GROOVY PROGRAMMING COOKBOOK

Hot Recipes for Groovy Development

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Groovy Programming Cookbook

Contents

Copyright (c) Exelixis Media P.C., 2016

All rights reserved. Without limiting the rights under
copyright reserved above, no part of this publication
may be reproduced, stored or introduced into a retrieval system, or
transmitted, in any form or by any means (electronic, mechanical,
photocopying, recording or otherwise), without the prior written
permission of the copyright owner.

Preface

Apache Groovy is an object-oriented programming language for the Java platform. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. It can be used as a scripting language for the Java Platform, is dynamically compiled to Java Virtual Machine (JVM) bytecode, and interoperates with other Java code and libraries. Groovy uses a Java-like curly-bracket syntax. Most Java code is also syntactically valid Groovy, although semantics may be different.

Groovy 1.0 was released on January 2, 2007, and Groovy 2.0 in July, 2012. Since version 2, Groovy can also be compiled statically, offering type inference and performance very close to that of Java. Groovy 2.4 was the last major release under Pivotal Software's sponsorship which ended in March 2015. Groovy has since changed its governance structure to a Project Management Committee (PMC) in the Apache Software Foundation. (Source: https://bit.ly/2bIoaO4)

In this ebook, we provide a compilation of GWT examples that will help you kick-start your own projects. We cover a wide range of topics, from sample applications and interview questions, to Callback functionality and various widgets. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at https://www.javacodegeeks.com/

Chapter 1

Groovy Script Tutorial for Beginners

In this article we will see how easy it is to write scripting in `Groovy`, with a practical example that serves a common use case. As you would probably know, Groovy is a JVM Language which is defined to run on top of Java Virtual Machine. Having said that it borrows all Java's capabilities alongside providing its own extra features to simplify the tasks of the developers to achieve things in a easy and efficient manner.

Pre-requisite: The readers are expected to have a good exposure on Java programming language and the basic SQL (like table creation, inserting and reading values from a table etc.,) alongside a beginner level introduction to the Groovy at least to start off with. However we will cover the required aspects of the elementary semantics of Groovy, wherever required during the flow. You can read a good set of introductory tutorials of Groovy here.

1.1 Environment

The examples shown below are executed in the *Command Prompt / Shell* and the *GroovyConsole*. But they are guaranteed to work with any of the IDEs (like *Eclipse, IntelliJ IDEA, Netbeans*) through the respective plugins. The examples are executed with the Groovy Version **Groovy Version: 2.4.3**.

For our example Order Processing System, we will use `MySQL` open source database. We use **MySQL V 5.6 for Windows**, with **JDBC driver for MySQL - MySQL JDBC Connector 5.1.29**.

1.2 Scripting Basics

Script is an executable program which is accomplished to automate the mundane tasks in a simple and easy manner that requires a minimal or no human intervention thereby resulting in time saving. However there are various scripting languges and tools with their owns pros and cons depending on the features they have to offer, the simplified syntax they have in store, the limitations they have by nature etc.,

In general, the scripting languages have one thing in common - their simplified syntax and ease of use as that is the main attracting point to the users (or developers) who aims to get the job done quickly and efficiently.

Groovy is a programming language which can be used as a Scripting language due to its very simplified syntax which is very easy to use. If you are an experienced Java professional for more than 5 years and have explored Groovy for sometime, you would undoubtedly agree with the above statement.

1.3 Example - Order Processing

We will see an example of an Order Processing System where we will get the details related to an order in a flat file (a pipe separated / delimited) file where each line contains the information pertaining to an order. We will parse (read and process) the line entries from the file and add an Order for each line into the Database to store it permanently.

For the sake of our example, imagine a system that dumps such a flat file in our file system at regular intervals. Our Groovy Script will continuously monitor the same directory in the file system at regular intervals for the new files. Upon finding the files, it will process them and move the files into a different directory, so as not to process them in the next iteration.

We will be using Groovy language basics for covering this example like String Concatenation, Collections - List and Map, File Handling, SQL Handling etc., You may please check the References section at the bottom of this article to have a quick glance on each of the topics to facilitate your understanding of scripts.

1.4 Planning and approach

For our order processing script, we will do the following in order as a step by step approach.

- **Business Model (Domain Object)** - We will create a Domain Object (Business Object) to carry all the information of an order as a single entity, otherwise it will be tedious and an overhead if we were to pass all the individual attributes of an order. It is a common practice to wrap all the relevant attribute into an Object of a corresponding class. In Java, it is called as POJO (Plain Old Java Object) where in Groovy it is called as POGO (Plain Old Groovy Object).
- **Application Business Scripts** - We will write a few simple Groovy scripts for creating random Order entries and write them into a delimited file (pipe separated) in a directory so that our Database scripts will be able to parse (read and process) them for inserting them into Database.
- **Database Scripts** - We will write a few simple Groovy Scripts to interact with the database for creating a Database Table, reading the records from the table, truncating the table (wiping off the records), dropping a table etc., with a help of a JDBC Driver (MySQL JDBC Connector).

1.5 Groovy Scripts for Business Objects

In this section, we will write a few simple Groovy Scripts pertaining to the business purpose.

1.5.1 Order POGO

This is a very essential script that carries the class structure for a POGO (Plain Old Groovy Object) to have the essential properties of an Order - like the Order Id, Date, Customer Name, Currency, Price, Country, Payment Mode etc.,

We need to carry all the properties of an Order in a single entity - which is a Wrapper Object. It is also called as BO (Business Object) or a DTO (Data Transfer Object) - due to its nature of transferring the data from one layer to another within an application.

We will need to create as many as objects based on the number of order entries. The ratio between the order entry in a flat file Vs the Order POGO instance will be `1:1`.

The following Groovy scripts has different member variables for each of the Order attributes/properties plus few methods to be used for a common usage - like `toString()` to print the state of an object in a meaningful manner, `initOrder()` - a method to initialize an object with a line of text being read from the text file etc.,

Please remember the fact that Groovy does not really mandate the presence of a semicolon at the end of every executable statement as in Java. This is one of the many significant ease of uses of Groovy.

All the scripts in this article will be under a package `com.javacodegeeks.example.groovy.scripting` which helps us to organize all the related classes and scripts in a same folder/directory (which is called as a Package in Java's term).

We will compile this Groovy class using `groovyc Order.groovy` so that the compiled .class file gets generated in a folder matching with `com/javacodegeeks/example/groovy/scripting` in the present directory. It will be referred by other scripts to load the Order class.

Order.groovy

```
// package statement should be the very first executable statement in any script
// packaging helps us to organize the classes into a related folder
package com.javacodegeeks.example.groovy.scripting

/**
  * A POGO (Plain Old Groovy Object) used as a DTO (Data Transfer Object)
  * or a BO (Business Object) which will carry the essential particulars
  * of an Order
  */
class Order
{
    /** Self explanining fields and the default values for the few */
    //unique value from Database to denote a particular order
    def id = 0
    def customerName, country='India', price
    def currency='INR', paymentMode='Cash'
    def orderDate // will be the current time in DB during insert

    /**
     * A method to format the value with the specified pattern (format),
     * useful in having a fixed width while displaying in Console
     */
    def String format(format, value)
    {
        String.format(format, value)
    }

    /**
     * A method to parse a line of text and initialize an Order instance
     * with the parsed tokens
     */
    def initOrder(strAsCSV)
    {
        // tokenize() will split the line of text into tokens, and trimmed to
        // remove the extra spaces if any
        // the tokens are assigned to the variables in order - as per
        // the declaration on the left side

        def(nameParsed, countryParsed, priceParsed, currencyParsed, modeParsed) = strAsCSV. ↩
            tokenize("|")*.trim()

        this.customerName=nameParsed
        this.country=countryParsed
        this.price=priceParsed
        this.currency=currencyParsed
        this.paymentMode=modeParsed

        return this
    }

    /**
      * An overridden toString() method of this Order class to have a
      * meaningful display wherever it is invoked (which otherwise
      * will be a typical hashcode and the classname)
      */
    def String toString()
    {
        //preparing the output in a fixed width format via format() method defined above
        def output = String.format("%-15s", customerName) + " | "   +
            String.format("%-10s", country) + " | " + String.format("%-8s", price) + " | "  ↩
                +
```

```
            String.format("%-8s", currency) + " | " + String.format("%-12s", paymentMode) + ↩
                " | "

        // add the OrderDate in the output Data if is not null
        if(orderDate!=null && orderDate.trim().length()>0)
            output += String.format("%-20s", orderDate) + " | "

        // add the Id if it is valid (not zero - meaning it was inserted in the database)
        if(id!=0)
            output = String.format("%-5s", id) + " | " + output

        output
      }
}
```

1.5.2 First Level Unit Test

Let us just quickly test the `Order.groovy` to ensure that this class behaves as expected. This will be very important to complete one module perfectly so that we can be comfortable in proceeding with the rest of the scripts. This is the best and recommended practice for the correctness of the script execution.

This script will have 3 different Order instances created and printed on the screen. The script has a self explaining comment at every other step.

TestOrder.groovy

```
package com.javacodegeeks.example.groovy.scripting

/**
 *  A straight forward way to instantiate and initialize an Order instance
 *  Field name and value is separated by a colon (:) and
 *  each name value pair is separated by a commma (,)
 */
def order = new Order(customerName: 'Raghavan', country : 'India', id : '1',
        price : '2351', currency: 'INR', paymentMode : 'Card')
println order

/**
  * Another approach to initialize the Order instance where
  * one property or field is intialized during instantiation (invoking the constructor)
  * and rest of the properties are initialized separately
  */
def order2 = new Order(customerName: 'Kannan')
order2.country='Hong Kong'
order2.id='2'
order2.price='1121'
order2.currency='HKD'
println order2

/**
 * Yet another way to initialize the Order instance with one complete line of formatted ↩
    line
 * pipe separated values ("|")
 */
def lineAsCSV = "Ganesh      | Hong Kong  | 1542.99  | HKD    | Cheque       |"
def order3 = new Order().initOrder(lineAsCSV)
println order3
```

The script produces the following output.

```
1     | Raghavan         | India      | 2351     | INR      | Card         |
2     | Kannan           | Hong Kong  | 1121     | HKD      | Cash         |
Ganesh          | Hong Kong  | 1542.99  | HKD      | Cheque       |
```

The output in each line consists of the different pieces of an Order each separated by a pipe character. The output being displayed is generated out of the `toString()` method of the `Order.groovy` class. The fields displayed are *Order Id (if not null), Customer Name, Country, Amount, Currency, Payment Mode*

The first two instances are having the order Id because it was given during the instantiation. However the 3rd order instance did NOT have the Order Id and hence the output is slightly different from the first 2 instances. The 3rd instance is very important because the actual lines in the flat file will resemble the same as we will not have the order Id at first. It will be created after inserting the record into database, as it should be typically an auto-generated Id.

1.5.3 Test Order - Random Entries Onetime

Now that we have made our Business Object - Order.groovy and we had also tested that successfully, we will proceed further with the next step. We will create some random Order instances and write them into a text file in a delimited - pipe separated format.

This class have all few set of names, country and currencies, amounts and payment modes in separate data structures (List and Map) - using which we will pickup some values at random to constitute an Order instance. We will generate a random number to be used as an index (pointer) to the corresponding data structure.

This script will generate a set of entries based on a max limit that is configured in the script itself - `noOfRecords`, which can be passed as a command line argument if needed. Otherwise this will have a default value of 5 - meaning 5 entries will be generated and stored in a text file.

TestOrderRandomBulk.groovy

```
package com.javacodegeeks.example.groovy.scripting

// a list of values for customer names
def namesList = ['Raghavan', 'Ganesh', 'Karthick', 'Sathish', 'Kanna', 'Raja',
         'Karthiga', 'Manoj', 'Saravanan',  'Adi', 'Dharma', 'Jhanani',
         'Ulags', 'Prashant', 'Anand']

// a Map to hold a set of countries and the currency for each
def countryCurrencyMap = ['India':'INR', 'Australia' : 'AUD', 'Bahrain' : 'BHD',
              'Dubai' : 'AED', 'Saudi' : 'SAR', 'Belgium' : 'EUR',
              'Canada' : 'CAD', 'Hong Kong' : 'HKD', 'USA' : 'USD',
              'UK' : 'GBP', 'Singapore' : 'SGD']

// a list to hold random amount values
def amountList = ['1234.00', '1542.99', '111.2', '18920.11', '4567.00', '9877.12',
      '2500.00', '50000.00', '6500.18', '7894.11', '1234.56', '1756.55', '8998.11']

// a list to hold different payment modes
def paymentModeList = ['Cash', 'Card', 'Cheque']

int noOfRecords = 5

println " "

// Take a default value of 5 for the total number of entries to be printed
// if nothing was specified in the command prompt while invoking the script
if(args.length>0) {
   noOfRecords = Integer.parseInt(args[0])
} else {
   println "Considering the default value for total # of entries"
}
```

```
println "No of entries to be printed : " + noOfRecords

def random = new Random()
def randomIntName, randomIntCountry, randomIntAmount, randomIntMode
def orderRandom, mapKey

def outputToWrite = ""

for(int i = 0; i
    writer.writeLine outputToWrite
}

println ""
println "Contents are written to the file -> [$fileName] in the directory [$baseDir]"
println ""

// 'line' is an implicit variable provided by Groovy
new File(baseDir, fileName).eachLine {
        line -> println line
}

println "********** "
println " == COMPLETED ==  "
println "********** "
```

The script produces the following output.

```
Considering the default value for total # of entries
No of entries to be printed : 5

Contents are written to the file -> [outputTest.txt] in the directory [outputFiles]

Manoj           | USA         | 9877.12  | USD      | Cheque       |
Jhanani         | Canada      | 4567.00  | CAD      | Cash         |
Kanna           | Singapore   | 9877.12  | SGD      | Card         |
Karthiga        | Saudi       | 9877.12  | SAR      | Cash         |
Saravanan       | Australia   | 8998.11  | AUD      | Cheque       |

**********
 == COMPLETED ==
**********
```

As you see, the script produced 5 different Order Instances with random values picked up from the respective data structure and stored each Order instance as a pipe separated values. Each order instance was accumulated and all the lines were stored in a text file - `outputTest.txt` in the directory `outputFiles`.

At the end of the script, we had read the same file `outputTest.text` and printed the contents of the file into screen - for our quick and easy verification.

1.5.4 Test Order - Random Entries at Intervals

We saw how to write a script to write a set of Order Instances into a text file. We will see how to generate a similar set of random entries at regular intervals by making the script sleep for a while in between. This is to simulate a real time environment where one system will be dumping the flat files at regular intervals in a shared location where the other system will be watching the files for processing further.

For the sake of brevity, we will make our script execute 2 iterations where each iteration will generate 5 random Order instances and store them in a separate text file whose names will have the timestamp appended for distinguishing with each other.

The script will generate the files in `outputTest.txt_count_<yyyyMMdd_HHmmss>` pattern. We have an utility method `getNow()` to give the present date and time in the prescribed format, which we will call everytime while writing to a new file. The output file will be written in the same directory, which will be processed by a next database script which, after processing will move the file into a different target directory to avoid these files being reprocessed in the subsequent iterations.

As usual the script below has a good documentation at every step that will be self explanatory for the readers to follow along.

TestOrderRandomInterval.groovy

```
package com.javacodegeeks.example.groovy.scripting

// a list of values for customer names
def namesList = ['Raghavan', 'Ganesh', 'Karthick', 'Sathish', 'Kanna', 'Raja',
                 'Karthiga', 'Manoj', 'Saravanan',  'Adi', 'Dharma', 'Jhanani',
                 'Ulags', 'Prashant', 'Anand']

// a Map to hold a set of countries and the currency for each
def countryCurrencyMap = ['India':'INR', 'Australia' : 'AUD', 'Bahrain' : 'BHD',
                          'Dubai' : 'AED', 'Saudi' : 'SAR', 'Belgium' : 'EUR',
                          'Canada' : 'CAD', 'Hong Kong' : 'HKD', 'USA' : 'USD',
                          'UK' : 'GBP', 'Singapore' : 'SGD']

// a list to hold random amount values
def amountList = ['1234.00', '1542.99', '111.2', '18920.11', '4567.00', '9877.12',
          '2500.00', '50000.00', '6500.18', '7894.11', '1234.56', '1756.55', '8998.11']

// a list to hold different payment modes
def paymentModeList = ['Cash', 'Card', 'Cheque']

def getNow() {
    new Date().format("yyyyMMdd_HHmmss")
}

// default value if nothing is specified during runtime at the command prompt
int noOfRecords = 5

println ""

// Take a default value of 5 for the total number of entries to be printed
// if nothing was specified in the command prompt while invoking the script
if(args.length>0) {
   noOfRecords = Integer.parseInt(args[0])
} else {
   println "Considering the default value for total # of entries : 10"
}

println "Total # of records to be printed : $noOfRecords"

def random = new Random()
def randomIntName, randomIntCountry, randomIntAmount, randomIntMode
def orderRandom, mapKey

// a variable to hold the output content to be written into a file
def outputToWrite = ""

//no of iterations this script will execute
def MAX_ITERATIONS = 2;

// interval to let this script sleep (1000ms=1s, here it will be for 1 min)
def SLEEP_INTERVAL = 1000*60*1;

// Directory to which the output files to be written
def baseDir = "."
```

```
// Prefix of the output file
def fileNamePrefix = 'outputTest.txt'
// Name of the output file which will vary in the loop below
def fileName = fileNamePrefix

println " "
println "[*] This script will write $noOfRecords records for $MAX_ITERATIONS iterations"
println " "

for(int iteration = 1; iteration <= MAX_ITERATIONS; iteration++)
{
    /* Empty the buffer to avoid the previous stuff to be appended over and over */
    outputToWrite = ""

    for(int i = 0; i
        writer.writeLine outputToWrite
    }

    println getNow() + " : Contents are written to the file -> [$fileName] in the directory ↩
         [$baseDir]"
    println ""

    // 'line' is an implicit variable provided by Groovy
    new File(baseDir, fileName).eachLine {
            line -> println line
    }

    println getNow() + " ...... Script will sleep for ${SLEEP_INTERVAL} milliseconds [1000 ↩
        ms=1s]......"
    println " "
    sleep(SLEEP_INTERVAL)
    println getNow() + " .............. Script awaken ......................"
    println ""
}

println " "
println "*********** "
println " == COMPLETED ==  "
println "*********** "
```

The script produces the following output.

```
Considering the default value for total # of entries : 5
Total # of records to be printed : 5

[*] This script will write 5 records for 2 iterations

20160415_122040 : Contents are written to the file -> [outputTest.txt_1_20160415_122040] in ↩
     the directory [.]

Kanna           | UK         | 8998.11  | GBP      | Card         |
Manoj           | Australia  | 9877.12  | AUD      | Cash         |
Prashant        | Saudi      | 9877.12  | SAR      | Card         |
Dharma          | Australia  | 1756.55  | AUD      | Card         |
Raja            | Belgium    | 18920.11 | EUR      | Cheque       |

20160415_122040 ...... Script will sleep for 60000 milliseconds [1000ms=1s]......

20160415_122140 .............. Script awaken ......................

20160415_122140 : Contents are written to the file -> [outputTest.txt_2_20160415_122140] in ↩
     the directory [.]
```

```
Prashant        | Canada     | 111.2    | CAD      | Cash         |
Dharma          | UK         | 50000.00 | GBP      | Card         |
Kanna           | Belgium    | 50000.00 | EUR      | Card         |
Saravanan       | Hong Kong  | 9877.12  | HKD      | Cheque       |
Manoj           | Singapore  | 8998.11  | SGD      | Cash         |

20160415_122140 ...... Script will sleep for 60000 milliseconds [1000ms=1s]......

20160415_122240 .............. Script awaken ......................


***********
 == COMPLETED ==
***********
```

As explained above, the script had produced two different flat (text) files with the pattern outputTest.txt_ with a sequence number along with the date and time pattern to distinguish the files from each other. We had also retrieved the file contents and displayed in the console for our quick verification.

We will see in the next subsection how to process these order entries for storing them into the database.

1.6 Groovy Scripts for Database

We have completed our scripts for producing the flat files to cater to the business needs. Now we need a place to store the data values into a permanent storage - a Database. Like how we created a main POGO `Order.groovy` to use it as a Business Object for our business oriented scripts, we need a database table to store the specific attributes of an Order, which were carried in an instance of Order class.

We will see in the subsequent sub-sections how we will write the simple groovy scripts to create a database table, insert or store values into the table, retrieve the values from the table, truncate (or wipe off the entries) from the table for us to clean and start over again, drop the database etc.,

Please ensure you have the **MySQL** database installed in your machine and you have the **MySQL JDBC Connector Jar file** before you proceed with the rest of the scripts in this section.

You may read a suitable tutorial or the MySQL website for installing the database in to your machine.

1.6.1 Configuration file to hold the database properties

A database is a different entity and it needs to be connected to before we could actually do any operation with it. After finising our operation we should close the connection as a best practice so that the database can support further clients and connections in a better manner.

We need certain important properties about the database like the Server IP Address or hostname where the database is running, the port number to which the database service is listening for client requests, the database type - like Oracle, DB2, MySQL etc., , the actual name of the database we need to connect to, the JDBC (Java Database Connectivity) driver url using which the underlying Java based Programming Language will talk to the database.

As we will have a few different scripts dealing with the database and each of them will need the same properties, it is better to capture them in a separate file and be it referred in all the relevant scripts whichever needs those properties. The advantage of doing this practice is if at all we need to make a change in any of the properties, we can do so in a single place rather than changing in all the files where it was referred individually.

Before we could write the separate database related scripts, we should see the configuration file (or a properties file in Java terminology) as to how we can store the values. The below `db.properties` stores the properties in a `key,value` pair where each pair is entered in a separate line and each key value pair is stored in the format `key=value`, where `key` is the actual key we will be referring to pick up the value from the file and the `value` is the actual value of the property.

The configuration file can be of any extension but it is a standard practice to have it stored in the meaningful `.properties` extension.

The lines starting with a `#` will be treated as a comment and will be ignored. It is more like a `//` in a Java like Programming language to indicate a meaningful description or a comment.

db.properties

```
# Properties needed to construct a JDBC URL programmatically inside a script
db.type=mysql
db.host=localhost
db.port=3306
db.name=test

# Username and password for authenticating the client everytime a database connection is  ↩
    obtained
db.user=root
db.pwd=root

# The JDBC Driver class that needs to be loaded from the classpath (mysql-connector-x.y.z. ↩
    jar)
# which does the actual talking to the database from within the Java based Programming  ↩
    language
db.driver=com.mysql.jdbc.Driver

# Database table name
db.tableName=GroovyScriptTest
```

1.6.2 Parsing the configuration file

We will write a script to parse the database configuration file so that we can read and process the values and get them stored in to respective variables of a class. This way we can use these variables from the class any number of times we need, otherwise we may need to read it from the configuration file which is typically be slow and a time consuming operation when compared to reading it from the Groovy object variable.

The below Groovy script will read the configuration file from the file system and parse the values and store them into respective variables. To make it efficient reading, we read them through `static` initializer blocks which will be executed only once during the class loading time. The values read will be retained until the script completes its execution or the JVM shuts down. Just to verify the properties being read, we will print them in the console via a helper method called `printValues()`.

We will compile this Groovy class using `groovyc DBProps.groovy` so that the compiled .class file gets generated in a folder matching with `com/javacodegeeks/example/groovy/scripting` in the present directory. It will be referred by other scripts to load the Order class.

dbProperties.groovy

```
package com.javacodegeeks.example.groovy.scripting

class DBProps
{
    // meaningful class level variables for each of the db properties
    static String dbType, dbHost, dbPort, dbName, dbUser, dbPwd, dbUrl, dbDriver, dbTblName

    // class level varaibles for the properties file
    static String baseDir=".", propsFileName = 'db.properties'

    // class level variables to hold the properties loaded
    static def props = new Properties(), config

    //static initializer block (gets invoked when the class is loaded and only once)
    static
    {
```

```
        // Groovy's simple way to load a configuration file from a directory and
        // create a Properties (java.util.Properties) instance out of it
        new File(baseDir, propsFileName).withInputStream {
          stream -> props.load(stream)
        }

        // ConfigSlurper supports accessing properties using GPath expressions (dot notation)
        config = new ConfigSlurper().parse(props)

        // Assign the value of each property through ConfigSlurper instance
        dbType    = config.db.type
        dbName    = config.db.name
        dbHost    = config.db.host
        dbPort    = config.db.port
        dbUser    = config.db.user
        dbPwd     = config.db.pwd
        dbDriver  = config.db.driver
        dbTblName = config.db.tableName

        dbUrl = 'jdbc:' + dbType + '://' + dbHost + ':' + dbPort + '/' + dbName
        //assert dbUrl=='jdbc:mysql://localhost:3306/test'
    }

    static printValues()
    {
        println "Db Type     : " + DBProps.dbType
        println "Db Name     : " + DBProps.dbName
        println "Db Host     : " + DBProps.dbHost
        println "Db Port     : " + DBProps.dbPort
        println "Db User     : " + DBProps.dbUser
        println "Db Pwd      : " + DBProps.dbPwd
        println "Db URL      : " + DBProps.dbUrl
        println "JDBC Driver : " + DBProps.dbDriver
        println "Table Name  : " + DBProps.dbTblName
    }
}

// To test the values through the same script
DBProps.printValues()
```

The script produces the following output.

```
Db Type     : mysql
Db Name     : test
Db Host     : localhost
Db Port     : 3306
Db User     : root
Db Pwd      : root
Db URL      : jdbc:mysql://localhost:3306/test
JDBC Driver : com.mysql.jdbc.Driver
Table Name  : GroovyScriptTest
```

As you see, the values read from the configuration / properties file were stored into a respective variables of a class, which will be referred in different other scripts for the efficient and ease of use.

1.6.3 Create Table

We will now write a Groovy script to create a database table `GroovyScriptTest` to capture the Order related information for our example. This table will have a placeholder to store every bit of information about our Order - as we discussed in

Business Object in the `Order.groovy`. However the data types and declaration in the database may be little different from the Programming language(s).

For executing the database scripts, you should specify the directory where the MySQL JDBC Jar file is present, as a classpath option while invoking the groovy script. For example, to invoke the below script the command should be `groovy -classpath C:commonj2eelibsmysql-connector-java-5.1.29-bin.jar dbCreateTable.groovy`, where the mysql jar file `mysql-connector-java-5.1.29-bin.jar` is present in the directory `C:commonj2eelibs`

dbCreateTable.groovy

```
package com.javacodegeeks.example.groovy.scripting

// Database related methods are present in Sql class
// we need to import the Sql class from groovy.sql package
import groovy.sql.Sql

// we read the database properties from DBProps class
// which had already read those values from the configuration file
def url = DBProps.dbUrl
def user = DBProps.dbUser
def password = DBProps.dbPwd
def driver = DBProps.dbDriver
def tableName = DBProps.dbTblName

// Groovy's way of executing SQL statement after obtaining a connection
// from database. 'withInstance' is a special construct that helps the
// connection closed automatically even in case of any errors
Sql.withInstance(url, user, password, driver) { sql ->

  sql.execute '''
    CREATE TABLE ''' + tableName + '''
    (
        Id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
        NAME VARCHAR(30),
        COUNTRY VARCHAR(20),
        PRICE FLOAT(8,2),
        CURRENCY VARCHAR(3),
        PAYMENT_MODE VARCHAR(10),
        ORDER_DATE DATETIME DEFAULT CURRENT_TIMESTAMP
    );
  '''

  println "Table [${tableName}] created successfully"
}
```

The script produces the following output.

```
Table [GroovyScriptTest] created successfully
```

The above script created a table named `GroovyScriptTest` in the MySQL database named *test* with the columns *Id* being auto-generated primary key (unique value per row), *Name* of String datatype with maximum 30 characters limit, *country* of String datatype of maximum 20 characters limit, *price* of floating data type of 8 digits with 2 digits for precision, *currency* as a String datatype of 30 characters limit, *Payment_Mode* of String datatype of 10 characters limit and *Order_date* of datetime datatype with the default value of the current date and time in the System, if not specified.

1.6.4 Insert Rows from a Single File

Now that we had created a table in the MySQL database, let us write/ insert some data into the table from a flat file. We had already executed a script `TestOrderRandomBulk.groovy` where it had produced a flat file `outputTest.txt` in the directory `outputFiles`. The below script `dbInsertValueSingleFile.groovyy` will read the same flat file and insert into the `GroovyScriptTest` table.

dbInsertValueSingleFile.groovy

```
package com.javacodegeeks.example.groovy.scripting

import groovy.sql.Sql

// Properties of DBProps class parsed out of the configuration file
def url = DBProps.dbUrl
def user = DBProps.dbUser
def password = DBProps.dbPwd
def driver = DBProps.dbDriver
def tableName = DBProps.dbTblName

def baseDir="./outputFiles"
def fileName = 'outputTest.txt'

// a list to hold the list of valid (non-empty) lines in the flat file
def list = new File(baseDir, fileName).readLines()*.trim()

def orderList = []

list.each { it ->
       // Consider ONLY the non-empty lines in the flat file
       if(it.trim().length() > 0)
          orderList.add(new Order().initOrder(it))
}

Sql.withInstance(url, user, password, driver) { sql ->
  //process each line which is an Order Instance
  orderList.each { it ->
      sql.execute """
          INSERT INTO ${Sql.expand(tableName)}
            (NAME, COUNTRY, PRICE, CURRENCY, PAYMENT_MODE)
            VALUES
            (
                ${it.customerName}, ${it.country},
                ${it.price}, ${it.currency}, ${it.paymentMode}
            );
      """
  }

  println "Total Rows Inserted to db : " + orderList.size()
}


println " "
println "*********** "
println " == COMPLETED ==  "
println "*********** "
```

The script produces the following output.

```
package com.javacodegeeks.example.groovy.scripting
```

The script produces the following output.

```
Total Rows Inserted to db : 5

***********
 == COMPLETED ==
***********
```

The above script successfully read the flat file and inserted each line as a separate Order entry in the database table. The output confirmed that there were 5 Order instances successfully inserted.

Let us verify the values inserted in the database table with another script as discussed in the next subsection.

1.6.5 Select Total Count

Let us write a simple Groovy script to get the total number of rows present in the database table.

dbSelectTotal.groovy

```
package com.javacodegeeks.example.groovy.scripting

import groovy.sql.Sql

def url = DBProps.dbUrl
def user = DBProps.dbUser
def password = DBProps.dbPwd
def driver = DBProps.dbDriver
def tableName = DBProps.dbTblName

// We will have a explicit handle on sql instance through newInstance() method
// which will require us to manually close the sql instance after the work is done
def sql = Sql.newInstance(url, user, password, driver)

query="""Select COUNT(*) as TotalRows from """ + tableName

def totatlRows

sql.query(query) { resultSet ->
    while(resultSet.next()) {
        totalRows = resultSet.getString(1)
        println "Total Rows in the table [${tableName}] : ${totalRows}"
    }
}

// Sql.withInstance() automatically closes, else we need to manually close the sql ↩
    connection)
sql.close()
```

The script produces the following output.

```
Total Rows in the table [GroovyScriptTest] :: 5
```

1.6.6 Select All Rows

Let us write yet another Groovy script to retrieve all the rows available in the `GroovyScriptTest` database table.

dbSelectRows.groovy

```
package com.javacodegeeks.example.groovy.scripting

import groovy.sql.Sql

def url = DBProps.dbUrl
def user = DBProps.dbUser
def password = DBProps.dbPwd
def driver = DBProps.dbDriver
def tableName = DBProps.dbTblName
```

```
def sql = Sql.newInstance(url, user, password, driver)

def query= '''select count(*) as total_rows from ''' +  tableName

def totalRows

sql.query(query) { resultSet ->
    while(resultSet.next()) {
        totalRows = resultSet.getString(1)
        println "Total Rows in the table [${tableName}] : ${totalRows}"
    }
}

def fieldsList = "Id, Name, Country, Price, Currency, Payment_Mode, Order_Date"

query='''
    Select
       Id, Name, Country, Price, Currency, Payment_Mode, Order_Date
    from
        ''' + tableName

// declare the individual variables to hold the respective values from database
def id, name, country, orderNo, price, currency, mode, orderDate

// a list to hold a set of Order Instances
def orderList = []

// a variable to hold an Order Instance prepared out of the values extracted from database
def order

sql.query(query) { rs ->
    while(rs.next())
    {
        // extract each field from the resultset(rs) and
        // store them in separate variables
        id = rs.getString(1)
        name = rs.getString(2)
        country = rs.getString(3)
        price = rs.getString(4)
        currency = rs.getString(5)
        mode = rs.getString(6)
        orderDate = rs.getString(7)

        // create an Order Instance from the values extracted
        order = new Order('id' : id, 'customerName': name, 'country': country, 'price':  ↩
            price,
                    'currency' : currency, 'paymentMode' : mode, 'orderDate' : orderDate)

        // add the order instances into a collection (List)
        orderList.add(order)
    }
}

// Prepare the column headers with a fixed width for each
def fieldListDisplay = String.format("%-5s", "Id") + " | " +
         String.format("%-15s", "Customer Name") + " | "  +
         String.format("%-10s", "Country") + " | " +
         String.format("%-8s", "Price") + " | " +
         String.format("%-8s", "Currency") + " | " +
         String.format("%-12s", "Payment Mode") + " | " +
         String.format("%-21s", "Order Date") + " | "
```

```
// an utility method to display a dash ("-") for a visual aid
// to keep the data aligned during the display in the console
def printDashes(limit) {
    for(int i = 0 ; i 0)
{
    //print the Headers
    println ""
    def dashWidth = 102
    printDashes(dashWidth)
    println fieldListDisplay
    printDashes(dashWidth)

    // Print each of the order instances
    // Here the toString() method will be invoked on the Order instance
    orderList.each { it ->
         println it
    }

    // To print a line with dashes to indicate the end of the display
    printDashes(dashWidth)
 }

// Dont' forget to close the database connection
sql.close()
```

The script produces the following output.

```
Total Rows in the table [GroovyScriptTest] :: 5

**********************************************************************
Id    | Customer Name    | Country    | Price    | Currency | Payment Mode | Order Date  ↩
              |
**********************************************************************
1     | Manoj            | USA        | 9877.12  | USD      | Cheque       | 2016-04-15  ↩
    16:53:14.0 |
2     | Jhanani          | Canada     | 4567.00  | CAD      | Cash         | 2016-04-15  ↩
    16:53:14.0 |
3     | Kanna            | Singapore  | 9877.12  | SGD      | Card         | 2016-04-15  ↩
    16:53:14.0 |
4     | Karthiga         | Saudi      | 9877.12  | SAR      | Cash         | 2016-04-15  ↩
    16:53:14.0 |
5     | Saravanan        | Australia  | 8998.11  | AUD      | Cheque       | 2016-04-15  ↩
    16:53:14.0 |
**********************************************************************
```

As you see, the above script extracted all the 5 rows from the database table and created a separate Order Instance using the values, in an iteration. All those Order instances were printed with the help of `toString()` method that helps us to get the values of an Order Instance with a fixed width - to have a better visual aid while displaying. The values are displayed with the headers to make it meaningful to indicate which value belongs to which column.

Please observe the values for **Id** column where it is in sequence starting from 1 to 5. It was because we declared the syntax for the column while creating the database table as an *auto generated, identity* column.

1.6.7 Insert Rows from the matching files at interval

Now we had successfully inserted the records from a single flat file and also verified the total record count plus extracted all the rows available in a table. We will enhance the script further to make it process the files in iteration at regular intervals while letting the script sleep for a while. The duration for sleeping, number of iterations are configured in the script.

Note: This script will look for the files with the matching pattern `outputTest.txt_count_<yyyyMMdd_HHmmss>` which were generated out of the script `TestOrderRandomInterval.groovy`

For the brevity, the script will process two iterations and will exit. However, to make it execute indefinitely, you can make the while loop as follows.

```
while(true) {
        ...
    }
```

dbInsertValueInterval.groovy

```
package com.javacodegeeks.example.groovy.scripting

import groovy.sql.Sql
import groovy.io.FileType

def url = DBProps.dbUrl
def user = DBProps.dbUser
def password = DBProps.dbPwd
def driver = DBProps.dbDriver
def tableName = DBProps.dbTblName

def baseDir = "."
def targetDir="./outputFiles"
def fileNamePattern = 'outputTest.txt_'

def dir = new File(baseDir)
def list, fileName, fileNameMatches
def orderlist = []
// interval to let the script sleep (1000ms = 1s, here it indicates 1 minute)
def SLEEP_INTERVAL = 1000*60*1
def MAX_ITERATIONS = 2, i = 1

// an utility method to print the current date and time in a specified format
def getNow()
{
    new Date().format("yyyyMMdd_HHmmss")
}

println " "
println "Script will try ${MAX_ITERATIONS} times for the matching input files while ↩
    sleeping for ${SLEEP_INTERVAL/60000} minutes"

while(i++

       fileName = file.name
       fileNameMatches = false

       if(fileName.startsWith(fileNamePattern))
       {
           println "Processing the file -> $fileName"
           fileNameMatches = true
       }

       if(fileNameMatches)
       {
            // store all the lines from a file into a List, after trimming the spaces on ↩
                each line
            list = new File(baseDir, fileName).readLines()*.trim()
            orderList = []

            // Process only the valid (non-empty) lines and prepare an Order instance
            // by calling the initOrder() method by passing the pipe delimited line of text
            list.each { it ->
```

```
                if(it.trim().length()>0)
                  orderList.add(new Order().initOrder(it))
            }

            Sql.withInstance(url, user, password, driver) { sql ->
              orderList.each { it ->
                  sql.execute """
                      INSERT INTO ${Sql.expand(tableName)}
                    (NAME, COUNTRY, PRICE, CURRENCY, PAYMENT_MODE)
                    VALUES
                    (
                        ${it.customerName}, ${it.country},
                        ${it.price}, ${it.currency}, ${it.paymentMode}
                    );
                  """
              }

              println "Total Rows Inserted to db : " + orderList.size()
            }

            // Move the processed file into a different directory
            // so that the same file will NOT be processed during next run
            "mv ${fileName} ${targetDir}".execute()
            println "File ${fileName} moved to dir [${targetDir}] successfully."
           }
        }

    println ""
    println getNow() + " ... script will sleep for ${SLEEP_INTERVAL} milliseconds..."
    println ""
    sleep(SLEEP_INTERVAL)
    println getNow() + " ......... script awaken ........ "
    println ""
}

println " "
println "*********** "
println " == COMPLETED ==  "
println "*********** "
```

The script produces the following output.

```
Script will try 2 times for the matching input files while sleeping for 1 minutes

************************************************************-----------
20160415_172226 ========== SCANNING for files matching with outputTest.txt_  ↩
    ==================
************************************************************-----------

Processing the file -> outputTest.txt_1_20160415_122040
Total Rows Inserted to db : 5
File outputTest.txt_1_20160415_122040 moved to dir [./outputFiles] successfully.
Processing the file -> outputTest.txt_2_20160415_122140
Total Rows Inserted to db : 5
File outputTest.txt_2_20160415_122140 moved to dir [./outputFiles] successfully.

20160415_172227 ... script will sleep for 60000 milliseconds...

20160415_172327 ......... script awaken ........


************************************************************-----------
```

```
20160415_172327 =========== SCANNING for files matching with outputTest.txt_  ↩
    ===================
**********************************************************------------


20160415_172427 ... script will sleep for 60000 milliseconds...

20160415_172427 ......... script awaken ........

***********
 == COMPLETED ==
***********
```

You can verify the total values inserted by executing the `dbSelectRows.groovy` where it will display all the rows in the table with a fixed width column for each of the values as we saw in *section 6.6.*

1.6.8 Truncate Table

We will also see a Groovy Script to truncate the database table. Truncating helps to retain the table structure but wipe off (delete) all the rows in the table. It will be helpful whenever you want to start the operations afresh from the beginning.

Once the table is truncated successfully, you can insert the values again, for which the auto-generated Id value will start from 1.

dbTruncateTable.groovy

```
package com.javacodegeeks.example.groovy.scripting

import groovy.sql.Sql

def url = DBProps.dbUrl
def user = DBProps.dbUser
def password = DBProps.dbPwd
def driver = DBProps.dbDriver
def tableName = DBProps.dbTblName

Sql.withInstance(url, user, password, driver) { sql ->
  sql.execute """TRUNCATE TABLE """ + tableName

  println "Table [${tableName}] truncated successfully"
}
```

The script produces the following output.

```
Table [GroovyScriptTest] truncated successfully
```

1.6.9 Drop Table

At times we need to drop the table to start everything afresh. Let us write a simple Groovy script for dropping the table from the database.

Once the table is dropped successfully, you need to recreate the table by executing the script `dbCreateTable.groovy` before you can insert the values.

dbDropTable.groovy

```
package com.javacodegeeks.example.groovy.scripting

import groovy.sql.Sql

def url = DBProps.dbUrl
```

```
def user = DBProps.dbUser
def password = DBProps.dbPwd
def driver = DBProps.dbDriver
def tableName = DBProps.dbTblName

Sql.withInstance(url, user, password, driver) { sql ->
  sql.execute '''DROP TABLE ''' + tableName

  println "Table [${tableName}] dropped successfully"
}
```

The script produces the following output.

```
Table [GroovyScriptTest] dropped successfully
```

1.7 Conclusion

Hope you had enjoyed this article - Groovy Script Tutorial for Beginners. In this article we have seen how to write simple Groovy scripts to create a Business Object, generate random Order entries, write them into a text file, read the flat file and parse the order entries, read a property/configuration file for the database related values, create a database table, store values into a table, read the values from the table, truncate and drop the table.

Though the article aimed at a practical scenario, there may be few different conditions and best practices for a requirement if any different you have at hand. In such case, you are required to go through the documentation of the respective language constructs for a better scripting.

1.8 Code References

You may please refer the following URLs for further reading.

- Groovy Official Documentation
- MySQL Downloads
- MySQL JDBC Connector Jar Download
- Groovy String Example Java Code Geeks Example
- Groovy Array Example Java Code Geeks Example
- Groovy Closures Example Java Code Geeks Example
- Groovy List Example from Java Code Geeks Example
- Groovy Map Example from Java Code Geeks Example
- Groovy Working with IO
- Groovy Database Interaction

1.9 Download the Source Code

This is an example of how to write scripting in Groovy, tested with the Command Prompt / Shell and Groovy Console against Groovy Version 2.4.3.

Download

You can download the full source code of this example here: **Groovy Scripting Tutorial**. Please read the **ReadMe.txt** file for how to execute the scripts.

Chapter 2

Groovy Dictionary Example

In this article we will see how to have a Dictionary in `Groovy`. Groovy does not have a built in dictionary like `Python` language. However we can use the `Map` data structure for manipulating a Dictionary.

To understand this article, the reader should have a better understanding of how Map works in Groovy. You can read this Groovy-Map-Example for the same.

2.1 Environment

The examples shown below are executed in the *Command Prompt / Shell*. But they are guaranteed to work with any of the IDEs (like *Eclipse, IntelliJ IDEA, Netbeans*) through the respective plugins. The examples are executed with the Groovy Version **Groovy Version: 2.4.3**.

2.2 Concept behind Dictionary

Ideally a Dictionary is a Collection of things being stored under a key-value pair and that can be retrieved quickly anytime on demand. Though we would achieve the Dictionary via a Map, we need to have few boundaries set.

A Map will allow a null key or value to be set in the collection. However a dictionary is not supposed to have a null key or value, which otherwise will defeat its purpose. Hence, we will have to restrict the same in our Groovy code.

As far as the key/value pairs are concerned, they can be any valid identifiers and the objects. However if you are putting an Object of a custom class (your implementation) as a value, you need to ensure the `equals()` and `hashCode()` methods are implemented appropriately, as it is still based on Java Map Interface which works on Hashing. The good implementation of these two methods will guarantee an efficient and collision-free manipulation of key-value in the collection.

2.3 Examples

We will see two different examples in this article.

- Custom Dictionary - Illustration
- Country Captial Dictionary - Realistic Example

2.3.1 Custom Dictionary - Illustration

In this example, let us have a custom class to facilitate a Dictionary with the validations in place to ensure there are no null key/values added. It is mandatory otherwise the `Map` interface will allow the null key / value by default. As explained above, this will only ensure the right implementation of a Dictionary.

GroovyCustomDictionary.groovy

```
package com.javacodegeeks.example.groovy.util;

def scriptSimpleName = this.class.getSimpleName()

println "================================================ "
println "Output of Script : " + scriptSimpleName
println "Executed at      : " + new java.util.Date()
println "================================================ "
println " "

class Dictionary
{
    def key
    def value

    def dict = [:]

    // Empty No-arg constructor. Required for the overloaded constructor below
    Dictionary() {}

    // A two-arg constructor to facilitate an entry to be added during instantiation
    Dictionary(key, value) {
      put(key,value)
    }

    // Method to validate the key, value inputs for not-null
    def validate(key, value)
    {
        if(key==null)
            throw new RuntimeException("Null key is not permitted")

        if(value==null)
            throw new RuntimeException("Null value is not permitted")
    }

    // Actual method to store the key-value pairs.
    // Exception message printed if any of them is null.
    def put(key, value) {
        try {
            validate(key,value)
            this.dict[key]=value
            printInfo()
        } catch(Exception exception) {
            println "  #### ERROR #### --> " + exception.getMessage()
            println " "
        }
    }

    // Overridden toString() to have a meaningful display
    String toString() {
        "[Dictionary] hashCode = ${this.hashCode()}, Dict : ${dict}"
    }

    // Utility method for printing the information
```

```
    def printInfo() {
        println "myDict Custom Object : ${this}"
        println " "
    }
}

def myDictObj = new Dictionary()
println " ==> 1. Initial Empty Dictionary ..."
myDictObj.printInfo()
println " ==> 2. Attempting to Store values ... "
myDictObj.put('name', 'Sam')
myDictObj.put('age', 14)
println " ==> 3. Attempting to store Null Key ... "
myDictObj.put(null, 'NotNullValue');
println " ==> 4. Attempting to store Null value ... "
myDictObj.put("NullKey",null);
println " ==> 5. Attempting with duplicate key (value will be replaced) ... "
myDictObj.put('name', 'Samuel')

println " "
println " *********** END OF SCRIPT EXECUTION ***********----  "
println " "
```

The above script produces the following output.

```
================================================
Output of Script : GroovyCustomDictionary
Executed at      : Thu Mar 24 23:39:02 IST 2016
================================================

 ==> 1. Initial Empty Dictionary ...
myDict Custom Object : [Dictionary] hashCode = 1300638095, Dict : [:]

 ==> 2. Attempting to Store values ...
myDict Custom Object : [Dictionary] hashCode = 1300638095, Dict : [name:Sam]

myDict Custom Object : [Dictionary] hashCode = 1300638095, Dict : [name:Sam, age:14]

 ==> 3. Attempting to store Null Key ...
  #### ERROR #### --> Null key is not permitted

 ==> 4. Attempting to store Null value ...
  #### ERROR #### --> Null value is not permitted

 ==> 5. Attempting with duplicate key (value will be replaced) ...
myDict Custom Object : [Dictionary] hashCode = 1300638095, Dict : [name:Samuel, age:14]

 *********** END OF SCRIPT EXECUTION ************----
```

2.3.2 Country Captial Dictionary - Realistic Example

In this example, we will load the inputs from a text file `country-capital.txt` which has got a few countries and their capitals in each line as a comma separated value. We will load this text file and create a Dictionary (Map) at runtime.

The country-capital.txt file has the following contents (for simplicity, only few lines of the file is shown below).

GroovyCountryCapitalDictionary.groovy

```
# Ref URL : https://www.countries-ofthe-world.com/capitals-of-the-world.html

Afghanishtan, Kabul
```

```
Australia, Canberra
Austria, Vienna
Bahamas, Nassau
Bahrain, Manama
```

The following example shows the pattern of how a Dictionary can be used in a real world as follows.

- A Dictionary is created with the fixed set of inputs parsed out of a text file (by excluding comments and blank lines)
- Inputs are validated for not being NULL before getting added into the Dictionary Collection
- Program is executed in a Command line with two different options - 1. Print the Dictionary Contents 2. Get the country or capital matching with the input passed in the command line
- Program warns and assists the user with the Usage Information if the arguments are inappropriate
- Program does NOT facilitate addition/removal of entries to and from Dictionary.

GroovyCountryCapitalDictionary.groovy

```
package com.javacodegeeks.example.groovy.util;

def file = new File('country-capital.txt')
//println file.getAbsolutePath()

def validLines = []

file.each { line ->
    if(line.trim().length()>0 && !line.startsWith("#"))
       validLines.add(line)
}

def dict = [:]
def key, value
def tokens

validLines.each { line ->
    tokens = line.tokenize(",")
    key = tokens[0].trim()
    value = tokens[1].trim()
    validate(key,value)
    dict[key]=value
}

/* Method to validate the key, value inputs for not-null */
def validate(key, value)
{
    if(key==null)
        throw new RuntimeException("Null key is not permitted")

    if(value==null)
        throw new RuntimeException("Null value is not permitted")
}

def getCountryStartsWith = { String s ->
    def result = []
    dict.keySet().each {
        if(it.startsWith(s))
           result.add(it)
    }
    result
}
```

```
def getCapitalStartsWith = { String s ->
    def result = []
    dict.values().each {
        if(it.startsWith(s))
          result.add(it)
    }
    result
}

def printUsage() {
    def scriptName = this.class.getSimpleName()
    println ""
    println "[Usage] Please use any of the following : "
    println "  1.  $scriptName <[Country:] [Capital:]"
    println "  2.  $scriptName print --> to print the dictionary"
    System.exit(1)
}

def handleInvalidArgs() {
    println " "
    println " ## Please specify the input in proper format"
    printUsage()
}

if(args.length<1) {
   printUsage()
}

if(args[0].equalsIgnoreCase("print")) {
    println " "
    println "Total Elements in Dictionary : " + dict.size()
    println " "
    println " Dictionary Contents  "
    println "***********----------"
    println dict
    System.exit(0)
}
else
{
    def argType
    def argTokens
    def inputArg = args[0]

    argTokens = args[0].tokenize(':')

    if(argTokens.size()<2) {
        handleInvalidArgs()
    }

    argType = argTokens[0]

    if(argType.equalsIgnoreCase("country")) {
        def countryAlphabet = argTokens[1]
    println ""
        println "Country starts with '${countryAlphabet}' : "
        println getCountryStartsWith("${countryAlphabet}")
    }
    else if(argType.equalsIgnoreCase("capital")) {
        def capitalAlphabet = argTokens[1]
        println ""
        println "Capital starts with '${capitalAlphabet}' : "
```

```
        println getCapitalStartsWith("${capitalAlphabet}")
    }
    else {
        handleInvalidArgs()
    }
}
```

The above script produces the following output when simply executed without any arguments. The output displayed will be the Usage information with the format.

```
[Usage] Please use any of the following :
  1.  GroovyCountryCapitalDictionary <[Country:] [Capital:]
  2.  GroovyCountryCapitalDictionary print --> to print the dictionary
```

When you specify the starting alphabet(s) of a Country that you are looking for, the program displays the output with the countries matching with the characters/alphabets specified in the command line.

```
groovy GroovyCountryCapitalDictionary.groovy Country:I

Country starts with 'I' :
[India, Indonesia, Ireland, Italy]
```

You can also specify the alphabet(s) for the Capital, as follows.

```
groovy GroovyCountryCapitalDictionary.groovy Capital:A

Capital starts with 'A' :
[Athens, Amsterdam, Ankara, Abu Dhabi]
```

If the input pattern is wrong, the program displays the Usage information to give a correct input next time. Note that the below invocation of the program does NOT have the starting alphabet, which is *purposefully omitted*.

```
 groovy GroovyCountryCapitalDictionary.groovy Country:

 ## Please specify the input in proper format

[Usage] Please use any of the following :
  1.  GroovyCountryCapitalDictionary <[Country:] [Capital:]
  2.  GroovyCountryCapitalDictionary print --> to print the dictionary
```

Hope you found this example series helpful in manipulating the Dictionary in Groovy. For the real scenarios, you are requested to add more validations and testing to keep the logic appropriate.

2.4 Code References

You may please refer the following URLs for further reading.

Groovy Map Example from Java Code Geeks Example

Javadoc for Map Interface

GroovyMap API Javadoc

2.5 Download the Source Code

This is an example of how to manipulate a Dictionary in Groovy, tested with the Command Prompt / Shell against Groovy Version 2.4.3.

Download

You can download the full source code of this example here: **Groovy Dictionary Example**

Chapter 3

Groovy Json Example

In this article we will see how to manipulate `JSON` data in `Groovy`. JSON (JavaScript Object Notation) is the much preferred data format these days for the exchange of data between the interested parties (client and server), due to its light weight nature and ease of use.

Groovy has a built in support to work with JSON data - be it with the literal data or the the Java objects in memory. It offers a simple way to parse and build the data into JSON format.

This article will show the various situations in which you need to have a JSON format.

3.1 Environment

The examples shown below are executed in the *Command Prompt / Shell*. But they are guaranteed to work with any of the IDEs (like *Eclipse, IntelliJ IDEA, Netbeans*) through the respective plugins. The examples are executed with the Groovy Version **Groovy Version: 2.4.3**.

3.2 Important Classes

Since version 1.8, Groovy offers two classes viz `JsonSlurper` and `JsonBuilder`. These classes offer various methods to help us deal with the JSON data.

JsonSlurper is a parser which reads the input data and gets the JSON content backed up by a Java Collection (Map or List). By iterating the Collection we can get the required JSON data in the form of Domain Model class (POJO/POGO) to be used in further layers of the application. The term POJO stands for Plain Old Java Object and POGO stands for Plain Old Groovy Object. The JsonSlurper can read from various different inputs ranging from literal String, File, URL, Map etc.,

JsonBuilder class helps you to build the JSON data from your existing Domain Class (or POGO). Using the JSONBuilder object you can either get the JSON data in a ordinary String (which will be compressed and not so good for human reading), prettyPrint format (neatly formatted content for display which will be human readable), write the data into a file etc.,

In addition to these classes, we have an another useful Utiltity class called `JsonOutput`.

JSONOutput is a helper class facilitating the transformation of the JSON content from various source formats through its overloaded method toJson(). You can use the relevant method to get the JSON output from the underlying source which is passed as argument. It also offers a `prettyPrint` method for a human readable output.

For example, `toJson(String string)` produces the JSON representation for the String data passed, whereas `toJson(Map map)` returns the JSON representation for a Map that is passed as an argument.

We will use a mixture of all these in the below set of examples.

Please note that these classes belong to the package `groovy.json` and this package is **NOT** imported by default. We need to explicitly import the classes to be available in our groovy scripts.

3.3 JSON Examples

We will see a set of examples for different scenarios to work with JSON output. The source code and output are mostly self explanatory and hence will have a minimal explanation wherever required.

3.3.1 Producing JSON data from a literal String

Let us see a very simple example where we can produce the JSON data from a plain text (literal String) without actually generating an object. We will use `JsonOutput` class for this purpose.

JsonLiteral.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonOutput;

def jsonLiteral = ["name": "Raghavan", "id" : 1]

println "JSON Literal as String : " + jsonLiteral
println "JSON Literal as JSON : " + JsonOutput.toJson(jsonLiteral)
println "JSON Literal as JSON formatted : "
println JsonOutput.prettyPrint(JsonOutput.toJson(jsonLiteral))
```

The above script produces the following output.

```
JSON Literal as String : [name:Raghavan, id:1]
JSON Literal as JSON   : {"name":"Raghavan","id":1}
JSON Literal as JSON formatted :
{
    "name": "Raghavan",
    "id": 1
}
```

3.3.2 Producing JSON data from a POGO

Let us see an example where we can generate the JSON output from a `POGO (Plain Old Groovy Object).`

JsonFromPOGO.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonOutput

class Person
{
    String name
    int age

    String toString()
    {
        "[Person] name : ${name}, age : ${age}"
    }
}
def person = new Person(name:'Raghavan',age:34)
println "Person Object : " + person
println "Person Object in JSON : " + JsonOutput.toJson(person)
println "JSON Pretty Print"
println "***********-"
// prettyPrint requires a String and NOT an Object
```

```
println JsonOutput.prettyPrint(JsonOutput.toJson(person))
println ""

/* Let us work with a list of Person instances */
def json = JsonOutput.toJson([new Person(name : "Vignesh", age : 50),
            new Person(name : "Murugan", age: 45)])
println "JSON Object List : " + json
println "JSON Pretty Print "
println "***********--"
println JsonOutput.prettyPrint(json)
```

Please note that we have used two different `toJson(..)` methods above. Depending on the input argument passed (`Object`, `List`) the corresponding overloaded `toJson()` method will be invoked.

The above script produces the following output.

```
Person Object : [Person] name : Raghavan, age : 34
Person Object in JSON : {"age":34,"name":"Raghavan"}
JSON Pretty Print
***********-
{
    "age": 34,
    "name": "Raghavan"
}

JSON Object List : [{"age":50,"name":"Vignesh"},{"age":45,"name":"Murugan"}]
JSON Pretty Print
***********--
[
    {
        "age": 50,
        "name": "Vignesh"
    },
    {
        "age": 45,
        "name": "Murugan"
    }
]
```

3.3.3 Producing JSON data via JSONSlurper using literal String

`JsonSlurper` object gives us a JSON Object as a result of successful parsing of the input content passed to it. It has several overloaded methods parse() which accepts various different input sources right from literal String, File, URL, Map etc.,

We can use this JSON Object to fetch any properties of our interest by using the relevant hierarchy / relation.

We will see how we can generate the JSON output with the JsonSlurper instance below.

JsonSlurper1Basic.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonSlurper

def jsonSlurper = new JsonSlurper()
def inputText = '{"name" : "Groovy", "year": 2005}'

def jsonObject = jsonSlurper.parseText(inputText)
println "JSONObject generated out of JsonSlurper : " + jsonObject

println "jsonObject is of type : " +  jsonObject.getClass()
println "jsonObject is a Map ? " + (jsonObject instanceof Map)
```

```
assert jsonObject instanceof Map

println ""
println "Individual Attributes"
println "======================"
println "Object.name -> [" + jsonObject.name + "]"
println "Object.year -> [" + jsonObject.year + "]"
```

The above script produces the following output.

```
JSONObject generated out of JsonSlurper : [name:Groovy, year:2005]
jsonObject is of type : class groovy.json.internal.LazyMap
jsonObject is a Map ? true

Individual Attributes
======================
Object.name -> [Groovy]
Object.year -> [2005]
```

In this example, we have used a literal text to be parsed by `JsonSlurper` instance and as a result we get a JSON Object. You can see that *jsonObject* is of type *Map* (`groovy.json.internal.LazyMap`).

Later, we can retrieve the properties from a map using the usual dot notation (".") as in jsonObject.name and jsonObject.year respectively.

3.3.4 Producing JSON data via JSONSlurper using POGO Instance(s)

We can see how we can pass a literal String with a List object to be parsed by `JsonSlurper` and how we can manipulate the contents of a list.

JsonSlurper2List.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonSlurper

def jsonSlurper = new JsonSlurper()
def jsonObject = jsonSlurper.parseText('{ "vowels" : ["a", "e", "i", "o", "u"] }')

println "Json Object : " + jsonObject
println "Json Object Type : " + jsonObject.getClass()
println ""

println "Json Object - vowels : " + jsonObject.vowels
println "Json Object - vowels Type : " + jsonObject.vowels.getClass()
println "Json Object - vowels is a List ? " + (jsonObject.vowels instanceof List)
def listSize = jsonObject.vowels.size
println "Json Object - vowels Size : " + listSize
println "Json Object - vowels = first element : " + jsonObject.vowels.get(0)
println "Json Object - vowels = last  element : " + jsonObject.vowels.get(listSize-1)
```

The above script produces the following output.

```
Json Object : [vowels:[a, e, i, o, u]]
Json Object Type : class groovy.json.internal.LazyMap

Json Object - vowels : [a, e, i, o, u]
Json Object - vowels Type : class java.util.ArrayList
Json Object - vowels is a List ? true
Json Object - vowels Size : 5
Json Object - vowels = first element : a
Json Object - vowels = last  element : u
```

3.3.5 Producing JSON data via JSONSlurper and verify the data type of attributes

Below example code snippet shows that we can retrieve the data type of each of the members parsed out of JsonSlurper, which will be helpful for us to take a cautious decision while passing it further into the application.

JsonSlurper3DataTypes.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonSlurper

def jsonSlurper = new JsonSlurper()
def jsonObject = jsonSlurper.parseText '''
    {
        "name"        : "Raghavan",
        "age"         : 23,
        "temp"        : 98.4,
        "salary"      : 40000.25
    }
'''

println "JSON Object :  " + jsonObject
println "JSON Object class : " + jsonObject.getClass()
println ""
println "Individual Attributes and Data types "
println "======================================"
println "Datatype of name :  " + jsonObject.name.class
println "Datatype of age : " + jsonObject.age.class
println "Datatype of temp : " + jsonObject.temp.class
println "Datatype of salary : " + jsonObject.salary.class
```

In this example, we have used a different way of passing a literal text to JsonSlurper through a set of triple quotes (') for a multi-line string literal input.

The above script produces the following output, where you can see the data type of each of the members printed out in order. Please note that Groovy prefers `BigDecimal`, unlike Java which prefers a `float` or `double` for the floating point values.

```
JSON Object :  [age:23, name:Raghavan, salary:40000.25, temp:98.4]
JSON Object class : class groovy.json.internal.LazyMap

Individual Attributes and Data types
======================================
Datatype of name :  class java.lang.String
Datatype of age : class java.lang.Integer
Datatype of temp : class java.math.BigDecimal
Datatype of salary : class java.math.BigDecimal
```

3.3.6 Parsing JSON data via JsonSlurper by reading an input file

We will see an example of how to read the json file from disk and work with the contents retrieved inside our groovy script.

We will read the contents of the file *employee.json* and also display the same with and without pretty print for your ease of use and verification.

JsonSlurper4File.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonSlurper
import groovy.json.JsonOutput

String inputFile = 'employee.json'
```

```
String fileContents = new File(inputFile).getText('UTF-8')

def jsonSlurper = new JsonSlurper()
def jsonObject = jsonSlurper.parseText(fileContents)

println "JSONObject : " + jsonObject
println ""
println "JSONObject type : " + jsonObject.getClass()
println " "
println "JSONObject pretty printed"
println "========================="
println JsonOutput.prettyPrint(fileContents)
println ""
println "Individual Attributes"
println "***********-----"
println "JSONObject employee firstName : " + jsonObject.employee.firstName
println "JSONObject employee age : " + jsonObject.employee.age
println "JSONObject employee project name : " + jsonObject.employee.project.name
```

The above script produces the following output. You can see the attributes of the object being retrieved from the JSON Object.

```
JSONObject : [employee:[age:35, country:India, department:Technology, firstName:Raghavan,  ↩
    lastName:Muthu, project:[manager:Mark, name:Payroll Automation]]]

JSONObject type : class groovy.json.internal.LazyMap

JSONObject pretty printed
=========================
{
    "employee": {
        "firstName": "Raghavan",
        "lastName": "Muthu",
        "age": 35,
        "country": "India",
        "department": "Technology",
        "project": {
            "name": "Payroll Automation",
            "manager": "Mark"
        }
    }
}

Individual Attributes
*************-----
JSONObject employee firstName : Raghavan
JSONObject employee age : 35
JSONObject employee project name : Payroll Automation
```

3.3.7 Creating JSON data via JSONBuilder from POGO

`JsonBuilder` class helps you to generate JSON from the Java objects. It is the reverse of `JsonSlurper` which reads and parses to get you the JSON Object.

We use JsonBuilder to generate a JSON specific data that can be either written to a file, for example configuration files for an application, or to a different URL source via a suitable `java.io.Writer` Implementation class.

The advantage and specialty of the JsonBuilder class is that you can very easily construct an object by specifying the members and their values in a literal String, without actually creating an instance of a separate POGO class. That is you can build your object at runtime. This way it is very helpful to create configuration classes which are JSON specific.

In the below example, the code snippet shows two parts - first generating a JSON object on the fly using the JsonBuilder and then parsing it into a Java object via JsonSlurper for manipulating its properties one by one.

JsonBuilder1Basic.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonBuilder
import groovy.json.JsonSlurper

// Part 1 - Create a JSON content on the fly using JsonBuilder

def builder = new JsonBuilder()
builder.book {
    title 'Head First Java'
    publisher 'Orielly'
    author 'Kathy Sierra', 'Bert Bates'
    year '2005'
    currency 'USD'
    price 44.95
    format 'pdf', 'print'
}

println builder
println ""
println builder.toPrettyString()

/* Part 2 - Parse the Json Content into a JSON Object via JsonSlurper for further usage */

def jsonSlurper = new JsonSlurper()
def jsonBookObj = jsonSlurper.parseText(builder.toString())

println "JsonBook Object : "
println jsonBookObj
println "JsonBook Object type : " + jsonBookObj.getClass()
println "JsonBook Object size : " + jsonBookObj.size()
println ""

def book = jsonBookObj.book
println "Book Instance : "
println jsonBookObj.book
println "Book Instance Type : " + book.getClass()
println "Book Instance size : " + jsonBookObj.book.size()
println ""
println "Individual Attributes"
println "***********------"
println "Title : " +  jsonBookObj.book.title + " || Type : " + book.title.getClass()
println "Author : " + book.author + " || Type : " + book.author.getClass() + " || Size : "  ↩
    + book.author.size
println "Price : " + book.price + " || Type : " + book.price.getClass()
println "Currency : " + book.currency +  " || Type : " + book.currency.getClass()
println "Format : " + book.format +  " || Type : " + book.format.getClass() + " || Size : " ↩
     + book.format.size
```

The above script produces the following output.

```
{"book":{"title":"Head First Java","publisher":"Orielly","author":["Kathy Sierra","Bert  ↩
    Bates"],"year":"2005","currency":"USD","price":44.95,"format":["pdf","print"]}}

{
    "book": {
        "title": "Head First Java",
        "publisher": "Orielly",
```

```
        "author": [
            "Kathy Sierra",
            "Bert Bates"
        ],
        "year": "2005",
        "currency": "USD",
        "price": 44.95,
        "format": [
            "pdf",
            "print"
        ]
    }
}
JsonBook Object :
[book:[author:[Kathy Sierra, Bert Bates], currency:USD, format:[pdf, print], price:44.95,  ↩
    publisher:Orielly, title:Head First Java, year:2005]]
JsonBook Object type : class groovy.json.internal.LazyMap
JsonBook Object size : 1

Book Instance :
[author:[Kathy Sierra, Bert Bates], currency:USD, format:[pdf, print], price:44.95,  ↩
    publisher:Orielly, title:Head First Java, year:2005]
Book Instance Type : class groovy.json.internal.LazyMap
Book Instance size : 7

Individual Attributes
************------
Title : Head First Java || Type : class java.lang.String
Author : [Kathy Sierra, Bert Bates] || Type : class java.util.ArrayList || Size : 2
Price : 44.95 || Type : class java.math.BigDecimal
Currency : USD || Type : class java.lang.String
Format : [pdf, print] || Type : class java.util.ArrayList || Size : 2
```

3.3.8 Creating JSON data via JSONBuilder and Write to a File

In this example, we will see how we can generate the JSON data using the `JsonBuilder` and use its API method to write the JSON into a file in the disk.

This will be handy when we need to write our JSON specific configuration into a file which can later be read by a Container for serving our application to users.

In this example will write the contents into a file and subsequently we will read the file contents and display for our verification.

JsonBuilder2File.groovy

```
package com.javacodegeeks.example.groovy.json;

import groovy.json.JsonBuilder
import groovy.json.JsonOutput

def jsonBuilder = new JsonBuilder()

jsonBuilder.config
{
    env : "Prod"
    database {
        host "example.com"
        port 3306
        type "MySQL"
        user 'dbUser'
        pass 'dbPass'
```

```
        driver 'com.mysql.jdbc.Driver'
    }
    threadPool 10
    useBridge 'Y'
}

println "JSONBuilder Object : " + jsonBuilder
println ""
println "JSON Pretty Printed Config "
println "==========================="
println JsonOutput.prettyPrint(jsonBuilder.toString())
println ""

String outputFile = 'config.json'
def fileWriter = new FileWriter(outputFile)
jsonBuilder.writeTo(fileWriter)
fileWriter.flush() /* to push the data from  buffer to file */
println "Config details are written into the file '${outputFile}'"

println ""
def fileContents = new File(outputFile).text
println "File contents : " + fileContents
println ""
println "File Contents PrettyPrint"
println "=========================="
println JsonOutput.prettyPrint(fileContents)
```

The above script produces the following output.

```
JSONBuilder Object : {"config":{"database":{"host":"example.com","port":3306,"type":"MySQL" ↩
    ,"user":"dbUser","pass":"dbPass","driver":"com.mysql.jdbc.Driver"},"threadPool":10," ↩
    useBridge":"Y"}}

JSON Pretty Printed Config
==========================
{
    "config": {
        "database": {
            "host": "example.com",
            "port": 3306,
            "type": "MySQL",
            "user": "dbUser",
            "pass": "dbPass",
            "driver": "com.mysql.jdbc.Driver"
        },
        "threadPool": 10,
        "useBridge": "Y"
    }
}

Config details are written into the file 'config.json'

File contents : {"config":{"database":{"host":"example.com","port":3306,"type":"MySQL"," ↩
    user":"dbUser","pass":"dbPass","driver":"com.mysql.jdbc.Driver"},"threadPool":10," ↩
    useBridge":"Y"}}

File Contents PrettyPrint
==========================
{
    "config": {
        "database": {
            "host": "example.com",
```

```
            "port": 3306,
            "type": "MySQL",
            "user": "dbUser",
            "pass": "dbPass",
            "driver": "com.mysql.jdbc.Driver"
        },
        "threadPool": 10,
        "useBridge": "Y"
    }
}
```

Hope you found this example series helpful in manipulating the JSON data with Groovy scripts, for the simple use cases. However if you have a different (or advanced) scenario at hand, you are recommended to read the API for the variants of each class and its methods.

3.3.9 Code References

You may please refer the following URLs for further reading.

Groovy Json from Groovy Language Documentation

JsonSlurper Groovy API

JsonBuilder Groovy API

JsonOutput Groovy API

3.4 Download the Source Code

This is an example of how to read and write JSON data in Groovy, tested with the Command Prompt / Shell against Groovy Version 2.4.3.

Download

You can download the full source code of this example here: **Groovy JSON Example**

Chapter 4

Groovy String Example

4.1 Introduction

String related manipulations have been always important in almost all programming languages. If you are using Groovy, you will do String manipulation in an easy way. In other words, it is very easy to split, add, manipulate, initialize, escape Strings in Groovy. In this tutorial I will show you how to use Groovy String efficiently.

4.2 Warmup

Let's have a look at following examples for warming up.

GroovyStringBasic.groovy

```
package main.javacodegeeks.groovy

class GroovyStringBasic {

        static main(args) {
                def name = 'John' // John
                println name

                println 'The quick brown fox jumps over the lazy dog'.length() // 43
                println 'Non-Blocking IO'.indexOf("o") // 1
                println 'AngularJS Service vs Factory vs Provider'.substring(32) //  ↩
                    Provider
                println 'C# is the best'.replace('C#', 'Java') // Java is the best
                println 'I am very angry'.toUpperCase() // I AM VERY ANGRY
        }

}
```

As in other programming languages, Groovy is also capable of doing basic operations. On `line 06` we have defined a String and then printed it on `line 07` as you may guess. On `line 09`, we have printed the length of the given String. On `line 10`, we have printed the index of the first occurrence of character `o` which is 1. On `line 11`, we have applied sub string operation in order to pull string from specific starting point that we desired. Replacement is one of the basic operations in Strings. We have replaced `C#` with `Java` on `line 12`. And finally, we have changed the case of the string letters by using `toUpperCase()`

4.3 Concatenation vs GString

In order to concatenate string in Groovy you can use following example.

GroovyStringConcat.groovy

```
package main.javacodegeeks.groovy

class GroovyStringConcat {

        static main(args) {
                def name = 'John'
                def surname = 'Doe'
                println 'Hello ' + name + ' ' + surname // Hello John Doe
        }

}
```

As you can see we have used + sign to concate two given string. There is another way to show two string in combined way. When you look at following example.

GroovyGString.groovy

```
package main.javacodegeeks.groovy

class GroovyGString {

        static main(args) {
                def name = 'John'
                def surname = 'Doe'

                println "Hello ${name} ${surname}" // Hello John Doe
                println 'Hellow ${name} ${surname}' // Hellow ${name} ${surname}
        }

}
```

You will see that, the expression on `line 09` compiled as string, but on `line 10` the string is printed as it is. The double quote " is called in Groovy as GStrings and this accepts expressions inside.

4.4 Operators

You can apply math operators to Groovy strings to add, multiply or subtract them. Let's see following examples.

GroovyOperators.groovy

```
package main.javacodegeeks.groovy

class GroovyOperators {

        static main(args) {
                println 'I am very long sentence' - 'very long ' // I am sentence
                println 'I will ' + ' be very long sentence' // I will  be very long  ↩
                    sentence
                println 'Ice ' * 2 + ' baby' // Ice Ice  baby
        }

}
```

On `line 06`, we have removed part of the sentence by using − operator. It is something like removing string by using sub string, or apply replace operation on a string. This is very simple operation as you see in the example. On `line 07`, we have concatenated the strings by using + sign. The strange one is multiplication of the stirng on `line 08`. When you multiply a string, that string will be self concatenated it self by multiplication factor.

4.5 Multiple Lined Strings

You can use triple quote """ for the multiple lined strings like below.

GroovyMultiLinedString.groovy

```
package main.javacodegeeks.groovy

class GroovyMultiLinedString {

        static main(args) {
                def multiLine = """
Hi everyone, I will
write lots of things here
because I am not restricted with
one line. I am capable of
multi lines
                """
                println multiLine
        }

}
```

As you can see, we have written multi line string in triple quoted strings and it will printed as it is.

4.6 String Tokenize

Tokenize is used for splitting Strings by a delimiter. If you do not provide delimiter, it will be splitted by space, tab, or next line. Let's see how it works.

GroovyTokenize.groovy

```
package main.javacodegeeks.groovy

class GroovyTokenize {

        static main(args) {
                def text = 'Hello World'
                println text.tokenize() // [Hello, World]

                def textWithComma = 'Hello,World'
                println textWithComma.tokenize(',') // [Hello, World]

                def textWithTab = 'Hello        World'
                println textWithTab.tokenize()
        }

}
```

On `line 07`, it was delimited by space by default and printed an array with two elements that is `[Hello, World]`. On `line 10`, we have used `,` as delimiter and it has also delimited to an array. And in last example, it was automatically delimited by the tab.

4.7 Conclusion

Groovy String is very flexible usage while you are developing application. It lets you to save lots of lines due to its practical usage. You can do lots of manipulation on Groovy Strings with power of Groovy.

Download

You can download the full source code of this example as an Eclipse project here: **GroovyStringExample**

Chapter 5

Groovy Closure Example

5.1 Introduction

A closure is an anonymous code block that can take arguments, return value, and also can be assigned to a variable. When it comes to Groovy, a closure can contain a variable scope defined outside of the closure expression. Groovy offers extended features to formal closure definition. By using this features, we can write our code more dynamically by applying closures to functions as parameter, caling anonymous code blocks inside functions, etc… Let's see closure in action with supporting examples.

5.2 Closure Declaration

ClosureDeclaration.groovy

```
package com.javacodegeeks.groovy.closure

class ClosureDeclaration {

        static main(args) {
                def myClosure = { println "Hello World!" }
                myClosure.call() // Hello World!
                myClosure() // Hello World!
        }

}
```

In this example, we have defined a closure on `line 06`, and then we have called it by using `call()` function. Alternatively, you can call closures by acting as function directly as on `line 07`

5.3 Closure Parameters

Sometimes, we need to pass arguments to the closures to make them more dynamically according to our needs. You can write a closure that accepts parameters while calling them. Let's see following example to see how it works.

ClosureParameters.groovy

```
package com.javacodegeeks.groovy.closure

class ClosureParameters {
```

```
        static main(args) {

                def squareWithImplicitParameter = { it * it }
                println squareWithImplicitParameter(4) // 16

                def sumWithExplicitTypes = { int a, int b -> return a + b }
                println sumWithExplicitTypes(11, 8) // 19

                def sumWithOneExplicitOneOptionalTypes = { int a, b -> return a + b }
                println sumWithOneExplicitOneOptionalTypes(20, 13) // 33

                def sumWithDefaultParameterValue = { a, b = 5 -> return a + b }
                println sumWithDefaultParameterValue(4)  // 9

        }

}
```

On line 07, we have used implicit argument called `it` that defines default parameter provided to the closure. In square closure, we accessed the default parameter with `it`. Actually, it is equal to below code.

Square function without "it"

```
def square = {x -> x * x}
println square(4)
```

And this function takes only one parameter and returns the multiplication of the same number. Remember that, the last line in the functions returned as default in Groovy language. In `ClosureParameters.groovy` class on `line 10`, we have defined closure with 2 parameters and both of them has explicit types that is `int`. In same way, it takes two arguments and returns sum of them. You do not need to provide explicit types, you can use optional types in arguments also. When you look at `line 13`, you will see that `a` has type `int`, but `b` don't. Groovy closures also support default parameter values. On `line 16`, you need to provide first argument, but if you do not provide second parameter, it will be `5` as default.

5.4 VarArgs

You can define variable argument in closures for accepting dynamic arguments to closures. Let's have a look at following example to see how it works

ClosureVarArgs.groovy

```
package com.javacodegeeks.groovy.closure

class ClosureVarArgs {

        static main(args) {
                def combine = { String... names ->
                        names.join(',')
                }

                println combine('John', 'Doe', 'Betty') // John,Doe,Betty
        }

}
```

As you can see on `line 06`, we have defined a closure that takes variable arguments with `String...names` and we have joined them. Yes, you can also use array as one argument then join them, but this example is for showing how VarArgs can be used.

5.5 Closure Passing

You can pass closure as parameter to another closure easily. Let's look at following example.

PassingClosure.groovy

```
package com.javacodegeeks.groovy.closure

class PassingClosure {

	static main(args) {
		def funcClosure = { x, func ->
			func(x)
		}

		println funcClosure([1, 2, 3], { it.size()}) // 3
	}

}
```

As you can see on `line 06` there are two closure parameter, but they are not just a parameter like `String` or `Integer`. When you look at the closure body, we are using `func(x)`. This means, `func` is an another closure that accepts closure expression. On `line 10`, `funcClosure([1, 2, 3], { it.size()})`, first parameter is a list and second parameter is a expression that uses it, and that means first parameter of the closure `funcClosure`

5.6 Closure Composition

In Groovy, you can compose different closures to simulate compound functions in Math science. In this section, we will simulate `f(g(x)) =...`. Let say that, you will sum two numbers and then you will apply square function to sum result. You can do that by following.

PassingClosure.groovy

```
package com.javacodegeeks.groovy.closure

class ClosureComposition {

	static main(args) {

		def sum = { a, b -> return a + b }
		def square = { it * it }
		def squareOfSum = square << sum

		println squareOfSum(2, 3) // 25

	}

}
```

On `line 07`, we defined sum closure and on `line 08` square function. On line 09, we pass sum function to square function and actually it turns into following.

```
{ { a, b -> return a + b } * { a, b -> return a + b }}
```

5.7 Conclusion

Closures helps us to define anonymous-like functions in program to use that closure as parameter or function body. We can directly call it by using call(), or we can use it as parameter to another closure or function.

Download

You can download the full source code of this example as an Eclipse project here: **GroovyClosureExample**

Chapter 6

Groovy Regex Example

6.1 Introduction

Regular Expression is a character sequence defines search pattern especially for pattern matching with strings. You may see Regular Expression as **Regex** or **Regexp** in software world. In this tutorial, I will show you how to use regex operations in Groovy by using pretty easy methods.

6.2 Quick Start

In Groovy, you can define a pattern by using the tilda operator (~) for a String. When you define a pattern by using tilda operator, you will get a result in a type `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html[java.util.regex.Pattern]`. And this means, you can apply all the rules that you do in Java code. You can see quick example below.

GroovyRegex.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegex {

        static main(args) {
                def ipAddress = ~/([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)/
                println ipAddress.class.name // java.util.regex.Pattern
        }

}
```

You can see that the pattern class name is `java.util.regex.Pattern`. In order to test whether if provided string matches with the pattern, you can use following.

GroovyRegexMatch.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegexMatch {

        static main(args) {
                def nameRegex = ~'john'
                println nameRegex.matcher("john").matches() // true

                def ipAddressRegex = ~/([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)/
                println ipAddressRegex.matcher("127.0.0.1").matches() // true
```

```
        }

}
```

On line 06, we defined a pattern and we tested string "john" whether it matches with pattern ~'john' or not on line 07. On line 09, we have defined a pattern for ip address and checked the string 127.0.0.1 for pattern matching in same way on line 10.

6.3 Advanced Usage

Groovy lets you to create pattern matchers easily. For example, if you need to test a string if it starts with *L* and ends with *s*, you can use following.

GroovyRegexMatchAdvanced.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegexMatchAdvanced {

        static main(args) {
                def pattern = ~'L....e'
                println pattern.matcher("Little").matches() // true
        }

}
```

On line 06 we have defined the pattern. It says that, it will start with 'L' and then has 4 characters, and finally it ends with 'e'. On line 07, it matches with the string 'Little' which starts with 'L', ends with 'e' and has 'ittl' in the middle.

GroovyRegexMatchAdvanced.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegexMatchAdvanced {

        static main(args) {
                def pattern = ~'L....e'
                println pattern.matcher("Little").matches() // true
        }

}
```

You can also use isCase() for the pattern matching in Groovy. Let say that, you only know the starting and ending letter of the string and you want to check whetter it matches with the pattern or not. You can use following for that case.

GroovyRegexMatchAdvanced.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegexMatchAdvanced {

        static main(args) {
                def isCasePattern = ~/L\w+e/
                println isCasePattern.isCase("Little")
        }

}
```

On `line 06`, the pattern contains starting letter `L` and ending letter `e`, it contains any length of whitespace character inside `L` and `e`.

GroovyRegexMatchAdvanced.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegexMatchAdvanced {

        static main(args) {
                def grepPattern = ~/A\w+/
                def cities = ['Alabama', 'Los Angeles', 'Arizona']
                println cities.grep(grepPattern) // [Alabama, Arizona]
        }

}
```

In above example, we have defined our pattern as `~/Aw+/` which means, things that start with `A` and then we grep the cities that matches with pattern on `line 08`.

6.4 Matchers

In previous examples, we have used an expression like below.

```
pattern.matcher("Little")
```

This is what a matchers is. We have called `matches()` function over matchers. In Groovy, there is a easier way to define matchers and run `matches()` function. You can see below for example matcher.

GroovyRegexMatcher.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegexMatcher {

        static main(args) {
                def matcher = ("127.0.0.1" =~ /([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)/)
                println matcher.class.name // java.util.regex.Matcher
        }

}
```

As you can see, we have used =~ for defining a matcher. As a result, we got an object that instance of `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html[java.util.regex.Matcher]`. You can call this an IP address matcher. Let's see a couple of examples to understand matcher deeper.

GroovyRegexMatcher.groovy

```
package com.javacodegeeks.groovy.regex

class GroovyRegexMatcher {

        static main(args) {
                def numbers = ('5 is greater than 4' =~ /\d+/)
                println numbers // java.util.regex.Matcher[pattern=\d+ region=0,19  ↩
                    lastmatch=]
                println numbers.size() // 2
                println numbers[0] // 5
        }

}
```

In above example, on `line 06` we have defined a matcher `'5 is greater than 4' =~ /d+/` which says that something matches with numbers. This will find two numbers in string which are `5`, and `4`. On line 07, it prints the value of the matcher object. If you want to get the values found by matcher, you can use array notation as on `line 09`.

6.5 Conclusion

Groovy lets us to use regex as in the java but it has some extra features and easy usages. There are two section in regex world which are Pattern and Matcher. Pattern is the regular expression we define. Matcher is an expression with provided string and the regular expression. We call `matches()` function over matchers to find the things you want

Download

You can download the full source code of this example as an Eclipse project here: **GroovyRegexExample**

Chapter 7

Groovy Collect Example

7.1 Introduction

Groovy collect() is used for iterate through collections to apply closure to each element. In this tutorial, we will see how to use `collect()` for collection operations. Let's get started

7.2 Collect

When you have a look at the method signatures of collect below, we will see that collect can be executed with default closure which is `Closure.IDENTITY`, with a closure, or with a supplied collector and closure

Overloaded Methods

```
public List collect()
public List collect(Closure transform)
public Collection collect(Collection collector, Closure transform)
```

Let say that you have a list of fruits and you want to apply uppercase operation to each element of fruit list. You can do that by following code.

GroovyCollect.groovy

```
package com.javacodegeeks.groovy.collect

class GroovyCollect {

        static main(args) {
                def fruits = ["Banana", "Apple", "Grape", "Pear"]
                def upperCaseFruits = fruits.collect { it.toUpperCase() }
                println upperCaseFruits // [BANANA, APPLE, GRAPE, PEAR]
        }

}
```

As you can see on `line 07`, `upperCase()` function applied to each element, and `it` there means current element while iterating. As we said above, collect function takes closure as parameter and our closure here is `{ it.toUpperCase() }`. Let me give you another example. You have `Fruit` class and you want to construct `Fruit` object list by using fruit list. You can do that by simply apply new `Fruit` object creation closure to the collect function. You can see example below.

GroovyCollectForClass.groovy

```
package com.javacodegeeks.groovy.collect

import groovy.transform.ToString

class GroovyCollectForClass {

        static main(args) {
                def fruits = ["Banana", "Apple", "Grape", "Pear"]
                def fruitObjects = fruits.collect { new Fruit(name: it) }
                println fruitObjects // [com.javacodegeeks.groovy.collect.Fruit(name:Banana ↩
                    , amount:0), com.javacodegeeks.groovy.collect.Fruit(name:Apple, amount ↩
                    :0), com.javacodegeeks.groovy.collect.Fruit(name:Grape, amount:0), com. ↩
                    javacodegeeks.groovy.collect.Fruit(name:Pear, amount:0)]
        }

}

@ToString(includeNames = true)
class Fruit {
        def name
        def amount = 0
}
```

On `line 16`, we have declared a class with an annotation `@ToString(includeNames =true)` to show field names to be shown when we print class object. After class declaration, we apply object creation `new Fruit(name:it)` on `line 09`. While iterating fruit elements, the closure will be like this, `new Fruit(name:"Banana"), new Fruit(name: "Apple"),new Fruit(name:"Grape"),new Fruit(name:"Pear")`

7.3 Collect with Supplied Collector

Sometimes, you may need to start collect operation wiht an initial list, or whenever you collect an element, you may need to add it to different type of collection instead of list. In that case, you can use a supplementary collectors beside the closure in side `collect()` function. Let say that, you have initial fruit list, and you wan to collect another fruit list by providing initial fruit list to collect function as supplementary collector. You can see following example to understand what is going on.

GroovyCollectWithCollector.groovy

```
package com.javacodegeeks.groovy.collect

class GroovyCollectWithCollector {

        static main(args) {
                def initialFruits = ["Orange", "Lemon"]
                def fruits = ["Banana", "Apple", "Grape", "Pear"]
                def totalFruits = fruits.collect(initialFruits, { it.toUpperCase() })

                println totalFruits // [Orange, Lemon, BANANA, APPLE, GRAPE, PEAR]
                println initialFruits // [Orange, Lemon, BANANA, APPLE, GRAPE, PEAR]
        }

}
```

As you can see on `line 10`, collect operation started with `initialFruits` and closure applied to fruits elements. When you supply a collector to `collect()` function, you will get a modified version of the collector as you can see on `line 11`. `initialFruits` list has also changed, because it is the collector we supplied and collect result will be populated in that variable..

Let say that, you have list of fruits like below.

```
def fruits = ["Banana", "Apple", "Grape", "Pear", "Banana"]
```

and you want to get distinct value of fruits. You can do that by using following.

GroovyCollectWithCollectorDistinct.groovy

```
package com.javacodegeeks.groovy.collect

class GroovyCollectWithCollectorDistinct {

        static main(args) {
                def fruits = ["Banana", "Apple", "Grape", "Pear", "Banana"]
                def distinctFruits = fruits.collect(new HashSet(), { it })

                println distinctFruits // [Apple, Pear, Grape, Banana]
        }

}
```

While you are iterating elements with `collect()` on `line 07`, collected elements will be added to collector, and that collector is a `HashSet()`. This collection type is for keeping distinct values as you may already remember from Java.

7.4 Conclusion

Groovy has powerful components for the Collections SDK, and one of them is `collect()`. By using `collect()`, we have applied closures to list elements. Also, we have used different overloaded functions of `collect()` function for using default collector, or provide our own collectors like `HashSet()`.

Download

You can download the full source code of this example as an Eclipse project here: **GroovyCollectExample**

Chapter 8

Groovy Date Example

8.1 Introduction

Date operations may be painful in most of the programming languages. You may spend most of your time to convert dates from one format to another one. In Groovy, date operations are very easy. Groovy has lots of functions extended from JDK Date API, this allows us to use date related operations in an easier way. Let's have a look at how can we use Groovy date extension.

8.2 Warmup

In Java, we use Date or Calendar in for specific cases. `Date` class is a simple class and it has backward compatibility. If you want to do some date arithmetic operations, use `Calendar` class instead. In Groovy, we can use both `Date` or `Calendar`. You can see following example for initializing and getting current date.

GroovyDateInitialization.groovy

```
package com.javacodegeeks.groovy.date

import static java.util.Calendar.*

class GroovyDateInitialization {

        static main(args) {
                def date = new Date()
                println date // Sat Sep 26 19:22:50 EEST 2015

                def calendar = Calendar.instance
                println calendar // java.util.GregorianCalendar[time=1443284933986, ...
        }

}
```

As you can see, `date` variable is a simple date, but `calendar` has a value class with some instance variables. I have provided only one instance variable, because it is very long. This is simple in Java too, but what about parsing?

8.3 Date Parsing

Date parsing is the operation of conversion date string to date object. You can simply parse date string to date object easily like below.

GroovyDateParsing.groovy

```
package com.javacodegeeks.groovy.date

import static java.util.Calendar.*

class GroovyDateParsing {

        static main(args) {
                def date = new Date().parse("dd.MM.yyy", '18.05.1988')
                println date // Wed May 18 00:00:00 EEST 1988

                def extendedDate = new Date().parse("dd.MM.yyy HH:mm:ss", '18.05.1988 ↩
                    12:15:00')
                println extendedDate // Wed May 18 12:15:00 EEST 1988
        }

}
```

On `line 08`, we used parse function to parse provided date string by using date format provided as first argument. In `line 11`, **hour**, **minute**, and **seconds** parameters provided in the format and date string in order to be parse extended date. In first example on `line 08`, it printed 00:00:00, because we did not provided hour, minute, and seconds section. You can have a look at here to see full date formats.

8.4 Date Formatting

Sometimes, we need to format date object into desired format. In Groovy, you can use format function to format date object. You can see a quick example below.

GroovyDateFormatting.groovy

```
package com.javacodegeeks.groovy.date

import static java.util.Calendar.*

class GroovyDateFormatting {

        static main(args) {
                def date = new Date().parse("dd.MM.yyy", '18.05.1988')
                def formattedDate = date.format("dd/MM/yyy")
                println formattedDate // 18/05/1988
        }

}
```

In this example, we simply parsed the date string into a date object on `line 08`, and then we have formatted that date object by using format function on `line 09`. In order to format a date object, you can provide date format as argument.

8.5 Date Arithmetic

The main advantage of the Groovy date extension is the doing arithmetic operations on date object. If you have used Joda Time before, you will be very familiar to this topic. Let say that, you have a date and you want to find the date object which is 5 days after from previous date. In order to do that, you can use following example.

GroovyDateArithmetic.groovy

```
package com.javacodegeeks.groovy.date

class GroovyDateArithmetic {
```

```
        static main(args) {
                def date = new Date().parse("dd.MM.yyy", '18.05.1988')

                def datePlus = date.clone()
                def dateMinus = date.clone()

                datePlus = datePlus + 5

                println datePlus // Mon May 23 00:00:00 EEST 1988

                datePlus = datePlus.plus(5)

                println datePlus // Sat May 28 00:00:00 EEST 1988

                dateMinus = dateMinus - 10

                println dateMinus // Sun May 08 00:00:00 EEST 1988

                dateMinus = dateMinus.minus(10)

                println dateMinus // Thu Apr 28 00:00:00 EEST 1988

                def dateInterval =  dateMinus..<datePlus

                println dateInterval // [Thu Apr 28 00:00:00 EEST 1988,.., Fri May 27  ↩
                    00:00:00 EEST 1988]
        }

}
```

In above example, we have done some arithmetic operations. First, we have cloned date to two variables `dateMinus`, and `datePlus` to simulate two different dates. In order to increment date object by 5 days, we have used `datePlus =datePlus + 5` on `line 11`. Alternatively you can use `plus` function like on `line 15`. In same way, we have decremented the `dateMinus` by 10 by using subtracting sign in `dateMinus =dateMinus -10`. Alternatively, you can use `minus` function like on `line 23`. We have also find the dates between two dates `dateMinus` and `datePlus` by using the range(..) function in Groovy. You can see the date ranges on `line 29`.

8.6 Subscript Operators

In some cases, you may need to get year, month, date values separately from date objects, or you may need to set those fields in date objects separately. Subscript operators helps us to get that specific values. In fact, Groovy uses `getAt()` and `putAt()` respectively to get and set specific subscript fields in date object. Let's see some examples.

GroovyDateSubscriptOperators.groovy

```
package com.javacodegeeks.groovy.date

import static java.util.Calendar.*

class GroovyDateSubscriptOperators {

        static main(args) {
                def date = new Date().parse("dd.MM.yyy", '18.05.1988')

                println date[YEAR] // 1988
                println date[MONTH] // 4
                println date[DATE] // 18

                date[YEAR] = 1999
```

```
                date[MONTH] = 7
                date[DATE] = 20

                println date // Fri Aug 20 00:00:00 EEST 1999
        }

}
```

In this example, on `line 10`, `11`, and `12` we get the year, month, and day value from date object. This is something like getting value from map by providing the map key. But, where did `YEAR`, `MONTH`, and `DATE` come from? They come from the static `import import static java.util.Calendar.*` on `line 03`. Be careful about the month value on `line 11`, and it is the index of the month. You can also set month, year, and day values like setting a map on `line 14`, `15`, and `16`.

8.7 Conclusion

Date operations are the operations that we used during software development and it is really painful on converting dates from one format to another. In Groovy, this operations are very simplified, and you can easily perform date operations like applying arithmetic in Math. You can parse date string, format a date object, increment - decrement date, and apply subscript operators on date objects without using any third party libraries.

Download

You can download the full source code of the project here: **GroovyDateExample**

Chapter 9

Groovy Array Example

9.1 Introduction

In previous tutorials, we have mentioned about Groovy List and provided lots of examples. In this tutorial, I will show you how to use Groovy arrays. Even if array and list seems to be same, they have differences in common. For example, arrays have fixed size while lists have dynamic size that means you can add item as much as you can after initialization of the list. If you try to add item to array with a size bigger than initial size of the array after initialization, you will get MissingMethodException or ArrayIndexOutOfBoundsException. After some theory, let's see some Groovy arrays in action.

9.2 Array Declaration

In Groovy, you can declare array in Java style or Groovy style. Let's have a look at following example for array declaration.

GroovyArrayDeclaration.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayDeclaration {

        static main(args) {
                def birds = new String[3]
                birds[0] = "Parrot"
                birds.putAt(1, "Cockatiel")
                birds[2] = "Pigeon"

                println birds // [Parrot, Cockatiel, Pigeon]

                def birdArr = ["Parrot", "Cockatiel", "Pigeon"] as String[] // You say that ↩
                     this is an array of Strings

                println birdArr // [Parrot, Cockatiel, Pigeon]
        }

}
```

As you can see in the example, you can declare an array with a size, and then you can put elements in different ways in Java style. In second example, the array is directly declared with initial values. Also, it has been casted to the String that means this array has elements of type String.

9.3 Access Array Items

In Groovy, you can access array item by using square bracket with an index (`[index]`) or using `getAt(index)` function. Let's have a look at following example to see what is going on.

GroovyArrayAccessItem.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayAccessItem {

        static main(args) {

                def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

                println birds[0] // Parrot
                println birds[2] // Pigeon
                println birds.getAt(1) // Cockatiel

                println birds[-1] // Pigeon
                println birds[-3] // Parrot
        }

}
```

As you may already know, it is easy to understand the case on `line 09`, `line 10` and `line 11`. They are simple operations to get items on specific index. What about on `line 13`? Why we use negative index? It is simple, when you use negative index, it will be accessed from the end of the array with 1 based index. When you say `-1` it means the first element from the end. And in same way, when you say `-3`, it means third element from the end.

9.4 Add Item Exception

As we said earlier, if you try to add items to an array with a bigger size than the fixed length, you will get an exception as in the example below.

GroovyAddItemException.groovy

```
package com.javacodegeeks.groovy.array

class GroovyAddItemException {

        static main(args) {
                def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

                birds << "Eagle" // MissingMethodException

                birds.putAt(3, "Owl") // ArrayIndexOutOfBoundsException
        }

}
```

On `line 08`, the method << belongs to list, that is why we got `MissingMethodException`. On `line 10` we have exception due to overflow the size of the array wit a size 3.

9.5 Array Length

In Java, you can use `size()` method for the list, and `length` method for the arrays in order to get actual size of the object. In Groovy, it has been simplified and you can use size method for both arrays or lists. You can see simple example below.

GroovyArrayLength.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayLength {

	static main(args) {
		def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

		println birds.length // 3

		println birds.size() // 3
	}

}
```

9.6 Array Min & Max Value

Let say that, you have array of numbers. You can easily find the minimum and maximum value by using the `min()` and `max()` functions over arrays as below.

GroovyArrayMinMax.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayMinMax {

	static main(args) {

		def numbers = [32, 44, 12, 9, 100, 180]

		println numbers.max() // 180
		println numbers.min() // 9

		def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

		println birds.max { it.size() } // Cockatiel
		println birds.min { it.size() } // Parrot
	}

}
```

On `line 09` and `line 10` you can easily understand the case but, the trick on `line 14` and `line 15` is, the minimum and maximum value is determined by the length of the current string value. Let say that you are iterating to find minimum and maximum value, `it` means current value in the array.

9.7 Remove Item

Removing item here is not actually removing item from array. It is something like assign the array with one item removed version to another variable. Let see it in action.

GroovyArrayRemoveItem.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayRemoveItem {

	static main(args) {
```

```
                def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

                def birdsWithoutParrot = birds - "Parrot"

                println birds // [Parrot, Cockatiel, Pigeon]

                println birdsWithoutParrot // [Cockatiel, Pigeon]
        }

}
```

As you can see on `line 09`, we are using subtracting method to remove one item and assign final version to another variable. It is not removed from the original array actually as you can see on `line 11`. However, you can see another array without "Parrot" on `line 13`.

9.8 Array Ordering

If you are interacting with collections, you may need to do ordering operations time to time. For example, you may want to reverse array, or sort the array items. You can see following example to see how it looks like.

GroovyArrayOrdering.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayOrdering {

        static main(args) {

                def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

                println birds.reverse() // [Pigeon, Cockatiel, Parrot]

                println birds.sort() // [Cockatiel, Parrot, Pigeon]
        }

}
```

In above example, array is reversed and items printed on the screen in reverse way on `line 09`. Also, array sorted in alphabetical order by using `sort()` function. You can see it on `line 11`

9.9 Array Lookup

You can perform lookup operations on arrays. In following example we will reverse every item text and also we will find an item by matching a provided regex. Let's do it.

GroovyArrayLookup.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayLookup {

        static main(args) {

                def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

                def revertedBirds = birds.collect { it.reverse() }
```

```
                println revertedBirds // [torraP, leitakcoC, noegiP]

                def founded = birds.find { it =~ /Cockatiel/ }

                println founded // Cockatiel
        }

}
```

On `line 11`, we have iterated by using `collect` function and reversed each item by using `reverse()` function. Again, `it` stands for the current item in the code. On `line 13`, we have used `find` function to find specific item by using regular expression.

9.10 Conversion

You can also convert array to list in Groovy easily. You can see following example for simple usage.

GroovyArrayConversion.groovy

```
package com.javacodegeeks.groovy.array

class GroovyArrayConversion {

        static main(args) {

                def birds = ["Parrot", "Cockatiel", "Pigeon"] as String[]

                def birdList = birds.toList()

                println birdList.class.name // java.util.ArrayList

                def birdsAgain = birdList as String[]

                println birdsAgain.class.name // [Ljava.lang.String;
        }

}
```

On `line 09`, we have used toList function to convert array to list. When you check the class name of `birdList` you will see it is instance of `java.util.ArrayList`. As you can guess, you can convert list to array. Check out line 13 and it is easily casted to array. You can check the class name on `line 15`, and you will see it is `[Ljava.lang.String;`

9.11 Conclusion

To sum up, array and list has same properties in common, but there are major differences between array and list. One of the well known difference is array has fixed size and list has dynamic size in usage. If you try to add more item bigger than the array size, you will get exception. Additionally, you can perform many operations on array like list.

Download

You can download the full source code of the project here: **GroovyArrayExample**

Chapter 10

Groovy Console Example

10.1 Introduction

In this tutorial, I will show you how to use Groovy Console to run your Groovy scripts and also give you some details about Groovy Console itself. Actually, there are several ways to run Groovy scripts. You can run it on your favourite IDE, you can run it from command line, or you can use Groovy Console to run your scripts. Groovy Console is a Swing GUI for perform several operations to your Groovy scripts. You may want to use Groovy Console to write and run Groovy scripts instead of IDE or command line. Let's see features together.

10.2 Action Buttons

In Groovy Console, there are lots of actions that you can see below.

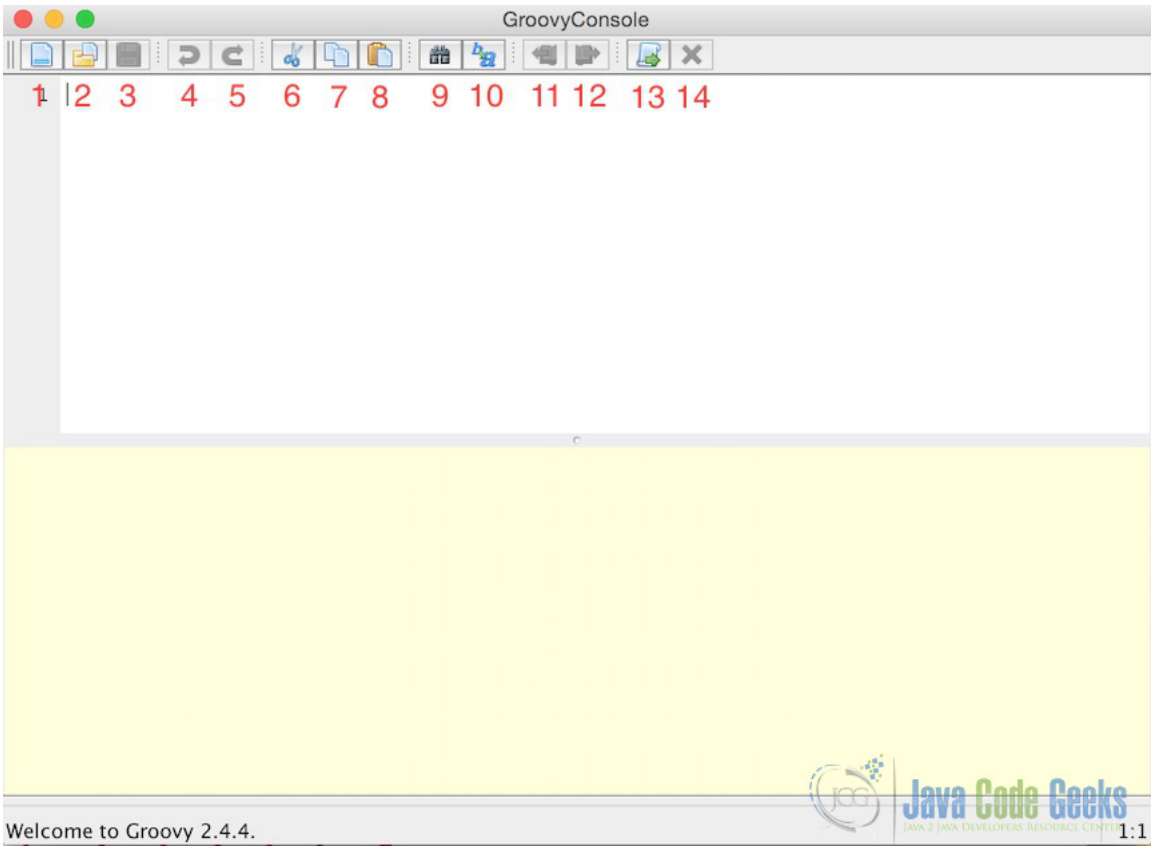

Figure 10.1: Action Buttons

Most of them are the actions that you may already know, but you can have a look at below to refresh your knowledge.

- `Action 1` is for creating new script to write your code in a fresh area
- `Action 2` is for **Opening** file
- `Action 3` is for **Saving** your code as project
- `Action 4` and `Action 5` are for **Undo** and **Redo** respectively
- `Action 6, 7, 8` are for **Cut**, **Copy**, **Paste** respectively
- `Action 9` is for looking up something with in your code
- `Action 10` is for **Replacing** text with another text within your code
- In Groovy Console, your executing actions are kept historically. You can go forward and backward along running history with `Action 11` and `Action 12`
- `Action 13` is for **Running** your current script
- `Action 14` is for **Interrupting** current running script

10.3 Quick Start

I assume you have already installed Groovy on your computer and added `GROOVY_HOME/bin` to your `PATH` to run groovy binaries. You can open Groovy console by executing `groovyconsole` command. You will see Groovy Console GUI like following.

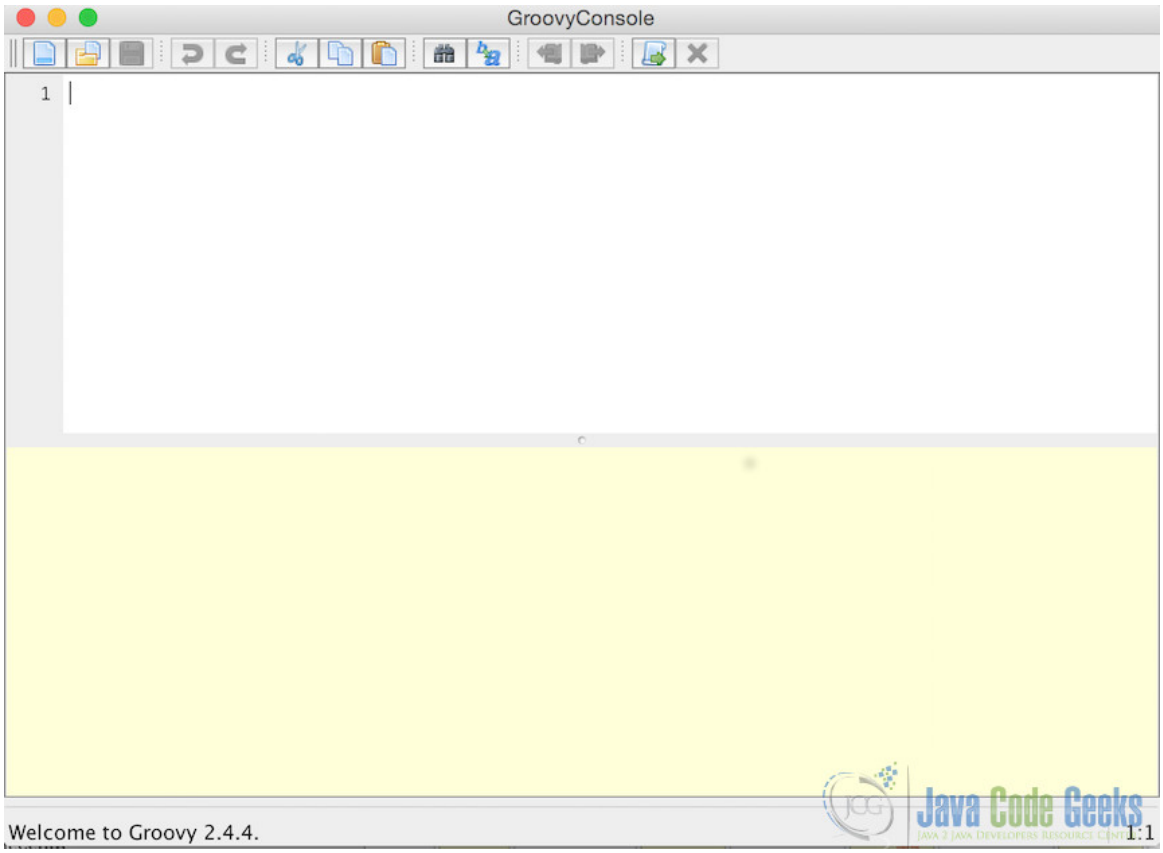

Figure 10.2: Groovy Console Start

There are two area in the GUI: Input Area, Output Area. You can write your code in Input Area and click Run button at the top right of the GUI (or Ctrl + Enter), and see result in Output Area as below.

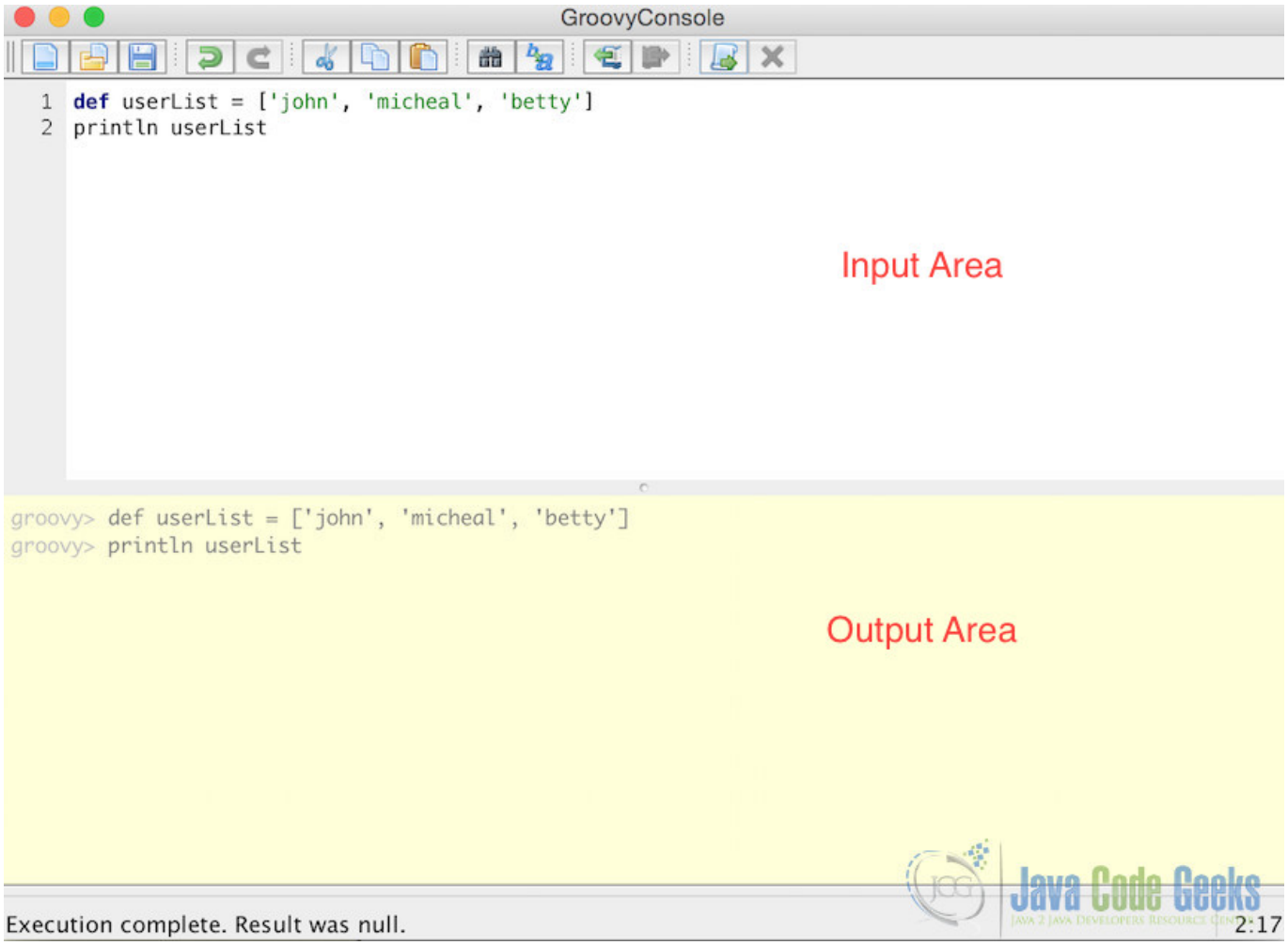

Figure 10.3: Groovy Console Input and Output Area

As you can see, there are only codes in the output area, because results are printed in the command line console where you started Groovy Console from. If you want to see results also in the output area, you need to enable **Capture Standard Output** like below.

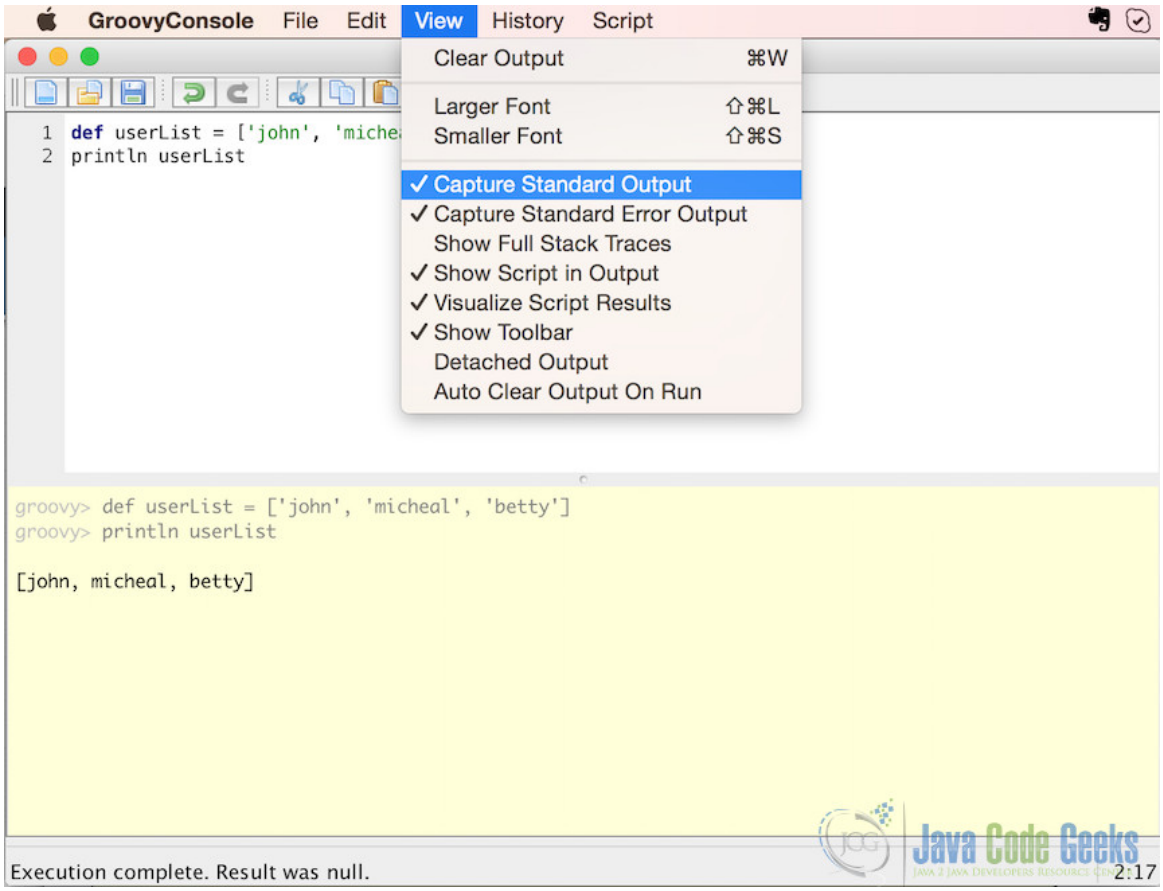

Figure 10.4: Enable Output Capture

You can also open Groovy script file by using **File > Open** menu and the script will loaded in the input area. Then you can run loaded script.

10.4 Run Selection

Additionally, you can run only selected portion of the code in Input Area. If you select a section in the Input Area and click **Run Selection (or Ctrl + Shift + R)** in the **Script** menu, only the selected code will be executed as below.

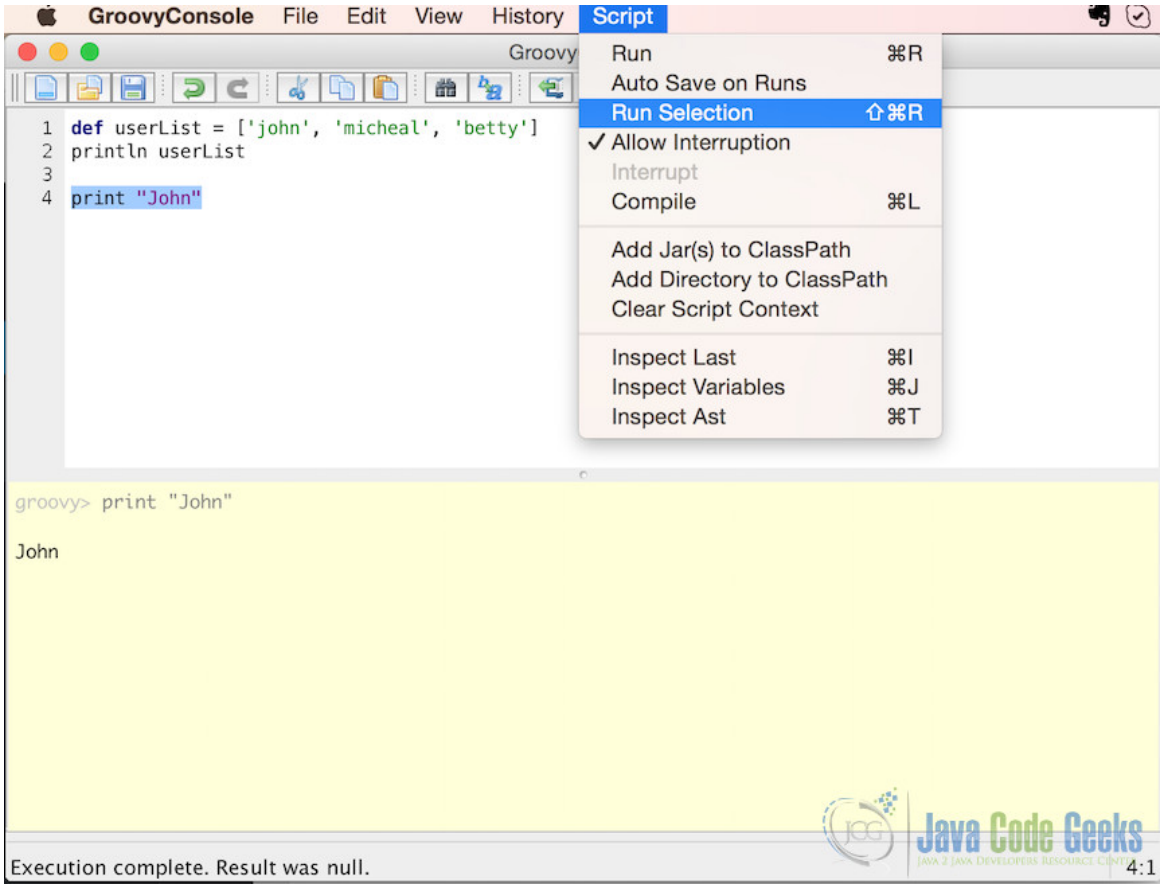

Figure 10.5: Run Selection

10.5 Interruption

You can interrupt current running script in Groovy Console after enabling interruption by selecting **Script > Allow Interruption** menu. What I mean by interruption here is interrupting current thread in running script. For example, in order to interrupt following script.

```
try {
    while(1) {
        println "something"
        Thread.currentThread().sleep(500)
    }
} catch(InterruptedException ex) {
    println "Interrupted from Groovy Console"
}
```

You can first Run script and then you can click **Interrupt** button on the right side of the **Run** button. You will see an output like below.

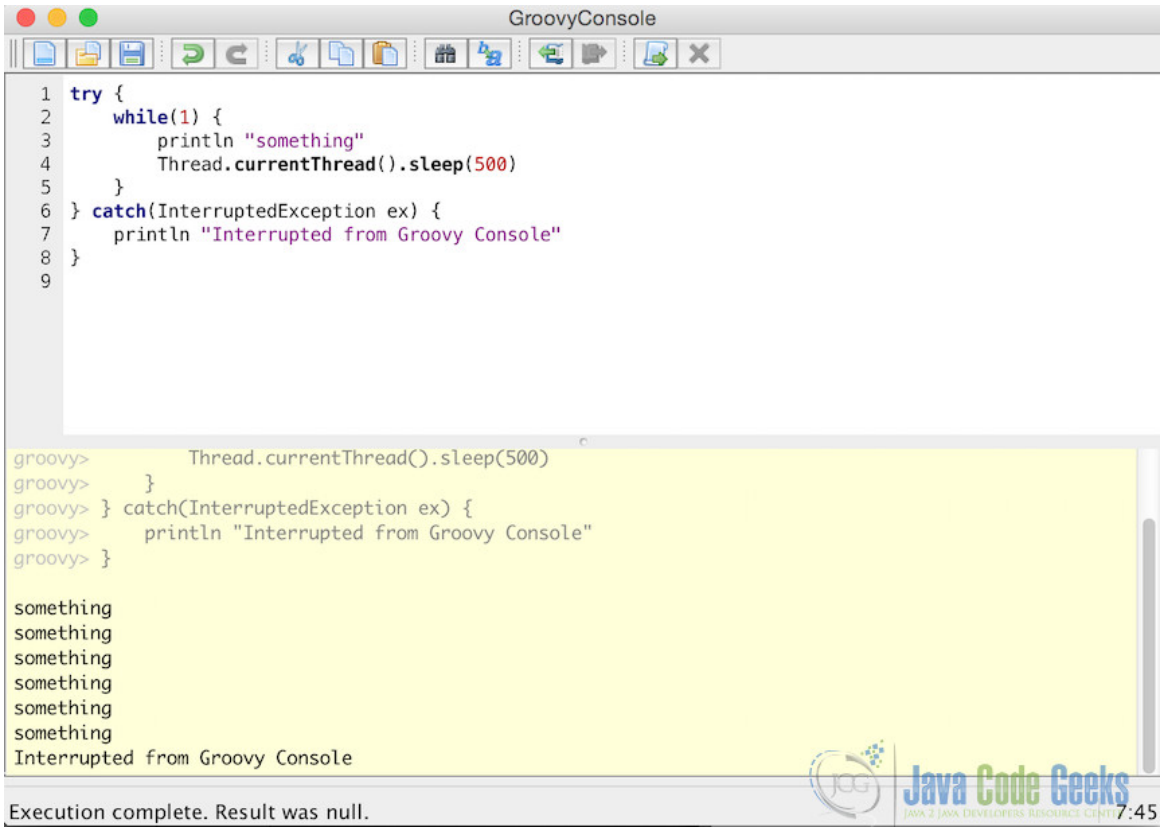

Figure 10.6: Interruption

10.6 Embedding Console

If you want to use Groovy Console within your Java or Groovy application, you can easily embed Groovy Console by using following example.

```
package main.java.javacodegeeks.groovyconsole

import groovy.ui.Console;

class GroovyConsole {

        static main(args) {
                Console console = new Console();
                console.setVariable("name", "John");
                console.setVariable("surname", "Doe");
                console.run();
        }

}
```

When you execute above script, you will see Groovy Console opened and variables `name` and `surname` will be predefined. And you will be able to use that variables. You can see an example below.

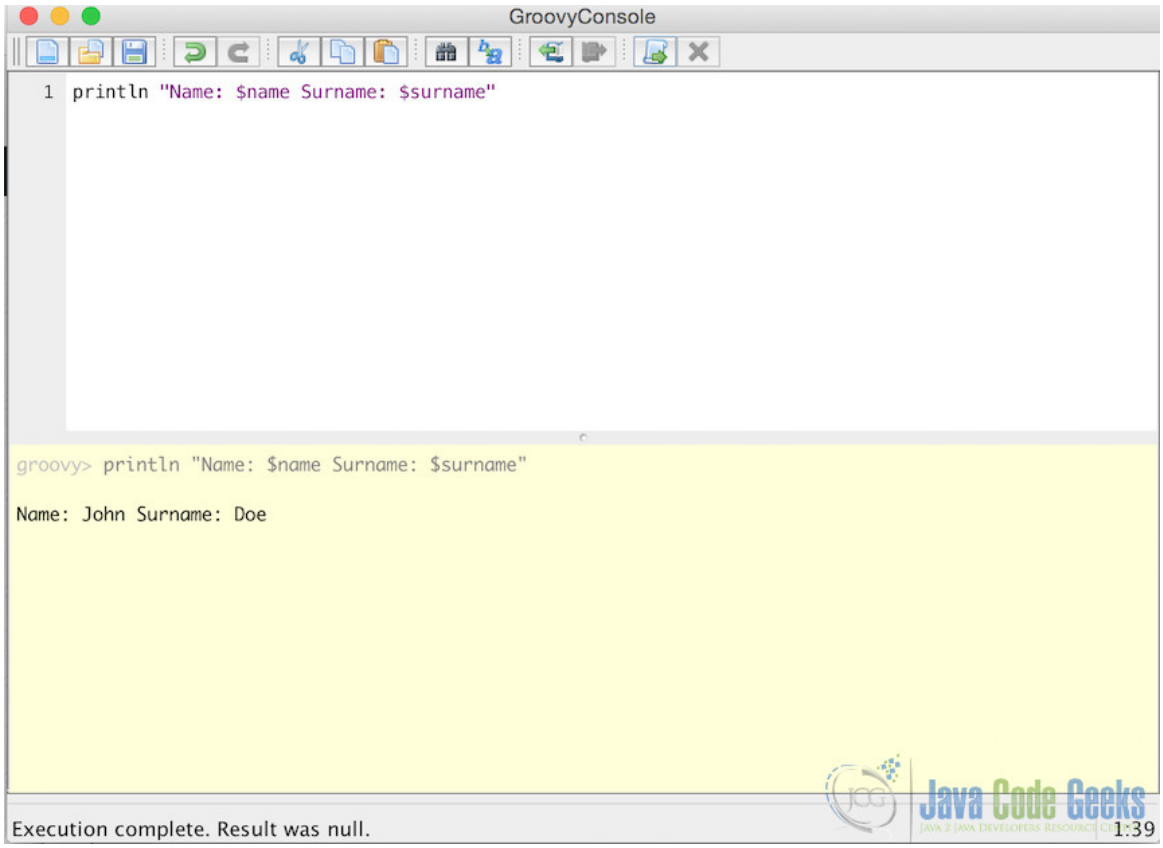

Figure 10.7: Groovy Console Embed Example

As you can see, we are able to use `$name` and `$surname` even if we did not explicitly defined them.

10.7 Conclusion

Groovy Console is an alternative to Groovy Sh or your favourite IDE. You can quickly write your code and execute to see what is going on with entire Groovy code. Also, you can embed Groovy Console to your application to define some variables to console and run it with that variables.

Download

You can download the full source code of the project here: **GroovyConsoleExample**

Chapter 11

Grails tutorial for beginners

Grails is an open source framework based on Groovy and Java. It also supports MVC architecture to for developing web application.

This tutorial will describe more details about Grails and represent a simple web application with Grails.

11.1 Groovy

Groovy is a dynamic object-oriented programming language for the Java platform. It is compiled to Java Virtual Machine (JVM) byte code and integrates with all existing Java classes and libraries.

11.2 Grails MVC

Grails uses Spring MVC as the underlying web application framework to implement the web applications based on Model-View-Controller (MVC) design pattern. It includes three core framework components: Controller, Domain, and Grails Servlet Page. The diagram below shows how different part of Grails works together.

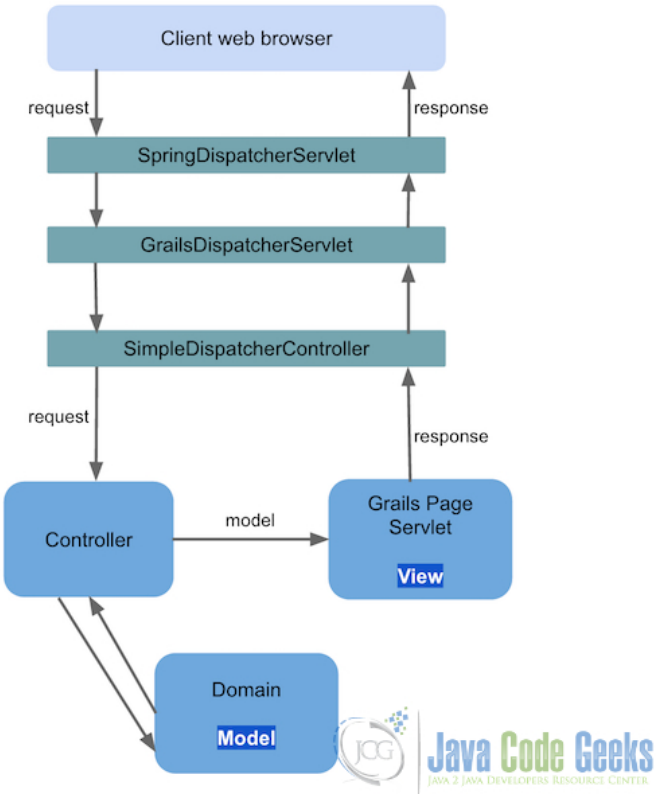

Figure 11.1: grails-architecture

11.2.1 Controller

The GrailsDispatcherServlet uses SpringDispatcherServlet to bootstrap the Grails environment. There is a single Spring MVC controller called SimpleGrailsController which handles all Grails controller requests.The SimpleGrailsController delegates to a class called SimpleGrailsControllerHelper that actually handles the client request and look up a controller for the request. The **Controller** which is responsible for transferring the domain to view, also determines which gsp should render the view.

11.2.2 Domain

Domain stores the data that can be used by controllers and views. Controllers might create a **model** or may be just process them.

11.2.3 Groovy server pages

Groovy server pages creates a **view** for the client response. GSP is responsible for how to display the model to user. It could use grails tags besides any other UI grails plugin.

11.3 Grails Simple Application

This example is implemented in Groovy/Grails Tool Suite and using Grails 2.4.4. Now, lets create a simple application with Grails.

First, create a Grails Project in GGTS. The GGTS includes Grails, so, if you wish to use a different version of Grails you need to install it and update the new Grails path in GGTS.

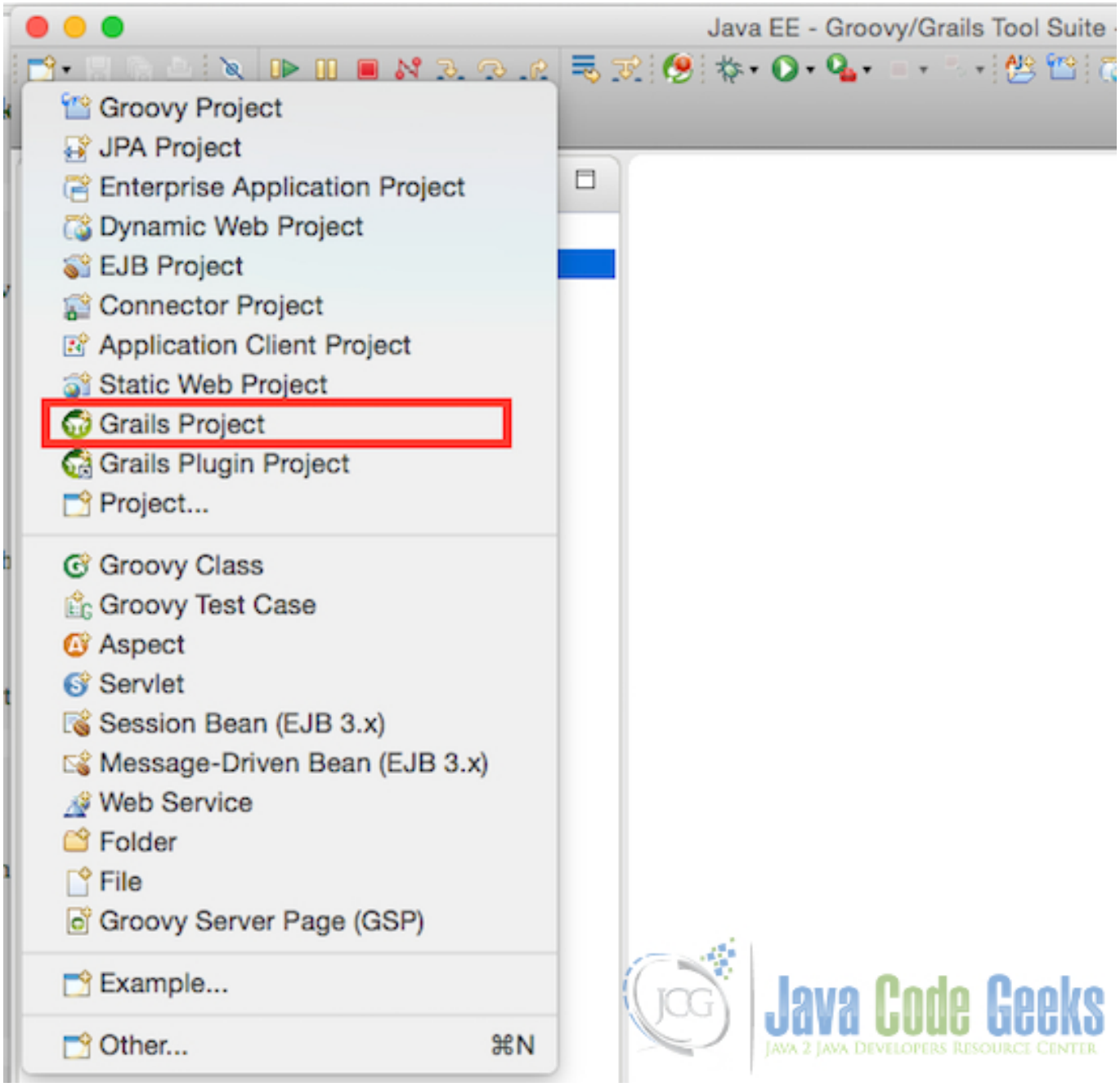

Figure 11.2: create-grails-project

11.3.1 Controller

We only have one controller here. The `UserController` is responsible to handle client requests and return proper result when they submit the form. There is a save method which defined with `def`. When we declare the return type as `def`, it means the method can return any type of object. Inside the method, an instance of User class is created and parameters are set in the domain object and transfered to the view.

UserController.groovy

```
package com.java.code.geeks

class UserController {
```

```
    def index() {
    }

    def save() {
        def user = new User(params)
        user.save()
        render (view: "user", model: [user: user])
    }

}
```

11.3.2 Model

Creating a domain class is very simple in Grails as there is no need to define setter and getter methods. Grails will handles it for the application.

User.groovy

```
package com.java.code.geeks

class User {

        String firstName
        String lastName

}
```

11.3.3 View

The following `gsp` file represents a form to user which includes two items with the same name as model attributes.

index.gsp

```
<!DOCTYPE html>
<html>
        <head>
                <title>Welcome to Grails Tutorial</title>
                <style>
            .form, .text-field, .submit{
                margin: 20px;
            }
                </style>
        </head>
        <body>
                <g:form name="form" controller="user" id="form">
                        <label>First Name: </label><g:textField name="firstName" value="${ ↩
                            firstName}" />
                        <label>Last Name: </label><g:textField name="lastName" value="${ ↩
                            lastName}" />
                        <g:actionSubmit value="Submit" action="save"/>
                </g:form>
        </body>
</html>
```

Grails uses `${ }` to to get model attributes which is `firstName` and `lastName` here. When we use

user.gsp

```
<!DOCTYPE html>
<html>
        <head>
                <title>User page</title>
                <style>
            .user-panel{
                margin: 20px;
            }
                </style>
        </head>
        <body>

                        Welcome ${user.firstName} ${user.lastName}!

        </body>
</html>
```

11.3.4 Run the web application

Now, it is time to run the web application. To run the web application, click on the Grails Command in the Toolbar and type run-app in the popup.

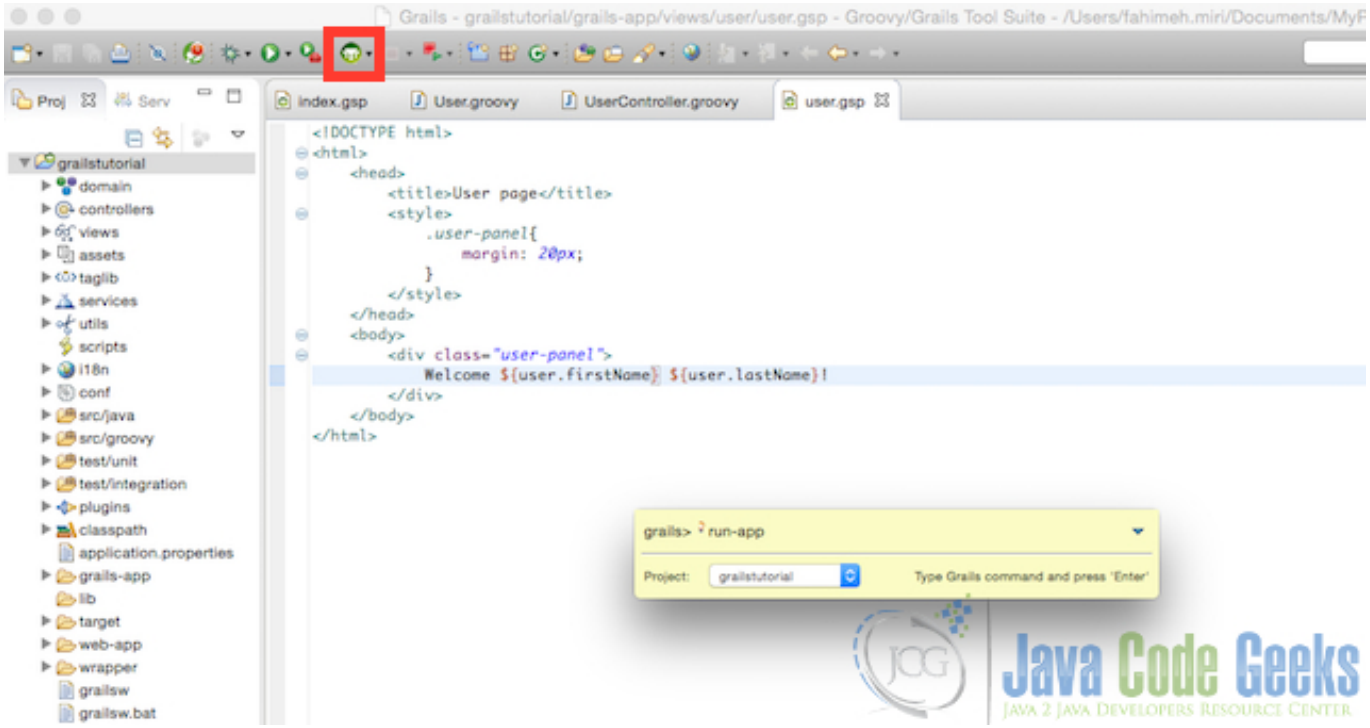

Figure 11.3: runing-grails-application

And here are the pages.

11.3.4.1 Index page

Figure 11.4: grails tutorial - user page

11.3.4.2 User page

Figure 11.5: grails tutorial user page

11.4 Download the source code

This was a Grails Tutorial for beginners.

Download

You can download the full source code of this example here: **Grails tutorial**