**Assignment 5**

Introduction to Java Time:

• ISO 8601 Core Calendar – java.time

– LocalTime, LocalDate, LocalDateTime, ZonedDateTime, ...

– Clock, Instant, Duration, Period, ZoneId, Month, DayOfWeek, ...

• Parsing and Formatting – java.time.format

– DateTimeFormat, DateTimeFormatBuilder, standard formats, patterns, styles, ...

• TimeZone – java.time.zone

– ZoneRules, transitions, etc.

• Regional Calendars – java.time.chrono

– Chronology, Era, ChronoLocalDate, ChronoLocalDateTime, ChronoZonedDateTime, ...

– Japanese, ThaiBuddhist, Minguo, Hijrah calendars

• Framework – java.time.temporal

– Units, Fields, Adjusters, Temporal, TemporalAccessor, TemporalAmount, TemporalQuery,

Java Time API Design:

• Fluent API

• Immutable instances

• Thread safe

• Strong types

• Fit for purpose types

Vs

java.util Date and Calendar:

• Not Fluent

• Mutable instances – clone needed

• Not Thread safe

• Weakly typed calendars

• One size fits all API

Calendar Neutral API:

• The core java.time APIs are designed to avoid common errors

• For other calendars common assumptions may be invalid

– Don't assume the number of months of year

– Use the API to do all calendar arithmetic. Plus/minus days, months, years

– Do not assume roll-over at particular numbers of days-per-month or months-per-year

– Don't assume the week starts on Monday (ISO), use WeekFields.

– Don't assume the month numbers are bound to specific months, use

DateTimeFormatter to get names

• A Chronology is added to provide the correct semantics for the calendar

DateTimeFormatter Builder:

**Customizing parsing and formatting:**

• Factory used to build a template

for a sequence of fields to be parsed or formatted

• Literals, for example “/”, “-”, “:”, or any string

• Numeric values with control of width, sign, leading zeros

• Text indexed by locale and style – short, narrow, full, standalone

• Patterns indexed by locale - date, time, date-time

• Fraction control, with control of width and decimal points

• Field padding to width and character

• ZoneId, ZoneOffset, Chronology, Instant specialized fields

• Case sensitive vs. in-sensitive parsing

• Optional fields

• Default values if not present in input

• Strict vs. Lenient parsing mode

• Concatenation of Formatters

Tip: Comparing Date Time Values:

**When compareTo is not the same as timeline order**

• The familiar Comparable.compareTo follows Object.equals semantics

– Values in date time types are compared field by field; hour, minute, second, zone, etc.

– For simple date time types, the results are the same

– For ZonedDateTime and Calendar Neutral types, use time line order

• Time line order is equivalent to converting to Instant

– Normalizes the effects of different ZoneId/ZoneOffsets

– Normalizes the effects of different Chronologies

– isBefore, isAfter, isEqual methods provide timeline order comparisons

– ChronoLocalDateTime, ChronoZonedDateTime provide timeLineOrder Comparator

Tip: Converting from Joda-Time:

**Designs are very similar but not every Joda-Time function is supported:**

• Mapping of class names

– LocalDate, LocalTime, LocalDateTime are the same

– Joda-Time DateTime is java.time.ZonedDateTime

– Joda-Time DateTimeZone maps to java.time.ZoneId/ZoneRules/ZoneOffset

• Period and Duration

– Joda-Time Period is years to seconds; Java.time.Period is simpler, only ISO date-based

– Joda-Time Duration is same as java.time.Duration

• Java time is more open and flexible

– Adjusters and queries are new

– Ability to define own fields and units is essentially new

– Adding calendar systems is simple rather than very hard

Summary:

**New Improved Date Time API:**

• Fluent, Immutable, Thread Safe, Easy to use

• Strong typing with fit for purpose types

• Easy to use formatting and parsing

• Extensible with Units, Fields, and Chronologies

• Interoperable with java.util.Calendar

• Supports Regional Calendars

• The essential ISO Calendar for global business

Regular Expressions in Java:

* A regular expression is a kind of pattern that can be applied to text (Strings, in Java)
* A regular expression either matches the text (or part of the text), or it fails to match
* If a regular expression matches a part of the text, then you can easily find out which part
* If a regular expression is complex, then you can easily find out which parts of the regular expression match which parts of the text
* With this information, you can readily extract parts of the text, or do substitutions in the text
* Regular expressions are an extremely useful tool for manipulating text
* Regular expressions are heavily used in the automatic generation of Web pages

Example:

* The regular expression "[a-z]+" will match a sequence of one or more lowercase letters

[a-z] means any character from a through z, inclusive

+ means “one or more”

* Suppose we apply this pattern to the String "Now is the time"
* There are three ways we can apply this pattern:

1. To the entire string: it fails to match because the string contains characters other than lowercase letters
2. To the beginning of the string: it fails to match because the string does not begin with a lowercase letter
3. To search the string: it will succeed and match ow
4. If applied repeatedly, it will find is, then the, then time, then fail

import java.util.regex.\*;

public class RegexTest {

public static void main(String args[]) {

String pattern = "[a-z]+";

String text = "Now is the time";

Pattern p = Pattern.compile(pattern);

Matcher m = p.matcher(text);

while (m.find()) {

System.out.print(text.substring(m.start(), m.end()) + "\*");

}

}

}

Output: ow\*is\*the\*time\*

Boundary matchers:

These patterns match the empty string if at the specified position:

^ the beginning of a line

$ the end of a line

\b a word boundary

\B not a word boundary

\A the beginning of the input (can be multiple lines)

\Z the end of the input except for the final terminator, if any

\z the end of the input

\G the end of the previous match

Greedy quantifiers:

Assume *X* represents some pattern

Assume X represents some pattern

X? optional, X occurs once or not at all

X\* X occurs zero or more times

X+ X occurs one or more times

X{n} X occurs exactly n times

X{n,} X occurs n or more times

X{n,m} X occurs at least n but not more than m times

Types of Quantifiers:

A greedy quantifier will match as much as it can, and back off if it needs to

We’ll do examples in a moment

A reluctant quantifier will match as little as possible, then take more if it needs to

You make a quantifier reluctant by appending a ?:

X?? X\*? X+? X{n}? X{n,}? X{n,m}?

A possessive quantifier will match as much as it can, and never let go

You make a quantifier possessive by appending a +:

X?+ X\*+ X++ X{n}+ X{n,}+ X{n,m}+

Quantifier examples:

Suppose your text is aardvark

Using the pattern a\*ardvark (a\* is greedy):

The a\* will first match aa, but then ardvark won’t match

The a\* then “backs off” and matches only a single a, allowing the rest of the pattern (ardvark) to succeed

Using the pattern a\*?ardvark (a\*? is reluctant):

The a\*? will first match zero characters (the null string), but then ardvark won’t match

The a\*? then extends and matches the first a, allowing the rest of the pattern (ardvark) to succeed

Using the pattern a\*+ardvark (a\*+ is possessive):

The a\*+ will match the aa, and will not back off, so ardvark never matches and the pattern match fails

Capturing groups:

In regular expressions, parentheses are used for grouping, but they also capture (keep for later use) anything matched by that part of the pattern

Example: ([a-zA-Z]\*)([0-9]\*) matches any number of letters followed by any number of digits

If the match succeeds, \1 holds the matched letters and \2 holds the matched digits

In addition, \0 holds everything matched by the entire pattern

Capturing groups are numbered by counting their opening parentheses from left to right:

( ( A ) ( B ( C ) ) )

1 2 3 4

\0 = \1 = ((A)(B(C))), \2 = (A), \3 = (B(C)), \4 = (C)

Example: ([a-zA-Z])\1 will match a double letter, such as letter

Example:

Suppose word holds a word in English

Also suppose we want to move all the consonants at the beginning of word (if any) to the end of the word (so string becomes ingstr)

Pattern p = Pattern.compile("([^aeiou]\*)(.\*)");

Matcher m = p.matcher(word);

if (m.matches()) {

System.out.println(m.group(2) + m.group(1));

}

Note the use of (.\*) to indicate “all the rest of the characters”

Double backslashes:

Backslashes have a special meaning in regular expressions; for example, \b means a word boundary

Backslashes have a special meaning in Java; for example, \b means the backspace character

Java syntax rules apply first!

If you write "\b[a-z]+\b" you get a string with backspace characters in it--this is not what you want!

Remember, you can quote a backslash with another backslash, so "\\b[a-z]+\\b" gives the correct string

Note: if you read in a String from somewhere, this does not apply--you get whatever characters are actually there

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

Spaces:

There is only one thing to be said about spaces (blanks) in regular expressions, but it’s important:

Spaces are significant!

A space stands for a space--when you put a space in a pattern, that means to match a space in the text string

It’s a really bad idea to put spaces in a regular expression just to make it look better

Additions to the String class:

All of the following are public:

public boolean matches(String regex)

public String replaceFirst(String regex, String replacement)

public String replaceAll(String regex, String replacement)

public String[ ] split(String regex)

public String[ ] split(String regex, int limit)

If the limit n is greater than zero then the pattern will be applied at most n - 1 times, the array's length will be no greater than n, and the array's last entry will contain all input beyond the last matched delimiter.

If n is non-positive then the pattern will be applied as many times as possible

Note that these are all *postfix* operators, that is, they come *after* the operand