (empty text area)
- Search for JUnit 4 annotations (instead of JUnit 3)
- Package Filter:** (empty text field)
- Classname:** PurchaseManagerTest
- Constructor String Label:** (empty text field)
- Test Method:** setUpClass

In the following example, it contains test methods of the PurchaseManagerTest from the previous chapter. You need to include methods annotated with @BeforeClass (the setupClass method) and @AfterClass (the tearDownClass method) in the test plan. They will not be called automatically.



Java Request

Comparing with the JUnit Request sampler, the Java Request sampler has less restriction since it allows you to add any Java classes only if they implement the `JavaSamplerClient` interface or extend the `AbstractJavaSamplerClient` class. Developers are encouraged to extend the `AbstractJavaSamplerClient` class. There are five methods in the `AbstractJavaSamplerClient` class:

`Logger getLogger()`: gets a `Logger` which can be used to log messages.

`Arguments getDefaultParameters()`: defines a list of parameters provided by this test and also what will be displayed in the GUI. Parameters are represented by the `Arguments` class. Parameters are added through the `addArgument(Argument arg)` or `addArgument(String name, String value)` method.

`void setupTest(JavaSamplerContext context)`: does initialization for this test. It is only called once per thread.

`void teardownTest(JavaSamplerContext context)`: does cleanup for this test. It is only called once per thread.

`SampleResult runTest(JavaSamplerContext context)`: defines actual operations in a test. This is an abstract method that you need to override. This method is called per iteration.

The `JavaSamplerContext` class is used to get values of parameters. You can get the value of a parameter through the `getParameter(String name)` or `getParameter(String name, String defaultValue)` method for a `String`. For an integer, you can use the `getIntParameter(String name)` or `getIntParameter(String name, int defaultValue)` method. For a long, you can use the `getLongParameter(String name)` or `getLongParameter(String name, long defaultValue)` method.

The `SampleResult` class is used to contain returned results. To calculate elapsed time, you can use the `sampleStart()` method to start recording and the `sampleEnd()` method to end recording. To set this test as successful, you can use the `setSuccessful(boolean success)` method.

For detailed information, you can check the JMeter API documentation in the `docs/api` directory.

To compile it, you need to include `ApacheJMeter_core.jar` and `ApacheJMeter_java.jar` under the `lib/ext` directory in the classpath. To deploy it, you can pack it in a JAR file and put it under the `lib/ext` directory.

JAVACC

JavaCC (Java Compiler Compiler) is an open source lexical analyzer generator and a parser generator for use with the Java applications, which takes a grammar specification (e.g., EBNF) and generates the Java source code of a lexical analyzer and a parser. A lexical analyzer breaks a sequence of characters into tokens and identifies the type of each token. A parser takes a sequence of tokens from a lexical analyzer and then analyzes them to determine the structure and generates output depending on your need. For an interpreter, it is executed directly (e.g., a calculator with an arithmetic expression as the output). For a compiler, the output can be compiled code. Other than that, you can use JavaCC to create a text processor which extracts text based on certain lexical tokens in a file.

You can download JavaCC from <http://javacc.java.net/>. The latest version is 5.0 at the time of writing.

STRUCTURE OF A GRAMMAR FILE

EBNF (Extended Backus-Naur Form) contains a family of notations used to express the grammar of a formal language (not necessary a programming language). A grammar is defined by (unordered) production rules that specify how symbols can be combined together. For example, the following are EBNF production rules to define simple numbers:

```
<dot> ::= “.”
```

```
<digits> ::= ([“0” – “9”])+
```

```
<number> ::= <digits> | <digits> <dot> <digits> | <digits> <dot> | <dot> <digits>
```

An EBNF rule contains two parts (separated by ::= symbol): LHS (left-hand side) is the rule name and RHS (right-hand side) is the definition associated with the rule name. Options (separated by commas) are enclosed within brackets [and]. At most one of them is chosen. To put a range of characters in options, you can use a hyphen - between two characters. Choices are separated by vertical bars |. One of them is chosen. They can be parenthesized by (and) and suffixed with + (one or more), * (zero or more) or ? (zero or one) for repetitions.

JavaCC allows us to define grammars similar to EBNF. But, it separates grammars into tokens and production rules. JavaCC grammars are defined in a grammar file with jj as extension name. You can use JavaCC command line javacc.bat (for Windows) or javacc (for Unix/Linux) to generate Java files and then compile them into Java classes using a Java compiler.

The following is the basic structure of a JavaCC grammar file:

```
/* This section defines a set of options for the grammars. */
```

```
options {
```

```
    ...
```

```
}
```

```
/* This section defines the parser class to be generated and contains Java code. */
```

```
PARSER_BEGIN(parser_class)
```

```
public class parser_class {
```

```
    ...
```

```
}
```

```
PARSER_END(parser_class)
```

```
/* This section defines lexical specifications. Here, you specify token names and the corresponding regular expressions. The lexical analyzer breaks a sequence of characters into tokens and classifies tokens. */
```

```
/* Contains Java code to be included in the generated token manager. */
TOKEN_MGR_DECLS: {
    // Java code
}
/* Defines characters not to pass on to the parser. */
SKIP [[IGNORE_CASE]]: {
    ...
}
/* Defines tokens in the grammars */
TOKEN [[IGNORE_CASE]]: {
    ...
}
/* This section defines production rules in the grammars. Production rules can contain
Java code. They look like Java methods. This is where you define actions of a parser. */
...
```

Options

Options section allows you to provide options that control the behavior of generated Java code. Options can be specified either in the grammar file or from the javacc in the command line. Options set from javacc overrides those in the grammar file. Options section is optional and option names are not case-sensitive. For example,

```
options {  
    STATIC = false;  
    IGNORE_CASE = true;  
}
```

When `STATIC` is false, methods in the generated parser and token manager are not static. This allows multiple instances of parser to be present. By default, `STATIC` is true.

When `IGNORE_CASE` is true, the generated token manager ignore case in the token and input data. By default, `IGNORE_CASE` is false.

You can set `DEBUG_TOKEN_MANAGER` as true if you need to obtain debugging information of the token manager. For additional information about available options, you can find them from `javaccgrm.html` in doc directory.

Class declaration

The names inside the `PARSER_BEGIN` and `PARSER_END` should be the same and this is the name of generated Java class name for the parser. Some of generated Java files are preceded with the class name declared here.

You can have any Java code (just like creating a Java class) inside as long as it contains a class whose name is the same as the generated parser. If you need to put the generated files in a package, you can put a package name right after `PARSER_BEGIN`. If you have any variables (or methods) that can be used by other production rules, you can put them here too. Since JavaCC only generates Java source code and does not check Java syntax, you won't see any errors if there is any. You need to wait until they are compiled into class files.

Token manager declarations

The section of token manager declarations contains Java code to be included in the generated token manager and any declarations here are accessible by any lexical actions in lexical specifications.

Lexical specifications

Lexical specifications (or regular expression productions) define lexical entities (grammar tokens) processed by the token manager. The token manager is generated based on information here. A lexical specification contains a list of regular expression specifications separated by vertical bars |:

```
[lexical state list] {SKIP|TOKEN|MORE} [IGNORE_CASE] : {  
    regular_expression [Java block][:lexical_state]  
    | regular_expression [Java block][:lexical_state]  
    ...  
}
```

A lexical state list contains a list of lexical states (comma separated) surrounded by angular brackets < and > or you can use <*> for all states. It is optional. The default state is DEFAULT. When a regular expression is matched, a Java block is executed (an optional lexical action) and then move to next lexical state for further processing if it is specified. The token manager starts initially in the DEFAULT state. A Java block is a block of Java code surrounded by braces { and }. A lexical state is a Java identifier.

[IGNORE_CASE] is optional. It has the same effect as IGNORE_CASE in the options except [IGNORE_CASE] applies locally.

SKIP defines tokens that can be ignored by the token manager. For example, to skip certain whitespaces, you can use:

```
SKIP : {  
    " "  
    | "\r"  
    | "\t"  
}
```

TOKEN defines tokens in the grammar. Each token is treated as an instance of the Token class matched a regular expression in the token manager. Each token is surrounded by angular brackets < and > and separated by a vertical bar |. Token definition has two parts: token name and a regular expression (with optional Java code and lexical state). The format is <[#]{token_name}:{regular expression}>.

Token name: You can label a token with a name. A token name follows Java conventions on Java identifiers. If the token name is preceded by a # symbol, it is treated as a private regular expression and not matched as a token by the token manager. But, it still can be referred to from other regular expressions. It is a convenient way to reuse the same regular expression in other places.

Regular expression: A regular expression can contain a string literal (following Java string conventions) or a list of complex regular expressions separated by vertical bars |. A regular expression can also refer to other labeled tokens (surrounded token name by angular brackets < and >). A complex regular expression contains a sequence of items from the following: string literal, token, options, choices. These items can be parenthesized by (and) and suffixed with + (one or more), * (zero or more) or ? (zero or one) for repetitions. Options can be prefixed by a tilde ~ to exclude certain characters.

[IGNORE_CASE] is optional. It has the same effect as IGNORE_CASE in the options except [IGNORE_CASE] applies locally. A grammar file can have more than one Token section.

The following is an example of tokens defined in a calculator:

```
TOKEN: {
    < PLUS: "+" >
    | < MINUS: "-" >
    | < MULTIPLY: "*" >
    | < DIVIDE: "/" >
    | < COMMA: "," >
    | < #DOT: "." >
    | < OPEN_PAR: "(" >
    | < CLOSE_PAR: ")" >
    | < #DIGITS: ([ "0"-"9" ])+ >
    | < NUMBER: (<DIGITS> | <DIGITS> <DOT> <DIGITS> | <DIGITS> <DOT> |
<DOT> <DIGITS>) >
    | < EOL: "\n" >
}

TOKEN [IGNORE_CASE]: {
    < SUM: "SUM" <OPEN_PAR> >
    | < AVG: "AVG" <OPEN_PAR> >
}
```

MORE is used to build up a token gradually. Matched characters to MORE regular expression continue to store in a buffer until a match to a TOKEN or SKIP regular expression. For a match to TOKEN regular expression, a token is formed with the content in the buffer combined with current match and then passed to the parser. For a SKIP, the whole content is discarded.

Production rules

Each production rule (or production) has two parts. One is the Java block that contains declarations and Java code. This part declares the Java method to be generated and variables can be used in the production rule. A checked exception `ParseException` is thrown in the generated code. If other exceptions need to be thrown, you can add them here. The other is the expansion part for EBNF rule expansion. Each expansion part contains expansion units. Each expansion unit can contain a block of Java code surrounded by braces `{` and `}`. This is where action is executed during the parsing process. The following is the basic structure of a production rule:

```
{void or data type} {rule name()} [throws exception1,...] :  
{  
}  
{  
// expansion part that contains nested expansion units  
}
```

The following example shows an EBNF production rule and the corresponding production rule in JavaCC for an arithmetic expression with addition and subtraction operations:

EBNF production rule:

```
<expression> ::= <term> ( <plus> <term> | <minus> <term>)*
```

JavaCC production rule:

Double **expression()**:

```
{  
    Double a;  
    Double b;  
}  
{  
    a = term()  
    (  
        <PLUS> b = term()  
        {  
            if(a == null && b != null)  
                a = b;  
            else if(a != null && b != null)
```

```

a = new Double(a.doubleValue() + b.doubleValue());
}
| <MINUS> b = term()
{
if(a == null && b != null)
a = new Double( b.doubleValue()* (-1));
else if(a != null && b != null)
a = new Double(a.doubleValue() - b.doubleValue());
}
)*
{
return a;
}
}

```

The corresponding parts in the EBNF rule are highlighted in bold. `term()` is a production rule. The following is a snippet of generated code for production rule `expression()`:

```

final public Double expression() throws ParseException {
    Double a;
    Double b;
    a = term();
    label_3:
    while (true) {
        switch ((jj_ntk== -1)?jj_ntk():jj_ntk) {
        case PLUS:
        case MINUS:
        ...
        }
    }
}

```

A SIMPLE CALCULATOR

Now, we will use JavaCC to create a simple calculator that supports:

Binary operator: + (addition), - (subtraction), * (multiplication), / (division)

Unary operator: - (negative), + (positive)

Function: sum (summation), avg (average)

Grouping: (and)

Function names are case-insensitive and arguments for functions are comma separated expressions (or simple values). For example, `sum(1,2,3)`, `sum(1+2,3,4)` or `sum(1,2,avg(1,2,3))`.

EBNF grammars

A good practice is to start with EBNF grammars and then convert them to JavaCC grammars. EBNF grammars are shown as below:

$\langle \text{expression} \rangle ::= \langle \text{term} \rangle (\langle \text{plus} \rangle \langle \text{term} \rangle | \langle \text{minus} \rangle \langle \text{term} \rangle)^*$

$\langle \text{term} \rangle ::= \langle \text{unary} \rangle (\langle \text{multiply} \rangle \langle \text{element} \rangle | \langle \text{divide} \rangle \langle \text{element} \rangle)^*$

$\langle \text{element} \rangle ::= \langle \text{number} \rangle | \langle \text{open par} \rangle \langle \text{expression} \rangle \langle \text{close par} \rangle | \langle \text{function} \rangle$

$\langle \text{unary} \rangle ::= (\langle \text{plus} \rangle | \langle \text{minus} \rangle) \langle \text{element} \rangle$

$\langle \text{function} \rangle ::= (\langle \text{sum} \rangle | \langle \text{avg} \rangle) \langle \text{open par} \rangle (\langle \text{expression} \rangle [\langle \text{comma} \rangle \langle \text{expression} \rangle]^*)$

$\langle \text{close par} \rangle$

$\langle \text{sum} \rangle ::= \text{“SUM”}$

$\langle \text{avg} \rangle ::= \text{“AVG”}$

$\langle \text{open par} \rangle ::= \text{“ (“}$

$\langle \text{close par} \rangle ::= \text{“)”}$

$\langle \text{plus} \rangle ::= \text{“+”}$

$\langle \text{minus} \rangle ::= \text{“-“}$

$\langle \text{multiply} \rangle ::= \text{“*”}$

$\langle \text{divide} \rangle ::= \text{“/”}$

$\langle \text{number} \rangle ::= \langle \text{digits} \rangle | \langle \text{digits} \rangle \langle \text{dot} \rangle \langle \text{digits} \rangle | \langle \text{digits} \rangle \langle \text{dot} \rangle | \langle \text{dot} \rangle \langle \text{digits} \rangle$

$\langle \text{comma} \rangle ::= \text{“,”}$

$\langle \text{dot} \rangle ::= \text{“.”}$

$\langle \text{digits} \rangle ::= ([\text{“0”} - \text{“9”}])^+$

$\langle \text{EOL} \rangle ::= \text{“\n”}$

JavaCC grammars

Once the grammars are defined, you can convert them to JavaCC grammars. The following is the content of grammar file (Calculator.jj):

```
options {
    STATIC = false;
}

PARSER_BEGIN(Calculator)

/**
 * This is a lexical analyzer and parser for a calculator.
 */
public class Calculator {
    public static void main(String args[]) throws ParseException {
        Calculator parser = new Calculator(System.in);
        while(true) {
            System.out.println(parser.parseOneLine());
        }
    }
}

PARSER_END(Calculator)

SKIP : {
    " "
    | "\r"
    | "\t"
}

TOKEN: {
    < PLUS: "+" >
    | < MINUS: "-" >
    | < MULTIPLY: "*" >
```

```

| < DIVIDE: “/” >
| < COMMA: “,” >
| < #DOT: “.” >
| < OPEN_PAR: “(” >
| < CLOSE_PAR: “)” >
| < #DIGITS: ([“0”-“9”])+ >
| < NUMBER: (<DIGITS> | <DIGITS> <DOT> <DIGITS> | <DIGITS> <DOT> |
<DOT> <DIGITS>) >
| < EOL: “\n” >
}

```

```

TOKEN [IGNORE_CASE]: {
    < SUM: “SUM” <OPEN_PAR> >
    | < AVG: “AVG” <OPEN_PAR> >
}

```

Double parseOneLine():

```

{
    Double a;
}
{
    a = expression() (<EOF> | <EOL>)
    {
        return a;
    }
}

```

Double sum():

```

{
    Double result = null;
    Double param;
}

```

```

{
    <SUM> result = expression()
    (
    <COMMA> param = expression()
    {
    if(result == null && param != null)
    result = param;
    else if(result != null && param != null)
    result = new Double(param.doubleValue() + result.doubleValue());
    }
    )* <CLOSE_PAR>
    {
    return result;
    }
}

```

Double avg():

```

{
    int i = 1;
    Double result;
    Double param;
}
{
    <AVG> result = expression()
    (
    <COMMA> param = expression()
    {
    if(result == null && param != null)
    result = param;
    else if(result != null && param != null)
    {
    result = new Double(result.doubleValue() + param.doubleValue());

```

```

i++;
}
}
)* <CLOSE_PAR>
{
if(result != null && i != 0)
return new Double(result.doubleValue()/(double)i);
return null;
}
}

```

Double expression():

```

{
  Double a;
  Double b;
}
{
  a = term()
  (
  <PLUS> b = term()
  {
  if(a == null && b != null)
  a = b;
  else if(a != null && b != null)
  a = new Double(a.doubleValue() + b.doubleValue());
  }
  | <MINUS> b = term()
  {
  if(a == null && b != null)
  a = new Double( b.doubleValue()* (-1));
  else if(a != null && b != null)
  a = new Double(a.doubleValue() - b.doubleValue());
  }
  }
}

```

```
}
)*
{
return a;
}
}
```

Double term():

```
{
  Double a;
  Double b;
}
{
  a = element()
  (
  <MULTIPLY> b = element()
  {
  if(a == null || b == null)
  a = null; // If a number is multiplied by NULL, NULL is returned.
  else
  a = new Double(a.doubleValue() * b.doubleValue());
  }
  | <DIVIDE> b = element()
  {
  if(a == null || b == null)
  a = null; // If there exists any NULL value, NULL will be returned.
  else if(b.doubleValue() == 0)
  a = new Double(Double.NaN); // divided-by-zero
  else
  a = new Double(a.doubleValue() / b.doubleValue());
  }
  )*
}
```

```
{
return a;
}
}
```

Double unary():

```
{
    Double a;
}
{
    <MINUS> a = element()
    {
        return new Double(-a.doubleValue());
    }
    | <PLUS> a = element()
    {
        return a;
    }
}
```

Double element():

```
{
    Token t;
    Double a;
}
{
    t = <NUMBER>
    {
        return new Double(t.toString());
    }
    | a = sum()
    {
```

```
return a;
}
| a = avg()
{
return a;
}
| <OPEN_PAR> a = expression() <CLOSE_PAR>
{
return a;
}
| a = unary()
{
return a;
}
}
```


Generating code

Next, we can use javacc (under the bin directory of JavaCC installation) to generate Java code by running:

```
javacc Calculator.jj
```

The following Java files are generated:

Calculator.java: The generated parser.

CalculatorConstants.java: A list of constants for tokens.

CalculatorTokenManager.java: The generated token manager (or lexical analyzer).

ParseException.java: This exception is thrown for any parsing errors.

SimpleCharStream.java: This class contains characters to be processed by the lexical analyzer.

Token.java: This class represents a token.

TokenMgrError.java: This error is thrown for any lexical errors.

The first three files are grammar specific. The last four files are standard files generated by JavaCC.

In Calculator.java, JavaCC generates the following constructors automatically:

```
public Calculator(java.io.InputStream stream)
```

```
public Calculator(java.io.InputStream stream, String encoding)
```

```
public Calculator(java.io.Reader stream)
```

Inside these constructors, the lexical analyzer (the CalculatorTokenManager class) gets characters (the SimpleCharStream class) from the input (InputStream or Reader) and then the parser gets tokens from the lexical analyzer. The parser starts by calling the parseOneLine() method which accepts single line input. In this example, the input is from the console.

A FORMULA CALCULATOR

Now, we want to improve the calculator to accept a formula and calculate the formula based on variable values. For example, a formula calculates grades:

“midterm*0.3+final*0.3+report*0.4”.

To support defining variables in the formula, we add the following two new EBNF rules:

```
<letter> ::= [“a”-“z”,“A”-“Z”]
```

```
<variable> ::= <letter> ([“a”-“z”,“A”-“Z”,“0”-“9”, “_”])*
```

And, add <variable> into <element>

```
<element> ::= <number> | <open par> <expression> <close par> | <function> |  
<variable>
```

Next step is to change the grammar file. To put generated files in a package, we add `OUTPUT_DIRECTORY` to specify an output directory for the generated files. Also, add a new constructor to allow two parameters: one is a map that contains name-value pairs for variables and the other is a formula. To make it simple, only single value is allowed for variables.

```
options {  
    STATIC = false;  
    OUTPUT_DIRECTORY = “calculator”;  
}
```

```
PARSER_BEGIN(Calculator)
```

```
package calculator;
```

```
import java.io.StringReader;
```

```
import java.util.Map;
```

```
/**
```

```
* This is a lexical analyzer and parser for a formula calculator.
```

```
*/
```

```

public class Calculator {
    private Map<String, Double> variables;
    public Calculator(Map<String, Double> variables, String formula) throws
ParseException {
        this(new StringReader(formula));
        this.variables = variables;
    }
}
PARSER_END(Calculator)

```

Add the following in Token:

```
#LETTER: ["a"-“z”,“A”-“Z”] >
```

```
< VARIABLE: <LETTER> ([“a”-“z”,“A”-“Z”,“0”-“9”, “_”])* >
```

Now, we add a production rule to handle variables. The image field of Token is a string value that the token represents. We can use it to get the variable name.

Double variable():

```

{
    Token t;
    Double result = null;
}
{
    t = <VARIABLE>
    {
        String key = t.image;
        if(variables.containsKey(key))
            result = variables.get(key);
        else
            throw new ParseException(“Cannot find value for ” + key);

        return result;
    }
}

```

The Last thing we need to do is to add variable as part of element list by adding it into the production rule element():

```
a =variable()
{
    return a;
}
```

The following example shows how to use the improved calculator:

```
import java.util.Map;
import java.util.HashMap;

import calculator.Calculator;
import calculator.ParseException;

public class CalculateGrades {
    public static void main(String args[]) {
        Map<String, Double> grades = new HashMap<String, Double>();
        grades.put("midterm", 80.0);
        grades.put("final", 90.0);
        grades.put("report", 95.0);
        String formula = "midterm*0.3+final*0.3+report*0.4";
        try {
            Calculator calc = new Calculator(grades, formula);
            System.out.println(calc.parseOneLine());
        } catch (ParseException ex) {
            System.out.println(ex);
        }
    }
}
```

A TEXT PROCESSOR

Next, we are going to use an example to demonstrate how to use MORE and lexical states. As we mentioned before, the token manager starts initially in the “DEFAULT” state. If a lexical state is specified, the token manager moves to that state for further processing. At this example, the parser takes a Java source file as an input (through a standard input) and generates an output after stripping comments in the original file. This example covers cases of multiple line (`/* */` block) and single line (starting with `//`) comments. To make it simple, it does not cover cases of quoted strings. To handle quoted strings, you need to define a token for quoted string that covers all possible character escape codes (e.g., `\t`, `\`, `\"...`) supported by Java:

```
options {
```

```
    STATIC = false;
```

```
}
```

```
PARSER_BEGIN(CommentsRemover)
```

```
/**
```

```
* This is a lexical analyzer and parser for a comments remover.
```

```
*/
```

```
public class CommentsRemover {
```

```
    private static StringBuilder buffer = new StringBuilder();
```

```
    public static void main(String args[]) throws ParseException {
```

```
        CommentsRemover parser = new CommentsRemover(System.in);
```

```
        parser.parse();
```

```
        System.out.println(buffer);
```

```
    }
```

```
}
```

```
PARSER_END(CommentsRemover)
```

```
SKIP : {
```

```
    “/*” : MultiLineComment
```

```
}
```

```
<MultiLineComment> SKIP : {  
    “*/” : DEFAULT  
}
```

```
SKIP : {  
    “//” : SingleLineComment  
}
```

```
<SingleLineComment> SKIP : {  
    “\n” { input_stream.backup(1); } : DEFAULT  
}
```

```
<MultiLineComment,SingleLineComment> MORE : {  
    <~[]>  
}
```

```
TOKEN : {  
    <OTHER: ~[]>  
}
```

```
void parse():  
{  
    Token t;  
}  
{  
    (t = <OTHER>  
    {  
        buffer.append(t.image);  
    })*  
    <EOF>  
}
```

input_stream field represents an instance of SimpleCharStream in the token manager. By

calling `input_stream.backup(1)`, it backs up current position one character to not skip line feed character.

Apache Solr

Apache Solr is an open source search platform based on Apache Lucene running as a standalone server (starting from Solr 5.0, Solr is no longer distributed as a WAR). Lucene (<http://lucene.apache.org>) provides Java-based full-text indexing and search technology. Solr provides features like full-text indexing, hit highlighting, faceted search, rich documents (e.g., PDF, MS Word) indexing and database integration. Solr provides REST-like APIs which can be called over HTTP to make it easy to use. Solr allows customization through configuration and plugin architecture. There are also projects that provide libraries or plugins for integrating Solr with many programming languages or applications. For example, SolrJ is a Java client to access Solr through API calls.

You can download Apache Solr from <http://lucene.apache.org/solr>. The latest version is 5.4 (Java 7 or greater is needed) at the time of writing.

GETTING STARTED

Solr binary distribution includes a self-contained Solr web application using an installation of Jetty (a pure Java based HTTP server and servlet container as part of Eclipse Foundation). You can start a standalone Solr server using a script under the bin directory:

```
solr start
```

This will launch Jetty with Solr web application on port 8983 in the background. To access Solr Admin UI (Solr administration user interface), you can open link <http://localhost:8983/solr> in a web browser.



To stop a Solr server bound to port 8983, you can run

```
solr -stop -p 8983
```

Basic directory structure

Solr binary distribution contains an instance of Jetty with Solr web application (under the server directory). Solr 5.4 uses Jetty 9.2 and it contains the following directories:

contexts: This directory contains additional configurations for web applications deployed in Jetty. Here, solr-jetty-context.xml defines application context and the location of Solr web application.

etc: This directory contains configuration files for Jetty.

lib: This directory contains runtime JAR files for Jetty.

logs: This directory contains log files if logging is configured.

solr-webapp: Solr web application is deployed under webapp of this directory.

solr: the default Solr home directory

The default Solr home directory is “solr”. To use configuration other than the default one, you can specify that at the solr.solr.home system property while starting Solr server. For example, if you want to use “mysolr” as the home directory, you can start Solr with:

```
solr start -s mysolr
```

To learn more about Jetty, you can check Jetty Documentation Wiki from <http://www.eclipse.org/jetty/>.

Solr home directory

A typical Solr home directory contains the following items:

solr.xml: This is the primary configuration file for all cores. In Solr 4, this file specifies a list of Solr cores it should load. Starting from Solr 5, <cores> section is not supported anymore and automatic core discovery replaces manual core listing. Each <core> section is now replaced by core.properties in each Solr core directory. Automatic core discovery will try to find core.properties at any depth in subdirectories of the Solr home directory.

configsets: This is the base directory for configsets. A configset is identified by the name of directory. This allows cores to share configuration by specifying a configset while creating a new core. A path can be configured in solr.xml using the configSetBaseDir element.

Individual Solr core directories: For each Solr instance, it can have more than one core. Multiple cores allow one Solr instance with separate configurations and indexes, but still have the convenience of managing them as one application. That is one of the strategies to manage multiple indexes under one servlet container.

A library directory shared across all cores: This is optional. A path can be configured in solr.xml using the sharedLib element.

Solr core directory

A typical Solr core directory contains the following items:

core.properties: A Solr core is configured through a Java properties file. For example, to specify a name, you can use:

```
name=core1
```

This file can be empty. In which case, the core name is the directory this file is located.

conf: This directory is mandatory and must contain solrconfig.xml

(<http://wiki.apache.org/solr/SolrConfigXml>) and schema.xml

(<http://wiki.apache.org/solr/SchemaXml>).

data: This is the default location for indexes. It will be created if it does not exist. You can override this location in the solrconfig.xml.

lib: This directory is optional. Solr will load any JARs in this directory if it exists. You can override this location in the solrconfig.xml.

<https://cwiki.apache.org/confluence/display/solr/Defining+core.properties>

Creating a new core

You can create a new core using one of built-in configsets from the command line. For example, to create a new core, `basic_core`, with minimal configuration, you can use:

```
solr create_core -c basic_core -d basic_configs
```

Configset `basic_configs` provides a starting point to create a new core in Solr.

From the Admin UI, you can see the newly created core. You can switch to it from the Core Selector dropdown:



Other than using the command line to create a new core by copying an existing configset, you also can create the whole directory structure with required files into the Solr home directory directly.

Other than built-in configsets, Solr binary distribution also includes a few examples under the example directory. To start an example such as `techproducts`, which contains comprehensive examples of Solr features, you can start a Solr instance with the following command:

```
solr start -e techproducts
```

A core, `techproducts`, with sample data will be created.

SolrCloud

For high availability and fault tolerance, you can configure a cluster of Solr servers to handle distributed search and indexing through SolrCloud. To handle distributed indexing, an index (or a logical index) is split into several pieces (or shards). To handle distributed search, each shard can have more than one copy (or replica). Each replica exists as one core in a Solr server. In a cluster, a collection is a logical index that contains one or more shards spanning multiple machines.

DOCUMENT SCHEMA

The basic unit in Solr is document. Each document contains fields. The type of a field defines how a field is indexed and how it can be queried. For those familiar with database, a document is like a table row and a field is like a table column. Adding a new field into the schema won't affect existing data. All existing documents won't have the newly added field. It's the same for removing a field from the schema. It won't affect existing data. But, updating a field requires reindexing all documents. Or, part of document can be updated through atomic updates. This is a feature added in Solr 4.0 and it requires all fields to be configured as `stored="true"` except for fields which are destinations of `<copyField>`.

The fundamental concept of Solr is in indexing documents and running queries to return matched documents. In Solr, it's using inverted indexing. An inverted index stores a list of documents that each term appears in (term-to-document), this is useful for doing queries by terms. In some cases, such as sorting, faceting, and grouping, term-to-document is not efficient. `docValues` can be enabled for a field by adding `docValues="true"`. This will use forward indexing for a field (document-to-term).

To create your own schema, a good starting point is from a built-in configset. The basic structure of a document schema is shown below:

```
<schema>
  <fields>
    <field/>
    ...
    <dynamicField/>
    ...
  </fields>
  <uniqueKey>field name</uniqueKey>
  <copyField/>
  ...
  <types>
    <fieldType/>
    ...
  </types>
</schema>
```


Document fields

The <fields> element is where fields are declared. <fields> contains a list of <field> and <dynamicField> elements. The <field> element contains many attributes. Name and type are required. Name should consist of alphanumeric characters or underscore only, and should not start with a digit. Type is the name of a field type from the <fieldType> element in the <types> element. For example, id field is defined as:

```
<field name="id" type="string" indexed="true" stored="true" required="true"
multiValued="false" />
```

type: The field type can affect case-sensitivity in searches. The string type (the StrField class) is not analyzed (no pre-processing in input data). You search a string with an exact match. The text_general (the TextField class) performs tokenization and lowercasing. Hence, it allows partial and case-insensitive matches.

indexed: true if this field should be indexed to be searchable, sortable, and facetable.

stored: true if the original value of this field should be returned in a search.

required: true if this field is required. An error will be thrown if the value does not exist.

multiValued: true if this field can contain multiple values per document.

By default, a new document will replace existing document with the same unique key (a field defined as a uniqueKey in the schema.xml).

<dynamicField> allows you to declare dynamic fields. This gives you flexibility not to declare every field to be indexed explicitly. Field names are matched by patterns with a wildcard "*" symbol at the start or the end. For example,

```
<dynamicField name="*_i" type="int" indexed="true" stored="true"/>
```

Once a field is indexed, it is added to the schema. After that, it is just like a regular field. This means you cannot add field values with conflicting data type. Also, in the queries, you need to provide exact field names. Dynamic fields provide flexibility. But, new fields might be created accidentally because of typos in the field names of documents. Special attention is needed especially in the production environment.

<copyField> copies one field to another at the time a document is added to the index. This is a convenient way to index the same data differently in different fields. For example, one field for searching (tokenized, case-folded and punctuation-stripped) and the other field for sorting (un-tokenized, case-folded).

You also can add multiple fields to the same field. For example,

```
<copyField source="cat" dest="text"/>
```

```
<copyField source="name" dest="text"/>
```

Here, the text field needs to be a multivalued field.

To see fields defined in a schema, you can use Schema Browser from the Admin UI:

name

Field

name

Copied to

text

Type

text_general

Unique Key Field

id

Field: name

Field-Type: org.apache.solr.schema.TextField

PI_Gap: 100

Docs: 1

Flags: Indexed Tokenized Stored

Properties	✓	✓	✓
Schema	✓	✓	✓
Index	✓	✓	✓

? Index Analyzer: org.apache.solr.analysis.TokenizerChain

? Query Analyzer: org.apache.solr.analysis.TokenizerChain

i Load Term Info

Field types

Solr field types are subclasses of the FieldType class, such as TextField, StrField. Field type definitions in the schema can be created based on those classes. The following are some basic type definitions in a Solr sample schema:

boolean

int (or tint)

float (or tfloat)

long (or tlong)

double (or tdouble)

date (or tdate)

string

text_general

The following are two field definitions for a numeric type, integer:

```
< fieldType name="int" class="solr.TrieIntField" precisionStep="0"
positionIncrementGap="0"/>
```

```
< fieldType name="tint" class="solr.TrieIntField" precisionStep="8"
positionIncrementGap="0"/>
```

Smaller precisionStep values (in bits) will generate more tokens indexed per value and can have faster range queries. But, the index size is larger too. A precisionStep of 0 disables indexing at different precision levels. positionIncrementGap can be used to put additional spaces between terms in multivalued fields to prevent false phrase matching.

A field of TextField type usually contains an analyzer. An analyzer is responsible for analyzing contents of text and generates tokens. At index time, the outputs are used to build indices. At query time, the outputs are used to match query results. An analyzer contains a sequence of tokenizers and filters. Tokenizers can break text into tokens and filters can be used to filter those tokens. The same operations can be used at both index time and query time. Or, they can contain different operations. The following is a text type, text_general, defined in the sample schema:

```
< fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
  < analyzer type="index">
    < tokenizer class="solr.StandardTokenizerFactory"/>
    < filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt" />
    <!-- in this example, we will only use synonyms at query time
    < filter class="solr.SynonymFilterFactory" synonyms="index_synonyms.txt"
ignoreCase="true" expand="false"/>
```

—>

```
<filter class="solr.LowerCaseFilterFactory"/>
```

```
</analyzer>
```

```
<analyzer type="query">
```

```
<tokenizer class="solr.StandardTokenizerFactory"/>
```

```
<filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt" />
```

```
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
```

```
<filter class="solr.LowerCaseFilterFactory"/>
```

```
</analyzer>
```

```
</fieldType>
```

If you need to define a field with non-tokenized and case-insensitive content, you can define it as:

```
<fieldType name="text_ci" class="solr.TextField" sortMissingLast="false"
omitNorms="true">
```

```
<analyzer>
```

```
<!-- KeywordTokenizer does not do any tokenization. The entire input string is
preserved as a single token. -->
```

```
<tokenizer class="solr.KeywordTokenizerFactory"/>
```

```
<filter class="solr.LowerCaseFilterFactory" />
```

```
<!-- Removes any leading or trailing whitespace -->
```

```
<filter class="solr.TrimFilterFactory" />
```

```
</analyzer>
```

```
</fieldType>
```

Field analysis

To see how a field value is analyzed for index or query, you can use Analysis from the Admin UI.

Field Value (Index): Solr, the Enterprise Search Server

Field Value (Query):

Analyse Fieldname / FieldType: name Verbose Output [Analyse Values](#)

	text	Solr	the	Enterprise	Search	Server
ST	raw_bytes	[53 6f 6c 72]	[74 68 65]	[45 6e 74 65 72 70 72 69 73 65]	[53 65 61 72 63 68]	[53 65 72 76 65 72]
	start	0	6	10	21	28
	end	4	9	20	27	34
	positionLength	1	1	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3	4	5
SE	raw_bytes	[53 6f 6c 72]	[74 68 65]	[45 6e 74 65 72 70 72 69 73 65]	[53 65 61 72 63 68]	[53 65 72 76 65 72]
	start	0	6	10	21	28
	end	4	9	20	27	34
	positionLength	1	1	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3	4	5
LCE	raw_bytes	[73 6f 6c 72]	[74 68 65]	[65 6e 74 65 72 70 72 69 73 65]	[73 65 61 72 63 68]	[73 65 72 76 65 72]
	start	0	6	10	21	28
	end	4	9	20	27	34
	positionLength	1	1	1	1	1
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	3	4	5

UPDATING DATA

So far, the server does not contain any data. To add/update data into a Solr index, you need to post commands to the server. You can use update request handler registered in the solrconfig.xml:

```
<requestHandler name="/update" class="solr.UpdateRequestHandler"/>
```

The UpdateRequestHandler supports commands specified in a variety of formats: XML, CSV, JSON, or JAVABIN. To post commands, you can use the following methods:

post.jar: This is a simple command line tool (using HTTP POST method) in the “example/exampledoc” directory. It allows you to post data from files in a variety of formats (including rich documents such as PDF or MS Office files). You can run post.jar with -h option for detailed usage and examples.

cURL: This is a command line tool for transferring data with URL syntax. It supports a variety of operating systems. You can download it from <http://curl.haxx.se>.

HTTP GET method: There is no limit in HTTP specification for the length of a URL. But, many clients and servers have such limit in length. Do not use it to send big chunk of data. It is only good for short requests. A GET request needs to be URL-encoded.

XML messages

XML is a popular and simple way to store structured data and transport data between locations. Solr allows posting XML messages for updating a Solr index. It is possible to combine multiple commands in one XML message. Since an XML document only allows one root element, you need to enclose commands in an <update> element. All commands in an XML message are executed in order. If any command fails, everything below it will not be executed. For more information on XML elements and attributes for XML messages, please see <http://wiki.apache.org/solr/UpdateXmlMessages>.

add

You can add/replace documents using the <add> command. By default, existing documents will be replaced newly added documents with the same unique key defined in the schema. This can be disabled by setting an optional attribute *override* for <add> as false. Multiple documents can be specified in a single <add> command. Each document can have more than one field. A multivalued field can appear multiple times in a document. The following is the basic structure

```
<add>
  <doc>
    <field name="field1">value1</field>
    <field name="field2">value2</field>
    ...
  </doc>
  <doc>...</doc>
  ...
</add>
```

Next, you can use post.jar to add documents by using sample files in “example/exampledoc” directory. Before you can do that, you need to create a core with a schema that can process those sample documents from a built-in configset, sample_techproducts_configs, by running the following command:

```
solr create_core -c sample_core -d sample_techproducts_configs
```

Now, we are ready to post the first document to this core by running

```
java -Dc=sample_core -jar post.jar solr.xml
```

solr.xml only contains one document. You will get the following response:

```
SimplePostTool version 5.0.0
```

```
Posting files to [base] url http://localhost:8983/solr/sample_core/update using content-type
```

application/xml...

POSTing file solr.xml to [base]

1 files indexed.

COMMITting Solr index changes to http://localhost:8983/solr/sample_core/update...

Time spent: 0:00:00.458

You have just indexed your first document to Solr. By default, the host name is localhost and the port number is 8983. You can use `-Dhost` to specify a host name and `-Dport` to specify a port number. Since the default content type is `application/xml`, you do not need to use `-Dtype` to specify content type.

To perform the same operation through `cURL`, you can run:

```
curl http://localhost:8983/solr/sample_core/update -H "Content-Type: text/xml" --data-binary @solr.xml
```

You will get the following response in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader"><int name="status">0</int><int name="QTime">13</int>
</lst>
</response>
```

`QTime` means how long it took to execute your query. The unit is milliseconds.

A valid content header is required to use HTTP POST. Using `-H "Content-Type: text/xml; charset=utf-8"` if that is for UTF-8. You can either contain XML message (enclosed by double quotes) or a file (starting the data file with a `"@"` symbol) with `—data-binary` option. If you need an explicit commit to happen immediately, you can use `update?commit=true`.

Atomic document updates allows updating individual fields without sending the entire document to Solr. This can be done through the `update` attribute in the field element. Available options are `add`, `set` and `inc`. If you need to set the value of a field to null, you can use `update="set" null="true"` in the field element. For example, to replace the price field of a document with a new value, you can run:

```
curl http://localhost:8983/solr/sample_core/update?commit=true -H "Content-Type: text/xml" --data-binary "<add><doc><field name=\"id\">SOLR1000</field><field name=\"price\" update=\"set\">200</field></doc></add>"
```

For a multivalued field, you can use `update="add"` to add a new value to it. To increase value of a field, you can use `update="inc"`.

To update documents atomically, make sure `<updateLog>` is configured in the `solrconfig.xml`. For example:

```
<updateHandler class="solr.DirectUpdateHandler2">
```



```
<updateLog>
  <str name="dir">${solr.ulog.dir}</str>
</updateLog>
</updateHandler>
```

This enables the transaction log for recovery.

commit

Changes to an index are not visible to new search requests until a commit command is sent. There are two types of commit: hard commit and soft commit. A hard commit syncs changes to the index and roll over the log (if the updateLog is enabled). No data will lose in case of system failure. This can prevent the log from growing too big (for uncommitted updates). A soft commit only makes changes visible and does not sync changes to a persistent storage. In case of system failure, changes that occurred after the last hard commit could be lost. Soft commit is much faster than hard commit. For near real time (NRT) search, you need to use soft commit more and hard commit less. By default, hard commit is performed for a commit operation. You can set optional attribute softCommit as true to perform soft commit. For example,

```
<commit softCommit="true"/>
```

You can send a commit command after sending the last document. Or, you can use Solr to perform a hard commit (or soft commit) automatically under certain conditions. You can combine both hard commit and soft commit to fine tune performance. You can configure them under the updateHandler element in the solrconfig.xml. For example,

```
<updateHandler>
  ...
  <autoCommit> <!-- hard commit -->
    <maxDocs>10000</maxDocs> <!-- maximum uncommitted documents before auto
commit -->
    <maxTime>15000</maxTime> <!--maximum time (ms) after adding a new document
before auto commit -->
    <openSearcher>false</openSearcher> <!-- If false, don't open a new searcher on hard
commit. This means newly added documents are not visible. -->
  </autoCommit>
  <autoSoftCommit> <!-- soft commit -->
    <maxTime>1000</maxTime>
  </autoSoftCommit>
  ...
</updateHandler>
```

Another commit strategy is to ask Solr to commit `<add>` commands within a certain time by using `commitWithin`. For example, `<add commitWithin="1000">` is to ask Solr to commit this `<add>` within 1000ms.

The following example is to send a `<commit>` command through HTTP GET. You can use a browser to run it directly. The request needs to be URL-encoded:

```
http://localhost:8983/solr/sample_core/update?stream.body=%3Ccommit/%3E
```

delete

You can delete documents using the `<delete>` command. You can specify the value of a unique key field in the delete XML message (delete by ID). For example,

```
<delete><id>SOLR1000</id></delete>
```

Or, you can delete documents by a query (delete by query). For example,

```
<delete><query>name:Solr</query></delete>
```

Delete by query uses the Lucene query parser. We will discuss about query syntax later.

Also, you can have multiple delete operations (delete by ID, delete by query) in the same `<delete>` section.

If you want to clear Solr index, you can use:

```
<delete><query>*:*</query></delete>
```

For example,

```
java -Dc=sample_core -Ddata=args -jar post.jar "<delete><query>*:*</query></delete>"
```

```
curl http://localhost:8983/solr/sample_core/update?commit=true -H "Content-Type: text/xml" --data-binary "<delete><query>*:*</query></delete>"
```

```
http://localhost:8983/solr/sample_core/update?stream.body=<delete><query>*:*</query></delete>&commit=true
```

CSV

Solr accepts data in CSV (comma separated values) format. That includes files exported from Excel or MySQL. Furthermore, parameters such as separator or escape are configurable. You can use either `/update/csv` or `/update` to post CSV data. The content type is either `application/csv` or `text/csv`. The following is a sample CSV data:

```
id,cat,name
00001,"book,Java",Effective Java
00002,"book,C",Programming in C
```

By default, the first line is the header of CSV data. It contains a list of comma separated field names. If you need to preserve characters such as separator or whitespace in the field value, you can surround it by encapsulator. The default encapsulator is double quote `"`. If the encapsulator itself needs to appear in the encapsulated value, you can use two encapsulators to escape it. CSV data supports multivalued fields. To process a multivalued field, you can set `split` parameter of that field as `true`. Once it is `true`, the field value is split into multiple values by another CSV parser. For example, `cat` field in above sample is a multivalued field. You can use `f.cat.split=true` in the request URL to index them as separate values:

```
curl "http://localhost:8983/solr/update?commit=true&f.cat.split=true" -H "Content-Type: text/csv" --data-binary @books.csv
```

By default, the separator of multivalued fields is the same as the field separator. You can change that by using `f.<field name>.separator=value` (e.g., `f.cat.separator=|`).

Note: URL is surrounded by double quotes because there is more than one parameter in the query string. The separator `&` in the query string might be treated as part of a command in the command line mode (e.g., a command separator or run preceding command in the background). Similar problem may happen in `-Dparams` option of `post.jar` if there is more than one query parameter. You can surround the whole option by double quotes if it happens.

QUERYING DATA

Request handler

To provide flexibility, Solr allows pluggable custom code. This is called plugin. There are a variety of classes in Solr can be treated as a plugin. For example, request handlers are used to handle query requests. A request handler implements the `SolrRequestHandler` interface. The `SearchHandler` class is the primary request handler provided by Solr to handle search requests.

In Solr, a request is dispatched to a specific request handler based on the path specified in the request URL. All request handlers are defined in the `solrconfig.xml`. There are different built-in request handlers in Solr. For example, a search request handler is defined as:

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str> <!-- indicates what kind of request parameters
should be included in the response. "explicit" means only the parameters specified in the
request are included. -->
    <int name="rows">10</int> <!-- max. number of documents returned per request -->
    <str name="df">text</str> <!-- default search field -->
  </lst>
</requestHandler>
```

Default values for query parameters can be specified inside the `<requestHandler>` element under the `<lst>` element with name "defaults". Other than "defaults", there are two additional sets of query parameters: `appends` and `invariants`. "appends" defines additional parameter values that will be included. "invariants" defines parameter values that will not be overridden.

Default values can be overridden in the request URL. To reach a handler, you can use the request URL with the following format:

```
http://{host:port}/{context root}/{core}/{handler name}
```

For example, to reach a search request handler mentioned above in `sample_core`, you can use:

```
http://localhost:8983/solr/sample_core/select
```

Multiple instances of the same request handler can be registered more than once with different names (and different parameters).

Query parameters

Search handlers are handlers that process search queries and return a list of results. The primary search handler provided by Solr is the SearchHandler class. There are multiple instances of SearchHandler registered with different names in the solrconfig.xml. The following are main query parameters supported by the SearchHandler:

q: This is the only mandatory query parameter. It specifies the main query for the request. Solr supports different types of queries and syntaxes. Hence, a query parser is needed to convert the main query to the correct Lucene query. The standard Solr query parser syntax is a superset of the Lucene query parser. That is the default query parser. For available query parsers, please check <http://wiki.apache.org/solr/SolrQuerySyntax>.

fl: This parameter specifies a set of fields (comma or space separated) to be returned. The string “score” can be used to indicate the score of each document should be returned. The string “*” can be used to indicate all fields (the default value).

fq: This parameter is for filter queries. fq can be specified multiple times in the query string. Filter queries can be used to filter the search results from the main query (using q parameter) without influencing score. Since filter queries are cached independently from the main query, it can increase query speed. In faceted search (will be introduced later), filter queries allows users to drill down and narrow search results.

start and rows: These two parameters are used to paginate results from a query. start (default value is 0) indicates the offset in the complete result set. rows (default value is 10) indicates the maximum number of documents to return per request.

sort: This parameter is for defining sorting on one or more fields (comma separated) with a sort direction (asc or desc). The default value for sorting is “score desc”. Score is a pseudo field for document score.

defType: This parameter specifies the query parser for the main query.

debugQuery: If additional debugging information needs to be added in the response, you can include this parameter (regardless of its value) in the query.

By default, Solr is using the standard Lucene query parser. To use q parameter in the query, you need to understand query syntax first:

Fields

You can search any field with the value you are looking for with <field>:<term>. <field> is case-sensitive. If the field name is missing, the default search field will be used. There are two types of terms: single terms and phrases. A single term is a single word. For example, to search for word “Solr” in the name field, you can use name:Solr. A phrase is a group of words surrounded by double quotes. For example, to search for phrase “phone features” in the name field, you can use features:”phone book”.

Boolean operators

If multiple terms are involved, you can combine them through Boolean operators. Lucene supports the following operators:

OR (or ||): matches documents where either term exists. This is the default operator if there is no Boolean operator between two terms.

AND (or &&): matches documents where both terms exist.

+: requires the term after the + symbol exists in a document.

NOT (or !): excludes documents that contain the term after NOT.

-: excludes documents that the term after the - symbol exists.

Boolean operators must be all capitals. For example, “name:Solr or name:One” means matching documents on the name field for “Solr”, or on the default search field for “or”, or on the name field for “One”. That is different from “name:Solr OR name:One”.

Grouping searches

You can use parentheses to group clauses to form sub queries. For example, (name:Solr OR name:One) AND inStock=true. Also, you can use parentheses to group multiple clauses to a single field. For example, name:(+Solr +One).

Wildcard searches

You are allowed to use single character wildcard (?) and multiple character wildcard (*) within single terms (not within phrases). But, you cannot use * or ? as the first character of a search. They are only allowed in the middle or the end of a search.

Regular expression searches

Regular expression is supported starting from 4.0. To do a regular expression search, use a pattern surrounded by /. Please check the RegExp class in Lucene for the supported syntax. For example, to search documents containing iPod or iPad in the name field, you can use `name:/iP[ao]d/`.

Proximity searches

If you need to find words within a specific distance away, you can do proximity searches. An exact match is proximity 0. To do a proximity search, use the ~ symbol with a number at the end of a phrase. For example, to search documents that “power” and “iPod” are within 2 words away, you can use “power iPod”~2.

Range searches

To match documents with field values between the lower and upper bounds, you can use range searches. Range queries can be inclusive (use [or]) or exclusive (use { or }) of the upper and lower bounds. For example, price: [100 TO 1000]. * is allowed in the range searches. For example, price:[100 TO *] is for price values greater than or equal to 100. To find all document with a value for a field, you can use [* TO *].

Boosting terms

You can make a term more significant than others by boosting it. To boost a term, you use ^ after the term with a number (boost factor). The higher the boost factor, the more relevant the term will be. Therefore, boosting a term can control the relevance of corresponding document (and sorting order). By default, the boost factor is 1 and must be positive.

Special characters

If you need to use any special characters that are part of the query syntax, you need to escape them. The following are special characters:

+ - && || ! () { } [] ^ “ ~ * ? : \ /

To escape these special characters, you use the \ before the character.

To query data, you can use HTTP GET method on the /select with parameters mentioned above. Assuming you already posted and committed all XML documents in the `exampledocs` directory. Now, you can run the following query to search on the name field for “Solr” and request name and price fields to be returned:

```
http://localhost:8983/solr/sample_core/select?q=name:Solr&fl=name,price
```

The response is:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">52</int>
    <lst name="params">
      <str name="fl">name,price</str>
      <str name="q">name:Solr</str>
    </lst>
  </lst>
  <result name="response" start="0" numFound="1">
    <doc>
      <str name="name">Solr, the Enterprise Search Server</str>
      <float name="price">0.0</float>
    </doc>
  </result>
</response>
```

By default, all fields are returned. You can use the `fl` parameter to specify fields to be returned. By default, response is in XML format. Available format types are defined by the `<queryResponseWriter>` element in the `solrconfig.xml`. You can include the `wt` parameter in the query to generate different format. For example, you can change response format to CSV by using `wt=csv`:

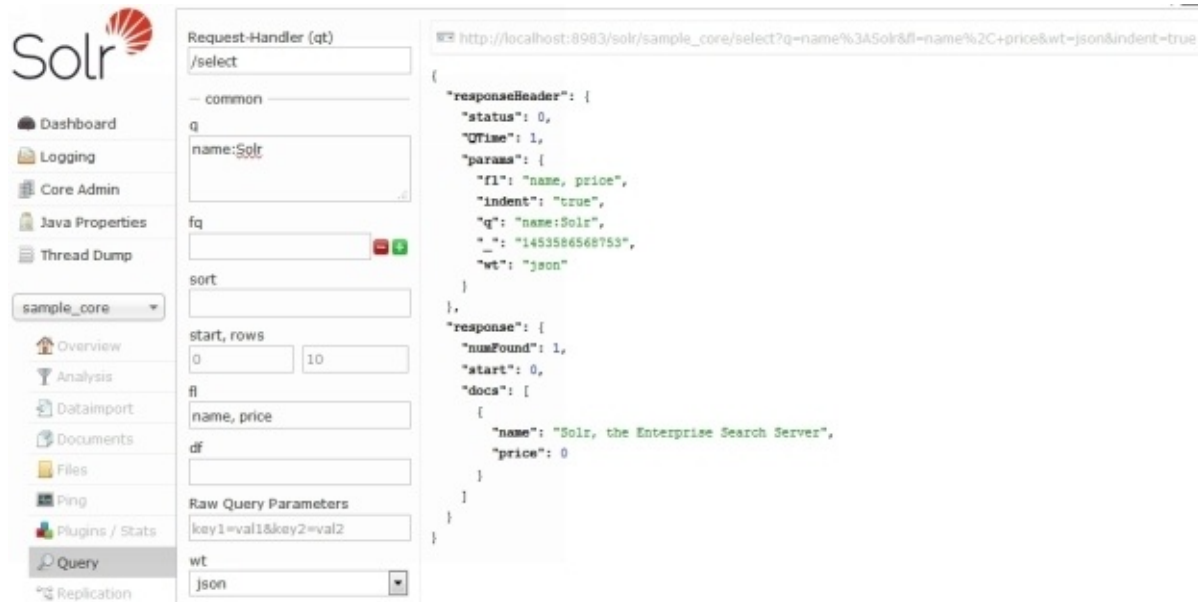
http://localhost:8983/solr/sample_core/select?q=name:Solr&fl=name,price&wt=csv

The response is:

name,price

“Solr, the Enterprise Search Server”,0.0

Other than using command line to query data, you also can use the Query tab in Solr Admin UI to query data:



The screenshot displays the Solr Admin UI interface. On the left is a navigation sidebar with options like Dashboard, Logging, Core Admin, Java Properties, Thread Dump, and Query (which is selected). The main area is titled 'Request-Handler (qt) /select'. It contains several input fields: 'q' with the value 'name:Solr', 'fq' (empty), 'sort' (empty), 'start, rows' with '0' and '10', 'fl' with 'name, price', 'df' (empty), 'Raw Query Parameters' with 'key1=val1&key2=val2', and 'wt' set to 'json'. The right side of the interface shows the resulting JSON response in a code editor. The response includes a 'responseHeader' with status, QTime, and parameters, and a 'response' object containing 'numFound': 1, 'start': 0, and a 'docs' array with one document: {'name': 'Solr, the Enterprise Search Server', 'price': 0}.

```
http://localhost:8983/solr/sample_core/select?q=name:Solr&fl=name,price&wt=json&indent=true
```

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 1,
    "params": {
      "fl": "name, price",
      "indent": "true",
      "q": "name:Solr",
      "_": "1453586568753",
      "wt": "json"
    }
  },
  "response": {
    "numFound": 1,
    "start": 0,
    "docs": [
      {
        "name": "Solr, the Enterprise Search Server",
        "price": 0
      }
    ]
  }
}
```

Sorting

By default, query results are sorted by score in descending order (by relevance). If you need to sort results on indexed fields, you can use the sort parameter. The format is:

```
sort=<field name> <direction>[,<field name> <direction>...]
```

Direction is either asc (ascending) or desc (descending). Sorting is only allowed on fields with multiValued="false" and indexed="true". For example,

```
http://localhost:8983/solr/sample_core/select?  
q=name:CORSAIR+AND+inStock:true&sort=price+asc,popularity+desc
```

Query string needs to be URL-encoded. If you are using cURL, you can use `—data-urlencode` to encode query string for you. For example,

```
curl —data-urlencode "q=name:CORSAIR AND inStock:true&sort=price asc,popularity  
desc" http://localhost:8983/solr/sample_core/select
```


Hit highlighting

Solr provides hit highlighting that returns highlighted matches and snippets in field values. To enable highlighting, you need to set the hl parameter as true and specify a list of fields to highlight using the hl.fl parameter. Fields are comma or space separated. A "*" symbol can be used. For example, to highlight "GPU" on features field of matched documents, you can use:

```
http://localhost:8983/solr/sample_core/select?q=GPU&wt=xml&hl=true&hl.fl=features
```

A response contains two sections. The first section contains matched documents and the second section contains highlighting fields. If matched documents are not needed, you can include rows=0 in the query to improve performance. Highlighted words are surrounded with elements. For example,

```
<result name="response" numFound="2" start="0">
  <doc>...</doc>
  <doc>...</doc>
</result>
<lst name="highlighting">
  <lst name="EN7800GTX/2DHTV/256M">
    <arr name="features">
      <str>NVIDIA GeForce 7800 GTX <em>GPU</em>/VPU clocked at 486MHz</str>
    </arr>
  </lst>
  <lst name="100-435805">
    <arr name="features">
      <str>ATI RADEON X1900 <em>GPU</em>/VPU clocked at 650MHz</str>
    </arr>
  </lst>
</lst>
```

By default, the size of the snippets is 100 characters. You can use the hl.fragsize parameter to change it. For additional highlighting parameters, please see

<http://wiki.apache.org/solr/HighlightingParameters>.

Faceted search

Faceted search can break results into categories and generate counts for each category. This allows users to drill down and narrow results by using the selected value to construct a filter query. To enable faceting, you set the facet parameter as true.

Field faceting

The facet.field parameter allows you to specify a field as a facet to generate count on each indexed term in that field. It can be used multiple times for multiple facets. Since faceting fields are served as human-readable text and value of drill-down filter query, they should be defined as a string type (of type StrField) and won't be analyzed (e.g., tokenized or lowercased). For example, to have facets on 2 fields – cat and inStock, you can use:

```
http://localhost:8983/solr/sample_core/select?
q=*&rows=0&facet=true&facet.field=cat&facet.field=inStock
```

If the response format is XML, the following section of XML is appended to the end of response for a variety of facet counts:

```
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="cat">
      <int name="electronics">12</int>
      <int name="currency">4</int>
      ...
    </lst>
    <lst name="inStock">
      <int name="true">17</int>
      <int name="false">4</int>
    </lst>
  </lst>
  ...
</lst>
```

To further drill down all items in stock, you can add a filter query fq=inStock:true as follows:

```
http://localhost:8983/solr/sample_core/select?
q=*&rows=0&facet=true&facet.field=cat&fq=inStock:true
<lst name="facet_counts">
```

```
<lst name="facet_queries"/>
<lst name="facet_fields">
  <lst name="cat">
    <int name="electronics">8</int>
    <int name="currency">4</int>
  </lst>
  <lst>
    ...
</lst>
```

By default, sorting is based on count (highest count first). To limit the maximum number of terms can be returned, you can use the `facet.limit` parameter. Combining with the `facet.offset` parameter, you can page through faceting result. Many parameters can be specified on a per field basis with `f.<field name>.<param>=<value>`, such as `f.cat.facet.limit=10`.

Query faceting

The `facet.query` parameter allows you to specify a query using default query syntax as a facet constraint to generate a count. It can be used multiple times for multiple facets. For example,

```
http://localhost:8983/solr/sample_core/select?
q=cat:memory&rows=0&facet=true&facet.query=[0 TO 100]&facet.query=[100 TO *]
```

The response for facet queries is:

```
<lst name="facet_counts">
  <lst name="facet_queries">
    <int name="[0 TO 100]">0</int>
    <int name="[100 TO *]">3</int>
  </lst>
  <lst name="facet_fields"/>
  <lst name="facet_dates"/>
  <lst name="facet_ranges"/>
</lst>
```

Range faceting

Date faceting has been deprecated as of Solr 3.1 in favor of the more general range faceting. You can use range faceting on any fields (date or numeric) that support range queries.

facet.range: This parameter specifies a field to create a range facet. It can be used multiple times for multiple facets.

facet.range.start: This parameter defines lower bound of the range. By default, the lower bound is inclusive. If there are multiple fields, this parameter can be used on a per field basis with format f.<field name>.facet.range.start.

facet.range.end: This parameter defines upper bound of the range. By default, the upper bound is exclusive. This parameter can be used on a per field basis.

facet.range.gap: This parameter defines the distance between values. This parameter can be used on a per field basis. Variable width gaps are allowed by using a list of comma separated values.

For example,

```
http://localhost:8983/solr/sample_core/select/?q=price:[0 TO
*]&rows=0&facet=true&facet.range=price&facet.range.start=0&facet.range.end=500&fac
```

The response is:

```
<result name="response" numFound="16" start="0"/>
```

```
<lst name="facet_counts">
```

```
....
```

```
<lst name="facet_ranges">
```

```
<lst name="price">
```

```
<lst name="counts">
```

```
<int name="0.0">7</int>
```

```
<int name="100.0">2</int>
```

```
<int name="200.0">1</int>
```

```
<int name="300.0">3</int>
```

```
<int name="400.0">1</int>
```

```
</lst>
```

```
<float name="gap">100.0</float>
```

```
<int name="before">0</int>
```

```
<int name="after">2</int>
```

```
<int name="between">14</int>
```

```
<float name="start">0.0</float>
```

```
<float name="end">500.0</float>
```

```
</lst>
```

```
</lst>
```

....

</lst>

numFound indicates there are 16 matched documents in the query. But, there are only 14 documents in the result of range faceting. That is because there are 2 documents with field values greater than the upper bound of the last range. You can this kind of information from facet.range.other=all.

Pivot Faceting

Field faceting allows you to do faceting on one level only. To work on hierarchical structure of data, you can use a facet.pivot parameter to specify a list of fields to pivot through and generate facet counts for possible permutations. It can be used multiple times for multiple pivot facets. The following example shows a two-level pivot faceting:

http://localhost:8983/solr/sample_core/select?
q=*&rows=0&facet=true&facet.pivot=cat,inStock

```
<lst name="facet_pivot">
  <arr name="cat,inStock">
    <lst>
      <str name="field">cat</str>
      <str name="value">electronics</str>
      <int name="count">12</int>
    </arr name="pivot">
    <lst>
      <str name="field">inStock</str>
      <bool name="value">>true</bool>
      <int name="count">8</int>
    </lst>
    <lst>
      <str name="field">inStock</str>
      <bool name="value">>false</bool>
      <int name="count">4</int>
    </lst>
  </arr>
</lst>
<lst>
```

```
<str name="field">cat</str>
<str name="value">currency</str>
<int name="count">4</int>
<arr name="pivot">
<lst>
<str name="field">inStock</str>
<bool name="value">true</bool>
<int name="count">4</int>
</lst>
</arr>
</lst>
```

...

For additional parameters on faceted search, please see <http://wiki.apache.org/solr/SolrFacetingOverview>.

Result Grouping

For matched documents, they can be grouped based on a common field (cannot be a multivalued field) through result grouping. To enable result grouping, you need to set `group=true` and specify a field for grouping through the `group.field` parameter. The following example shows a result grouping based on values of `inStock` field:

`http://localhost:8983/solr/sample_core/select?q=inStock:`

`[*%20TO%20*]&fl=name,inStock&group=true&group.field=inStock&group.ngroups=tru`

```
<lst name="grouped">
  <lst name="inStock">
    <int name="matches">21</int>
    <int name="ngroups">2</int>
    <arr name="groups">
      <lst>
        <bool name="groupValue">true</bool>
        <result name="doclist" numFound="17" start="0">
          <doc>
            <str name="name">Test with some GB18030 encoded characters</str>
            <bool name="inStock">true</bool>
          </doc>
        </result>
      </lst>
      <lst>
        <bool name="groupValue">>false</bool>
        <result name="doclist" numFound="4" start="0">
          <doc>
            <str name="name">Belkin Mobile Power Cord for iPod w/ Dock</str>
            <bool name="inStock">>false</bool>
          </doc>
        </result>
      </lst>
    </arr>
  </lst>
</lst>
```

</lst>

By default, group.limit is 1. That's why there is only one document returned for each group. To group results by query, you can use group.query.

To do grouped faceting, you can set group.facet=true. Grouped facets are computed based on facet.field. If there is more than one group, the first group will be used. For example, to get facet counts of category by unique manufacture ID, you can use:

http://localhost:8983/solr/sample_core/select?q=manu_id_s:

[*%20TO%20*]&fl=name,manu_id_s,cat&group=true&**group.field=manu_id_s**&facet=t

<int name="electronics">8</int>

<int name="currency">4</int>

<int name="graphics card">2</int>

<int name="hard drive">2</int>

<int name="camera">1</int>

For additional parameters on result grouping, please see

<http://wiki.apache.org/solr/FieldCollapsing>.

INDEXING RICH DOCUMENTS

To index rich documents, you can use Solr cell update request handler (the `ExtractingRequestHandler` class). The URL is `/update/extract`. The `ExtractingRequestHandler` class uses Apache Tika to extract text from documents and then index them. Apache Tika (<http://tika.apache.org/>) is a content analysis toolkit that detects and extracts metadata and structured text content using various content format parsers such as HTML, XML, MS Office, or PDF. The `ExtractingRequestHandler` class is inside `solr-cell-5.4.0.jar`. You need to make sure that the following two lines in the `solrconfig.xml` are configured correctly:

```
<lib dir="${solr.install.dir:../../../../}/contrib/extraction/lib" regex="*.jar" />
<lib dir="${solr.install.dir:../../../../}/dist/" regex="solr-cell-.*.jar" />
```

For example, to send Solr a file using `post.jar`:

```
java -Durl=http://localhost:8983/solr/sample_core/update/extract -
Dparams=literal.id=solr_sample -Dtype=text/html -jar post.jar sample.html
```

Or to use `cURL`:

```
curl "http://localhost:8983/solr/sample_core/update/extract?literal.id=
solr_sample&commit=true" -H "Content-type:text/html" --data-binary @sample.html
```

You can create your own field with the specified value using the `literal.<field name>` parameter. Here, `literal.id` creates an `id` field to store a unique document ID since the `id` field is defined as a unique field in the `schema.xml`. The extract text (HTML tags are removed) is added to the `content` field. Based on the `schema.xml`, this field is defined as `index="false" stored="true"`. That's because this field is used for returning and highlighting document content. This field is copied to `text` field using `<copyField>`. The `text` field is defined as `index="true" stored="false"`. That's because the `text` field is used for searching. Using `stored="false"` is to save space. The following is the document:

```
<doc>
  <arr name="links">
    <str>rect</str>
    <str>http://www.apache.org</str>
  </arr>
  <str name="id">solr_sample</str>
  <arr name="title">
    <str>Welcome to Solr</str>
  </arr>
  <arr name="content_type">
```

```
<str>text/html; charset=windows-1252</str>
</arr>
<arr name="content">
<str>Welcome to Solr...</str>
</arr>
<long name="_version_">1524764848508895232</long>
</doc>
```

Now, you can execute the following query to search on the default search field (the text field) and highlight on the content field:

```
http://localhost:8983/solr/sample_core/select?
q=solr&hl=true&hl.fl=content&hl.fragsize=500
```

If you need to store all metadata (e.g., `stream_size`, `stream_content_type`) either produced by Tika or added by Solr, you can use the `uprefix=<prefix>` parameter. For example, `uprefix=attr_` causes all generated fields that are not defined in the schema to be prefixed with `attr_` (`attr_` is a dynamic field defined in the `schema.xml`). The metadata varies on type of document. You can set the `extractOnly` parameter as `true` to see actual values without indexing the document.

For password protected documents, you can provide password through the `resource.password` or `passwordsFile` parameter if filename pattern to password mappings are stored in a file.

To send a PDF document, for example, you can use:

```
java -Durl=http://localhost:8983/solr/sample_core/update/extract "-
Dparams=literal.id=solr_word&uprefix=attr_" -Dtype=application/pdf -jar post.jar solr-
word.pdf
```

You can find that PDF has different metadata as shown below:

```
<doc>
  <arr name="attr_meta">
    <str>meta:save-date</str>
    <str>2008-11-13T13:35:51Z</str>
    ...
  </arr>
  <str name="id">solr_word</str>
  <arr name="attr_meta_save_date">
    <str>2008-11-13T13:35:51Z</str>
  </arr>
```

<arr name="attr_dc_subject">

<str>solr, word, pdf</str>

</arr>

<str name="subject">solr word</str>

...

<arr name="content_type">

<str>application/pdf</str>

</arr>

<arr name="content">

<str>...</str>

</arr>

<long name="_version_">1524766191217803264</long>

</doc>

ACCESSING SOLR PROGRAMMATICALLY

SolrJ is a Java client to access Solr through API calls. There are several JARs used by SolrJ. Under the Solr installation directory, you can find solr-solrj-5.4.0.jar in the dist directory and several JARs in the dist/solrj-lib directory. For API documentation of SolrJ, you can find it under docs/solr-solrj.

Updating data

The following are basic steps to update data in Solr server using SolrJ API:

Step 1

To connect to a Solr server, you need to construct a SolrClient object by using the HttpSolrClient class. The base URL of Solr server is specified in the constructor. The base URL contains the path of Solr Core. It does not include the path of the request handler. You specify that later. The HttpSolrClient class allows setting connection properties such as max retries or connection timeout. Since the HttpSolrClient class is thread-safe, you only need to create one (static) instance per Solr server and reuse it for all requests.

Step 2

Next step is to create documents for posting. You can use the SolrInputDocument class to create Solr documents. Document fields are defined using the addField(String name, Object value) method. The field names and the data type of values should match those defined in the schema.xml.

Step 3

To create an update request, you can use the UpdateRequest class. Here, you can specify the path of an update request handler. You can use the setAction(AbstractUpdateRequest.ACTION action, boolean waitFlush, boolean waitSearcher) method to set parameters for the given action (commit or optimize) and use the add(SolrInputDocument doc) method to add documents for posting. Finally, you use the process(SolrClient client) method to post documents to a Solr server and get an UpdateResponse object as the return value. If you have any additional request parameters, you can use the setParam(String param, String value) or setParams(ModifiableSolrParams params) method.

In the following example, two documents are created and commit to a Solr server:

```
import java.util.ArrayList;
import java.util.Collection;
import java.io.IOException;

import org.apache.solr.common.SolrInputDocument;
import org.apache.solr.client.solrj.SolrClient;
import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrClient;
import org.apache.solr.client.solrj.request.AbstractUpdateRequest;
import org.apache.solr.client.solrj.request.UpdateRequest;
```

```
import org.apache.solr.client.solrj.response.UpdateResponse;

public class SolrPostDocExample {

    public static void main(String[] args) {
        try {
            SolrPostDocExample example = new SolrPostDocExample();
            String coreUrl = "http://localhost:8983/solr/sample_core";
            HttpSolrClient client = new HttpSolrClient(coreUrl);
            // connection timeout defines the time a connection is established
            client.setConnectionTimeout(20000);
            // socket timeout defines the time waiting for a response
            client.setSoTimeout(10000);
            example.postDocuments(client);
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }

    public void postDocuments(SolrClient client)
        throws IOException, SolrServerException {

        // construct documents
        SolrInputDocument doc1 = new SolrInputDocument();
        doc1.addField("id", "1001");
        doc1.addField("name", "Kindle Fire");
        doc1.addField("price", 270);
        doc1.addField("cat", "Electronics");
        doc1.addField("cat", "Tablet");
        SolrInputDocument doc2 = new SolrInputDocument();
        doc2.addField("id", "1002");
        doc2.addField("name", "ASUS VivoBook");
```

```
doc2.addField("price", 485);
doc2.addField("cat", "Electronics");
doc2.addField("cat", "Laptop");
```

```
Collection<SolrInputDocument> docs = new ArrayList<SolrInputDocument>();
docs.add(doc1);
docs.add(doc2);
```

```
// commit and receive response
```

```
UpdateRequest req = new UpdateRequest("/update");
req.setAction(AbstractUpdateRequest.ACTION.COMMIT, false, false);
req.add(docs);
UpdateResponse rsp = req.process(client);
System.out.println(rsp);
}
```

```
}
```

The following is the response:

```
{responseHeader={status=0,QTime=505}}
```

For a multivalued field, you can use the `addField()` method with the same name to add additional values. Or, you can use a `Collection` as the value in the `addField()` method. To implement atomic updating, you can use a `map` and use one of the options as the key. For example,

```
SolrInputDocument doc = new SolrInputDocument();
Map<String, String>atomicUpdate = new HashMap<String, String>();
atomicUpdate.put("set", "200");
doc.addField("id", " SOLR1000");
doc.addField("price", atomicUpdate);
```

To delete documents, you can use `deleteById` or `deleteByQuery` methods.

If you need to upload a file, you can use the `ContentStreamUpdateRequest` class. Both `ContentStreamUpdateRequest` and `UpdateRequest` are subclasses of the `AbstractUpdateRequest` class. A file is added using the `addFile(File file, String contentType)` method. The following sample method is used to post a PDF file:

```
public void postFile(SolrClient client)
    throws IOException, SolrServerException {
```

```
ContentStreamUpdateRequest up
= new ContentStreamUpdateRequest("/update/extract");
up.addFile(new File("curl.pdf"), "application/pdf");

up.setParam("literal.id", "curl");
up.setParam("uprefix", "attr_");
up.setAction(AbstractUpdateRequest.ACTION.COMMIT, false, false);

UpdateResponse rsp = up.process(client);
System.out.println(rsp);
}
```


Querying data

To query data, you can use the SolrQuery class. The SolrQuery class has methods that support functionality in querying data introduced previously (including hit highlighting and faceted searches). The following sample methods are to query documents posted in the sample_core.

Query

```
public void queryDocuments(SolrClient client)
    throws IOException, SolrServerException {

    SolrQuery query = new SolrQuery();
    // the maximum time allowed for this query
    query.setTimeAllowed(8000);
    query.setQuery("cat:electronics");
    query.setFields("name", "price");
    QueryResponse rsp = client.query(query);
    SolrDocumentList docList = rsp.getResults();
    System.out.println("# of documents: " + docList.getNumFound());
    for(SolrDocument doc : docList) {
        System.out.println(doc);
    }
}
```

The following is the response:

```
# of documents: 12
```

```
SolrDocument{name=Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133, price=92.0}
```

```
SolrDocument{name=Maxtor DiamondMax 11 - hard drive - 500 GB - SATA-300, price=350.0}
```

```
SolrDocument{name=Belkin Mobile Power Cord for iPod w/ Dock, price=19.95}
```

```
SolrDocument{name=iPod & iPod Mini USB 2.0 Cable, price=11.5}
```

```
SolrDocument{name=Apple 60 GB iPod with Video Playback Black, price=399.0}
```

```
SolrDocument{name=CORSAIR XMS 2GB (2 x 1GB) 184-Pin DDR SDRAM Unbuffered DDR 400 (PC 3200) Dual Channel Kit System Memory - Retail, price=185.0}
```

```
SolrDocument{name=CORSAIR ValueSelect 1GB 184-Pin DDR SDRAM Unbuffered  
DDR 400 (PC 3200) System Memory - Retail, price=74.99}
```

```
SolrDocument{name=A-DATA V-Series 1GB 184-Pin DDR SDRAM Unbuffered DDR  
400 (PC 3200) System Memory - OEM}
```

```
SolrDocument{name=Canon PIXMA MP500 All-In-One Photo Printer, price=179.99}
```

```
SolrDocument{name=Canon PowerShot SD500, price=329.95}
```

Field faceting

```
public void fieldFacets(SolrClient client)  
    throws IOException, SolrServerException {  
  
    SolrQuery query = new SolrQuery();  
    query.setQuery("*:*");  
    query.setRows(0);  
    query.setFacet(true);  
    query.addFacetField("cat", "inStock");  
    QueryResponse rsp = client.query(query);  
    SolrDocumentList docList = rsp.getResults();  
    System.out.println("# of documents: " + docList.getNumFound());  
    List<FacetField> facetFields = rsp.getFacetFields();  
    for(FacetField facetField : facetFields) {  
        System.out.println(facetField);  
    }  
}
```

The following is the response:

```
# of documents: 34
```

```
cat:[electronics (12), currency (4), memory (3), connector (2), graphics card (2), hard  
drive (2), search (2), software (2), camera (1), copier (1), electronics and computer1 (1),  
electronics and stuff2 (1), multifunction printer (1), music (1), printer (1), scanner (1)]
```

```
inStock:[true (17), false (4)]
```

Result grouping

```
public void groupedByField(SolrClient client)  
    throws IOException, SolrServerException {
```

```

SolrQuery query = new SolrQuery();
query.setQuery("inStock:[* TO *]");
query.addFacetField("name", "inStock");
query.add("group", "true");
query.add("group.ngroups", "true");
query.add("group.field", "inStock");
QueryResponse rsp = client.query(query);
List<GroupCommand> groupCommandList = rsp.getGroupResponse().getValues();
for(GroupCommand groupCommand : groupCommandList) {
    System.out.println("Grouped by: " + groupCommand.getName() + ", # of groups: " +
groupCommand.getNGroups());
    List<Group> groups = groupCommand.getValues();
    for(Group group : groups) {
        SolrDocumentList docList = group.getResult();
        System.out.println("Group: " + group.getGroupValue() + ", # of documents: " +
docList.getNumFound());
        for(SolrDocument doc : docList) {
            System.out.println(doc);
        }
    }
}

```

The following is the response:

Grouped by: inStock, # of groups: 2

Group: true, # of documents: 17

```

SolrDocument{id=GB18030TEST, name=Test with some GB18030 encoded characters,
features=[No accents here, ? 是一 ? 功能 , This is a feature (translated), ? 份文件是很有
光 ?, This document is very shiny (translated)], price=0.0, price_c=0.0,USD,
inStock=true, _version_=1524195539913015296}

```

Group: false, # of documents: 4

```

SolrDocument{id=F8V7067-APL-KIT, name=Belkin Mobile Power Cord for iPod w/
Dock, manu=Belkin, manu_id_s=belkin, cat=[electronics, connector], features=[car power
adapter, white], weight=4.0, price=19.95, price_c=19.95,USD, popularity=1,
inStock=false, store=45.18014,-93.87741, manufacturedate_dt=Mon Aug 01 12:30:25

```

EDT 2005, _version_=1524195540386971648}

Apache James

The Apache James (Java Apache Mail Enterprise Server) is part of the Apache James Project. The James Server is a Java-based mail server (or a mail transfer agent, MTA, technically), which supports SMTP and POP3 protocols. Also, it can serve as an NNTP news server. Something special about James Server is that it provides a mailet container. Just like servlets are used to process HTTP requests for a servlet container. Mailables are used to process emails for a mailet container. Through configurations, you can use mailables to do complex email processing tasks. That is what makes James Server flexible and powerful. There are standard mailables provided by James Server. Also, you can build your own mailables by using the Mailet API which is part of the Apache James Project. This allows you to develop and deploy custom applications to James Server for email processing.

For James Server, the current stable version is 2.3.2 at the time of writing (there is 3.0 beta available, but still not stable). You can download it from <http://james.apache.org/server>. For the Mailet API, the current stable version is 2.5.0 at the time of writing. You can download it from <http://james.apache.org/mailet>. James 2.3.2 is shipped with the Mailet API 2.3.

INSTALLING JAMES SERVER

The binary distribution of James Sever comes with the Phoenix server. James is a mail server running as a Phoenix server application. So, you start the Phoenix server to start the James. This scenario is quite similar with web applications that are shipped with Jetty or Tomcat server. But, the Phoenix server is not a servlet container. Before you start the Phoenix server, you can find james.sar inside the apps directory. It will be unpacked automatically once Phoenix is started. SAR (Service ARchive) files are using the ZIP file format. You can unpack or pack a SAR file using the same method you are using on ZIP file format. The following is the directory structure of a SAR file:

conf

META-INF

SAR-INF

lib

To start James, you can use run.bat or run.sh under the bin directory. Once it is started, you can see the following message:

Phoenix 4.2

James Mail Server 2.3.2

Remote Manager Service started plain:4555

POP3 Service started plain:110

SMTP Service started plain:25

NNTP Service started plain:119

FetchMail Disabled

If the server does not start, you can check phoenix.log under the logs directory. It contains information specific to Phoenix. Do not mix this directory with the logs directory for James, which is under the apps/james/logs directory.

There are many directories under James installation. The following is the directory structure you should know about after james.sar is unpacked:

apps

 james

 conf

 logs

 SAR-INF

var

work

james-xxx

SAR-INF

lib

conf: This directory contains additional configuration files.

logs: This directory contains logs files specific to James.

SAR-INF: This is the main directory you will be working on specially the following two configuration files:

config.xml : This is the configuration file for James.

environment.xml: This configuration file configures behavior on class loading and logging.

var: This directory is the home directory of repositories (used to store mail and news messages, user information).

As you can see, the lib directory inside james.sar does not exist in the james directory after it is unpacked. Instead, it is under the work directory. The work directory is a temporary working directory. It will be removed by the server. So, even after james.sar is unpacked, you cannot remove this file. It is still needed to start James. Those unpacked files will not be replaced even you put in a new version of james.sar unless they are missing. If you want to distribute a custom mail application based on James, it is better to add files on top of deployed James.

CONFIGURING JAMES SERVER

The following are configuration files that you can modify to customize James based on your needs. You need to restart James for the changes to take effect:

config.xml

The most important configuration file for James is config.xml under the SAR-INF directory. Basically, you can break down XML elements in the config.xml into the following categories:

Services

James supports POP3, SMTP and NNTP services. By default, they are enabled. You can configure or disable them from the following XML elements:

<pop3server>: POP3 service

<smtpserver>: SMTP service

<nntpserver> and <nntp-repository>: NNTP news service and NNTP repositories

Also, they can be configured to support TLS (SSL) connections. In that case, you need to set up the “ssl” server socket factory in the <sockets> element.

Spool manager

The spool manager is responsible for processing emails received by James. The spool manager is a maillet container. James is shipped with some standard maillets (and matchers). You define package names that contain maillets or matchers in the following elements:

<mailetpackages>: packages for maillets

<matcherpackages>: packages for matchers

The following are for maillets provided by James:

<mailetpackage>org.apache.james.transport.maillets</mailetpackage>

<mailetpackage>org.apache.james.transport.maillets.smime</mailetpackage>

The following are matchers provided by James:

<matcherpackage>org.apache.james.transport.matchers</matcherpackage>

<matcherpackage>org.apache.james.transport.matchers.smime</matcherpackage>

For custom maillets and matchers, you define package names here too. The spool manager (represented by the <spoolmanager> element) has many processors (represented by the <processor> element) as children. Each processor has a unique name. But, the name “ghost” cannot be used because it is used for messages that are no longer need further processing. Each processor contains zero or more maillets (represented by the <mailet> element). The order of maillets does matter.

There are processors defined in the config.xml already. But, only the root processor and error processor are required. But, you can reuse some of them. James routes all mails on the spool to the root processor first. The spool is a temporary location for incoming mail messages waiting to be processed. The spool repository is defined in the

<spoolrepository> element. By default, it is located at var/mail/spool under the root directory of James application. If there is any error during mail processing, the mail causing error will be redirected to the error processor. By default, the error processor will store mails under var/mail/error.

The spool manager is multithreading. The total number of threads can be defined by the <threads> element.

Repositories

In James, repositories are used to store mail and news messages, and user information. There are three types of repositories based on storage type (defined in the <mailstore> element):

File repository: This is the default storage type. It starts with “file”. The format is: file://<path_to_application_root>. For example, to store data in var/mail/spool under the root directory of James, you use file://var/mail/spool.

Database repository: This is a more efficient way to store data. It starts with “db”. The format is: db://<data source>/<table>/<repository>. The Apache Derby database is shipped with James. But, all major database systems are supported. You can check conf/sqlResources.xml to find supported database systems. The data sources are defined in the <database-connections> element.

DBFile repository: This is a mix of file and database repositories. The body of a mail message is stored in the file system and headers are stored in the database. It starts with “dbfile”. The format is the same as the database repository. The default location for file storage is var/dbmail.

General settings

<James>: defines email address of the postmaster, host names/IP addresses for this instance of James, and the location for user’s inboxes.

<dnserver>: defines a list of DNS servers to be used by James.

You can define connection properties in the following elements:

<connections>: defines the number of milliseconds for an idle client to timeout and the maximum number of connections.

<thread-manager>: defines available thread pools. A thread pool with the name “default” is required.

User accounts

The user repositories are defined in the <users-store> element. The default location is var/users. To manage user accounts, you can do that through the remote manager. The remote manager is defined in the <remotemanager> element. To connect to the remote manager, you can use any tool that can support Telnet protocol such as PuTTY. The default port number is 4555. Login id for the administrator is root and the password is the same. To add a user, use command:

adduser <username> <password>

To remove a user, use command:

deluser <username>

environment.xml

The environment.xml is under the SAR-INF directory. Two elements are defined here. One is classloading and the other is for logging.

By default, a classloader is defined here already. The lib and classes directories under SAR-INF directory are for custom mail application. To create a custom mail application on top of James, you override existing configurations and put JAR or class files under the lib or classes directory. You also can put JDBC driver for the database you are using here.

For logging, there are many log files in James. But, most of them will be empty. You can change logging level on some of them or even change filename pattern to reduce number of log files.

So, which one do you check if there were any errors? Usually, you can look at the following files first:

mailet-*.log: This file logs activities on mailets. Usually, this is the first file you will check.

smtpserver-*.log: This is the log file for SMTP service. If mail messages could not be delivered to the spool manager, you can look at this file to see if there were any errors.

spoolmanager-*.log: This is the log file for the spool manager. It logs information about initialization of matchers and mailets in the spool manager. This is the file to check if James cannot start properly after you made any changes on mailets (or matchers).

If you want to use log4j to replace existing logging framework, just follow the following steps:

Copy log4j JAR file to the lib directory under the Phoenix home directory

Copy log4j.dtd to the SAR-INF directory

Replace <logs version="1.1">...<logs> with the following:

```
<logs version="log4j" xmlns:log4j="http://jakarta.apache.org/log4j/" debug="false">
  <appender name="CONSOLE_LOG" class="org.apache.log4j.ConsoleAppender">
    <param name="Threshold" value="ERROR"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %-5p [%t] %c{1}:%L - %m%n"/>
    </layout>
  </appender>
  <appender name="FILE_LOG" class="org.apache.log4j.RollingFileAppender">
    <param name="File" value="${phoenix.home}/../../logs/james.log"/>
    <param name="MaxBackupIndex" value="5"/>
```

```
<param name="MaxFileSize" value="10240KB"/>
<layout class="org.apache.log4j.PatternLayout">
<param name="ConversionPattern" value="%d %-5p [%t] %c{1}:%L - %m%n"/>
</layout>
</appender>
<logger name="smtpserver">
<level value="INFO"/>
</logger>
<logger name="spoolmanager">
<level value="INFO"/>
</logger>
<logger name="James.Maillet">
<level value="INFO"/>
</logger>
<!-- default logging level -->
<root>
<level value="WARN"/>
<appender-ref ref="FILE_LOG"/>
<appender-ref ref="CONSOLE_LOG" />
</root>
</logs>
```

Everything will be logged to james.log located under the logs directory of the Phoenix home. The default logging level is WARN for all components. Since the smtpserver, spoolmanager and maillet are the most important components, the logging level is INFO.

james-fetchmail.xml

The james-fetchmail.xml is under the conf directory. fetchmail can fetch mails from external mail servers (IMAP or POP3) and inject them to the spool in James for processing. This allows you to use the mail processing capabilities of James on mails from other mail servers.

To set up fetchmail, first you need to enable fetchmail by changing the enabled attribute from false to true:

```
<fetchmail enabled="true">
```

Each fetch task is defined under the <fetch> element. You can have as many fetch tasks as possible only if the name is unique. Basically, fetch task is defined by domain.

Accounts to be fetched are defined under the <accounts> element. Each account is defined under the <account> element. Many accounts are allowed and they will run concurrently. For example, to have mails fetched from two accounts on the same domain:

```
<accounts>
```

```
  <account user="account1@fakedomain.com" password="password1"  
  recipient="tester1@localhost" ignorercpt-header="true"/>
```

```
  <account user="account2@fakedomain.com" password="password2"  
  recipient="tester2@localhost" ignorercpt-header="true"/>
```

```
</accounts>
```

Since the recipient is specified, the ignorercpt-header attribute is set as true.

Now, you can specify the server to fetch mails from:

```
<host>pop.mail.fakedomain.com</host>
```

And, how frequently the server is checked (in milliseconds):

```
<interval>600000</interval>
```

That is the basic settings you need to do to set up fetchmail. But, you can fine tune it by changing other settings. The james-fetchmail.xml is self-documented. Also, you can find some examples under the samples directory.

A QUICK TEST

Now, we will create a simple SMTP client to test the installation of James. This SMTP client is capable of sending out mail messages stored in files. For details on the SMTP spec, you can check <http://tools.ietf.org/html/rfc821> and <http://www.ietf.org/rfc/rfc2821.txt>. Also, for details on the format of mail messages, you can check <http://www.ietf.org/rfc/rfc822.txt> and <http://www.ietf.org/rfc/rfc2822.txt>.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintStream;
import java.io.Reader;
import java.net.Socket;
import java.net.UnknownHostException;

public class SendMessage {
    private static int SMTP_PORT = 25;
    private static int retry = 5;
    private static String EOL = "\r\n";

    public static void main(String[] args) {
        if(args.length < 4) {
            System.out.println("usage: SendMessage <message file> <smtp server> <sender>
<recipient>");
            System.exit(1);
        }
        String messageFile = args[0];
        String smtpServer = args[1];
        String sender = args[2];
        String recipient = args[3];
        try {
```



```
File file = new File(messageFile);
sendMessage(sender, recipient, file, smtpServer);
System.out.println("Message sent: " + file);
} catch(Exception ex) {
System.out.println(ex);
}
}
```

```
public static void sendMessage(String sender, String recipient, File messageFile,
String smtpServer)
throws IOException, UnknownHostException {
Socket socket = null;
BufferedReader socketReader = null;
PrintStream socketWriter = null;
BufferedReader reader = null;
try {
socket = new Socket(smtpServer, SMTP_PORT);
socketReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
socketWriter = new PrintStream(socket.getOutputStream());
int num = 0;
// initiate a conversation with the mail server
while(true) {
num++;
sendMessage(socketWriter, "HELO " + smtpServer);
String response = getResponse(socketReader, 100);
if(response.startsWith("250")) // completed
break;
else if(num == retry) {
throw new IOException("Failed in HELO");
}
}
// read mail content
```

```
reader = new BufferedReader(new FileReader(messageFile));
String line = null;
// sender
sendCommand(socketWriter, "MAIL FROM:<" + sender + ">");
verifyResponse(getResponse(socketReader, 50));
// receiver
sendCommand(socketWriter, "RCPT TO:<" + recipient + ">");
verifyResponse(getResponse(socketReader, 50));
// beginning of mail data
sendCommand(socketWriter, "DATA");
getResponse(socketReader, 50);
while((line = reader.readLine()) != null) {
sendCommand(socketWriter, line);
}
// end of mail data
sendCommand(socketWriter, ".");
getResponse(socketReader, 50);
} finally {
try {
if(reader != null)
reader.close();
if(socketReader != null) {
sendCommand(socketWriter, "QUIT");
socketReader.close();
}
if(socketWriter != null)
socketWriter.close();
if(socket != null)
socket.close();
} catch(IOException ex) {}
}
}
```

```
public static void sendCommand(PrintStream socketWriter, String command)
throws IOException {
    socketWriter.print(command);
    socketWriter.print(EOL);
    socketWriter.flush();
}
```

```
public static String getResponse(BufferedReader socketReader, long delay) throws
IOException {
    try {
        Thread.sleep(delay);
    } catch(Exception e) {}

    return socketReader.readLine();
}
```

```
public static boolean verifyResponse(String response) throws IOException {
    if(response.startsWith("250")) // requested action was completed
        return true;
    else
        throw new IOException(response);
}
}
```

For example, to send a message to user tester1 in James, you can run:

```
Java SendMessage mail.txt dell530 tester@dell530 tester1@dell530
```

The following is the content of mail.txt:

From: tester@dell530

Subject: Test

To: tester1@dell530

This is a test.

Assume tester1 exists. If not, the message will be in var/mail/error. Now, you can check var/mail/inboxes/tester1. This is the inbox for user tester1. There are two files associated with each mail message. For example,

4D61696C313336373839343537383232332D33.Repository.FileObjectStore

4D61696C313336373839343537383232332D33.Repository.FileStreamtStore

One is with object serialization of a MailImpl object (file extension FileObjectStore) and the other is plain text (file extension FileStreamStore).

4D61696C313336373839343537383232332D33, which is part of a mail filename, is an encoded string (with hexadecimal pairs). The decoded string is Mail1367894578223-3 which represents a mail name and a sting key for storage. This is also the name logged in the log files. You can find it in the Message-ID header below.

Now, you can open the file with extension FileStreamStore to check the mail message directly without using a mail client since it is a text file. The following is the content of received message:

Return-Path: <tester@dell530>

Message-ID: <14425860.3.1367894578223.JavaMail.javamailuser@localhost>

MIME-Version: 1.0

Content-Type: text/plain; charset=us-ascii

Content-Transfer-Encoding: 7bit

Delivered-To: tester1@dell530

Received: from 192.168.1.12 ([192.168.1.12])

by Dell530 (JAMES SMTP Server 2.3.2) with SMTP ID 956

for <tester1@dell530>;

Mon, 6 May 2013 22:42:50 -0400 (EDT)

Date: Mon, 6 May 2013 22:42:50 -0400 (EDT)

From: tester@dell530

Subject: Test

To: tester1@dell530

This is a test.

A few additional headers were added by James. If you want to use James to relay messages without using SMTP authorization (as an open relay), you need to add IP address of the client to the following two places:

<mailet match="RemoteAddrNotInNetwork">=...> under the "transport" processor

<authorizedAddresses> under the <smtpserver> element

Or, you will get the following error message:

Rejected message - xxx not authorized to relay to xxx

MAILETS AND MATCHERS

Processors play very important role in James. A processor relies on mailets to process mail messages. The following is the basic structure of a processor:

```
<processor name="processorName">
  <mailet match="MatcherClass=parameter" class="MailletClass">
    <parameter>mailet parameter</parameter>
    ...
  </mailet>
  ...
</processor>
```

For each mailet, there are two attributes. A class representing a mailet is defined in the class attribute. Usually, a mailet needs to meet a certain condition to be executed. The match attribute allows you to specify a matcher that defines the condition a mailet will be executed. The parameter for the matcher is optional (separated by an equal sign). Similarly, a mailet can have parameters too (child nodes of the mailet element).

In the following example, the original root processor is replaced by a very simple one as shown below:

```
<processor name="root">
  <mailet match="SubjectStartsWith=Unsubscribe" class="ToProcessor">
    <processor>unsubscribe</processor>
  </mailet>
  <mailet match="All" class="Null"/>
</processor>

<processor name="unsubscribe">
  <mailet match="All" class="ToRepository">
    <repositoryPath>file://var/mail/unsubscribe</repositoryPath>
  </mailet>
</processor>
```

Mail processing starts from the root processor. To redirect mail processing from one processor to another one, you can use the ToProcessor mailet. In this example, a mail message with the subject starting with "Unsubscribe" will be redirected to the unsubscribe processor. All other mail messages will be ghosted (deleted automatically). The Null mailet is used to indicate the end of mail processing for a mail message. A mail message

needs to reach a destination or to be ghosted. If not, you will get a warning message in the spoolmanager-*.log to inform you that this mail message is deleted automatically.

To store a copy of the mail message in a repository, you can use ToRepository. You do not need to create the unsubscribe directory. It will be created automatically when James is started. By default, the original mail message is deleted automatically after it is copied. You can set the optional parameter passThrough as true to continue processing.

You can run the SMTP client to send the same mail message again. It will not reach the inbox this time because the maillet to send mails to the transport processor is not there. You can modify the subject of the same mail to “Unsubscribe” and send it again. You will find it in the unsubscribe directory.

James is shipped with some maillets and matchers. They are pretty useful and can save you time in coding. For a complete list of maillets, you can check http://james.apache.org/server/2/provided_mailets.html. For a complete list of matchers, you can check http://james.apache.org/server/2/provided_matchers.html.

CREATING A CUSTOM MATCHER

One of special features in James is the Maillet API. It allows us to create a custom mail application through custom matchers and maillets. Even though James provides some maillets and matchers, we still need to create custom ones sometimes.

First, we will discuss about how to create a custom matcher. A matcher is represented by the `Matcher` interface. The life cycle of a matcher involves the following three methods:

`void init(MatcherConfig config)`: This method is called exactly once right after the matcher is instantiated. This method is used to initialize shared resources.

`Collection match(Mail mail)`: All calls to the matcher are handled by this method. A mail message is represented by the `Mail` interface. It returns a `Collection` of recipients (`MailAddress` objects) in the mail message that meet the criteria. If nothing is matched, `null` is returned.

`void destroy()`: This method is called exactly once (after all threads calling this matcher have exited) when a matcher is taken out of service. This method is used to release shared resources.

Since a maillet container is multithreaded, matchers can handle calls concurrently. It is similar with servlets running in a servlet container. You need to make sure the shared resources are thread-safe in the `match` method.

The Maillet API provides two abstract classes (`GenericMatcher` and `GenericRecipientMatcher`) that implement the `Matcher` and `MatcherConfig` interfaces. It is easier to create a custom matcher by extending one of them.

To get the optional parameter of a matcher, you can use the `getCondition()` method. If the parameter does not exist, `null` is returned.

The Maillet API provides logging. But, you can not specify logging level. To log message to the maillet log file, you can use `log(String message)` or `log(String message, Throwable t)`.

GenericMatcher

In the following example, we create a matcher to improve the existing `SubjectStartsWith` matcher. This matcher tries to match the subject using a regular expression and ignoring case-sensitivity. For example, to match a subject starts with the word “Unsubscribe”, you can use:

```
match="HasSubjectRegex=^\bUnsubscribe\b"
```

The `HasSubjectRegex` class extends `GenericMatcher`. The `GenericMatcher` class implements `init` and `destroy` methods. Also, it provides a convenience method `init()` that can be overridden without calling `super.init(config)`.

```
package matcher;
```

```
import java.util.Collection;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
import javax.mail.MessagingException;
```

```
import org.apache.mailer.GenericMatcher;
```

```
import org.apache.mailer.Mail;
```

```
public class HasSubjectRegex extends GenericMatcher {
```

```
    private static Pattern pattern;
```

```
    @Override
```

```
    public void init() throws MessagingException {
```

```
        String condition = getCondition();
```

```
        if(condition == null || condition.trim().length() == 0) {
```

```
            throw new MessagingException("Condition is missing");
```

```
        }
```

```
        pattern = Pattern.compile(condition, Pattern.CASE_INSENSITIVE);
```

```
    }
```

```
    @Override
```

```

public Collection match(Mail mail) throws MessagingException {
    String subject = mail.getMessage().getSubject();
    if(subject != null) {
        subject = subject.trim();
        Matcher matcher = pattern.matcher(subject);
        if(matcher.find())
            return mail.getRecipients();
    }

    return null;
}
}

```

To compile it, you need the following JAR files: mailer-2.3.jar and mailer-api-2.3.jar from the Mailer API, mail-1.4.1.jar from the JavaMail API. You can find them from the work directory.

To deploy this matcher, it involves the following two steps:

Add HasSubjectRegex.class to SAR-INF/classes/matcher

Add <matcherpackage>matcher</matcherpackage> to <matcherpackages>

Now, you can restart James. If James could not start, you can check logs/phoenix.log first to see if there were any errors while loading it. Next file you can check is the spoolmanager-*.log.

GenericRecipientMatcher

The GenericRecipientMatcher class extends GenericMatcher. It overrides the match method to go through each recipient and call the following abstract method:

```
abstract boolean matchRecipient(MailAddress recipient)
```

When a recipient is matched, this recipient is added to a Collection of recipients to be returned. So, instead of overriding match method, the subclass of GenericRecipientMatcher needs to override the matchRecipient method.

Here, the MailAddress class represents an email address specified in the MAIL FROM and RCPT TO commands in a SMTP session. The recipients in the Cc or Bcc header are not included. That is because the Maillet API is built on top of JavaMail API. A Mail object wraps a MimeMessage (from the JavaMail API) with SMTP routing information such as sender and recipients, the current state (which processor it is running at) and the mail session attributes. The MimeMessage is the actual mail message. So, if you need to process message headers or message content, you need to get the MIME (Multipurpose Internet Mail Extensions) message of a Mail. To get the MIME message of a Mail, you can use the getMessage() method and a MimeMessage is returned. The following is a sample MIME multipart message. A boundary is placed between parts and each part contains its own content headers:

From: tester@dell530

Subject: A MIME Message

To: tester1@dell530

MIME-version: 1.0

Content-Type: multipart/mixed; boundary="__boundary__"

—__boundary__

Content-Type: text/plain

The original message was received at Sat, 4 May 2013 15:12:31 -0400

from tester1@dell530

<someone@dell530>... Deferred: Connection timed out
with fakedomain.com.

Message could not be delivered for 5 days

—__boundary__

Content-Type: message/delivery-status

Original-Recipient: someone@dell530

Action: failed

Status: 4.0.0

—__boundary__

Content-Type: message/rfc822

original text

—__boundary__

To access the Mailet API documentation, you can check <http://james.apache.org/server/2/apidocs/index.html>. To access the JavaMail API documentation, you can check <http://javamail.java.net/nonav/docs/api>.

CREATING A CUSTOM MAILET

Creating a custom maillet is similar with creating a custom matcher. A maillet is represented by the Maillet interface. The life cycle of a maillet also involves three methods. Both the init and destroy methods are the same. But, the match method is replaced by the service(Mail mail) method. An abstract class, GenericMaillet, which implements the Maillet and MailletConfig interfaces, can make it easier to create a custom maillet. All you need to do is to override the service method. Also, it provides a convenience method init() that can be overridden without calling super.init(config).

To get the optional parameters of a maillet, you can use the following methods:

```
String getInitParameter(String name)
```

```
String getInitParameter(String name, String defaultValue)
```

```
Iterator getInitParameterNames()
```

When a Mail is being processing, how do you persist certain data in the whole process across maillets? It can be achieved through the mail session attributes. For a Mail, you can use the following methods to manipulate session attributes. The attribute needs to be Serializable:

```
boolean hasAttributes()
```

```
Serializable getAttribute(String name)
```

```
Iterator getAttributeNames()
```

```
Serializable setAttribute(String name, Serializable object)
```

```
void removeAllAttributes()
```

```
Serializable removeAttribute(String name)
```

As we mentioned previously, a Mail object also contains state information. Before a maillet finishes processing a mail, you can choose to change its current state. The following are related methods:

```
String getState()
```

```
void setState(String state)
```

If you do nothing, then it will pass through and move on to next maillet. If the state is Mail.GHOST, it will be discarded.

In the following example, we create a custom maillet which counts number of lines in the message body of a mail. When the number of lines is beyond a threshold specified in the maillet parameter, the mail is discarded. Otherwise, the mail is moved to a file repository. Here, a mail session attribute MESSAGE_SIZE is set in the maillet MessageSorter. And, a matcher HasMailAttributeWithValue provided by James is used to verify attribute value in the next maillet. The following is the modified root processor in the config.xml:

```

<processor name="root">
  <mailet match="All" class="MessageSorter">
    <threshold>25</threshold>
  </mailet>
  <mailet match="HasMailAttributeWithValue=MESSAGE_SIZE,small"
class="ToProcessor">
    <processor>small</processor>
  </mailet>
  <mailet match="All" class="Null"/>
</processor>
<processor name="small">
  <mailet match="All" class="ToRepository">
    <repositoryPath>file://var/mail/small</repositoryPath>
  </mailet>
</processor>

```

For MessageSorter to handle a MIME message, it needs to check if the content is a multipart message or not. But, we need to further check if it is a nested multipart message or not. Once it reaches the bottom of the hierarchy, we can get the input stream of the message body to get the content. We are only interested in the first component since it contains a plain text message body:

```
package mailet;
```

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

import javax.mail.BodyPart;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.internet.MimeMessage;

import org.apache.mailet.GenericMaillet;

```

```
import org.apache.maillet.Mail;
```

```
public class MessageSorter extends GenericMaillet {
```

```
    private static int threshold;
```

```
    @Override
```

```
    public void init() throws MessagingException {
```

```
        String thresholdParam = getInitParameter("threshold");
```

```
        if(thresholdParam == null || thresholdParam.trim().length() == 0) {
```

```
            throw new MessagingException("The threshold parameter is required.");
```

```
        }
```

```
        try {
```

```
            threshold = Integer.parseInt(thresholdParam);
```

```
        } catch(NumberFormatException ex) {
```

```
            throw new MessagingException(ex.getMessage());
```

```
        }
```

```
    }
```

```
    @Override
```

```
    public void service(Mail mail) throws MessagingException {
```

```
        MimeMessage message = mail.getMessage();
```

```
        Object content = null;
```

```
        try {
```

```
            content = message.getContent();
```

```
        } catch(IOException ioex) {
```

```
            throw new MessagingException(ioex.getMessage());
```

```
        }
```

```
        BufferedReader reader = null;
```

```
        try {
```

```

InputStream in = null;
if(content instanceof Multipart) {
// a multipart message
Multipart multiPart = (Multipart)content;
int count = multiPart.getCount();
if(count >= 1) {
// check the 1st component of the multipart
BodyPart body = (BodyPart)multiPart.getBodyPart(0);
Object nestedContent = body.getContent();
if(nestedContent != null && nestedContent instanceof Multipart) {
// for a nested multipart, body.getInputStream()
// returns all parts (including headers) without decoding
multiPart = (Multipart)nestedContent;
count = multiPart.getCount();
if(count >= 1) {
// check the 1st component of the multipart
body = (BodyPart)multiPart.getBodyPart(0);
// get decoded content (no headers)
in = body.getInputStream();
}
} else {
in = body.getInputStream();
}
} else {
// not a multipart message
in = message.getInputStream();
}

// message body only (no headers)
reader = new BufferedReader(new InputStreamReader(in));
String line;

```



```
int lineCount = 0;
mail.setAttribute("MESSAGE_SIZE", "small");
while((line = reader.readLine()) != null) {
lineCount++;
if(lineCount > threshold) {
mail.setAttribute("MESSAGE_SIZE", "large");
break;
}
} catch(IOException ioex) {
throw new MessagingException(ioex.getMessage());
} finally {
try {
reader.close();
} catch(Exception ex) {}
}
}

@Override
public String getMailletInfo() {
return "Message Sorter";
}
}
```

READING AN OBJECT STORE FILE

There are two files associated with each mail in James. A `FileStreamStore` file only contains the mail message. If you need to retrieve data from the original `Mail` object, you still need to access the `FileObjectStore` file. If you need to access a `FileObjectStore` file, it is a little bit tricky since it is a binary file. A `FileObjectStore` file contains a serialized `MailImpl` object. It needs to be de-serialized. You can use the `ObjectInputStream` class to de-serialize a serialized object. Other than that, you also need `mailet-api-2.3.jar`, `james-2.3.2.jar` and `avalon-framework-api-4.3.jar` to compile the code. You can find the first two files from the work directory and the last one from the lib directory. To run the code, you need an additional file, `mail-1.4.1.jar`, from the work directory. You will not be able to retrieve the `MimeMessage` object itself since it is not included in the serialized `MailImpl` object. The following is the code to de-serialize a `FileObjectStore` file:

```
import java.io.BufferedInputStream;
```

```
import java.io.FileInputStream;
```

```
import java.io.ObjectInputStream;
```

```
import org.apache.mailet.Mail;
```

```
import org.apache.james.core.MailImpl;
```

```
public class FileObjectStoreReader {
```

```
    public static void main(String[] args) {
```

```
        if(args.length < 1) {
```

```
            System.out.println("usage: FileObjectStoreReader <file>");
```

```
            System.exit(1);
```

```
        }
```

```
        String filename = args[0];
```

```
        try {
```

```
            // read a file object store
```

```
            ObjectInputStream objectIn = new ObjectInputStream(new BufferedInputStream  
(new FileInputStream(filename)));
```

```
            // print out what's inside the object
```

```
            Mail mail = (MailImpl)objectIn.readObject();
```

```
            System.out.println("Mail name: " + mail.getName());
```

```
System.out.println("State: " + mail.getState());
System.out.println("Sender: " + mail.getSender());
System.out.println("Recipients: " + mail.getRecipients());
objectIn.close();
} catch(Exception ex) {
System.out.println(ex);
}
}
}
```


Jackson

JSON (JavaScript Object Notation) is based on the object notation from the JavaScript programming language. Just like XML, JSON is a format that is used for data storage and data exchange. But, the advantage of JSON is that you can use it in the JavaScript programs easily because a JSON string can be converted to a JavaScript object. A common use case is to use JSON data between back end and front end in web-based applications. Modern browsers have native support on JSON. For example, you can use `JSON.parse` to convert a JSON string to a JavaScript object and `JSON.stringify` to convert a JavaScript object to a JSON string. In Java, you can use Jackson API to convert Java objects to and from JSON. The original purpose of Jackson API was for data binding on JSON data. Now, it also contains packages that can support formats such as XML, CSV. But, this chapter is only focused on JSON.

For Jackson data binding, the current stable version is 2.6.3 at the time of writing. You can download it from <https://github.com/FasterXML/jackson-databind>. Jackson data-binding package depends on Jackson Core (<https://github.com/FasterXML/jackson-core>) and Jackson Annotations (<https://github.com/FasterXML/jackson-annotations>).

POJO MODEL

In JSON, an object is enclosed by curly braces { and }. And, it can contain the following data structures:

- . A collection of name-value pairs. A field name is enclosed by double quotes and is separated with its value by a colon. Each name-value pair is separated by a comma.
- . An ordered list of values starts with a left bracket [and ends with a right bracket]. Values are separated by comma.

A value can be a string, a number, a Boolean, an object, or an array. A string value is quoted by double quotes. A null value is represented by null (without double quotes). A Boolean value is either true or false.

Jackson data binding can convert Java beans (POJOs) to and from JSON. The default rules for property auto-detection are:

- . All public fields
- . All public getters (for serialization)
- . All setters (for deserialization)

The ObjectMapper class, a mapper (or data binder), provides functionality for conversion between Java objects and JSON. Converting POJOs to JSON is called serialization. For example, you can use

```
String writeValueAsString(Object value)
```

to serialize any Java object to a String. To write the output to an output stream, you can use

```
void writeValue(OutputStream out, Object value)
```

Converting JSON to POJOs is called deserialization. For example, you can deserialize a JSON String to a Java object by using

```
<T> T readValue(String content, Class<T> valueType)
```

There are overloaded methods of readValue that provide a variety of input sources, such as InputStream, File, URL, etc.

The following example demonstrates how to use an object mapper to do serialization and deserialization:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

```
public class JsonExample {  
    private int number;
```

```
private String text;
private int[] numbers;

public static void main(String[] args) throws Exception {

    ObjectMapper mapper = new ObjectMapper();

    JsonExample example1 = new JsonExample();
    example1.setNumber(100);
    example1.setText("my text");
    example1.setNumbers(new int[]{1, 2, 3});

    System.out.println(mapper.writeValueAsString(example1));

    String json = "{\n  \"number\":100,\n  \"text\":\n  \"my text\",\n  \"numbers\":[1,2,3]\n}";
    JsonExample example2 = mapper.readValue(json, JsonExample.class);
}

public int getNumber() {
    return number;
}

public void setNumber(int number) {
    this.number = number;
}

public String getText() {
    return text;
}

public void setText(String text) {
    this.text = text;
}
```

```
}
```

```
public int[] getNumbers() {  
    return numbers;  
}
```

```
void setNumbers(int[] numbers) {  
    this.numbers = numbers;  
}
```

```
}
```

The following is the output:

```
{“number”:100,“text”:“my text”,“numbers”:[1,2,3]}
```

You can use a List to replace the array in the code and get the same result.

Based on the auto-detection rules, you need to be careful when naming methods. For example, adding the following public getter:

```
public String getText1() {  
    return text + ” 1”;  
}
```

You will get

```
{“number”:100,“text”:“my text”,“numbers”:[1,2,3],“text1”:“my text 1”}
```

An additional property is added to the output.

JSON PROPERTIES

Property naming and inclusion

By default, a Java field name is used as a JSON property name without any modifications. To indicate a JSON property name for a (non-static) Java field, you can use `@JsonProperty`. Other than that, you also can use `@JsonProperty` to make a (non-static) method as a getter or setter depending on the method signature. This allows you to override the auto-detection rules. In the following example, `@JsonProperty` is used to change a JSON property name, define a getter and a setter:

```
public class Simple1 {

    @JsonProperty("text") // rename
    private String s1;
    private String s2;

    // a default constructor is needed
    public Simple1() {
    }

    public Simple1(String s1, String s2) {
        this.s1 = s1;
        this.s2 = s2;
    }

    public String getS1() {
        return s1;
    }

    public void setS1(String s1) {
        this.s1 = s1;
    }
}
```

```
public String getS2() {  
    return s2;  
}
```

```
public void setS2(String s2) {  
    this.s1 = s2;  
}
```

```
@JsonProperty // define a getter  
private String getExtraProperty() {  
    return s1 + "_extra";  
}
```

```
@JsonProperty // define a setter  
private void setExtraProperty(String extra) {  
    this.s1 = extra;  
}  
}
```

The following object

```
Simple1 s1 = new Simple1("string 1", "string 2");
```

is serialized to

```
{"s2":"string 2","extraProperty":"string 1_extra","text":"string 1"}
```

You might notice a default constructor is defined. A default constructor is needed in deserializing JSON data to construct an object. If a default constructor is not defined, you will get the following error message:

```
Exception in thread "main" com.fasterxml.jackson.databind.JsonMappingException: No suitable constructor found for type [simple type, class Simple1]: can not instantiate from JSON object (missing default constructor or creator, or perhaps need to add/enable type information?)
```

Property exclusion

In JSON serialization, you can use annotation `@JsonIgnore` to exclude Java fields. An annotated field or method (getter) will be excluded from serialization. For example,

```
@JsonIgnore  
private String s1;
```

Or,

```
@JsonIgnore  
public String getS1()
```

You also can use `@JsonIgnore` in JSON deserialization. An annotated field or method (setter) will be excluded from deserialization. Its value will not be changed after being deserialized. For example,

```
@JsonIgnore  
private String s1;
```

Or,

```
@JsonIgnore  
Public void setS1(String s1)
```

Using `@JsonIgnore` to annotate a field can affect both serialization and deserialization.

In deserializing JSON data to a Java object, you can choose to exclude properties that are not defined in the Java class. For example, serializing the following data into an instance of class `Simple1`:

```
{“s2”:“string 2”,“extraProperty”:“extra”,“text”:“string 1”,“extra”:“extra value”}
```

You will get the following error:

```
Exception in thread “main”  
com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException: Unrecognized field  
“extra” (class Simple1), not marked as ignorable (3 known properties:  
“extraProperty”,text”, “s2”])
```

To exclude JSON properties in JSON deserialization, you can use annotation `@JsonIgnoreProperties`, such as:

```
@JsonIgnoreProperties({“extra”})  
public class Simple1 {  
...  
}
```

Also, you can this annotation in JSON serialization.

To exclude all unknown JSON properties, you can use:

```
@JsonIgnoreProperties(ignoreUnknown=true)
```

```
public class Simple1 {
```

```
...
```

```
}
```

Ordering

To override implicit orderings, you can use `@JsonPropertyOrder` to define ordering when serializing properties. For example,

```
@JsonPropertyOrder({"t1", "n1"})
```

Or, to apply alphabetic order, you can use:

```
@JsonPropertyOrder(alphabetic=true)
```

For details on available Jackson annotations, please check <http://fasterxml.github.io/jackson-annotations/javadoc/2.6/>.

MAP

So far, the POJOs contain properties that are predefined and cannot be changed at runtime. To define a POJO with properties that can be changed at runtime dynamically. You can use a Java Map. Jackson can serialize a Java Map too. For example,

```
public Map<String, String> getStringMap() {  
    return stringMap;  
}
```

might be serialized as:

```
{"stringMap":{"str1":"string 1","str2":"string 2"}}
```

One issue here is that an additional level, `stringMap`, is introduced. This does not work properly to use a Map to define dynamic properties of a POJO. To remove an additional level, you can use:

`@JsonAnyGetter`

Such as,

`@JsonAnyGetter`

```
public Map<String, String> getStringMap() {  
    return stringMap;  
}
```

The JSON string might look like:

```
{"str1":"string 1","str2":"string 2"}
```

For deserialization, you can use

`@JsonAnySetter`

Such as,

`@JsonAnySetter`

```
public void setStringMap(Map<String, String> stringMap) {  
    this.stringMap = stringMap;  
}
```

This annotation also can be used to handle any unknown properties.

ENUM

Previously, we have discussed JSON serialization and deserialization for POJO models. What about enum types?

```
public enum OperatorEnum {  
  
    EQUAL("eq"), GREATER("gt"), LESS("lt");  
  
    private String id;  
    OperatorEnum(String id) {  
        this.id = id;  
    }  
    public String getId() {  
        return this.id;  
    }  
}  
  
public class ExpressionTerm {  
    private OperatorEnum operator;  
    private String value;  
    public ExpressionTerm() {  
    }  
    public ExpressionTerm(OperatorEnum operator, String value) {  
        this.operator = operator;  
        this.value = value;  
    }  
    public OperatorEnum getOperator() {  
        return operator;  
    }  
    public void setOperator(OperatorEnum operator) {  
        this.operator = operator;  
    }  
    public String getValue() {  
        return value;  
    }  
}
```

```

    }
    public void setValue(String value) {
        this.value = value;
    }
}

```

The following instance of ExpressionTerm:

```
ExpressionTerm term = new ExpressionTerm(OperatorEnum.EQUAL, "2");
```

is serialized to

```
{"operator": "EQUAL", "value": "2"}
```

For enum types, What if you do not want to use enum name. You can use `@JsonValue` annotation to annotate a method whose returned value is used as the single value in JSON serialization. Only one method in a class can be annotated with this annotation. For example, to use id as the serialized value, you can use:

```

@JsonValue
public String getId() {
    return this.id;
}

```

And, the serialized data becomes:

```
{"operator": "eq", "value": "2"}
```

To deserialize JSON that uses `@JsonValue`, a matching annotation for deserialization is `@JsonCreator`. This annotation can be used in constructors and factory methods to instantiate new instances of annotated class.

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
import com.fasterxml.jackson.annotation.JsonCreator;
```

```
import com.fasterxml.jackson.annotation.JsonValue;
```

```
public enum OperatorEnum {
```

```
    EQUAL("eq"), GREATER("gt"), LESS("lt");
```

```
    private static Map<String, OperatorEnum> lookup = new HashMap<String, OperatorEnum>();
```



```
static {  
    for(OperatorEnum operator : OperatorEnum.values()) {  
        lookup.put(operator.getId(), operator);  
    }  
}
```

```
private String id;  
OperatorEnum(String id) {  
    this.id = id;  
}  
  
@JsonValue  
public String getId() {  
    return this.id;  
}
```

```
@JsonCreator  
public static OperatorEnum getValueById(String id) {  
    return lookup.get(id);  
}
```

```
}
```

Another approach is to define a constructor as a creator, such as:

```
@JsonCreator  
OperatorEnum(String id) {  
    this.id = id;  
}
```

FORMATTING

To change how a property is serialized, you can use `@JsonFormat`. For example, date/time values are written as numeric timestamps. To format a `Date` as a string, you can use the *pattern* element to define a `SimpleDateFormat` pattern and a *timezone* property to define a `TimeZone` format:

```
@JsonFormat(pattern="yyyy-MM-dd",timezone="GMT-05:00")
public Date getDate() {
    return date;
}
```

This feature behaves symmetrically and it will affect deserialization.

Through the *shape* element, a value can be serialized using an alternative JSON data type. For example, POJOs are serialized as JSON objects (`JsonFormat.Shape.OBJECT`) by default, such as:

```
{"col1":"1","col2":"2","col3":"3"}
```

By using

```
@JsonFormat(shape=JsonFormat.Shape.ARRAY)
```

The output becomes:

```
["1","2","3"]
```

POLYMORPHIC TYPES

For polymorphic types, type information should be preserved during serialization and to be used in deserialization. `@JsonTypeInfo` can be used to link an interface (or an abstract class) to its implementations (concrete classes). For example,

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY,
property = "type")
```

The *use* element specifies what kind of type metadata to be used in serialization/deserialization. `JsonTypeInfo.Id` is an enum that defines different identifiers, such as, `JsonTypeInfo.Id.NAME` is for logical name.

The *include* element specifies a mechanism to include type metadata. `JsonTypeInfo.As` is an enum that defines different inclusion mechanisms, such as, `JsonTypeInfo.As.PROPERTY` indicates a property is added automatically to JSON content in serialization. The *property* element is used to define a property name for `JsonTypeInfo.As.PROPERTY` inclusion mechanism.

To associate subtypes to corresponding types, `@JsonSubTypes` is used to register names of subtypes to allow deserializer to find subtypes. For example,

```
@JsonSubTypes({
    @Type(value = Cat.class, name = "Cat"),
    @Type(value = Dog.class, name = "Dog") })
```

The *value* element specifies the class of a subtype. The *name* element specifies a name used as an identifier for the class.

In the following example, `Animal` has two subtypes, `Cat` and `Dog`:

```
import com.fasterxml.jackson.annotation.JsonSubTypes;
import com.fasterxml.jackson.annotation.JsonSubTypes.Type;
import com.fasterxml.jackson.annotation.JsonTypeInfo;
```

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY,
property = "type")
```

```
@JsonSubTypes({
    @Type(value = Cat.class, name = "Cat"),
    @Type(value = Dog.class, name = "Dog") })
```

```
public interface Animal {
    String getName();
}
```

```
public class Cat implements Animal {
```

```
    private String name;
```

```
    @Override
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
    public void setName(String name) {
```

```
        this.name = name;
```

```
    }
```

```
}
```

```
public class Dog implements Animal {
```

```
    private String name;
```

```
    @Override
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
    public void setName(String name) {
```

```
        this.name = name;
```

```
    }
```

```
}
```

A serialized JSON content might look like:

```
{“type”:“Dog”,“name”:“dog 1”}
```

```
{“type”:“Cat”,“name”:“cat 1”}
```

In deserialization, you might get the following error when a subtype is not registered:

```
Exception in thread “main” com.fasterxml.jackson.databind.JsonMappingException:  
Could not resolve type id ‘Cat’ into a subtype of [simple type, class Animal]: known type
```

```
ids = [Animal, Dog]
```

As an alternative to using `@SubTypes` to register subtypes in a parent class, you can use the following method to register classes as subtypes:

```
registerSubtypes(Class<?>... classes)
```

For example, you can remove `@SubTypes` from `Animal` class from previous example and add the following to register its subtypes:

```
ObjectMapper mapper = new ObjectMapper();  
mapper.registerSubtypes(Dog.class, Cat.class);
```

By default, an unqualified class name is used. So, an output might look like:

```
{"type": "Dog", "name": "dog 1"}  
{"type": "Cat", "name": "cat 1"}
```

You can use `@JsonTypeName` to define logical names in subtypes. For example,

```
@JsonTypeName("dog")  
public class Dog implements Animal  
  
@JsonTypeName("cat")  
public class Cat implements Animal
```

This will change the output to:

```
{"type": "dog", "name": "dog 1"}  
{"type": "cat", "name": "cat 1"}
```

To register subtypes with specified logical names, you can use the following method:

```
registerSubtypes(NamedType... types)
```

For example,

```
mapper.registerSubtypes(  
    new NamedType(Dog.class, "doggy"),  
    new NamedType(Cat.class, "kitty")  
);
```

And, this will change the output to:

```
{"type": "doggy", "name": "dog 1"}  
{"type": "kitty", "name": "cat 1"}
```

FILTERING

Views

Previously, we have introduced how to hide fields (or methods) through `@JsonIgnore` annotation. But, `@JsonIgnore` defines properties to be ignored statically. To have the ability to define a set of properties to be ignored dynamically, you can use

`@JsonView`

First, you need to define views. Views are defined as classes. Classes can represent view hierarchies through inheritance. The child views inherit properties from the parent views. For example,

```
public class ProfileJsonViews {  
    static class PublicView { }  
    static class ContactView extends PublicView { }  
}
```

`ProfileJsonViews` defines two views: `Public View` and `ContactView`. `ContactView` inherits `PublicView`. So, `ContactView` includes properties from `PublicView` too. Next, we can apply view definitions on properties, as follows:

```
import com.fasterxml.jackson.annotation.JsonView;
```

```
public class Profile1 {  
  
    @JsonView(ProfileJsonViews.PublicView.class)  
    private String name;  
    @JsonView(ProfileJsonViews.PublicView.class)  
    private String company;  
    @JsonView(ProfileJsonViews.ContactView.class)  
    private String phone;  
    @JsonView(ProfileJsonViews.ContactView.class)  
    private String email;  
  
    public String getName() {  
        return name;  
    }  
}
```

```
}
```

```
public void setName(String name) {  
    this.name = name;  
}
```

```
public String getCompany() {  
    return company;  
}
```

```
public void setCompany(String company) {  
    this.company = company;  
}
```

```
public String getPhone() {  
    return phone;  
}
```

```
public void setPhone(String phone) {  
    this.phone = phone;  
}
```

```
public String getEmail() {  
    return email;  
}
```

```
public void setEmail(String email) {  
    this.email = email;  
}
```

```
}
```

Views are annotated statically in POJOs. But, view can be chosen per call during serialization through `writerWithView()` method. This is a factory method that creates an `ObjectWriter` using specified JSON view.

```

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.MapperFeature;

public class JsonViewExample1 {
    public static void main(String[] args) throws Exception {

        ObjectMapper mapper = new ObjectMapper();
        // all properties without explicit view definition are excluded in serialization
        mapper.configure(MapperFeature.DEFAULT_VIEW_INCLUSION, false);

        Profile1 profile1 = new Profile1();
        profile1.setName("my name");
        profile1.setCompany("my company");
        profile1.setPhone("my phone");
        profile1.setEmail("my email");

        String json = mapper.writeValueAsString(profile1);
        System.out.println(json);

        json =
mapper.writerWithView(ProfileJsonViews.PublicView.class).writeValueAsString(profile1)
        System.out.println(json);

        json =
mapper.writerWithView(ProfileJsonViews.ContactView.class).writeValueAsString(profile1)
        System.out.println(json);
    }
}

```

The following is the output:

```
{“name”:“my name”,“company”:“my company”,“phone”:“my phone”,“email”:“my email”}
```

```
{“name”:“my name”,“company”:“my company”}
```

```
{“name”:“my name”,“company”:“my company”,“phone”:“my phone”,“email”:“my
```


email”}

Filters

To do dynamic filtering completely, you can use

`@JsonFilter`

In the previous example, the class is annotated with `@JsonFilter`, such as:

```
import com.fasterxml.jackson.annotation.JsonFilter;
```

```
@JsonFilter("profileFilter")
```

```
public class Profile2 {
```

```
...
```

```
}
```

Now, you can use `SimpleBeanPropertyFilter` to determine to filter out certain properties dynamically by using a `filterOutAllExcept` or `serializeAllExcept` method. Filters can be chosen per call during serialization through the `writer` method. This is a factory method that creates an `ObjectWriter` using specified `FilterProvider`. The following is an example:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

```
import com.fasterxml.jackson.databind.ser.FilterProvider;
```

```
import com.fasterxml.jackson.databind.ser.impl.SimpleBeanPropertyFilter;
```

```
import com.fasterxml.jackson.databind.ser.impl.SimpleFilterProvider;
```

```
public class FilterExample1 {
```

```
    public static void main(String[] args) throws Exception {
```

```
        ObjectMapper mapper = new ObjectMapper();
```

```
        Profile2 profile1 = new Profile2();
```

```
        profile1.setName("my name");
```

```
        profile1.setCompany("my company");
```

```
        profile1.setPhone("my phone");
```

```
        profile1.setEmail("my email");
```

```
        SimpleBeanPropertyFilter filter1 =  
SimpleBeanPropertyFilter.filterOutAllExcept("name", "company");
```

```
FilterProvider filters1 = new SimpleFilterProvider().addFilter("profileFilter", filter1);  
String json = mapper.writer(filters1).writeValueAsString(profile1);  
System.out.println(json);
```

```
SimpleBeanPropertyFilter filter2 =  
SimpleBeanPropertyFilter.serializeAllExcept("name", "company");  
FilterProvider filters2 = new SimpleFilterProvider().addFilter("profileFilter", filter2);  
json = mapper.writer(filters2).writeValueAsString(profile1);  
System.out.println(json);  
}  
}
```

The following is the output:

```
{"name":"my name","company":"my company"}  
{"phone":"my phone","email":"my email"}
```

CUSTOM SERIALIZERS AND DESERIALIZERS

ObjectMapper uses JsonParser and JsonGenerator for actual reading/writing of JSON content. Both classes are in the Jackson Core. Usually, you do not use them directly unless you need to customize serialization/deserialization.

Custom Serializers

A custom serializer usually extends `StdSerializer`, a subclass of `JsonSerializer`. This is a base class used by all standard serializers. The following method is called to serialize a value entity:

```
void serialize(T value, JsonGenerator jgen, SerializationProvider provider) throws  
IOException
```

You can apply custom serializers to fields, methods, or value classes by using the following annotation:

```
@JsonSerialize
```

Previously, we use `@JsonValue` to serialize an enum type, `OperatorEnum`. In the following example, a custom serializer is created to do serialization:

```
import java.io.IOException;
```

```
import com.fasterxml.jackson.core.JsonGenerator;
```

```
import com.fasterxml.jackson.databind.SerializerProvider;
```

```
import com.fasterxml.jackson.databind.ser.std.StdSerializer;
```

```
public class OperatorSerializer extends StdSerializer<OperatorEnum> {
```

```
    public OperatorSerializer() {  
        super(OperatorEnum.class);  
    }
```

```
    @Override
```

```
        public void serialize(OperatorEnum value, JsonGenerator jgen, SerializerProvider  
provider)
```

```
            throws IOException {
```

```
                // null is handled by a default NullSerializer
```

```
                jgen.writeString(value.getId());
```

```
            }
```

```
        }
```

Now, we can apply this serializer on operator field:

```
@JsonSerialize(using = OperatorSerializer.class)
```

```
private OperatorEnum operator;
```

Custom Deserializers

Similarly, a custom deserializer usually extends `StdDeserializer`, a subclass of `JsonDeserializer`. The following method is called to parse JSON content and a `JsonNode` is returned (a JSON tree model). Now, you can extract information from it. And, construct an instance of value entity (deserialized value) in the end.

T `deserialize(JsonParser jp, DeserializationContext ctxt)` throws `IOException`, `JsonProcessingException`

You can apply custom deserializers to fields, methods, or value classes by using the following annotation:

`@JsonDeserialize`

In the following example, a custom deserializer is created to do deserialization on `OperatorEnum` to replace `@JsonCreator`:

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.deser.std.StdDeserializer;

public class OperatorDeserializer extends StdDeserializer<OperatorEnum> {

    public OperatorDeserializer() {
        super(OperatorEnum.class);
    }

    @Override
    public OperatorEnum deserialize(JsonParser jp, DeserializationContext ctxt)
        throws IOException, JsonProcessingException {
        // current token
        String value = jp.getText();

        return OperatorEnum.getValueById(value);
    }
}
```

```
}
```

Now, we can apply this deserializer on operator field:

```
@JsonDeserialize(using = OperatorDeserializer.class)
```

```
private OperatorEnum operator;
```


CONFIGURATIONS

Instances of `ObjectMapper` are thread-safe. You should try to reuse the same instance of `ObjectMapper` if possible. Before an instance of `ObjectMapper` can be used for any serialization or deserialization calls, a mapper can be configured by the following methods:

```
ObjectMapper configure(MapperFeature f, boolean state)
```

```
ObjectMapper configure(SerializationFeature f, boolean state)
```

```
ObjectMapper configure(DeserializationFeature f, boolean state)
```

`MapperFeature`, `SerializationFeature`, and `DeserializationFeature` are enumerations that define on/off features for `ObjectMapper`.

For example,

```
ObjectMapper mapper = new ObjectMapper();
```

```
// exclude null values
```

```
mapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
```

```
// serialize enum values from toString()
```

```
mapper.configure(SerializationFeature.WRITE_ENUMS_USING_TO_STRING, true);
```

```
// ignore unknown properties
```

```
mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
```

```
// date format
```

```
TimeZone tz = TimeZone.getTimeZone("UTC");
```

```
DateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
```

```
df.setTimeZone(tz);
```

```
mapper.setDateFormat(df);
```


Hibernate Validator

Input validations can happen at different places in applications. Custom and possible duplicate code can be anywhere in the applications. Not to mention they are usually part of logic in the applications. Hibernate Validator is a reference implementation of Bean Validation (<http://beanvalidation.org/>). Bean Validation (added as part of Java EE 6) is a framework that defines a metadata model and API for JavaBeans validation. Constraints on JavaBeans can be expressed via annotations (the default metadata model) and can be extended through XML constraint mappings. Bean Validation 1.1 allows put constraints to the parameters or return values on methods or constructors.

For Hibernate Validator, the current stable version is 5.2.2, at the time of writing. You can download it from <http://hibernate.org/validator/>.

APPLYING CONSTRAINTS

Field-level constraints

Let's start with a simple example to show how to apply constraints to fields defined in a class. Applying constraints on instance fields directly is called field-level constraints. Constraints on static fields are not supported. More than one constraint can apply on the same field and constraints are combined by a logical AND.

Hibernate Validator extends Bean Validation. The built-in constraints include those defined in Bean Validation API (under package `javax.validation.constraints`) and those added to Hibernate Validator API (under package `org.hibernate.validator.constraints`). In the following example, name should not be blank (using `@NotBlank` constraint) and price should not be less than 0 (using `@Min` constraint):

```
import javax.validation.constraints.Min;
import org.hibernate.validator.constraints.NotBlank;
```

```
public class Book {
    @NotBlank
    private String name;
    @Min(value=0)
    private double price;

    public Book() {
    }

    public Book(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }
}
```

```
public void setName(String name) {  
    this.name = name;  
}
```

```
public double getPrice() {  
    return price;  
}
```

```
public void setPrice(double price) {  
    this.price = price;  
}
```

```
}
```

Next, we can do a validation on those constraints via an instance of `Validator` created from a `ValidatorFactory`. Both `ValidatorFactory` and `Validator` are thread-safe. The `validate(T object, Class<?>... groups)` method in the `Validator` is to do validation on all constraints of an object. A set of `ConstraintViolation` objects is returned. If a validation succeeds, an empty set is returned. In the following example, an instance of `Book` that violates all constraints defined in the `Book` class:

```
import java.util.Set;
```

```
import javax.validation.ConstraintViolation;
```

```
import javax.validation.Validation;
```

```
import javax.validation.Validator;
```

```
import javax.validation.ValidatorFactory;
```

```
public class BookExample1 {
```

```
    public static void main(String[] args) {
```

```
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
```

```
        Validator validator = factory.getValidator();
```

```
        Book book1 = new Book(null, -1);
```

```
        Set<ConstraintViolation<Book>> violations = validator.validate(book1);
```

```
        for(ConstraintViolation<Book> violation : violations) {
```

```
            System.out.println(violation.getPropertyPath() + ": " + violation.getMessage());
```

```
}  
}  
}
```

The following is the output:

price: must be greater than or equal to 0

name: may not be empty

Error messages

Each constraint annotation has a default error message. It can be replaced by an optional *message* element. For example,

```
@NotBlank(message="Cannot be blank")
```

An error message can contain additional information through message parameters or message expressions. A message parameter is enclosed in {}. This allows referencing to elements in the annotation. A message expression is enclosed in \${}. This allows using expressions defined in Unified Expression Language (EL), an expression language based on JSP EL. For example,

```
@Min(value=0, message="Invalid value '${validatedValue}'. It must be greater than or equal to {value}.")
```

An error message can be provided from a resource bundle too. All you need to do is to create a file, `ValidationMessages.properties`, and add it to the classpath. For example, `{constraints.price.error}` is a message parameter that is used as a key in the resource bundle:

```
@Min(value=0, message="{constraints.price.error}")
```

And, add an entry in the `ValidationMessages.properties`:

```
constraints.price.error=Invalid value '${validatedValue}'. It must be greater than or equal to {value}.
```

Property-level constraints

To do a validation on a property, you can use the `validateProperty(T object, String propertyName, Class<?>... groups)` method. For example,

```
validator.validateProperty(book1, "price");
```

Similarly, you can apply property-level constraints by annotating getter methods on classes that follows JavaBeans standard. But, do not mix field-level constraints with property-level constraints within a class. This might cause a field to be validated more than once.

VALIDATING PARAMETERS

Validations can be performed to methods or constructors by applying constraints to parameters and return values. In the following example, constraints are added to the add method to make sure a Book object is not null and quantity is at least one:

```
import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public class BookManager {

    public void add(@NotNull Book book, @Min(value=1) int quantity) {
        ...
    }
}
```

To do a validation on parameters, you need to get an instance of ExecutableValidator. For methods, you can use

```
validateParameters(T object, Method method, Object[] parameterValues, Class<?>...
groups)
```

For example,

```
import java.lang.reflect.Method;
import java.util.Set;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.ValidatorFactory;
import javax.validation.executable.ExecutableValidator;

public class BookExample1 {
    public static void main(String[] args) throws Exception {

        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        ExecutableValidator executableValidator = factory.getValidator().forExecutables();
```

```

BookManager manager = new BookManager();
Method method = BookManager.class.getMethod("add", Book.class, int.class);
Book book = new Book("Java", 25);
Object[] params = {book, 0};
Set<ConstraintViolation<BookManager>> violations =
executableValidator.validateParameters(
    manager, method, params);
for(ConstraintViolation<BookManager> violation : violations) {
System.out.println(violation.getPropertyPath() + ": " + violation.getMessage());
}
}
}
}

```

The validation performed in this example is not cascaded. That means when a Book object violates any constraints, it is not going to fail on validation because no validation is performed on a Book. A cascaded validation is to validate contained objects on those annotated with `@Valid`. For example,

```
public void add(@NotNull @Valid Book book, @Min(value=1) int quantity)
```

To validate return values on methods, you can use

```
validateReturnValue(T object, Method method, Object returnValue, Class<?>... groups)
```

For constructors, you can use

```
validateConstructor(Constructor<? extends T> constructor, Object[] parameterValues,
Class<?>... groups)
```

for parameters, and

```
validateConstructorReturnValue(Constructor<? extends T> constructor, T createdObject,
Class<?>... groups)
```

for created object.

INHERITANCE

Constraints are inherited through inheritance. For example, an Item interface is declared as:

```
public interface Item {  
  
    @NotBlank  
    public String getName();  
  
    @Min(value=0)  
    public double getPrice();  
}
```

Now, the Book class implements the Item interface:

```
public class Book implements Item {  
  
    private String name;  
    private double price;  
  
    public Book(String name, double price) {  
        this.name = name;  
        this.price = price;  
    }  
  
    @Override  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @Override
```

```
public double getPrice() {  
    return price;  
}
```

```
public void setPrice(double price) {  
    this.price = price;  
}  
}
```

Run the following code, same validation still applies:

```
Item book1 = new Book(null, -1);
```

```
Set<ConstraintViolation<Item>> violations = validator.validate(book1);
```

For fields, any additional constraints in overriding methods will be validated on top of those defined in the super classes. For example, modify the Book class by adding a property-level constraint, such as:

```
@Min(value=5)  
public double getPrice() {  
    return price;  
}
```

Now, you will get the following output with an additional validation:

price: must be greater than or equal to 5

name: may not be empty

price: must be greater than or equal to 0

Note: Constraints are evaluated in no particular order. You might see the output in different order.

Parameter constraints can be inherited too. For example, a Manager interface is declared as:

```
public interface Manager {  
  
    public void add(@NotNull Item item, @Min(value=1) int quantity);  
}
```

Now, the BookManager class implements the Manager interface, such as:

```
public class BookManager implements Manager {
```

```
@Override
public void add(Item item, int quantity) {
    ...
}
}
```

What if BookManager overrides the add() method with different constraints, such as:

```
public class BookManager implements Manager {
```

```
@Override
public void add(@NotNull @Valid Item item, @Min(value=1) int quantity) {
    ...
}
}
```

You will get the following error message during runtime because this is not allowed:

Exception in thread "main" javax.validation.ConstraintDeclarationException: HV000151: A method overriding another method must not alter the parameter constraint configuration, but method public void BookManager.add(Item,int) changes the configuration of public abstract void Manager.add(Item,int)...

GROUPING CONSTRAINTS

As you can see, all the validation methods introduced earlier take a varargs parameter, `Class<?>... groups`, as the last parameter. When an optional *groups* element is not specified in a constraint, the default group, `javax.validation.groups.Default`, is used. In some cases, only a subset of constraints needs to be validated. This can be done through groups. Each group has to be an interface (a marker interface).

Let's use a process of shopping cart as an example. Part of process is to ask a shopper to sign in as a member or to remain as a guest. For a member, only username and password are needed. For a guest, only address and email are needed. So, validation can be done by breaking constraints into two groups: member and guest:

```
public class UserInfo {  
  
    @NotBlank(groups=MemberGroup.class)  
    private String username;  
    @NotBlank(groups=MemberGroup.class)  
    private byte[] password;  
    @NotBlank(groups=GuestGroup.class)  
    private String address;  
    @NotBlank(groups=GuestGroup.class)  
    private String email;  
  
    ...  
}
```

Groups are defined as follows:

```
public interface MemberGroup {  
}  
  
public interface GuestGroup {  
}
```

The following example is to run validations on the same instance of `UserInfo` in three scenarios: using default group, using guest group, and using member group:

```
public class GroupingExample1 {  
  
    private static Validator validator;
```

```
public static void main(String[] args) {
```

```
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
```

```
    validator = factory.getValidator();
```

```
    UserInfo user1 = new UserInfo(null, null);
```

```
    validate(user1);
```

```
    validate(user1, GuestGroup.class);
```

```
    validate(user1, MemberGroup.class);
```

```
}
```

```
private static <T> void validate(T obj, Class<?>... groups) {
```

```
    Set<ConstraintViolation<T>> violations = validator.validate(obj, groups);
```

```
    if(!violations.isEmpty()) {
```

```
        System.out.println("Violations:");
```

```
        for(ConstraintViolation<T> violation : violations) {
```

```
            System.out.println(violation.getPropertyPath() + ": " + violation.getMessage());
```

```
        }
```

```
    } else {
```

```
        System.out.println("No violations");
```

```
    }
```

```
}
```

```
}
```

The following is the output:

No violations

Violations:

email: may not be empty

address: may not be empty

Violations:

username: may not be empty

password: may not be empty

PROGRAMMATIC CONSTRAINTS

Hibernate Validator also provides API for configuring constraints programmatically. This provides flexibility on changing constraints dynamically at runtime instead of annotating constraints at Java classes.

To configure constraints programmatically, you need to create a new `ConstraintMapping`, a top level class for constraint configuration, for constraint mapping. Then, constraints can be configured on classes.

The following example replaces the example of field-level constraints without using constraint annotations:

```
import java.lang.annotation.ElementType;
import java.util.Set;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;

import org.hibernate.validator.HibernateValidator;
import org.hibernate.validator.HibernateValidatorConfiguration;
import org.hibernate.validator.cfg.ConstraintMapping;
import org.hibernate.validator.cfg.defs.MinDef;
import org.hibernate.validator.cfg.defs.NotBlankDef;

public class BookExample1 {
    public static void main(String[] args) {

        HibernateValidatorConfiguration configuration = Validation
            .byProvider(HibernateValidator.class)
            .configure();
        ConstraintMapping constraintMapping = configuration.createConstraintMapping();
        constraintMapping.type(Book.class)
            .property("name", ElementType.FIELD)
            .constraint(new NotBlankDef())
            .property("price", ElementType.FIELD)
```



```
.constraint(new MinDef().value(0));
```

```
Validator validator = configuration.addMapping(constraintMapping)
```

```
.buildValidatorFactory()
```

```
.getValidator();
```

```
Book book1 = new Book(null, -1);
```

```
Set<ConstraintViolation<Book>> violations = validator.validate(book1);
```

```
for(ConstraintViolation<Book> violation : violations) {
```

```
System.out.println(violation.getPropertyPath() + ": " + violation.getMessage());
```

```
}
```

```
}
```

```
}
```

CREATING A CUSTOM CONSTRAINT

To create a custom constraint using an annotation, an annotation type needs to be declared first. An annotation type is declared with the `@Interface` keyword. Several predefined Java annotation types can be included in other annotation types, such as:

`@Target` restricts what kind of Java elements the annotation can be applied to, such as fields, methods, or parameters. For example, `ElementType.FIELD` indicates the annotation can be applied to a field or property.

`@Retention` specifies how the annotation is retained, such as source level, compile time, or runtime. For example, `RetentionPolicy.RUNTIME` indicates the annotation can be used at runtime.

`@Documented` indicates the annotation is included in the Java doc. By default, this is not true.

`@Inherited` indicates the annotation type can be inherited from the super class. By default, this is not true.

All elements of an annotation are declared similar to a method. Optional default values can be provided through the `default` keyword.

The following example is to add a new constraint annotation that can validate the code field in the `Book` class:

```
public class Book {  
    @NotBlank  
    private String name;  
    @Min(value=0)  
    private double price;  
    @NotBlank  
    @CodeConstraint(prefixList={"A-", "B-"})  
    private String code;  
    ...  
}
```

Creating a constraint annotation

To make an annotation type as a constraint annotation, you need to use `@Constraint`. Also, any constraint annotation needs to provide three required elements: `message`, `groups`, and `payload`. `@CodeConstraint` has an additional element, `prefixList`.

```
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 javax.validation.Constraint;
import javax.validation.Payload;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Constraint(validatedBy=CodeConstraintValidator.class)
public @interface CodeConstraint {
    String message() default "Invalid code";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    String[] prefixList();
}
```

The purpose of *message* element is to provide an error message when a validation fails. Here, a default message is provided. An error message can be provided from a resource bundle too. All you need to do is to create a file, `ValidationMessages.properties`, and add it to the classpath. For example,

```
String message() default "{constraints.code.error}";
```

And, add an entry in the `ValidationMessages.properties`:

constraints.code.error=Invalid code

Creating an validator

A constraint validator needs to implement the following interface:

```
ConstraintValidator<A extends Annotation, T>
```

The first parameter is the constraint annotation. The second parameter is the data type of object to be validated. `ConstraintValidator` defines two methods:

`void initialize(A annotation)`: This method is called to initialize the validator before the `isValid` method is called.

`boolean isValid(T value, ConstraintValidatorContext context)`: This contains the actual logic of validation. This method can be accessed concurrently. You need to make sure it is thread-safe.

```
import javax.validation.ConstraintValidator;
```

```
import javax.validation.ConstraintValidatorContext;
```

```
public class CodeConstraintValidator implements
```

```
    ConstraintValidator<CodeConstraint, String> {
```

```
    private String[] prefixList;
```

```
    @Override
```

```
    public void initialize(CodeConstraint annotation) {
```

```
        this.prefixList = annotation.prefixList();
```

```
    }
```

```
    @Override
```

```
    public boolean isValid(String object, ConstraintValidatorContext context) {
```

```
        if(object == null) {
```

```
            return false;
```

```
        }
```

```
        boolean flag = false;
```

```
        for(String prefix : prefixList) {
```

```
            if(object.startsWith(prefix)) {
```

```
                flag = true;
```

```
break;
```

```
}
```

```
}
```

```
return flag;
```

```
}
```

```
}
```