Useful Classes

Date, SimpleDateFormat ,Calendar, Math,

Random, StringTokenizer

Date and Time

- To work with date and time there are 3 important classes
  - **`Date`**
  - **`Calendar`**
  - **`GregorianCalendar`**
- All of these classes are in **`java.util`**
- **`GregorianCalendar`** is subclass of **`Calendar`**
- It is recommended that **`Calendar`** class be used whenever possible because most of the methods in **`Date`** class are deprecated.

java.util.Date

- **`Date`** object is represented internally a single long number which represents the number of milliseconds since January 1, 1970, 00:00:00 GMT.
- Most of its methods are deprecated because many of them are not amenable to internationalization.
- **`Constructors`**
  - **`Date()`** **`:`** Initializes to the nearest millisecond with system's current date and time measured with respect to January 1, 1970, 00:00:00 GMT
  - **`Date(long)`** : initializes it to the specified milliseconds since the January 1, 1970, 00:00:00 GMT

Date methods

- **boolean after(Date when)**
- **boolean before(Date when)**
- **long getTime()**

  Returns the number of milliseconds since January 1, 1970, 00:00:00(*Epoch)* GMT represented by this Date object.

- **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, 00:00:00

- **Object clone( )**
- **int compareTo(Object)**
- **String toString( )**

Example: Date

```
import java.util.Date;
public class DateEx{
public static void main(String[] args) {
    Date d= new Date();
    System.out.println(d);
    Date d1=new Date(d.getTime()+1000);
    System.out.println(d.after(d1));
    System.out.println(d.before(d1));
    System.out.println(d1.compareTo(d));
    System.out.println(d.compareTo(d1));}}
```

```
Fri Nov 18 14:14:36 IST 2011
false
true
1
-1
```


HCL

Formatting Dates

- **`java.text.SimpleDateFormat`** class is used for formatting and parsing dates in a locale-sensitive manner.
- In the constructor the date format can be specified using predefined letters that correspond to some meaning.
- Constructors :
  - **`SimpleDateFormat()`**
    - uses the default pattern and date format symbols for the default locale.
  - **`SimpleDateFormat(String pattern)`**
    - uses the given pattern and the default date format symbols for the default locale.

Pattern letters (JSE documentation)

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

`SimpleDateFormat` Methods

- **`final String format(Date date)`**
  - Formats a Date into a date/time string as per specification in the constructor.
- **`Date parse(String source) throws ParseException`**
  - Parses text from the beginning of the given string to produce a date based on the format specification in the constructor.
  - **`java.text.ParseException`** is a checked exception which is thrown if the expected string is not matching specified format
- **`Calendar getCalendar()`**
  - Gets the calendar associated with this date/time formatter.

Example: SimpleDateFormat

```
import java.text.*;
import java.util.Date;
public class Palindrome {
public static void main(String[] args) throws
ParseException {
      Date now = new Date( );
       SimpleDateFormat ft =
       new SimpleDateFormat ("E dd MMM yyyy 'at'
hh:mm:ss a zzz");
       System.out.println(t.format(now));
       SimpleDateFormat ft1 =
       new SimpleDateFormat ("dd.mm.yyyy");
      Date d= ft1.parse("10.7.1967");
       System.out.println(t.format(d));
}}
```

Fri 18 Nov 2011 at 02:53:09 PM IST
Tue 10 Jan 1967 at 12:07:00 AM IST

Formatting character for Date in `printf`

We have seen how to format numbers and string using ***`System.out.printf`*** *statement. Do you recall them?*

- **`System.out.printf`** statement can be used to display date in the desired format.
- The format characters for time and date are in the next slides.
- These characters must have prefix of 't' and 'T' conversions
- Example:

```
Date now = new Date( );

System.out.printf(" %1$tA %1$tB %1$td, %1$tY
 %1$tH:%1$tM:%1$tS %1$tp %1$tZ ",now);
```

This prints: **`Friday November 18, 2011 15:50:02 pm`**

`Calendar` and its subclass can also be used in place of **`Date`**

- Be careful when you use **`printf`**. Small error would cause **`UnknownFormatConversionException`** to be thrown at runtime

Date format characters (JSE)

	Purpose	Example
B	Locale-specific full month name,	"January","February"
b,h	Locale-specific abbreviated month name	"Jan","Feb"
A	Locale-specific full name of the day of the week,	"Sunday","Monday"
a	Locale-specific short name of the day of the week,	"Sun","Mon"
C	Four-digit year divided by 100, formatted as two digits with leading zero as necessary	00-99
Y	Year, formatted as at least four digits with leading zeros as necessary	, e.g. 0092 equals 92 CE for the Gregorian calendar
y	Last two digits of the year, formatted with leading zeros as necessary	00 - 99.
j	Day of year, formatted as three digits with leading zeros as necessary	001 - 366 for the Gregorian calendar
m	Month, formatted as two digits with leading zeros as necessary	01 - 12
d	Day of month, formatted as two digits with leading zeros as necessary	01 - 31
e	Day of month, formatted as two digits	1 - 31

HCL

Time format characters (JSE)

	Purpose	Example
H	Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary	00 - 23
I	Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary,	01 - 12
k	Hour of the day for the 24-hour clock	0 - 23
l	Hour for the 12-hour clock	1 - 12
M	Minute within the hour formatted as two digits with a leading zero as necessary	00 - 59
S	Seconds within the minute, formatted as two digits with a leading zero as necessary	00 - 60
L	Millisecond within the second formatted as three digits with leading zeros as necessary	000 - 999
N	Nanosecond within the second, formatted as nine digits with leading zeros as necessary	000000000 – 999999999
p	Locale-specific morning or afternoon marker in lower case. Use prefix 'T' for upper case.	“am" or "pm".
z	RFC 822 style numeric time zone offset from GMT	0800
Z	A string representing the abbreviation for the time zone. The Formatter's locale will supersede the locale of the argument (if any).	
s	Seconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC	
Q	Milliseconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC	

Calendar and GregorianCalendar

- **`Calendar`** is an abstract class.
- **`GregorianCalendar`** is a concrete subclass of **`Calendar`**. This class provides the standard calendar system used by most of the world.
- To create an instance of **`Calendar`** class **`getInstance()`** static method is used. This a **`Calendar`** object with the system's date and time.
- Gregorian Calendar internally the value is stored as time in millisecond represented by a value that is an offset from the *Epoch*, January 1, 1970 00:00:00.000 GMT.
- Constructor:
  - **`GregorianCalendar()`**
  - **`GregorianCalendar(int year, int month, int dayOfMonth,[int hourOfDay, int minute, int second])`**

Calendar/ GregorianCalendar members

- **`boolean after(Object when)`**
- **`boolean before(Object when)`**
- **`void clear()`**

  Sets all the calendar field values and the time value to *Epoch*
- **`final int get(int field)`**
- **`final void set(int field, int value)`**

  field is defined using static constants(next slide) used to get components of date like year, month etc.
- **`void set(int year, int month, int date)`**
- **`void set(int year, int month, int date, int hourOfDay, int minute)`**

- **`Date getTime()`**

- **`boolean equals(Object obj)`**
- **`Object clone()`**
- **`int compareTo(Calendar anotherCalendar)`** → same as **`Date`**
- **`String toString()`**

GregorianCalendar members

- **void add(int field, int amount**)

  Adds or subtracts the specified amount of time to the given calendar field. This is an abstract method in **Calendar**.

- **void roll(int field, int amount)**

  Adds the specified (signed) amount to the specified calendar field without changing larger fields. A negative amount means to roll down.

- Understanding the differences between the two methods by example

```
Calendar cal = Calendar.getInstance();
cal.set(2011,2,1);
// line 1
System.out.println(cal.getTime());
```

1. `cal.roll(Calendar.DATE, -1);` at line 1 prints **Thu Mar 31 10:19:55 IST 2011**
2. `cal.add(Calendar.DATE, -1);` at line 1 prints **Mon Feb 28 10:23:05 IST 2011**

- **boolean isLeapYear(int year)**

- Static constants
  - **JANUARY to DECEMBER (values from 0 to 11)**
  - **SUNDAY to SATURDAY (values from 1 to 7)**
  - **MONTH,YEAR,DATE**
  - **HOUR,MINUTE,SECOND MILLISECOND**
  - **DAY_OF_WEEK**

Examples: using Calendar

```
/*cal value depends on the current system date and
time.*/
Calendar cal =Calendar.getInstance();

System.out.println(cal instanceof GregorianCalendar);
// true
System.out.println(cal.getTime());
//Fri Nov 18 16:13:12 IST 2011

cal.clear();
System.out.println(cal.getTime())
//Thu Jan 01 00:00:00 IST 1970
```

Example: Get date components

```
import java.util.*;
class Test{
    public static void main(String str[]){
    String
      month[]={"Jan","Feb","Mar","Apr","May","Jun","Jly"
      ,"Aug","Sep","Oct","Nov","Dec"};
    Calendar c = Calendar.getInstance();
    String m=month[c.get(Calendar.MONTH)];
    System.out.println(m);
    System.out.println(c.get(Calendar.DATE));
    System.out.print(c.get(Calendar.HOUR)+":");
    System.out.print(c.get(Calendar.MINUTE)+":");
    System.out.println(c.get(Calendar.SECOND));
}
}
```

```
Nov
18
5:4:52
```

Example: Week computation

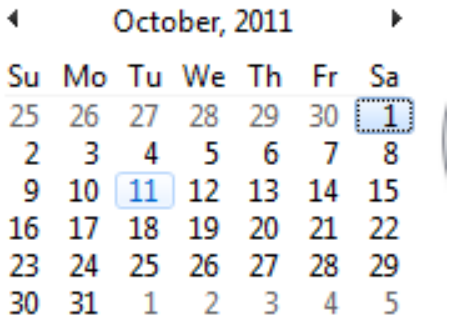
```
import java.util.*;
class Test{
public static void main(String str[]){
   String
   week[]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
    GregorianCalendar c = new GregorianCalendar();

    c.set(2011,10,11);
    int wk=c.get(c.DAY_OF_WEEK);
    System.out.println(wk);
    System.out.println(week[wk-1]);
    System.out.println(c.isLeapYear(2000));
}
}
```

```
6
Fri
true
```

Tell me why?

- `c.set(2011,10,11)` is a Tuesday but the code displays it as Friday. Why?



- If you look at the static constants once again you will find that values for month begin from 0 and not from 1. So 0 indicates January. Therefore 10 here represents November and not October.
- Also look at the week computation. Sun to Sat static constants range from 1 to 7. So the array's 0^{th} index position does not work for week computation as it worked for month computation in the example before this.

Example: Adding to dates

```
class Test{
public static void main(String str[]){
    String month[]=

    {"Jan","Feb","Mar","Apr","May","Jun","Jly","Aug",
    "Sep","Oct","Nov","Dec"};
    GregorianCalendar  c=new
    System.out.prinGregorianCalendar(2010,11,1);
    c.add(Calendar.MONTH,5);
    tln(month[c.get(Calendar.MONTH)]);
    c.add(Calendar.YEAR, 1);
    System.out.println(c.get(Calendar.YEAR));

}
```

May
2012

java.lang.Math

- A **final** class containing methods to perform mathematical operation. All the methods are **static**.
- **XXX abs(XXX a)** where **XXX** is **int, long double, float**
- Trigonometric operations like
  - **double acos(double a)**
  - **double asin(double a)**
  - **double atan(double a)**
  - **double cos(double a)**
  - **double cosh(double x)**
  - **double sin(double a)**
  - **double sinh(double x)**
  - **double tan(double a)**
  - **double tanh(double x)**

Static constants

E (2.718281828459045)

PI (3.141592653589793)

HCL

- Power of a number
  - **`double pow(double a, double b)`**
  - **`double sqrt(double a)`**
  - **`double cbrt(double a)`**
- Round/truncation operations
  - **`double ceil(double a)`**
  - **`double floor(double a)`**
  - **`long round(double a)`**
  - **`int round(float a)`**
- **`double random()`** returns a number between 0 and 1.
- Log
  - **`double log(double a)`**
  - **`double log10(double a)`**

Example: using Math

```
double a = Math.PI;
System.out.println( Math.round(a));
System.out.println( Math.abs(a));
System.out.println( Math.ceil(a));
System.out.println(Math.exp(0));
System.out.println( Math.floor(a));
System.out.println( Math.max(11,4.5));
System.out.println( Math.min(11,4.5));
System.out.println("Power = " + Math.pow(10,3));
System.out.println( Math.sqrt(49));
```

```
3
3.141592653589793
4.0
1.0
3.0
11.0
4.5
Power = 1000.0
7.0
```

Test your understanding

- Can you guess what will the code display ?

```
double a = 3.5;
System.out.println( Math.round(a));
System.out.println( Math.ceil(a));
System.out.println( Math.floor(a))
```

- That should have been easy. What will it display if the number is negative?

```
double a = -3.5;
```

Recall

- *Do you remember that we used Math to generate random numbers?*
- *How will you generate random numbers between 1 to 9?*

java.util.Random

- This class is used to generate a stream of pseudorandom numbers.
- Constructor
  - **`Random()`**
  - **`Random(long seed)`**
  - The seed is the initial value of the internal state of the pseudorandom number generator which is maintained by method **`next(int)`**
- Methods
  - **`xxx nextXxx()`** where **`xxx`** is **`boolean, int, long, double or float`**
  - **`int nextInt(int n)`**
  - **`void nextBytes(byte[] bytes)`**: Generates random bytes and places them into a user-supplied byte array

Example 1 : using Random class

```
Random r2= new Random();
System.out.println( r2.nextInt(100));
System.out.println( r2.nextFloat());
byte b[]=new byte[5];
r2.nextBytes(b);
for(byte c:b)
System.out.print(c+ ",");
```

```
38
0.3184449
-13,0,-2,14,-90,
```

Example 2 : using Random class

- If two instances of **`Random`** are created with the same seed, and the same sequence of method calls is made for each, they will generate and return same sequences of numbers.

```
Random r1= new Random(100);
 Random r2= new Random(100);

 System.out.print("1st instance: ");
 System.out.print( r1.nextInt(10)+" ");
 System.out.println( r1.nextInt(100));

 System.out.print("2st instance: ");
 System.out.print( r2.nextInt(10)+" ");
 System.out.println( r2.nextInt(100));
```

```
1st instance: 5 50
2st instance: 5 50
```

java.util.StringTokenizer

- Allows a string to be split into tokens based on delimiter. Default delimiter is space.
- Constructor
  - **`StringTokenizer(String s)`**
  - **`StringTokenizer(String s, String d)`**
  - **`StringTokenizer(String s, String d, boolean flag)`**
- Methods
  - **`String nextToken()`**
  - **`String nextToken(String delim)`**
  - **`int countTokens()`**
  - **`boolean hasMoreTokens`**
  - **`boolean equals(Object o)`**
  - **`String toString()`**

Example 1 : using StringTokenizer

```
StringTokenizer st = new StringTokenizer("europe asia
   america");
while (st.hasMoreTokens()) {
System.out.println(st.nextToken());}
```

Prints

europe

asia

america

Example 2 : using StringTokenizer

Example converts comma separated number into double

```
import java.util.*;
public class Test {
public static void main(String[] args) {
double number=0;
int pow=1;
StringTokenizer st =
            new StringTokenizer("23,44,345.8", ",");
while (st.hasMoreTokens()) {
String str1="";
String str=st.nextToken();
```

```
if(str.contains("."))
str1=str.substring(0,str.indexOf("."));
else
str1=str;
pow=(int)Math.pow(10,str1.length());
        number=number*pow+Double.parseDouble(str);
}
System.out.println(number);
}
}
```

Prints 2344345.8

Are you able to work out the logic for this by yourself?

Test your knowledge

- Do you remember any other way in which strings can be split?

Performance issue

- We saw two ways to split a string:
  - Using **`StringTokenizer`**
  - Using **`split()`** method of **`String`**
- Between the two, **`split()`** method of **`String is believed to give better performance.`**
- Java documentation states that

  "**`StringTokenizer`** is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the **`split`** method of **`String`** or the **`java.util.regex`** package instead."