**Regular Expressions**

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Regular expressions are not easy to use at first. It’s a bunch of punctuation, not words. The individual pieces are not hard, but it takes practice to learn to put them together correctly.

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

Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions. The package includes the following classes:

* Pattern Class - Defines a pattern (to be used in a search)
* Matcher Class - Used to search for the pattern
* PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

**Example:**

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main {

public static void main(String[] args) {

Pattern pattern = Pattern.compile("w3schools", Pattern.CASE\_INSENSITIVE);

Matcher matcher = pattern.matcher("Visit W3Schools!");

boolean matchFound = matcher.find();

if(matchFound) {

System.out.println("Match found");

} else {

System.out.println("Match not found");

}

}

}

**Output**

Match found

In this example, The word "w3schools" is being searched for in a sentence.

First, the pattern is created using the Pattern.compile() method. The first parameter indicates which pattern is being searched for and the second parameter has a flag to indicates that the search should be case-insensitive. The second parameter is optional.

The matcher() method is used to search for the pattern in a string. It returns a Matcher object which contains information about the search that was performed.

The find() method returns true if the pattern was found in the string and false if it was not found.

## **Regular Expression Patterns**

The first parameter of the Pattern.compile() method is the pattern. It describes what is being searched for.

Brackets are used to find a range of characters:

[abc]-Find one character from the options between the brackets

[^abc]-Find one character NOT between the brackets

[0-9]-Find one character from the range 0 to 9

## **Metacharacters**

Metacharacters are characters with a special meaning:

**Metacharacter Description**

| Find a match for any one of the patterns separated by | as in:cat|dog|fish

. Find just one instance of any character

^ Finds a match as the beginning of a string as in: ^Hello

$ Finds a match at the end of the string as in: World$

\d Find a digit

\s Find a whitespace character

\b Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b

\uxxxx Find the Unicode character specified by the hexadecimal number xxxx

**Escaping Metacharacters**

A lot of special characters--parentheses, brackets, braces, stars, plus signs, etc.--are used in defining regular expressions; these are called metacharacters.

Suppose you want to search for the character sequence a\* (an a followed by a star)

"a\*"; doesn’t work; that means “zero or more as”

"a\\*"; doesn’t work; since a star doesn’t need to be escaped (in Java

String constants), Java just ignores the \

"a\\\*" does work; it’s the three-character string a, \, \*

Just to make things even more difficult, it’s illegal to escape

a non-metacharacter in a regular expression

## **Quantifiers**

Quantifiers define quantities:

**Quantifier Description**

n+ Matches any string that contains at least one n

n\* Matches any string that contains zero or more occurrences of n

n? Matches any string that contains zero or one occurrences of n

n{x} Matches any string that contains a sequence of *X* *n*'s

n{x,y} Matches any string that contains a sequence of X to Y *n*'s

n{x,} Matches any string that contains a sequence of at least X *n*'s

## **Java Date and Time APIs**

Java provide the date and time functionality with the help of two packages java.time and java.util. The package java.time is introduced in Java 8, and the newly introduced classes tries to overcome the shortcomings of the legacy java.util.Date and java.util.Calendar classes.

**Date Time API Classes**

The primary classes before Java 8 release were:

**Java.lang.System:** The class provides the currentTimeMillis() method that returns the current time in milliseconds. It shows the current date and time in milliseconds from January 1st 1970.

**java.util.Date:** It is used to show specific instant of time, with unit of millisecond.

**java.util.Calendar:** It is an abstract class that provides methods for converting between instances and manipulating the calendar fields in different ways.

**java.text.SimpleDateFormat:** It is a class that is used to format and parse the dates in a predefined manner or user defined pattern.

**java.util.TimeZone:** It represents a time zone offset, and also figures out daylight savings.

## **Display Current Date**

To display the current date, import the java.time.LocalDate class, and use its now() method:

Example

import java.time.LocalDate; // import the LocalDate class

public class Main {

public static void main(String[] args) {

LocalDate myObj = LocalDate.now(); // Create a date object

System.out.println(myObj); // Display the current date

}

}

**Output:**

2021-10-10

## **Display Current Time**

To display the current time (hour, minute, second, and nanoseconds), import the java.time.LocalTime class, and use its now() method:

**Example:**

import java.time.LocalTime; // import the LocalTime class

public class Main {

public static void main(String[] args) {

LocalTime myObj = LocalTime.now();

System.out.println(myObj);

}

}

**Output:**

23:03:59.247274

## **Display Current Date and Time**

To display the current date and time, import the java.time.LocalDateTime class, and use its now() method:

**Example:**

import java.time.LocalDateTime; // import the LocalDateTime class

public class Main {

public static void main(String[] args) {

LocalDateTime myObj = LocalDateTime.now();

System.out.println(myObj);

}

}

Output:

2021-10-10T23:03:59.287467

**Formatting Date and Time**

The "T" in the example above is used to separate the date from the time. You can use the DateTimeFormatter class with the ofPattern() method in the same package to format or parse date-time objects. The following example will remove both the "T" and nanoseconds from the date-time:

**Example**

import java.time.LocalDateTime; // Import the LocalDateTime class

import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class

public class Main {

public static void main(String[] args) {

LocalDateTime myDateObj = LocalDateTime.now();

System.out.println("Before formatting: " + myDateObj);

DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

String formattedDate = myDateObj.format(myFormatObj);

System.out.println("After formatting: " + formattedDate);

}

}

**Output:**

Before Formatting: 2021-10-10T23:03:59.288877  
After Formatting: 10-10-2021 23:03:59

**Drawbacks of existing Date/Time API's**

1. **Thread safety:** The existing classes such as Date and Calendar does not provide thread safety. Hence it leads to hard-to-debug concurrency issues that are needed to be taken care by developers. The new Date and Time APIs of Java 8 provide thread safety and are immutable, hence avoiding the concurrency issue from developers.
2. **Bad API designing:** The classic Date and Calendar APIs does not provide methods to perform basic day-to-day functionalities. The Date and Time classes introduced in Java 8 are ISO-centric and provides number of different methods for performing operations regarding date, time, duration and periods.
3. **Difficult time zone handling:** To handle the time-zone using classic Date and Calendar classes is difficult because the developers were supposed to write the logic for it. With the new APIs, the time-zone handling can be easily done with Local and ZonedDate/Time APIs.