TEMA 5 : PAQUETES Y LIBRERÍAS

"Comentar el código es como limpiar el cuarto de baño;
nadie quiere hacerlo,
pero el resultado es siempre una experiencia más agradable para uno mismo y sus invitados"

Librería o paquete en java es lo mismo: conjunto de clases relacionadas entre sí. Por ejemplo el paquete java.io agrupa todas las clases que permiten a un programa realizar operaciones de entrada y salida de datos.

Un paquete nos permite organizar las clases en grupos.

Dos clases pueden llamarse igual si se encuentran en paquetes diferentes

Todas las clases de un mismo paquete se "ven" entre ellas. Si queremos utilizar una clase que se encuentra en otro utilizamos la sentencia import.

import java.io.*; //importamos todas las clases de ese paquete

import.java.io.BufferedReader: // importamos sólo esa clase del paquete
Como el rendimiento no se degrada por importar el paquete entero, es lo que se suele hacer.

Tabla 2.1. Librerías Java

Paquete o librería	Descripción
java.io	Librería de Entrada/Salida. Permite la comunicación del programa con ficheros y periféricos.
java.lang	Paquete con clases esenciales de Java. No hace falta ejecutar la sentencia import para utilizar sus clases. Librería por defecto.
java.util	Librería con clases de utilidad general para el programador.
java.applet	Librería para desarrollar *applets*.
java.awt	Librerías con componentes para el desarrollo de interfaces de usuario.
java.swing	Librerías con componentes para el desarrollo de interfaces de usuario. Similar al paquete awt.
java.net	En combinación con la librería java.io, va a permitir crear aplicaciones que realicen comunicaciones con la red local e Internet.
java.math	Librería con todo tipo de utilidades matemáticas.
java.sql	Librería especializada en el manejo y comunicación con bases de datos.
java.security	Librería que implementa mecanismos de seguridad.
java.rmi	Paquete que permite el acceso a objetos situados en otros equipos (objetos remotos).
java.beans	Librería que permite la creación y manejo de componentes *javabeans*.

Para encontrar una clase java necesita dos cosas:

- El nombre del paquete
- Las rutas de los paquetes y las clases (CLASSPATH). El CLASSPATH sirve para localizar clases que no son parte de la plataforma Java.

Claúsula package

Un package es una agrupación de clases afines. Equivale al concepto de librería existente en otros lenguajes o sistemas. Una clase puede definirse como perteneciente a un package y puede usar otras clases definidas en ese o en otros packages.

Los packages delimitan el espacio de nombres (space name). El nombre de una clase debe ser único dentro del package donde se define. Dos clases con el mismo nombre en dos packages distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.

Una clase se declara perteneciente a un package con la clausula package, cuya sintaxis es:

package nombre_package;

La clausula package debe ser la primera sentencia del archivo fuente. Cualquier clase declarada en ese archivo pertenece al package indicado.

Por ejemplo, un archivo que contenga las sentencias:

```
package miPackage;
. . .
class miClase {
. . .
```

declara que la clase miClase pertenece al package miPackage.

La claúsula package es opcional. Si no se utiliza, las clases declaradas en el archivo fuente no pertenecen a ningún package concreto, sino que pertenecen a un package por defecto sin nombre.

La agrupación de clases en packages es conveniente desde el punto de vista organizativo, para mantener bajo una ubicación común clases relacionadas que cooperan desde algún punto de vista.

Claúsula import

Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package. Por ejemplo:

```
package Geometria;
. . .
class Circulo {
   Punto centro;
   . . .
}
```

En esta declaración definimos la clase Circulo perteneciente al package Geometria. Esta clase usa la clase Punto. El compilador y la JVM asumen que Punto pertenece también al package Geometria, y tal como está hecha la definición, para que la clase Punto sea accesible (conocida) por el compilador, es necesario que esté definida en el mismo package.

Si esto no es así, es necesario hacer accesible el espacio de nombres donde está definida la clase Punto a nuestra nueva clase. Esto se hace con la clausula import. Supongamos que la clase Punto estuviera definida de esta forma:

```
package GeometriaBase;
class Punto {
   int x , y;
}
```

Entonces, para usar la clase Punto en nuestra clase Circulo deberiamos poner:

```
package GeometriaAmpliada;

import GeometriaBase.*;

class Circulo {
   Punto centro;
   . . .
}
```

Con la claúsula import GeometriaBase.*; se hacen accesibles todos los nombres (todas las clases) declaradas en el package GeometriaBase. Si sólo se quisiera tener accesible la clase Punto se podría declarar: import GeometriaBase.Punto;

La claúsula import simplemente indica al compilador donde debe buscar clases adicionales, cuando no pueda encontrarlas en el package actual y delimita los espacios de nombres y modificadores de acceso. Sin embargo, no tiene la implicación de 'importar' o copiar código fuente u objeto alguno. En una clase puede haber tantas sentencias import como sean necesarias. Las cláusulas import se colocan después de la cláusula package (si es que existe) y antes de las definiciones de las clases.

Nombres de los packages

Los packages se pueden nombrar usando nombres compuestos separados por puntos, de forma similar a como se componen las direcciones URL de Internet. Por ejemplo se puede tener un package de nombre misPackages.Geometria.Base. Cuando se utiliza esta estructura se habla de packages y subpackages. En el ejemplo misPackages es el Package base, Geometria es un subpackage de misPackages y Base es un subpackage de Geometria.

De esta forma se pueden tener los packages ordenados según una jerarquía equivalente a un sistema de archivos jerárquico.

El API de java está estructurado de esta forma, con un primer calificador (java o javax) que indica la base, un segundo calificador (awt, util, swing, etc.) que indica el grupo funcional de clases y opcionalmente subpackages en un tercer nivel, dependiendo de la amplitud del grupo. Cuando se crean packages de usuario no es recomendable usar nombres de packages que empiecen por java o javax.

Ubicación de packages en el sistema de archivos

Además del significado lógico descrito hasta ahora, los packages también tienen un significado físico que sirve para almacenar los módulos ejecutables (ficheros con extensión .class) en el sistema de archivos del ordenador.

Supongamos que definimos una clase de nombre miClase que pertenece a un package de nombre misPackages.Geometria.Base. Cuando la JVM vaya a cargar en memoria miClase buscará el módulo ejecutable (de nombre miClase.class) en un directorio en la ruta de acceso misPackages/Geometria/Base. Está ruta deberá existir y estar accesible a la JVM para que encuentre las clases.

Si una clase no pertenece a ningún package (no existe clausula package) se asume que pertenece a un package por defecto sin nombre, y la JVM buscará el archivo .class en el directorio actual.

Para que una clase pueda ser usada fuera del package donde se definió debe ser declarada con el modificador de acceso public, de la siguiente forma:

```
package GeometriaBase;

public class Punto {
   int x , y;
}
```

Si una clase no se declara public sólo puede ser usada por clases que pertenezcan al mismo package.

STRINGS

Son secuencias de caracteres, como "hola".

Es una clase (String) predefinida en el paquete java.lang para tratar secuencias de caracteres.

Para crear un objeto String no necesito constructor, directamente le asigno una secuencia de caracteres entre dobles comillas y el objeto queda creado.

String e = ""; //String vacío, distinto de String h; que vale null y da error al tratarlo.

String saludo= "Hola";

Además de los métodos que contiene la clase para trabajar con cadenas, el lenguaje java ofrece el operador "+" para concatenar

String p ="uno";
String q="dos";
String todo=p+q; --→"unodos"

Si concateno un String con un valor que no lo es, el último es convertido a String.

String pp = "PG" + 14; →"PG14"

Métodos de la clase String

- int length(): calcula la longitud de una cadena

  int n = saludo.length(); →4

- Comparación:

  boolean s.equals(t); → s y t pueden ser constantes o variables !! Nunca usar == !!

  int s.compareTo(String other); → <0 si s menor que other
  >0 si s mayor que other
  =0 si s es igual a other

- Substring: Obtiene un substring de otro. Coge del String el subString comprendido entre la primera posición pasada como parámetro y la última pasada como

parámetro menos 1. Si no se pone el segundo parámetro retorna hasta el final de la cadena

String saludo="hola";
String s=saludo.substring(0,3); -→ "hol" (Si te sales de rango, error en ejecución)

- char charAt(int pos) : retorna el carácter que se encuentra en esa posición en la cadena, comenzando desde 0.

  char saludo.charAt(2); →'l'

- Reemplazar: Crea un nuevo String en el que reemplaza el carácter viejo por el nuevo.

  String replace(char oldChar, char newChar)

- Mayusculas y minúsculas:

  String toLowerCase(); minúsculas

  String toUpperCase(); mayúsculas

- String trim() elimina los blancos de una cadena al principio y final de la misma

- int indexOf(String s, int desde);

  desde es el índice desde donde se comienza la búsqueda.

Retorna la posición donde ha encontrado el String s, -1 en caso de error.

- boolean startsWith(String s) : retorna true si la cadena comienza por la cadena s

  String cadena=”Hola”;
  cadena.startsWith(“Ho”);

- boolean endsWith(String s): análoga a la anterior pero para el final de la cadena

FECHAS EN JAVA

El trabajo con fechas en java, en versiones anteriores al jdk 8, era un poco confuso. En esta nueva versión se ha implementado una API que hace este trabajo más fácil y cómodo para el programador. A los paquetes java.util.Date y java.util.Calendar, le han añadido el paquete java.time que incorpora nuevas utilidades. Es posible que nos encontremos código implementado en una versión anterior, quizás utilice las siguientes clases: Calendar, GregorianCalendar, Date de los paquetes mencionados antes. Si disponemos de una versión 8, no deberíamos utilizarlas.

Fechas con java.util (No las usamos)

CLASE GREGORIANCALENDAR

Constructores

- GregorianCalendar(): Obtiene la fecha del día.
- GregorianCalendar(int year, int month, int day): Crea una fecha con los datos pasados como parámetros.
- GregorianCalendar(int year, int month, int day,int hour, int minute, int second)

Si construyo una fecha errónea no se produce excepción, no se verifica que lo es.

Métodos

- void add(int field, int amount): Añade a una fecha un número de segundos, minutos, horas, dias, meses o años.

El campo field puede tener uno de estos valores: Calendar.DATE Calendar.MONTH Calendar.YEAR Calendar.HOUR Calendar.MINUTE Calendar.SECOND

La clase Calendar es abstracta y sus campos estáticos

- Int get(int field): Para obtener el valor de un campo de la fecha.

Por ejemplo, si tengo la fecha 17 de marzo de 2007 en el objeto fecha,

fecha.get(Calendar.DATE)…… 17
fecha.get(Calendar.MONTH) …….. 2 (los meses empiezan en 0)

• long getTimeInMillis(): retorna la fecha en milisegundos transcurridos desde el 1 de Enero de 1970.

- Date getTime(): Me devuelve la fecha de tipo Date asociada.
- void set(int field, int value): Pone el valor value en el campo correspondiente.
- void setTime(Date d): Modifica la fecha con los valores pasados en Date.
- boolean after(Object o): Devuelve true si una fecha es posterior a otra.
- boolean before(Object o): Devuelve true si una fecha es anterior a otra.

Ejemplo

GregorianCalendar calendario = new GregorianCalendar();
calendario.add(Calendar.DATE, -numero_dias);
Restamos ese número de días

CLASE DATE (Deprecated)

Constructores:

- Date() : Me crea un objeto fecha con la fecha del sistema.
- Date(int y, int m, int d);
- Date(long segundos) : Me crea un objeto fecha a partir de un número de segundos.

Métodos:

- int compareTo(Date another) : compara dos fechas, devuelve un valor menor de 0 si la primera es menor que la segunda, un valor mayor de 0 si la primera es mayor que la segunda, 0 si son iguales.
- long getTime(): Devuelve la fecha en milisegundos.
- void setTime(long time): Modifica la fecha con los segundos.

CLASE SIMPLEFORMAT

Sirve para formatear fechas.
Constructor:

- SimpleDateFormat(String formato)

Métodos:

- String format(Date date): Para obtener la fecha del sistema en un formato concreto se puede jugar con las clases Date y SimpleDateFormat. Por un lado podemos sacar la fecha actual del sistema de la siguiente forma:

  Date fechaActual = new Date();

  Ojo, estamos trabajando con el Date del paquete java.util no con el de java.sql

Una vez que lo tenemos se especifica el formato en que queremos la cadena
SimpleDateFormat formato = new SimpleDateFormat("yyyyMMdd");
y finalmente se obtiene la cadena

String cadenaFecha = formato.format(fechaActual);

Para el formato podemos jugar con las siguientes posibilidades
dd.MM.yy 09.04.98
yyyy.MM.dd G 'at' hh:mm:ss z 1998.04.09 AD at 06:15:55 PDT
EEE, MMM d, ''yy Thu, Apr 9, '98
h:mm a 6:15 PM
H:mm 18:15
H:mm:ss:SSS 18:15:55:624
K:mm a,z 6:15 PM,PDT
yyyy.MMMMM.dd GGG hh:mm aaa

Fechas con java.time

CLASES LOCALDATE , LOCALTIME, LOCALDATETIME

La primera de ellas representa una fecha en formato yyyy-mm-dd, la segunda una hora en formato hh:mm:ss:ns y la tercera ambas cosas a la vez. Esta última es útil , por ejemplo, en el caso de los archivos log.

- Para obtener la fecha y hora actual (la que tenga el sistema), se declara el objeto y se llama a su método now(). **Es un método estático, por lo que no instanciamos un objeto para llamarlo**

```
import java.time.*;
public class FechaHoraActual {
        public static void main(String[] args) {
                LocalDate fechaActual = LocalDate.now();
                System.out.println(fechaActual);
                LocalTime horaActual = LocalTime.now();
                System.out.println(horaActual);
                LocalDateTime ahora = LocalDateTime.now();
                System.out.println(ahora); }
 }
```

- Para construir una fecha u hora concretas

```
LocalDate fechaNacimiento=LocalDate.of(1999, 3, 28);
System.out.println(fechaNacimiento);
LocalTime horaRecreo=LocalTime.of(11, 25);
System.out.println("Salimos al recreo a las " + horaRecreo);
LocalDateTime heNacido=LocalDateTime.of(1999, 3, 28,20, 40);
System.out.println("Momento exacto de mi nacimiento: "+ heNacido);
```

Si construyo fecha errónea se produce una excepción DateTimeException.

- Para añadir a una fecha días, semanas, meses o años: plusDays, plusWeeks, plusMonths, plusYears

```
fechaNacimiento=fechaNacimiento.plusDays(5); // Añade 5 días a la fecha dada
```

- Para quitar a una fecha días, semanas, meses o años: minusDays, minusWeeks, minusMonths, minusYears

```
fechaNacimento=fechaNacimiento.minusYears(3); // Quita 3 años a la fecha dada
```

- Comparamos fechas con isAfter, isBefore, equals:

```
LocalDate miFechaNacimiento, tuFechaNacimiento;

…

if (tufechaNacimiento.isBefore(miFechaNacimiento))

    System.out.println("Eres mayor que yo");
```

- Diferencia entre dos fechas:

  Usando ChronoUnit:

```
LocalDateTime fecha1,fecha2;
…
long difftotaldias= ChronoUnit.DAYS.between(fecha1,fecha2);
```

  Usando Period: El resultado viene agrupado en días, meses y años.

```
Period period = Period.between(fecha1,fecha2);
int diffdias = period.getDays();
int diffmeses = period.getMonths();
int diffaños = period.getYears();
```

- Diferencia entre dos LocalDateTime o LocalTime:

  Usando ChronoUnit:

```
LocalDateTime hora1,hora2;
…
long diff = ChronoUnit.HOURS.between(hora1,hora2);
```

  Usando Duration:

```
Duration duration = Duration.between(hora1,hora2);
long diff = Math.abs(duration.toHours());
```

- El paquete tiene definidos enumerados para los días de la semana y los meses

Existe un enum donde se definen todos los días de la semana java.time.DayOfWeek
DayOfWeek lunes = DayOfWeek.MONDAY;
Este enum tiene algunos métodos interesantes que permite manipular días hacía adelante y hacia atrás:
DayOfWeek lunes=DayOfWeek.MONDAY;
DayOfWeek otro=lunes.plus(5);

```
System.out.println(otro);
System.out.println(otro.minus(3));
```

Con el método getDisplayName() se puede acceder al texto que corresponde a la fecha, dependiendo del Locale actual

```
DayOfWeek lunes=DayOfWeek.MONDAY;
System.out.println(lunes.getDisplayName(TextStyle.FULL, Locale.getDefault()));
LocalDate fechaNacimiento=LocalDate.of(1999, Month.MARCH, 28);
System.out.println(fechaNacimiento.getDayOfWeek());
```

Para los meses, existe el enum java.time.Month que básicamente hace lo mismo

```
LocalDate fechaNacimiento=LocalDate.of(1999, Month.MARCH, 28);
System.out.println(fechaNacimiento);
Month mes=Month.FEBRUARY;
System.out.println("Dos meses más y será: ,"+ mes.plus(2));
System.out.println("Hace 1 mes fué: "+ mes.minus(1));
```

- Dispone de una clase YearMonth que representa el mes de un año específico.

```
YearMonth mes=YearMonth.now();
YearMonth febNoBisiesto=YearMonth.of(2015, Month.FEBRUARY);
YearMonth febBisiesto=YearMonth.of(2016,Month.FEBRUARY);
System.out.println(mes.getMonth());
System.out.println("Días febrero 2015: "+ febNoBisiesto.lengthOfMonth());
System.out.println("Días febrero 2016: "+ febBisiesto.lengthOfMonth());
```

- Disponemos de las clases MonthDay , Year que representan un día del mes en particular y un año específico

```
MonthDay dia=MonthDay.of(Month.FEBRUARY, 29);
 if (dia.isValidYear(2016))
        System.out.println("El día "+ dia.getDayOfMonth()+ " es válido para el año 2015");
else
        System.out.println("El día "+ dia.getDayOfMonth()+ " no es válido para el año 2015");
Year anyo=Year.now();
Year otroAnyo=Year.of(2015);
esBisiesto(anyo);
esBisiesto(otroAnyo);

public static void esBisiesto(Year y){
        if (y.isLeap())
                System.out.println(y.getValue()+" es Bisiesto");
        else System.out.println(y.getValue()+ " No Bisiesto"); }
```

- Disponemos de métodos para obtener hora, minutos , segundos,dia, mes,....

```
LocalTime justoAhora = LocalTime.now();
System.out.println("En este momento son las " + justoAhora.getHour()+ " horas " +
justoAhora.getMinute()+ " minutos y "+ justoAhora.getSecond()+" segundos");
LocalDate hoy=LocalDate.now();
// Día del mes
 int dia=hoy.getDayOfMonth();
// Día de la semana en número
int dia=hoy.getDayOfWeek().getValue());  -→ej. 4
// Día de la semana en String, en el idioma en el que estés ejecutando
String diasem= hoy.getDayOfWeek().getDisplayName(TextStyle.FULL,Locale.getDefault())); -→ ej. jueves
```

```
// Mes en número
int mes=hoy.getMonthValue();
// Año
int anyo=hoy.getYear();
```

Dar formato a las fechas

Según la aplicación que estemos desarrollando, querremos mostrar las fechas en un formato u otro: el mes por su número en el calendario o por su nombre, el nombre completo o abreviado,....Para resolver este problema formateamos las fechas. Para ello debemos utilizar el paquete **java.time.format**

Hay parámetros que nos permiten obtener los valores mencionados en el párrafo anterior. Veamos cuales son los más utilizados:

- **y**, nos permite acceder al año en formato de cuatro o dos dígitos (2014 o 14).
- **D**, nos permite obtener el número de día del año (225).
- **d,** nos devuelve el número del día del mes (27).
- **L**, el mes del año en forma numérica
- **M** nos da el mes en texto.
- **H**, nos da la hora.
- **s**, nos da los segundos.
- **m**, nos permite obtener los minutos.
- **a**, nos da el am o pm de la hora.
- **z,** nos permite acceder al nombre de la zona horaria.

El formato por defecto de una fecha es: yyyy-LL-dd

La Clase DateTimeFormatter Esta clase dispone del método ofPattern() que recibe los parámetros indicados antes y establece el patrón o formato que tendrá la fecha a la cual le apliquemos el método format de dicha clase. El método format retorna un String.
En estos ejemplos a partir de una fecha obtenemos un String con el formato deseado:

```
 LocalDate hoy=LocalDate.now();

DateTimeFormatter formato1 = DateTimeFormatter.ofPattern(“dd/LL/yy");
System.out.println(formato1.format(hoy));

formato1 = DateTimeFormatter.ofPattern("yyyy/MMMM/dd");
System.out.println(formato1.format(hoy));

formato1 = DateTimeFormatter.ofPattern("dd/MMMM/yy"); System.out.println(formato1.format(hoy));
```

Podemos hacer lo contrario, a partir de un String que representa una fecha obtenemos un objeto LocalDate

```
String fechaTexto="2015-12-31";
LocalDate fecha=LocalDate.parse(fechaTexto); // El formato por defecto que aplica es yyyy-LL-dd
System.out.println(fecha);
```

Si queremos usar un formato distinto al formato por defecto tenemos que utilizar un patrón que le indica al método parse cómo es la cadena. Hay que controlar la Excepción `DateTimeParseException` que se producirá si la cadena no tiene el formato adecuado:

```
DateTimeFormatter patron=DateTimeFormatter.ofPattern("yyyy/LL/dd");
System.out.println(LocalDate.parse("2014/11/15",patron));
```

En cualquier caso la fecha que obtenemos es un objeto de tipo LocalDate.

Clase StringTokenizer

La clase *StringTokenizer* nos ayuda a dividir un string en substrings o tokens, en base a otro string (normalmente un carácter) separador entre ellos denominado delimitador.

Supongamos un string consistente en el nombre, y los dos apellidos de una persona separados por espacios en blanco. La clase *StringTokenizer* nos ayuda a romper dicho string en tres substrings basado en que el carácter delimitador es un espacio en blanco.

Supongamos un fichero en el que guardamos una determinada información linea a linea La clase *StringTokenizer* nos permite dividir el string obtenido en un número de substrings o tokens igual al número de líneas de texto, basado en que el carácter delimitador es **'\n**'.

Para usar la clase *StringTokenizer* tenemos que importar el paquete útil o bien

```
import java.util.StringTokenizer;
```

Los constructores

Creamos un objeto de la clase *StringTokenizer* llamando a uno de los constructores que tiene la clase. Al primer constructor, se le pasa el string *nombre* que va a ser dividido teniendo en cuenta que el espacio en blanco es el delimitador por defecto.

```
String nombre="Angel Franco García";
StringTokenizer tokens=new StringTokenizer(nombre);
```

Creamos un objeto *tokens* de la clase *StringTokenizer*, pasándole el string *strDatos* y el delimitador **"\n**"

```
String strDatos="6.3\n6.2\n6.4\n6.2";
StringTokenizer tokens=new StringTokenizer(strDatos, "\n");
```

Puedo utilizar varios caracteres separadores

```
StringTokenizer tokens=new StringTokenizer(strDatos, "\n.");
```

Obtención de los tokens

La clase *StringTokenizer* implementa el interface *Enumeration*, por tanto define las funciones *nextElement* y *hasMoreElements*.

```
public class StringTokenizer implements Enumeration {
            //...
 public boolean hasMoreElements() {
//...
 }
 public Object nextElement() {
//...
 }
}
```

Para el programador es más cómodo usar las funciones miembro equivalentes *nextToken* y *hasMoreTokens*. Para extraer el nombre, el primer apellido y el segundo apellido en el primer ejemplo, escribiremos

```
String nombre="Angel Franco García";
StringTokenizer tokens=new StringTokenizer(nombre);
while(tokens.hasMoreTokens()){
    System.out.println(tokens.nextToken());

 }
```

El siguiente ejemplo, requiere un poco más de trabajo, ya que además de extraer los tokens del string *strDatos*, hemos de convertir cada uno de los substrings en un valor numérico de tipo **double** y guardarlos en el array *datos* del mismo tipo.

```
String str=tokens.nextToken();
datos[i]=Double.valueOf(str).doubleValue();
```

El número de tokens o de datos *nDatos* que hay en un string *strDatos*, se obtiene mediante la función miembro *countTokens*. Con este dato establecemos la dimensión del array *datos*.

```
int nDatos=tokens.countTokens();
double[] datos=new double[nDatos];
```

El código completo para extraer los tokens del string *strDatos* y guardarlos en un array *datos*, es el siguiente.

```
String strDatos="6.3\n6.2\n6.4\n6.2";
StringTokenizer tokens=new StringTokenizer(strDatos, "\n");
 int nDatos=tokens.countTokens();
```

```
 double[] datos=new double[nDatos];
int i=0;
while(tokens.hasMoreTokens()){
        String str=tokens.nextToken();
        datos[i]=Double.valueOf(str).doubleValue();
        System.out.println(datos[i]);
         i++}
```

Generar números aleatorios

Podemos generar números aleatorios con la clase Random del paquete util

```
Random r = new Random();
int valorAleatorio = r.nextInt(6)+1;  // Entre 0 y 5, más 1.
```

O con el método random de la clase Math:

```
double num=Math.random(); //genera un número entre 0 y 1, pero nunca el 1.
int valorAleatorio=(int)(Math.random()*6)+1; // Entre 0 y 6, más 1.
```