



# **JAVA REGEX CRASH COURSE**

**The Ultimate Beginner's Course to  
Learning Java Regular Expressions  
in Under 12 Hours**

**EPROGRAMY**

# JAVA REGEX CRASH COURSE

The Ultimate Beginner's Course to Learning Java  
Regular Expressions in Under 12 Hours

By Eprogramy

---

© Copyright 2015

All rights reserved. No portion of this book may be reproduced -mechanically, electronically, or by any other means, including photocopying- without the permission of the publisher.

## **Disclaimer**

The information provided in this book is designed to provide helpful information on the subjects discussed. The author's books are only meant to provide the reader with the basics knowledge of Java Regular Expressions without any warranties regarding whether the student will, or will not, be able to incorporate and apply all the information provided. Although the writer will make her best effort share his insights. This book, nor any of the author's books constitute a promise that the reader will learn Java Regular Expressions within a certain timeframe. The illustrations are guidance

# Table of Contents

[Check Out the Complete: Programming Series on Amazon](#)

[Introduction: Welcome to the Ultimate Crash Course on Regex](#)

[Chapter 1: Our First Steps in Java Regular Expressions](#)

[What Is Regex?](#)

[Common Uses](#)

[Understanding the Regex Engine](#)

[Algorithms Used in Regex](#)

[Converting NFA to DFA](#)

[Simulating NFA Directly \(DFA/NFA Algorithm\)](#)

[Backtracking](#)

[Chapter 2: Basic Characters Used in Regex](#)

[Common Regex Syntaxes](#)

[Predefined Character Classes](#)

[Boundary Matchers](#)

[Quantifiers](#)

[Greedy Quantifiers](#)

[Reluctant Quantifier](#)

[Possessive Quantifier](#)

[Special Constructs](#)

[Regex Samples](#)

[Basic Expressions](#)

[Grouping](#)

[Matching/Validating](#)

[Extracting/Capturing](#)

[Modifying/Substitution](#)

[Match One of Many Characters](#)

[Using Special Constructs](#)

[Look Ahead & Look Behind](#)

[Positive Look Ahead](#)

[Negative Look Ahead](#)

[Positive Look Behind](#)

[Negative Look Behind](#)

## [Chapter 3: Java Regex Classes](#)

### [Java String Methods](#)

[matches\(\) method](#)

[replaceAll\(\)](#)

[replaceFirst\(\)](#)

[split\(\)](#)

### [Java Util Regex](#)

[Pattern Class Methods](#)

[The Matcher Class Methods](#)

[Pattern Syntax Exception Class](#)

## [Chapter 4: Regex for Username](#)

[Quiz](#)

[Solution](#)

## [Chapter 5: Regex for Password](#)

[Quiz](#)

[Solution](#)

## [Chapter 6: Regex for Email](#)

[Quiz](#)

[Solution](#)

## [Chapter 7: Regex Color Coding](#)

## [Chapter 8: Regex for Image Files](#)

## [Chapter 9: Regex for IP Address](#)

## [Chapter 10: Regex for Time Format](#)

[Time in 12-Hour Format Regex Pattern](#)

[Time in 24-Hour Format Regex Pattern](#)

## [Chapter 11: Regex for Date Format](#)

[Quiz](#)

[Solution](#)

## [Chapter 12: Regex for HTML Tag](#)

[Chapter 13: Regex for Finding HTML Links](#)

[Chapter 14: Regex for URL](#)

[Chapter 15: Regex for Phone Number](#)

[Chapter 16: Regex for International Numbers](#)

[Chapter 17: Other Commonly Used Patterns](#)

[Validate Only Alphanumeric Pattern](#)

[Float Point Numbers](#)

[Deleting Duplicating Words](#)

[Validating SSN](#)

[Quiz](#)

[Solution](#)

[Validating Number Range](#)

[Validating Zip Code](#)

[Quiz](#)

[Solution](#)

[Limit the Length of Text](#)

[Limit the Number of Input Text](#)

[Limit the Number of Lines in Text](#)

[Quiz](#)

[Solution](#)

[Match Something at the Start and/or the End of Line](#)

[Match Whole Words Using Word Boundaries](#)

[Validating the Trademark <sup>TM</sup> Symbol](#)

[Validating Currency Symbol](#)

[Chapter 18: Common Mistakes While Using Regex](#)

[Mistake When Using a Simple String Matching](#)

[Careful Usage of "" Character](#)

[Usage of Character Sequence](#)

[Other Common Mistakes When Using Regex](#)

[Chapter 19: Regex in Eclipse](#)

[Chapter 20: In Closing](#)

[Preview of PYTHON CRASH COURSE - The Ultimate Beginner's Course to Learning Python Programming in Under 12 Hours](#)

[About the Author](#)

*Dedicated to those who want more out of life and are determined to get it.  
Never stop learning,*





## Check Out the Complete Programming Series on Amazon



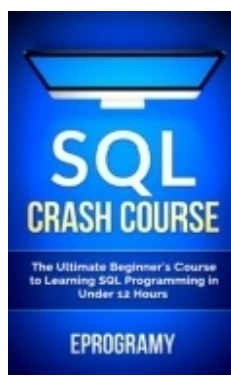
[Java: Crash Course - The Ultimate Beginner's Course to Learning Java Programming in Under 12 Hours](#)



[JavaScript: Crash Course - The Ultimate Beginner's Course to Learning JavaScript Programming in Under 12 Hours](#)



[C: Crash Course - The Ultimate Beginner's Course to Learning C Programming in Under 12 Hours](#)



*SQL: Crash Course - The Ultimate Beginner's Course to Learning SQL Programming in Under 12 Hours*

*These are just some of our books. To check out the complete series Click Here!*

# Introduction

## Welcome to the Ultimate Crash Course on Regex

So, you've decided to learn Java Regular Expressions? Well, congratulations and welcome to your new Crash Course Language! You're going to love it!

In Eprogramy we believe that the foundation of a good education is to provide the tools able to open the doors of the future. It is indisputable that the world has entered an era in which we all have everything at the fingertips. Social needs have changed. Thus, today communication has changed. Communication opens doors, so it is our challenge to learn how to communicate. In Eprogramy we want you to learn how to communicate with a new language: The programming language. This language allows us to use our PC or notebook to create.

Maybe the road is difficult and hard, but we will give the reader the tools to make learning enjoyable and fruitful. We'll show all aspects necessary to learn how to program. From the ABC's to the solution of the most common programming problems and much, much more. Always with the help of numerous examples that will contribute to a better understanding

We know what we do and believe that you are fully capable of incorporating our teachings.

The doors of the future are here. Let's go step by step together.

Let's get started!

Eprogramy Team

# **Chapter 1**

## **Our First Steps in Java Regular Expressions**

# What Is Regex?

In today world, almost all applications are required to process text for search and validation. This process mostly involves complex logic and may result to many complex and hectic codes which is very hard to understand.

A *regular expression* or *REGEX* is explained as a specific kind of text pattern that can be used in many advanced applications and with any programming language. They are used for searching, editing and manipulating data.

For example, using regular expressions we can verify if a particular input string matches a given text pattern or to find out a set of characters from a large batch of characters. They are also used in replacing and re-arranging a block of text or splitting a big chunk of data into smaller subsets. Regular expressions are generally not language specific and follow a similar pattern in most of the programming languages, but with slight variation.

Regular expressions are powerful tools and would reduce the time taken for processing a job, when your program needs to manipulate or extract text on computer. By using them skillfully, regular expressions help us to perform many tasks that wouldn't be feasible at all.

Regular expressions are so useful in real life computing that, the various systems and languages have evolved to provide both a fundamental and protracted standard for the grammar and syntax for usage of modern regular expressions. Also Regular expression processors are found in major of the search engines, search and replace place-holders of various word processors and text editors, and in the command lines of utilities that re used in processing text inputs.

Example:

Let's assume we need to validate the following email.

String emailid = “ [xyz@abc.com](#)”.

Steps required to validate: (For now are not interested in a valid email address but just the format)

Just validate the format with “@” symbol.

There must be a character preceding and succeeding the “@” symbol without space.

The text/ string preceding @ symbol must contain only characters, digit, underscores and hyphen.

There should not be two @ symbols in the email address

There should be a . in the domain name succeeding the @ symbol

If you need to verify the above email id, without regular expressions, a program would need hundreds of lines of code for implementation. We can use the `emailid.indexOf (“@”)` from the String class for the first step of validation and then on keep adding lines of code for the rest of the steps resulting in a tedious job

of coding as well as maintenance would be a nightmare after that.

```
package com.sample.regexsample;
```

```
public class ValidteEmail {
```

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

```
        String str1 = "myname@abc.com";
```

```
        int x = str1.indexOf("@");
```

```
        // returns index of first character of the substring "tutorials"
```

```
        System.out.println("index = " + x);
```

```
    }
```

```
}
```

Output:

Index = 6.

Once after obtaining the index, further calculations are required to find out if there are one to many characters in the domain name, followed by a “.” Symbol and it end with 2 or 3 characters.

This is where regular expression plays an important role .We can accomplish it in just one line of code using the regex patterns. ( We’ll see the example codes in the coming chapters)

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class EmailValidator {
```

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

```
        String input =
```

```
        "^[_A-Za-z0-9-\\+](\\.[_A-Za-z0-9-\\+])*@[A-Za-z0-9-\\+](\\.[A-Za-z0-9-\\+])*(_A-Za-z
```

```
{2,})$"; Regex For email ... let’s discuss more on this in further chapters... Now we need to understand Regex simplifies the process.
```

```
        String emailAddress = "xyz@abc.com";
```

```
        Pattern p = Pattern.compile(input);
```

```
Matcher m = p.matcher(emailAddress);
```

```
System.out.println ("The email 1 matches:" + m.matches());
```

```
}
```

```
}
```

## Common Uses

Regular expressions are majorly used in a wide variety of text processing tasks, and more generally string processing, where the data need not be textual. Common applications include data validation, data scraping, data wrangling, simple parsing, the production of syntax highlighting systems, and many other tasks.

While regular expressions would be useful on Internet search engines, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex.

Some important uses of regular expressions are:

1. URL validation
2. Email Validation
3. Validation of numbers, characters and special characters
4. Internet address
5. Extracting information from text such as code, log files, spreadsheets, or documents.
6. Search and replace Strings

Note:

- Keep in mind, when using regular expression **everything is essentially a character**, and we are writing patterns to match a specific sequence of characters (also known as a string).
- Generally patterns are provided in ASCII, which includes letters, digits, punctuation and other symbols on your keyboard like %#\$@!
- Unicode characters are used to match any type of international text



# Understanding the Regex Engine

One of the most important things to do is to take a look at How a Regex Engine Works Internally because knowing how the regex engine works will help us to craft better regexes more easily. It will also help in understanding quickly why a particular regex does work in the way it was initially expected to do so. This will also save lots of guesswork and head scratching when we need to write more complex regexes.

There are two kinds of regular expression engines:

1. text-directed engines
2. Regex-directed engines.

Mostly all the regex flavors available are based on regex-directed engines. This is because certain very useful features, such as lazy quantifiers and back references, can only be implemented in regex-directed engines.

You can easily find out whether the regex flavor you intend to use has a text-directed or regex-directed engine. If back references and/or lazy quantifiers are available, you can be certain the engine is regex-directed.

You can do the test by applying the regex `<text|text not>` to the string “text not”. If the resulting match is only “text”, the engine is regex-directed. If the result is “text not”, then it is text-directed. The reason behind this is that the regex-directed engine is an early starter.

Before looking into the examples provided, understanding how the regex engine works will enable you to use its full power and help you avoid common mistakes.

The Regex-Directed Engine Always Returns the Leftmost Match.

There are some important points to be noted when working with engines:

- A regex-directed engine will always return the leftmost match, even if a more suitable match could be found later.
- When applying a regex to a string, the engine will start at the first character of the string. It will try all possible variations of the regular expression at the first character. Only if all possibilities have been tried and found to fail, will the engine continue with the second character in the text. Otherwise it will stop there
- Again, it will try all possible variations of the regex, in exactly the same order till it finds a match.
- The result is that the regex-directed engine will return the leftmost match.
- Let us consider the example, searching `<book>` to “He bought a bookshelf for his book.” The engine will try to match the first token in the regex `<b>` to the first character in the match “H”.

This fails. There are no other possible permutations of this regex, because it merely consists of a sequence of literal characters. So the regex engine tries to match the «b» with the “e”. This fails too, as does matching the «b» with the space. Arriving at the 4th character in the match, «b» matches “b”. The engine will then try to match the second token «o» to the 5th character, “o”. This succeeds too. But then, «o» fails to match “u”. At that point, the engine knows the regex cannot be matched starting at the 4th character in the match. So it will continue with the 5th: “a”. Again, «b» fails to match here and the engine carries on. At the 30th character in the match, «b» again matches “b”.

- The engine then proceeds to attempt to match the remainder of the regex at character 30th and finds that «o» matches “o”, again <<o>> matches “o” and «k» matches “k”. The complete regex could be matched starting at character 30. The engine is an early starter to report a match. It will therefore report the first three letters of catfish as a valid match. The engine never proceeds beyond this point to see if there are any “better” matches. In this first example of the engine’s internals, our regex engine simply appears to work like a regular text search routine.

A text-directed engine would have returned the same result too. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works will have a profound impact on the matches it will find. Some of the results may be surprising. But they are always logical and predetermined, once you know how the engine works.

## Algorithms Used in Regex

When using regex, there are at least three different algorithms that decide whether and how a given regular expression matches a string.

### Converting NFA to DFA

This is the first and quickest method. It is based on a result in formal language theory that permits every nondeterministic finite automaton (NFA) to become into a deterministic finite automaton (DFA). The DFA can be explicitly constructed then run on the resulting input string one by one symbol at a time.

### Simulating NFA Directly (DFA/NFA Algorithm)

The next method is to simulate the NFA directly, constructing each DFA state on demand and then discarding it at the next execution step. This method keeps the DFA implicit and avoids increase of the construction cost exponentially, but has an overhead of rising the running cost to  $O(mn)$ . The explicit and implicit approaches are called the DFA algorithm and the NFA algorithm respectively. Adding caching to the NFA algorithm is often called the "lazy DFA" algorithm or just the DFA algorithm without making a distinction.

### Backtracking

The third and final algorithm is to match the pattern against the input string by backtracking. This algorithm is usually called as NFA, but this terminology can be confusing at times. This is used in simple implementations which exhibit when matching against expressions like  $(a|aa)^*b$  that contain both alternation and unbounded quantification. This kind of processing can force the algorithm to consider an exponentially increasing number of sub-cases. This might also lead to security problems called Regular expression Denial of Service.

# **Chapter 2**

## **Basic Characters Used in Regex**

# Common Regex Syntaxes

A regular expression is in the bottom a string patterns that represents text. These descriptions can be applied in several ways. The basic language constructs embrace character classes, quantifiers, and meta-characters. The below section explains the various options we can use to define regular expression.

## 1.1) String Literals

String literals are used to search a particular match in the text. For example, if we are going to search for a text “test” then we can simply write the code like this: Here text and regex both are same.

```
Pattern.matches("test", "test")
```

## 1.2) Character Classes

A character class is used to match a single character in the input text against multiple allowed characters in the character class. a character class has no relation with a class construct or class files in Java.

Examples:

1. [Tt]est would match all the occurrences of String “test” with either lower case or upper case “T”.
2. "WEB@JDOJO.COM" matches the pattern "[ABC]@.", because @ is preceded by B.
3. The string "A@BAND@YEA@U" matches the pattern "[ABC]@" twice even though the string contains three @ signs.
4. The second @ is not a part of any match, because it is preceded by D and not A, B, or C.

Few more samples:

```
Pattern.matches("[pqr]", "abcd"); It would give false as no p,q or r in the text
```

```
Pattern.matches("[pqr]", "r"); Return true as r is found
```

```
Pattern.matches("[pqr]", "pq"); Return false as any one of them can be in text not both.
```

The meta characters [ and ] (left and right brackets) are used to specify a character class inside a regular expression. Sometimes we limit the characters that produce matches to a special set of characters. Here is the sample list of various character classes constructs:

1. **Simple:** consists in a group of characters set up side by side and matches only those characters. Example: [abc] matches characters a, b, and c.

Let’s take a look at the following example:

```
[csw] cave matches c in [csw] with c in cave.
```

2. **Negation:** starts with the ^ meta-character and matches only those characters not in that class.

Example: `[^abc]` matches all characters except a, b, and c.

Let's see a second example:

`[^csw]` cave matches a, v, and e with their counterparts in cave.

3. **Range:** involves all characters starting with the character on the left of a hyphen meta character (-) and finishing with the character on the right of the hyphen meta character, matching only those characters in that range.

Example: `[a-z]` matches all lowercase alphabetic characters.

Let's see another example:

`[a-c]` clown matches c in `[a-c]` with c in clown.

4. **Union:** involves multiple nested character classes and matches all characters that fit to the resulting union.

Example: `[a-d[m-p]]` matches characters a through d and m through p.

Let's see a second example:

`[ab[c-e]]` abcdef matches a, b, c, d, and e with their counterparts in abcdef.

5. **Intersection:** involves characters usual to all nested classes and matches only common characters.

Example: `[a-z&&[d-f]]` matches characters d, e, and f.

Other example:

`[aeiou&&[y]]` party matches y in `[aeiou&&[y]]` with y in party..

6. **Subtraction:** involve all characters less those indicated in nested negation character classes and matches the remaining characters.

Example: `[a-z&&[^m-p]]` matches characters a through l and q through z.

The following command line offers a second example:

`[a-f&&[^a-c]&&[^e]]` abcdefg matches d and f with their counterparts in abcdefg.

# Predefined Character Classes

Character	Explanation
.	Equate a Dot, any character (may or may not match line terminators)
\d	Equate a digit: [0-9]
\D	Equate a non-digit: [^0-9]
\s	Equate a whitespace character: [ \t\n\r\n\b\f]
\S	Equate a non-whitespace character: [^\s]
\w	Equate a word characters
\W	Equate a non-word characters
\A	Equate beginning of string.
\Z	Equate end of string.

However; in Java, you will need to “double escape” these backslashes.

String pattern = "\\d \\D \\W \\w \\S \\s";

For e.g.

Pattern.matches("\\d", "1"); would return true

Pattern.matches("\\D", "z"); return true

Pattern.matches(".p", "qp"); return true, dot(.) represent any character

# Boundary Matchers

Expression	Explanation
^	Equate the beginning of a line.
\$	Equate then end of a line.
\b	Equate a word boundary.
\B	Equate a non-word boundary.
\A	Equate the beginning of the input text.
\G	Equate the end of the previous match
\Z	Equate the end of the input text except the final terminator if any
\z	Equate the end of the input text

For e.g.

Pattern.matches("^Hello\$", "Hello"): return true, Begins and ends with Hello  
Pattern.matches("^Hello\$", "Namaste! Hello"): return false, does not begin with Hello  
Pattern.matches("^Hello\$", "Hello Namaste!"): return false, Does not end with Hello



# Quantifiers

A quantifier is a regular expression construction that ties a numeric value to a pattern. A quantifier will loop to the expression group to its immediate left. That numeric value establish how many times to match a pattern.

We have six main quantifier categories which can be replicate in each of three major categories: greedy, reluctant, and possessive.

## Greedy Quantifiers

The greedy quantifier will match the longest string with the pattern.

Greedy Quantifier	Explanation
X?	Equate once or not at all
X*	Equate zero or more times
X+	Equate one or more times
X{n}	Equate exactly n times
X{n,}	Equate at least n times
X{n,m}	Equate at least n but not more than m times

## Reluctant Quantifier

The reluctant quantifier is used to match with the shortest possible string that matches the pattern

Reluctant Quantifier	Explanation
X??	Equate once or not at all
X*?	Equate zero or more times
X+?	Equate one or more times
X{n}?	Equate exactly n times
X{n,}?	Equate at least n times
X{n,m}?	Equate at least n but not more than m times

## Possessive Quantifier

The possessive quantifier will match the regex with the entire string. It will only match once the string accomplish preset the criteria

Possessive Quantifier	Explanation
X?+	Equate once or not at all
X*+	Equate zero or more times
X++	Equate one or more times

X{n}+	Equate exactly n times
X{n,}+	Equate at least n times
X{n,m}+	Equate at least n but not more than m times

### Sample Code for Quantifiers:

```

package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForQuantifiers {

    public static void main(String[] args) {
        String regex = "a?"; --> to match a in abaa once or not at all
        String regex1 = "a*";    to match a in abaa zero or more times
        String regex2 = "a+";    greedy quantifier to match a in abaa one or more times
        String regex3 = "a{2}";  greedy quantifier to match every aa sequence in aababbbaaab
        String regex4 = "a{2,}"; greedy quantifier to match two or more consecutive as in
                                aababbbaaab
        String regex5 = "a{1,3}"; greedy quantifier to match every a, aa, or aaa in
                                aababbbaaab
        String regex6 = "a+?";   reluctant quantifier to match a in abaa one or more times
        String regex7 = ".*+end"; uses a possessive quantifier to match all characters
                                followed by end in This is the end zero or more times

        String      text = "abaa";
        String      text1 = "abaa";
        String      text2 = "abaa";
        String      text3 = "aababbbaaab";
        String      text4 = "aababbbaaab";
        String      text5 = "aababbbaaab";
        String      text6 = "abaa";
        String      text7 = "This is the end";

        Pattern p1 = Pattern.compile(regex);
        Pattern p2 = Pattern.compile(regex1);
        Pattern p3 = Pattern.compile(regex2);
        Pattern p4 = Pattern.compile(regex3);
        Pattern p5 = Pattern.compile(regex4);
        Pattern p6 = Pattern.compile(regex5);
        Pattern p7 = Pattern.compile(regex6);
        Pattern p8 = Pattern.compile(regex7);

        Matcher m1 = p1.matcher(text);
        Matcher m2 = p2.matcher(text1);
        Matcher m3 = p3.matcher(text2);
    }
}

```

```
Matcher m4 = p4.matcher(text3);
Matcher m5 = p5.matcher(text4);
Matcher m6 = p6.matcher(text5);
Matcher m7 = p7.matcher(text6);
Matcher m8 = p8.matcher(text7);
```

```
while(m1.find())
{
    System.out.println("Start index-->"+m1.start()+"End index-->"+m1.end());
}
```

```
System.out.println("-----");
```

```
while(m2.find())
{
    System.out.println("Start index-->"+m2.start()+"End index-->"+m2.end());
}
```

```
System.out.println("-----");
```

```
while(m3.find())
{
    System.out.println("Start index-->"+m3.start()+"End index-->"+m3.end());
}
```

```
System.out.println("-----");
```

```
while(m4.find())
{
    System.out.println("Start index-->"+m4.start()+"End index-->"+m4.end());
}
```

```
System.out.println("-----");
```

```
while(m5.find())
{
    System.out.println("Start index-->"+m5.start()+"End index-->"+m5.end());
}
```

```
System.out.println("-----");
```

```
while(m6.find())
{
    System.out.println("Start index-->"+m6.start()+"End index-->"+m6.end());
}
```

```
System.out.println("-----");
```

```
while(m7.find())
{
    System.out.println("Start index-->"+m7.start()+"End index-->"+m7.end());
}
```

```
System.out.println("-----");
```

```

        while(m8.find())
        {
            System.out.println("Start index-->" + m8.start() + "End index-->" + m8.end());
        }
    }
}

```

Output:

Start index-->0End index-->1  
 Start index-->1End index-->1  
 Start index-->2End index-->3  
 Start index-->3End index-->4  
 Start index-->4End index-->4

-----

Start index-->0End index-->1  
 Start index-->1End index-->1  
 Start index-->2End index-->4  
 Start index-->4End index-->4

-----

Start index-->0End index-->1  
 Start index-->2End index-->4

-----

Start index-->0End index-->2  
 Start index-->6End index-->8  
 Start index-->8End index-->10

-----

Start index-->0End index-->2  
 Start index-->6End index-->10

-----

Start index-->0End index-->2  
 Start index-->3End index-->4  
 Start index-->6End index-->9  
 Start index-->9End index-->10

-----

Start index-->0End index-->1  
 Start index-->2End index-->3  
 Start index-->3End index-->4

-----

# Special Constructs

Constructs	Explanation
(?<name>X)	X, as a named-capturing group
(?:X)	X, as a non-capturing group
(?idmsuxU-idmsuxU)	Nothing, but turns match flags i d m s u x U on – off
(?idmsux-idmsux:X)	X, as a non-capturing group with the given flags i d m s u x on – off
(?=X)	X, via zero-width positive lookahead
(?!X)	X, via zero-width negative lookahead
(?<=X>)	X, via zero-width positive lookbehind
(?<!X)	X, via zero-width negative lookbehind
(?>X)	X, as an independent, non-capturing group

# Regex Samples

## Basic Expressions

In java every string is a regular expression. Let us consider for example, the string, “I have a car”, is a regex that will match the text, “I have a car”, and will not listen to everything else.

What if we want to be able to let others know more things we have? We can change the word car with a character class expression that will match any character.

```
"I have a \\w+"
```

You can notice; in this pattern both a character class and a quantifier are used. “\w” says match a word character, and “+” says match one or more. So when linked, the pattern says “match one or more word characters.” Now the pattern will equate any word in place of car” E.g. “I have a book” or “I have a dress”.

But if we need to verify “I have a: boat”,it will not match because of the ":" symbol, which is not a word character. If we want the expression to be able to handle this situation, then we need to make a small change.

```
"I have a:? \\w+"
```

Now the expression will let an optional ":" directly then the word ‘a’.

## Grouping

Grouping is an important matter of regex. It is the aptitude to assembly sections of a pattern, and provide different matches.

The following metacharacters are important parts of regex.

|            Alternation (‘or’ statement)

()          Grouping

For instance, in the first example we had a car. What if we know the list of objects we require to inform the world and nothing else? Let’s use a group (), with an ‘or’ meta-character in order to set a list of expressions to permit in our match.

```
"I have a:? (car|book|dress|pen)"
```

The new expression will now equal the starting point of the string “I have a”, an optional ":", and then any one of the expressions in the group, separated by alternators, "|"; any one of the following: ‘car’, ‘book’, ‘dress’, or ‘pen’ would be a match.

"I have a car" matches

"I have a books" matches the 's' is not needed, is ignored

"I have a: car" matches

"I have a- car" doesn't match '- ' is not allowed in our pattern

"I have a: bo" doesn't match all of 'book' is needed

Example

```
package com.sample.regexsample;
```

```
import java.util.regex.Pattern;
```

```
import java.util.regex.Matcher;
```

```
public class DemoForGrouping {
```

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

```
        String text =  
            "John writes about this, and John writes about that," +  
            " and John writes about everything. "  
        ;
```

```
        String patternString1 = "(John)";    Find the group John
```

```
        Pattern pattern = Pattern.compile(patternString1);  
        Matcher matcher = pattern.matcher(text);
```

```
        while(matcher.find()) {  
            System.out.println("found: " + matcher.group(1));
```

```
        }
```

```
    }
```

```
}
```

Output

found: John

found: John

found: John

## Matching/Validating

Using Regex it is possible to find all instances of text that match a certain pattern, and return a Boolean

value if the pattern is found/not found.

Example:

```
package com.sample.regexsample;
```

```
import java.util.ArrayList;  
import java.util.List;  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;
```

```
public class ValidateSSN {
```

```
    public static void main(String[] args) {  
        List<String> input = new ArrayList<String>();  
        input.add("123-45-6789");  
        input.add("9876-5-4321");  
        input.add("987-65-4321 (attack)");  
        input.add("987-65-4321 ");  
        input.add("192-83-7465");
```

```
        String patternString= "^(\d{3}-?\d{2}-?\d{4})$";
```

```
        for (String ssn : input) {  
            if (ssn.matches("^(\d{3}-?\d{2}-?\d{4})$")) {  
                System.out.println("Found good SSN: " + ssn);  
            }  
        }
```

*} This is the match's method of the string class. It can be used for*

*simple validations*

```
        Pattern pattern = Pattern.compile(patternString);
```

```
        for (String ssn : input) {
```

```
            Matcher m = pattern.matcher(ssn);
```

```
            if (m.matches()) {
```

```
                System.out.println("Found good SSN: " + ssn);
```

```
            }
```

*} This is the code using the Pattern and Matcher methods of the regex*

*class .*

```
    }
```

```
}
```

Output:



Found good SSN: 123-45-6789

Found good SSN: 192-83-7465

Explanation of the pattern used in the sample:

`"^(\d{3}-?\d{2}-?\d{4})$"`

`^` *equal the beginning of the line*

`()` *group everything within the parenthesis as group 1*

`\d{n}` *equal n digits, where n is a number equal to or greater than zero*

`-?` *optionally match a dash*

`$` *equal the end of the line*

## Extracting/Capturing

Regular expression can be used for selecting specific values out of a large complex body of text.

Sample:

```
package com.sample.regexsample;
```

```
import java.util.ArrayList;
```

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

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class SampleGrouping {
```

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

```
        String input = "I have a kindle, but I like to read book better.";
```

```
        Pattern p = Pattern.compile("(kindle|read|book|wolf|bear|human)");
```

```
        Matcher m = p.matcher(input);
```

```
        List<String> animals = new ArrayList<String>();
```

```
        while (m.find()) {
```

```
            System.out.println("Found a " + m.group() + ".");
```

```
            animals.add(m.group());
```

```
        }
```

```
    }
```

```
}
```

Output

Found a kindle.

Found a read.

Found a book.

*Explanation of the pattern used:*

**"(kindle|read|book|wolf|bear|human)"**  
**()** *group everything within the parenthesis as group 1*  
**kindle** *match the text 'kindle'*  
**|** *alternation: match any one of the sections of this group*  
**read** *match the text 'read'*  
**... and so on**

## Modifying/Substitution

Using Regular Expression, values in text can be substitute with new values. Let's see, you could change all instances of the word 'userId=', followed by a number, with a "screen" to hide the original text.

Sample:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MaskingDemo {

    public static void main(String[] args) {
        String input =
            "User userId=23421. Some more userId=33432. This is userNum=100";

        Pattern p = Pattern.compile("(userId=)(\\d+)");
        Matcher m = p.matcher(input);

        StringBuffer result = new StringBuffer();
        while (m.find()) {
            System.out.println("Masking: " + m.group(2));
            m.appendReplacement(result, m.group(1) + "***masked***");
        }
        m.appendTail(result);
        System.out.println(result);
    }
}
```

Output

Masking: 23421

Masking: 33432

User userId=\*\*\*masked\*\*\*. Some more userId=\*\*\*masked\*\*\*. This is userNum=100

## Explanation of the pattern

**"(userId=)(\\d+)"**  
**(userId=)** *group everything within the parenthesis as group 1*

`userId=`                      *equal the text 'userId='*  
`(\\d+)`                        *group everything within the parenthesis as group 2*  
`\\d+`                         *equal one or more digits*

## Match One of Many Characters

Using `[]` meta characters and different combinations of characters , we can match one of the many characters present . For example, lets create a regular expression to match all common spellings like cat, bat, rat or mat, so you can find this word in a document.

Example:

```
b|m|r|c[a][ts]
```

Sample Code:

```
package com.sample.regexsample;

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class SampleGrouping {

    public static void main(String[] args) {
        String input = "I have a cat, dog, mat,cas and rat.";

        Pattern p = Pattern.compile("(c|d|r|m)[a][ts]");
        Matcher m = p.matcher(input);

        List<String> animals = new ArrayList<String>();
        while (m.find()) {
            System.out.println("Found a " + m.group() + ".");
            animals.add(m.group());
        }
    }
}
```

Output:

```
Found a cat.
Found a mat.
Found a cas.
Found a rat.
```

## Using Special Constructs

Special constructs are used when we want to find a piece of text when it came before another pattern in that string? Or to find a piece of text that was not followed by another piece of text? Normally, with standard string searching, we require to write a somewhat complex function to perform the exact operation you wanted. This can, however, all be done on one line using regular expressions.

## Look Ahead & Look Behind

Look-ahead and look-behind operations are similar to grouping but these patterns do not capture values. Therefore, using these constructs, no values will be stored for later retrieval, and they do not affect group numbering. Look-ahead operations look forward, starting from their location in the pattern, continuing to the end of the input. Look-behind expressions do not search backwards, but instead start at the beginning of the pattern and continue up to/until the look-behind expression.

E.g.:

We need to ensure in our input string the word “incident” must be present but that the word “theft” should not be found anywhere. We can use a negative look-ahead to ensure that there are no occurrences.

“(?!.\*theft).\*incident.\*”

This expression exhibits the following behavior:

"There was a crime incident"	matches
"The incident involved a theft"	does not match
"The theft was a serious incident"	does not match

Look ahead are of 2 types. Positive look ahead and negative look ahead

## Positive Look Ahead

Positive lookahead is used if you want to match something that should be followed by some pattern or string.

Sample Code for positive look ahead

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForPositiveLookAhead {

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

```
String regex = "(?=^255).*";

Pattern pattern = Pattern.compile(regex);
String candidate = "255.0.0.1";
Matcher matcher = pattern.matcher(candidate);
String ip = "not found";
if (matcher.find())
    ip = matcher.group();
String msg = "ip: " + ip;
System.out.println(msg);
}
```

Output:  
ip: 255.0.0.1

*Explaining the pattern used:*

`"(?=^255).*";`  
`(?=^255)` # This denotes a positive look ahead and should start with 255  
`.*` # Could be followed by anything

## Negative Look Ahead

Negative lookahead is used if you want to match something not followed by something else.

Sample Code for Negative Look – ahead

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForNegativelookAhead {
    public static void main(String args[]) throws Exception {
        String regex = "John (?!Heins)[A-Z]\\w+";
        Pattern pattern = Pattern.compile(regex);

        String candidate = "I think that John Heins ";
        candidate += "is a fictional character. His real name ";
        candidate += "might be John Jackson, John Westling, ";
        candidate += "or John Holmes for all we know.";

        Matcher matcher = pattern.matcher(candidate);

        String tmp = null;

        while (matcher.find()) {
```

```

        tmp = matcher.group();
        System.out.println("MATCH:" + tmp);
    }
}

```

```

}

```

Output

MATCH:John Jackson

MATCH:John Holmes

**Explaining the pattern:**

*"John (?!Heins)[A-Z]\\w+";*

*John # the pattern should start with John*

*(?!Heins) # It should not contain the word Heins*

*(A-Z)\\w+ # John Should be followed by a word that begins with capital letter followed by any word.*

*In the above example though we have John Westling, it doesn't match because it does not begin with a capital letter*

## Positive Look Behind

Positive lookbehind is used if you want to match something that should be preceded by some pattern or string

Sample for positive look-behind

```

package com.sample.regexsample;

```

```

import java.util.regex.Matcher;

```

```

import java.util.regex.Pattern;

```

```

public class DemoForPositiveLookBehind {

```

```

    public static void main(String[] args) {
        String reg = "(?<=http://)\\S+";
        Pattern p = Pattern.compile(reg);
        String str = "http://www.a.com. My name";
        Matcher m = p.matcher(str);

```

```

        while (m.find()) {
            String msg = ":" + m.group() + ":";
            System.out.println(msg);
        }
    }
}

```

Output  
:www.a.com.:

### *Explaining the pattern*

*(?<+http://)\S+*

*# look – behind starting with http:// until a white space is encountered. i.e. the string to be checked should be preceded by http://*

## Negative Look Behind

Negative lookbehind is used if you want to match something not preceded by something else.

Sample for negative look-behind

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForNegativelookBehind {
```

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

```
        Pattern p = Pattern.compile("(?!Y.{0,40})X");
```

```
String txt = "Y less than 40 before X"; // Since you want 'not' then this should fail to match.
```

```
Matcher m = p.matcher(txt);
```

```
System.out.println(m.find());
```

```
txt = "Y less ddddddddddddddddddddddd than 40 before X"; // Since you want 'not' this should match
```

```
m.reset(txt);
```

```
System.out.println(m.find());
```

```
    }
```

```
}
```

Output:

false

true

### *Explaining the pattern:*

*"(?!Y.{0,40})X"*

*# Find a pattern X if a Y doesn't exist between 0 to N characters before the X.*

## Chapter 3

# Java Regex Classes

In Java when we work on complex projects, we encounter many inputs that commonly requires code to equal text against patterns. It is this capability that is frequently used in email header validation, text searches, and custom text creation from generic text and so on.



# Java String Methods

Before moving in to the `java.util.regex` package, let us look at few of the methods provided by the `java String` class for using regular Expressions. These methods can be used for fewer simple validations. But if we are going to use regular expression very often, these methods would be a performance overhead

.

## matches() method

`sampleString.matches("regex")` returns true or false depending whether the string can be matched entirely by the regular expression. An important point to be remembered is that `String.matches()` only returns true if the entire string can be matched.

Example:

```
package com.sample.regexsample;
```

```
public class DemoForStringMatches {
```

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

```
        // Prepare a regular expression to represent a pattern
```

```
        String regex = ".@";
```

```
        // Try matching many strings against the regular expression
```

```
        DemoForStringMatches.matchIt("a@k", regex);
```

```
        DemoForStringMatches.matchIt("webmaster@jdojo.com", regex);
```

```
        DemoForStringMatches.matchIt("r@j", regex);
```

```
        DemoForStringMatches.matchIt("a%N", regex);
```

```
        DemoForStringMatches.matchIt(".@", regex);
```

```
    }
```

```
    public static void matchIt(String str, String regex) {
```

```
        // Test for pattern match
```

```
        if (str.matches(regex)) {    Uses the match's method of String class
```

```
            System.out.println(str + " matches the regex " + regex);
```

```
        }
```

```
        else {
```

```
            System.out.println(str + " does not match the regex " + regex);
```

```
        }
```

```
    }
```

```
}
```

Output:

a@k matches the regex .@.

webmaster@jdojo.com does not match the regex .@.

r@j matches the regex .@.

a%N does not match the regex .@.

.@. matches the regex .@.

## replaceAll()

sampleString.replaceAll("regex", "replacement") replaces all regex matches inside the string with the replacement string you specified. All parts of the string that match the regex are replaced.

Example:

```
package com.sample.regexsample;

public class DemoForReplaceAll {

    public static void main(String[] args) {
        // Prepare a regular expression to represent a pattern
        String regex = ".@.";
        String replacementString = "****";
        // Try matching many strings against the regular expression
        DemoForReplaceAll.replaceMatch("webd@koj", regex, replacementString);

    }
    public static void replaceMatch(String str, String regex, String replacementString ) {
        // Test for pattern match
        String newStr = str.replaceAll(regex, replacementString);
        System.out.println(regex + " replaced " + newStr);
    }

}
```

Output:

.@. replaced web\*\*\*oj

## replaceFirst()

The replaceFirst() method replaces the first occurrence of the match with the replacementString. It returns the new string after replacement.

Sample:

```
package com.sample.regexsample;

public class DemoForReplaceFirst {

    public static void main(String[] args) {
        // Prepare a regular expression to represent a pattern
        String regex = ".@.";
        String replacementString = "****";
```

```
// Try matching many strings against the regular expression
DemoForReplaceFirst.replaceMatch("A@BandH@G", regex, replacementString);

}
public static void replaceMatch(String str, String regex, String replacementString ) {
// Test for pattern match
String newStr = str.replaceFirst(regex, replacementString);
System.out.println(regex + " replaced " + newStr);
}

}
```

Output  
 .@. replaced \*\*\*andH@G

## split()

sampleString.split("regex") splits the string at each regex match. The return of this method will be an array of strings, in which the elements are the original string between two regex matches.

Example:

```
package com.sample.regexsample;

public class DemoForSplit {

    public static void main(String[] args) {
        String str = "one-two-three";
        String[] temp;

        /* delimiter */
        String delimiter = "-";
        /* given string will be split by the argument delimiter provided. */
        temp = str.split(delimiter);
        /* print substrings */
        for(int i =0; i < temp.length ; i++)
            System.out.println(temp[i]);

    }

}
```

Output:  
 one  
 two  
 three

# Java Util Regex

The java.util.regex package primarily consists of the following three classes: (Source: Oracle Java documentation, [www.docs.oracle.com](http://www.docs.oracle.com))

- **Pattern Class:** A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
- **Matcher Class:** A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the matcher method on a Pattern object.
- **PatternSyntaxException:** A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

## Sample:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternSample {

    public static void main(String[] args) {
        Pattern pattern = Pattern.compile(".ab.");
        Matcher matcher = pattern.matcher("WabY");
        Matcher m1 = pattern.matcher("WxxY");
        System.out.println("Input String matches regex - "+matcher.matches());
        System.out.println("Input String matches regex - "+m1.matches());
    }
}
```

Output of the above code is as follows:

```
Input String matches regex - true
Input String matches regex - false
```

## Sample Code Explanation:

In the above example, the compile() method of the Pattern class is used to compile a regular expression that needs to be verified. Then the string to be validated is passed using the pattern.matcher() method for verification against the compiled regex.

The regex pattern used in the above example is “.ab.” which says there should be one character before ab

and one after it and “ab” should be present in between them. Any string that follows this specific pattern will return true for matching. Hence in the above example, the first patter “WabY” matches and returns true where as “WxxY” does not match and returns false.

## Pattern Class Methods

### Pattern.compile()

Pattern.compile() method is used for compiling a regular expression if we require to perform frequent search occurrences of an expression or if we need to add any extra parameters while performing a search (e.g. Like case insensitive search).

This is how this method can be used for this case.

```
String content = "This is a my regex";  
String patternString = ".*reGeX";  
Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
```

In this code sample we have used a flag Pattern.CASE\_INSENSITIVE for case insensitive search, there are several other flags that can be used for different purposes. The table below gives the list of flags that can be used along with their embedded characters.

FLAGS	EXPLANATION
Pattern.CANON_EQ	Used for enabling canonical equivalence. This flag is used when we need to match two expression only if there canonical decomposition matches.
Pattern.CASE_INSENSITIVE	Used when performing searches or matching without considering the case of the expression
Pattern.COMMENTS	Allows whitespace and comments in the pattern. By using this flag, white spaces and comment starting with # can be ignored in the expression.
Pattern.DOTALL	Used for enabling dotall mode. In this mode a “.” represents any character including a line terminator.
Pattern.LITERAL	Used of enabling the literal parsing of the he given expression. When this flag is used, the entire expression given for validation is treated as literal characters and not as a whole string
Pattern.MULTILINE	Used for enabling multi line mode. By default the meta characters ^ and \$ matches the beginning and ending of an expression. But when this flag is used,

Pattern.UNICODE_CASE	these characters match before and after the line terminator ie. “.” Symbol . This flag is used in combination with CASE_INSENSITIVE flag when we want to compare Unicode cases ( Normally only ASCII characters are compared)
Pattern.UNIX_LINES	This flag enables the Unix lines mode. In this “/n” is considered as line terminator and not the “.” Character.

These flags can be embedded along with the regular expression as a character itself. The following table gives the embedded character equivalent of the flags.

FLAGS	EMBEDDED EQUIVALENT
Pattern.CANON_EQ	None
Pattern.CASE_INSENSITIVE	(?i)
Pattern.COMMENTS	(?x)
Pattern.MULTILINE	(?m)
Pattern.DOTALL	(?s)
Pattern.LITERAL	None
Pattern.UNICODE_CASE	(?u)
Pattern.UNIX_LINES	(?d)

### *Pattern.matches()*

This method is the simplest and easiest way available to search a String in a Text using regex. For example, consider the code snippet given below:

```
package com.sample.regexsample;
```

```
import java.util.regex.Pattern;
```

```
public class PatternSample {
```

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

```
        String content = "This is my regex sample";
        String patternString = ".*regex.*";
        boolean isMatch = Pattern.matches(patternString, content);
        System.out.println("The text contains 'tutorial'? " + isMatch);
```

```
    }
}
```

Output:  
The text contains 'regex': true

As given in the example, we have used matches() method of Pattern class to search the pattern in the given text. The pattern .\*regex.\* allows zero or more characters at the beginning and end of the String “regex”.

Drawbacks: Using this method we can search a single occurrence of a pattern in a text. But if we need to check for matching of multiple occurrences we should use the Pattern.compile() method (discussed in the previous section) in conjunction with the Pattern.matcher() method .

### *Pattern.matcher() method*

This method is used to return the instance of the Matcher class and is used to perform validations multiple times or when grouping is used.

Consider the example given below:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternSample {

    public static void main(String[] args) {

        String content = "This is my regex sample";
        String patternString = ".*rEgEX.*";
        Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher(content);
        boolean isMatched = matcher.matches();
        System.out.println("Is it a Match?" + isMatched);

    }
}
```

Output:  
Is it a Match?true

### *Multiple grouping sample*

In the below code provided, the compile and matcher methods are used to find string from a grouped expression.

```
package com.sample.regexsample;
```

```
import java.util.ArrayList;  
import java.util.List;  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;
```

```
public class PatternSample {  
  
    public static void main(String[] args) {  
  
        String input = "I have a cat, but I like my dog better.";  
        String patternString = "(mouse|cat|dog|wolf|bear|human)";  
  
        Pattern p = Pattern.compile(patternString);  
        Matcher m = p.matcher(input);  
  
        List<String> animals = new ArrayList<String>();  
        while (m.find()) {  
            System.out.println("Found a " + m.group() + ".");  
            animals.add(m.group());  
        }  
    }  
}
```

Output:

Found a cat.

Found a dog.

Note : m.find()- What is this? Well we would discuss that in the next part when we discuss the Matcher class.

### ***Pattern.split()***

This method is used to split a given text into multiple string based on a delimiter ( specified using regex). Sample code below shows how this method works.

```
package com.sample.regexsample;
```

```
import java.util.ArrayList;  
import java.util.List;  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;
```



```

public class PatternSample {

    public static void main(String[] args) {

        String text = "ItIsRegexSample.ItIsProvidedByMe";
        // Pattern for delimiter
        String patternString = "is";
        Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
        String[] myStrings = pattern.split(text);
        for(String temp: myStrings){
            System.out.println(temp);
        }
        System.out.println("Number of split strings: "+myStrings.length);
    }
}

```

In the above code snippet , “is” is used as a delimiter to split the text “ItIsRegexSample.ItIsProvidedByMe”. The output of the above sample is given below.

Output

```

It
RegexSample.It
ProvidedByMe
Number of split strings: 3

```

## The Matcher Class Methods

In the previous section when discussing the matcher() method of the Pattern class, we created an instance of the Matcher class. It is used for multiple searches and grouped searches. Code sample is given below for reference

### Creating a Matcher instance

```

String content = "Some text";
String patternString = ".*somesstring.*";
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(content);

```

The pattern.matcher() method returns an instance of the Matcher class. Using this instance we can access the other methods of the Matcher class.

### Matches() method

This method is used to match the regular expression against the whole string provided to the Pattern.matcher() method while creating Matcher instance.

Code Snippet:

```
Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(content);
boolean isMatched = matcher.matches();
```

### *lookingAt()*

This method is similar to matches() method ,except that it matches the regular expression only against the starting of the text, while matches() search in the whole text.

Code snippet:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternSample {

    public static void main(String[] args) {

        String REGEX = "Saa";
        String INPUT = "SaaaaaaaS";

        Pattern pattern = Pattern.compile(REGEX);
        Matcher matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "+REGEX);
        System.out.println("Current INPUT is: "+INPUT);

        System.out.println("lookingAt(): "+matcher.lookingAt());
        System.out.println("matches(): "+matcher.matches());
    }
}
```

Output:

```
Current REGEX is: Saa
Current INPUT is: SaaaaaaaS
lookingAt(): true
matches(): false
```

In the above example, the lookingAt() method returns true whereas the matches() method returns false because, it matches the whole string, whereas the other method just compares the beginning.

## *find()*

This method is used for searching the occurrences of the regular expressions in the text. It is frequently used when we are searching for multiple occurrences of a string within a huge chunk of data.

Code snippet:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternSample {

    public static void main(String[] args) {

        String REGEX = "too";
        String INPUT = "too";

        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT); // get a matcher object

        while(m.find()) {
            System.out.println("start(): "+m.start());
            System.out.println("end(): "+m.end());
        }
    }
}
```

Output:  
start(): 0  
end(): 3

## *find(int start)*

This method attempts to find the subsequence of the input sequence that equals the pattern, starting at the specified index.

### 1) start() and end()

Both these methods are generally used in combination with the find() method. The start method is used for getting the start index and the end method for getting the end index of a string or match that is being

searched using the find() method. This method returns int value.

Code snippet:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternSample {

    public static void main(String[] args) {

        String content = "I have a big and beautiful house and a good car";

        String string = "and";
        Pattern pattern = Pattern.compile(string);
        Matcher matcher = pattern.matcher(content);

        while(matcher.find()) {
            System.out.println("Found at: "+ matcher.start()
                               +
                               " - " + matcher.end());
        }
    }
}
```

Output:

Found at: 13 - 16

Found at: 33 - 36

## 2) start(int group) and end(int group)

These are overloaded methods of start and end. These methods are used when grouping is used and have to find a start or end index of an expression from a particular group.

Code snippet:

```
package com.sample.regexsample;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternSample {

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

```
String REGEX = "(cat)(\\scattie\\s)";  
String INPUT = "cat cat cat cattie cat";
```

```
Pattern p = Pattern.compile(REGEX);  
Matcher m = p.matcher(INPUT); // get a matcher object  
int count = 0;  
  
while(m.find()) {  
    count++;  
    System.out.println("start(1): "+m.start(1));  
    System.out.println("end(1): "+m.end(1));  
    System.out.println("start(2): "+m.start(2));  
    System.out.println("end(2): "+m.end(2));  
}  
}
```

```
}  
Output:  
start(1): 8  
end(1): 11  
start(2): 11  
end(2): 19
```

### *appendReplacement(StringBuffer sb, String replacement)*

It reads characters from the input sequence, starting at the append position, and appends them to the given string buffer. After this it appends the given replacement string to the string buffer to create a new text.

Code Snippet:

```
package com.sample.regexsample;  
  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;  
  
public class PatternSample {  
  
    public static void main(String[] args) {  
  
        Pattern p = Pattern.compile("dog");  
        Matcher m = p.matcher("one dog two dogs in the house");  
        StringBuffer sb = new StringBuffer();  
        while (m.find()) {  
            m.appendReplacement(sb, "rabbit");  
        }  
        m.appendTail(sb);  
    }  
}
```

```
System.out.println(sb.toString());
```

```
    }  
}
```

Output:  
one rabbit two rabbits in the house

### *appendTail(StringBuffer sb)*

This method is used in conjunction with the appendReplacement() method to form the new string after replacement.

### *replaceAll(String replacement)*

This method substitute every subset of the input string that matches the given pattern with the replacement string provided.

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class PatternSample {
```

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

```
        String input = "aaahorsebbbhorsecccchorse";
```

```
        String patternString = "horse";
```

```
        String output="";
```

```
        Pattern p = Pattern.compile(patternString);
```

```
        Matcher m = p.matcher(input);
```

```
        output = m.replaceAll("dog");
```

```
        System.out.println("Input:"+input);
```

```
        System.out.println("Output:"+output);
```

```
    }
```

```
}
```

Output  
Input:aaahorsebbbhorsecccchorse  
Output:aaadogbbbdogcccdog

### *replaceFirst(String replacement)*

This method replaces the first subset of the input string that matches the given pattern with the replacement string provided.

```
package com.sample.regexsample;
```

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternSample {

    public static void main(String[] args) {

        String input = "aaahorsebbbhorseccchorse";
        String patternString = "horse";
        String output="";

        Pattern p = Pattern.compile(patternString);
        Matcher m = p.matcher(input);
        output = m.replaceFirst("dog");
        System.out.println("Input:"+input);
        System.out.println("Output:"+output);
    }
}

```

Output  
Input:aaahorsebbbhorseccchorse  
Output:aaadogbbbhorseccchorse

### *quoteReplacement(String s)*

This method return a literal replacement String for the given input String. This literal replacement string which can be used as a literal replacement in the appendReplacement method of the Matcher class.

## Pattern Syntax Exception Class

A PatternSyntaxException is an unchecked exception that means a syntax error in a regular expression pattern. The PatternSyntaxException class gives us the next methods to solve what is inaccurate.

1     public String getDescription()

Provides the error description

2     public int getIndex()

Retrieves index of the error.

3     public String getPattern()

Recover the regular expression pattern which resulted in the error.

4     public String getMessage()

Returns a multi-line string containing the description of the syntax error and its index, the incorrect regex pattern, and a visual indication of the error index within the pattern.

Sample Code:

```
package com.sample.regexsample;

import java.io.Console;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class DemoForPatternSyntaxexception {

    public static void main(String[] args){
        Pattern pattern = null;
        Matcher matcher = null;

        String regex = "?i)foo";
        while (true) {
            try{
                pattern =
                    Pattern.compile(regex);

                matcher =
                    pattern.matcher("FOO");
            }
            catch(PatternSyntaxException pse){
                System.out.println("There is a problem" +
                    " with the regular expression!%n");
                System.out.println("The pattern in question is: %s%n"+pse.getPattern());
                System.out.println("The description is: %s%n"+pse.getDescription());
                System.out.println("The message is: %s%n"+pse.getMessage());
                System.out.println("The index is: %s%n"+pse.getIndex());
                System.exit(0);
            }
            boolean found = false;
            while (matcher.find()) {
                System.out.println("I found the text" +
                    "\"%s\" starting at " +
                    "index %d and ending at index %d.%n"+
                    matcher.group()+
                    matcher.start()+
                    matcher.end());
                found = true;
            }
            if(!found){
```



```
System.out.println("No match found.%n");
```

```
}
```

```
}
```

```
}
```

```
}
```

Output:

There is a problem with the regular expression!%n

The pattern in question is: %s%n?i)foo

The description is: %s%nDangling meta character '?'

The message is: %s%nDangling meta character '?' near index 0

?i)foo

^

The index is: %s%n0

***From this output, we can appreciate that the syntax error is a hanging meta character (the question mark) at index 0. A missing opening parenthesis is the guilty character.***

## Chapter 4

# Regex for Username

This syntax explains how regular expression can be used for validating a user name. This is the common pattern used across various websites and applications.

Syntax

```
^[a-z0-9_-]{3,15}$
```

Explanation:

^	# Start of the line
[a-z0-9_-]	# Equal characters and symbols in the list, a-z, 0-9, underscore, hyphen
{3,15}	# Length at least 3 characters and maximum length of 15
\$	# End of the line

The whole combination means, 3 to 15 characters with any lower case character, digit or special symbol “\_-” only. In case we need to add upper case include A-Z in the group. Also if we need to increase max character, change 15 to the value required.

Sample Code for user Name:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class UserNameValidator {

    public static void main(String[] args) {
        String input =
            "^[a-z0-9_-]{3,15}$";
        String userName ="Karen_3";
        String userName1="MyName$";

        Pattern p = Pattern.compile(input);
        Matcher m = p.matcher(userName);
        Matcher m2 = p.matcher(userName1);
        System.out.println ("The user name 1 matches:"+ m.matches());
        System.out.println ("The user name 2 matches:"+ m2.matches());
    }
}
```

Output:

The user name 1 matches:true

The user name 2 matches:false

## Quiz

Write class and test case for validating the following user names. Use a string array or ArrayList to pass the entire set of values:

1. myname34
2. myname\_2015
3. myname-2015
4. my\_name-2020
5. my
6. my@name
7. myname\_1234\_mynameAgain

## Solution

CLASS:

```
package com.sample.RegexExamples;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class UserNameValidator {
```

```
private Pattern pattern;
```

```
private Matcher matcher;
```

```
private static final String USERNAME_PATTERN = "^(?i)[a-z0-9_-]{3,15}$";
```

- A simple variation from the pattern given in the explanation is the addition of (?i) which checks the username with case sensitivity.

```
public UserNameValidator(){
```

```
    pattern = Pattern.compile(USERNAME_PATTERN);
```

```
}
```

```
/**
```

```
 * Validate username with regular expression
```

```
 * @param username username for validation
```

```
 * @return true valid username, false invalid username
```

```
 */
```

```
public boolean validate(final String username){
```

```
    matcher = pattern.matcher(username);
```

```
    return matcher.matches();
```

```
}
```

```
}
```

UNIT TEST CASE:

```
package com.sample.RegexTest;
```

```
import static org.junit.Assert.*;
```

```
import org.junit.Assert;
```

```
import org.junit.Before;
```

```
import org.junit.Test;
```

```

import com.sample.RegexExamples.UserNameValidator;

public class UserNameValidatorTest {

    private UserNameValidator usernameValidator;

    @Before
    public void initData(){
        usernameValidator = new UserNameValidator();
    }

    @Test
    public void ValidUsernameTest() {

        String Username[] = new String[] {
            "myname34", "myname_2015", "myname-2015", "my3-4_Name", "LastName_25", "LAsNaMe-
78"
        };

        for(String temp : Username){
            boolean valid = usernameValidator.validate(temp);
            System.out.println("Username is valid : " + temp + " , " + valid);
            Assert.assertEquals(true, valid);
        }
    }

    @Test
    public void InvalidUsernameTest() {

        String[] UserName1 = new String[] {
            "my", "my@LastName", "myName123456789_-
", "lastName@1234234253", "aaaaaaaaaaaaavvvcwerwe"
        };

        for(String temp : UserName1){
            boolean valid = usernameValidator.validate(temp);
            System.out.println("username is valid : " + temp + " , " + valid);
            Assert.assertEquals(false, valid);
        }
    }
}

```

Output:

Username is valid : myname34 , true

Username is valid : myname\_2015 , true

Username is valid : myname-2015 , true  
Username is valid : my3-4\_Name , true  
Username is valid : LastName\_25 , true  
Username is valid : LAsTNaMe-78 , true  
username is valid : my , false  
username is valid : my@LastName , false  
username is valid : myName123456789\_- , false  
username is valid : lastName@1234234253 , false  
username is valid : aaaaaaaaaaaaaavvvcwerwe , false

# Chapter 5

## Regex for Password

Syntax:

```
((?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%]).{6,20})
```

Explanation:

```
(  
    (?=.*\d)           # must contains one digit from 0-9  
    (?=.*[a-z])        # must contains one lowercase characters  
    (?=.*[A-Z])        # must contains one uppercase characters  
    (?=.*[@#$%])       # must contains one special symbols in the list "@#$%"  
    .                  # equal anything with previous condition checking  
    {6,20}             # length at least 6 characters and maximum of 20  
)
```

?= – means apply the assertion condition, meaningless by itself, always work with other combination

The entire regular expression combination means, 6 to 20 characters string with at least one digit, one upper case letter, one lower case letter and one special symbol (“@#\$%”).

Sample Code:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class PasswordValidator {
```

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

```
        String input =
```

```
        "((?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%]).{6,20})";
```

```
        String userName = " myname1A@";
```

```
        String userName1="mY1A@";
```

```
        Pattern p = Pattern.compile(input);
```

```
        Matcher m = p.matcher(userName);
```

```
        Matcher m2 = p.matcher(userName1);
```

```
        System.out.println ("The password 1 matches:" + m.matches());
```

```
        System.out.println ("The password 2 matches:" + m2.matches());
```

```
    }
```

```
}
```



Output:

The password 1 matches:true

The password 2 matches:false

## Quiz

Write class and test case for validating the following user names.

1. mynameA@34
2. mY1A@
3. myname12@
4. myName12\*
5. lastName\$\$
6. LASTNAME12\$
7. lastName1A@
8. myNaME@12\$

# Solution

CODE:

```
package com.sample.RegexExamples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PasswordValidator {

    private Pattern pattern;
    private Matcher matcher;

    private static final String PASSWORD_PATTERN =
        "((?=.*\\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%]).{6,20})";

    public PasswordValidator(){
        pattern = Pattern.compile(PASSWORD_PATTERN);
    }

    /**
     * Validate password with regular expression
     * @param password password for validation
     * @return true valid password, false invalid password
     */
    public boolean validate(final String password){

        matcher = pattern.matcher(password);
        return matcher.matches();

    }

}
```

UNIT TEST:

```
package com.sample.RegexTest;

import static org.junit.Assert.*;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import com.sample.RegexExamples.PasswordValidator;
```

```

public class PasswordValidatorTest {

    private PasswordValidator passwordValidator;

    @Before
    public void initData(){
        passwordValidator = new PasswordValidator();
    }

    @Test
    public void ValidPasswordTest() {

        String[] password = new String[]
{"mynameA@34","myName12@","myNaME@12$","lastName2$"};
        for(String temp : password){
            boolean valid = passwordValidator.validate(temp);
            System.out.println("Password is valid : " + temp + " , " + valid);
            Assert.assertEquals(true, valid);
        }

    }

    @Test
    public void InvalidPasswordTest() {

        String[] password = new String[]{"mY1A@","myname12@
","myName12*","LASTNAME12$","LASTName$"};
        for(String temp : password){
            boolean valid = passwordValidator.validate(temp);
            System.out.println("Password is valid : " + temp + " , " + valid);
            Assert.assertEquals(false, valid);
        }

    }

}

```

Output:

```

Password is valid : mY1A@ , false
Password is valid : myname12@ , false
Password is valid : myName12* , false
Password is valid : LASTNAME12$ , false
Password is valid : LASTName$ , false
Password is valid : mynameA@34 , true
Password is valid : myName12@ , true
Password is valid : myNaME@12$ , true

```

Password is valid : lastName2\$, true

# Chapter 6

## Regex for Email

### Email Regular Expression Pattern

```
^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*  
@[A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$;
```

### Explanation

^	#start of the line
[_A-Za-z0-9-\\+]+	# must start with string in the bracket [ ], must contains one or more (+)
(	# start of group #1
\\.[_A-Za-z0-9-]+	# follow by a dot "." and string in the bracket [ ], must contains one or more (+)
)*	# end of group #1, this group is optional (*)
@	# must contains a "@" symbol
[A-Za-z0-9-]+	# follow by string in the bracket [ ], must contains one or more (+)
(	# start of group #2 - first level TLD checking
\\.[A-Za-z0-9-]+	# follow by a dot "." and string in the bracket [ ], must contains one or more (+)
)*	# end of group #2, this group is optional (*)
(	# start of group #3 - second level TLD checking
\\.[A-Za-z]{2,}	# follow by a dot "." and string in the bracket [ ], with minimum length of 2
)	# end of group #3
\$	#end of the line

The regular expression combination means, email address must start with “\_A-Za-z0-9-\\+” , optional follow by “[\_A-Za-z0-9-]”, and end with a “@” symbol. The email’s domain name must start with “A-Za-z0-9-“, follow by first level Tld (.com, .net) “[A-Za-z0-9]” and optional follow by a second level Tld (.com.au, .com.my) “\\.[A-Za-z]{2,}”, where second level Tld must start with a dot “.” and length must equal or more than 2 characters.

### Sample:

```
package com.sample.regexsample;  
  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;  
  
public class EmailValidator {  
    public static void main(String[] args) {  
        String input =  
            "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$";  
    }
```

```
String emailAddress = "myName@gmail.com";  
String emailAddress2 = "myName@gmail.c";  
String emailAddress3 = "mylastName@yahoo.co.in";
```

```
Pattern p = Pattern.compile(input);  
Matcher m = p.matcher(emailAddress);  
Matcher m2 = p.matcher(emailAddress2);  
Matcher m3 = p.matcher(emailAddress3);  
System.out.println ("The email 1 matches:" + m.matches());  
System.out.println ("The email 2 matches:" + m2.matches());  
System.out.println ("The email 3 matches:" + m3.matches());
```

```
}
```

```
}
```

Output:

The email 1 matches:true

The email 2 matches:false

The email 3 matches:true

## Quiz

Write class and test case for validating the following email addresses.

1. name@abc.com
2. forget\_1@te
3. lastnameE12\_45@test.co.in
4. myName12\*
5. user\$\$Name@myname.b
6. LASTNAME12@test.com



## Solution

CODE:

```
package com.sample.RegexExamples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class EmailValidator {

    private Pattern pattern;
    private Matcher matcher;

    private static final String EMAIL_PATTERN =
        "^[_A-Za-z0-9-\\+](\\.[_A-Za-z0-9-]+)*@"
        + "[A-Za-z0-9-](\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$";

    public EmailValidator() {
        pattern = Pattern.compile(EMAIL_PATTERN);
    }

    /**
     * Validate hex with regular expression
     *
     * @param hex
     *      hex for validation
     * @return true valid hex, false invalid hex
     */
    public boolean validate(final String hex) {

        matcher = pattern.matcher(hex);
        return matcher.matches();

    }

}
```

UNIT TEST :

```
package com.sample.RegexTest;
```

```
import org.junit.Assert;
```

```
import org.junit.Before;
```

```

import org.junit.Test;

import com.sample.RegexExamples.EmailValidator;

public class EmailValidatorTest {

    private EmailValidator emailValidator;

    @Before
    public void initData() {
        emailValidator = new EmailValidator();
    }

    String[] ValidEmailProvider = new String[] { "mkyong@yahoo.com",
        "mkyong-100@yahoo.com", "mkyong.100@yahoo.com",
        "mkyong111@mkyong.com", "mkyong-100@mkyong.net",
        "mkyong.100@mkyong.com.au", "mkyong@1.com",
        "mkyong@gmail.com.com", "mkyong+100@gmail.com",
        "mkyong-100@yahoo-test.com" };

    String[] InvalidEmailProvider= new String[] { "mkyong", "mkyong@.com.my",
        "mkyong123@gmail.a", "mkyong123@.com", "mkyong123@.com.com",
        ".mkyong@mkyong.com", "mkyong()*@gmail.com", "mkyong@%*.com",
        "mkyong..2002@gmail.com", "mkyong.@gmail.com",
        "mkyong@mkyong@gmail.com", "mkyong@gmail.com.1a" };

    @Test
    public void ValidEmailTest() {

        for (String temp : ValidEmailProvider) {
            boolean valid = emailValidator.validate(temp);
            System.out.println("Email is valid : " + temp + " , " + valid);
            Assert.assertEquals(valid, true);
        }

    }

    @Test
    public void InvalidEmailTest() {

        for (String temp : InvalidEmailProvider) {
            boolean valid = emailValidator.validate(temp);
            System.out.println("Email is valid : " + temp + " , " + valid);
            Assert.assertEquals(valid, false);
        }

    }
}

```

}

}

OUTPUT:

Email is valid : mkyong , false  
Email is valid : mkyong@.com.my , false  
Email is valid : mkyong123@gmail.a , false  
Email is valid : mkyong123@.com , false  
Email is valid : mkyong123@.com.com , false  
Email is valid : .mkyong@mkyong.com , false  
Email is valid : mkyong()\*@gmail.com , false  
Email is valid : mkyong@%\*.com , false  
Email is valid : mkyong..2002@gmail.com , false  
Email is valid : mkyong.@gmail.com , false  
Email is valid : mkyong@mkyong@gmail.com , false  
Email is valid : mkyong@gmail.com.1a , false  
Email is valid : mkyong@yahoo.com , true  
Email is valid : mkyong-100@yahoo.com , true  
Email is valid : mkyong.100@yahoo.com , true  
Email is valid : mkyong111@mkyong.com , true  
Email is valid : mkyong-100@mkyong.net , true  
Email is valid : mkyong.100@mkyong.com.au , true  
Email is valid : mkyong@1.com , true  
Email is valid : mkyong@gmail.com.com , true  
Email is valid : mkyong+100@gmail.com , true  
Email is valid : mkyong-100@yahoo-test.com , true

# Chapter 7

## Regex Color Coding

### Hexadecimal Color Code Regular Expression Pattern

```
^#[A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})$
```

### Description

<code>^</code>	<code>#start of the line</code>
<code>#</code>	<code># must contains a "#" symbols</code>
<code>(</code>	<code># start of group #1</code>
<code>[A-Fa-f0-9]{6}</code>	<code># any strings in the list, with length of 6</code>
<code> </code>	<code># ..or</code>
<code>[A-Fa-f0-9]{3}</code>	<code># any strings in the list, with length of 3</code>
<code>)</code>	<code># end of group #1</code>
<code>\$</code>	<code>#end of the line</code>

The hexadecimal color coding string combination means, string must start with a “#” symbol , follow by a letter from “a” to “f”, “A” to “Z” or a digit from “0” to 9” with exactly 6 or 3 length.

### Sample Code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class HexCodeValidator {

    public static void main(String[] args) {
        String input = "^#[A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})$";

        String hexCode1 = "#1f1f1F";
        String hexCode2 = "12345";

        Pattern p = Pattern.compile(input);
        Matcher m = p.matcher(hexCode1);
        Matcher m2 = p.matcher(hexCode2);

        System.out.println ("The hexCode 1 matches:" + m.matches());
        System.out.println ("The hexCode 2 matches:" + m2.matches());

    }
```

```
}
```

Output:

The hexCode 1 matches:true

The hexCode 2 matches:false

# Chapter 8

## Regex for Image Files

Image File Extension Regular Expression Pattern

```
([^\s]+(\.(?i)(jpg|png|gif|bmp)))$)
```

Description

(	# Start of the group #1
[^\s]+	# must contains one or more anything (except white space)
(	# start of the group #2
\.	# followed by a dot "."
(?i)	# ignore the case sensitive checking for the following characters
(	# start of the group #3
jpg	# includes characters "jpg"
	# ..or
png	# includes characters "png"
	# ..or
gif	# includes characters "gif"
	# ..or
bmp	# includes characters "bmp"
)	# end of the group #3
)	# end of the group #2
\$	# end of the string
)	#end of the group #1

The regular expression means, must have 1 or more strings (but not white space), follow by dot “.” and string end in “jpg” or “png” or “gif” or “bmp” , and the file extensive is case-insensitive.

This regex pattern is very use for a lot of file extensive checking for example you could modify the final combination (jpg, png, gif, bmp) in order to suit your need.

Sample Code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForImageFile {

    public static void main(String[] args) {
        String input = "([^\s]+(\.(?i)(jpg|png|gif|bmp)))$";
        String file1 ="Karen_3.gif";
        String file2="MyName$.jpeg";
    }
}
```

```
Pattern p = Pattern.compile(input);  
Matcher m = p.matcher(file1);  
Matcher m2 = p.matcher(file2);  
System.out.println ("The file name 1 matches:" + m.matches());  
System.out.println ("The file name 2 matches:" + m2.matches());  
}
```

Output:

The file name 1 matches:true

The file name 2 matches:false

# Chapter 9

## Regex for IP Address

### IP Address Regular Expression Pattern

```
^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])$
```

### Description

<code>^</code>	<code>#start of the line</code>
<code>(</code>	<code># start of group #1</code>
<code>[01]?\d\d?</code>	<code># Can be one or two digits. If three digits show up, it must start either 0 or 1</code>
<code> </code>	<code># ...or</code>
<code>2[0-4]\d</code>	<code># start with 2, follow by 0-4 and end with any digit (2[0-4][0-9])</code>
<code> </code>	<code># ...or</code>
<code>25[0-5]</code>	<code># start with 2, follow by 5 and ends with 0-5 (25[0-5])</code>
<code>)</code>	<code># end of group #2</code>
<code>\.</code>	<code># follow by a dot "."</code>
<code>....</code>	<code># repeat with 3 times (3x)</code>
<code>\$</code>	<code>#end of the line</code>

The regular expression on whole means, digit from 0 to 255 and follow by a dot “.”, repeat 4 time and ending with no dot “.” Valid IP address format is “0-255.0-255.0-255.0-255”.

### Sample Code:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForIpAddress {
```

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

```
        String IPADDRESS_PATTERN =
```

```
            "^([01]?\d\d?|2[0-4]\d|25[0-5])\. " +
```

```
            "([01]?\d\d?|2[0-4]\d|25[0-5])\. " +
```

```
            "([01]?\d\d?|2[0-4]\d|25[0-5])\. " +
```

```
            "([01]?\d\d?|2[0-4]\d|25[0-5])$";
```

```
        String ipAddress1 = "192.2.1";
```

```
        String ipAddress2 = "100.23.255.1";
```

```
        Pattern p = Pattern.compile(IPADDRESS_PATTERN);
```



```
        Matcher m = p.matcher(ipAddress1);
        Matcher m2 = p.matcher(ipAddress2);
        System.out.println ("The user name 1 matches:" + m.matches());
        System.out.println ("The user name 2 matches:" + m2.matches());
    }
}
```

Output:

The user name 1 matches:false

The user name 2 matches:true

# **Chapter 10**

## **Regex for Time Format**

# Time in 12-Hour Format Regex Pattern

```
(1[012]|[1-9]):[0-5][0-9](\\s)?(?i)(am|pm)
```

Description

(	# start of group #1
1[012]	# start with 10, 11, 12
	# or
[1-9]	# start with 1,2,...9
)	# end of group #1
:	# follow by a semi colon (:)
[0-5][0-9]	# follow by 0..5 and 0..9, which means 00 to 59
(\\s)?	# follow by a white space (optional)
(?i)	# next checking is case insensitive
(am pm)	# follow by am or pm

The regular expression on whole means, The 12-hour clock format is start from 0-12, then a semi colon (:) and follow by 00-59 , and end with am or pm.

Sample Code :

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForTimeFormat {
```

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

```
        String input =
```

```
            "(1[012]|[1-9]):[0-5][0-9](\\s)?(?i)(am|pm)";
```

```
        String time1="10:15 pm";
```

```
        String time2="12:45 a.m";
```

```
        String time3="01:45 am";
```

```
        String time4="2:45 am";
```

```
        Pattern p = Pattern.compile(input);
```

```
        Matcher m = p.matcher(time1);
```

```
        Matcher m2 = p.matcher(time2);
```

```
        Matcher m3 = p.matcher(time3);
```

```
        Matcher m4 = p.matcher(time4);
```

```
        System.out.println ("The time 1 matches:" + m.matches());
```

```
        System.out.println ("The time 2 matches:" + m2.matches());
```

```
        System.out.println ("The time 3 matches:" + m3.matches());
```

```
        System.out.println ("The time 4 matches:" + m4.matches());
```

```
}
```

```
}
```

Output:

The time 1 matches:true

The time 2 matches:false

The time 3 matches:false

The time 4 matches:true

# Time in 24-Hour Format Regex Pattern

`([01]?[0-9]|2[0-3]):[0-5][0-9]`

Description

<code>(</code>	<code># start of group #1</code>
<code>[01]?[0-9]</code>	<code># start with 0-9,1-9,00-09,10-19</code>
<code> </code>	<code># or</code>
<code>2[0-3]</code>	<code># start with 20-23</code>
<code>)</code>	<code># end of group #1</code>
<code>:</code>	<code># follow by a semi colon (:)</code>
<code>[0-5][0-9]</code>	<code># follow by 0..5 and 0..9, which means 00 to 59</code>

The regular expression for the 24-hour clock format is start from 0-23 or 00-23 then a semi colon (:) and follow by 00-59.

Sample Code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoforTimeFormat2 {
    public static void main(String[] args) {
        String input =
            "([01]?[0-9]|2[0-3]):[0-5][0-9]";

        String time1="10:15";
        String time2="12:45";
        String time3="25:45";
        String time4="23:45";

        Pattern p = Pattern.compile(input);
        Matcher m = p.matcher(time1);
        Matcher m2 = p.matcher(time2);
        Matcher m3 = p.matcher(time3);
        Matcher m4 = p.matcher(time4);

        System.out.println ("The time 1 matches:"+ m.matches());
        System.out.println ("The time 2 matches:"+ m2.matches());
        System.out.println ("The time 3 matches:"+ m3.matches());
        System.out.println ("The time 4 matches:"+ m4.matches());
    }
}
```

Output:

The time 1 matches:true  
The time 2 matches:true  
The time 3 matches:false  
The time 4 matches:true

# Chapter 11

## Regex for Date Format

Date Format (dd/mm/yyyy) Regular Expression Pattern

```
(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/((19|20)\d\d)
```

Description

```
(
    # start of group #1
0?[1-9]      # 01-09 or 1-9
|           # ..or
[12][0-9]    # 10-19 or 20-29
|           # ..or
3[01]        # 30, 31
)           # end of group #1
/          # follow by a "/"
(
    # start of group #2
0?[1-9]      # 01-09 or 1-9
|           # ..or
1[012]       # 10,11,12
)           # end of group #2
/          # follow by a "/"
(
    # start of group #3
(19|20)\d\d  # 19[0-9][0-9] or 20[0-9][0-9]
)           # end of group #3
```

The above regex is used to validate the date format in “dd/mm/yyyy”, as usual you can work with this regex in order to suit your need. Anyway, I will be a little complicated to validate the leap year , 30 or 31 days of a month, Let’s see how can we do it.

Sample Code:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class HexCodeValidator {
```

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

```
        String input="(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/((19|20)\d\d)";
```

```
        String date1="31/08/2016";
```

```
        String date2="12345";
```

```
Pattern p = Pattern.compile(input);
Matcher m = p.matcher(date1);
Matcher m2 = p.matcher(date2);

System.out.println ("The date 1 matches:" + m.matches());
System.out.println ("The date 2 matches:" + m2.matches());
```

```
}
```

```
}
```

Output:  
The date 1 matches:true  
The date 2 matches:false



## Quiz

Used Named grouping(special construct) to validate the date format for leap year

# Solution

## Code Sample

```
package com.sample.regexsample;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForDateValidation {

    public static void main(String[] args) {
        String input = "^(?<month>[0-3]?[0-9])/(?<day>[0-3]?[0-9])/(?<year>(?:[0-9]{2})?[0-9]{2})$";

        List<String> dates = new ArrayList<String>();

        dates.add("02/31/2011"); //Invalid date
        dates.add("02/27/2011");

        Pattern pattern = Pattern.compile(input);

        for (String date :dates )
        {
            Matcher matcher = pattern.matcher(date);
            matcher.matches();

            //Get date parts here
            String day = matcher.group(2);
            String month = matcher.group(1);
            String year = matcher.group(3);

            String formattedDate = month + "/" + day + "/" + year;

            System.out.println("Date to check : " + formattedDate);
            System.out.println("Is date valid : " + isDateValid(formattedDate));
        }
    }
}
```

```

    }
    public static boolean isDateValid(String date)
    {
        String DATE_FORMAT = "MM/dd/yyyy";
        try {
            DateFormat df = new SimpleDateFormat(DATE_FORMAT);
            df.setLenient(false);
            Date d = df.parse(date);

            System.out.println("Parsed Date : " + d);
            return true;
        } catch (ParseException e) {
            return false;
        }

    }
}

```

Output:

```

Date to check : 02/31/2011
Is date valid : false
Date to check : 02/27/2011
Parsed Date : Sun Feb 27 00:00:00 AEDT 2011
Is date valid : true

```

# Chapter 12

## Regex for HTML Tag

### HTML tag Regular Expression Pattern

```
<("[^"]*"|'[^']*'|"[^"]>")*>
```

### Description

<	# start with opening tag "<"
(	# start of group #1
"[^"]*"	# allow string with double quotes enclosed - "string"
	# ..or
'[^']*'	# allow string with single quote enclosed - 'string'
	# ..or
[^">]	# cant contains one single quotes, double quotes and ">"
)	# end of group #1
*	# 0 or more
>	# end with closing tag ">"

The regular expression on whole means, HTML tag, start with an opening tag “<” , follow by double quotes "string", or single quotes 'string' but does not allow one double quotes (") "string, one single quote (') 'string or a closing tag > without single or double quotes enclosed. At last , end with a closing tag “>”

### Example:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForHTMLTag {
```

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

```
        String HTML_PATTERN = "<(\"[^\"]*"|'[^']*'|\"[^\"]>")*>";
```

```
        String htmlTag = "<br>";
```

```
        Pattern p = Pattern.compile(HTML_PATTERN);
```

```
        Matcher m = p.matcher(htmlTag);
```

```
        System.out.println ("The html tag matches:" + m.matches());
```

```
    }
```

```
}  
Output  
The html tag matches:true
```

# Chapter 13

## Regex for Finding HTML Links

Extract A tag Regular Expression Pattern

```
(?i)<a([^\>]+)>(.*?)</a>
```

Extract Link From A tag Regular Expression Pattern

```
\s*(?i)href\s*=\s*(\"([^\"]*\")|'([^']*>\s)+));
```

Description

```
(          # start of group #1
?i        # all checking are case insensitive
)          # end of group #1
<a        # start with "<a"
(          # start of group #2
[^\>]+    # anything except (">"), at least one character
)          # end of group #2
>         # follow by ">"
(.*?)     # match anything
</a>      # end with "</a>"

\s*       # can start with whitespace
(?i)      # all checking are case insensitive
href      # follow by "href" word
\s*=\s*   # allows spaces on either side of the equal sign,
(         # start of group #1
"([^\"]*)" # allow string with double quotes enclosed - "string"
|         # ..or
'[^']*'    # allow string with single quotes enclosed - 'string'
|         # ..or
([^\>"])+ # can't contains one single quotes, double quotes ">"
)         # end of group #1
```

Code Example:

```
package com.sample.regexsample;
```

```
import java.util.Vector;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForLink {
```

```

String HTML_A_TAG_PATTERN = "(?i)<a([>]+)>(.*?)</a>";
String HTML_A_HREF_TAG_PATTERN =
    "\\s*(?i)href\\s*=\\s*(\"([^\"]*\")|'[^']*'|([^\">\\s]+))";

Pattern patternTag = Pattern.compile(HTML_A_TAG_PATTERN);
Pattern patternLink = Pattern.compile(HTML_A_HREF_TAG_PATTERN);

Matcher matcherTag, matcherLink;

/**
 * Validate html with regular expression
 *
 * @param html
 *      html content for validation
 * @return Vector links and link text
 */
public void grabHTMLLinks(final String html) {

    matcherTag = patternTag.matcher(html);

    while (matcherTag.find()) {

        String href = matcherTag.group(1); // href
        String linkText = matcherTag.group(2); // link text

        matcherLink = patternLink.matcher(href);

        while (matcherLink.find()) {

            String link = matcherLink.group(1); // link
            System.out.println("Link is "+link);
            System.out.println("Linktext is "+linkText);

        }

    }

}

public static void main(String[] args) {
    String TEST_LINK = "http://www.google.com";
    DemoForLink htmlLinkExtractor = new DemoForLink();
    String html = "abc hahaha <a href=\"" + TEST_LINK + "\">google</a>";
    htmlLinkExtractor.grabHTMLLinks(html);
}

```

}

}

Output:

Link is 'http://www.google.com'

Linktext is google



## Expression

```
^(?i)(https?|ftp|file|www)://[a-zA-Z0-9+&@#/%?~_!:,;]*[a-zA-Z0-9+&@#/%=~_]
```

## Description

```
(           # start of group #1
?i         # all checking are case insensitive
)          # end of group #1
(
https?     # contains http or https
|          # ..or
ftp        # contains ftp
|          # ..or
file       # contains file
|          # ..or
www        # contains www
)          # end of group 2
://        # contains a : followed by //
```

[a-zA-Z0-9+&@#/%?~\_!:,;] #contains any characters between A-Z digits between 0-9 and list of symbols given followed by a .(optional)  
\*[a-zA-Z0-9+&@#/%=~\_] #contains any characters between A-Z digits between 0-9 and list of symbols given (This group is also optional)

## Example:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForURL {
```

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

```
        String lRegex = "^(https?|ftp|file|www)://[a-zA-Z0-9+&@#/%?~_!:,;]*[a-zA-Z0-9+&@#/%=~_]";
```

```
        String htmlTag = "www://mysiteindex.com";
```

```
        Pattern p = Pattern.compile(lRegex);
```

```
        Matcher m = p.matcher(htmlTag);
```

```
System.out.println ("The html tag matches:"+ m.matches());
```

```
}
```

```
}
```

Output:

The html tag matches:true

# Chapter 15

## Regex for Phone Number

Regular expression

```
^\((?([0-9]{3})\))?[-.\\s*]?([0-9]{3})[-.\\s*]?([0-9]{4})$
```

Explanation:

```
^    # Assert position at the beginning of the string.
\((  # Match a literal "("
?    # between zero and one time.
(    # Capture the enclosed match to backreference 1:
[0-9] # Match a digit
{3}   # exactly three times.
)     # End capturing group 1.
\)    # Match a literal ")"
?     # between zero and one time.
[-. ] # Match one hyphen, dot, or space
?     # between zero and one time.
      # [Match the remaining digits and separator.]
$     # Assert position at the end of the string.
```

Example:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForPhoneNumber {
```

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

```
        String lRegex = "^\((?([0-9]{3})\))?[-.\\s*]?([0-9]{3})[-.\\s*]?([0-9]{4})$";
```

```
        String phoneNumber1 = "123-456-7890";
```

```
        String phoneNumber2 = "(123)456-7890";
```

```
        String phoneNumber3 = "123 456 7890";
```

```
        String phoneNumber4 = "1234567890";
```

```
        String phoneNumber5 = "123-456-789";
```

```
        Pattern p = Pattern.compile(lRegex);
```

```
        Matcher m = p.matcher(phoneNumber1);
```

```
        Matcher m1 = p.matcher(phoneNumber2);
```

```
Matcher m2 = p.matcher(phoneNumber3);  
Matcher m3 = p.matcher(phoneNumber4);  
Matcher m4 = p.matcher(phoneNumber5);
```

```
System.out.println ("The html tag matches:"+ m.matches());  
System.out.println ("The html tag matches:"+ m1.matches());  
System.out.println ("The html tag matches:"+ m2.matches());  
System.out.println ("The html tag matches:"+ m3.matches());  
System.out.println ("The html tag matches:"+ m4.matches());
```

```
}
```

```
}  
Output:  
The html tag matches:true  
The html tag matches:true  
The html tag matches:true  
The html tag matches:true  
The html tag matches:false
```

# Chapter 16

## Regex for International Numbers

Regular expression

```
^\++(?:[0-9]\\s*?){6,14}[0-9]$
```

Explanation:

`^` # Assert position at the beginning of the string.  
`\\+` # Match a literal "+" character.  
`(?:` # Group but don't capture:  
`[0-9]` # Match a digit.  
`\\s*` # Match a space character  
`?` # between zero and one time.  
`)` # End the noncapturing group.  
`{6,14}` # Repeat the group between 6 and 14 times.  
`[0-9]` # Match a digit.  
`$` # Assert position at the end of the string.

Example:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForIntPhone {
```

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

```
        String regex = "^\++(?:[0-9]\\s*?){6,14}[0-9]$";
```

```
        String phoneNumber1 = "+91 1234567890";
```

```
        String phoneNumber2 = "(123)456-7890";
```

```
        String phoneNumber3 = "+891234567890";
```

```
        Pattern p = Pattern.compile(regex);
```

```
        Matcher m = p.matcher(phoneNumber1);
```

```
        Matcher m1 = p.matcher(phoneNumber2);
```

```
        Matcher m2 = p.matcher(phoneNumber3);
```

```
        System.out.println("The html tag matches:" + m.matches());
```

```
        System.out.println("The html tag matches:" + m1.matches());
```

```
System.out.println ("The html tag matches:"+ m2.matches());
```

```
}
```

```
}  
Output:  
The html tag matches:true  
The html tag matches:false  
The html tag matches:true
```

# **Chapter 17**

## **Other Commonly Used Patterns**

# Validate Only Alphanumeric Pattern

Pattern:

`^[a-zA-Z0-9]*$`

Explanation:

`"^[a-zA-Z0-9]*$"`

<code>^</code>	<code># Start of the pattern</code>
<code>[</code>	<code># Start of the String</code>
<code>a-zA-Z0-9</code>	<code># Word comprises characters from a to z , without case sensitivity and numerals</code>
<code>from</code>	<code>0 to 9</code>
<code>]</code>	<code># End of the String</code>
<code>*</code>	<code># Optional alphabets or numerals</code>
<code>\$</code>	<code># End of Pattern</code>

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForAlphaNumericCheck {
```

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

```
    {
```

```
        String patternString= "^[a-zA-Z0-9]*$";
```

```
        String name1="Teststring123";
```

```
        String name2="lastname";
```

```
        String name3="12345";
```

```
        String name4="la12345@";
```

```
        Pattern pattern = Pattern.compile(patternString);
```

```
        Matcher matcher1 = pattern.matcher(name1);
```

```
        Matcher matcher2 = pattern.matcher(name2);
```

```
        Matcher matcher3 = pattern.matcher(name3);
```

```
        Matcher matcher4 = pattern.matcher(name4);
```

```
        System.out.println("YAy found out alphanumeric "+matcher1.matches());
```

```
        System.out.println("YAy found out alphanumeric "+matcher2.matches());
```

```
        System.out.println("YAy found out alphanumeric "+matcher3.matches());
```

```
        System.out.println("YAy found out alphanumeric "+matcher4.matches());
```

```
    }
```

```
}
```



Output:  
YAy found out alphanumeric true  
YAy found out alphanumeric true  
YAy found out alphanumeric true  
YAy found out alphanumeric false

# Float Point Numbers

## Pattern

`^[-+]?[0-9]*\\.[0-9]+$`

## Explanation:

`^` # Start of the pattern  
`[-+]?` # Optional Sign  
`[0-9]*` # Followed by an series of digits (optional)  
`\\.?` # Followed by an optional "." symbol  
`[0-9]` # Followed by an series of digits (optional)

## Sample Code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForFloatingPointNumber {

    public static void main(String[] args) {
        String patternString = "^[-+]?[0-9]*\\.[0-9]+$";

        String num1="123";
        String num2="lastname";
        String num3="12.345";
        String num4="la12345@";

        Pattern pattern = Pattern.compile(patternString);
        Matcher matcher1 = pattern.matcher(num1);
        Matcher matcher2 = pattern.matcher(num2);
        Matcher matcher3 = pattern.matcher(num3);
        Matcher matcher4 = pattern.matcher(num4);

        System.out.println("Found out floating number "+matcher1.matches());
        System.out.println("Found out floating number "+matcher2.matches());
        System.out.println("Found out floating number "+matcher3.matches());
        System.out.println("Found out floating number "+matcher4.matches());
    }
}
```

Output:

```
Found out floating number    true
Found out floating number    false
Found out floating number    true
Found out floating number    false
```

Another pattern:

```
[-+]?([0-9]*\.[0-9]+|[0-9]+).
```

This regex equals an optional sign, that is either ensue by zero or more digits followed by a dot and one or more digits (a floating point number with optional integer part), or that is ensue by one or more digits (an integer).

Furthermore optimization:

We can optimize this regular expression as: `[-+]?[0-9]*\.[0-9]+`.

Explanation

`[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?`. The complete exponent part optional by grouping it together, rather than making each element in the exponent optional.

Sample Code:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForFloatingPointNumber {
```

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

```
        String patternString = "[-+]?([0-9]*\.[0-9]+|[0-9]+)";
```

```
        String num1="123";
```

```
        String num2="lastname";
```

```
        String num3="+12.345";
```

```
        String num4="-la12345@";
```

```
        String num5="-2.3";
```

```
        String num6="-12";
```

```
        Pattern pattern = Pattern.compile(patternString);
```

```
        Matcher matcher1 = pattern.matcher(num1);
```

```
        Matcher matcher2 = pattern.matcher(num2);
```

```
        Matcher matcher3 = pattern.matcher(num3);
```

```

        Matcher matcher4 = pattern.matcher(num4);
        Matcher matcher5 = pattern.matcher(num5);
        Matcher matcher6 = pattern.matcher(num6);

        System.out.println("Found out floating number "+matcher1.matches());
        System.out.println("Found out floating number "+matcher2.matches());
        System.out.println("Found out floating number "+matcher3.matches());
        System.out.println("Found out floating number "+matcher4.matches());
        System.out.println("Found out floating number "+matcher5.matches());
        System.out.println("Found out floating number "+matcher6.matches());
    }

```

Output:

```

Found out floating number    true
Found out floating number    false
Found out floating number    true
Found out floating number    false
Found out floating number    true
Found out floating number    true

```

Sample Code 2:

```

package com.sample.regexsample;

```

```

import java.util.regex.Matcher;

```

```

import java.util.regex.Pattern;

```

```

public class DemoForFloatingPointNumber {

```

```

    public static void main(String[] args) {
        String patternString = "[+-]?[0-9]*\\.?[0-9]+([eE][+-]?[0-9]+)?";
    }

```

```

        String num1="123";
        String num2="lastname";
        String num3="+12.23E45";
        String num4="-1a12345@";
        String num5="-2.3e6";
        String num6="-12.e45";
    }

```

```

        Pattern pattern = Pattern.compile(patternString);
        Matcher matcher1 = pattern.matcher(num1);
        Matcher matcher2 = pattern.matcher(num2);
        Matcher matcher3 = pattern.matcher(num3);
        Matcher matcher4 = pattern.matcher(num4);
        Matcher matcher5 = pattern.matcher(num5);
        Matcher matcher6 = pattern.matcher(num6);
    }

```

```
        System.out.println("Found out floating number "+matcher1.matches());  
        System.out.println("Found out floating number "+matcher2.matches());  
        System.out.println("Found out floating number "+matcher3.matches());  
        System.out.println("Found out floating number "+matcher4.matches());  
        System.out.println("Found out floating number "+matcher5.matches());  
        System.out.println("Found out floating number "+matcher6.matches());  
    }  
  
}
```

Output:

```
Found out floating number  true  
Found out floating number  false  
Found out floating number  true  
Found out floating number  false  
Found out floating number  true  
Found out floating number  false
```

# Deleting Duplicating Words

## Pattern

```
"\\b(\\w+)\\b\\s+\\b\\1\\b"
```

## Explanation

```
\\b    # word boundary
(\\w+) # word for which duplicate is to be found
\\b    # end of word
\\s+   # optional white space
\\b    # word boundary
\\1    # references to the captured match of the first group, i.e., the first word.
\\b    # end of word
```

## Sample Code

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForDuplication {
    public static void main(String args[]) throws Exception {
        String input="This This is my pen pen and pencil pencil text another another";

        String output = "";

        Pattern p = Pattern.compile("\\b(\\w+)\\b\\s+\\b\\1\\b",
Pattern.MULTILINE+Pattern.CASE_INSENSITIVE);
        Matcher m = p.matcher(input);

        if (!m.find())
            output = "No duplicates found, no changes made to data";
        else
        {
            while (m.find())
            {

                if (output == "") {
                    output = input.replaceFirst(m.group(), m.group(1));
                } else {
                    output = output.replaceAll(m.group(), m.group(1));
                }
            }
            input = output;
            m = p.matcher(input);
```

```

while (m.find())
{
    output = "";
    if (output == "") {
        output = input.replaceAll(m.group(), m.group(1));
    } else {
        output = output.replaceAll(m.group(), m.group(1));
    }
}
}
System.out.println("After removing duplicate the final string is " + output);
}

```

}

Output:

After removing duplicate the final string is this is my pen and pencil text another

# Validating SSN

## Pattern

`^\d{3}-\d{2}-\d{4}$`

## Explanation:

`^` # Start of the string  
`\d{3}` # Should start with a 3 digit number  
`-` # followed by a hyphen  
`\d{2}` # Followed by a 2 digit number  
`-` # followed by a hyphen  
`\d{4}` # Followed by a 4 digit Number  
`$` # End of the String

## Sample Code:

```
package com.sample.regexsample;

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForSSN {

    public static void main(String args[])
    {
        String patternString= "^\d{3}-\d{2}-\d{4}$";

        List<String> input = new ArrayList<String>();
        input.add("123-45-6789");
        input.add("9876-5-4321");
        input.add("987-65-4321 (attack)");
        input.add("987-65-4321 ");
        input.add("192-83-7465");

        Pattern pattern = Pattern.compile(patternString);

        for (String ssn : input) {
            Matcher m = pattern.matcher(ssn);
            if (m.matches()) {
                System.out.println("Found good SSN: " + ssn);
            }
        }
    }
}
```



```
}
```

Output:

Found good SSN: 123-45-6789

Found good SSN: 192-83-7465

## Quiz

Write a program to validate SSN for the following conditions

Social Security number CANNOT :

1. Contain all zeroes in any specific group (ie 000-##-####, ###-00-####, or ###-##-0000)
2. Begin with '666'.
3. Begin with any value from '900-999'

## Solution

Sample Code:

```
package com.sample.regexsample;
```

```
import java.util.ArrayList;
```

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

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForSSN {
```

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

```
    {
```

```
        //String patternString= "^(\d{3}-\d{2}-\d{4})$";
```

```
        String patternString = "^(?:666|000|9\d{2})\d{3}-(?!00)\d{2}-(?!0{4})\d{4}$";
```

```
        List<String> input = new ArrayList<String>();
```

```
        input.add("123-45-6789");
```

```
        input.add("9876-5-4321");
```

```
        input.add("987-65-4321 (attack)");
```

```
        input.add("987-65-4321 ");
```

```
        input.add("192-83-7465");
```

```
        input.add("666-83-7465");
```

```
        input.add("999-83-7465");
```

```
        input.add("000-83-7465");
```

```
Pattern pattern = Pattern.compile(patternString);
```

```
for (String ssn : input) {  
    Matcher m = pattern.matcher(ssn);  
    if (m.matches()) {  
        System.out.println("Found good SSN: " + ssn);  
    }  
    else  
    {  
        System.out.println("Found bad SSN: " + ssn);  
    }  
}  
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

# Validating Number Range

Pattern

`^(\d?[1-9]|[1-9]0)$`

`^` # Start of the String  
`((\d?[1-9]|[1-9]0)` # Should contain a digit that can contain 1 to 9 or 01 to 09, 11 to 19, 21 to 29,  
..., 91 to 99 or 10, 20, ..., 90  
`$` # End of String

Sample Code:

```
package com.sample.regexsample;

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForNumberRange {

    public static void main(String[] args) {

        String patternString = "^(\d?[1-9]|[1-9]0)$";

        List<String> input = new ArrayList<String>();
        input.add("01");
        input.add("9876");
        input.add("11ab");
        input.add("21");
        input.add("19");

        Pattern pattern = Pattern.compile(patternString);

        for (String ssn : input) {
            Matcher m = pattern.matcher(ssn);
            if (m.matches()) {
                System.out.println("Found good number: " + ssn);
            }
        }
    }
}
```

Output:

Found good number: 01

Found good number: 21

Found good number: 19

# Validating Zip Code

Pattern ( US ZIP CODE)

```
^[0-9]{5}(?:-[0-9]{4})?$
```

Explanation:

<code>^</code>	<code># State the position at the beginning of the string.</code>
<code>[0-9]{5}</code>	<code># Equal a digit, exactly five times.</code>
<code>(?:</code>	<code># Group but don't capture:</code>
<code>-</code>	<code># Equal a literal "-".</code>
<code>[0-9]{4}</code>	<code># Equal a digit, exactly four times.</code>
<code>)</code>	<code># End the non-capturing group.</code>
<code>?</code>	<code># Set up the group optional.</code>
<code>\$</code>	<code># State position at the end of the string.</code>

Sample Code

```
package com.sample.regexsample;
```

```
import java.util.ArrayList;
```

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

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForZipCode {
```

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

```
        List<String> zips = new ArrayList<String>();
```

```
        //Valid ZIP codes
```

```
        zips.add("12345");
```

```
        zips.add("12345-6789");
```

```
        //Invalid ZIP codes
```

```
        zips.add("123456");
```

```
        zips.add("1234");
```

```
        zips.add("12345-678");
```

```
        zips.add("12345-67890");
```

```
        String regex = "^[0-9]{5}(?:-[0-9]{4})?$";
```

```
        Pattern pattern = Pattern.compile(regex);
```

```
        for (String zip : zips)
```

```

    {
        Matcher matcher = pattern.matcher(zip);
        System.out.println("ZIPCODE " + zip + " Matches " + matcher.matches());
    }

}

```

Output:

```

ZIPCODE 12345 Matches true
ZIPCODE 12345-6789 Matches true
ZIPCODE 123456 Matches false
ZIPCODE 1234 Matches false
ZIPCODE 12345-678 Matches false
ZIPCODE 12345-67890 Matches false

```

## Quiz

Write programs to validate the zip code

1. Australian zip code (4 – digits)
2. Indian zip code ( 6 digits)
3. European zip codes
4. Canadian zip codes

## Solution

```

package com.sample.regexsample;

```

```

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

```

```

public class DemoForZipCode {

```

```

    public static void main(String[] args) {
        List<String> UKzips = new ArrayList<String>();
        List<String> INDzips = new ArrayList<String>();
        List<String> AUzips = new ArrayList<String>();

        List<String> CANzips = new ArrayList<String>();
    }
}

```

```
//Valid ZIP codes
CANzip.add("K1A 0B1");
CANzip.add("B1Z 0B9");
```

```
//Invalid ZIP codes
CANzip.add("K1A 0D1");
CANzip.add("W1A 0B1");
CANzip.add("Z1A 0B1");
```

```
//Valid ZIP codes
INDzip.add("400 117");
INDzip.add("600 117");
INDzip.add("623 4315");
INDzip.add("400050");
```

```
//Invalid ZIP codes
INDzip.add("0200");
INDzip.add("7312=-3535");
INDzip.add("2415 4545");
INDzip.add("320645");
```

```
//Valid ZIP codes
UKzip.add("SW1W 0NY");
UKzip.add("PO16 7GZ");
UKzip.add("GU16 7HF");
UKzip.add("L1 8JQ");
```

```
//Invalid ZIP codes
UKzip.add("Z1A 0B1");
UKzip.add("A1A 0B11");
```

```
//Valid ZIP codes
AUzip.add("0200");
AUzip.add("7312");
AUzip.add("2415");
AUzip.add("3206");;
```

```
//Invalid ZIP codes
AUzip.add("0300");
AUzip.add("7612");
UKzip.add("2915");
UKzip.add("12345-67890");
```

```
String regexUK ="^[A-Z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}$";
```

```
String regexIndia ="^\\d{3}\\s?\\d{3}$";
```

```
String regexAu = "^(0[289][0-9]{2})|([1345689][0-9]{3})|(2[0-8][0-9]{2})|(290[0-9])|(291[0-4])|(7[0-4][0-9]{2})|(7[8-9][0-9]{2})$";
```

```
String regexCAN = "^(?!.*[DFIOQU])[A-VXY][0-9][A-Z] ?[0-9][A-Z][0-9]$";
```

```
Pattern pattern = Pattern.compile(regexUK);
```

```
Pattern pattern2 = Pattern.compile(regexIndia);
```

```
Pattern pattern3 = Pattern.compile(regexAu);
```

```
Pattern pattern4 = Pattern.compile(regexCAN);
```

```
for (String zip : CANzips)
```

```
{
```

```
    Matcher matcher = pattern4.matcher(zip);
```

```
    if(matcher.matches())
```

```
    {
```

```
        System.out.println("ZIPCODE "+ zip +" Matches "+ matcher.matches());
```

```
    }
```

```
    else
```

```
    {
```

```
        System.out.println("ZIPCODE "+ zip +" does not match "+
```

```
matcher.matches());
```

```
    }
```

```
}
```

```
for (String zip : UKzips)
```

```
{
```

```
    Matcher matcher = pattern.matcher(zip);
```

```
    if(matcher.matches())
```

```
    {
```

```
        System.out.println("ZIPCODE "+ zip +" Matches "+ matcher.matches());
```

```
    }
```

```
    else
```

```
    {
```

```
        System.out.println("ZIPCODE "+ zip +" does not match "+
```

```
matcher.matches());
```

```
    }
```

```
}
```

```
for (String zip : INDzips)
```

```
{
```

```
    Matcher matcher = pattern2.matcher(zip);
```

```
    if(matcher.matches())
```



```

        {
            System.out.println("ZIPCODE "+ zip +" Matches "+ matcher.matches());
        }
        else
        {
            System.out.println("ZIPCODE "+ zip +" does not match "+
matcher.matches());
        }
    }

    for (String zip : AUzips)
    {
        Matcher matcher = pattern3.matcher(zip);
        if(matcher.matches())
        {
            System.out.println("ZIPCODE "+ zip +" Matches "+ matcher.matches());
        }
        else
        {
            System.out.println("ZIPCODE "+ zip +" does not match "+
matcher.matches());
        }
    }

}

}

```

Output:

```

ZIPCODE K1A 0B1 Matches true
ZIPCODE B1Z 0B9 Matches true
ZIPCODE K1A 0D1 does not match false
ZIPCODE W1A 0B1 does not match false
ZIPCODE Z1A 0B1 does not match false
ZIPCODE SW1W 0NY Matches true
ZIPCODE PO16 7GZ Matches true
ZIPCODE GU16 7HF Matches true
ZIPCODE L1 8JQ Matches true
ZIPCODE Z1A 0B1 does not match false
ZIPCODE A1A 0B11 does not match false
ZIPCODE 2915 does not match false
ZIPCODE 12345-67890 does not match false
ZIPCODE 400 117 Matches true
ZIPCODE 600 117 Matches true
ZIPCODE 623 4315 does not match false
ZIPCODE 400050 Matches true

```

ZIPCODE 0200 does not match false  
ZIPCODE 7312=-3535 does not match false  
ZIPCODE 2415 4545 does not match false  
ZIPCODE 320645 Matches true  
ZIPCODE 0200 Matches true  
ZIPCODE 7312 Matches true  
ZIPCODE 2415 Matches true  
ZIPCODE 3206 Matches true  
ZIPCODE 0300 does not match false  
ZIPCODE 7612 does not match false

# Limit the Length of Text

Pattern:

```
^(?i)[A-Z]{1,10}$
```

Explanation:

```
^      # State the position at the beginning of the string.
(?i)   # case insensitivity
[A-Z]   # Equal one letter from A to Z
{1,10}  # between 1 and 10 times.
$       # State position at the end of the string
```

Sample Code:

```
package com.sample.regexsample;

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForInputRange {

    public static void main(String[] args) {
        List<String> names = new ArrayList<String>();

        names.add("KAREN");
        names.add("JAVACRAZY");
        names.add("KARENWORKSFROMHOME"); //Incorrect
        names.add("KARU123MYNAMEATHOME"); //Incorrect

        String regex = "[A-Z]{1,10}$";

        Pattern pattern = Pattern.compile(regex);

        for (String name : names)
        {
            Matcher matcher = pattern.matcher(name);
            System.out.println(matcher.matches());
        }

    }
}
```

Output:

true  
true  
false  
false

# Limit the Number of Input Text

Pattern:

```
^\\W*(?:\\w+\\b\\W*){2,10}$
```

Explanation:

**^** # State the position at the beginning of the string.  
**\\W\*** # Optional non word character  
**\\w+** # word character  
**\\b** # boundary character  
**\\W\*** # One more time an optional non word character  
**{2,10}** # Limit between 2 to 10 characters  
**\$** # State the position at the end of the String

Sample Code:

```
package com.sample.regexsample;
```

```
import java.util.ArrayList;
```

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

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForInputRange {
```

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

```
        List<String> inputs = new ArrayList<String>();
```

```
        inputs.add("KAREN"); //Incorrect
```

```
        inputs.add("JAVA CRAZY");
```

```
        inputs.add("KAREN NEX FROM ITALY");
```

```
        inputs.add("test whether number of words in input text is between some minimum and  
maximum limit"); //Incorrect
```

```
        String regex = "^\\W*(?:\\w+\\b\\W*){2,10}$";
```

```
        Pattern pattern = Pattern.compile(regex);
```

```
        for (String input : inputs)
```

```
        {
```

```
            Matcher matcher = pattern.matcher(input);
```

```
            System.out.println("The input " + input + " matches " + matcher.matches());
```

```
        }
```

```
    }
```

```
}
```

Output:

The input KAREN matches false

The input JAVA CRAZY matches true

The input KAREN NEX FROM ITALY matches true

The input test whether number of words in input text is between some minimum and maximum limit matches false

# Limit the Number of Lines in Text

## Pattern

```
^(?>[^\r\n]*(?>\r\n?|\n)){0,3}[^\r\n]*$
```

## Explanation

<code>^</code>	<code># Assert position at the beginning of the string.</code>
<code>(?&gt;</code>	<code># Group but don't capture or keep backtracking positions:</code>
<code>[^\r\n]*</code>	<code># Match zero or more characters except CR and LF.</code>
<code>(?&gt;</code>	<code># Group but don't capture or keep backtracking positions:</code>
<code>\r\n?</code>	<code># Match a CR, with an optional following LF (CRLF).</code>
<code> </code>	<code># Or:</code>
<code>\n</code>	<code># Match a standalone LF character.</code>
<code>)</code>	<code># End the noncapturing, atomic group.</code>
<code>{0,4}</code>	<code># End group; repeat between zero and four times.</code>
<code>[^\r\n]*</code>	<code># Match zero or more characters except CR and LF.</code>
<code>\$</code>	<code># Assert position at the end of the string.</code>

## Sample Code:

Verification for zero to three input line

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForInputLines {

    public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();

        StringBuilder builder1 = new StringBuilder();
        builder1.append("Test Line 1");
        builder1.append("\n");
        builder1.append("Test Line 2");
        builder1.append("\n");

        StringBuilder builder2 = new StringBuilder();
        builder2.append("Test Line 1");
        builder2.append("\n");
        builder2.append("Test Line 2");
        builder2.append("\n");
        builder2.append("Test Line 3");
        builder2.append("\n");
    }
}
```

```

builder2.append("Test Line 4");
builder2.append("\n");

String regex = "\\A(?:[^\r\n]*(?:\r\n?\n)){0,3}[^\r\n]*\z";

Pattern pattern = Pattern.compile(regex);

Matcher matcher = pattern.matcher(builder.toString());
System.out.println("Input 1 Zero lines " +matcher.matches());

Matcher matcher1 = pattern.matcher(builder1.toString());
System.out.println("Input 2 Zero lines " +matcher1.matches());

Matcher matcher2 = pattern.matcher(builder2.toString());
System.out.println("Input 3 Four lines " +matcher2.matches());

    }

}

```

Output:

```

Input 1 Zero lines  true
Input 2 Zero lines  true
Input 3 Four lines  false

```

## Quiz

1. Write programs to verify 6 lines and 8 lines input feeds.

## Solution

Sample Code For 6 lines:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForInputLines {
```

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

```
        StringBuilder builder = new StringBuilder();
```

```
        StringBuilder builder1 = new StringBuilder();
```

```
        builder1.append("Test Line 1");
```



```
builder1.append("\n");  
builder1.append("Test Line 2");  
builder1.append("\n");
```

```
StringBuilder builder2 = new StringBuilder();  
builder2.append("Test Line 1");  
builder2.append("\n");  
builder2.append("Test Line 2");  
builder2.append("\n");  
builder2.append("Test Line 3");  
builder2.append("\n");  
builder2.append("Test Line 4");  
builder2.append("\n");
```

```
StringBuilder builder3 = new StringBuilder();  
builder3.append("Test Line 1");  
builder3.append("\n");  
builder3.append("Test Line 2");  
builder3.append("\n");  
builder3.append("Test Line 3");  
builder3.append("\n");  
builder3.append("Test Line 4");  
builder3.append("\n");  
builder3.append("Test Line 5");  
builder3.append("\n");  
builder3.append("Test Line 6");  
builder3.append("\n");
```

```
StringBuilder builder4 = new StringBuilder();  
builder4.append("Test Line 1");  
builder4.append("\n");  
builder4.append("Test Line 2");  
builder4.append("\n");  
builder4.append("Test Line 3");  
builder4.append("\n");  
builder4.append("Test Line 4");  
builder4.append("\n");  
builder4.append("Test Line 5");  
builder4.append("\n");  
builder4.append("Test Line 6");  
builder4.append("\n");  
builder4.append("Test Line 7");  
builder4.append("\n");
```

```
String regex = "\\A(?:[\\r\\n]*(?>\\r\\n?|\\n)){0,6}[\\r\\n]*\\z";
```

```
Pattern pattern = Pattern.compile(regex);
```

```

Matcher matcher = pattern.matcher(builder.toString());
System.out.println("Input 1 Zero lines " +matcher.matches());

Matcher matcher1 = pattern.matcher(builder1.toString());
System.out.println("Input 2 Two lines " +matcher1.matches());

Matcher matcher2 = pattern.matcher(builder2.toString());
System.out.println("Input 3 Four lines " +matcher2.matches());

Matcher matcher3 = pattern.matcher(builder3.toString());
System.out.println("Input 4 Six lines " +matcher3.matches());

Matcher matcher4 = pattern.matcher(builder4.toString());
System.out.println("Input 5 Seven lines " +matcher4.matches());

    }

}

```

Output:

```

Input 1 Zero lines  true
Input 2 Two lines  true
Input 3 Four lines  true
Input 4 Six lines  true
Input 5 Seven lines false

```

Sample Code For 8 Lines:

```

package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForInputLines {

    public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();

        StringBuilder builder1 = new StringBuilder();
        builder1.append("Test Line 1");
        builder1.append("\n");
        builder1.append("Test Line 2");
        builder1.append("\n");
    }
}

```

```
StringBuilder builder2 = new StringBuilder();  
builder2.append("Test Line 1");  
builder2.append("\n");  
builder2.append("Test Line 2");  
builder2.append("\n");  
builder2.append("Test Line 3");  
builder2.append("\n");  
builder2.append("Test Line 4");  
builder2.append("\n");
```

```
StringBuilder builder3 = new StringBuilder();  
builder3.append("Test Line 1");  
builder3.append("\n");  
builder3.append("Test Line 2");  
builder3.append("\n");  
builder3.append("Test Line 3");  
builder3.append("\n");  
builder3.append("Test Line 4");  
builder3.append("\n");  
builder3.append("Test Line 5");  
builder3.append("\n");  
builder3.append("Test Line 6");  
builder3.append("\n");
```

```
StringBuilder builder4 = new StringBuilder();  
builder4.append("Test Line 1");  
builder4.append("\n");  
builder4.append("Test Line 2");  
builder4.append("\n");  
builder4.append("Test Line 3");  
builder4.append("\n");  
builder4.append("Test Line 4");  
builder4.append("\n");  
builder4.append("Test Line 5");  
builder4.append("\n");  
builder4.append("Test Line 6");  
builder4.append("\n");  
builder4.append("Test Line 7");  
builder4.append("\n");
```

```
StringBuilder builder5 = new StringBuilder();  
builder5.append("Test Line 1");  
builder5.append("\n");  
builder5.append("Test Line 2");  
builder5.append("\n");  
builder5.append("Test Line 3");  
builder5.append("\n");
```

```
builder5.append("Test Line 4");
builder5.append("\n");
builder5.append("Test Line 5");
builder5.append("\n");
builder5.append("Test Line 6");
builder5.append("\n");
builder5.append("Test Line 7");
builder5.append("\n");
builder5.append("Test Line 8");
builder5.append("\n");
```

```
StringBuilder builder6 = new StringBuilder();
builder6.append("Test Line 1");
builder6.append("\n");
builder6.append("Test Line 2");
builder6.append("\n");
builder6.append("Test Line 3");
builder6.append("\n");
builder6.append("Test Line 4");
builder6.append("\n");
builder6.append("Test Line 5");
builder6.append("\n");
builder6.append("Test Line 6");
builder6.append("\n");
builder6.append("Test Line 7");
builder6.append("\n");
builder6.append("Test Line 8");
builder6.append("\n");
builder6.append("Test Line 9");
builder6.append("\n");
```

```
String regex = "\\A(?:>[^\r\n]*(?:>\r\n?|\n)){0,8}[^\r\n]*\z";
```

```
Pattern pattern = Pattern.compile(regex);
```

```
Matcher matcher = pattern.matcher(builder.toString());
System.out.println("Input 1 Zero lines " + matcher.matches());
```

```
Matcher matcher1 = pattern.matcher(builder1.toString());
System.out.println("Input 2 Two lines " + matcher1.matches());
```

```
Matcher matcher2 = pattern.matcher(builder2.toString());
System.out.println("Input 3 Four lines " + matcher2.matches());
```

```
Matcher matcher3 = pattern.matcher(builder3.toString());
```

```
System.out.println("Input 4 Six lines " +matcher3.matches());

Matcher matcher4 = pattern.matcher(builder4.toString());
System.out.println("Input 5 Seven lines " +matcher4.matches());

Matcher matcher5 = pattern.matcher(builder5.toString());
System.out.println("Input 6 eight lines " +matcher5.matches());

Matcher matcher6 = pattern.matcher(builder6.toString());
System.out.println("Input 7 nine lines " +matcher6.matches());

    }

}
```

Output:

```
Input 1 Zero lines  true
Input 2 Two lines   true
Input 3 Four lines  true
Input 4 Six lines   true
Input 5 Seven lines true
Input 6 eight lines true
Input 7 nine lines  false
```

# Match Something at the Start and/or the End of Line

Pattern

```
^wordToSearch
```

Explanation:

```
^          # State the position at the beginning of the String
wordToSearch # The word to be searched
```

Pattern:

```
wordToSearch$
```

Explanation:

```
wordToSearch # The word to be searched
$           # State the position at the end of the string.
```

MultiLine Search Pattern

```
(?m)^wordToSearch
```

Explanation:

```
(?m)      # Enable multiline mode
^          # State the position at the beginning of the String
wordToSearch # The word to be searched
```

Pattern

```
(?m)wordToSearch$
```

Explanation:

```
(?m)      # Enable multiline mode
wordToSearch # The word to be searched
$           # State the position at the end of the string.
```

Sample Code for single line mode:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForMultiLineSearch {
```

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

```
        String content =  "begin here to start, and go there to end\n" +  
        "come here to begin, and end there to finish\n" +  
        "begin here to start, and go there to end";
```

```
        String regex  =  "^begin";  
        String regex2 =  "end$";
```

```
        Pattern pattern =  Pattern.compile(regex,  
Pattern.CASE_INSENSITIVE);
```

```
        Matcher matcher =  pattern.matcher(content);  
        System.out.println("Finding a word at beginning");  
        while (matcher.find())  
        {  
            System.out.print("Start index: " + matcher.start());  
            System.out.print(" End index: " + matcher.end() + " ");  
            System.out.println(matcher.group());  
        }
```

```
        Pattern pattern2 =  Pattern.compile(regex2,  
Pattern.CASE_INSENSITIVE);
```

```
        Matcher matcher2 =  pattern2.matcher(content);  
        System.out.println("Finding a word at end");  
        while (matcher2.find())  
        {  
            System.out.print("Start index: " + matcher2.start());  
            System.out.print(" End index: " + matcher2.end() + " ");  
            System.out.println(matcher2.group());  
        }
```

```
    }
```

```
}
```

Output:

Finding a word at beginning  
Start index: 0 End index: 5 begin  
Finding a word at end

Start index: 122 End index: 125 end

Sample Code for multi Line mode:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForMultiLineSearch2 {
```

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

```
        String content = "begin here to start, and go there to end\n" +  
        "come here to begin, and end there to finish\n" +  
        "begin here to start, and go there to end";
```

```
        String regex3 = "(?m)^begin";
```

```
        String regex4 = "(?m)end$";
```

```
        Pattern pattern3 = Pattern.compile(regex3, Pattern.CASE_INSENSITIVE);
```

```
        Matcher matcher3 = pattern3.matcher(content);
```

```
        System.out.println("Finding a word at start in multiline mode");
```

```
        while (matcher3.find())
```

```
        {
```

```
            System.out.print("Start index: " + matcher3.start());
```

```
            System.out.print(" End index: " + matcher3.end() + " ");
```

```
            System.out.println(matcher3.group());
```

```
        }
```

```
        Pattern pattern4 = Pattern.compile(regex4, Pattern.CASE_INSENSITIVE);
```

```
        Matcher matcher4 = pattern4.matcher(content);
```

```
        System.out.println("Finding a word at end in multiline mode");
```

```
        while (matcher4.find())
```

```
        {
```

```
            System.out.print("Start index: " + matcher4.start());
```

```
            System.out.print(" End index: " + matcher4.end() + " ");
```

```
            System.out.println(matcher4.group());
```

```
        }
```

```
    }
```

```
}
```

Output:

Finding a word at start in multiline mode

Start index: 0 End index: 5 begin

Start index: 85 End index: 90 begin

Finding a word at end in multiline mode



Start index: 37 End index: 40 end  
Start index: 122 End index: 125 end

# Match Whole Words Using Word Boundaries

This is mainly used when searching for words in a huge junk of text like log files.

Pattern:

`\bword\b`

Explanation

`\b` # indicates the boundary

Word # words to be searched

Some additional information how “\b” works. It usually matches in these three positions:

1. Before the first character in the data, if the first character is a word character
2. After the last character in the data, if the last character is a word character
3. Between two characters in the data, where one is a word character and the other is not a word character

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForBoundary {
```

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

```
        String data1 = "Today, java is object oriented language";
```

```
        String regex = "\\bjava\\b";
```

```
        Pattern pattern = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
```

```
        Matcher matcher = pattern.matcher(data1);
```

```
        while (matcher.find())
```

```
        {
```

```
            System.out.print("Start index: " + matcher.start());
```

```
            System.out.print(" End index: " + matcher.end() + " ");
```

```
            System.out.println(matcher.group());
```

```
        }
```

```
    }
```

```
}
```

Output:

Start index: 7 End index: 11 java



# Validating the Trademark™ Symbol

Pattern:

\u2122

Explanation:

This is the Unicode character for trademark symbol

Sample Code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForTrademark {

    public static void main(String[] args)
    {
        String data1 = "Searching in trademark character ™ is so easy when you know it.";

        String regex = "\u2122";

        Pattern pattern = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher(data1);
        while (matcher.find())
        {
            System.out.print("Start index: " + matcher.start());
            System.out.print(" End index: " + matcher.end() + " ");
            System.out.println(matcher.group());
        }
    }
}
```

Output:

Start index: 33 End index: 34 ™

# Validating Currency Symbol

Pattern:

`\p{Sc}`

Explanation:

The Unicode symbol used for currency symbols

Sample Code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForCurrency {

    public static void main(String[] args)
    {
        String content = "Let's find the symbols or currencies : $ Dollar, € Euro, ¥ Yen";

        String regex = "\\p{Sc}";

        Pattern pattern = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher(content);
        while (matcher.find())
        {
            System.out.print("Start index: " + matcher.start());
            System.out.print(" End index: " + matcher.end() + " ");
            System.out.println(" : " + matcher.group());
        }
    }
}
```

Output:

Start index: 39 End index: 40 : \$

Start index: 49 End index: 50 : €

Start index: 57 End index: 58 : ¥

# Chapter 18

## Common Mistakes While Using Regex

The following are the usual mistakes when we use REGEX:

## Mistake When Using a Simple String Matching

For example, let us consider we want to search the string: “asdfdfdasdfdfdf” for occurrences of “dfd”. If we look at the string, it says we can find it four times in the String.

Let’s use a regular expression to evaluate and find out what our little Java program says.

Sample Code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForMistakes {

    public static void main(String[] args) {

        String source = "asdfdfdasdfdfdf";
        Pattern pattern = Pattern.compile("dfd");
        Matcher matcher = pattern.matcher(source);
        int hits = 0;
        while (matcher.find()) {
            hits++;
        }
        System.out.println(hits);
    }
}
```

Output:

2 instead of 4

So what is the problem? Either our code is wrong, or the logic works differently than expected. And indeed, it does.

The first important rule while using a regular expressions in Java is: The search runs from left to right, and if a character has been used in search it will not be reused. So when we see “dfdfd” we used the first three letters for the match, and only “fd” remains, which is no match any more.

Hence we need to be careful while writing the expressions which are to be evaluated.

## Careful Usage of "\" Character

Let us consider that our application wants to read the IP addresses out of some long text that we have received from a service.

First point to be considered is the definition of , how an IP address can look like. We have 4 pairs of numbers separated by dots ranging from 0 to 255.

Let's write this as a regular expression. This might not be the best solution, if you have any better or more elegant feel free to leave a comment.

Pattern: `\b(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)\{3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b`

Basically it takes the allowed numbers for the first three blocks ,separated by dot and then quantified by 3. And the final block with the same pattern but without the trailing dot.

Sample Code:

```
package com.sample.regexsample;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
public class DemoForIPMistake {
```

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

```
        String logtext = "asdfsfgewg 215.2.125.32 alkejo 234 oij8982jld" +  
            "kja.lkjwech . 24.33.125.234 kadjeladfeladkj";
```

```
        String regexpatter = "\b(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)\{3}" +  
            "(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b";
```

```
        Pattern pattern = Pattern.compile(regexpatter);
```

```
        Matcher matcher = pattern.matcher(logtext);
```

```
        while (matcher.find()) {
```

```
            System.out.println(logtext.substring(matcher.start(),matcher.end()));
```

```
        }
```

```
    }
```

```
}
```

Output:

It will print nothing because the “\b” character is not used with \ backlash and the boundary is not defined



Proper code:

```
package com.sample.regexsample;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DemoForIPMistake {

    public static void main(String[] args) {

        String logtext = "asdfesgewg 215.2.125.32 alkejo 234 oij8982jld" +
            "kja.lkjwech . 24.33.125.234 kadjeladkfjeladj";

        String regexpatter = "\\b(?:?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\\.)}{3}" +
            "(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\\b";

        Pattern pattern = Pattern.compile(regexpatter);
        Matcher matcher = pattern.matcher(logtext);

        while (matcher.find()) {
            System.out.println(logtext.substring(matcher.start(),matcher.end()));
        }

    }
}
```

Output:  
215.2.125.32  
24.33.125.234

You have to take care to escape the \ properly, otherwise you'll end with an empty result or an invalid escape sequence error.

# Usage of Character Sequence

A mistake most usually made by basic regular expression users is to try to save a few characters by using the character class range `<[A-z]>` when we want to use all the upper and lower cases of alphabets. At first glance, this might seem like a very good trick to allow all uppercase and lowercase letters. However, the ASCII character table includes several punctuation characters in positions between the A–Z and a–z ranges. Hence, `<[A-z]>` is actually equivalent to `<[A-Z[\]^_`a-z]>`. So we might end up allowing the in between characters though our requirement is to allow only characters or might not get the desired output.

## 1. `java.lang.IllegalStateException`: No match found while using Regex Named Groups

This exception can be expected when we are trying to use named groups in java regex validation or matching.

The Error log will show the message like below:

```
Exception in thread "main" java.lang.IllegalStateException: No match found
    at java.util.regex.Matcher.group(Unknown Source)
```

The root cause of this error is ,we are probably not using “`matcher.matches()`” before fetching the named group from matcher.

Let us consider below piece of code:

```
List<String> dates = new ArrayList<String>();
dates.add("02/31/2011"); //Invalid date
dates.add("02/27/2011"); //Valid date
String regex = "^(?<month>[0-3]?[0-9])/(?<day>[0-3]?[0-9])/(?<year>(?:[0-9]{2})?[0-9]{2})$";
Pattern pattern = Pattern.compile(regex);
for (String date : dates)
{
    Matcher matcher = pattern.matcher(date);
    //This is the root cause of error. If this step is not done then we would get illegal state!
    matcher.matches();
    //Get date parts here
    String day = matcher.group(2);
    String month = matcher.group(1);
    String year = matcher.group(3);
    String formattedDate = month + "/" + day + "/" + year;
    System.out.println("Date to check : " + formattedDate);
}
```

## Other Common Mistakes When Using Regex

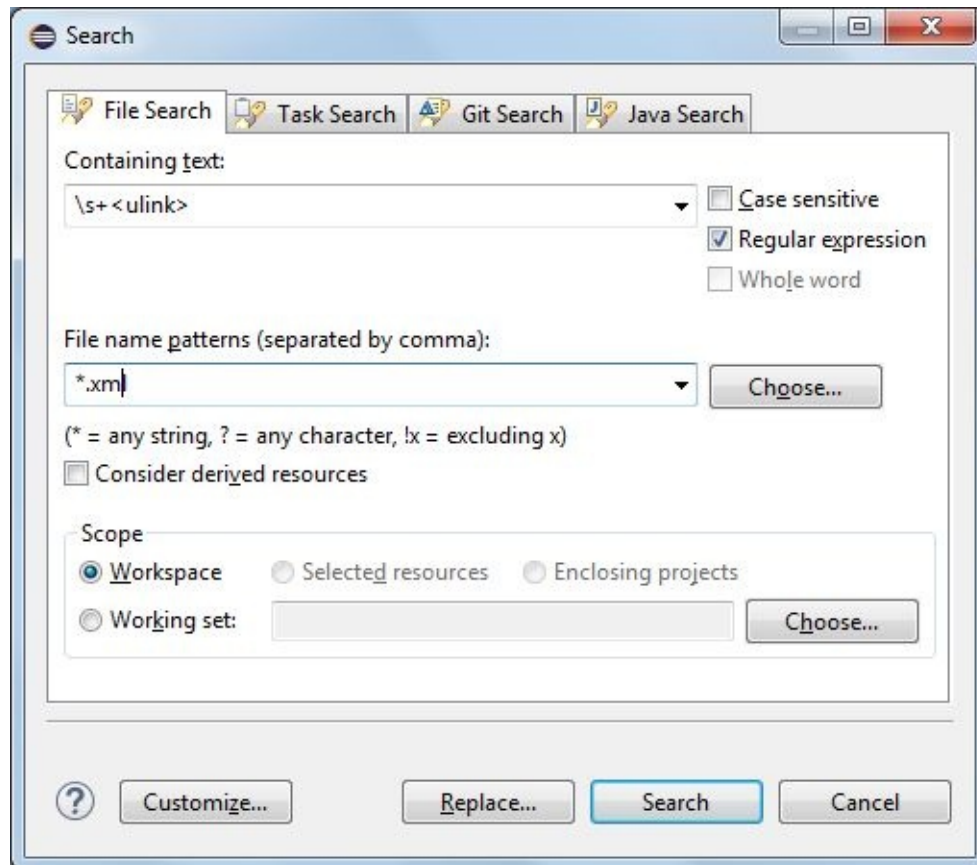
1. Forgetting to enclose the regular expression in “”.
2. Including spaces in the regular expressions (other than spaces you want).
3. Not escaping special characters you want treated literally: e.g. '()' instead of '\(\)'.  
`re.compile('()')`
4. Forgetting the `^` and `$` when you want an entire line to match a regular expression rather than some substring of the line.  
`re.compile('abc')`
5. Forgetting that something `*` includes the null string. For example, the regular expression `(aaa|bbb)*` matches every line!

## Chapter 19

# Regex in Eclipse

The Eclipse IDE allows to perform search and replace across a set of files using regular expressions. In Eclipse use the Ctrl+H shortcut to open the Search dialog.

Select the File Search tab and check the Regular expression flag before entering your regular expression. You can also specify the file type and the scope for the search and replace operation.



The above screenshots demonstrate how to search for the `</ulink>` XML tag with leading whitespace and how to remove the whitespace.

# Chapter 20

## In Closing

In a nutshell, Regular expressions provide an extremely flexible and powerful text processing system. We describe the regular expression as a textual pattern used to search in text. We do so by "matching" the regular expression against the text. The result of matching a regular expression against a text is either a true / false value, specifying if the regular expression matched the text, or a set of matches - one match for every occurrence of the regular expression found in the text.

For instance, we could use a regular expression to search an HTML page for email addresses, URLs, telephone numbers etc. This would be done by matching different regular expressions against the HTML page. The result of matching each regular expression against the HTML page would be a set of matches - one set of matches for each regular expression (each regular expression may match more than one time).

But we need to understand that:

1. Regular expressions are not easy to use at first
  - The individual pieces are not hard, but it takes a lot of practice (and tears why not?) to learn to put them together correctly
2. Regular expressions form a miniature programming language
  - It's a different kind of programming language than Java, and requires you to learn new thought patterns
  - In Java you can't just use a regular expression; you have to first create Patterns and Matchers
  - Java's syntax for String constants doesn't help, either
  - Despite all this, regular expressions bring so much power and convenience to String manipulation that they are well worth the effort of learning.

To conclude, imagine if all computer problems or projects were like this, where every single problem you encountered, no matter how simple or complex, was only written as English because nobody understood the symbols. This would drive us crazy, even if you didn't know the basic of programming.

Thankfully, humans invented symbols to succinctly describe the things, that symbols are best at describing, and left human languages to describe the rest. The symbolic languages act as building blocks to construct complex structures in a short space that would take entire books in English to describe.

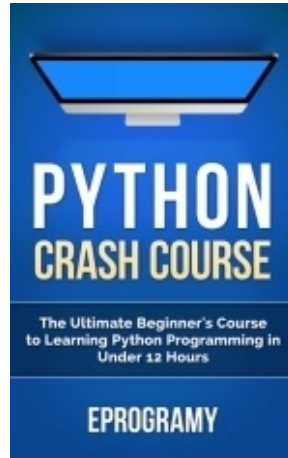
***Eprogramy***

*PD:* One more thing. Here in Eprogramy we want to give you a gift. If you enjoy Java Regex as much as

we do, you'll probably love Python too. So In the next section you will find a preview our **"PYTHON CRASH COURSE - The Ultimate Beginner's Course to Learning Python Programming in Under 12 Hours"**

I know you'll love it!

You can find it on Amazon, under our name, *Eprogramy*, or by following this link:



**<http://www.amazon.com/Python-Ultimate-Beginners-Learning-Programming-ebook/dp/B00ZCRSCFG/>**

# **Preview of PYTHON CRASH COURSE - The Ultimate Beginner's Course to Learning Python Programming in Under 12 Hours**



# **Introduction**

## **Welcome to Your New Programming Language**

So, you've decided to learn Python Programming? Well, congratulations and welcome to your new Programming Language! You're going to love it!

In Eprogramy we believe that the foundation of a good education is to provide the tools able to open the doors of the future. It is indisputable that the world has entered an era in which we all have everything at the fingertips. Social needs have changed. Thus, today communication has changed. Communication opens doors, so it is our challenge to learn how to communicate. In Eprogramy we want you to learn how to communicate with a new language: The programming language. This language allows us to use our PC or notebook to create.

Maybe the road is difficult and hard, but we will give the reader the tools to make learning enjoyable and fruitful. We'll show all aspects necessary to learn how to program. From the ABC's to the solution of the most common programming problems and much, much more. Always with the help of numerous examples that will contribute to a better understanding

We know what we do and believe that you are fully capable of incorporating our teachings.

The doors of the future are here. Let's go step by step together.

Let's get started!

Eprogramy Team

# **Chapter 1**

## **Python Programming Language**

# History of Python

Python is known as the go-to language for beginners as it is recommended by computer programmers around the globe as a good language for beginners to learn. This should not be misinterpreted for its powerful nature.

The Python language was created in 1990 by Guido Von Rossum at Stichting Mathematisch Centrum in the Netherlands. The language itself has been actually developed by a large team of volunteers and is available to modify. In the development stage that it was in, it had classes with inheritance, exception handling, functions, and the core datatypes of list, dict, str and more.

In May 2000, Guido and the Python development team moved to BeOpen.com to form the BeOpen PythonLabs team. In the same year, the PythonLabs team moved to Digital Creations which is now known as Zope Corporation.

Python 2.0 was released on the 16<sup>th</sup> of October in the year 2000 with features including a garbage collector which helps maintain memory handling related issues in programming. The great thing about Python was that it is backed by a community and it has a transparency behind the users that utilize the Python language.

Soon after, Python 3.0 which was a major backwards-incompatible release was released on December 3<sup>rd</sup> 2008 after a long period of testing. The major features of Python 3.0 also have been backported to backwards-compatible Python 2.6 and 2.7. In this guide, we'll be going over Python 3.4.

Python Version Release Dates:

- **Python 1.0 - January 1994**
  - Python 1.5 - December 31, 1997
  - Python 1.6 - September 5, 2000
- **Python 2.0 - October 16, 2000**
  - Python 2.1 - April 17, 2001
  - Python 2.2 - December 21, 2001
  - Python 2.3 - July 29, 2003
  - Python 2.4 - November 30, 2004
  - Python 2.5 - September 19, 2006
  - Python 2.6 - October 1, 2008
  - Python 2.7 - July 3, 2010
- **Python 3.0 - December 3, 2008**
  - Python 3.1 - June 27, 2009
  - Python 3.2 - February 20, 2011
  - Python 3.3 - September 29, 2012
  - Python 3.4 - March 16, 2014

# What is Python?

Python is a freely available object-oriented, high-level programming language with dynamic semantics. Many programmers say great things about Python because of the increased productivity that it provides since the edit-test-debug cycle is incredibly fast compared to other programming languages.

The Python language has a simple, easy to learn syntax which is empowered with many English words for easier readability and helps for increased productivity and efficiency. When coding in Python, it feels more like you are writing out the solution to a problem in your own thoughts rather than trying to refer to ambiguous symbols that are required in the language to commit to certain functionalities.

Python could be used to automate measurements and process data interactively. The language is able to handle large databases and compute large numbers strain-free compared to many other programming languages. It can be used as an internal scripting language so that it is executed for certain functions by other programs. By learning Python, you will be able to write complex software like intricate web crawlers. It is truly an all-purpose language.

The great thing about Python is that it has a giant library for web crawling and is used a lot for its capability in scraping the web. A web crawler is simply a program that can navigate the web depending on the parameters set out for it and scrapes content that you would like for it to scrape.

All in all, Python can be easy to pick up whether you're a beginner in programming or an experienced one for other languages. It's a fast, friendly and easy to learn language but don't mistake that for its powerful nature.

## Chapter 2

# Installation of Python

In order to install Python on to a machine, you must download the following:

1. Python Interpreter
2. Python IDE

The download of these two tools will put you on your way to becoming a Python programmer. An IDE (integrated development environment) is a packaged application program used by programmers because it contains necessary tools in order to process and execute code. An IDE contains a code editor, a compiler, a debugger, and a graphical user interface (GUI). There are many different type of IDE's but the most commonly used one is PyCharm for Python. Before attaining the PyCharm IDE from JetBrains, you must first install the Python Interpreter (IDLE version 3.4) on to your system so PyCharm is able to detect Python on the computer.

In this guide, it would be recommended to use PyCharm because it will be used by this guide. If you use the same IDE, you can follow me easier. In order to download the Python Interpreter, please use the following link:

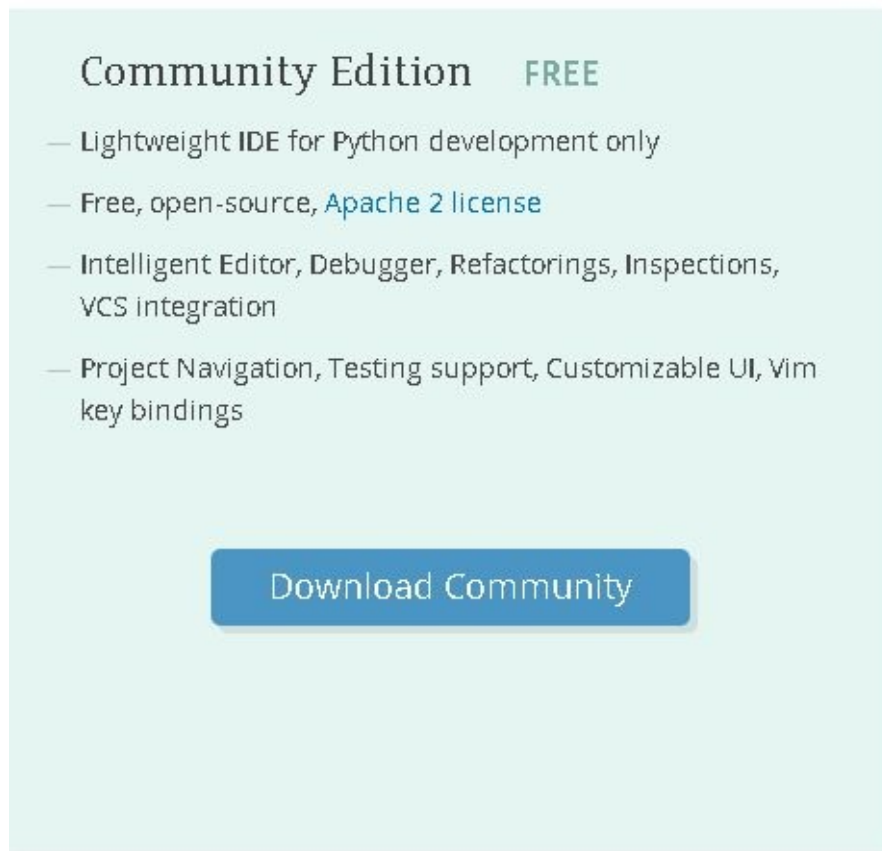
<https://www.python.org/downloads/>

Once you have reached this link, you will have to find this:



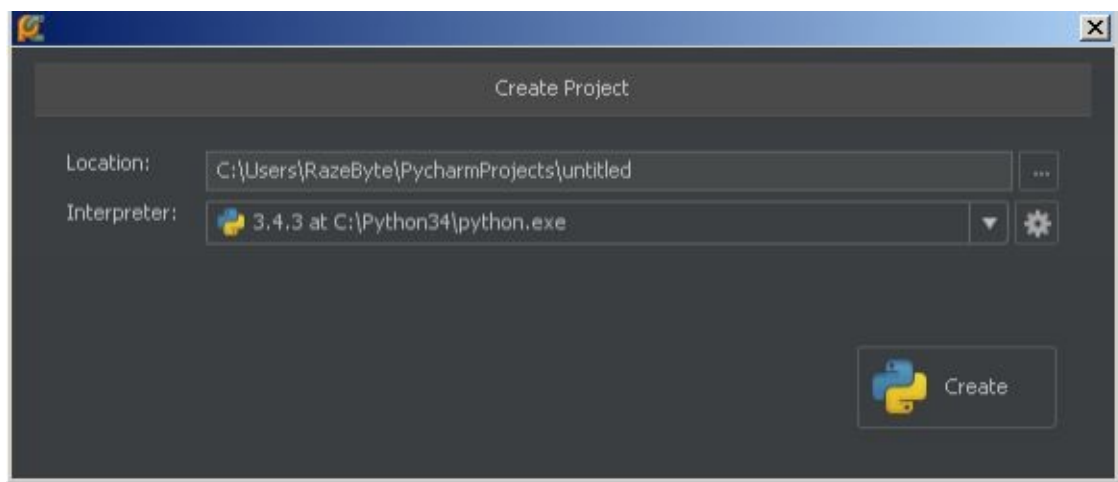
Then, click download and a file will be downloaded accordingly. Once you have this file, execute it and follow the instructions on the wizard to install the Python Interpreter.

Once the Python Interpreter has been installed, we can move on to downloading PyCharm. In order to install PyCharm, please visit the following link:



You should now be on this screen (or something similar) – click the “Download Community” button and continue to download PyCharm. During this process, it will automatically detect the Python 3.4 IDLE but in rare cases you might need to specify the directory that you installed Python in.

Once you have completed the installation of PyCharm and the Python 3.4 IDLE, go to File → New Project. You will reach this screen:

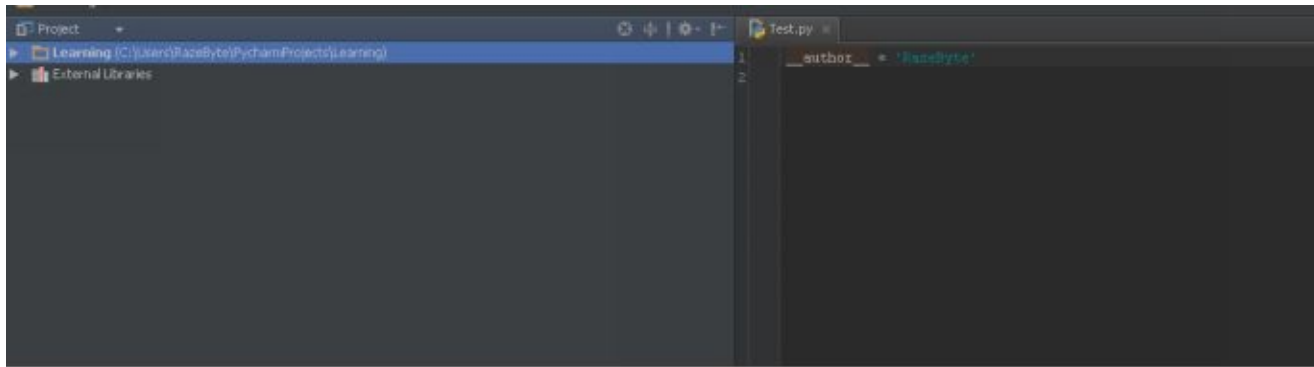


You can rename the “untitled” portion of the Location field to a specific project name. Then click the “Create” button and you will end up being in an empty project. The next thing we must do is create a new Python file which you can do by going to:

File → New → Python File

Set the name of your python file accordingly and it will show under your project name in the Project Explorer. Now double click on the file and you will be met with a page where you can start coding Python.

This is how it should look like:



You have successfully completed the installation of Python!

# Chapter 3

## Python Language Structure

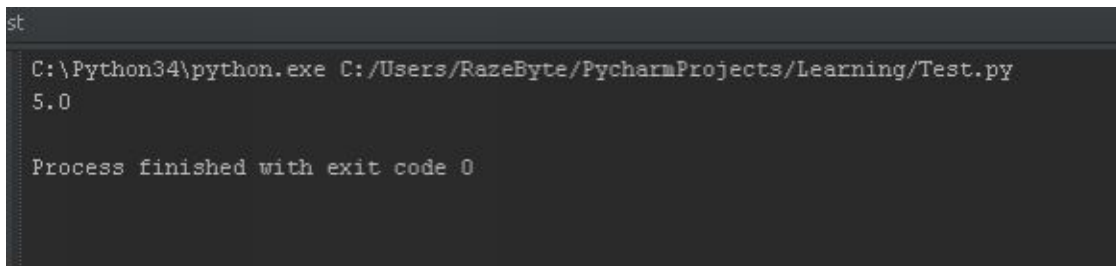
Using the sample code as an example, we will learn how Python as a programming language is structured through the following sample code:

```
import math

class Program:
    def Execute(self):
        x = math.sqrt(25)
        print(x)

run = Program()
run.Execute()
```

Output:

A screenshot of a terminal window with a dark background. The first line shows the command prompt 'st' followed by the command 'C:\Python34\python.exe C:/Users/RazeByte/PycharmProjects/Learning/Test.py'. The second line shows the output '5.0'. The third line shows the message 'Process finished with exit code 0'.

In order to actually execute the code, you must use the shortcut alt + shift + f10 or execute it using the Run menu item in the main navigation bar in PyCharm.

The first line import math is using a special keyword **import** which allows the programmer to import tools from the Python library which aren't included by default when coding. After the keyword **import** - the programmer specifies a specific directory that is within the Python library. In this case, we specified the math directory, if we wanted to specifically specify something within the System directory, we would add a dot separator (period) and then added a sub directory name. If the library's folders are more intricate, you must use notation like the following:

```
import bs4 from BeautifulSoup
```

The BeautifulSoup library is an example of this where we want to import the bs4 directory from the BeautifulSoup library of directories, which is why the “**from**” keyword is used in the beginning of the program.

In our case, we imported the math module because it contains functions (you will learn more about what these are later on in the guide – they can be defined as functions or methods). For the sake of a brief explanation, a function is simply a block of code that has a series of steps to complete a certain function and can be called to compute numbers for us. Many libraries contain functions that are able to be called so we don't have to code our own functions from scratch. In this case, we used the square root function



from the math module. In this case, we used `math.sqrt()` and then printed out the value it returned relative to the input it received.

We first typed in `math` to indicate the module we are used and then a dot separator to reference the function within the module which is the `sqrt` function. The `sqrt` function takes in a value in its parameter or parenthesis. While the `sqrt` function computes the number 25 and returns the value of 5, this value is stored in the `x` variable which is why it has an equal sign to show that it is equal to the value that is computed from the `sqrt` function. Again, this might not make a lot of sense to you but will be covered in depth later on in this guide.

Now as we run through each statement, one thing to note is that in order to run the next line, we must have a separate line for each statement. The compiler (process to convert code in to readable code that the computer can understand; binary) will then know when statements end and when they start through the use of line breaks.

Now let's back up and talk about the class `Program`: line.

**class** - A class can be thought of as a "section" of code. For example:

**Section:** Everything in here is the contents of the section. Again, you will gain a better understanding how classes are useful in Inheritance and Polymorphism.

**Program** - This second element of this important line is "Program" which simply is a custom name that the user can redefine. You can call this anything, as all it is doing is giving the "Section" or "Class" a name. You can think of it this way:

Section Name: Essay/Contents

In code, it would be presented:

```
class Program: Code
```

Another thing you must be scratching your head looking at is the colon: ":" and "}" - all that the colon does is simply tells the compiler when a specific section starts. This could be thought of as when someone is writing an essay and they have to start their sentences with the appropriate words or indent their paragraphs.

One thing to note is that the spacing in code does matter. If you are creating a class, the content of that class will be indented once using the tab key in order for it to interpret that the code that the class has authority over is the tabbed code underneath it. Any code that is not tabbed underneath class is not part of that class. This goes the same for any function you create as well. As shown in the above example, a function is defined with a colon and tabbed content underneath it to indicate that it is part of that function. The way Python works is that it has classes, which are sections and functions which are subsections of the class. These classes can be declared as they are declared in the main program after the class is declared and then called with the functions that they carry within them. In this case, it is `Execute`. Again, this might not make sense right now but it will later on in the guide.

# About the Author



Eprogramy Academy was created by a group of professionals from various areas of IT with a single purpose: To provide knowledge in the 3.0 era.

Education is changing as well as social needs. Today, in the era of information, education should provide the tools to create and to solve problems in a 3.0 world.

At Eprogramy we understand this and work to give people appropriate responses in this context. Keeping this objective in mind, we offer a wide variety of courses to teach the basics of many programming languages. We believe that anyone can learn a programming language and apply the lessons in order to solve problems. In our academy we provide the essential tools to allow everyone to incorporate into the daily life a set of solutions obtained through programming.

Possibilities and solutions are endless.

In short, at Eprogramy we are committed to help everybody to decodify the messages of the future.