APUNTES PROGRAMACIÓN ORIENTADA A OBJETOS

Fernando Moreno Jabato

Apuntes de Programación Orientada a Objetos (Java)
Actualizado: 25/2/14

Los siguientes apuntes están centrados en desarrollar el temario de la asignatura Programación Orientada a Objetos del Grado en Ingeniería de la Salud en la Universidad de Málaga. En ellos trataremos algunos conceptos básicos de los lenguajes orientados a objetos y algunas clases principales del lenguaje Java.
La parte final del temario no está desarrollada en estos apuntes.
Tenemos que saber que los archivos de java tienen la extensión **.java** o **.class** si son clases. La orden para compilarlos en el **cmd** (consola de Windows) es **javac <nombre>.java**
Es importante saber que en este lenguaje las separaciones admitidas **<,> <;> <:> <{}>**
/* */ --> Comentarios de varias líneas
// --> Comentarios

1.- OPERADORES

+	Suma o concatenación de cadenas
<=	Bool menor igual
>=	Bool mayor igual
++	Incrementa
--	Resta le
+=	Suma le
-=	Réstale
***=**	Multiplícale
/=	Divídelo entre
%	Devuelve el resto de la división
!=	Bool, si los términos son diferentes devuelve true
<boolexpresion> ? <valor1> : <valor2>	Si la expresión devuelve "true" coge valor1 si es "false" coge valor2

&& "Y" o "and", si se cumplen ambas condiciones → true.
& y" o "and" igual que antes pero comparación a nivel de bits.
|| "o" o "or", si se cumple una de las condiciones → true
| "o" o "or", igual que antes pero comparación a nivel de bits.

2.- LISTA DE CARACTERES

\t --> Tabulación
\r --> Retorno de carro
\n --> Línea nueva

3.- VARIABLES Y WRAPPERS

Se declaran poniendo un "tipo" seguido del nombre de la variable del valor si así se desea.
Los diferentes tipos son:

boolean	**char**	**int**
double	**float**	**byte**
short	**long**	

Para crear una variable constante es suficiente con añadir **final** antes del nombre. Este "final" hace que las clases que hereden de la original no puedan alterar este procedimiento o variable.
Cuando queramos crear una variable de clase, no de estancia, usamos la etiqueta **static**. Con esto conseguimos que esa variable sea común a cualquier objeto de esa clase, es decir, no cada objeto tiene la suya propia sino que todos apuntan, para esa variable, al mismo espacio de memoria.

Hay momentos en los que quisiéramos trabajar con objetos en vez de variables básicas, para ellos existen los **wrappers** que no son más que objetos que se comportan como las variables, pero son objetos. Todos los tipos de variables tienen su wrapper asociado:

Variable	Wrapper
boolean	**Boolean**
char	**Character**
int	**Integer**
Doublé	**Double**
float	**Float**
short	**Short**
long	**Long**
byte	**Byte**

Los wrappers al ser objetos también tienen métodos

4.- CLASES

Las clases son la base de cualquier objeto/instancia. Son “plantillas” en las que se especifica el comportamiento de cualquier objeto que sea de esa clase.
Han de guardarse en ficheros que tengan exactamente el mismo nombre identificador que se le ponga a la clase (cuidado, java diferencia entre mayúsculas y minúsculas). A estos ficheros les pondremos la extensión ".java"
Para declararlos hay que usar la palabra reservada **class** seguido de un identificador (nombre) y unas llaves que engloben al "comportamiento" (conjunto de métodos) de la clase.

class <nombre> {
<comportamiento>;
}

En el comportamiento de una clase declaramos tanto los **atributos**, que no son más que las variables internas de cada objeto de una clase, y los **métodos**, que son subrutinas.

Los atributos siguen la sintaxis típica de las variables simples.

Los métodos siguen una sintaxis parecida a ésta:

<tipoDevuelve> <identificador> (<parámetros>) { <sentencias>; }

Es importante saber que se pueden definir métodos con igual nombre y que devuelvan diferentes cosas, lo importante es que si tienen el mismo identificador pero son métodos "diferentes" han de tener parámetros de entrada diferentes.

Hay un tipo de métodos que se llaman "constructores" los cuales no llevan ningún tipo pues no devuelven nada (aun así NO ES UN VOID). Su función es realizar las acciones que se les encomienda nada más se crea el objeto. Es importante que el identificador del constructor sea el mismo que el de la clase.

Además se puede usar el identificador **this** para referirnos al objeto que estamos usando en ese momento. **This()** hace referencia a un constructor de una clase del mismo tipo (leer apartado para uso de **super()** en "herencia de clases") mientras que **this.** hace referencia al mismo objeto. Ejemplo:

```
class MiPunto{
    this(2,3);      // This" equivale a "MiPunto" en éste caso.
}
```

Si tenemos una clase heredera y otra clase madre, y enviamos un mensaje a un objeto de la clase heredera pero sobre un método heredado, el método heredado se ejecutará de la clase madre pues es ahí donde está definido, pero si este método incluye un *this*, éste hará referencia al objeto heredero que es el que ha recibido el mensaje, aunque el *this* está definido en la clase madre. Igual ocurre con el *super*

Java admite constructores por defecto (no declarados) en el que, al ejecutarlo, las variables de estancia acogen los valores **0** si son numéricas, **false** si son booleanas, **""** (caracter vacío) si son char, o **null** si es una referencia a un objeto.

5.- NEW

La orden **new** genera un nuevo "objeto" del tipo que se especifique y devuelve una referencia a éste (crea un puntero). Un ejemplo sería (damos por hecho que hay una clase "Punto").

Punto p1 = new Punto(2,3); //genera una referencia a un punto con los valores (2,3) y esta referencia se almacena en p1

6.- OBJETOS

Los objetos son los que reciben y responden a los mensajes del programa en base a los comportamientos preestablecido en la declaración de las clases.

Para crear un nuevo objeto hay que utilizar la siguiente sintaxis:

<nombreClase> <nombreObjeto> = new <constructorClase>;

Como hemos dicho, los objetos son los que reciben los mensajes y responden a estos devolviendo valores, cambiando parámetros....realizando la acción que corresponda. Si queremos enviar un mensaje a un objeto seguimos la siguiente sintaxis:

<nombreObjeto>.<método > (<parámetros>);

Obviamente, si el objeto nos va a devolver algo habrá que recogerlo en alguna variable:

int i=p.valor(); // si el método "valor()" devuelve un valor tipo integrer

Si no devuelve nada, digamos que cambia algún atributo del objeto, entonces no haría falta recogerlo en ningún lugar y se escribiría tal y como está escrito en la sintaxis para enviar mensajes a objetos.

Hay una forma de saber si un objeto es de una clase o no con una función que devuelve un boolean que sigue la siguiente nomenclatura:

boolean **<objeto> instanceof <clase>**

*Que d*evuelve true si el objeto es de la clase *clase* o false si no lo es

7.- MODIFICADORES DE ACCESO

Dependiendo de qué fines y quienes queramos que puedan usar las clases o los comportamientos podemos distinguir 4 niveles diferentes de seguridad:

public	Es "público", cualquiera puede usarlo
private	Sólo se puede acceder desde métodos, o invocando desde métodos, de la misma clase.
protected	Permite acceso "público" para las clases derivadas (herencia) pero da acceso privado al resto de clases.
sin clase*	Accesible desde cualquier clase que pertenezca al mismo paquete que la designada de esta manera.

Hasta el momento no habíamos comentado esta parte para no añadir dificultad al comenzar en el entendimiento de los lenguajes orientados a objetos, pero como podemos ver, podemos asignar niveles de acceso a métodos y variables de clase y la posición en la que debemos escribirla es al principio de la sentencia. Ejemplo de método público y variable de clase privada:

public int método() {return 1;}
private boolean variable;

8.- HERENCIA EN CLASES

Las clases pueden heredar cualidades de otras clases principales (respecto de la heredera). En los mapas de clases se representan las clases herederas con unas flechas que apuntan a la clase principal.

Cuando una clase hereda de otra, los métodos que se pueden usar con la clase principal también pueden usarse con la clase heredera.

Ocurre que las cosas con acceso "private" no pueden ser usadas por clases herederas, sin embargo si que se podrían usar si tuvieran el modificador de acceso del tipo "protected" en vez de "private".

Las subclases pueden modificar el comportamiento heredado, es decir, pueden redefinir algún método heredado.

Cuando creamos una clase que hereda de otra sigue la siguiente sintaxis:

<modAcceso> class <nombrHeredera> extends <nombrPrincipal> { <comportamiento> }

Al crearse una herencia aparece un nuevo método que es **super()** que lo que hace es realizar una invocación al constructor de la clase principal correspondiente. Esto actúa como **this** que hace referencia a un constructor de la misma clase y **super** hace referencia a un constructor de una clase de la que hereda la actual.

Cuando en un método de una clase llamamos a otro método que corresponde a la clase *madre* no hace falta llamar a un objeto tipo *super* usando una sintaxis parecida a esta: *super.<metodo>();* . Al realizar una llamada a un método primero se busca si dicho método está recogido en la lista de métodos aceptados por el objeto, al mirar en esta lista no se mira si es un método heredado o no, simplemente si está presente. Una vez se comprueba que el método es válido se lanza al objeto y éste lo ejecuta de la forma más adecuada, en el caso de que sea un método heredado, buscará algún objeto *super* asociado (que seguramente habrá sido generado en el constructor) que

podrá responder al método, si no lo encuentra es muy probable que devuelva *null*, *caracter vacío*, etc, o lance un error (objeto *exception*)

Importante es saber que una **clase sólo puede tener una clase de la que es heredera pero las interfaces si pueden heredar de varias interfaces**. También es importante saber que podemos almacenar instancias de una clase *madre* en un objeto de tipo *heredero*, pero no viceversa.

```
//B hereda de A
B nuevoObj=new A();  //Bien
A nuevObj=new B();  //Mal
```

9.- ARRAYS

Un array es una estructura de datos que contiene varios datos de un mismo tipo. Se declaran usando la siguiente nomenclatura:

<tipo> <nombre> []...[]= new <tipo>[dim1]…[dimN];

Notar que los **[]** a la izquierda del igual están vacíos, esto es porque las dimensiones correspondientes se declaran en su correspondiente **[]** a la derecha. Ejemplo de una matriz de enteros 3x3: *int matrix[][]=new int[3][3];*

Java tiene la cualidad de que podemos no definir el tamaño de alguna de las dimensiones, es decir, dejamos una dimensión de tamaño variable.

boolean arrayDeBool [][] = new boolean[3][];

Con este código hemos generado un array de booleanos que tiene 3 filas y cada una de estas filas tiene un número de columnas que dependerá de los elementos que introduzcamos en ellas.

10.- FOR

El **for** es una estructura de control que presenta la siguiente nomenclatura:

for (<declaracionVariables>; <condiciones>; <incrementos/decrementos) { <sentencias>;}

En la de claración de variables podemos declarar más de una, al igual que en las condiciones, un ejemplo de esto sería:

for (int i=1, int j= i+10; i<=5; ++i, --j) { }

Hay un tipo de for iteránte de mayor nivel en el que hay que aportar menos información, éste equivaldría a un **foreach** pero sigue la siguiente estructura:

for(<tipoVar> <nombreVar> : <array>) {<sentencia/as>;)

Esto hace un **foreach** que recorre el array y para cada variable del tipo de la variable *nombreVar* que declaremos, ejecutará las sentencias. Con esto no necesitamos conocer dónde está el fin del array y sabremos que hemos recorrido todos los huecos de memoria que nos interesan.

11.- IF-ELSE y ELSE-IF

La estructura de control **if- else** es un tipo de estructura en la que si la condición es cierta (true) se realiza el bloque de "if" y si es "false" se realiza el bloque de "else". Aunque también se puede enganchar el "ELSE" con otro "IF" con la estructura "ELSE IF". Observémoslo:

```
if (<condiciones1>)
   { <sentencias>; }
else if (<condiciones2>)
   { <sentencias>; }
else
   { <sentencias>; }
```

Si las "sentencias" son sólo una sentencia, se pueden obviar las llaves.

12.- SWITCH

En switch sólo podemos usar variables que sean sucesivas, es decir, que sus valores tenga un anterior y un posterior, los tipos de variables que cumplen esto son: "INT", "CHAR" "ENUM".

El switch tiene una sintaxis que responde a esta forma:

```
switch (<variable>)
 {
  case <valor1>: <sentencia(as);>;
  break;
  case <valor2>: <sentencia(as);>;
  break;
  ...
  default: <sentencia(as);>;
 }
```

Este tipo de función es muy interesante. En el apartado "variable" declaramos la variable que vamos a estudiar y en los apartados de "valor" declaramos el valor para el que hay que ejecutar las correspondientes sentencias.

NOTA IMPROTANTE: el "break" hace que se salga del switch al realizar las sentencias, si no los ponemos se realizarán todas las sentencias que se tengan que hacer, es decir, si hay casos posteriores que también se cumplen también se realizarán, si se pone el break se saldría del switch.

TRUCO: crear un if extraño. Imagina que quieres que cuando la variable "int i" tenga los valores (1,8 y 9) se haga una cosa, pues puedes escribir esto:

```
Switch (i){
    Case 1:
    Case 8:
    Case 9:
        <sentencia>;
        Break;
```

Como variable para el switch hay que utilizar variables de tipos numerables, pero desde la versión Java 1.7 se permite introducir strings como variables.

13.- WHILE

While crea un bucle que se sigue realizando mientras se cumpla una condición (condición==true). Su sintaxis responde a la siguiente forma:

```
while(<condición>)
 {<sentencia(as)>;}
```

También está la estructura de control **do...while** donde la condición se analiza después de haber ejecutado, al menos una vez.

14.- DATOS ENUMERADOS [java.lang]

Podemos crear tipos de variables que tengan valores restringidos, es decir, que tienen una lista de valores posibles. Para crearlos los declaramos en el archivo de una clase pero sin introducirlo en ésta y seguimos la siguiente nomenclatura:

```
enum <nombreTipoEnum> {<val1> , <val2> , ... , <valN>};
```

Cuando creemos una variable de un tipo **enum** lo podemos hacer de dos maneras:

```
<nombrTipoEnum> <nombreVar> = <nombrTipoEnum>.<valor>;

<nombrTipoEnum> <nombreVar> = <nombrTipoEnum>.valueOf("<valor>");
```

Los objetos tipo enum tienen métodos que nos pueden ser útiles como:

enum<T> [] **values()** --> Devuelve un array donde cada posición es uno de los valores del enum.

15.- SENTENCIAS DE SALTO

BREAK: Orden que al ejecutarla provoca que se pase a la ejecución del siguiente bloque de sentencias. Si está en un bucle, sale de éste y si está en un "if" también se sale de éste.

CONTINUE: Orden que sólo se puede usar en bucles. Cuando se ejecuta, el programa obvia las sentencias que falten hasta el final del bloque y vuelve a evaluar las condiciones por lo que el bucle continuará o no dependiendo de si se siguen cumpliendo las condiciones o no.

16.- STRING [java.lang]

Los string son cadenas de caracteres. Para crearlos hay dos maneras, creando un objeto **String** e igualándolo a una cadena literal:

String <nombre> = "<cadena>"

Otra forma es creándolo como los demás objetos:

String <nombre> = new String(<cadena>)

Los string que creemos seran objetos de la clase String así que podemos mandarles mensajes con los siguientes métodos:

int **length()**	Devuelve la longitud del string
char **charAt(int)**	Devuelve el caracter que hay en dicha posición
string **concat(String)**	Concatena ambos strings
boolean **contents(CharSequence)**	True si el string contiene la secuencia introducida
boolean **endsWith(String)**	Testea si el string termina o no con el string introducido
boolean **equalsIgnoreCase(String)**	Testea si ambos strings son iguales ignorando mayus y minus
int **hashcode()**	Devuelve un hascode para ese string
boolean **isEmpty()**	Devuelve true si length()=0
string **replace(char , char)**	Sustituye el primer char por el otro en toda la cadena
string **replaceAll(String , String)**	Sustituye todas las subcadenas que sean como el primer string, por el segundo string
string **replaceFirst(String, String)**	Igual pero sólo con la primera subcadena coincidente que encuentre
boolean **startsWith(String)**	Testea si empieza o no con el string dado
boolean **startsWith(String, int)**	(...) empezando a contar a partir de la posición que indica el int
string **substring(int)**	Devuelve una subcadena empezando desde la posición que indique el int
string **substring(int, int)**	Devuelve una subcadena que empieza en la pos del primer int y acaba en la pos del segundo
char[] **toCharArray()**	Transforma el string en un array de char
string **toLowerCase()**	Pasa todo el string a minúsculas
string **toUpperCase()**	(...) a mayúsculas
string **trim()**	Devuelve una copia de la cadena omitiendo el espacio en blanco inicial y final

17.- CLASES GENÉRICAS

Desde Java1.5 podemos usar clases genéricas al implementar interfaces y clases. ¿Qué son las clases genéricas? Son clases que representan a *cualquier* clase, es decir, al implementar el código es como si no diéramos el nombre exacto de a qué clase se refiere, esto hace que funcionen con cualquier objeto que les pasemos sean de la clase que sean pues el método no espera una clase en concreto.

Para usarlas tenemos que añadir en el encabezado de la clase o interfaz algo que siga esta sintaxis:

> ***< nombreClassGen >*** *IMPORTANTE: he roto la sintaxis usada hasta ahora, los símbolos "<" y ">" hay que escribirlos no como he venido haciendo hasta ahora*

Luego usaremos el nombre que hemos puesto entre < > de forma normal como si fuera el nombre de una clase sólo que el ordenador sabrá que esa "clase" puede ser cualquier clase. Ejemplo:

```
class Prueba <T> {
    public T objetoGenerico;
    public Prueba(T obj){
        objetoGenerico=obj;
    }
    public T getObject(){
        return objetoGenerico;
    }
}
```

18.- INTERFACES

Las interfaces son estructuras que sirven como guía para un concepto (clase) y que debe hacer, pero sin llegar a implementar o dar solución a los métodos.

¿Qué significa esto? Simple, una interfaz es una estructura que tiene un nombre y un conjunto de variables y métodos, al igual que una clase. Obviamente no se comporta como las clases, la diferencia radica en que **los métodos no están implementados**, es decir, son abstractos (todos) y **las variables de una interfaz nunca se comportan como variables de instancia sino como variables tipo final (v.clase)**.

La forma de crear una interfaz es usando la palabra reservada **interface** y la siguiente sintaxis:

> **interface <nombreInterfaz> {**
>
> **<varaiables>;**
>
> *Las variables las escribimos igual que para las clases, no tienen nada especial*
>
> **<métodos>;**
>
> *Los métodos los escribimos igual que si fueran abstractos pero sin usar la palabra reservada abstrac. Ejemplo: boolean compareTo();*
>
> **}**

Y ¿para qué sirve todo esto? Pues para que haya una estructura de algunos métodos y comportamientos que van a realizar muchas clases pero que cada una va a realizar de forma diferente, por ello será en las clases donde se implementen los métodos de las interfaces, para ello hay que usar la palabra reservada **implements** en las clases y después implementar el método de forma normal en la clase de la siguiente manera:

```
class <nombreClase> implements <interfaz> {
    <metodoInterfaz> { <implementacion>;}
}
```

19.- EXCEPCIONES

Cuando se produce un error en un método se crea un objeto del tipo **Exception** o herederos y el sistema interrumpirá la normal ejecución del programa y buscará una solución, es decir, algo que pueda continuar con la normal ejecución.

Todos los objetos del tipo *exception* son de la clase **Throwable** o herederos. Estos objetos incluyen listas de métodos relacionados con los tipos de errores. Esta clase está definida en el paquete **java.lang**

Es importante saber que las Exception pueden ser de tipo **checked** o **not checked**, las primeras son de obligado tratamiento y las otras no. Todas las excepciones que creemos serán de tipo *checked* siempre que no sean herederas de **RuntimeException** o herederas de ésta.

Cuando queremos controlar las excepciones debemos utilizar, al menos, tres tipos de bloques que recogen nuestro código:

Bloque: "try"

El bloque *try* engloba al código que puede producir alguna excepción. Su sintaxis es simple y presenta un aspecto como este:

```
try {
    <sentencia/as>;
}
```

Cada bloque *try* tiene asociado uno o más bloques *catch* o *trhow* que incluyen los pasos a seguir en caso de que se de una excepción y de que tipo sea.

Este bloque define el ámbito de manejadores de excepciones asociados.

Bloque: "catch"

El bloque *catch* va asociado al bloque *try* y lo complementa añadiendo manejadores de excepción a dicho bloque. Su sintaxis es la siguiente:

```
catch (<objetoTipoThrowable> <nombreVariable>) {
    <sentencia/as>;
}
```

El objeto tipo Throwable corresponde al tipo de error que lanza el bloque *try* y el "nombreVariable" es el nombre con el que nos podemos referir al objeto *exception* que ha sido lanzado (útil para obtener información de él; se explicará algo más de los objeto *exception* más adelante).

Los bloques *catch* constituyen, como ya hemos dicho, manejadores de excepciones, estos trabajan con objetos tipo Throwable o subclases de estas. Lo bueno que tienen los manejadores de excepciones es que sirven para cualquier error del tipo que se declare en el apartado *objetoTipoThrowable* y para cualquiera que herede de éste, es decir, si en dicho apartado ponemos un tipo de error que no tiene subclases diremos que tenemos un manejador específico, pues sólo actúa ante un tipo de errores, pero si declaramos un tipo de error que si tiene subclases habremos creado un manejador más general pues actuará si se lanza un error del tipo declarado o de alguna subclase de este tipo.

IMPORTANTE: nunca ha de haber código entre el bloque *try* y sus respectivos bloques *catch*:

```
try {
    ...
} catch (...) {
    ...
} catch (...) {
    ...
}
```

Bloque: "finally"

Este bloque engloba al conjunto de sentencias que se han de ejecutar pase o no algo en el las sentencias del bloque *try*. Este bloque puede ser necesario o ser *azúcar* pero su utilidad es clara, sirve para realizar limpieza del código, es decir, imaginemos que tenemos un programa que abre un fichero y realiza diferentes acciones con éste. Si englobamos estas acciones en un bloque *try* y definimos algunos manejadores con bloques *catch* tendremos cubierto problemas si no se ejecuta todo el código del bloque *try*. Pero, ¿Y si surge un error en la ejecución y no se cierra el fichero porque no se llega a dicha línea de código (imaginemos que está especificada al final del conjunto de acciones a realizar)? Para eso está el bloque *finally* en el que sería bueno que escribiéramos algunas líneas de código que cerraran el fichero si se ha lanzado algún error o que no hicieran nada si no ha pasado nada (ya que el fichero estaría cerrado).

En este caso también podríamos jugar con no cerrar el fichero en el bloque *try* y cerrarlo en el bloque *finally* ya que se ejecutará igualmente

Su sintaxis sigue un patrón parecido a éste:

```
finally {
    <sentencia/as>;
}
```

Finalmente un código que incluye un control de excepciones tendrá un aspecto parecido a éste:

```
try {
    <sentencia/as>;
} catch ( ... ) {
    <sentencia/as>;
}catch ( ... ){
    <sentencia/as>;
} finally {
    <sentencia/as>;
}
```

Al generar un objeto *exception* se incluyen en el mucha información, como puede ser el momento en el que se generó, información que se estaba utilizando en dicho momento, un mensaje de error, o la pila de métodos en la que se ha producido el error.

Esto último es muy interesante ya que esta pila de datos no es más que una lista ordenada de los métodos que estaban ejecutando la orden que ha producido el error. Mejor explicado, cuando nosotros escribimos una línea de código que *abreFichero* el ordenador al ejecutarla realiza algo parecido a este procedimiento: *ordenAbreFichero --> metodo1 --> metodo2 --> metodo3 --> seAbreFichero.* Esta lista de métodos que se han seguido o se estaban siguiendo en el momento en el que se produce el error es recogida en el objeto *exception.* Esto es útil ya que al capturar con un manejador de excepciones, éste será utilizado por el método de la pila de datos que pueda utilizarlo. Además surge una nueva opción que es declarar manejadores que actúen dependiendo de cuál sea el método y el tipo de excepción lanzada, esto se hace usando **throws**. La sintaxis es algo parecido a esto:

```
<nombreMetodo> throws <objetoTipoThrowable> {
    <sentencia/as>;
}
```

Algunas excepciones que es bueno conocer son:

JAVA.LANG

ClassCastException --> Problema al intentar comparar (cast-->casting) dos elementos que no pueden ser comparados debido a que son clases diferentes

NegativeArraySizeException --> Se lanza si se intenta crear un array con un parámetro longitud negativo

NoSuchMethodException --> No se encuentra el método que ha sido llamado en la clase

NullPointerexception --> Se ha intentado usar un objeto *null* cuando se necesita el uso de un objeto

NumberFormatException --> Problema al intentar pasar a String un tipo numérico y el tipo numérico no estaba en el formato adecuado

RuntimeException --> Excepción genérica, se lanza cuando no hay una excepción específica para el error

StringIndexOutOfBoundsException --> Lanzado cuando intentamos trabajar con una posición en un string que o es negativa o es mayor que la longitud del string

JAVA.UTIL

NoSuchElementException --> Lanzado cuando al volcar elementos de una enumeración (es decir, numéricos, no tienen por qué venir en un formato de varios números) no se vuelca nada porque no hay elementos numéricos

20.- CLASE SCANNER [java.util.Scanner]

Esta clase se utiliza, principalmente, para leer de teclado o para lectura desde cualquier flujo de entrada.

Podemos usarlo para leer del mismo modo que leíamos ficheros en C, es decir, leyendo tramos de contenido desglosando el flujo de entrada para coger lo que queramos.

Sus constructores siguen la siguiente sintaxis:

Scanner(InputStream)

Donde *InputStream* es el canal de entrada de datos, puede ser un string del programa (*String*), un fichero (*FileReader*) o un flujo de entrada como puede ser el teclado (*System.in*).

Los objetos de tipo Scanner tienen métodos como **.useDelimiter("[<delimitadores>]");** que reconoce como delimitadores a todos los caracteres que se escriban, sin espacios, en el apartado *<delimitadores>* (sin espacios porque el espacio cuenta como caracter y se incluiría como delimitador a tener en cuenta). Los delimitadores son los caracteres que marcan el final de una línea o frase. Si se añade un **+** después de los corchetes significaría que se puede tomar como delimitador la aparición de un caracter una o mas veces seguidas.

NOTA: el delimitador punto está siempre se añada o no en delimitadores

Otros métodos que nos pueden ser útiles son:

void **close()**	Cierra el objeto Scanner
string **findInLine(String)**	Busca el string introducido ignorando los delimitadores
boolean **hasNext()**	Devuelve true si el objeto Scanner todavía tiene información que volcar del input
boolean **hasNextLine()**	Devuelve true si el objeto Scanner todavía tiene una línea (marca el final un \n o retorno de carro) que volcar del input
boolean **hasNext(String)**	Devuelve true si en el input hay información que volcar que corresponda con el string introducido
boolean **hasNextBoolean()**	Devuelve true si el siguiente volcado corresponde con

	información que pueda ser interpretada como un boolean value
boolean **hasNextFloat()**	(...) como un float value
boolean **hasNextDouble()**	(...) como un double value
boolean **hasNextInt()**	(...) como un int value
string **next()**	Devuelve un strin que es la información que haya hasta el siguiente delimitador
string **nextLine()**	Devuelve un string que es la información que haya hasta el siguiente retorno de carro (\n)
int **nextInt()**	"" integrer value
double **nextDouble()**	"" double value
float **nextFloat()**	"" float value
boolean **nextBoolean()**	"" boolean value
Scanner **reset()**	Reinicia el Scanner.
Scanner **skip(String)**	Ignora información hasta encontrar el string introducido y se posiciona en ese lugar del input.

21.- <u>CLASE RANDOM</u> [java.util.Random]

Esta clase genera objetos que devuelven valores pseudoaleatorios.

Constructores:

Random()	Genera un objeto Random
Random(<long seed>)	Genera un objeto Random usando como semilla el objeto long introducido

Algunos de sus métodos que pueden sernos útiles son:

int **nextInt()**	Devuelve un valor integrer pseudoaleatorio
int **nextInt(Int)**	Devuelve un valor integrer pseudoaleatorio que esté entre el cero (incluido) y el numero dado (excluido)
double **nextDouble()**	Devuelve un valor double pseudoaleatorio
float **nextFloat()**	Devuelve un valor float pseudoaleatorio
boolean **nextBoolean()**	Devuelve un valor boolean pseudoaleatorio

22.- <u>CLASE SYSTEM</u> [java.lang.System]

Clase que hace referencia a las cosas del sistema. Tiene tres **fields** (variables) principales:

err --> Salida estándar de errores
in --> Entrada estándar de datos (normalmente el teclado)
out --> Salida estándar de datos (normalmente la pantalla)

Si enviamos información a **System.out** o **System.err** serán enviada al canal correspondiente y también podemos recoger información del canal **System.in**

La forma de escribir en pantalla es con el método **System.out.println(<aEscribir>)**

Algunos métodos que nos pueden interesar de la clase System son:

void **gc()** --> Ejecuta el recolector de basura (*"Runs garbage collector"*)
void **runFinalization()** --> Finaliza todos los métodos que estén pendientes de ser finalizados
void **setErr(PrintStream)** --> Reasigna el canal err
void **setIn(InputStream)** --> Reasigna el canal in
void **setOut(PrintStream)** --> Reasigna el canal out

23.- INTERFAZ MAP [java.util.Map]

La interfaz map se usa siempre con el siguiente formato:

Map<K,V> nombreMapa;

Donde K=key y V=Values. Para entender bien el concepto de cómo se estructura un Map lo mejor es imaginarse una tabla de dos columnas donde los valores de la primera columna (**K**) son los índices y los valores de la segunda columna (**V**) son los valores asociados a dicho índice.

Como valores *Key* y *Value* en la declaración del *Map* tenemos que introducir clases o interfaces, ejemplo de clases: *String*, ejemplo de interfaz: *List<K,V>*

Algunos de los métodos de la interfaz Map son:

boolean **containsKey(Object key)**	Devuelve true si el key introducido está entre los key del Map
boolean **constainsValue(Object value)**	Devuelve true si el value introducido está entre los value del Map
V **get(Object key)**	Devuelve el value asignado al key dado si éste está en el Map
boolean **isEmpty()**	Devuelve true si no hay keys en el maping
Set<K> **keySet()**	Devuelve un Set (colección) con todas las keys que hay en el Map
V **put(K key, V value)**	Relaciona el value dado con el key dado y los introduce en el Map
void **putAll(Map<? extends Key, ? extends Value> nombre)**	Introduce en el Map el nuevo Map proporcionado
V **remove(Object key)**	Borra todo el maping relacionado con la key dada
int **size()**	Devuelve el número de keys que hay en el Map
Collection<V> **values()**	Devuelve una colección con todos los values que hay en el Map

Algunas clases que implementan la interfaz Map son:

HashMap<K,V>	No permite elementos ordenados y los organiza mediante su hashCode. Método **hashCode()** necesario en Keys
TreeMap<K,V>	Organiza los elementos creando jerarquías de árboles, se necesita implementar **compareTo()** en los Keys
SortedMap<K,V>	Este tipo de Map garantiza la total organización de sus keys siempre que esté implementado la interfaz *Comparable<T>* en ellas

24.- CLASE FILE [java.io.File]

La clase file es un objeto que se relaciona con un fichero sirviendo para crear flujos entre el programa y éste.

Implementa la interfaz Comparable<File> así que tiene compareTo(File) y equals(File)

Uno de los constructores de esta clase es:

File(String <pathname>)	Crea un File usando el fichero que está en la dirección "pathname"

Algunos métodos que nos pueden ser útiles son:

boolean **canExecute()**	Devuelve true si se puede ejecutar el fichero mediante la aplicación
boolean **canRead()**	Devuelve true si se puede leer el fichero mediante la aplicación
boolean **canWrite()**	Devuelve true si se puede modificar el fichero mediante la aplicación
boolean **createNewFile()**	Crea un nuevo fichero vacío con el nombre del File si, y

	sólo si, el fichero con este nombre no existe en el directorio todavía. Devuelve true si lo crea.
boolean **delete()**	Borra el fichero con el nombre del File del directorio.
void **deleteOnExit()**	Se cerciora de que el fichero sea borrado cuando la Virtual Machine acabe
boolean **exists()**	Devuelve true si el fichero del File existe en el directorio
string **getName()**	Devuelve el nombre del fichero
string **getpath()**	Devuelve la posición del directorio (pathname) en forma de string (pathname string)
long **lastModified()**	Devuelve el momento en el que el fichero sufrió la última modificación
boolean **mkdir()**	Crea un directorio con el pathname del File
boolean **mkdirs()**	Crea el directorio con el pathname dado incluyendo todos los parent directores (directorios que contengan al que queremos crear),no existentes, que hagan falta
boolean **setExecutable(*boolean*)**	Modifica el permiso de "ejecución" de este directorio/fichero
boolean **setReadable(*boolean*)**	Modifica el permiso de "lectura" de este directorio/fichero
boolean **setWritable(*boolean*)**	Modifica el permiso de "modificación" de este directorio/fichero

Estos tres últimos métodos tienen una variable con dos argumentos *(boolean1,boolean2)* donde boolean1 indica el permiso general y boolean2 indica si el dueño (admin) del directorio/fichero puede o no realizar la acción.

boolean **setReadOnly()**	Modifica los permisos del directorio permitiendo sólo la lectura

Al crear un *File Object* hay una excepción de obligado tratamiento que es **FileNotFoundException()** también de **java.io**

25.- CLASE PRINTWRITER [java.io.PrintWriter]

Extends Writer.

Esta clase se utiliza para escribir en ficheros. Sus constructores son:

PrintWriter(File file)

PrintWriter(String nomFich)

Si no existe el archivo especificado, lo crea, si existe, borra el contenido y escribe en este documento pero ya en blanco.

Algunos métodos útiles para nosotros de esta clase son:

PrintWriter **append(Char c)**	Añade el caracter "c"
PrintWriter **append(CharSequence csc)**	Añade la secuencia de caracteres contenida en "csc"
PrintWriter **append(CharSequence csc, int i, int f)**	Añade la subsecuencia que comienza en "i" y acaba en "f"
void **close()**	Cierra el PrintWritter (flujo)
void **print(boolean b)**	Escribe el boolean value contenido en "b"
void **print(int i)**	Escribe el int value contenido en "i"
void **print(long l)**	""
void **print(double d)**	""
void **print(Object obj)**	Escribe lo que devuelva obj.toString()
void **print(String str)**	Escribe lo que contenga "str" y si str==null escribe el string "null"

Igual ocurre con los comandos println(...) que hacen lo mismo sólo que añaden un final de lineal (\n) tras escribir