Módulo: Programación Java

Objetivos formativos:

- Conocer lo que es Java, cuáles son sus características principales y una aproximación a la **Programación Orientada a Objetos**.
- Profundizar en las características principales del **lenguaje** y sus diferencias con aquellos lenguajes relacionados.
- Conocer la **estructura modular** en un programa Java, así como todos los elementos que podemos encontrar dentro de programación orientada a objetos en Java.
- Aprender como crear un programa Java, para posteriormente **compilarlo** y probar su **ejecución**.

Pantalla: 1. Introducción

Definición:

Java es simplemente, un lenguaje de programación orientado a objetos de propósito general, desarrollado por Sun Microsystems a principios de 1991, con el que se van a poder crear tanto programas asociados a páginas HTML (applets) como programas independientes de éstas (aplicaciones). Y todo ello, independiente de la plataforma de computación. Los programas hechos en Java podrán ejecutarse en INTEL, MOTOROLA, Solaris, Windows y Macintosh, entre otros.

Desarrollo:

Java es un lenguaje de **programación orientado a objetos**. Orientado a objetos significa que Java organiza sus programas en una colección de objetos. Ésto nos va a permitir estructurar los programas de una manera más eficiente y en un formato más fácil de comprender.

Además, Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos, entre las que destacan:

- Aritmética de punteros.
- Macros (#define).
- No existen referencias.
- Definición de tipos (typedef).
- Registros (struct).
- Necesidad de liberar memoria (free).

Más info:

Java es un Software formado por pequeños módulos que pueden proceder de sistemas distintos, de máquinas diferentes y que se irán ejecutando según sean necesarios.

Está diseñado para que las aplicaciones generadas por Java se puedan ejecutar en cualquier plataforma de computación que soporte una Máquina Virtual Java.

Una **Máquina Virtual Java (JVM)**, es una capa de software que reside o se ejecuta encima del sistema operativo de un ordenador, y que permite que la plataforma ejecute Java, siendo esta la que se encargará de interpretar las instrucciones según sea la arquitectura en la que nos encontremos.

Pantalla: 2. Características de Java

Introducción:

Veamos las características principales del lenguaje:

Desarrollo:

Simple:

Este lenguaje presenta una gran facilidad de aprendizaje. Java elimina la complejidad de los lenguajes como "C" y "C++". Todas aquellas personas familiarizadas con "C++" podrán migrar con relativa facilidad al entorno Java.

Con Java se pueden desarrollar pequeñas aplicaciones sin necesidad de conocer todos los entresijos del lenguaje, e ir añadiendo nuevos elementos y funcionalidades según vayamos avanzando en su conocimiento.

Orientado a Objetos:

La orientación a objetos es uno de los pilares básicos de Java.

La orientación a objetos descompone el problema a resolver en entidades independientes (objetos) pero relacionadas entre sí. El primer lenguaje orientado a objetos que apareció fue el C++, el cual estaba destinado a sustituir al lenguaje C, o por lo menos a utilizarse conjuntamente con éste.

En definitiva, la filosofía de la programación orientada a objetos en Java es diferente a la de la programación convencional. Los objetos van a agrupar en estructuras encapsuladas, tanto sus datos como los métodos que manipulan esos datos. Las definiciones de estos objetos se van a denominar **clases**. Además, como todo lenguaje orientado a objetos soporta aspectos tales como encapsulación, herencia y polimorfismo.

Distribuido:

Un sistema distribuido nos va a permitir compartir los recursos que nos proporcionan un conjunto de programas localizados en una serie máquinas situadas en redes diferentes. Aunque Java en sí no es distribuido, nos proporciona las librerías y herramientas necesarias para que los programas puedan serlo, es decir, para que se puedan ejecutar en varias máquinas y puedan interactuar entre sí.

Robusto:

Un lenguaje de programación que favorezca la robustez del software suele poner más restricciones al programador a la hora de escribir el código fuente y Java está diseñado para crear software altamente fiable. Su escritura de datos, por tanto, será más rigurosa. Así, proporciona gran cantidad de **comprobaciones** tanto en tiempo de compilación como en tiempo de ejecución.

Seguro:

Debido al carácter distribuido de Java, donde se van a poder descargar programas desde cualquier parte de la Red, la seguridad es un aspecto de vital importancia.

Por otra parte, el sistema de Java cuenta con ciertas políticas que evitan que se puedan codificar virus con este lenguaje, sin olvidar además que existen muchas otras restricciones que limitan lo que se puede o no se puede hacer con los recursos críticos de una máquina.

Interpretado:

Al compilar en Java el código fuente de un programa, se obtiene un archivo en formato byte-code, que posteriormente necesitará de una máquina virtual Java (JVM) para que lo interprete y lo ejecute.

Esto implica que la puesta en marcha de los programas Java resulte más lenta, ya que el código debe ser interpretado y no ejecutado directamente tal como ocurre en otros lenguajes de programación. Se está avanzando mucho en mejorar este aspecto de Java, la rapidez de ejecución.

Independiente de la Arquitectura:

Ésta es una de las principales características de Java. El código compilado de Java se va a poder usar en cualquier ordenador. Esto es debido a que, como se ha mencionado dicho código compilado, también conocido como byte code, es interpretado. De esta manera se consigue que el programa pueda ser ejecutado en una cantidad de plataformas sin tener que recompilar el código. No contiene instrucciones relacionadas específicamente con una plataforma. Esto se traduce en que si se escribe un programa en una Sun y se compila, el compilador generará el mismo byte code que un ordenador con Windows 98, por ejemplo.

Multihebra:

Java permite elaborar programas que permitan ejecutar varios procesos al mismo tiempo sobre la misma máquina. La característica que proporciona esta funcionalidad son los hilos de ejecución (multithreading). Son especialmente útiles en la creación de aplicaciones de red distribuidas.

Cualquier usuario de Internet, sabe lo frustrante que puede ser esperar por una gran imagen que se está trayendo. En Java, las imágenes se pueden ir trayendo en un thread independiente, permitiendo que el usuario pueda acceder a la información en la página sin tener que esperar por el navegador.

Pantalla: 3. Programación Orientada a Objetos

Introducción:

Desarrollo:

Frente a la programación tradicional, la programación orientada a objetos es una disciplina que descansa en el concepto de objeto para imponer una estructura modular a los programas.

Así, el paradigma orientado a objetos se basa en tres métodos de organización: emplea:

- la diferencia entre un objeto y sus atributos (por ejemplo, entre un camión y su altura o peso).
- la diferencia entre un objeto y sus componentes (por ejemplo, entre un camión y sus ruedas o motor).
- la formación y distinción entre clases de objetos (por ejemplo, entre camiones, turismos y bicicletas).

Objetos:

Básicamente podría definirse como un conjunto complejo de datos que poseen una estructura determinada y forman parte de una organización. Un objeto es la representación de un concepto.

En el ámbito de la programación podría decirse que todos los programas se van a dividir en componentes (objetos) reusables. Cada uno de estos componentes tiene la habilidad de interactuar con otros y crear un programa con la funcionalidad que nosotros queramos darle. Algunos de estos componentes tienen que ser creados. Otros se "tomarán prestados" de los objetos que vienen con el lenguaje Java.

Si nos centramos en términos de programación, un objeto podría definirse como un conjunto de variables (atributos) y de métodos relacionados con esas variables. Un poco más sencillo: un objeto contiene en sí mismo la información y los métodos o funciones necesarios para manipular esa información.

Clases:

Las clases son el núcleo de Java. Definen la forma y la naturaleza de un objeto y van a constituir la base de la programación orientada a objetos. Simplificando podría decirse que una clase es tan sólo un modelo (plantilla) para un objeto, y que define las variables y los métodos comunes a un cierto tipo de objetos.

Primero deberemos crear una clase antes de poder crear objetos o ejemplares de esa clase.

En Java las clases se emplean para definir nuevos tipos de datos. Cada nuevo tipo se podrá utilizar para crear objetos de ese tipo. Para definir una clase usaremos la palabra clave **class**.

Atributos y Métodos:

Un atributo es un valor o propiedad propia de un objeto. En términos de programación se asemeja a lo que es una variable. Por ejemplo, color, número de marchas o fabricante podrían ser atributos usados en un objeto "Bicicleta".

Todos los objetos en una clase comparten las mismas operaciones. Un método es la implementación de una operación para una clase, y su finalidad será controlar los datos de un objeto. Los métodos podrían asemejarse a lo que en la programación tradicional se conoce como Procedimientos o Funciones. Siguiendo con el mismo ejemplo, Acelerar, Frenar o Cambiar de marcha podrían ser métodos para un objeto "Bicicleta".

Encapsulación:

Hemos visto que un programa en Java es un conjunto de clases, cada una de las cuales contiene métodos (funciones) y atributos (datos) propios. De este modo, el código de una clase puede considerarse independiente del de todas las demás. Pero para que este mecanismo tenga sentido y sea verdaderamente útil, es preciso asegurar que los elementos contenidos dentro de una clase son responsabilidad única y exclusivamente de ésta, y no pueden ser modificados por ninguna otra.

El empaquetamiento de los atributos y sus métodos, dentro de un objeto, es lo que llamamos **encapsulación**. No es más que ocultar información. Su finalidad es la de proteger los atributos que componen el objeto y permitir o negar información al "público". Por ejemplo, para cambiar las marchas de una bicicleta no se necesita conocer el mecanismo del cambio, simplemente información de cómo mover el cambio. De esta forma la implementación podrá cambiarse sin que tengan que modificarse otras partes del programa que usen este objeto.

HERENCIAS:

Se van a poder crear clases partiendo de otras que ya existan. Éstas nuevas clases heredarán todos los atributos y métodos de la clase a partir de la cual fueron creadas. A la clase a partir de la cual fueron creadas las demás se la denomina Superclase y a las nuevas clases creadas, Subclases.

En principio, los métodos de la "clase padre" o Superclase pueden ser usados por las "clases hijas", aunque también existe la posibilidad de que éstos puedan ser redefinidos de nuevo por ellas, ocultando así los de su Superclase.

La principal aportación que nos va a ofrecer la herencia va a ser la posibilidad de reutilizar código.

POLIMORFISMO:

Existen dos tipos de polimorfismo:

- Funcional (Sobrecarga): consiste en referenciar, mediante el mismo identificador, diferentes funciones/métodos. El intérprete los distingue por el tipo y el orden de los parámetros.
- De Datos: es la capacidad de un mismo identificador para hacer referencia a objetos de distintas clases.

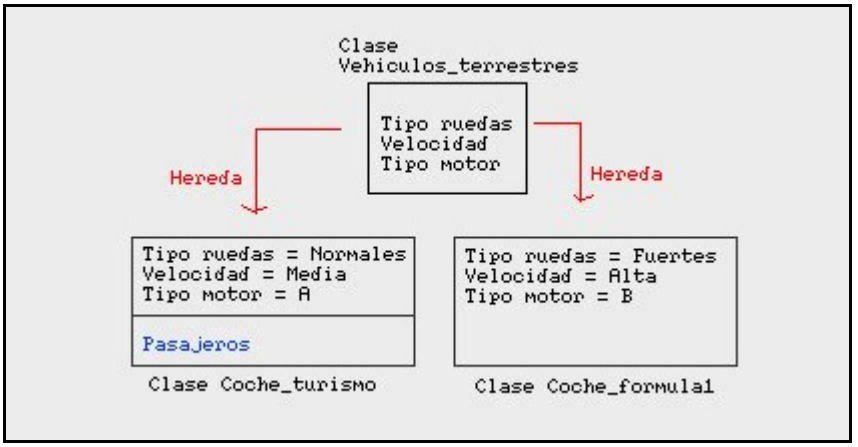
CLASES ABSTRACTAS

Un método abstracto es un método que no está implementado: simplemente se indica su nombre, la lista de parámetros y el tipo de dato devuelto. La clase que herede de esta clase abstracta es la que implementará el método abstracto.

Cuando una clase contiene un método abstracto debe declararse como abstracta. Sin embargo, no necesariamente todos los métodos de una clase abstracta tienen que ser abstractos. La finalidad principal de este tipo de clases es la de ser heredadas por otras. Son clases que no van a poder estar solas, se emplean únicamente para definir otras y nunca van a poder tener objetos ni ser instanciadas directamente.

EJEMPLO:

Veámoslo con un ejemplo:



En este caso la clase vehículo_terrestre es la Superclase y tanto Coche_turismo como Coche_formula1 las subclases. Las dos subclases heredan los atributos tipo ruedas, velocidad y tipo motor de la superclase Vehículos_terrestres. Además, la subclase coche_turismo cuenta con un atributo adicional, pasajeros.

Más info:

Clases:

En todo lenguaje orientado a objetos existe una jerarquía de clases relacionadas entre sí a través de términos de herencia. En java, el punto más alto de la jerarquía es la clase Object, quien proporciona una serie de capacidades básicas a *todas* las demás clases, es decir, todas las clases, tanto las que vienen con Java como las que creemos nosotros, heredan de Object.

Al proceso de crear un objeto de una clase se llamará **instanciar** un objeto y, a su vez, un objeto creado de una clase determinada se denominará instancia de esa clase.

Herencias:

En Java todas las clases van a ser subclases de una clase llamada Object, y como consecuencia van a heredar todos sus métodos y propiedades.

Pantalla 4. Compilación y Ejecución

Desarrollo

Vamos a realizar nuestro primer programa java, con ello podréis comprobar que la instalación de vuestro JDK (Java Development Kit) funciona correctamente.

Para compilar un fichero tenemos que teclear:

Javac mifichero.java

Ésto genera un fichero MiFichero.class. Este fichero contiene un código intermedio (formado por bytecodes) que es interpretado por la JVM (Java Virtual Machine), la máquina virtual Java.

Para ejecutar una aplicación Java tenemos que teclear:

Java michero

No siendo necesaria la inclusión de la extensión.

Ejemplo

El siguiente ejemplo mostrará un mensaje por pantalla.

```
import java.io.*;

class Saludo{
   public static void main(String args[]){
            System.out.println("Hola a todos");
   }
}
```

Este fichero tiene que ser guardado con el mismo nombre de la clase, es decir, Saludo.java (observe la segunda línea de código). Ahora sólo tenéis que compilarlo y ejecutarlo.

Más info.

Instalación

Para realizar un programa Java necesitamos el Kit de desarrollo JDK (Java Development Kit) que podéis obtener de forma gratuita de la página de Sun MicroSystems. Es recomendable utilizar las versiones 1.2 y superiores. Su instalación es sencilla y sólo tenéis que seguir los pasos que se os indica en la instalación.

Una vez instalado será necesario modificar la variable de entorno PATH para que sea posible encontrar el compilador desde cualquier ruta. El compilador y la máquina virtual Java se encuentran en el subdirectorio (carpeta) bin que cuelga del directorio donde habéis instalado el JDK, por ejemplo (para Windows) c:\jdk1.3\bin.

También os será necesario un editor para facilitar la creación de programas Java. Sería ideal que el editor tuviese como facilidad la compilación y la ejecución de programas Java. Alguno de estos editores puede ser WinEdit, Kawa, JPadPro, etc. Aunque siempre se puede utilizar un

simple editor que permita crear ficheros de textos y realizar la compilación y ejecución de programas Java.

Pantalla 5. Variables y Tipos de Datos en Java.

Desarrollo

Las variables nos permiten guardar valores y datos sobre los que podremos actuar con los **operadores**.

Tipos de Datos

Todas las variables en Java deben tener un **tipo de dato**. El tipo de la variable determina los valores que la variable puede contener y las operaciones que se pueden realizar con ella.

Existen dos categorías de datos principales en Java: los **tipos primitivos** y los **tipos referenciados** (objetos).

Primitivos

Los tipos primitivos contienen un sólo valor y pueden ser los enteros, coma flotante, los caracteres, etc. La tabla siguiente muestra todos los tipos primitivos soportados por Java:

TIPO	TAMAÑO Y FORMATO	VALOR POR DEFECTO	DESCRIPCIÓN	CLASE ASOCIADA
byte	8-bit complemento a 2	(byte)0	Entero de un byte	Byte
short	16-bit complemento a 2	(short)0	Entero corto	Short
int	32-bit complemento a 2	0	Entero	Integer
long	64-bit complemento a 2	0L	Entero largo	Long
float	32-bit IEEE 754	0.0f	Coma flotante de precisión simple	Float
double	64-bit IEEE 754	0.0d	Coma flotante de precisión doble	Double
char	16-bit carácter	\u000	Un carácter	Character
boolean	1-bit	false	Booleano (verdadero o falso)	Boolean

Referenciados

Para poder trabajar con los tipos primitivos como si fueran clases, cada tipo primitivo tiene una clase asociada, que puede almacenar el mismo tipo de información pero siempre a través de objetos o tipos referenciados.

Los tipos referenciados se llaman así porque el valor de una variable de referencia o manejador es un puntero hacia el valor real.

Identificadores

En un programa no referimos al valor de una variable por su identificador, que es el nombre que se le da a la variable.

Por convención, en Java, los nombres de las variables empiezan con una letra minúscula y los nombres de las clases empiezan con una letra mayúscula. Si una variable está compuesta de más de una palabra, las palabras se ponen juntas y cada palabra después de la primera empieza con una letra mayúscula.

También es posible la definición de varias variables del mismo tipo con una única sentencia e incluso inicializarlas en el momento de su definición.

Ejemplos

Declaración de Variables:

tipoVariable nombreVar1

```
int edadAlumno;
```

tipoVariable nombreVar1,nombreVar2= valorInicial2;

```
float pi=3.14,x;
```

Más info:

Identificadores

Un identificador de variable, en Java, tiene que cumplir:

- Debe ser un identificador legal de Java comprendido en una serie de caracteres Unicode. **Unicode** es un sistema de codificación que soporta texto escrito en distintos lenguajes humanos. Unicode permite la codificación de 34.168 caracteres. Esto le permite utilizar en sus programas Java varios alfabetos como el Japonés, el Griego, el Ruso o el Hebreo. Esto es importante para que los programadores puedan escribir código en su lenguaje nativo. Por lo tanto podremos utilizar la ñ como parte de un identificador de variable, clase, etc.
- No puede ser el mismo que una palabra clave o el nombre de un valor booleano (true or false).
- No deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito para evitar confusiones aunque si es posible su declaración.

Pantalla 6. Sentencias de Control de Flujo (I). If- Else

Introducción

Por defecto el flujo de ejecución de un programa es secuencial, cuando no nos interese que esto sea así usaremos las sentencias de control de flujo.

Desarrollo

Podemos dividir las sentencias de control de flujo en:

- Toma de decisión (if-else, switch-case)
- Bucles (for, while, do-while)

Sentencia if-else

Permite la ejecución de una serie de sentencias en función de que se cumpla una determinada condición. La estructura de la sentencia if es:

```
If  (condición){
        sentenciasSI;
}
else{
  sentenciasNO;
}
```

Existe otra forma de la sentecia else, el else if que ejecuta una sentencia basada en otra expresión, es lo que se llama if anidados.

Ejemplo:

Sentencia if-else

```
If  (a>3){
  a--;
}
else{
  b++;
  c*=2;
}
```

Ejemplo de sentencia if-else anidada.

Por ejemplo supongamos que creamos un programa que asigna notas basadas en la puntuación de un examen, un Sobresaliente para una puntuación del 90% o superior, un Notable para el 80% o superior y demás. Podríamos hacerlo de la siguiente manera:

```
int puntuacion;
String nota;

if (puntuacion >= 90) {
   nota = "Sobresaliente";
} else if (puntuacion >= 80) {
   nota = "Notable";
} else if (puntuacion >= 70) {
```

```
   nota = "Bien";
} else if (puntuacion >= 60) {
    nota = "Suficiente";
} else {
    nota = "Insuficiente";
}
```

Pantalla 7. Sentencias de Control de Flujo (II). Switch

Puede ser una alternativa a la sentencia if-else cuando se compara la misma expresión con distintos valores.

Su sintaxis es la siguiente:

```
switch (expresión){
  case value1: sentencias1;break;
  case value2: sentencias2;break;
  case value3: sentencias3;break;
  case valuen: sentenciasn;break;
  default: sentenciasD;break;
}
```

Se va evaluando la expresión por cada case, hasta cumplir la condición, momento en el cual se ejecuta la sentencia correspondiente.

Si el valor de la expresión no coincide con el valor de ningún case se ejecuta la rama **default**, si es que aparece, ya que ésta es opcional.

Si no aparece la sentencia **break,** se ejecuta una sentencia case y se ejecutan también todas las que van a continuación hasta que se llega a un break o hasta que se termina la sentencia switch.

Ejemplo

El siguiente código evalúa la variable identificada como *letra*, e imprime un mensaje por pantalla indicando si ésta es consonante o vocal.

Para mostrar un mensaje por la pantalla utilizamos la sentencias
s**ystem.out.println(cadena);**

```
switch (letra){
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u': System.out.println(" Es una vocal ");break;
  default: System.out.println(" Es una consonante ");break;
}
```

Pantalla 8. Sentencias de Control de Flujo (III). Bucles

Definición

Los bucles se usan para repetir un proceso un determinado número de veces.

Desarrollo

En java nos encontramos tres tipos de bucles:

- Bucles while
- Bucles for
- Bucles do while

While

Una sentencia **while** realiza una acción mientras se cumpla una cierta condición. Este tipo de bucle se suele utilizar cuando no conocemos el número de veces que se tiene que ejecutar las sentencias del bucle.

La sintaxis general de la sentencia while es: **while (expresión) sentencia**

O sea, mientras la expresión sea verdadera, se ejecutará la sentencia, donde sentencia puede ser una única sentencia o un bloque de sentencias.

Esto es, mientras la expresión sea verdadera, ejecutará la sentencia. sentencia puede ser una sola sentencia o puede ser un bloque de sentencias. Si sólo se quiere ejecutar una sentencia no es necesario utilizar ponerlas llaves.

Ejemplo

El siguiente código va leyendo y contando caracteres hasta que se encuentra "–1", mostrando al final un mensaje con el total de caracteres.

```
while (System.in.read() != -1) {
  contador++;
  System.out.println("Se ha leido un el carácter = " + contador);
}
```

For

El bucle **for** se utiliza generalmente cuando conocemos el número de veces que queremos ejecutar el cuerpo de un bucle. La forma general del bucle for es la siguiente:

```
for (inicialización;condición;incrementos){
  sentencias
}
```

Donde **inicialización** se ejecuta al comienzo del for la primera vez del bucle, e **incrementos** después de cada ejecución de **sentencias**. La condición se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de **condición** no se cumple.

Ejemplo:

El siguiente código va contando e imprimiendo por pantalla: Número1, Número2…. hasta Número9, iteración a partir de la cual i vale 10 y dejará ya de cumplirse la condición.

```
for (int i=0;i<10;i++){
  System.out.println("Número " + i);
}
```

Do-While

Este tipo de bucles se usa cuando queremos que se ejecuten las instrucciones del bucle al menos una vez, ya que en este tipo de bucles la condición se evalúa al salir.

Su sintaxis básica es la siguiente:

```
do{
  sentencias
}while(condición);
```

Primero se ejecuta la sentencia y luego se avalúa la condición, si ésta se cumple se volvería a ejecutar la sentencia y así sucesivamente hasta que la condición no se cumpla.

Ejemplo

El siguiente ejemplo genera al menos un número aleatorio entre 0 y 10 y lo imprime por pantalla, y si el número generado es menor a 5 se sigue ejecutando el bucle hasta que i valga 5.

```
int i=(int)Math.random()*10;
do{
  System.out.println("Número " + i);
  i++;
}while(i<5);
```

Mas info:

Un **bloque de sentencias** en Java es un conjunto de sentencias contenidas dentro de corchetes('{'y '}').

Pantalla 9. Sentencias de Control de Flujo (IV). Break, Continue, Return.

Desarrollo

Dentro de un bucle podemos encontrarnos los siguientes comandos:

Break

Hace que se salga inmediatamente del bucle o bloque que se está ejecutando sin finalizar el resto de las sentencias.

Continue

Se utiliza en los bucles y finaliza la iteración "i" que en ese momento se está ejecutando (no ejecuta el resto de setencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración "i+1".

Return

Esta sentencia se utiliza para salir del método actual y volver a la sentencia siguiente a la que originó la llamada en el método original.

Existen dos formas de return: una que devuelve un valor y otra que no lo hace. Para devolver un valor, simplemente se pone el valor (o una expresión que calcule el valor) detrás de la palabra return.

Ejemplo que devuelve un valor: **return ++count;**

Ejemplo que no devuelve valor: **return;**

Ejemplo

El siguiente ejemplo muestra en mensaje "Número " para todos los valores de i de 0 a 9 menos cuando vale 5 que con la sentencia continue se provoca el salto a la siguiente iteración del bucle, sin ejecutar la sentencia de impresión en pantalla, correpondiente a i con valor igual a 5.

```
for (int i=0;i<10;i++){
  if (i==5) continue;
  System.out.println("Número " + i);
}
```

Mientras que en el siguiente ejemplo cuando i vale 5 se interrumpe la ejecución del bucle sin ejecutar ninguna iteración más.

```
for (int i=0;i<10;i++){
  if (i==5) break;
  System.out.println("Número " + i);
}
```

Pantalla 10. Clases

Definición:

Una clase es una agrupación de datos (variables, campos o atributos) y de funciones (métodos) que operan sobre esos datos.

Desarrollo:

Su sintaxis básica es como sigue:

```
class NombreClase{
  //definición de variables y métodos
  ...
}
```

Los nombres de las clases deben ser un identificador legal de Java y, por convención, deben empezar por una letra mayúscula.

Un objeto o instancia es un ejemplar concreto de una clase. Las clases son como tipos de variables, mientras que los objetos son como variables concretas de un tipo determinado.

Cada objeto de una clase tiene su propia copia de las variables miembro.

Una declaración de variable puede tener la sintaxis siguiente:

[modificadordeAcceso] [static] [final] [transient] [volatile] tipo nombredeVariable

Las variables miembro pueden ir precedidas en su declaración por modificadores de visibilidad o acceso. Éstos estableceran qué clases y métodos va a tener permiso para utilizar la clase, sus métodos y variables miembro. Estos modificadores los veremos más adelante.

Ejemplo

```
class Cuadrado{
  int x=0,y=0;  //variables miembro
  int lado;       //variable miembro

  public int area(){
      ...
  }
}
```

Pantalla 11. Clases

Desarrollo

La variables de una clase pueden ser **variables miembro de la clase** declaradas como **static**, o pueden ser **variables finales** declaradas como **final.**

Variables miembro de la clase

Una clase puede tener variables propias de la clase y no de cada objeto, estas variables son las **variables miembro de la clase** o **variables static**. Las variables static se suelen utilizar para definir constantes comunes para todos los objetos de la clase. Por ello no es necesario crear un objeto para poder hacer referencia a una variable de clase y se pueden referenciar directamente.

Podemos referenciar una variable de este tipo de dos formas: mediante el nombre de la clase o utilizando un objeto de esa clase. Se recomienda la primera forma.

Las variables static es lo más parecido que tiene Java a las variables globales.

Variables finales

El valor de una variable declarada como **final** no puede cambiar durante la ejecución de un programa Java. Es equivalente a la creación de una constante, pero no se ajusta exactamente al concepto de constante que conocemos en programación.

Java permite separar la definición de la inicialización de una variable final. La inicialización puede realizarse con posterioridad a la definición, pero una vez establecido un valor éste no podrá ser cambiado durante la ejecución del programa. Por lo tanto esta variable será constante (no puede cambiar) pero eso no quiere decir que en todas las ejecuciones del programa tenga el mismo valor.

Ejemplo

Ejemplo variables miembro de la clase

```
class Coche{
  public int precio;
  public static int plVA;
}
```

Podemos hacer referencia a plVA utilizando el nombre de la clase, **Coche.plVA** o creando un objeto de la clase:

```
Coche c= new Coche();
System.out.println("Porcentaje de IVA " + c.plVA);
```

Con esto conseguimos que todos los objetos de una clase compartan una variable. En este caso el porcentaje de IVA (plVA) es el mismo para todos los objetos Coche.

Ejemplo. Variables finales.

```
class Avo {
   final double AVOGADRO = 6.023e23;

   public double calcular(){
      .....
      return x* AVOGADRO;
   }

   public double calcular2(){
      .....
      AVOGADRO= 6.02; //esto no es posible. Error de compilación
      return x* AVOGADRO;
   }

}
```

Más info:

Las variables miembro static se inicializan siempre antes que cualquier objeto de la clase.

Declarar como final un objeto miembro de una clase hace constante la referencia, pero no el propio objeto, que puede ser modificado.

Curiosidades:

En Java no es posible hacer que un objeto sea constante.

Pantalla 12. MÉTODOS

Los métodos son funciones definidas dentro de una clase. Salvo los métodos static, se aplican siempre a un objeto (**argumento implícito**) de la clase por medio del operador punto(.).

```
[modificadordeVisibilidad] [static] [final] tipo nombreMétodo(argumentos){
  //definición de variables locales al método
  //cuerpo del método
  ...
}
```

modificadordeVisibilidad: establece el ámbito que tiene el método, es decir, desde donde es visible el método.

- **tipo**: tipo de datos que devuelve el método.
- **nombreMétodo**: identificador Java válido.
- **argumentos**: lista de argumentos separados por comas en los que se indica el tipo del argumento y un identificador válido.

El tipo del **valor de retorno** puede ser un tipo primitivo o el nombre de una clase, para el caso en el que el método devuelva una referencia a un objeto de una determinada clase.

Los métodos pueden definir variables locales. El ámbito de estas variables es el bloque en el que son definidas. No es necesario inicializar las variables en el punto en el que se definen pero el compilador obliga a darles un valor antes de usarlas. En este caso el compilador no las inicializa con un valor por defecto.

Mas info:

Un método siempre tiene visibilidad sobre las variables miembros del objeto sobre el que se ejecuta el método.

También podemos acceder a éstas variables mediante la referencia **this** que siempre apunta al objeto sobre el que se ejecuta un método.

Ejemplo:

```
class Coche{
  public int precio;
  public static int plVA=16;

  public double calcularPrecio(){
    double p=(double)precio; //declaración de variable local p.
                             //visibilidad de la vble miembro precio
    p+=p*(plVA/100.0);       //uso de expresiones y operadores y acceso vble static
    return p;                //devolvemos el valor que produce el método
  }
}
```

Pantalla 13. Métodos sobrecargados

Definición

La sobrecarga de métodos o polimorfismo funcional es la posibilidad de tener **métodos distintos** con el **mismo nombre** que se diferencia por el número y/o tipo de los argumentos.

Desarrollo

Veamos un ejemplo:

```
public class Buscador{
  public static int buscador(Lista l, Elemento e){
        ...
  }
  public static int buscador(Cola c, Elemento e){
        ...
  }
}
```

En la clase **Buscador** se definen dos métodos static con el mismo nombre, **buscador.** Ambos métodos realizan la búsqueda de un elemento en una estructura de datos y devuelve el índice que ocupa el elemento buscado en la estructura. El método buscador está sobrecargado y se diferencia uno del otro en el tipo de sus argumentos, más concretamente en el tipo de datos del primer parámetro (uno es Lista y el otro es Cola).

Las reglas que sigue Java para determinar el método que debe llamar en un momento determinado son:

- Si existe un método en el que coincide exactamente con los tipos de los argumentos y el número de estos, se ejecuta este método.
- Si no existe un método que se ajuste exactamente, se intenta promocionar los argumentos al tipo inmediatamente superior (char a int, int a long, float a double, etc.).
- Si el caso anterior no es posible, el programador puede realizar un **cast** para ajustarse al tipo del parámetro que espera el método, pero queda a responsabilidad del programador.
- Nunca se puede sobrecargar por el tipo del valor de retorno. No puede haber dos métodos que sólo se diferencien en el tipo de retorno.

Más info:

En Java se realizan de modo automático conversiones implícitas de un tipo primitivo a otro de más precisión, por ejemplo de int a long, de float a double, etc. Estas conversiones se hacen al realizar operaciones en las que los operandos son de distinto tipo o cuando realizamos sentencias de asignación en el que el miembro izquierdo tiene un tipo distinto que el resultdo de evaluar el miembro derecho.

Las conversiones de un tipo mayor a otro menor **no son seguras**, puesto que se puede perder información. Por este motivo estas conversiones tiene que hacerlas el programador de forma explícita. Este tipo de conversiones se llaman **cast**. El **cast** se hace poniendo el tipo al que se desea transformar entre paréntesis.

Ejemplo d=double(h/7)

Pantalla 14. Métodos de la clase

Un método static está asociado a la clase y no a objetos en particular. Un método static no se ejecuta sobre ningún objeto concreto, por lo que dentro del método no se tiene acceso a los elementos no estáticos de la clase (variables miembro) ya que éstos sólo tienen sentido cuando se trabaja con instancias (objetos) concretas de la clase.

Veamos un ejemplo:

```
class Coche{
  public int precio;
  public static int plVA;

  public static void setplVA(int n){
        plVA= n;
  }
}
```

El método static **setplVA** actualiza el valor de la variable static **plVA**. Dentro de este método no podremos usar nunca la variable **precio**, ya que es una variable miembro y nunca podremos hacer referencia a **this**.

Pantalla 15. Constructores

Definición

Un **constructor** es un método que se llama automáticamente cuando se crea un objeto de una clase.

Desarrollo

La misión del constructor es la reserva de memoria e inicializar las variables miembro de la clase.

Los constructores no devuelven ningún valor (ni siquiera void) y tienen el mismo nombre que la clase a la que pertenecen.

Un constructor de una clase puede llamar a otro de la misma clase previamente definido. En este caso, la primera sentencia del constructor tiene que ser la llamada al otro constructor de la clase. Para ello utilizamos la palabra reservada **this**.

Más info:

Una clase puede tener varios constructores (con el mismo nombre, sobrecarga). Si en un momento determinado no definimos ningún constructor, Java siempre nos creará el **constructor por defecto.** El constructor por defecto es aquel que no recibe parámetros, que no tiene argumentos.

Ejemplo:

```
public class Persona{
    private String nombre;
    private String apellidos;
    private int edad;
    private int dni;

    Persona(String nombre,String apellidos,int edad,int dni){
        this.nombre= nombre;     //evitamos confusion entre argumentos y vble
miembro
        this.apellidos= apellidos;
        this.edad= edad;
        this.dni= dni;
    }

    Persona(String n,String a){ //sobrecargamos el constructor
        this(n,a,0,0); //llamada al constructor de la misma clase.Tiene que ser la
primera línea
    }

    Persona(String n){ //sobrecargamos el constructor
        apellidos = "López";
        this(n,"",0,0); //No es posible. Tiene que ser la primera línea del
constructor
    }

        ...
}
```

Como se puede comprobar en el ejemplo la referencia ***this*** *siempre referencia al objeto implícito (objeto sobre el que se ejecuta el método) y en el caso de los constructores una llamada a un constructor de la propia clase. El uso de this(...) sólo se puede hacer dentro de un constructor.*

Pantalla 16 Objetos

Definición

Todo en Java son objetos salvo los tipos primitivos.

Un objeto es un ejemplar concreto de una clase. En las clases se definen como serán todos los objetos que pertenezcan a una misma clase.

Creación de objetos

En Java, para crear un objeto utilizamos el operador **new**.

Siempre es necesario hacer un new para crear un objeto.

Es necesario tener muy claro que cuando se declara una referencia de una clase: **Circulo c;**

no estamos creando un objeto de la clase Circulo, sino que hemos creado una variable (c) que tendrá la referencia a un objeto de la clase Círculo.

Para crear un objeto de la clase Circulo tenemos que utilizar el operador **new** seguido por un constructor de la clase, de la siguiente forma:

Circulo c = new Circulo(2.4,3.5);

Hemos creado un objeto e inicializado con el constructor de la clase Circulo (que tiene dos argumentos de la clase double). Ahora la variable c referencia a un objeto, concreto.

Usar objetos

Una vez hemos creado un objeto querremos acceder a su variables, ejecutar métodos sobre esos objetos, etc. Para ello, en Java, se utiliza el operador (**.**).

Para acceder a las variables de un objeto utilizamos la sintaxis **objeto.variable**, para ejecutar un método de la clase a la que pertenece el objeto utilizamos la sintaxis **objeto.metodo(argumentos)**:

Si deseamos ejecutar un método de la clase a la que pertenece el objeto también utilizamos el operador (.). Siempre una llamada a un método llevará paréntesis y entre ellos los argumentos que necesite. En el caso de que el método no necesite parámetros también es necesario poner los paréntesis.

Destrucción de objetos

En Java no nos tenemos que preocupar de liberar la memoria que ocupan los objetos. Es el sistema el que se ocupa de liberar la memoria cuando algún objeto **pierde la referencia**. Perder la referencia consiste en que no haya una variable que contenga una referencia al objeto, es decir no existe ninguna variable por la que se pueda acceder al objeto. A esta característica de Java se le llama **recolector de basura** (garbage collection).

Ejemplo

Definición de la clase y creación del objeto.

```
public class Circulo{
  public double x;
  public double r;
  public static double PI = 3.1416;

  public Circulo(double x, double r){
        this.x= x;
        this.r= r;
  }

  public double area(){
    return PI * r* r;
  }
}

Circulo c = new Circulo(3.0,5.0);
```

Uso del objeto:

```
c.r;
c.x;
double d = c.area();
```

Con la sentencia **c.r;** accedemos a la variable **r** del objeto, cuyo valor es 5.0. Podemos acceder a esta variable de forma directa porque el modificador de acceso de la variable es **public**.

De igual foma accedemos a la variable **x**. Y por ultimo ejecutamos el método **area**.

Pantalla 17. Herencia (i)

Desarrollo:

Se puede construir una clase a partir de otra mediante el mecanismo de la **herencia**. Para indicar que una clase deriva de otra se utiliza la palabra **extends**, de la forma:

class Subclase extends SuperClase{ ...}

En ejemplo más concreto:

class Punto3D extends Punto{ ...}

La clase Punto3D hereda/amplia de/a la clase Punto.

Cuando una clase deriva de otra **hereda todas sus variables y métodos**. Estas variables y métodos miembro pueden ser redefinidos en la clase derivada, además éstas podrán definir nuevas variables y métodos. Para entendernos podríamos decir que la **subclase** (clase derivada) "contiene" un objeto de la **superclase** (clase de la que se deriva o clase padre); en realidad lo "amplía" con nuevas variables y métodos.

Más info:

Java no permite que una clase herede de varias a la vez, esto se llama herencia múltiple. Por supuesto, varias clases pueden heredar de una misma superclase.

Todas las clases de Java tienen una superclase común, es la clase **java.lang.Object**. Cuando no se indica explícitamente una superclase de la que se hereda (utilizando extends), la clase heredará de **java.lang.Object**.

La clase Object es la raíz de toda la jerarquía de clases en Java.Todas las clases de Java derivan de Object.

En la clase Object hay definido una serie de métodos que serán heredados por todas las clases que haya creadas o que creemos. Estos métodos son muy útiles algunos de ellos son: **clone()**, **equals()**, **toString()**, **finalize()**, etc.

Pantalla 18 Herencia (ii).

Desarrollo

Una clase puede **reescribir** cualquiera de los métodos que hereda de su clase padre, excepto los declarado como **final**. Cuando declaramos que un método es **final** estamos indicando que no queremos que ninguna posible subclase lo reescriba. Como ejemplo:

```
public class Circulo {
        ...
   public final double area(){ ...}
        ...
}
```

Ninguna clase que herede de Circulo podrá reescribir el método area().

No podemos reecribirlo pero si que podemos acceder mediante la palabra reservada **super.**

Mediante **super** podemos hacer referencia de forma explícita a métodos que están en la clase padre.

Los métodos redefinidos pueden ampliar los **permisos de acceso** pero nunca restringirlos.

Los métodos **static** no pueden ser redefinidos en las clases derivadas.

Ejemplo

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

   public Punto(int x,int y){
       this.x = x;
       this.y = y;
   }

   public void trasladar(int ix,int iy){
        x += ix;
        y += iy;
   }
}

public class Punto3D{
   public int z;
   public Punto3D(int x, int y, int z){
     super(x,y);
     this.z= z;
   }

   public void trasladar(int ix,int iy,int iz){ //redefinimos el método
        super.transladar(ix,iy); //hacemos una llamada al método del padre,
actualizando x e y
        z += iz;
   }
```

}

Más info:

Java no permite que una clase herede de varias a la vez, esto se llama herencia múltiple.

Por supuesto, varias clases pueden heredar de una misma superclase.

Pantalla 19. Herencia (iii). Constructores

Constructores

En las clases heredadas o subclases los constructores se crean igual que en todas las clases pero hay que tener en cuenta que:

- Java, en todo constructor de una subclase introduce una llamada al constructor por defecto de la clase padre. Es decir, realiza una llamada como esta **super()**. Es necesario tener cuidado con esto. Si la superclase no tiene constructor por defecto se producirá un error.
- Podemos indicar una llamada a un constructor de la clase explícitamente. En el ejemplo anterior : super(x,y). Realizamos una llamada al constructor de la clase padre que tiene 2 parámetros enteros. En este caso, Java ya no introduce la llamada al constructor por defecto de la superclase.
- Una llamada a un constructor de la clase padre (super(...)) tiene que ser la primera instrucción del constructor de una clase hija, al igual que hacíamos con this(...).

Estas llamadas de los constructores de la clase padre son fundamentales para inicializar variables miembro de la clase padre a las que no tiene acceso la clase hija, y que es fundamental de inicializar para un correcto funcionamiento de la clase.

Ejemplo

Continuando con el ejemplo Punto y Punto3D. Prestemos atención al constructor de la clase Punto3D. En éste hacemos una llamada al constructor de la clase padre (**super(x,y)**), esto provoca que se ejecute el constuctor **Punto(x,y)**. Con ello conseguimos que el objeto, de la clase Punto3D, que estamos creando inicialice las variables x e y. La siguiente sentencia del constructor inicializa la variable **z** que sólo está definida en la clase Punto.

Además, en la clase Punto3D redefinimos el método trasladar. Este método realiza una llamada a la versión de trasladar que se encuentra en el padre mediante la instrucción **super.trasladar(x,y)**. Con esto trasladamos las coordenadas x e y. Con la siguiente instruccion, **z= + iz**, trasladamos la z.

Pantalla 20. Herencias (iv). Clases abstractas y Clases finales

Clases y métodos abstractos

Una clase abstracta es una clase de la que no se puede crear objetos. La utilidad de estas clases estriba en que otras clases hereden de ésta, por lo que con ello conseguiremos reutilizar código. Para declarar una clase como abstracta utilizamos la palabra clave **abstract.**

```
abstract class NombreClase{
...
}
```

Por ejemplo:

```
public abstract class Figura{
....
}
```

Una clase declarada como **abstract** puede tener métodos **abstract**, en este caso no se les dota de implementación, dejamos el cuerpo del método vacio. Por ejemplo:

```
public abstract class Figura{
...
   public abstract double area(); / / no definimos el cuerpo del método
}
```

Todas las subclases que hereden de una clase abstracta tendrán que redefinir los métodos abstractos dándoles una implementación. En el caso de que no implementen alguno de esos métodos, la clase hija también será abstracta y tendrá que declararse como tal.

Si una clase tiene un método **abstract** es obligatorio que la clase sea abstract.

Una clase abstracta puede tener métodos que no son abstractos, las clases que heredan de ésta podrán hacer uso de estos métodos.

Clases y métodos finales

Una clase declarada como final no puede tener subclases y un método declarado como final no puede ser redefinido por una subclase.

```
public final class Circulo{
...
   public final double area(){...}
...
}
```

_____-

Más info:

Puesto que un método static no puede ser redefinido, un método abstract no pude ser static.

Pantalla 21. Excepciones (i)

Definición

Las excepciones son aquellas situaciones donde se produce un error en el programa, ya sea debido a un error del programador o a un error provocado por cualquier otra situación, por ejemplo que el programa intente de imprimir y la impresora esté desconectada.

Desarrollo

Para capturar una excepción tenemos que situar el código donde ésta se puede producir dentro de un bloque **try**. Asociado a un bloque try siempre hay, al menos, un bloque **catch**.

El código dentro del bloque try está "controlado": Si se produce una situación de error y se lanza como secuencia una excepción, el control pasa al bloque **catch** que es el que se encarga de decidir qué hacer ante la situación de error. Se pueden incluir tantos bloques catch como deseemos, cada uno de los cuales tratará un tipo de excepción.

Además podemos incluir un bloque **finally**. Los bloques finally pueden aparecer o no. **El bloque finally siempre se ejecuta** (se produzca error o no) y se suele utilizar para cerrar conexiones a bases de datos abiertas, cerrar ficheros abiertos, etc.

Por lo tanto la estructura sería la siguiente:

```
try{
  //código que puede producir alguna excepción
}
catch(Excepcion1 e1){
 //código de tratamiento para la excepción Exception1
}
catch(Excepcion2 e2){
 //código de tratamiento para la excepción Exception2
}
catch(ExcepcionN eN){
 //código de tratamiento para la excepción ExceptionN
}
finally{
  //  código que se ejecuta siempre, se produzca o no error
}
```

Pantalla 22. Excepciones (ii). Lanzar excepciones

Desarrollo

En el caso en que el código de un método pueda generar una excepción y no se desee solucionar el problema en el propio método, se tiene que indicar que el método puede producir excepciones. Para ello se utiliza la palabra reservada a **throws**.

En la cabecera de definición del método, justo después de la declaración de parámetros se sitúa la cláusula throws y tantos nombres de clases de excepciones separadas por comas.

```
public class BDConexion{
  public void consulta(int arg1,String arg2) throws IOException,SQLException{
    //este método puede producir dos excepciones.
    //Una IOException (error de entrada y salida)
    //y una SQLException (error al efectuar una operación en la BD)
  }
...

}
```

Algunas aclaraciones

Como consecuencia de ésto, el método desde el que ha sido llamado el método que puede producir excepciones deberá incluir bloques **try/ catch** para controlar las excepciones. Éste método también puede tomar la decisión de no capturar las excepciones y por lo tanto declarar otra cláusula throws con las excepciones que se puedan producir.

Mas info:

Las métodos pueden ir pasando las excepciones de un método a otro, mediante la cláusula throws hasta llegar al último método del programa, el de nivel más superior, el método **main()**. El método main() también podría decidir no capturar la excepción y entonces sería la Máquina Virtual de Java la que capturaría la excepción, mostraría el mensaje y acabaría la ejecución del programa.

Pantalla 23. Excepciones (iii)

Desarrollo

También es posible que seamos nosotros mismos los que queremos lanzar la excepción, es decir, llamarla mediante código. Para ello tendremos que crear un objeto de la clase a la que pertenece la excepción que queremos lanzar.

Una vez creado el objeto sólo tenemos que lanzar la excepción, para ello utilizamos la cláusula **throw**. Nótese que este throw no lleva s.

Algunas Aclaraciones

La clase Exception posee algunos métodos interesantes que nos aportan información del error que se produce:

- **getMessage()**: Devuelve el String dado en el constructor.
- **toString()**. Convierte la excepción en una cadena.
- **printStackTrace()**. Visualiza la pila de llamadas a métodos.

Hablar con Rafa para fusionar estos dos ejemplos en uno solo.

Ejemplo

```
public class UsoBD{
  public void listado() throws IOException,SQLException{
    //este método puede producir dos excepciones.
    //Una IOException (error de entrada y salida)
    //y una SQLException (error al efectuar una operación en la BD)
    BDConexion bd = new BDConexion();
    if (bd==null) throw new SQLException("Conexión fallida");

    bd.consulta(2,"Nombre");
    bd.mostrarDatos();
  }
...

}
```

Ejemplo:

```
public class UsoBD{
  public void listado(){
    BDConexion bd = new BDConexion();
    try{
       bd.consulta(2,"Nombre");
       bd.mostrarDatos();
    }
    catch (IOException e){
       //Tratamiento de la excepcion IOException.
      //e contiene el objeto IOException con la información
```

```
        //de la excepcion que se ha producido
        System.out.println(e.toString());
      }
      catch (SQLException e){
         //Tratamiento de la excepcion SQLException.
        System.out.println(e.getMessage());
        e.printStackTrace();
      }
      finally{
          //código que se ejecuta se produzca o no error
         bd.cerrarConexion();
      }

   }
...

}
```

Y recuerda que en el método anterior podíamos tomar la decisión de no tratar el error y dejar que el método que llame al método **listado** sea el que trate la excepción, para ello en la definición del método listado debíamos de poner la cláusula throws.

Mas info:

También es posible crear nuestras propias excepciones para ello tenemos que crear una clase que herede de la clase **Exception**.

Pantalla 24. Canales de Entrada/Salida

Introducción

Java abstrae una operación de entrada y salida mediante un concepto que se denomina **canal** o **stream**. Accederemos a un fichero a través de un canal, accederemos a la red a través de un canal, etc.

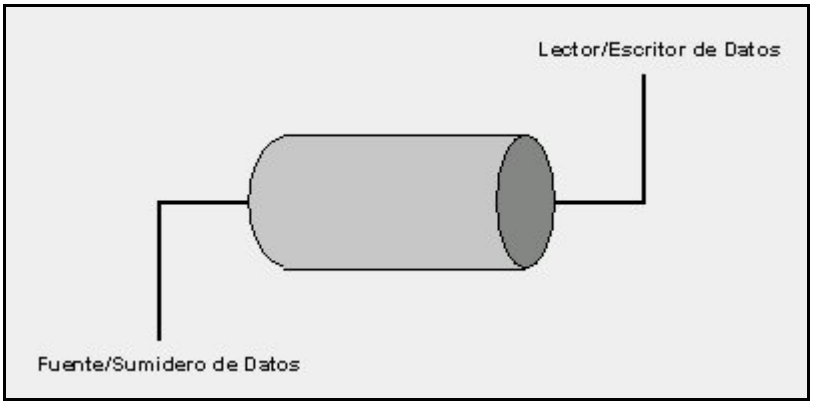
Dado que la **entrada/salida** depende de la plataforma en que Java se ejecute, el lenguaje Java no posee operaciones de E/S. Java posee, en su lugar, una potente **librería**. El package encargado de controlar las E/S en Java es **java.io**. Este package nos proporciona un numeroso grupo de clases para el manejo de las entradas y salidas desde distintos dispositivos.

Definición

Podemos considerar un **canal** como un flujo de datos del cual podemos **leer o escribir**, independientemente del dispositivo de entrada/salida que estemos usando (teclado, disco, internet,...); todas las operaciones de E/S en Java están basadas en los canales. Para poder leer o escribir datos de un dispositivo, el primer paso es crear un flujo o canal; una vez creado, leer o escribir datos es tan fácil como usar los métodos adecuados.

Desarrollo

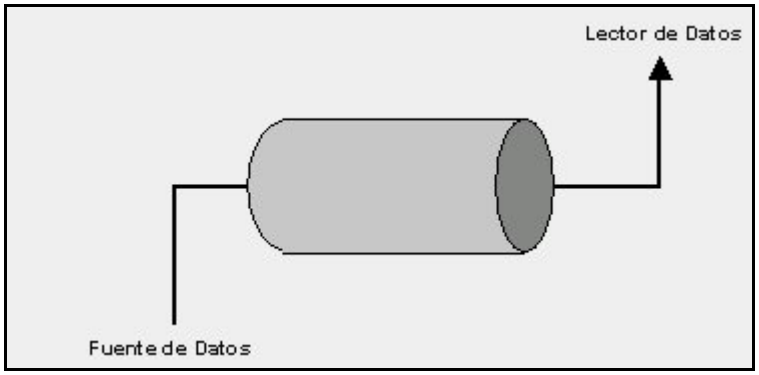
De una forma gráfica podríamos ver un canal como una **tubería** que tiene dos extremos. Uno de ellos está conectado a una fuente/sumidero de datos (dependiendo si el canal es de lectura o escritura) y el otro extremo se sitúa un lector/escritor. El siguiente gráfico ilustra lo anteriormente dicho.



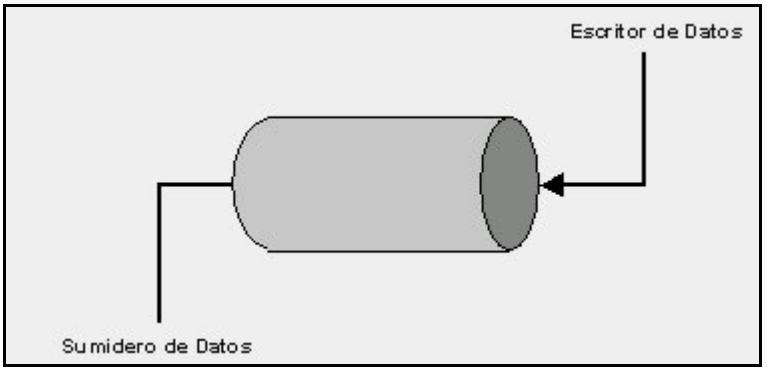
Pantalla 25 Tipos de Canales

Básicamente podemos considerar dos tipos de canales: **canales de entrada** y **canales de salida**.

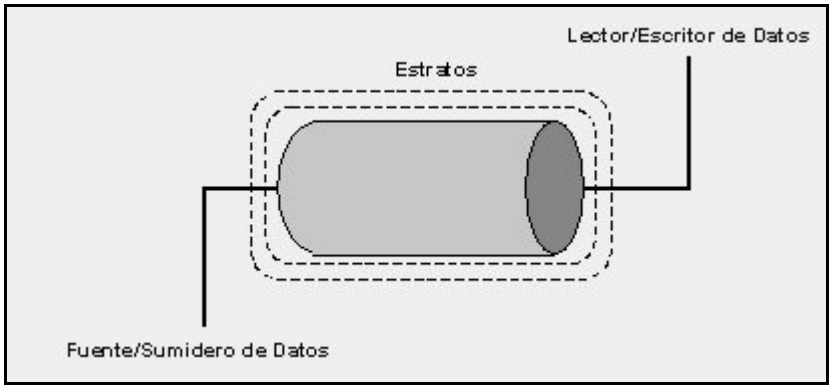
? Los **canales de entrada** son aquellos en el que uno de sus extremos está conectado una fuente de datos, que puede ser un fichero, una conexión a Internet, etc. De este tipo de canales leeremos datos.



? Los **canales de salida** son aquellos en el que uno de sus extremos está conectado un sumidero de datos (donde se van a almacenar datos), que puede ser un fichero, una conexion a Internet, etc. Sobre este tipo de canales escribiremos datos.



Como complemento a los canales, Java posee otras clases que dan funcionalidades (características) a los canales. Estas funcionalidades pueden ser que sea posible la lectura de datos primitivos, lectura de lineas de textos, que los datos que escriba sobre un canal se compriman en formato zip, etc. A estas clases se les llama **estratos**.



Pantalla 26. Canales de entrada

La clase **java.io.InputStream** sirve de clase base para todas las clases que modelan los canales de entrada. Los métodos descritos por esta clase leen correctamente los datos independientemente del dispositivo de lectura.

Es una clase abstracta, por lo que no podremos crear un objeto de esta clase directamente, sino crear objetos de sus clases derivadas (hijas). Estas clases heredaran de las clases padres los métodos de E/S y incorporaran otros más sofisticados, en función del dispositivo a usar y de como se quiera recibir los datos.

Por cada **tipo de fuente de datos** se tiene una subclase de **InputStream**.

Las fuentes y subclases son:

- Un array de bytes (**ByteArrayInputStream**)
- Un String (**StringBufferInputStream**)
- Un fichero en disco (**FileInputStream**)
- Otro stream (**PipedInputStream**)
- Una secuencia de canales (**SequenceInputStream**)
- Una conexión con Internet

Hay que decir que los métodos de la clase **InputStream** son bloqueantes, esto quiere decir que no se salta a la siguiente instrucción hasta que la operación de lectura ha sido realizada.

Más info:

Algunos métodos interesantes de la clase **InputStream**:

MÉTODO	DESCRIPCIÓN
public int available() throws IOException	Retorna el numero de bytes que pueden ser leídos de este flujo sin bloquear.
public void close() throws IOException	Cierra el canal.
public synchronized void mark(int readlimit)	Marca la posición en curso en el flujo.Una llamada posterior a reset() hace volver a la posición previa.
public boolean markSupported()	Retorna true si se puede utilizar el método mark(int) en este flujo.
public int read() throw IOException	Lee el siguiente byte del canal; devuelve -1 si no hay más datos.
public synchronized void reset() throws IOException	La posición de lectura se establece en la posición indicada en mark(int).
public long skip(long n) throws IOException	Descarta del canal n bytes. Retorna el numero de bytes descartados.

Pantalla 27. Canales de Salida

La clase **java.io.OutputStream** es la encargada de definir los métodos de escritura de datos, independientemente del dispositivo sobre el que se trabaje.

Es una clase abstracta, por lo que no podremos crear un objeto de esta clase directamente, sino crear objetos de sus clases derivadas (hijas). Estas clases heredaran de las clases padres los métodos de E/S y incorporaran otros más sofisticados, en función del dispositivo a usar y de como se quiera recibir los datos.

Por cada tipo de **destino** de datos se tiene una subclase de OutputStream. Los destinos y subclases son:

- Un array de bytes (**ByteArrayOutputStream**)
- Un fichero en disco (**FileOutputStream**)
- Otro canal (**PipedOutputStream**)

Hay que decir que los **métodos** de la clase OutputStream son **bloqueantes**, esto quiere decir que no se salta a la siguiente instrucción hasta que la operación de escritura ha sido realizada.

Más info:

Algunos métodos interesantes de la clase **OutputStream**:

MÉTODO	DESCRIPCIÓN
public abstract void write(int b) throws IOException	Introduce el numero debytes que pueden ser escritos de este flujo sin bloquear.
public void write(byte b[]) throws IOException	Escribe el array de bytesen el flujo.
public void write(byte b[],int off,int len) throws IOException	Escribe el array en el flujo desde la posicion indicada en off y el numero de bytes indicado en len.
public void flush( ) throws IOException	Fuerza la escritura de posibles datos que todavia permanecen en el buffer de E/S.
public void close() throws IOException	Cierra el flujo.

Pantalla 28. Estratificación

Java suministra una serie de clases que proporcionan distintas funcionalidades a los canales de entrada y de salida. Se pueden colocar en capas para suministrar más de una funcionalidad a la vez.

Algunos estratos para los canales de lectura:

- **DataInputStream**: Prepara al canal de entrada para recoger valores primitivos (int,char,long,etc.).
- **BufferedInputStream**: Evita que en cada operación de lectura se efectúe una lectura física. Crea un buffer interno.
- **LineNumberInputStream**: Lleva la cuenta del número de líneas que ha sido leídas.
- **PushbackInputStream**: Permite reinsertar caracteres leídos en el canal de entrada.

Algunos estratos para los canales de salida:

- **DataOutputStream**: Prepara al canal de salida para recoger valores primitivos (int,char,long,etc.).
- **PrintStream**: Para producir salida inteligible (un fichero de texto).
- **BufferedOutputStream**: Para que todas las opereaciones utilicen un buffer de memoria.

Ejemplos Veamos ejemplos de uso de canales, tanto de entrada como de salida, y estratos.

1.- Fichero de entrada con buffer.

```
try{

        DataInputStream in=
                new DataInputStream(
                        new BufferedInputStream(
                                new FileInputStream(new
String("MiFichero.txt"))));

        String s,s2 = new String();
        while ((s= in.readLine()!= null)
                        s2+ = s + "\ n";

        in.close();
} catch (FileNotFoundException e){
        System.out.println("Fichero no encontrado");
} catch(IOException e){
        System.out.println("Error E/ S");
}
```

Veamos el código en detalle:

```
DataInputStream in= new DataInputStream(
                        new BufferedInputStream(
                            new FileInputStream(
                                    new String("MiFichero.txt"))));
```

Creamos un objeto **FileInputStream** para abrir un fichero. Deseamos que el canal tenga funcionalidad de **leer mediante un buffer**, para ello creamos un objeto **BufferedInputStream** a partir del canal anterior. Obtenemos el canal con la nueva funcionalidad. Igualmente deseamos que el canal pueda **leer datos primitivos** para ello le aplicamos el estrato **DataInputStream**. Comenzamos a leer el fichero mediante el método **readLine()** que lee una línea del canal. Se llega al final del fichero cuando obtenemos null.

```
while ((s= in.readLine()!= null)
        s2+= s + "\n";
```

Controlamos las posibles excepciones que se puedan producir, como que no se encuentre el fichero (**FileNotFoundException**) o se produzca un error de lectura (**IOException**).

2.- Entrada desde una cadena.

```
// suponemos declarado un String s2 con una cadena
try{

  StringBufferInputStream in2=
                    new StringBufferInputStream(s2);
  int c;

  while ((c= in2.read())!= -1)
                    System.out.print((char)c);

}catch(IOException e){
          System.out.println("Error E/S");
}
```

En este caso, vamos leyendo carácter a carácter desde una cadena de caracteres. **read()** devuelve cada byte como un entero, luego hay que convertirlo a **char**.

La propia fuente establece un buffer, luego no es necesario acoplarle ningún estrato más.

Y por supuesto, hay que contolar las excepciones.

3.- Lectura de datos primitivos.

```
try{

  DataInputStream in3 =
      new DataInputStream(
         new StringBufferInputStream(s2));

  while(true)
            System.out.print((char)in3.readByte());

}catch(EOFException e){
          System.out.println("Fin del canal");
}catch(IOException e){
    System.out.println("Error E/S");
}
```

A la lectura de una cadena se le ha añadido la posibilidad de leer datos primitivos, aunque en este caso, la lectura se ha efectuado en bytes que posteriormente se han convertido en caracteres.

Leyendo bytes, incluso el carácter **<EOF>** se considera un carácter, por lo que no podemos saber cuando llegamos al final del fichero. Se detecta cuando se produce la excepción **EOFException**. Pero podemos utilizar el método **available()** que devuelve el número de bytes que quedan por leer.

```
try{

  DataInputStream in3 =
      new DataInputStream(
         new StringBufferInputStream(s2));

  while(in3.available()!=0)
            System.out.print((char)in3.readByte());

}catch(IOException e){
    System.out.println("Error E/S");
}
```

4.- Escritura en fichero de texto

```
//suponemos declarado un String s2 con una cadena
try{

  LineNumberInputStream li =
      new LineNumberInputStream(
         new StringBufferInputStream(s2));

  DataInputStream in4 =
      new DataInputStream(li);

   PrintStream out1 =
        new PrintStream(
          new BufferedOutputStream(
               new FileOutputStream("MiFichero.out")));

   while((s=in4.readLine())!=null)
          out1.println("Línea "+li.getLineNumber() + s);

out1.close();

}catch(EOFException e){
    System.out.println("Fin del canal");
}
```

Este ejemplo hace uso de **LineNumberInputStream**. Para obtener el número de línea que se está leyendo. Para poder ejecutar el método **getLineNumber()** necesitamos un objeto **LineNumberInputStream**. Pero puesto que nosotros deseamos poder leer datos primitivos necesitamos construir el **DataInputStream** en dos pasos. Pero tanto el objeto

LineNumberInputStream como el **DataInputStream** se construyen sobre el mismo canal. Son dos estratos sobre el mismo canal.

Lo que vamos leyendo lo almacenamos un el fichero **MiFichero.out.** Al canal sobre el fichero le damos la funcionalidad de tener un buffer y que la escritura sea en formato texto (**PrintStream**). Si no creamos este estrato el fichero sería binario.

5.- Guardar y recuperar datos

```
try{

  DataOutputStream out2 =
      new BufferedOutputStream(
         new FileOutputStream("Data.bin"));

  out2.writeBytes("El valor de pi: \n");
  out2.writeDouble(3.14159);

  out2.close();

  DataInputStream in5 =
      new DataInputStream(
          new BufferedInputStream(
              new FileInputStream("Data.bin")));

  System.out.println(in5.readLine());
  System.out.println(in5.readDouble());

}catch(EOFException e){
    System.out.println("Fin del canal");
}
```

Utilizamos un **DataOutputStream** para guardar datos en formato binario, y **DataInputStream** para recuperarlos. Puesto que estamos leyendo ficheros binarios es necesario saber exactamente qué tipos de datos se han guardado, y en qué orden.

Uso de entrada y salida estándar

Entrada y salida estándar

El siguiente ejemplo ilustra cómo podemos manejar la entrada estándar para realizar un programa que nos cuenta el numero de caracteres introducidos hasta un retorno de carro ("Enter").

```
// miType.java

import java.io.*;
public class miType {

        public static void main( String args[] ) throws IOException{

                int c;
```

```
                        int contador = 0;

                        while( (c = System.in.read() ) != '\n' ) {

                                contador++;
                                System.out.print( (char)c );

                        }
                        System.out.println(); // Línea en blanco
                        System.err.println( "Contados "+ contador +" bytes en
total." );

        }

}
```

Veamos otro ejemplo que nos muestra como leer distintos tipos de datos de la entrada estándar.

```
import java.io.*;

public class LecturaE{

        public static void main(String args[]){
                try{
                        DataInputStream d = new DataInputStream(System.in);
                        System.out.println("Inserta una cadena, un caracter, " +

"un entero, otro carácter y un float.");

                        String s = d.readLine();
                        char c = d.readLine().charAt(0);
                        int i = Integer.parseInt(d.readLine(),10);
                        char c2 = (char)(d.readByte());
                        float f = (new Float(d.readLine())).floatValue();

        System.out.println(s+"\n"+i+"\n"+c+"\n"+c2+"\n"+f+"\n");
                }
                catch (IOException e){
                        System.out.println("Error");
                }
        }
}
```

System.in permite leer sólo entrada binaria y caracteres. Sin embargo todas las entradas desde el teclado son secuencias de caracteres, consecuencia de las teclas pulsadas.

Para leer correctamente es necesario leer una tira de caracteres y convertirla al formato que nos interese. En este ejemplo es muy interesante ver cómo se realiza la conversión a **int** mediante el método **parseInt** de la clase **Integer**.

Pantalla 29. String y Stringbuffer

El paquete **java.lang** (paquete que se carga por defecto) posee dos tipos de cadenas bien diferenciadas. Por un lado, la clase **String** para trabajar con cadenas alfanuméricas que mantienen un valor constante. Su tamaño no cambia una vez que han sido creadas. Y la clase **StringBuffer**, usada para manipular cadenas alfanuméricas que pueden cambiar su tamaño.

Manejo de String

Las cadenas de la clase String son más eficientes para el sistema, al ser estos objetos constantes. La mayoría de funciones relacionadas con el tratamiento de cadenas esperan como argumento valores de la clase String, y devuelven valores pertenecientes a ella. Las funciones estáticas no consumen memoria del objeto, por lo que su uso es más conveniente.

String se utiliza cuando no se precisa modificar el valor de una cadena, por ejemplo: si implementamos un método que necesite una cadena de caracteres y el método no va a modificar la cadena.

Ejemplo:

En el siguiente ejemplo el método **reverseIt()** usa las clases **String** y **StringBuffer** para invertir el orden de una cadena alfanumérica.

```
public class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);
        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El método **reverseIt** admite el argumento **source** de la clase **String** indicándole así la cadena que se desea invertir. El método, usa dest (de la clase **StringBuffer** y de igual tamaño que source) para realizar la inversión de la cadena mediante un bucle. Finalmente, convierte **dest** de **StringBuffer** a **String** (mediante el método **toString()**) devolviendo esta última.

Use la clase **String** para enviar valores y devolver estos en métodos. Tal y como se muestra en el ejemplo de código anterior. La clase **StringBuffer** es usada para la construcción de datos, valores a devolver más tarde como **String**.

El uso apropiado de cada clase es importante, ya que inciden en el rendimiento de la aplicación.

Manejo de StringBuffer

Existen varios métodos **append()** para añadir varios tipos, como float, int, boolean, e incluso objetos, al final del StringBuffer. El dato es convertido a cadena antes de que tenga lugar la operación de adicción.

En ocasiones podríamos querer insertar caracteres en medio de un **StringBuffer**. Para ello podemos hacer uso del **insert()**.

```
StringBuffer sb = new StringBuffer("Bebe Caliente!");
sb.insert(6,"Java ");
System.out.println(sb.toString());
```

"Java " tiene que insertarse antes de la 'C' de "Caliente". El índice de la 'C' es el 6 (los índices empiezan en 0).

Algunas aclaraciones:

Podemos crear un objeto de la clase String de dos formas: mediante una **cadena entre comillas** dobles o creando un objeto con el operador **new**. Cuando se inicia el proceso de compilación, las cadenas entrecomilladas pasan a ser un objeto String, el propio compilador las traduce.

Más info:

Métodos de interés

length(): Usado para obtener la longitud de una cadena.
charAt(int): Devuelve el carácter que está situado en la posición indicada como argumento.
indexOf(): busca un caracter en una cadena empezando desde el principio de la cadena.
lastIndexOf(): busca un caracter en una cadena empezando desde el final de la cadena.

Veamos un ejemplo:

```
public  class NombreFichero {
    String path;
    char separador;

    NombreFichero(String str, char sep) {
       path = str;
       separador = sep;
    }

    String extension() {
       int punto = path.lastIndexOf('.');
       return path.substring(punto + 1);
    }

    String NombreFichero() {
       int punto = path.lastIndexOf('.');
       int sep = path.lastIndexOf(separador);
       return path.substring(sep + 1, punto);
    }

    String path() {
       int sep = path.lastIndexOf(separador);
       return path.substring(0, sep);
    }
 }
```

Una clase que sirva para probar la anterior podría ser:

```
public class NombreFicheroTest {
   public static void main(String[] args) {
      NombreFichero miHomePage = new
NombreFichero("/home/index.html", '/');
      System.out.println("Extension = " +
miHomePage.extension());
      System.out.println("Nombre de Fichero = " +
miHomePage.NombreFichero());
      System.out.println("Path = " + miHomePage.path());
   }
}
```

Para localizar el último caracter "." (punto) en el nombre del archivo usaremos **lastIndexOf()** . Si no encontramos el caracter, **lastIndexOf()** devolverá -1 y se producirá un **StringIndexOutOfBoundsException**.

La cadena comienza en la posición 0, luego el argumento para **substring()** será, la posición encontrada más una posición, en el código: **dot + 1** . La clase **String** admite distintas versiones para los métodos **indexOf()** y **lastIndexOf()**.

Pantalla 30. Conversion de objetos a String

En ocasiones es necesario convertir un objeto a un **String**. Por ejemplo, **System.out.println()** no acepta StringBuffers, luego será necesario convertir el StringBuffer a String para hacer uso del métoso System.out.println(). Cuando el compilador necesita convertir un objeto a cadena utiliza el método toString() que tenga definido ese objeto. Si no tiene ninguno utiliza el **toString()** de la clase **Object**.

String proporciona un método **valueOf()** para convertir variables de diferentes tipos a un **String**.

Por ejemplo para imprimir el número pi:

```
System.out.println(String.valueOf(Math.PI));
```

Para convertir una cadena en un número se proporcionan unos métodos de clase llamados **valueOf()** que convierten una cadena en un objeto de ese tipo numérico. Para la clase **Float**, por ejemplo:

```
String piStr = "3.14159"; Float pi = Float.valueOf(piStr);
```

Strings y el compilador Java

Para concatenar en Java se puede utilizar el operador +, por ejemplo:

```
String cat = "cat";
System.out.println("con" + cat + "enacion");
```

Los Strings no pueden modificarse, realmente el compilador utiliza **StringBuffer** para implementar la concatenación.

También se puede utilizar el operador + para añadir valores a una cadena que no son propiamente cadenas:

```
System.out.println("Java's Number " + 1);
```

El compilador convierte el valor 1 (el entero 1) a un objeto String antes de realizar la concatenación. Para ello utiliza el método **toString()** que tenga definido.

Pantalla 31. Manejo de Ficheros

Antes de realizar acciones sobre un fichero, necesitamos un poco de información sobre ese fichero. La clase File proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre esos ficheros:

Para crear un objeto File nuevo, se puede utilizar cualquiera de los 3 constructores siguientes:

```
File miFichero;

1. miFichero = new File( "/etc/kk" );

2. miFichero = new File( "/etc","kk" );

3. File miDirectorio = new File( "/etc" );
   miFichero = new File( miDirectorio,"kk" );
```

El constructor utilizado depende a menudo de otros objetos File necesarios para el acceso. Por ejemplo, si sólo se utiliza un fichero en la aplicación, el primer constructor es el mejor. Si en cambio, se utilizan muchos ficheros desde un mismo directorio, el segundo o tercer constructor serán más cómodos. Y si el directorio o el fichero es una variable, el segundo constructor será el más útil.

Mas info:

Una vez creado un objeto **File**, se puede utilizar uno de los siguientes métodos para reunir información sobre el fichero:

- **Nombres de fichero**
  String getName(): Nombre del fichero.
  String getPath(): Path del fichero.
  String getAbsolutePath(): Path absoluto.
  String getParent(): Path del directorio padre.
  boolean renameTo(File nuevoNombre): Renombrado de un fichero.
- **Comprobaciones**
  boolean exists(): Comprueba que un fichero existe.
  boolean canWrite(): Indica si se puede escribir en el fichero.
  boolean canRead(): Indica si el fichero es de solo lectura.
  boolean isFile(): Comprueba si un File es un fichero y existe.
  booleanisDirectory(): Comprueba si un File es un directorio.
  boolean isAbsolute(): Comprueba si el File es referenciado con un path absoluto.
- **Información general del fichero**
  long lastModified(): Ultima modificación.
  long length(): Tamaño del fichero.
- **Utilidades de directorio**
  boolean mkdir(): Creación de directorio.
  String[] list(): Listado de ficheros.

Ejemplo:

En la aplicación InfoFichero.java mostramos información sobre los ficheros pasados como parámetros en la linea de comandos:

```
import java.io.*;

public class InfoFichero {
    public static void main( String args[] ) throws IOException
     {
        if( args.length > 0 )
         {
          for( int i=0; i < args.length; i++ )
            {
              File f = new File( args[i] );
              System.out.println( "Nombre: "+ f.getName() );
              System.out.println( "Camino: "+ f.getPath() );
              if( f.exists() )
               {
                  System.out.print( "Fichero existente " );
                  System.out.print( (f.canRead() ? " y se puede Leer" : "" ) );
                  System.out.print( (f.canWrite() ? " y se puede Escribir" : "" ) );
                   System.out.println( "." );
                   System.out.println( "La longitud del fichero son "+ f.length()+"
bytes" );
               }
              else
                  System.out.println( "El fichero no existe." );
            }
          }
         else System.out.println( "Debe indicar un fichero." );
       }

  }
```

La ejecución de este programa se haría de la siguiente forma **java InfoFichero fichero1.txt fichero2.txt**

Como se puede observar los nombres de ficheros se recuperan, en el programa Java, mediante el argumento **args** del método main. En cada una de las posiciones del array se encuentra un String con el nombre del fichero.

Pantalla: Resumen

- Java es un lenguaje de programación, al mismo tiempo, **sencillo, potente, seguro, eficaz y universal**. Estas propiedades lo convierten en un instrumento ideal para el desarrollo de todo tipo de aplicaciones.

- Java es un software formado por pequeños **módulos** que pueden proceder de sistemas distintos, de **máquinas diferentes** y que se irán ejecutando según sean necesarios.

- Las aplicaciones creadas con Java podrán **transportarse** de forma transparente a través de la red. Es decir, está diseñado para aprovechar aplicaciones que se van a poder ejecutar en cualquier plataforma de computación que soporte una Máquina Virtual de Java.

  Una **Máquina Virtual de Java** (JVM) es una capa de software que reside o se ejecuta encima del sistema operativo del ordenador, y que permite que la plataforma ejecute Java.

- En la **programación orientada a objetos**, los objetos se pueden relacionar unos con otros permitiéndonos una mayor flexibilidad para incrementar o modificar el programa sin tener que cambiar el código fuente original, dándonos así la capacidad de organizar mejor los programas distribuidos.

- Una vez creado nuestro programa con un editor de texto, será necesario compilarlo mediante el comando **Javac fichero.java**, antes de poder ejecutarlos mediante el comando **Java fichero**.