CHP-15
Java Date and Time

Java provides the **Date** class available in **java.util** package, this class encapsulates the current date and time.

The Date class supports two constructors. The first constructor initializes the object with the current date and time.

```
Date()
```

The following constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970

```
Date(long millisec)
```

Once you have a Date object available, you can call any of the following support methods to play with dates:

SN Methods with Description

1
boolean after(Date date)
Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false.

2
boolean before(Date date)
Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false.

3 **Object clone( )**
Duplicates the invoking Date object.

4
int compareTo(Date date)
Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date.

5 **int compareTo(Object obj)**
Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException.

6
boolean equals(Object date)
Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false.

7 **long getTime( )**
Returns the number of milliseconds that have elapsed since January 1, 1970.

CHAPTER

8 **int hashCode( )**
Returns a hash code for the invoking object.

9
void setTime(long time)
Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970

10 **String toString( )**

Converts the invoking Date object into a string and returns the result.

Getting Current Date & Time

This is very easy to get current date and time in Java. You can use a simple Date object with *toString()*method to print current date and time as follows:

```
import java.util.Date;
public class DateDemo{
public static void main(String args[]){
// Instantiate a Date object
Date date =newDate();
// display time and date using toString()
System.out.println(date.toString());
}
}
```

This would produce the following result:

```
MonMay0409:51:52 CDT 2009
```

Date Comparison:

There are following three ways to compare two dates:

- You can use getTime( ) to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.
- You can use the methods before( ), after( ), and equals( ). Because the 12th of the month comes before the 18th, for example, new Date(99, 2, 12).before(new Date (99, 2, 18)) returns true.
- You can use the compareTo( ) method, which is defined by the Comparable interface and implemented by Date.

Date Formatting using SimpleDateFormat:

SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting. For example:

```
import java.util.*;
import java.text.*;
public class DateDemo{
public static void main(String args[]){
Date dNow =newDate();
```

```
SimpleDateFormat ft =
newSimpleDateFormat("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
System.out.println("Current Date: "+ ft.format(dNow));
}
}
```

This would produce the following result:

```
CurrentDate:Sun2004.07.18 at 04:14:09 PM PDT
```

Simple DateFormat format codes:

To specify the time format, use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following:

Character	**Description**	**Example**
G	Era designator	AD
Y	Year in four digits	2001
M	Month in year	July or 07
D	Day in month	10
H	Hour in A.M./P.M. (1~12)	12
H	Hour in day (0~23)	22
M	Minute in hour	30
S	Second in minute	55
S	Millisecond	234
E	Day in week	Tuesday
D	Day in year	360
F	Day of week in month	2 (second Wed. in July)
W	Week in year	40

W	Week in month	1
A	A.M./P.M. marker	PM
K	Hour in day (1~24)	24
K	Hour in A.M./P.M. (0~11)	10
Z	Time zone	Eastern Standard Time
'	Escape for text	Delimiter
"	Single quote	`

Date Formatting using printf:

Date and time formatting can be done very easily using **printf** method. You use a two-letter format, starting with **t** and ending in one of the letters of the table given below. For example:

```
import java.util.Date;
public class DateDemo{
public static void main(String args[]){
// Instantiate a Date object
Date date =new Date();
// display time and date using toString()
String str =String.format("Current Date/Time : %tc", date );
System.out.printf(str);
}
}
```

This would produce the following result:

```
CurrentDate/Time:SatDec1516:37:57 MST 2012
```

It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the index of the argument to be formatted.

The index must immediately follow the % and it must be terminated by a $. For example:

```
import java.util.Date;
public class DateDemo{
public static void main(String args[]){
// Instantiate a Date object
Date date =new Date();
// display time and date using toString()
System.out.printf("%1$s %2$tB %2$td, %2$tY",
"Due date:", date);
}
}
```

This would produce the following result:

```
Due date:February09,2004
```

Alternatively, you can use the < flag. It indicates that the same argument as in the preceding format specification should be used again. For example:

```
import java.util.Date;
public class DateDemo{
public static void main(String args[]){
// Instantiate a Date object
Date date =new Date();
// display formatted date
System.out.printf("%s %tB %<te, %<tY",
"Due date:", date);
}
}
```

This would produce the following result:

```
Due date:February09,2004
```

Date and Time Conversion Characters:

Character	Description	Example
c	Complete date and time	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 date	2004-02-09
D	U.S. formatted date (month/day/year)	02/09/2004
T	24-hour time	18:05:19

r	12-hour time	06:05:19 pm
R	24-hour time, no seconds	18:05
Y	Four-digit year (with leading zeroes)	2004
y	Last two digits of the year (with leading zeroes)	04
C	First two digits of the year (with leading zeroes)	20
B	Full month name	February
b	Abbreviated month name	Feb
m	Two-digit month (with leading zeroes)	02
d	Two-digit day (with leading zeroes)	03
e	Two-digit day (without leading zeroes)	9
A	Full weekday name	Monday
a	Abbreviated weekday name	Mon
j	Three-digit day of year (with leading zeroes)	069
H	Two-digit hour (with leading zeroes), between 00 and 23	18
k	Two-digit hour (without leading zeroes), between 0 and 23	18
I	Two-digit hour (with leading zeroes), between 01 and 12	06
l	Two-digit hour (without leading zeroes), between 1 and 12	6
M	Two-digit minutes (with leading zeroes)	05
S	Two-digit seconds (with leading zeroes)	19
L	Three-digit milliseconds (with leading zeroes)	047
N	Nine-digit nanoseconds (with leading zeroes)	047000000
P	Uppercase morning or afternoon marker	PM
p	Lowercase morning or afternoon marker	pm
z	RFC 822 numeric offset from GMT	-0800

Z	Time zone	PST
s	Seconds since 1970-01-01 00:00:00 GMT	1078884319
Q	Milliseconds since 1970-01-01 00:00:00 GMT	1078884319047

There are other useful classes related to Date and time. For more details, you can refer to Java Standard documentation.

Parsing Strings into Dates:

The SimpleDateFormat class has some additional methods, notably parse( ) , which tries to parse a string according to the format stored in the given SimpleDateFormat object. For example:

```
import java.util.*;
import java.text.*;
public class DateDemo{
public static void main(String args[]){
SimpleDateFormat ft =new SimpleDateFormat("yyyy-MM-dd");
String input = args.length ==0?"1818-11-11": args[0];
System.out.print(input +" Parses as ");
Date t;
try{
t = ft.parse(input);
System.out.println(t);
}catch(ParseException e){
System.out.println("Unparseable using "+ ft);
}
}
}
```

A sample run of the above program would produce the following result:

```
$ java DateDemo
1818-11-11ParsesasWedNov1100:00:00 GMT 1818
$ java DateDemo2007-12-01
2007-12-01ParsesasSatDec0100:00:00 GMT 2007
```

Sleeping for a While:

You can sleep for any period of time from one millisecond up to the lifetime of your computer. For example, following program would sleep for 10 seconds:

```
import java.util.*;
public class SleepDemo{
public static void main(String args[]){
try{
System.out.println(new Date()+"\n");
Thread.sleep(5*60*10);
System.out.println(new Date()+"\n");
}catch(Exception e){
System.out.println("Got an exception!");
```

```
}
}
}
```

This would produce the following result:

```
SunMay0318:04:41 GMT 2009
SunMay0318:04:51 GMT 2009
```

Measuring Elapsed Time:

Sometimes, you may need to measure point in time in milliseconds. So let's rewrite above example once again:

```
import java.util.*;
public class DiffDemo{
public static void main(String args[]){
try{
long start =System.currentTimeMillis();
System.out.println(new Date()+"\n");
Thread.sleep(5*60*10);
System.out.println(new Date()+"\n");
longend=System.currentTimeMillis();
long diff =end- start;
System.out.println("Difference is : "+ diff);
}catch(Exception e){
System.out.println("Got an exception!");
}
}
}
```

This would produce the following result:

```
SunMay0318:16:51 GMT 2009
SunMay0318:16:57 GMT 2009
Differenceis:5993
```

GregorianCalendar Class:

GregorianCalendar is a concrete implementation of a Calendar class that implements the normal Gregorian calendar with which you are familiar. I did not discuss Calendar class in this tutorial, you can look standard Java documentation for this.

The **getInstance( )** method of Calendar returns a GregorianCalendar initialized with the current date and time in the default locale and time zone. GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for GregorianCalendar objects:

SN Constructor with Description

1 **GregorianCalendar()**

Constructs a default GregorianCalendar using the current time in the default time zone with the default locale.

2 **GregorianCalendar(int year, int month, int date)**

Constructs a GregorianCalendar with the given date set in the default time zone with the default locale.

3

GregorianCalendar(int year, int month, int date, int hour, int minute)

Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.

4

GregorianCalendar(int year, int month, int date, int hour, int minute, int second)

Constructs a GregorianCalendar with the given date and time set for the default time zone with the default

locale.
5 **GregorianCalendar(Locale aLocale)**
Constructs a GregorianCalendar based on the current time in the default time zone with the given locale.
6 **GregorianCalendar(TimeZone zone)**
Constructs a GregorianCalendar based on the current time in the given time zone with the default locale.
7 **GregorianCalendar(TimeZone zone, Locale aLocale)**
Constructs a GregorianCalendar based on the current time in the given time zone with the given locale.
Here is the list of few useful support methods provided by GregorianCalendar class:
SN Methods with Description
1 **void add(int field, int amount)**
Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.
2 **protected void computeFields()**
Converts UTC as milliseconds to time field values.
3 **protected void computeTime()**
Overrides Calendar Converts time field values to UTC as milliseconds.
4 **boolean equals(Object obj)**
Compares this GregorianCalendar to an object reference.
5 **int get(int field)**
Gets the value for a given time field.
6 **int getActualMaximum(int field)**
Return the maximum value that this field could have, given the current date.
7 **int getActualMinimum(int field)**
Return the minimum value that this field could have, given the current date.
8 **int getGreatestMinimum(int field)**
Returns highest minimum value for the given field if varies.
9 **Date getGregorianChange()**
Gets the Gregorian Calendar change date.
10 **int getLeastMaximum(int field)**
Returns lowest maximum value for the given field if varies.
11 **int getMaximum(int field)**
Returns maximum value for the given field.
12 **Date getTime()**
Gets this Calendar's current time.
13 **long getTimeInMillis()**
Gets this Calendar's current time as a long.

14 **TimeZone getTimeZone()**
Gets the time zone.
15 **int getMinimum(int field)**
Returns minimum value for the given field.
16 **int hashCode()**
Override hashCode.
17 **boolean isLeapYear(int year)**
Determines if the given year is a leap year.
18 **void roll(int field, boolean up)**
Adds or subtracts (up/down) a single unit of time on the given time field without changing larger fields.
19 **void set(int field, int value)**
Sets the time field with the given value.
20 **void set(int year, int month, int date)**
Sets the values for the fields year, month, and date.
21 **void set(int year, int month, int date, int hour, int minute)**
Sets the values for the fields year, month, date, hour, and minute.
22 **void set(int year, int month, int date, int hour, int minute, int second)**
Sets the values for the fields year, month, date, hour, minute, and second.
23 **void setGregorianChange(Date date)**
Sets the GregorianCalendar change date.
24 **void setTime(Date date)**
Sets this Calendar's current time with the given Date.
25 **void setTimeInMillis(long millis)**
Sets this Calendar's current time from the given long value.
26 **void setTimeZone(TimeZone value)**
Sets the time zone with the given time zone value.

27 **String toString()**
Return a string representation of this calendar.

Example:

```
import java.util.*;
public class GregorianCalendarDemo{
public static void main(String args[]){
String months[]={
"Jan","Feb","Mar","Apr",
"May","Jun","Jul","Aug",
"Sep","Oct","Nov","Dec"};
int year;
// Create a Gregorian calendar initialized
// with the current date and time in the
// default locale and timezone.
GregorianCalendar gcalendar =new GregorianCalendar();
// Display current time and date information.
System.out.print("Date: ");
```

```
System.out.print(months[gcalendar.get(Calendar.MONTH)]);
System.out.print(" "+ gcalendar.get(Calendar.DATE)+" ");
System.out.println(year = gcalendar.get(Calendar.YEAR));
System.out.print("Time: ");
System.out.print(gcalendar.get(Calendar.HOUR)+":");
System.out.print(gcalendar.get(Calendar.MINUTE)+":");
System.out.println(gcalendar.get(Calendar.SECOND));
// Test if the current year is a leap year
if(gcalendar.isLeapYear(year)){
System.out.println("The current year is a leap year");
}
else{
System.out.println("The current year is not a leap year");
}
}
}
```

This would produce the following result:

```
Date:Apr222009
Time:11:25:27
The current year is not a leap year
```

For a complete list of constant available in Calendar class, you can refer to standard Java documentation.

CHP-16
Java Regular Expressions

Java provides the java.util.regex package for pattern matching with regular expressions. Java regular

expressions are very similar to the Perl programming language and very easy to learn.
A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.
The java.util.regex package primarily consists of the following three classes:
- **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.

Capturing Groups:

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression (dog) creates a single group containing the letters "d", "o", and "g".
Capturing groups are numbered by counting their opening parentheses from left to right. In the expression ((A)(B(C))), for example, there are four such groups:
- ((A)(B(C)))
- (A)
- (B(C))
- (C)

To find out how many groups are present in the expression, call the groupCount method on a matcher object. The groupCount method returns an int showing the number of capturing groups present in the matcher's pattern.
CHAPTER

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by groupCount.

Example:

Following example illustrates how to find a digit string from the given alphanumeric string:
```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
public class RegexMatches
{
public static void main(String args[]){
// String to be scanned to find the pattern.
String line ="This order was places for QT3000! OK?";
String pattern ="(.*)(\\d+)(.*)";
// Create a Pattern object
Pattern r =Pattern.compile(pattern);
// Now create matcher object.
Matcher m = r.matcher(line);
if(m.find()){
System.out.println("Found value: "+ m.group(0));
System.out.println("Found value: "+ m.group(1));
System.out.println("Found value: "+ m.group(2));
}else{
System.out.println("NO MATCH");
}
}
}
```

This would produce the following result:

```
Found value:This order was places for QT3000! OK?
Found value:This order was places for QT300
Found value:0
```

Regular Expression Syntax:

Here is the table listing down all the regular expression metacharacter syntax available in Java:

Subexpression	**Matches**
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
\A	Beginning of entire string
\z	End of entire string
\Z	End of entire string except allowable final line terminator.

re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more of the previous thing
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?: re)	Groups regular expressions without remembering matched text.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\n	Back-reference to capture group number "n"
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\Q	Escape (quote) all characters up to \E

\E Ends quoting begun with \Q

Methods of the Matcher Class:

Here is a list of useful instance methods:

Index Methods:

Index methods provide useful index values that show precisely where the match was found in the input string:
SN Methods with Description

1 **public int start()**
Returns the start index of the previous match.
2 **public int start(int group)**
Returns the start index of the subsequence captured by the given group during the previous match operation.
3 **public int end()**
Returns the offset after the last character matched.
4
public int end(int group)
Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Study Methods:

Study methods review the input string and return a Boolean indicating whether or not the pattern is found:
SN Methods with Description
1 **public boolean lookingAt()**
Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2 **public boolean find()**
Attempts to find the next subsequence of the input sequence that matches the pattern.
3
public boolean find(int start
Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4 **public boolean matches()**
Attempts to match the entire region against the pattern.

Replacement Methods:

Replacement methods are useful methods for replacing text in an input string:
SN Methods with Description
1 **public Matcher appendReplacement(StringBuffer sb, String replacement)**
Implements a non-terminal append-and-replace step.
2 **public StringBuffer appendTail(StringBuffer sb)**
Implements a terminal append-and-replace step.
3
public String replaceAll(String replacement)
Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
4
public String replaceFirst(String replacement)
Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
5
public static String quoteReplacement(String s)
Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement s in the appendReplacement method of the Matcher class.

The start and end Methods:

Following is the example that counts the number of times the word "cats" appears in the input string:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexMatches
```

```
{
private static final String REGEX ="\\bcat\\b";
private static final String INPUT ="cat cat cat cattie cat";
public static void main(String args[]){
Pattern p =Pattern.compile(REGEX);
Matcher m = p.matcher(INPUT);// get a matcher object
int count =0;
while(m.find()){
count++;
System.out.println("Match number "+count);
System.out.println("start(): "+m.start());
System.out.println("end(): "+m.end());
}
}
}
```

This would produce the following result:

```
Match number 1
start():0
end():3
Match number 2
start():4
end():7
Match number 3
start():8
end():11
Match number 4
start():19
end():22
```

You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred.
The start method returns the start index of the subsequence captured by the given group during the previous match operation, and end returns the index of the last character matched, plus one.

The *matches* and *lookingAt* Methods:

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.
Both methods always start at the beginning of the input string. Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
p
ublic class RegexMatches
{
```

```
private static final String REGEX ="foo";
private static final String INPUT ="fooooooooooooooooo";
private static Pattern pattern;
private static Matcher matcher;
public static void main(String args[]){
pattern =Pattern.compile(REGEX);
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());
}
}
```

This would produce the following result:

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt():true
matches():false
```

The *replaceFirst* and *replaceAll* Methods:

The replaceFirst and replaceAll methods replace text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences.
Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexMatches
{
private static String REGEX ="dog";
private static String INPUT ="The dog says meow. "+"All dogs say meow.";
private static String REPLACE ="cat";
public static void main(String[] args){
Pattern p =Pattern.compile(REGEX);
// get a matcher object
Matcher m = p.matcher(INPUT);
INPUT = m.replaceAll(REPLACE);
System.out.println(INPUT);
}
}
```

This would produce the following result:

```
The cat says meow.All cats say meow.
```

The *appendReplacement* and *appendTail* Methods:

The Matcher class also provides appendReplacement and appendTail methods for text replacement.
Here is the example explaining the functionality:

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

```
import java.util.regex.Pattern;
public class RegexMatches
{
private static String REGEX ="a*b";
private static String INPUT ="aabfooaabfooabfoob";
private static String REPLACE ="-";
public static void main(String[] args){
Pattern p =Pattern.compile(REGEX);
// get a matcher object
Matcher m = p.matcher(INPUT);
StringBuffer sb =new StringBuffer();
while(m.find()){
m.appendReplacement(sb,REPLACE);
}
m.appendTail(sb);
System.out.println(sb.toString());
}
}
```

This would produce the following result:

```
-foo-foo-foo-
```

PatternSyntaxException Class Methods:

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern.
The PatternSyntaxException class provides the following methods to help you determine what went wrong:

SN Methods with Description

1 **public String getDescription()**
Retrieves the description of the error.

2 **public int getIndex()**

Retrieves the error index.
3 **public String getPattern()**
Retrieves the erroneous regular expression pattern.
4
public String getMessage()
Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern.