



# 1. Introducción a la programación

	Autor	Xerach Casanova
	Clase	Programación
	Fecha	@Dec 7, 2020 2:45 PM

1. Introducción

2. Programas y programación

2.1. Buscando una solución

2.2. Algoritmos y programas

3. Paradigmas de la programación

4. Fases de la programación

4.1. Resolución del problema

Análisis

Diseño

4.2 Implementación

Codificación o construcción

Prueba de ejecución y validación

4.3. Explotación

5. Ciclo de vida del software

6. Lenguajes de programación.

6.1. Lenguaje máquina

6.2 Lenguaje ensamblador

6.3 Lenguajes compilados

6.4. Lenguajes interpretados

7. Java

7.1. ¿Qué y como es java?

7.2. Breve historia

7.3 La POO y Java

7.4. Independencia de la plataforma y trabajo en red

7.5. Seguridad y simplicidad.

Seguridad

Simplicidad

7.6 Java y los bytecodes

- 8. Programas java
  - 8.1. Estructura de un programa.
  - 8.2. El entorno básico de desarrollo Java
  - 8.3. La API de Java
  - 8.4. Tipo de aplicaciones Java
    - Aplicaciones de consola
    - Aplicaciones gráficas
    - Applets
    - Servlets
    - Midlets
- 9. Entornos integrados de desarrollor (IDE)
  - 9.1. ¿Qué son?
  - 9.2. IDE's actuales
  - 9.3. Netbeans
- Mapa conceptual

# 1. Introducción

Las acciones de la vida cotidiana están relacionadas con la programación y el tratamiento de la información, de una u otra manera. El volumen de datos que se maneja constituye un vastato territorio en el que los programadores tienen mucho que decir.

# 2. Programas y programación

## 2.1. Buscando una solución

Como en la vida real, la programación se basa en la búsqueda y obtención de soluciones a un problema determinado.

- **Análisis del problema.** Debe ser definido y comprendido claramente para analizarlo con detalle.
- **Diseño o desarrollo de algoritmos:** Se busca una solución sin entrar en detalles tecnológicos.
- **Resolución del algoritmo elegido en la computadora:** Convertimos el algoritmo en programa y al ejecutarlo se comprueba si se soluciona.

La solución al problema debe tener las siguientes virtudes:

- **Corrección y eficacia:** si se resuelve el problema adecuadamente.

- **Eficiencia:** si lo hace en tiempo mínimo y con uso óptimo de recursos.

Para llegar a la solución debemos tener en cuenta los siguientes conceptos.

- **Abstracción.** Se analiza el problema y se descompone en problemas más pequeños y de menor complejidad que se describen de manera precisa.
- **Encapsulación:** Se oculta la información de los diferentes elementos que forman el sistema.
- **Modularidad:** Se divide el proyecto en módulos independientes que tendrán su función correspondiente.

## 2.2. Algoritmos y programas

Un algoritmo es una secuencia ordenada de pasos descrita sin ambigüedades que conducen a la solución de un problema.

Son independientes del lenguaje de programación. Un algoritmo llevado a distintos lenguajes debe dar siempre la misma solución.

Los lenguajes de programación son el medio para expresar un algoritmo. El diseño de los algoritmos parte de la creatividad del desarrollador y de los conocimientos de las técnicas de programación. Un mismo problema puede tener algoritmos distintos igualmente válidos.

Los algoritmos deben cumplir las siguientes características:

- **Ser precisos** e indicar el orden paso a paso.
- **Estar definido.** Al ejecutarlo más de una vez con los mismos datos de entrada se debe obtener el mismo resultado. También debe dar respuesta a cualquier dato de entrada.
- **Ser finito.** Debe terminar.

Técnicas para representar gráficamente algoritmos:

- **Diagramas de flujo.** Símbolos gráficos para su representación. Se utiliza en fase de análisis).
- **Pseudocódigo.** Técnica basada en el uso de palabras clave en lenguaje natural, constantes, variables, otros objetos, instrucciones y estructuras de programación para expresar la solución del problema.
- **Tablas de decisión.** Técnica de apoyo al pseudocódigo en situaciones compleja que se basa en una tabla se representan las posibles condiciones

del problema con sus respectivas acciones.

## 3. Paradigmas de la programación

Es un modelo básico para el diseño y la implementación de programas. Este modelo determina como será el proceso de diseño y la estructura final el programa.

- **Programación declarativa.** Consiste en decirle a un programa lo que tiene que hacer en lugar de decirle como hacerlo. SQL está basado en este paradigma.
- **Programación imperativa.** Se basa en desarrollar algoritmos detallando de forma clara y específica los comandos para llegar a la solución. Se basa en variables, tipos de datos, expresiones y estructuras de control de flujo. Dentro de ella encontramos la programación convencional, estructurada, orientada a objetos, orientada a eventos, orientada a aspectos...

Los lenguajes pueden soportar múltiples paradigmas o estar solo basados en un paradigma.

## 4. Fases de la programación

El proceso de creación de software se divide en tres fases:

- Fase de resolución del problema
- Fase de implementación
- Fase de explotación y mantenimiento.

### 4.1. Resolución del problema

El problema debe ser definido y comprendido para poder analizarse. La fase de resolución pasa por dos etapas.

#### Análisis

En esta fase se dan por conocidas todas las necesidades que precisa la aplicación. Se especifican los procesos, estructuras y datos a emplear.

Este análisis ofrecerá una idea general de lo que se solicita y posteriormente se refinará para dar respuesta a las siguientes cuestiones.

- Cuál es la información que ofrecerá la resolución del problema - Salida de datos.
- Qué datos son necesarios para resolverlo - Entrada de datos.

En esta fase debemos analizar la documentación de la empresa, investigar, observar todo lo que rodea y recompilar información útil.

## Diseño

La fase de análisis se convierte en un diseño más detallado, se indica la secuencia lógica de instrucciones para resolver el problema. En resumen, se construyen algoritmos.

En esta fase la aplicación se plantea como una operación global y se descompone en operaciones sencillas, detalladas y específicas. Estas operaciones se asignan a módulos separados.

Antes de pasar a la siguiente fase debemos realizar prueba o traza del programa utilizando datos concretos. Si tiene errores debemos volver a la fase de análisis.

## 4.2 Implementación

Es la fase donde se lleva a la realidad nuestro algoritmo. Esto implica varias etapas, que son:

### Codificación o construcción

Traducimos los algoritmos a un determinado lenguaje de programación. Para comprobar la calidad y estabilidad se realizan pruebas que comprueban las funciones de cada módulo (pruebas unitarias), que los módulos funcionan entre ellos (pruebas de interconexión) y que todos funcionan bien en conjunto (pruebas de integración).

Seguidamente se compila. La compilación es el proceso en el que se traducen las instrucciones escritas en lenguaje de programación en lenguaje máquina (código binario). Y se lleva a cabo de dos formas:

- **A través de un compilador.** Programa que traduce completamente el código realizando un análisis lexicográfico, semántico y sintáctico, genera un código intermedio y finalmente genera el código máquina ejecutable.
- **A través de un intérprete.** Programa capaz de analizar y ejecutar programas escritos en lenguajes de alto nivel. Los intérpretes realizan la

traducción del programa a medida que sea necesaria. No se guarda el resultado de la traducción.

Una vez traducido el programa puede ser ejecutado.

## Prueba de ejecución y validación

Se implanta la aplicación en el sistema donde va a funcionar, se pone en marcha y se comprueba su funcionamiento. Se analiza si responde a los requerimientos especificados, si se detectan nuevos errores y si la interfaz es amigable.

No se puede avanzar mientras se detectan errores. El testeado se documenta mediante:

- **Documentación interna:** encabezados, descripciones, declaraciones del problema y comentarios que se incluyen en el código fuente.
- **Documentación externa:** manuales del programa.

## 4.3. Explotación

La fase de explotación es aquella en la que ya está siendo de utilidad para los usuarios. En ella se realizan evaluaciones periódicas, se llevan a cabo modificaciones e incluso se corrigen errores no detectados anteriormente. Para ello debemos tener a mano una documentación adecuada que facilite la comprensión y el uso del programa.

# 5. Ciclo de vida del software

Es la sucesión de estados o fases por las cuales pasa un software a lo largo de su vida:

- Especificación y análisis de requisitos.
- Diseño.
- Codificación.
- Compilación.
- Pruebas.
- Instalación y explotación.
- Mantenimiento.

# 6. Lenguajes de programación.

Un lenguaje de programación es un conjunto de reglas sintácticas, semánticas, símbolos y palabras especiales establecidas para la construcción de programas. Es un lenguaje artificial, una construcción mental del ser humano para expresar programas.

La gramática del lenguaje son reglas aplicables al conjunto de símbolos y palabras para la construcción de sentencias correctas. Dispone de:

- **Léxico:** Vocabulario del lenguaje.
- **Sintaxis:** Combinación de símbolos y palabras especiales.
- **Semántica:** es el significado de cada construcción del lenguaje.

En función de lo cerca que esté el lenguaje de programación del lenguaje humano o del ordenador se clasifican en distintos niveles.

## 6.1. Lenguaje máquina

O código binario. Es el lenguaje utilizado directamente por el procesador y el que puede interpretar un circuito microprogramable.

Fue el primer lenguaje utilizado de programación pero presenta los siguientes inconvenientes:

- Cada programa era válido solo para un tipo de procesador u ordenador.
- Lectura e interpretación extremadamente difícil, modificaciones costosas.
- Los programadores de la época debían memorizar largas combinaciones de ceros y unos.
- Los programadores se encargaban de introducir los códigos binarios en el ordenador.

## 6.2 Lenguaje ensamblador

Es la evolución directa del lenguaje máquina. Las secuencias binarias se sustituyen por códigos de operación elementales llamados mnemotécnicos.

También presenta múltiples dificultades:

- Los programas siguen dependiendo del hardware que los soporta.

- Los programadores debían conocer detalladamente la máquina en la que programan para utilizar sus recursos.
- La lectura, interpretación y modificación seguía siendo difícil.

El lenguaje ensamblador necesita un intermediario que traduzca las instrucciones en lenguaje máquina.

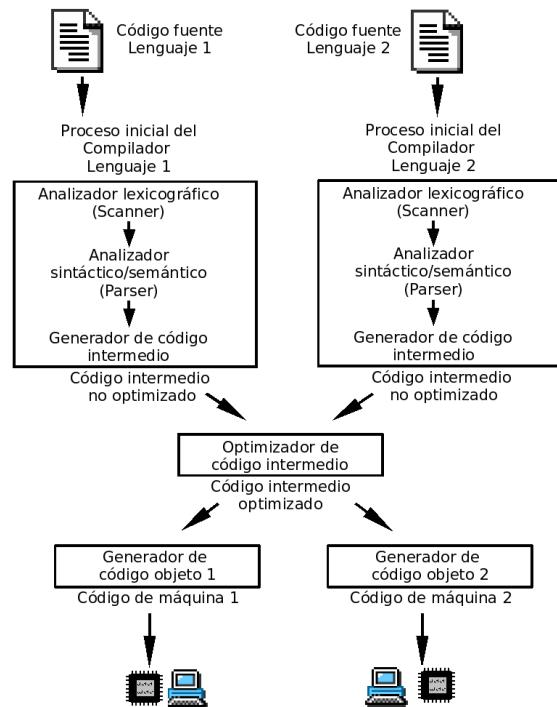
## 6.3 Lenguajes compilados

Nacen para resolver los problemas derivados del uso del lenguaje ensamblador y acercar la programación al humano. Se denominan lenguajes de alto nivel y sus ventajas son:

- Fáciles de aprender.
- Reducción de tiempo y costes para desarrollar software.
- Independientes del hardware.
- La lectura, interpretación y modificación resulta más sencilla.

Es menos eficiente que el código generado en lenguaje ensamblador y además debe traducirse en código máquina. Este trabajo lo hace el compilador.

El código objeto es el código máquina, el cual necesita añadir módulos de enlace o bibliotecas para obtener el código ejecutable.



## 6.4. Lenguajes interpretados

La ejecución se realiza a través de un intérprete. Cada instrucción se analiza, se traduce y se ejecuta al ser interpretada pero no se guarda en memoria.

El proceso de traducción y ejecución se llevan a cabo de manera simultánea. Presentan el inconveniente de ser algo más lentos y necesitan del programa intérprete ejecutándose.

A medio camino están los pseudo-compilados o pseudo-interpretados como Java, el cual se compila obteniendo el código binario en forma de bytecodes, estructuras parecidas al lenguaje máquina. Estos ficheros se encarga de interpretarlos la máquina virtual, la cual lo traducirá en lenguaje máquina.

## 7. Java

### 7.1. ¿Qué y como es java?

Es un lenguaje sencillo de aprender.

Su gran virtud es que es válido para cualquier plataforma ya que el código será ejecutado en la máquina virtual de java JVM, el cual interpreta el código convirtiéndolo en código específico de la plataforma que lo soporta: Write once, run everywhere.

Características del lenguaje java:

- El código generado por el compilador es independiente de la arquitectura.
- Orientado a objetos con gran cantidad de bibliotecas definidas.
- Sintaxis similar a C y C++
- Es distribuido, preparado para TCP/IP.
- Robusto, realiza comprobaciones de código en tiempo de compilación y ejecución.
- Seguridad garantizada ya que java no accede a zonas delicadas de memoria o de sistema.

La programación orientada a objetos nace para resolver los problemas que daba la programación estructurada, aunque estuviese dividida en programación modular. Los problemas se dividen en objetos que tienen propiedades e interactúan con otros objetos. El programador se centra en cada objeto para programar sus elementos y funciones.

### 7.2. Breve historia

Java surgió en 1991 por un grupo de

ingenieros de Sun Microsystems para programar pequeños dispositivos electrónicos.

En 1995 pasa a llamarse Java. El factor determinante de su éxito fue la incorporación del intérprete java en la versión 2.0.

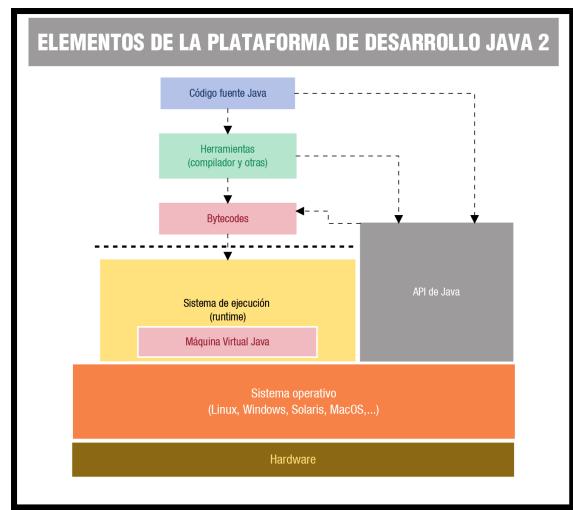
Para el desarrollo de programas es necesario utilizar el entorno JDK (Java Development Kit) el cual trae un compilador y un entorno de ejecución JRE (Java Run Environment) para los bytecodes generados.

Java 2 es la tercera versión del lenguaje e incluye:

- Lenguaje de programación Java
- Conjunto de bibliotecas (Java Core).
- Conjunto de herramientas para desarrollo de programas: compilador, generador de documentación, depurador...
- Entorno de ejecución o máquina virtual.

La plataforma Java 2 también incluyen otras funcionalidades:

- **J2SE**: relacionado con creación de aplicaciones y applets para ejecución en equipo y servidores
- **J2EE**: pensada para creación de aplicaciones web Java empresariales y del lado del servidor
- **Java FX**: crear e implementar aplicaciones de internet enriquecidas.
- **Java Embedded**: creada para su ejecución en sistemas embebidas.
- **Java Card**: tecnología que permite ejecución de pequeñas aplicaciones java en tarjetas inteligentes.



## 7.3 La POO y Java

Los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Estos tendrán un comportamiento (código) y estado (datos).

Los objetos permiten reutilización de código y son piezas reutilizables en proyectos distintos.

El paradigma de la programación orientada a objetos incluye:

- **Encapsulación.**
- **Herencia.**
- **Polimorfismo.**

Una **clase** es un prototipo o molde que indica qué características van a tener los elementos creados a partir de ella. Una **instancia** es el objeto creado a partir de esta clase.

## 7.4. Independencia de la plataforma y trabajo en red

Una característica importante de Java, además de la de independencia de la plataforma en la que se ejecuta, es la posibilidad de crear aplicaciones que trabajan en red.

Esta capacidad ofrece múltiples posibilidades para la comunicación TCP/IP, con bibliotecas que permiten acceso e interacción con HTTP, FTP, etc.

## 7.5. Seguridad y simplicidad.

### Seguridad

Los accesos a zonas de memoria sensible como en C y C++ se han eliminado en Java.

Además el código Java es comprobado y verificado para que no se produzcan efectos no deseados.

Tampoco permite apertura de ficheros en máquina local ni ejecución de aplicación nativa.

### Simplicidad

Es más potente que C o C++ pero más sencillo, eliminando la aritmética de punteros, registros, definición de tipos, gestión de memoria, etc.

Java borra automáticamente los objetos creados cuando determine que no se van a usar. A este proceso se le llama recolector de basura y permite al

programador liberarse de la gestión de memoria.

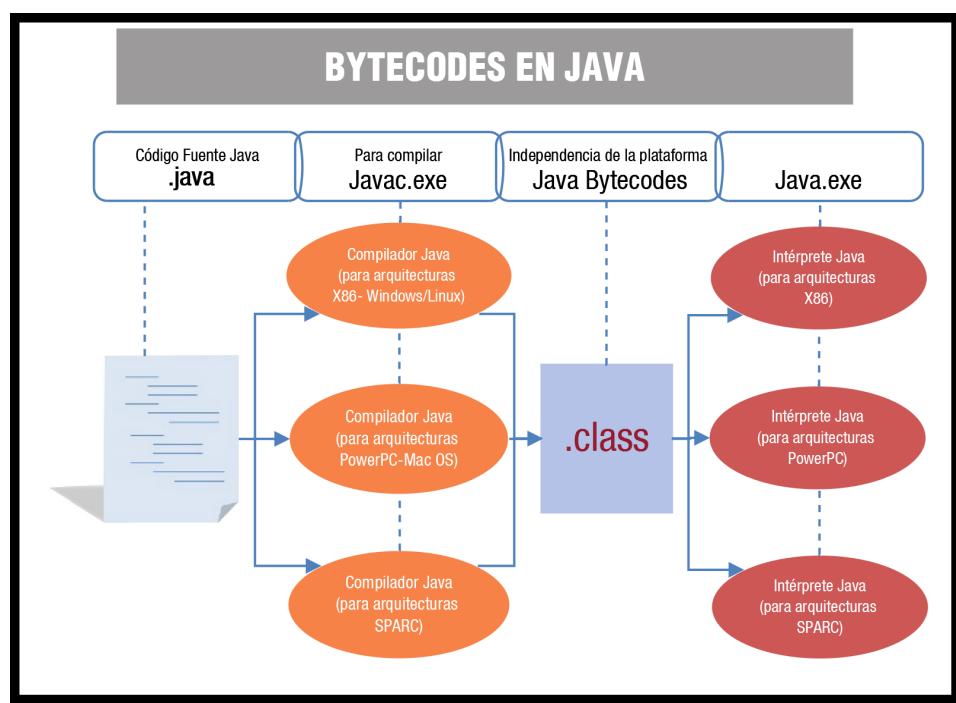
## 7.6 Java y los bytecodes

Una vez escrito el código, este se precompila generándose códigos de bytes (bytecodes) con extensión .class

Estos archivos son interpretados por la máquina virtual.

El proceso de precompilación consiste en verificar los códigos de bytes para asegurarse que:

- El código satisface las especificaciones de la máquina virtual.
- No existe amenaza contra el sistema.
- No se produce desbordamiento de memoria.
- Los parámetros y sus tipos son adecuados.
- No existen conversiones de datos no permitidas.



## 8. Programas java

### 8.1. Estructura de un programa.

```


    /**
     * Estructura general de un programa en Java
    */

public class Clase_Principal
{
    // Definición de atributos de la clase
    // Definición de métodos de la clase
    // Definición de otras clases de usuario

    // Declaración del método main
    public static void main (String[] args)
    {
        //Declaración de variables del método
        //Instrucciones que se quieren insertar aquí...
    }
}


```

- **public class Clase\_Principal:** Es la clase general en la que se incluyen todos los elementos del programa, entre otras cosas contiene el método **main()**, que es el programa principal desde el que se llevará la ejecución del programa. **Solo puede ser public. El nombre del fichero .java coincide con el nombre de la clase.**
  - **public static void main (String[] args):** es el método que representa al programa principal, desde él se podrá usar el resto de clases. Todos los programas tienen un main.
  - **Comentarios:** se añaden entre `/* */` para varias líneas o seguido de `//` para una sola línea.
  - **Bloques de código:** conjunto de instrucciones marcados entre llaves `{ }`
  - **Punto y coma:** cada instrucción debe terminar en `;` para no dar error.

## 8.2. El entorno básico de desarrollo Java

La herramienta básica para desarrollar aplicaciones en java es el JDK (Java Development Kit). También incorpora el JRE (Java Runtime Environment) que incluye la máquina virtual (JVM) y la biblioteca de clases así como otros ficheros para la ejecución de programas.

## 8.3. La API de Java

Es la biblioteca de clases orientada a objetos gratuita de java. Proporciona paquetes de clases útiles para realizar múltiples tareas en un programa, se organiza en paquetes lógicos.

## 8.4. Tipo de aplicaciones Java

### Aplicaciones de consola

- Programas independientes iguales que los creados con lenguajes tradicionales.
- Se componen como mínimo de un .class y un método main.
- No necesitan navegador web y se ejecutan invocando el comando java
- Las aplicaciones de consola se leen y escriben en la entrada y salida estándar, sin interfaz gráfica.

### Aplicaciones gráficas

- Para incluir otros paquetes se utiliza la instrucción import. Por ejemplo para incluir todo el paquete swing se utiliza la instrucción import javax.swing.\*;
- Utilizan clases con capacidades gráficas, como Swing (biblioteca gráfica)

### Applets

- Programas incrustados en otras aplicaciones, normalmente en webs que muestra un navegador. Se descarga el applet y se ejecuta.
- Se descargan desde el servidor y se ejecutan desde la máquina del cliente.
- No acceden a partes sensibles a menos que se le de permisos necesarios en el sistema.
- No tienen método principal.
- Son multiplataforma.

### Servlets

- Son componentes de la parte del servidor de Java EE, programas pensados para trabajar del lado de servidor y desarrollar webs que interactúen con el cliente.

## Midlets

- Aplicaciones creadas para ejecución en sistemas de propósito simple o móviles, como los juegos java.

# 9. Entornos integrados de desarrollor (IDE)

## 9.1. ¿Qué son?

Son aplicaciones para llevar a cabo el proceso completo de desarrollo de software a través de un único programa.

Podemos realizar labores de edición, compilación, depuración, detector de errores y ejecución. Además de otras capacidades únicas de cada entorno.

## 9.2. IDE's actuales

### De libre distribución:

- Netbeans
- Eclipse
- BlueJ
- Jgrasp
- Jcreator LE

### Propietarios o de pago:

- IntelliJ IDEA
- JBuilder
- JCreator
- JDeveloper

## 9.3. Netbeans

Es un producto libre y gratuito sin restricciones de uso. Proyecto de código abierto de gran éxito y comunidad numerosa.

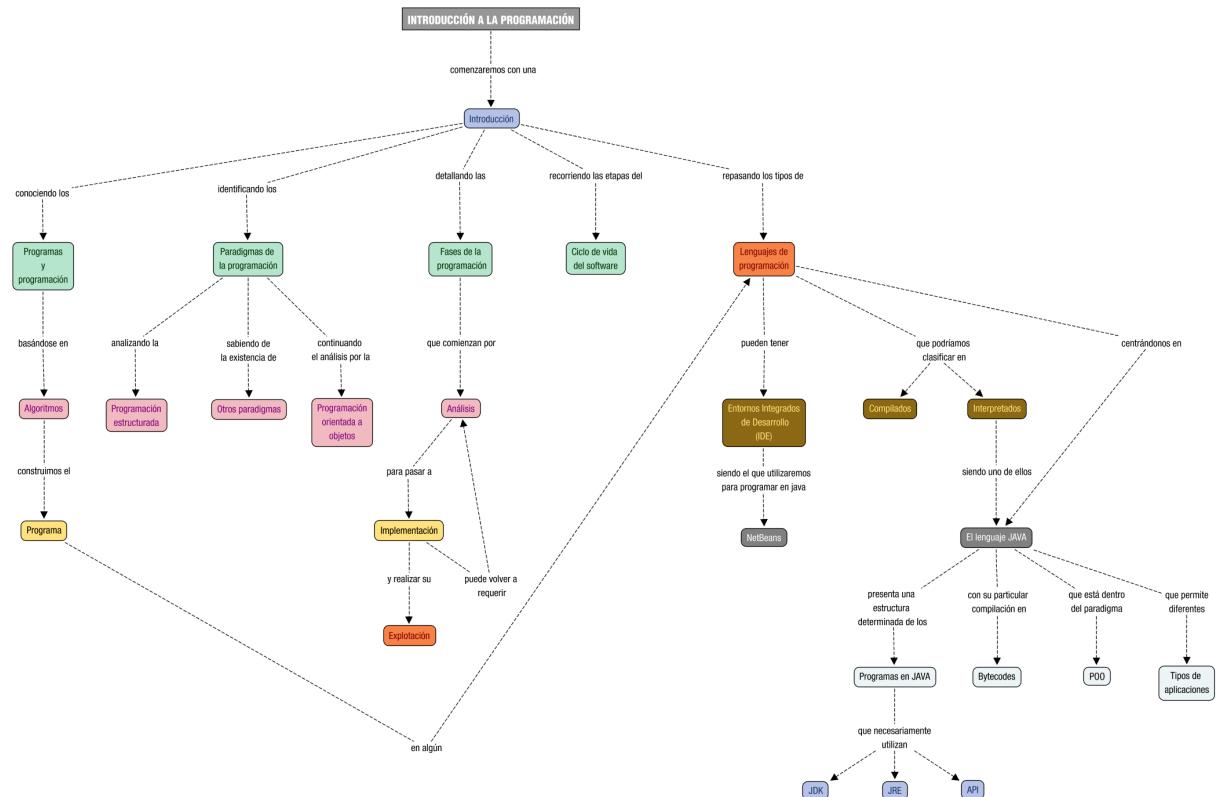
Ofrece posibilidad de escribir en C, C++, Ruby, Groovy, Javascript, CSS y PHP además de Java

Permite crear J2EE

Permite crear aplicaciones Swing de forma sencilla

Permite crear aplicaciones JME para dispositivos móviles.

# Mapa conceptual





# 2. Creación de mi primer programa

	Autor	Xerach Casanova
	Clase	Programación
	Fecha	@Dec 8, 2020 8:34 AM

1. Introducción

2. Las variables e identificadores

    2.1. Identificadores

    2.2. Convenios y reglas para nombrar variables

    2.3. Palabras reservadas

    2.4. Tipos de variables

3. Los tipos de datos

    3.1. Tipos de datos primitivos I

        3.1.1. Tipos primitivos II

    3.2. Declaración e inicialización

    3.3. Tipos referenciados

    3.2. Tipos Enumerados

4. Literales de los tipos primitivos

5. Operadores y expresiones

    5.1. Operadores aritméticos

        Operadores aritméticos básicos

        Operadores incrementales en Java

    5.2. Operadores de asignación

    5.3. Operador condicional

    5.4. Operadores de relación

    5.5. Operadores lógicos

    5.6. Operadores de bits

    5.7. Trabajo con cadenas

    5.8. Precedencia de operadores

6. Conversión de tipo

    6.1. Anexo Conversión de tipos

        Otras consideraciones con los tipos de datos

7. Comentarios

# 1. Introducción

Los programas de ordenador deben resolver un problema y para ellos se debe utilizar de forma inteligente y lógica todos los elementos que nos ofrece el lenguaje. Java es un lenguaje multiplataforma, robusto y fiable y además es orientado a objetos.

## 2. Las variables e identificadores

Una variable es una zona de memoria en el ordenador con un valor que se almacena para ser utilizado más tarde en el programa. Se componen de:

- **Nombre.** Permite al programa acceder al valor que contiene en memoria y debe ser un identificador válido.
- **Tipo de dato.** Especifica qué tipo de información guarda.
- **Rango de valores** que puede admitir.

### 2.1. Identificadores

Un identificador es una secuencia ilimitada sin espacios de letras y dígitos unicode. El primer símbolo debe ser una letra, subrayado (\_) o símbolo de dólar (\$), sin embargo, se desaconseja el uso distinto al del uso de letras.

Unicode es un código de caracteres que recoge los caracteres de prácticamente todos los idiomas importantes del mundo y las líneas de código de los programas se escriben usando unicode, por tanto java se puede utilizar en distintos alfabetos.

El estándar Unicode utilizaba 16 bits pudiendo representar hasta 65.536 caracteres distintos, sin embargo actualmente utiliza más o menos bits dependiendo del formato a utilizar: UTF-8, UTF-16 o UTF-32, en el cual a cada carácter le corresponde un número entero entre 0 a 2 elevado a n, siendo n el número de bits utilizados.

### 2.2. Convenios y reglas para nombrar variables

Aún no siendo obligatorias, existen una serie de normas de estilo de uso generalizado. Estas reglas son:

- Java distingue mayúsculas y minúsculas.
- No utilizar identificadores que comiencen con \$ o con \_ además por convenio el \$ no se utiliza nunca.
- No se puede utilizar true o false ni null.
- Los identificadores deben ser descriptivos. Mejor usar palabras completas en vez de abreviaturas.

Además se recomienda:

- Las variables deben comenzar por letra minúsculas y si hay más de una palabra, se colocan juntas y comenzando por mayúsculas.
- Las constantes se declaran en letras mayúsculas, separando las palabras con guión bajo.
- Las clases comienzan por letra mayúscula y si hay más de una palabra, estás comienzan por mayúscula.
- Las funciones comienzan en letra minúscula.

## 2.3. Palabras reservadas

También llamadas palabras clave o keywords. Su uso se reserva al lenguaje y no pueden utilizarse para crear identificadores.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Algunas palabras reservadas ya no se utilizan en la actualidad en Java **const** o **goto**. Tampoco puede utilizarse **true** y **false**, pero se consideran **literales booleanos** y tampoco **null**, considerado igualmente un literal.

Los editores e IDE utilizan colores para diferenciar palabras reservadas. Estos colores pueden modificarse en las opciones de menú (netbeans: tools - options /fonts & colors)

## 2.4. Tipos de variables

Los distintos tipos de variables dependen del tipo de datos que representan, si su valor cambia o no durante ejecución o qué papel llevan a cabo.

- **Variables de tipos primitivos y variables referencia.**

Según el tipo de información que contengan. En función a que grupo pertenezca, tipos primitivos o tipos referenciados, se podrán realizar con ellas unas operaciones u otras.

- **Variables y constantes.**

Dependen de si su valor cambia o no durante la ejecución del programa.

- **Variable:** representa una zona de memoria del ordenador que contiene un determinado valor y al que se accede a través del identificador.

- **Constantes o variables finales:** lo mismo que las variables, pero su valor no cambia en tiempo de ejecución.
- **Variables miembro y variables locales.**

En función del lugar donde aparezcan en el programa.

- **Variables miembro.** Se crean dentro de una clase, fuera de cualquier método. Pueden ser de tipos primitivos o referencia, variables o constantes.
- **Variables locales:** se crean y se usan dentro de un método o dentro de cualquier bloque de código. La variable deja de existir cuando la ejecución del bloque o método finaliza.

Ejemplo de utilización de variables:

**Constante:** PI

**Variable miembro:** x

**Variable local:** valorantiguo

```

  /**
   * Aplicación ejemplo de tipos de variables

   *
   * @author FMA
   */
public class ejemplovariables {
    final double PI = 3.1415926536; // PI es una constante
    int x; // x es una variable miembro
           // de clase ejemplovariables

    int obtenerX(int x) { // x es un parámetro
        int valorantiguo = this.x; // valorantiguo es una variable local
        return valorantiguo;
    }

    // el método main comienza la ejecución de la aplicación
    public static void main(String[] args) {
        // aquí iría el código de nuestra aplicación
    }
} // fin del método main
} // fin de la clase ejemplovariables

```

### 3. Los tipos de datos

En los lenguajes fuertemente tipados a todo dato: constante, variable o expresión, le corresponde un tipo que es conocido antes de que ejecute el programa.

Un lenguaje fuertemente tipado es un lenguaje que no permite la utilización de un dato distinto al tipo de dato de la variable a menos que se haga una conversión.

Un tipo de dato es una especificación de los valores que son válidos para la variable y de las operaciones que se pueden realizar con ellos.

El tipo de dato de una variable se conoce durante la revisión del compilador para detectar errores.

Los tipos de datos en java se dividen en

- **Tipos sencillos o primitivos:** valores simples predefinidos en el lenguaje: carácter, número...
- **Tipo de datos referencia:** se definen con un nombre o referencia (puntero) que contiene la dirección en memoria de un valor o grupo de valores, por ejemplo arrays o clases.

### 3.1. Tipos de datos primitivos I

Constituyen los tipos de información más habituales: números, caracteres y valores lógicos o booleanos.

**En Java, los datos primitivos no se consideran objetos.**

El compilador optimiza mejor el uso de los tipos primitivos comparado con los objetos. Los tipos primitivos tienen idéntico tamaño y comportamiento en todas las versiones de java y en todo tipo de ordenador, por tanto se asegura la portabilidad.

Tipos de datos primitivos en Java:

Tipo	Descripción	Bytes	Rango	Valor por default
byte	Entero muy corto	1	-128 a 127	0
short	Entero corto	2	-32,768 a 32,767	0
int	Entero	4	-2,147,483,648 a 2,147,483,647	0
long	Entero largo	8	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	0L
float	Numero con punto flotante de precisión individual con hasta 7 dígitos significativos	4	+/-1.4E-45 (+/-1.4 times 10 <sup>-45</sup> ) a +/-3.4E38 (+/-3.4 times 10 <sup>38</sup> )	0.0f
double	Numero con punto flotante de precisión doble con hasta 16 dígitos significativos	8	+/-4.9E-324 (+/-4.9 times 10 <sup>-324</sup> ) a +/-1.7E308 (+/-1.7 times 10 <sup>308</sup> )	0.0d
char	Carácter Unicode	2	\u0000 a \uFFFF	'\u0000'
boolean	Valor Verdadero o Falso	1	true o false	false

Entre los tipos de datos primitivos existe una peculiaridad, esta es el tipo de dato char, el cual es considerado por el compilador como numérico, ya que los valores que guarda son el código unicode. Por tanto puede operarse con él como si se tratase de un número entero.

Para manejar datos mayores a la cifra de 2.17.483.647, no podemos utilizar int ya que es el número máximo de combinaciones posibles de 32 bits, por tanto tendremos que utilizar long o float, pero tienen un problema. **La precisión.**

### 3.1.1. Tipos primitivos II

El tipo de dato real permite representar cualquier número con decimales, en función del número de bits usado, hay más de un tipo de dato real. Cuanto mayor sea ese número:

- Más grande podrá ser el número real representado en valor absoluto.
- Mayor será la precisión de la parte decimal

El almacenamiento de los valores reales que son resultado de otros dos números reales, la gran mayoría se representará de forma aproximada, ya que en el ordenador solo se puede almacenar un número finito de bits.

Un número se expresa como :

$$Valor = \text{mantisa} * 2^{\text{exponente}}$$

Y solo se almacena la mantisa y el exponente al que va elevada la base. Los bits empleados por la matisa representan la precisión del número real (el número de cifras decimales que puede tener, mientras que los bits del exponente expresan la diferencia entre el mayor y el menor número representable (intervalo de representación)).

En java, float tiene precisión 32 bits (24 mantisa / 8 exponente). La mantisa es un valor entre -1.0 y 1.0 y el exponente representa la potencia de 2 para obtener el valor... double tiene precisión 64 bits (53 mantisa / 11 exponente).

Se suele utilizar tipo double, ya que se asegura que los errores cometidos en aproximaciones sean menores. Java considera los valores de coma flotante por defecto en tipo double.

Java utiliza el estándar internacional IEEE 754 para representación interna de números en coma flotante.

## 3.2. Declaración e inicialización

Para declarar una variable podemos hacerlo en cualquier bloque de código dentro de llaves. Se hace indicando identificador y tipo de dato, separadas por , si se declaran varias a la vez:

```
int alumnos = 15;
double radio = 3.14, importe = 102.95;
```

No es obligatorio dar valor a las variables, aunque dependiendo del caso, el compilador producirá error:

- **Variables miembro:** si no le damos valor, se inicializan de manera automática (número = 0, boolean = false y referenciado = null).
- **Variables locales:** no se inicializan automáticamente y debemos asignarles un valor antes de ser usadas. En el siguiente ejemplo daría error, porque p puede no ser inicializada:

```
int p;
if (. . . )
    p = 5 ;
int q = p; // error
```

Para declarar constantes utilizamos la palabra reservada final:

```
final double PI=3.1415926536;
```

### 3.3. Tipos referenciados

Los tipos referenciados se construyen a partir de los 8 tipos primitivos. Se utilizan para almacenar la dirección de los datos en la memoria.

```
int[] arrayDeEnteros; //array del tipo int  
  
Cuenta cuentaCliente; //variable u objeto con una referencia de tipo cuenta.
```

Los arrays son datos agrupados en estructuras para facilitar el acceso a los mismos (datos estructurados).

Además de los 8 tipos primitivos, java trata a los textos o cadenas de caracteres mediante el tipo de dato String. Realmente son objetos y por tanto son tipos referenciados, pero se pueden utilizar como si fueran variables de tipos primitivos.

```
String mensaje= "el primer programa";
```

Para mostrar en pantalla se utiliza **System.out**, que es la salida estándar del programa. Este método escribe un conjunto de caracteres a través de la línea de comandos. Se puede utilizar:

- System.out.print
- System.out.println (sitúa el cursor al principio de la línea siguiente).

Los comentarios se realizan con //.

En los entornos de desarrollo se incluyen funcionalidades para ayudar a escribir código.

```
7  public static void main(String[] args) {  
8      // TODO code application logic here  
9      int i= 10;  
10     double d=3.24;  
11  
12     cannot find symbol  
13         symbol: class string  
14         location: class EjemplosTipos  
15     -----  
16         true;  
17         (Alt-Enter shows hints)  
18  
19         string msj= "Bienvenido a Java";  
20  
21         System.out.println ("La variable i es de tipo entero y su valor es:" + i);  
22     }  
23 }  
24 }
```

Si se coloca el ratón sobre el círculo rojo, Nebeans te informa sobre el error detectado.

También tiene las sugerencias de inserción de código y la generación automática de código, facilitando la inserción generando código automáticamente o sugiriéndote qué escribir en un determinado punto usando Ctrl + Espacio.

## 3.2. Tipos Enumerados

Es una forma de declarar una variable con un conjunto restringido de valores. Se utiliza la palabra reservada **enum** seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A estos valores se les considera constantes.

Las llaves se utilizan porque enum es una especie de clase en Java y todas las clases llevan su contenido entre llaves.

Al ser una clase, también podemos realizar operaciones con él.

A enum, al tener el tratamiento de clase, se le puede añadir métodos y campos o variables en la su declaración.

En el siguiente ejemplo se usa system.out.print. y a veces se utiliza la secuencia de escape llamada carácter de nueva línea (\n) que le indica al compilador que mueva el cursor al principio de la línea siguiente.

```

10  public class tiposenumerados {
11  ┌─ public enum Días {Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo};
12
13  ┌─ public static void main(String[] args) {
14      // código de la aplicación
15      Días diaactual = Días.Martes;
16      Días diasiguiente = Días.Miércoles;
17
18      System.out.print("Hoy es: ");
19      System.out.println(diaactual);
20      System.out.println("Mañana\nes\n" +diasiguiente);
21
22  } // fin main
23
24 } // fin tiposenumerados

```

## 4. Literales de los tipos primitivos

Un literal, valor literal o constante literal es un valor concreto para los tipos primitivos, String o Null.

Los **literales booleanos** tienen dos únicos valores que puede aceptar: true y false.

Los **literales enteros** se pueden representar en:

- **Decimal**
- **Octal:** Empieza siempre por cero seguido de dígitos octales de cero a siete.
- **Hexadecimal.** Empieza siempre por 0x seguido de dígitos hexadecimales (de 0 a 9 y de la a/A a la f/F).

Ejemplo de código:

```

public static void main(String[] args) {
    int value;
    value = 16;
    System.out.println( "16 decimal = " + value) ;
    value = 020;
    System.out.println("20 octal = " + value + " en decimal".);
    value = 0x10;
    System.out.println ("10 hexadecimal = " + value + " en decimal". ) ;
}

```

Mostrará en pantalla:

**16 decimal = 16**

**20 octal = 16 en decimal.**

## 10 hexadecimal = 16 en decimal.

Las **constantes literales** de tipo long se construyen añadiendo detrás I/L, si no, se considera por defecto tipo int (se suele utilizar en mayúsculas para no confundir con un 1).

Los **literales reales** o en coma flotante se expresan con coma decimal o en notación científica. El valor por defecto será double (D), para indicar que es Float se debe finalizar con una F. Ejemplos:

- 13.2D es lo mismo que 1.32e1 y lo mismo que 0.132E2
- .54 / 31,21E-5, 2.f, 6.0222137e23f...

Un **literal carácter** puede escribirse como un carácter entre comillas simples o por su código en tabla unicode anteponiendo la secuencia escape (\) si ponemos el valor en octal o (\u si lo ponemos en hexadecimal).

Ejemplo:

tanto en ASCII como en unicode, la letra A es el 65: en octal es 101 y en hexadecimal es 41: \101 en octal y \u0041 en hexadecimal.

Secuencias de escape en java.

Secuencia de escape	Significado	Secuencia de escape	Significado
\b	Retroceso	\r	Retorno de carro
\t	Tabulador	\"	Carácter comillas dobles
\n	Salto de línea	\'	Carácter comillas simples
\f	Salto de página	\\\	Barra diagonal

Los **literales de cadenas** de caracteres se indican entre comillas simples. Al construir cadenas de caracteres se puede incluir cualquier carácter unicode, excepto un carácter de retorno de carro.

```
String texto = "Juan dijo: \"Hoy hace un día fantástico...\"";
```

Los literales String no necesitan ser creados con la orden new, a pesar de ser objetos, ya que java los crea implicitamente.

# 5. Operadores y expresiones

Los operadores llevan a cabo operaciones sobre un conjunto de datos u operandos. Pueden ser unarios, binarios o terciarios, dependiendo del número de operandos que participen. Actúan sobre tipos de datos primitivos y devuelven datos primitivos.

Estos operadores se combinan con literales y/o identificadores para formar expresiones, que no es otra cosa que una combinación de operadores y operandos el cual después de evaluarse devolverá un único resultado de un tipo determinado.

Las expresiones combinadas con algunas palabras reservadas o por sí mismas forman sentencias o instrucciones: `i+1;` `sum= i+1;...`

## 5.1. Operadores aritméticos

### Operadores aritméticos básicos

Operador	Operación Java	Expresión Java	Resultado
-	Operador unario de cambio de signo	-10	-10
+	Adición	1.2 + 9.3	10.5
-	Sustracción	312.5 – 12.3	300.2
*	Multiplicación	1.7 * 1.2	1.02
/	División (entera o real)	0.5 / 0.2	2.5
%	Resto de la división entera	25 % 3	1

El resultado de expresiones depende de los operandos que intervengan:

Tipo de los operandos	Resultado
Un operando de tipo <code>long</code> y ninguno real ( <code>float</code> o <code>double</code> )	<code>long</code>
Ningún operando de tipo <code>long</code> ni real ( <code>float</code> o <code>double</code> )	<code>int</code>
Al menos un operando de tipo <code>double</code>	<code>double</code>
Al menos un operando de tipo <code>float</code> y ninguno <code>double</code>	<code>float</code>

### Operadores incrementales en Java

Son operadores unarios y podemos utilizarlos con notación prefija si aparece antes del operando o postfija si aparece después.

Tipo operador	Expresión Java	
<b>++ (incremental)</b>	Prefija: x=3; y=++x; // x vale 3 e y vale 4	Postfija: x=3; y=x++; // x vale 4 e y vale 3
<b>--(decremental)</b>	5-- // el resultado es 4	

Ejemplo:

```

10 public class operadoresaritmeticos {
11     public static void main(String[] args) {
12         short x = 7;
13         int y = 5;
14         float f1 = 13.5f;
15         float f2 = 8f;
16         System.out.println("El valor de x es " + x + " y el valor de y es " +y);
17         System.out.println("El resultado de x + y es " + (x + y));
18         System.out.println("El resultado de x - y es " + (x - y));
19         System.out.printf("%s\n%s\n%s\n","División entera:","x / y = ",(x/y));
20         System.out.println("Resto de la división entera: x % y = " + (x % y));
21         System.out.printf("El valor de f1 es %f y el de f2 es %f\n",f1,f2);
22         System.out.println("El resultado de f1 / f2 es " + (f1 / f2));
23     } // fin de main
24 } // fin de la clase operadoresaritmeticos

```

## 5.2. Operadores de asignación

El principal operador de asignación es "=", pero existen algunos compuestos:

Operador	Ejemplo en Java	Expresión equivalente
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>

## 5.3. Operador condicional

El operador condicional "?" evalúa una condición y devuelve un resultado en función de si es verdadera o falsa. Es el único operador ternario.

El primer operando es el que va a ser evaluado y debe ser una expresión booleana o condición. A continuación se pone el operador "?" y seguidamente las dos expresiones se paradas por ":", las cuales se devuelven dependiendo de si la condiciones verdadera (izquierda de los dos puntos) o falsa (derecha de los dos puntos).

Operador	Expresión en Java
:	condición ? exp1 : exp2

Por ejemplo:  $z = (x > y) \ x:y;$

Si x es mayor que y se devuelve variable z almacena x y si no es mayor almacena y.

## 5.4. Operadores de relación

Los operadores relacionales se utilizan para comparar datos de tipo primitivo (numérico, carácter y booleano). El resultado se utiliza en otras expresiones o sentencias que se ejecutarán en función de si se cumple o no la relación.

Operador	Ejemplo en Java	Significado
<code>==</code>	<code>op1 == op2</code>	op1 igual a op2
<code>!=</code>	<code>op1 != op2</code>	op1 distinto de op2
<code>&gt;</code>	<code>op1 &gt; op2</code>	op1 mayor que op2
<code>&lt;</code>	<code>op1 &lt; op2</code>	op1 menor que op2
<code>&gt;=</code>	<code>op1 &gt;= op2</code>	op1 mayor o igual que op2
<code>&lt;=</code>	<code>op1 &lt;= op2</code>	op1 menor o igual que op2

Para introducir valores en nuestro programa se puede utilizar la clase **Scanner** la cual nos permite leer datos que se escriben por teclado. Para ello debemos importar el paquete de clases que la contiene.

El programa espera se queda esperando a que el usuario escriba y pulse la tecla intro. Seguidamente continua la ejecución del programa:

```
public class ejemplorelacionales {
    // método principal que inicia la aplicación
    public static void main( String args[] )
    {
        // clase Scanner para petición de datos
        Scanner teclado = new Scanner( System.in );
        int x, y;
        String cadena;
        boolean resultado;
        System.out.print( "Introducir primer número: " );
        x = teclado.nextInt(); // pedimos el primer número al usuario
        System.out.print( "Introducir segundo número: " );
        y = teclado.nextInt(); // pedimos el segundo número al usuario
        // realizamos las comparaciones
        cadena=(x==y)?"iguales":"distintos";
        System.out.printf("Los números %d y %d son %s\n",x,y,cadena);
        resultado=(x!=y);
        System.out.println("x != y // es " + resultado);
        resultado=(x < y );
        System.out.println("x < y // es " + resultado);
        resultado=(x > y );
        System.out.println("x > y // es " + resultado);
        resultado=(x <= y );
        System.out.println("x <= y // es " + resultado);
        resultado=(x >= y );
        System.out.println("x >= y // es " + resultado);
    } // fin método main
} // fin clase ejemplorelacionales
```

## 5.5. Operadores lógicos

Realizan operaciones sobre valores booleanos o resultados de expresiones relacionales, dando como resultado un valor booleano.

En algunos casos el segundo operando de una expresión lógica no se evalúa ahorrando tiempo de ejecución, por ejemplo en la comparativa `a&&b`, si `a` es falso no se analizará `b` porque ya no se cumple. En el caso contrario `allb`, si `a` es verdadero, `b` no se analiza porque ya se cumple.

En estos casos e favorable colocar de primer operando el valor que sabemos que nos va a ahorrar tiempo de ejecución.

Operador	Ejemplo en Java	Significado
!	!op	Devuelve true si el operando es false y viceversa.
&	op1 & op2	Devuelve true si op1 y op2 son true
	op1   op2	Devuelve true si op1 u op2 son true
^	op1 ^ op2	Devuelve true si sólo uno de los operandos es true
&&	op1 && op2	Igual que &, pero si op1 es false ya no se evalúa op2
	op1    op2	Igual que  , pero si op1 es true ya no se evalúa op2

```

public class operadoreslogicos {
    public static void main(String[] args) {
        System.out.println("OPERADORES LÓGICOS");

        System.out.println("Negacion:\n ! false es : " + (! false));
        System.out.println(" ! true es : " + (! true));

        System.out.println("Operador AND (&):\n false & false es : " + (false & false));
        System.out.println(" false & true es : " + (false & true));
        System.out.println(" true & false es : " + (true & false));
        System.out.println(" true & true es : " + (true & true));

        System.out.println("Operador OR ():\n false | false es : " + (false | false));
        System.out.println(" false | true es : " + (false | true));
        System.out.println(" true | false es : " + (true | false));
        System.out.println(" true | true es : " + (true | true));

        System.out.println("Operador OR Exclusivo (^):\n false ^ false es : " + (false ^ false));
        System.out.println(" false ^ true es : " + (false ^ true));
        System.out.println(" true ^ false es : " + (true ^ false));
        System.out.println(" true ^ true es : " + (true ^ true));

        System.out.println("Operador &&:\n false && false es : " + (false && false));
        System.out.println(" false && true es : " + (false && true));
        System.out.println(" true && false es : " + (true && false));
        System.out.println(" true && true es : " + (true && true));

        System.out.println("Operador ||:\n false || false es : " + (false || false));
        System.out.println(" false || true es : " + (false || true));
        System.out.println(" true || false es : " + (true || false));
        System.out.println(" true || true es : " + (true || true));
    } // fin main
} // fin operadoreslogicos

```

## 5.6. Operadores de bits

Realizan operaciones sobre números enteros o char en su representación binaria (sobre cada dígito binario)

Operador	Ejemplo en Java	Significado
<code>~</code>	<code>~op</code>	Realiza el complemento binario de op (invierte el valor de cada bit)
<code>&amp;</code>	<code>op1 &amp; op2</code>	Realiza la operación AND binaria sobre op1 y op2
<code> </code>	<code>op1   op2</code>	Realiza la operación OR binaria sobre op1 y op2
<code>^</code>	<code>op1 ^ op2</code>	Realiza la operación OR-exclusivo (XOR) binaria sobre op1 y op2
<code>&lt;&lt;</code>	<code>op1 &lt;&lt; op2</code>	Desplaza op2 veces hacia la izquierda los bits de op1
<code>&gt;&gt;</code>	<code>op1 &gt;&gt; op2</code>	Desplaza op2 veces hacia la derecha los bits de op1
<code>&gt;&gt;&gt;</code>	<code>op1 &gt;&gt;&gt; op2</code>	Desplaza op2 (en positivo) veces hacia la derecha los bits de op1

## 5.7. Trabajo con cadenas

Para aplicar una operación a una variable de tipo String, se escribe su nombre seguido de la operación, separados por un punto. Estas son las operaciones que podemos realizar con String

- **Creación:** Simplemente asignando la cadena de caracteres entre comillas dobles.
- **Obtención de longitud:** método `length()`.
- **Concatenación:** se utiliza el operador `+` o el método `concat()`.
- **Comparación.** Se utiliza el método `equals()`, el cual devuelve un valor booleano. `equalsIgnoreCase()` hace lo mismo pero ignorando las mayúsculas.
- **Obtención de subcadenas.** Con el método `substring()`, indicándole el inicio y el fin de la subcadena a obtener.
- **Cambio a mayús/minus.** Con los métodos `toUpperCase()` y `toLowerCase()`.
- **Valueof.** Se utiliza para convertir un tipo de dato primitivo `int`, `long`, `float...` en variable de tipo String.

```

public class ejemplocadenas {
    public static void main(String[] args)
    {
        String cad1 = "CICLO DAM";
        String cad2 = "ciclo dam";

        System.out.printf( "La cadena cad1 es: %s y cad2 es: %s", cad1,cad2 );

        System.out.printf( "\nLongitud de cad1: %d", cad1.length() );

        // concatenación de cadenas (concat o bien operador +)
        System.out.printf( "\nConcatenación: %s", cad1.concat(cad2) );

        //comparación de cadenas
        System.out.printf("\ncad1.equals(cad2) es %b", cad1.equals(cad2) );
        System.out.printf("\ncad1.equalsIgnoreCase(cad2) es %b", cad1.equalsIgnoreCase(cad2) );
        System.out.printf("\ncad1.compareTo(cad2) es %d", cad1.compareTo(cad2) );

        //obtención de subcadenas
        System.out.printf("\ncad1.substring(0,5) es %s", cad1.substring(0,5) );

        //pasar a minúsculas
        System.out.printf("\ncad1.toLowerCase() es %s", cad1.toLowerCase() );

        System.out.println();
    } // fin main

} // fin ejemplocadenas

```

## 5.8. Precedencia de operadores

Las reglas de precedencia de operadores coinciden con las expresiones de álgebra convencional:

- Multiplicación, división y resto se evalúan primero.
- Suma y resta se aplican después de las anteriores.
- Si dentro de la misma operación existen varias operaciones de este tipo se evalúan de izquierda a derecha.

Cuando se evalúa una expresión es necesario tener en cuenta la asociatividad, la cual indica qué operador se evalúa antes, en condiciones de igualdad de precedencia.

Operador	Tipo	Asociatividad
<code>++--</code>	Unario, notación postfija	Derecha
<code>++--+- (cast) ! ~</code>	Unario, notación prefija	Derecha
<code>* / %</code>	Aritméticos	Izquierda
<code>+ -</code>	Aritméticos	Izquierda
<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>	Bits	Izquierda
<code>&lt; &lt;= &gt; &gt;=</code>	Relacionales	Izquierda
<code>== !=</code>	Relacionales	Izquierda
<code>&amp;</code>	Lógico, Bits	Izquierda
<code>^</code>	Lógico, Bits	Izquierda
<code> </code>	Lógico, Bits	Izquierda
<code>&amp;&amp;</code>	Lógico	Izquierda
<code>  </code>	Lógico	Izquierda
<code>?:</code>	Operador condicional	Derecha
<code>= += -= *= /= %=</code>	Asignación	Derecha

Ejemplos:

- En una operación de `10 / 2 * 5`, primero se realiza la división y después la multiplicación.
- En `x=y=z=1`; se comienza asignando el valor 1 a la variable z, luego a la y y por último a x. Además al contrario sería imposible de realizar.

## 6. Conversión de tipo

Se realizan las conversiones de tipo para hacer que el resultado de una expresión sea del tipo que nosotros deseamos.

Existen dos tipos de conversiones:

- **Automáticas:** si a una variable de un tipo se le asigna un valor de otro tipo numérico con menos bits, se realiza conversión automática (valor promocionado). En conversiones automáticas en operaciones aritméticas, el valor más pequeño promociona al valor más grande, ya que el tipo mayor

siempre puede representar cualquier tipo del menor de int a long o de float a double).

- **Explícitas.** Cuando hacemos una conversión de un tipo con más bits a uno con menos. En estos casos debemos indicar la conversión de manera expresa. Se realiza con el operador **cast**. Por ejemplo byte b = (byte) a;. Ejemplo:

```
int a;
byte b;
a = 12; // no se realiza conversión alguna
b = 12; // se permite porque 12 está dentro del rango permitido de valores para b
b = a; // error, no permitido (incluso aunque 12 podría almacenarse en un byte)
byte b = (byte) a; // Correcto, forzamos conversión explícita
```

## 6.1. Anexo Conversión de tipos

		Tipo destino								
		boolean	char	byte	short	int	long	float	double	
Tipo origen	boolean	-	N	N	N	N	N	N	N	
	char	N	-	C	C	CI	CI	CI	CI	
	byte	N	C	-	CI	CI	CI	CI	CI	
	short	N	C	C	-	CI	CI	CI	CI	
	int	N	C	C	C	-	CI	CI*	CI	
	long	N	C	C	C	C	-	CI*	CI*	
	float	N	C	C	C	C	C	-	CI	
	double	N	C	C	C	C	C	C	-	

**N:** Conversión no permitida.

**C1:** Conversión implícita o automática. Un asterisco indica que puede haber posible pérdida de datos.

**C:** Casting de tipos o conversión explícita.

Convertir tipos de coma flotante en tipos enteros implica utilizar siempre un casting, sabiendo que esto implica pérdida de datos.

## Otras consideraciones con los tipos de datos

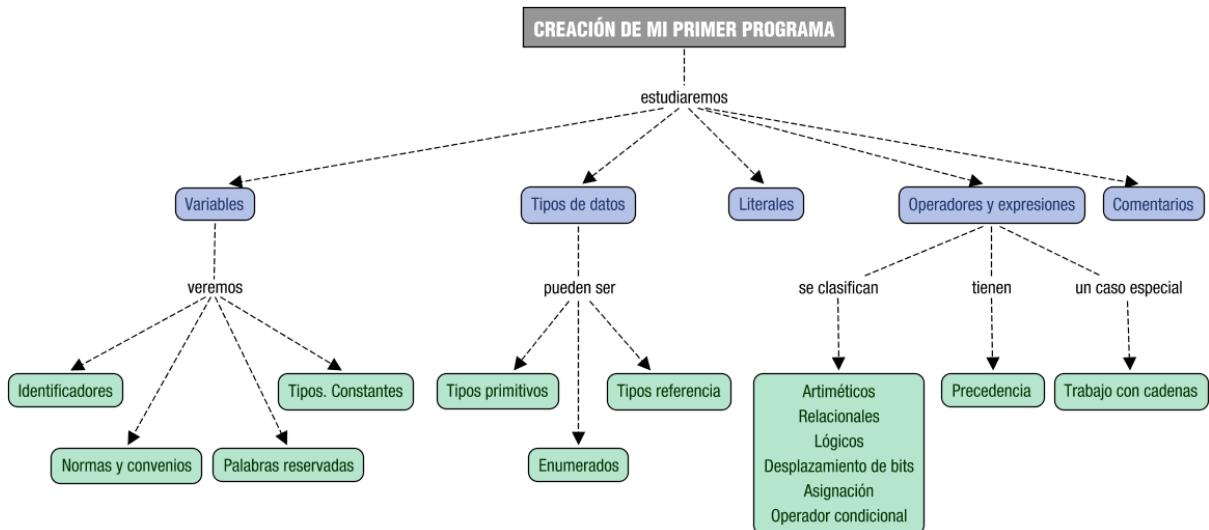
Conversiones de números en Coma flotante (float, double) a enteros (int)	Conversiones entre caracteres (char) y enteros (int)	Conversiones de tipo con cadenas de caracteres (String)
<p>Cuando convertimos números en coma flotante a números enteros, la parte decimal se trunca (redondeo a cero). Si queremos hacer otro tipo de redondeo, podemos utilizar, entre otras, las siguientes funciones:</p> <ul style="list-style-type: none"><li>✓ <b>Math.round(num):</b> Redondeo al siguiente número entero.</li><li>✓ <b>Math.ceil(num):</b> Mínimo entero que sea mayor o igual a num.</li><li>✓ <b>Math.floor(num):</b> Entero mayor, que sea inferior o igual a num.</li></ul> <pre>double num=3.5; x=Math.round(num);      // x = 4 y=Math.ceil(num);       // y = 4 z=Math.floor(num);      // z = 3</pre>	<p>Como un tipo <code>char</code> lo que guarda en realidad es el código <b>Unicode</b> de un carácter, los caracteres pueden ser considerados como números enteros sin signo.</p> <p><b>Ejemplo:</b></p> <pre>int num; char c;  num = (int) 'A';           // num = 65 c = (char) 65;             // c = 'A' c = (char) ((int) 'A' + 1); // c = 'B'</pre>	<p>Para convertir cadenas de texto a otros tipos de datos se utilizan las siguientes funciones:</p> <pre>num=Byte.parseByte(cadena); num=Short.parseShort(cadena); num=Integer.parseInt(cadena); num=Long.parseLong(cadena); num=Float.parseFloat(cadena); num=Double.parseDouble(cadena);</pre> <p>Por ejemplo, si hemos leído de teclado un número que está almacenado en una variable de tipo <code>String</code> llamada <code>cadena</code>, y lo queremos convertir al tipo de datos <code>byte</code>, haríamos lo siguiente:</p> <pre>byte n=Byte.parseByte(cadena);</pre>

## 7. Comentarios

Todos los lenguajes disponen de alguna forma de introducir comentarios en el código. En java se encuentran los siguientes:

- **Comentarios de una sola línea:** `//Comentario`
- **Comentarios de múltiples líneas:** `/* Comentario */`
- **Comentarios Javadoc:** `/** Comentario. Se emplean para generar documentación automática del programa.`

# Mapa Conceptual





# 3. Utilización de objetos

	Autor	Xerach Casanova
	Clase	Programación
	Fecha	@Dec 15, 2020 10:25 AM

1. Introducción

2. Fundamentos de la programación orientada a objetos

2.1. Conceptos

2.2. Beneficios

2.3. Características.

3. Clases y objetos. Características de los objetos

3.1. Propiedades y métodos de los objetos

3.2. Interacción entre objetos

3.3. Clases

4. Utilización de objetos

4.1. Ciclo de vida de los objetos

4.2. Declaración

4.3. Instanciación

4.4. Manipulación

4.5. Destrucción de objetos y liberación de memoria

5. Utilización de métodos

5.1. Parámetros y valores devueltos

5.2. Constructores

5.3. El operador This

5.4. Métodos estáticos

6. Librerías de objetos (paquetes)

6.1. Sentencia import

6.2. Compilar y ejecutar clases con paquetes

6.3. Jerarquía de paquetes

6.4. Librerías java

7. Programación de la consola: entrada y salida de la información

7.1. Conceptos sobre la clase System

7.2. Entrada por teclado. Clase System.

7.3. Entrada por teclado. Clase Scanner

7.4. Salida por pantalla

7.5. Salida de error.

# 1. Introducción

Los programas son el resultado de la búsqueda y obtención de una solución para un problema del mundo real. La programación orientada a objetos (POO), establece una serie de técnicas que permiten trasladar problemas del mundo real a nuestro sistema informático.

# 2. Fundamentos de la programación orientada a objetos

Dentro de la programación se destacan dos paradigmas fundamentales:

**Programación estructurada:** se crean funciones y procedimientos que definen las acciones a realizar y que posteriormente forman los programas.

**Programación orientada a objetos:**, considera los programas en términos de objetos y todo gira alrededor de ellos.

La programación estructurada consiste en descomponer el problema en unidades más pequeñas hasta llegar a acciones o verbos muy simples y fáciles de codificar. Un ejemplo sería resolver una ecuación de primer grado (pedir, calcular, mostrar):

1. Pedir valor de coeficientes
2. Calcular el valor de la incógnita
3. Mostrar el resultado.

La programación orientada a objetos, en lugar de descomponer en acciones, lo hace en objetos. Se trata de trasladar la visión del mundo real a nuestros programas.

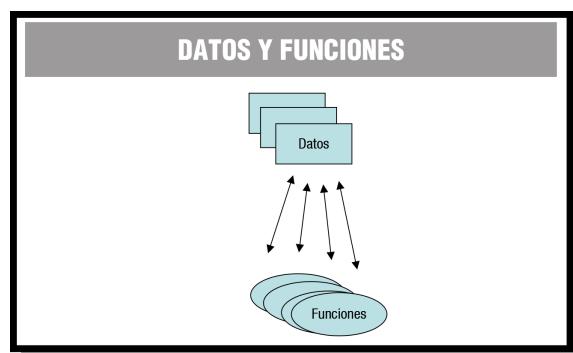
En definitiva, mientras la programación estructurada se centra en el conjunto de acciones a realizar en un programa, haciendo una división de procesos y datos, la programación orientada a objetos se centra en la relación que existe entre los datos y las acciones a realizar con ellos, y los encierra dentro del concepto objeto, tratando de realizar una abstracción lo más cercana al mundo real. Ambas no son excluyentes, ya que parte del paradigma orientado a objetos es implementado siguiendo los principios de la programación estructurada.

Se trata de representar entidades y objetos que nos encontramos en el mundo real mediante componentes de una aplicación.

## 2.1. Conceptos

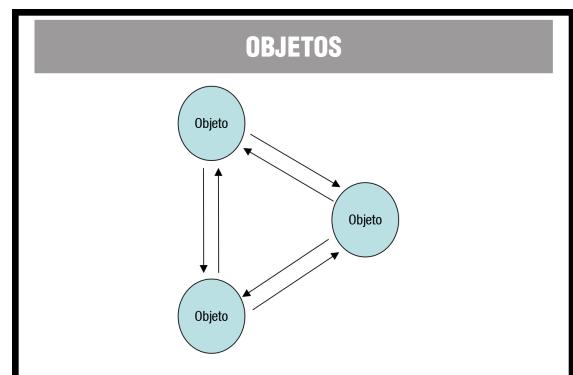
En la programación estructurada, dentro de las funciones se situaban las instrucciones del programa que manipulaban los datos. Funciones y datos se encontraban separados e independiente. Esto genera dos problemas:

- Los programas se creaban y estructuraban de acuerdo a la arquitectura del ordenador donde se van a ejecutar.
- Al estar todos los datos separados de las funciones, estos eran visibles en toda la aplicación y cualquier modificación de datos requería modificaciones en todas las funciones del programa.



En POO los objetos permiten mayor abstracción:

- El programador organiza su programa en objetos, representaciones del mundo real, más cercanas a la forma de pensar de las personas.
- Los datos y las funciones que los manipulan son parte interna de los objetos y no son accesibles al resto. Los cambios en los datos afectan solo a las funciones de ese objeto, pero no al resto de la aplicación.



La POO es la que presenta mayor facilidad para el desarrollo de programas basados en interfaces gráficas.

## 2.2. Beneficios

- **Comprepción:** Los conceptos del espacio del problema se refleja en el código, la lectura del código describe la solución del problema en el mundo real.
- **Modularidad:** aplicaciones mejor organizadas y más fáciles de entender al estar en módulos o archivos independientes.
- **Fácil mantenimiento.** Modificar las acciones hace que se refleje en los datos al estar estrechamente relacionados. Además, al estar las apps mejor organizadas es fácil de localizar cualquier elemento a modificar y corregir. El mayor coste de software se encuentra en el mantenimiento y no en su desarrollo.
- **Seguridad.** Se reduce la probabilidad de cometer errores, ya que no podemos modificar datos de objetos directamente, sino mediante acciones definidas para el objeto.
- **Reusabilidad.** Los objetos se definen como entidades reutilizables en programas que trabajan con la misma estructura de información.

## 2.3. Características.

- **Abstracción.** Definimos las características del objeto sin preocuparnos de como se escribirán en el código del programa. Simplemente se define de una forma general. En POO la herramienta de abstracción más importante es la clase, el cual es un tipo de dato que agrupa las características comunes de un conjunto de objetos. Por ejemplo. Pensando en una clase "vehículo" que agrupa las características comunes de todos ellos, podemos crear a partir de dicha clase objetos como coche, camión, moto... Vehículo es una abstracción de coche, camión y moto.
- **Modularidad.** Una vez representado el escenario del problema en nuestra aplicación, obtendremos un conjunto de objetos software a utilizar, que se crean a partir de una o varias clases. Cada clase es un archivo distinto, lo cual nos permite modificar las características de los objetos sin afectar al resto de clases.
- **Encapsulamiento (ocultación de la información).** Se ocultan las partes internas del objeto a los demás objetos y al mundo exterior, o se restringe el acceso para no ser manipulado de manera inadecuada. Por ejemplo. Persona y coche. El objeto persona utiliza el objeto coche para llegar a su destino utilizando sus acciones, pero no tiene por qué saber como funciona internamente.

- **Jerarquía.** Definimos jerarquías entre clases y objetos, las dos más importantes son: "**es un**" (generalización o especialización) y "**es parte de**" (llamada agregación).

- **La generalización o especialización** es conocida como herencia, la cual permite crear una clase nueva en términos de una clase ya existente (herencia simple) o de varias clases (múltiple). Ejemplo: Coche - Coche de carreras. Coche de carreras hereda las características de coche y además tiene las suyas propias.
- **La agregación o inclusión**, permite agrupar objetos relacionados entre sí dentro de una clase. Por ejemplo, motor, ruedas, frenos y ventanas son agregados de coche

## 2.4. Lenguajes de programación orientados a objetos

- **Simula (1962).** El primero en introducir el concepto clase como elemento que incorpora datos y sus operaciones sobre estos. En 1967 surge Simula 67 con mayor número de tipos de datos.
- **SmallTalk (1972).** Basado en Simula 67, la primera fue SmallTalk 72 y la siguiente **SmallTalk 76**. Soporta propiedades de POO y posee un entorno que facilita el rápido desarrollo de aplicaciones. Su contribución más importante es el modelo-vista-controlador.
- C++ (1985). Diseñador por Bjarne Stroustrup Deriva de C y se añaden mecanismos que lo convierten en lenguaje orientado a objetos. No tiene recolector de basura automática. En este lenguaje aparece el concepto de clase tal y como los conocemos actualmente
- Eiffel (1986). Creado por Bertrand Meyer, sintaxis parecida a C, no logró la aceptación de C y Java.
- Java (1995). Diseñado por Gosling de Sun Microsystems, fue diseñado desde cero, su característica principal es que produce un bytecode que se interpreta en una máquina virtual.
- C# (2000). Creado por Microsoft como ampliación de C, basado en C++ y Java. Evita muchos problemas de diseño de C++.

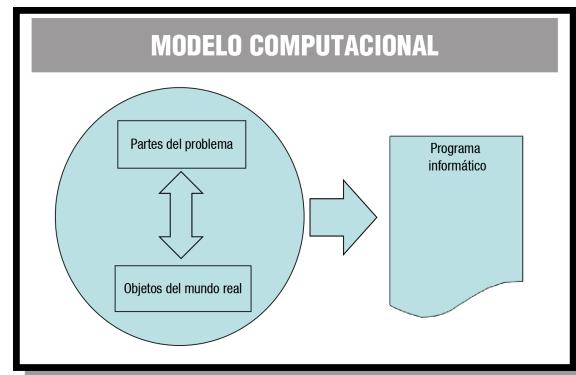
Lenguajes actuales no concebidos como POO, tales como JavaScript o PHP están migrando a POO.

# 3. Clases y objetos. Características de los objetos

Un objeto de software es una representación de un objeto en el mundo real, compuesto por una serie de características y comportamiento.

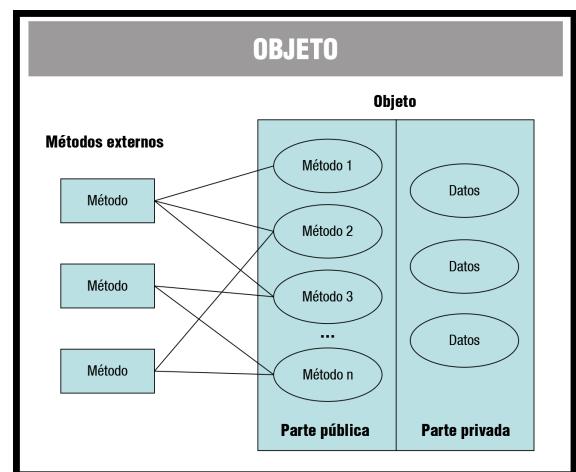
Es un conjunto de datos con las operaciones definidas para ellos. Los objetos tienen un estado y un comportamiento. Sus características son:

- **Identidad.** Permite diferenciar un objeto de otro, aunque dos objetos sean exactamente iguales en sus atributos, son distintos entre sí. Por ejemplo dos vehículos que salen de la misma cadena de montaje.
- **Estado.** Viene determinado por los parámetros o atributos que lo describen: marca, modelo, color, cilindrada...
- **Comportamiento.** Acciones que se pueden realizar sobre el objeto: arrancar(), parar(), acelerar(), frenar()...



## 3.1. Propiedades y métodos de los objetos

- **Campos, atributos o propiedades.** Almacena los datos del objeto, también llamadas variables miembro. Pueden ser de tipo primitivo, o ser a su vez otro objeto, por ejemplo en un objeto coche, puede ser la clase ruedas.
- **Métodos o funciones miembro.** Lleva a cabo las operaciones sobre los atributos definidos para ese objeto.



Un objeto debe reunir en una sola entidad los datos y operaciones. Para acceder a esos datos (privados) debemos utilizar los métodos definidos para el objeto. De esta manera evitamos que métodos externos puedan alterar datos del objeto de manera inadecuada (métodos encapsulados dentro del objeto).

## 3.2. Interacción entre objetos

Los objetos se comunican entre ellos llamando a sus métodos. Estos métodos describen el comportamiento de un objeto cuando recibe una llamada: Objeto1 ejecuta el método de objeto2 y recibe el Objeto2 recibe un mensaje de Objeto1.

Estos mensajes que reciben los objetos o a los que pueden responder reciben el nombre de protocolo de objeto.

El proceso de interacción entre objetos consiste en "envío de mensaje (petición a un objeto) y ejecución del código del método (determina que hacer con el mensaje).

Un programa se ejecuta generando las siguientes acciones:

- Creación de objetos a medida que se necesitan.
- Comunicación entre objetos mediante envíos de mensajes o el usuario a los objetos.
- Eliminación de objetos cuando no son necesarios y así dejar espacio libre en memoria.

## 3.3. Clases

Una clase es la descripción de un conjunto de objetos que comparten estructura y comportamiento común. A partir de esa clase se crean copias o instancias.

Por ejemplo. Tenemos la clase vehículo con un conjunto de propiedades: marca, potencia, velocidad máxima y conjunto de métodos: aumentarVelocidad, reducirVelocidad, etc... Podemos crear a partir de ahí el objeto objBMW318, objMercedes220.

La clase no existe en memoria, solo define la estructura de los datos, cuando un objeto es instanciado, se reserva espacio en memoria para alojar sus datos.

Una clase se comporta como un tipo de datos y el objeto es una instancia de esos tipos de datos (una variable de esos tipos de datos).

**Las clases constan de atributos (comunes a todos los objetos de esa clase) y métodos (que se utilizan para manejar esos objetos). Un programa informático se compone por un conjunto de clases, a partir de las cuales se crean objetos que interactúan entre sí.**

La declaración de una clase está compuesta por:

- **Cabecera.** Compuesta por una serie de modificadores, como public (indica que es pública y se puede acceder desde otras clases), la palabra class y el nombre de la clase.
  - **Cuerpo de la clase,** se especifica entre llaves los atributos y métodos de la clase.

```
1  /*
2   *   Estructura de una clase en Java
3   */
4
5  Cabecera de la clase
6  public class NombreClase { Cuerpo de la clase
7      // Declaración de los atributos
8
9      // Declaración de los métodos
10
11     public static void main (String[] args) {
12         // Declaración de variables y/o constantes
13
14         // Instrucciones del método
15     }
16
17
18
19 */
```

**El método main() se utiliza para indicar que se trata de una clase principal, a partir de la cual va a empezar la ejecución del programa. Este método no aparece si la clase que estamos creando no va a ser la clase principal del programa.**

## 4. Utilización de objetos

El concepto objeto e instancia de clase es lo mismo. Los objetos se crean a partir de clases y representan casos individuales de estas.

Por ejemplo, una clase persona reúne características comunes de las personas: color de pelo, ojos, peso, altura... y las acciones que pueden realizar: dormir, comer, crecer..., dentro del programa podemos crear un objeto trabajador que puede estar basado en la clase persona. **Trabajador es una instancia de la clase persona y la clase persona es una abstracción del objeto trabajador.**

Cada objeto tiene una zona de almacenamiento propia en memoria, donde se guarda su información que será distinta a la de cualquier otro objeto. A las

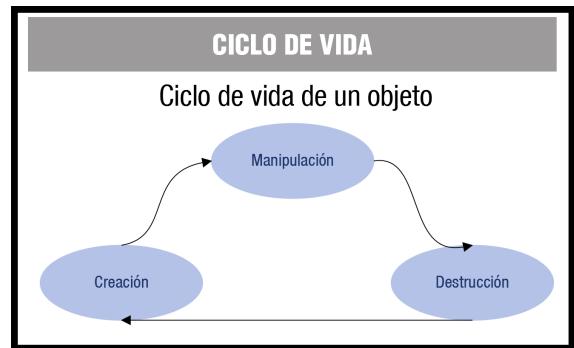
variables miembro instanciadas se les llama variables de instancia y a los métodos, métodos de instancia.

## 4.1. Ciclo de vida de los objetos

En java, todo programa parte de la clase principal, que ejecuta el contenido del método main(), el cual utilizará las demás clases del programa y lanzará mensajes a otros objetos.

Las instancias tienen un tiempo de vida determinado:

- Creación. Reserva de memoria e inicialización de atributos.
- Manipulación. Cuando se hace uso de los atributos y métodos.
- Destrucción. Eliminación y liberación de recursos.



## 4.2. Declaración

Para crear un objeto hay que declararlo (definir el tipo de objeto) e instanciarlo, con el operador new:

Para declarar un objeto se utiliza **<tipo> nombreobjeto;** - Por ejemplo:

```
Vehiculo coche;
```

Tipo es la clase a partir de la cual se creará el objeto y nombre\_objeto es el nombre de la variable de referencia.

Los tipos referenciados o referencias se utilizan para guardar la dirección de los datos en memoria.

Cuando creamos una referencia, se encuentra vacía, no contiene ninguna instancia y su valor es null. La referencia está creada pero no el objeto.

Por ejemplo, en **String mensaje;** String es un tipo de dato referenciado en el que String es realmente la clase a partir de la cual creamos un objeto llamado mensaje. Mensaje aún no contiene el objeto porque no ha sido instanciado.

Cuando creamos un objeto hacemos uso de una variable que almacena la dirección de ese objeto en memoria. Esa variable es una referencia o tipo de dato referenciado, porque no contiene el dato sino la posición del mismo en memoria.

```
String saludo = new String ("Bienvenido a Java");  
  
String s; //s vale null  
  
s = saludo; //asignación de referencias
```

En las instrucciones anteriores, las variables s y saludo apuntan al mismo objeto de la clase String. **Esto implica que cualquier modificación en el objeto saludo modifica también el objeto al que hace referencia la variable s, ya que realmente son el mismo.**

Los nombres de la clase empiezan con mayúscula, como String, y los nombres de los objetos con minúscula, como mensaje, así sabemos qué tipo de elemento utilizando.

### 4.3. Instanciación

Cuando creamos la referencia al objeto debemos instanciarla. Para ello utilizamos new:

```
nombre_objeto = new <Constructor_de_la_Clase>([<par1>, <par2>, ..., <parN>]);
```

- Nombre es el objeto de la variable referencia.
- new es el operador que crea el objeto.
- constructor de la clase es el método especial de la clase que se llama igual que ella e inicia el objeto, dándole valores iniciales a sus atributos.
- par1-parN son parámetros que puede necesitar el constructor para dar valores iniciales a los atributos.

Cuando instanciamos el objeto se reserva memoria para él, de esta tarea se encarga java y el recolector de basura, que se encargará de eliminar de la memoria los objetos no utilizados.

Por ejemplo para String utilizaríamos lo siguiente:

```
mensaje = new String;
```

En este ejemplo, el objeto se crea con la cadena vacía (""), pero también podríamos darle contenido:

```
mensaje = new String ("El primer programa");
```

Java permite utilizar la clase String como si de un tipo de dato primitivo se tratara, por eso no hace falta utilizar el operador new para instanciar un objeto de la clase String.

Por último, debemos tener en cuenta que podemos declarar e instanciar un objeto en la misma instrucción:

```
String mensaje = new String ("El primer programa");
```

## 4.4. Manipulación

Una vez creado e instanciado podemos acceder a su contenido a través de sus atributos y métodos utilizando el nombre del objeto y el operador punto, seguido del nombre del atributo o método a utilizar.

Por ejemplo para acceder a las variables de instancia o atributos:

```
nombre_objeto.atributo;
```

Para acceder a métodos o funciones miembro del objeto:

```
nombre_objeto.metodo([par1, par2, ..., parN]);
```

par1, par2,... son parámetros que utiliza el método y son opcionales, dependiendo de si el método lo necesita o no.

Por ejemplo, para crear un rectángulo se utiliza la clase Rectangle, que se obtiene de la biblioteca o paquete de librería java.awt.

Instanciamos el objeto utilizando el método constructor, llamado igual que el objeto, indicando los parámetros correspondientes a la posición y dimensión del rectángulo:

```
Rectangle rect = new Rectangle(50, 50, 150, 150);
```

Si queremos cambiar el valor de los atributos utilizaremos:

```
rect.height = 100;  
rect.width = 100;
```

o podemos utilizar un método para hacer lo anterior:

```
rect.setSize(200,200);
```

---

```
public class Manipular {  
  
    public static void main(String[] args) {  
  
        // Instanciamos el objeto rect indicando posicion y dimensiones  
        Rectangle rect = new Rectangle( 50, 50, 150, 150 );  
  
        //Consultamos las coordenadas x e y del rectangulo  
        System.out.println( "----- Coordenadas esquina superior izqda. -----");  
        System.out.println("tx = " + rect.x + "\ny = " + rect.y);  
  
        // Consultamos las dimensiones (altura y anchura) del rectangulo  
        System.out.println( "\n----- Dimensiones -----");  
        System.out.println("Alto = " + rect.height );  
        System.out.println( "Ancho = " + rect.width);  
  
        //Cambiar coordenadas del rectangulo  
        rect.height=100;  
        rect.width=100;  
  
        rect.setSize(200, 200);  
        System.out.println( "\n-- Nuevos valores de los atributos --");  
        System.out.println("tx = " + rect.x + "\ny = " + rect.y);  
        System.out.println("Alto = " + rect.height );  
        System.out.println( "Ancho = " + rect.width);  
  
    }  
}
```

## 4.5. Destrucción de objetos y liberación de memoria

En java corre a cargo del recolector de basura (garbage collector). Es un sistema de destrucción automática de objetos que ya no son utilizados. Lo que hace es

liberar una zona de memoria reservada previamente con new.

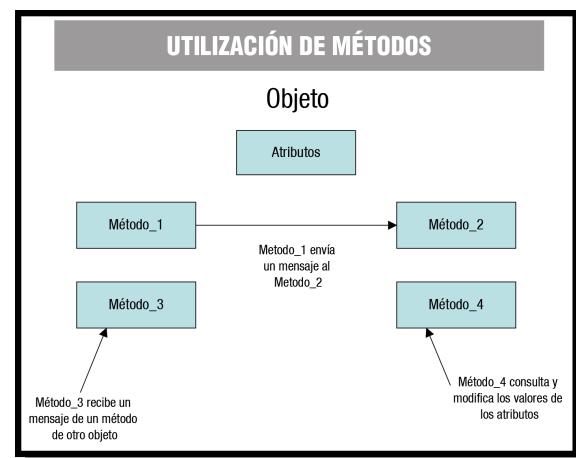
El recolector de basura se ejecuta en segundo plano y de manera muy eficiente para no afectar a la velocidad del programa. Periódicamente busca objetos que ya no son referenciado y cuando los encuentra los marca para ser eliminados, los cuales elimina en el momento que considere oportuno.

Un objeto eliminado se hace ejecutando el método finalize(), si por ejemplo queremos forzar la finalización de un objeto podemos utilizar el método runFinalization() de la clase System, la cual forma parte de la biblioteca de clases de Java y contiene diversas clases para entrada y salida de información, acceso a variables de entorno y otros métodos de utilidad.

## 5. Utilización de métodos

Los métodos contienen la declaración de variables locales y las operaciones que se pueden realizar para el objeto, que son ejecutadas cuando el método es invocado.

Se definen en el cuerpo de la clase y posteriormente son instanciados para convertirse en método instancia de objeto.



Los métodos se componen de cabecera y cuerpo. La cabecera también tiene modificadores. Dentro de un método nos encontramos el cuerpo, que contiene el código de la acción a realizar. Las acciones que un método puede realizar son:

- Inicializar atributos.
- Consultar valores de atributos.
- Modificar valores de atributos.
- Llamar a otros métodos del mismo objeto u objetos externos.

### 5.1. Parámetros y valores devueltos

Los métodos sirven para consultar información del objeto o para modificar su estado.

La información se devuelve a través de valores de retorno y la modificación se hace mediante la lista de parámetros.

La lista de parámetros se puede declarar de dos formas:

- **Por valor.** El valor no se devuelve al finalizar el método y la modificación de esos parámetros no tienen efecto una vez se salga de él. Dicho método recibe una copia de los argumentos, así que las modificaciones se hacen sobre la copia y no sobre las variables originales.
- **Por referencia.** Los valores de los parámetros sí tienen efecto tras la finalización del método. Cuando pasamos una variable a un método por referencia, en realidad estamos pasando la dirección de memoria del dato.

En java, todas las variables se pasan por valor, excepto los objetos, que se hace por referencia.

La declaración de un método en java tiene dos restricciones:

- **Un método siempre devuelve un valor** (no hay valor por defecto) y se llama valor de retorno. Se devuelve al método que lo llamó cuando este se termina de ejecutar. Puede ser tipo primitivo, referenciado o tipo void (que no devuelve ningún valor).
- **Tiene un número fijo de argumentos.** Los argumentos son variables a través que se pasan al método desde el lugar del que se llame, para que éste pueda utilizar sus valores. Reciben el nombre de parámetros cuando aparecen en la declaración del método.

La cabecera de un método se declara como sigue:

```
public tipo_de_dato_devuelto nombre_metodo (Tipo_Par1 NombrePar1, TipoPar2 NombrePar2...);
```

El tipo de dato que se devuelve se declara después del modificador (public) y se corresponde con el valor de retorno. La lista de parámetros aparece al final de la cabecera, encerrados entre paréntesis y separados por coma, indicando el tipo de dato.

La lista de argumentos en la llamada a un método debe coincidir en número, tipo y orden con los parámetros para que no produzca error.

## 5.2. Constructores

Un constructor es un método especial con el mismo nombre que la clase y que no devuelve ningún valor tras su ejecución.

Cuando instanciamos un objeto, el nombre de la clase a la que hacemos referencia realmente es el nombre del constructor, el cual es un método especial que sirve para inicializar valores.

Por ejemplo, la clase Date proporcionada por la biblioteca de clases de java, se utiliza instanciando un objeto a partir de ella:

```
Date fecha = new Date()
```

Estamos creando un objeto fecha de tipo Date, el cual contiene la fecha y hora actual del sistema.

La estructura de los constructores es similar a la del método, salvo que no devuelve ningún valor. En el cuerpo, se contiene la inicialización de atributos y el resto de instrucciones del constructor.

- El constructor es invocado automáticamente en la creación de un objeto y solo una vez.
- No empiezan con minúscula al llamarse igual que la clase.
- Puede haber varios constructores en una sola clase.
- El constructor puede tener parámetros para definir qué valores dar a los atributos del objeto.
- El constructor por defecto es el que no tiene parámetros. Si no definimos un constructor por defecto e intentamos utilizarlo tendremos un error de compilación.
- Toda clase tiene al menos un constructor. Si no se define un constructor el compilador lo crea vacío por defecto e inicializa los atributos a sus valores

### Estructura interna de un método constructor

```
public NombreClase (par1, par2, ..., parN)
{
    // Inicialización de atributos
    // Resto de instrucciones del constructor
}
// Fin del metodo
```

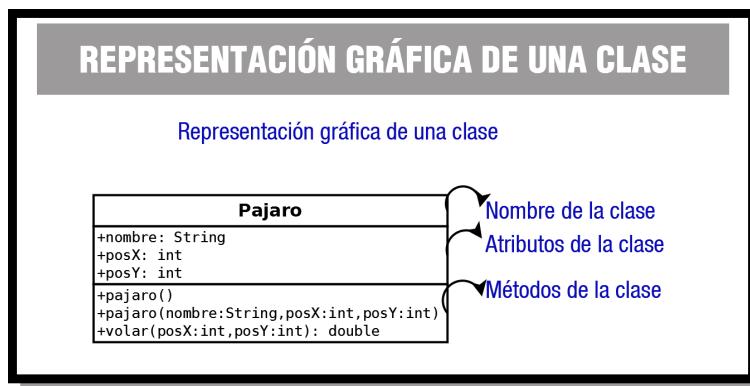
parámetros opcionales  
el modificador normalmente siempre es public  
puede llamar a otros métodos de la clase aunque no es recomendable

por defecto según el tipo que sean. El constructor al que se llama pertenece a su súper clase (clase de la que hereda) y si este no tiene constructor se produce error en la compilación.

## 5.3. El operador This

Constructores y métodos suelen utilizar este operador, que sirve para referirse a los atributos de un objeto cuando estamos dentro de él, sobre todo cuando existe ambigüedad entre el nombre de un parámetro y el de un atributo (`this.nombre_atributo`).

Ejemplo de utilización de objetos y métodos con uso de parámetros y operador `this`:



- `pajaro()` es el constructor por defecto, no contiene instrucción, java inicializa las variables miembro automáticamente si no les damos valor.
- `pajaro(String nombre, int posX, int posY)` es el constructor que recibe como argumentos una cadena de texto y dos enteros para inicializar el valor de los atributos.
- `volar(int posX, int posY)`. El método recibe como argumentos los dos enteros y devuelve un valor de tipo `double` como resultado, usando la palabra clave `return`. Ese valor devuelto es el resultado de aplicar un desplazamiento de acuerdo a la siguiente fórmula:

$$\text{desplazamiento} = \sqrt{\text{posX} \cdot \text{posX} + \text{posY} \cdot \text{posY}}$$

1. Creamos la clase pájaro con sus métodos y atributos:

```

public class Pajaro {

    String nombre;
    int posX, posY;

    public Pajaro() {
    }

    public Pajaro(String nombre, int posX, int posY) {
        this.nombre=nombre;
        this.posX=posX;
        this.posY=posY;
    }
    double volar (int posX, int posY) {

        double desplazamiento = Math.sqrt( posX*posX + posY*posY );
        this.posX = posX;
        this.posY = posY;

        return desplazamiento;
    }
}

```

Creamos un método main() en el cual situamos el código de nuestro programa. Este código consiste en crear una instancia de la clase pajaro y ejecutar sus métodos: el constructor y el método volar(). También podemos imprimir el resultado del método (crear, invocar, imprimir)

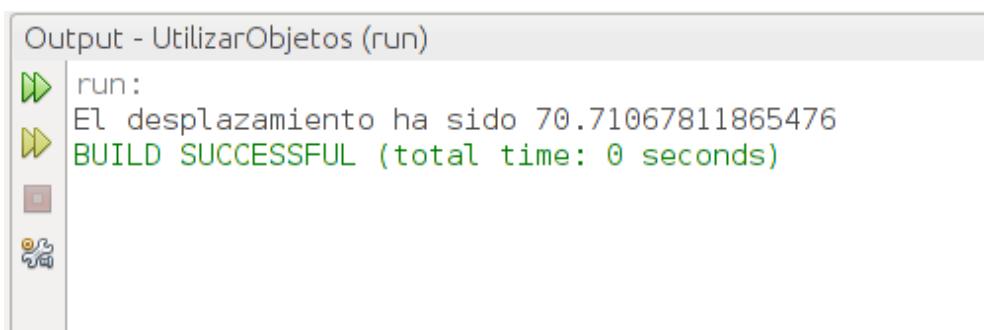
```

public static void main(String[] args) {

    Pajaro loro = new Pajaro("Lucy",50,50) ;
    double d = loro.volar(50,50);
    System.out.println("El desplazamiento ha sido " + d);
}

```

El resultado es:



The screenshot shows the 'Output' window from an IDE during a run. The title bar says 'Output - UtilizarObjetos (run)'. The window contains the following text:

```

run:
El desplazamiento ha sido 70.71067811865476
BUILD SUCCESSFUL (total time: 0 seconds)

```

## 5.4. Métodos estáticos

Los métodos estáticos son aquellos métodos definidos para una clase que se pueden utilizar sin necesidad de crear un objeto de dicha clase. También se llaman métodos de clase, por ejemplo los métodos de String son estáticos.

Para llamarlos utilizamos:

- Nombre del método si lo llamamos desde la misma clase en la que está definido.
- Nombre de la clase, seguido del operador punto, más el método estático, si lo llamamos desde una clase distinta.
- Nombre del objeto, seguido del operador punto, más el nombre del método estático, cuando tengamos un objeto instanciado de la clase en la que se encuentra un método estático.

Estos métodos no afectan al estado de los objetos instanciados (variables instancia) y suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase. Sirve, por ejemplo para contar el número de objetos instanciados de una clase, a través de una variable entera de la clase que se incrementa conforme se van creando objetos.

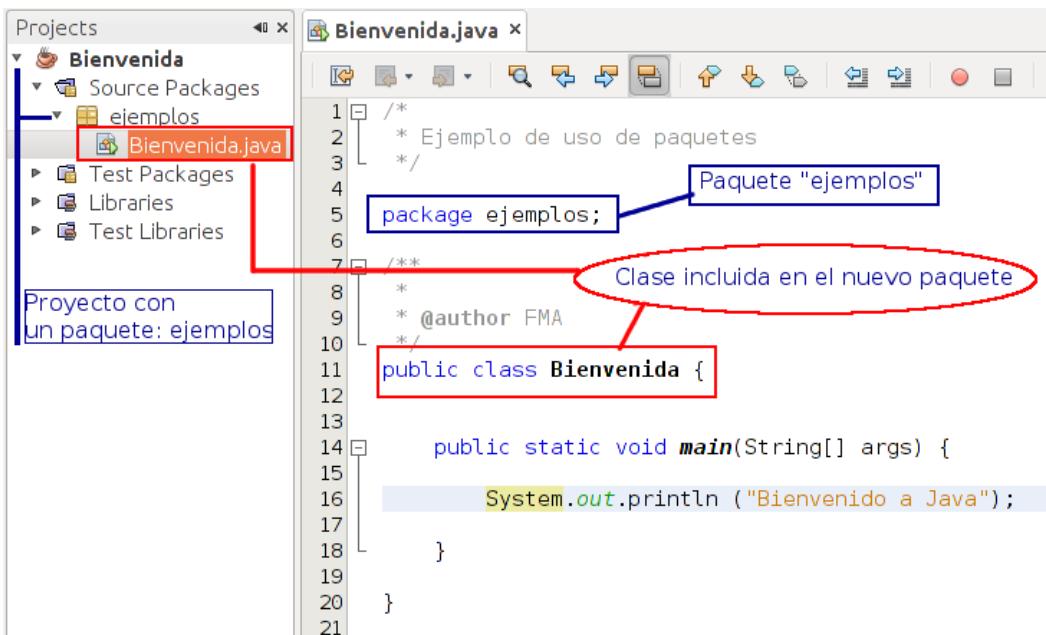
## 6. Librerías de objetos (paquetes)

Un paquete de clases es una agrupación de clases que consideramos que están relacionadas entre sí o tratan un tema común.

Las clases de un mismo paquete tienen acceso privilegiado a los atributos y métodos de otras clases del mismo paquete. Por ello, se considera a los paquetes, en cierto modo, unidades de encapsulación y ocultación de información.

Cada fichero .java se crea dentro del paquete que indiquemos. Los paquetes se declaran utilizando la palabra clave package, seguida del nombre del paquete:

```
package nombre_de_paquete;
```



En Netbeans se pueden crear paquetes en el paquete raíz "Source Packages" y dentro de un paquete ya existente.

## 6.1. Sentencia import

Para utilizar una clase que está en un paquete distinto a la clase que estamos utilizando, usamos la sentencia import. Por ejemplo, la clase Scanner se encuentra en el paquete java.util de la biblioteca de clases java. Podremos importar esa clase, o todas las clases del paquete:

```
import java.util.Scanner;
import java.util.*;
```

Esta sentencia debe aparecer después de la sentencia package, si existiese.

También se puede utilizar la clase sin la sentencia import, pero debemos indicar su ruta completa cada vez que queramos utilizarla:

```
java.util.Scanner teclado = new java.util.Scanner(System.in);
```

Trabajar con paquetes implica organizar directorios, compilar y ejecutar de cierta forma para que todo funcione bien.

## 6.2. Compilar y ejecutar clases con paquetes

Por ejemplo, para el archivo Bienvenida.java, que pertenece al paquete ejemplos, se debe crear el subdirectorio ejemplos y meter dentro el archivo.

Debemos prestar atención a las mayúsculas y minúsculas en las carpetas, utilizando el mismo nombre para todo.

Para compilar la clase Bienvenida.java debemos situarnos en el directorio padre del paquete y compilar desde ahí. Ejemplo estando en linux, teniendo la siguiente ruta:

/<directorio\_usuario>/Proyecto\_Bienvenida/ejemplos/Bienvenida.java

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida  
$ javac ejemplos/Bienvenida.java
```

Si todo va bien, en el directorio ejemplos nos aparece la clase compilada Bienvenida.class. El nombre del paquete es: paquete.clase, es decir: ejemplos. Bienvenida.

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida  
$ java ejemplos/Bienvenida  
Bienvenido a Java
```

### 6.3. Jerarquía de paquetes

Un paquete puede contener subpaquetes y así sucesivamente. Esto permite agrupar paquetes relacionados en un paquete más grande.

A nivel de sistema operativo, debemos crear subdirectorios con los nombres de los subpaquetes y meter dentro las clases que correspondan a cada una.

Para compilar, en el directorio del proyecto habría que compilar poniendo toda la ruta hasta llegar a la clase. Por ejemplo, teniendo el paquete "basicos" dentro del paquete ejemplos, la estructura de una clase HolaMundo dentro de este paquete sería: **ejemplos.basicos.HolaMundo** y la ruta en S.O. sería:

/<directorio\_usuario>/Proyecto\_Bienvenida/ejemplos/basicos/HolaMundo.java

Su compilación y ejecución sería:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida
```

```
$ javac ejemplos/basicos/HolaMundo.java  
$ java ejemplos/basicos/HolaMundo  
Hola Mundo
```

La biblioteca de clases de Java se organiza haciendo uso de esta jerarquía. Ejemplo, para la clase Date, la cual se encuentra dentro del paquete util que a su vez está dentro del paquete java.

```
import java.util.Date;
```

## 6.4. Librerías java

En el entorno de compilación y ejecución de Java, tenemos incluida la API de Java.

Utilizar las clases y métodos de esta biblioteca nos ayuda a reducir tiempo de desarrollo, es importante saber consultarla y conocer sus clases más utilizadas. Los paquetes más importantes son:

- **java.io.** Gestionan entrada y salida para manipular ficheros, leer o escribir en pantalla, en memoria, etc...
- **java.lang.** Contiene las clases básicas del lenguaje, no es necesario importarlo, ya que se importa automáticamente por el entorno de ejecución. En él se encuentra la clase Object, que es la raíz de jerarquía de clases en Java, o System, también se encuentran las clases que envuelven tipos primitivos de datos.
- **java.util.** De utilidad general para el programador, por ejemplo, la clase Scanner, la clase Date, etc...
- **java.math.** Herramientas de manipulaciones matemáticas
- **java.awt.** Clases relacionadas con la construcción de interfaces de usuario: ventanas, cajas de texto, botones...
- **java.swing.** Clases de construcción de interfaces avanzadas. Alternativa más potente que awt.
- **java.net.** Clases para programación en red local e internet.
- **java.sql** para programar acceso a bases de datos.
- **java.security** para implementar mecanismos de seguridad.

# 7. Programación de la consola: entrada y salida de la información

Los programas a veces necesitan acceder a recursos del sistema, como por ejemplo la entrada/salida estándar, para recoger datos del teclado y mostrarlos por pantalla.

La entrada por teclada y la salida por pantalla se realiza con la clase System del paquete java.lang.

Los atributos de System son tres objetos que se utilizan para entrada y salida estándar.

- System.in entrada estándar: teclado
- System.out salida estándar: pantalla
- System.err salida de error estándar, también por pantalla pero con un fichero distinto para distinguir la salida normal del programa de los mensajes de error y poder mostrarlos.

No se puede crear objetos con System, se utiliza directamente utilizando punto para llamar a sus métodos:

```
System.out.println("Bienvenido a Java");
```

## 7.1. Conceptos sobre la clase System

System.in es un atributo de la clase System, pero consultando la biblioteca de clases, observamos que es un objeto y como tal, debe ser instanciado, volviendo a consultar la biblioteca, System.in es una instancia de una clase de java llamada InputStream

### Field Summary

Fields	Modifier and Type	Field and Description
	static PrintStream	err The "standard" error output stream.
	static InputStream	in The "standard" input stream.
	static PrintStream	out The "standard" output stream.

## 7.2. Entrada por teclado. Clase System.

InputStream permite leer bytes, desde el teclado, un archivo o cualquier dispositivo de entrada. También podemos utilizar desde esta clase el método

`read()` que permite leer un byte de la entrada o `skip(long n)` que salta  $n$  bytes de la entrada.

Pero lo realmente interesante es poder leer texto o números y se utilizan estas clases:

- **InputStreamReader** convierte bytes leídos en caracteres y sirve para convertir el objeto `System.in` en otro tipo de objeto que permita leer caracteres
- **BufferedReader**. Lee hasta un fin de línea, la cual tiene un método (`readLine()`) que nos permite leer caracteres hasta el final de línea.

```
InputStreamReader isr = new InputStreamReader(System.in);
```

```
BufferedReader br = new BufferedReader (isr);
```

En el código anterior hemos creado un `InputStreamReader` a partir de `System.in` y pasamos dicho `InputStreamReader` al constructor de `BufferedReader`. El resultado es que las lecturas que hagamos con el objeto `br` son en realidad realizadas sobre `System.in`, pero con la ventaja de que podemos leer una línea completa. Así, por ejemplo, si escribimos una A, con:

```
String cadena = br.readLine();
```

Obtendremos en cadena una "A"

Sin embargo, seguimos necesitando hacer la conversión si queremos leer números. Por ejemplo, si escribimos un entero 32, en cadena obtendremos "32". Si recordamos, para convertir cadenas de texto a enteros se utiliza el método estático `parseInt()` de la clase `Integer`, con lo cual la lectura la haríamos así:

```
int numero = Integer.parseInt (br.readLine());
```

A continuación, ejemplo completo, en el que se incluye una excepción por si falla algo. Se captura esa excepción y avisamos al usuario de lo que ha pasado. Eso se realiza con `try {}`, colocando entre llaves el código que puede fallar y `catch {}` colocando el tratamiento de la excepción.

```

] import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/*
 * Ejemplo de entrada por teclado con la clase System
 */

] /**
 *
 * @author FMA
 */
public class entradateclado {
    public static void main(String[] args) {
        try {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);

            System.out.print("Introduce el texto: ");
            String cad = br.readLine();

            //salida por pantalla del texto introducido
            System.out.println(cad);

            System.out.print("Introduce un numero: ");
            int num = Integer.parseInt(br.readLine());

            // salida por pantalla del numero introducido
            System.out.println(num);

        } catch (Exception e) {
            // System.out.println("Error al leer datos");
            e.printStackTrace();
        }
    }
}

```

### 7.3. Entrada por teclado. Clase Scanner

La entrada y salida por System tiene el inconveniente de solo poder leer de manera fácil datos String. En cambio, la clase Scanner permite leer distintos tipos de datos: String, int, long, etc:

```

Scanner teclado = new Scanner (System.in);

int i = teclado.nextInt ();

```

O para una línea completa de texto, número o lo que sea:

```
String cadena = teclado.nextLine();
```

Ejemplo:

```
import java.util.Scanner;
/*
 * Ejemplo de entrada de teclado con la clase Scanner
 */

/**
 *
 * @author FMA
 */
public class EntradaTecladoScanner {

    public static void main(String[] args) {

        // Creamos objeto teclado
        Scanner teclado = new Scanner(System.in);

        // Declaramos variables a utilizar
        String nombre;
        int edad;
        boolean estudias;
        float salario;

        // Entrada de datos
        System.out.println("Nombre: ");
        nombre=teclado.nextLine();
        System.out.println("Edad: ");
        edad=teclado.nextInt();
        System.out.println("Estudios: ");
        estudias=teclado.nextBoolean();
        System.out.println("Salario: ");
        salario=teclado.nextFloat();

        // Salida de datos
        System.out.println("Bienvenido: " + nombre);
        System.out.println("Tienes: " + edad + " años");
        System.out.println("Estudios: " + estudias);
        System.out.println("Tu salario es: " + salario + " euros");
    }
}
```

## 7.4. Salida por pantalla

La salida por pantalla se hace con el objeto `System.out`. El cual es una instancia de la clase `PrintStream` del paquete `java.lang`. Entre sus métodos podemos destacar:

- **void print(String s):** escribe una cadena de texto.
- **void println(String x):** escribe una cadena de texto y termina la línea.

java.io

## Class PrintStream

java.lang.Object  
java.io.OutputStream  
java.io.FilterOutputStream  
java.io.PrintStream

All Implemented Interfaces:

Closeable, Flushable, Appendable, AutoCloseable

Direct Known Subclasses:

LogStream

- **void printf(String format, Object... args)** escribe una cadena de texto utilizando formato.

Para concatenar mensajes con valores de variables se utiliza el operador de concatenación +

```
System.out.Println("Bienvenido, " + nombre);
```

La orden printf() utiliza unos códigos de conversión para indicar el tipo que es.

Por ejemplo:

- %c escribe un carácter.
- %s escribe una cadena de texto.
- %d escribe un entero.
- %f escribe un número en punto flotante.
- %e escribe un número en punto flotante en notación científica.

Si por ejemplo queremos escribir el número float 12345.1684 con el punto de los miles y solo dos cifras sería:

```
System.out.printf("% ,.2f\n", 12345.1684);
```

Esta orden escribiría el número 12.345,17 por pantalla.

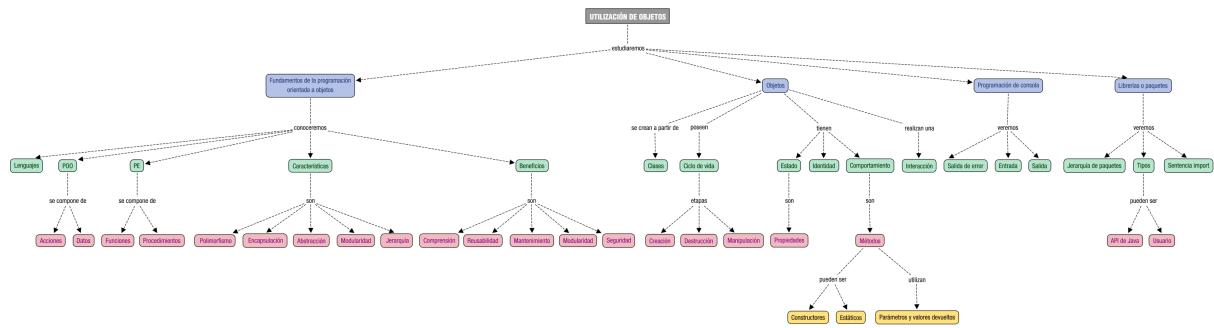
## 7.5. Salida de error.

Es una instancia de la clase PrintStream, así que podemos utilizar los mismos métodos.

En la consola de varios entornos integrados de desarrollo la salida de err se ve de un color distinta a la de out:

```
System.out.println("Salida estándar por pantalla");
System.err.println("Salida de error por pantalla");
```

## Mapa conceptual





# 4. Uso de estructuras de control.

	Autor	(X) Xerach Casanova
	Clase	Programación
	Fecha	@Dec 27, 2020 11:04 PM

- 1. Introducción
    - 2. Sentencias y bloques
  - 3. Estructuras de selección
    - 3.1. Estructura if / if - else
    - 3.2. Estructura Switch
  - 4. Estructuras de repetición
    - 4.1. Estructura for
    - 4.2. Estructura for/in
    - 4.2. Estructura while
    - 4.4. Estructura do-while
  - 5. Estructuras de salto
    - 5.1. Sentencias break y continue
    - 5.2. Sentencia return
  - 6. Excepciones
    - 6.2. Capturar una excepción
    - 6.2. El manejo de excepciones
    - 6.3. Delegación de excepciones con throws.
  - 7. Depuración de programas
  - 8. Documentación del código
    - 8.1. Etiquetas y posición
    - 8.2. Uso de las etiquetas
- Mapa conceptual

## 1. Introducción

La gran mayoría de lenguajes de programación poseen estructuras que permiten controlar el flujo de la información de los programas. Estas estructuras suelen ser comunes en todos y solo cambia su sintaxis.

Los tipos de estructura de programación que se emplean para controlar el flujo de datos son:

- **Secuencia:** compuestas por 0, 1 o N sentencias ejecutadas en el orden que se han escrito. Es la estructura más sencilla, sobre la que se construye el resto de estructuras.
- **Selección:** es un tipo de sentencia especial de decisión y de secuencias de instrucciones asociadas a ella. Según la evaluación de la sentencia (suele ser verdadero o falso), se ejecuta una secuencia de instrucciones u otras. Pueden ser simples, compuestas o múltiples.

- **Iteración:** sentencia especial de decisión y secuencia de instrucciones que se repiten según el resultado de la evaluación de la sentencia de decisión. La secuencia de instrucciones alojada dentro se ejecuta repetidamente si la sentencia de decisión arroja un valor correcto.
- **Sentencias de salto:** no son recomendables pero es necesario conocerlas.
- **Manejo de excepciones en Java:** para gestionar errores y situaciones especiales.

## 2. Sentencias y bloques

- Es necesario colocar las instrucciones o sentencias una detrás de otras en el orden exacto que deben ejecutarse.
- Cada sentencia debe estar escrita en una sola línea para que el código sea más legible y la localización de errores sea sencilla y rápida. Los errores en herramientas de programación se asocian a número de línea.
- Las sentencias pueden ocupar varias líneas del programa si son muy largas, pero siempre terminan en punto y coma.
- En algunas ocasiones, sobre todo en estructuras de control, no se utiliza punto y coma al final de la cabecera de la estructura.
- Existen sentencias nulas, solo contienen punto y coma y no hacen nada.
- Un bloque de sentencias es un conjunto de sentencias que se encierra en llaves y se ejecutan como una única orden. Agrupan sentencias y clarifican el código. Los bloques de sentencias se utilizan en casi todas las estructuras de control de flujo, clases, métodos, etc...
- En un bloque de sentencias, las instrucciones se deben colocar en un orden exacto o no, dependiendo de la situación.

Debemos tener en cuenta, a la hora de programar las siguientes normas en java:

- Declarar cada variable antes de utilizarla,
- Las variables declaradas dentro de un método, no se inicializan con un valor. Así que antes de trabajar con su valor, hay que dárselo. En el caso de las variables declaradas fuera de métodos se les da un valor inicial si el programador no se las da:
  - Numéricas: 0
  - Objetos: Null.
  - Booleanas: false
  - Char: ".

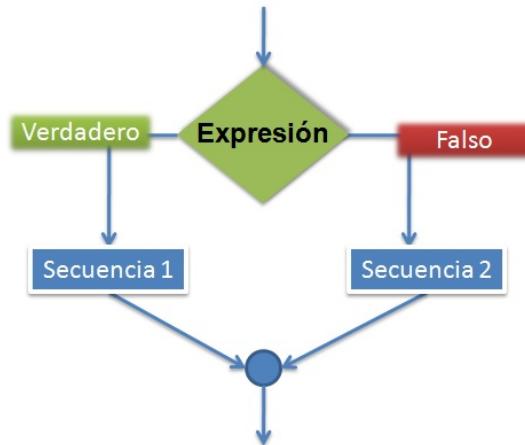
## 3. Estructuras de selección

Las estructuras de selección constan de una sentencia especial de decisión y un conjunto de secuencias de instrucciones. La sentencia de decisión es evaluada y devuelve valor verdadero o falso. En función de ese valor se ejecuta una secuencia de instrucciones u otras.

En C, verdadero o falso se representa mediante literal entero (0 para falso, 1 para true). En java se toman valores de true o false. Ojo: existe la clase Boolean y el tipo primitivo boolean.

La estructura de selección se divide en:

- simple: if.
- compuesta: if - else.
- basadas en el operador condicional.  
(condición) ? : valor verdadero : valor falso
- múltiples: switch.



### 3.1. Estructura if / if - else

Es una estructura condicional. En función del resultado se ejecuta una sentencia o bloque de estas.

**Estructura if simple.** Si la evaluación de la expresión lógica ofrece un resultado verdadero se ejecuta la sentencia o sentencias. Si es falso, no se ejecuta ninguna instrucción.

```
if (expresión-lógica) //al ser una sola instrucción, no son necesarias las llaves  
sentencia1;  
  
if (expresión-lógica)  
{  
    sentencia1;  
    sentencia2;  
    ...;  
    sentenciaN;  
}
```

**Estructura if de doble alternativa.** Si la evaluación de la expresión lógica ofrece resultado verdadero se ejecuta la sentencia o sentencias del bloque if. Si es falso se ejecuta la sentencia del bloque else

```
if (expresión-lógica) //al ser una sola instrucción, no son necesarias las llaves  
sentencia1;  
  
else  
    sentencia2;  
if (expresión-lógica)
```

```

{
    sentencia1;
    ...
    sentenciaN;
}

else
{
    sentencia1;
    ...
    sentenciaN;
}

```

La sentencia else no es obligatoria, pero sí es necesaria cuando hay que llevar a cabo alguna acción en caso de que la expresión lógica no se cumpla.

Las condiciones if e if - else, pueden anidarse, dentro de un bloque de sentencias if/if-else se puede incluirse otro if/if-else. Si el nivel de anidamiento es demasiado profundo puede provocar problemas de eficiencia y legibilidad y habría que plantearse elegir otro tipo de estructuras más adecuadas.

Cuando usamos anidamiento debemos poner especial atención en saber que estructura else está asociada a un cada if.

## 3.2. Estructura Switch

Es un tipo de selección múltiple, ideal si nuestro programa debe elegir entre más de dos alternativas.

```

switch (expresion) {

    case valor1:
        sentencia1_1;
        sentencia1_2;
        ...
        break;

        ...
        ...

    case valorN:
        sentenciaN_1;
        sentenciaN_2;
        ...
}

```

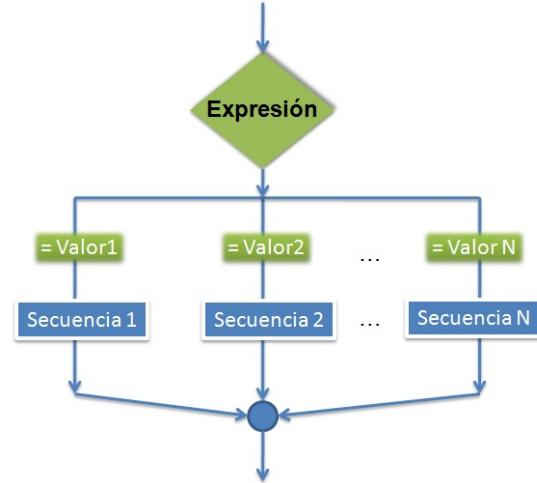
```

        break;

default:
    sentencias-default;
}

```

- La expresión debe ser char, byte, short o int. Las constantes de cada case deben ser del mismo tipo o compatible.
- La expresión va entre paréntesis.
- Cada case lleva asociado un valor y finaliza con dos puntos.
- El bloque de sentencias asociado a la cláusula default puede finalizar con la sentencia de ruptura break o no.



## 4. Estructuras de repetición

La función de estas estructuras es repetir la ejecución de una serie de instrucciones teniendo en cuenta una condición.

En Java existen cuatro tipos de estructuras de repetición, bucles o estructuras iterativas:

- Bucle for (repite para) - Controlado por contador.
- Bucle for/in (repite para cada) - Controlado por contador.
- Bucle While (repite mientras) - Controlado por sucesos.
- Bucle Do While (repite hasta) - Controlado por sucesos.

El uso de estos cuatro bucles depende de:

- Si sabemos cuantas veces necesitamos repetir un conjunto de instrucciones.
- Si hemos de repetir un conjunto de instrucciones si una condición satisface un conjunto de valores.
- Si sabemos hasta cuando debemos repetir un conjunto de instrucciones.
- Si sabemos si hemos de estar repitiendo un conjunto de instrucciones mientras se cumpla una condición.

Algunos de estos bucles son equivalentes entre sí y se puede solucionar el mismo problema utilizando distintos tipos de bucles.

### 4.1. Estructura for

Bucle controlado por contador. Tiene las siguientes características:

- Se ejecuta un número determinado de veces.
- Utiliza una variable contadora que controla las iteraciones.

Existen tres operaciones que se llevan a cabo en los bucles for, que se ven en el código siguiente...



```
for (inicialización; condición; iteración)

    sentencia; //Con una sola instrucción no es necesario utilizar llaves
for (inicialización; condición; iteración)
{
    sentencia1;
    sentencia2;
    ...
    sentenciaN;
}
```

- **Inicialización** de la variable que se encarga de controlar el final del bucle
- **Condición**. Una expresión que evaluará la variable de control, mientras sea falsa, se repetirá el cuerpo del bucle. Cuando se cumpla, terminará la ejecución.
- **Iteración**. Indica la manera en que la variable de control va cambiado en cada iteración (incremento o decremento y no solo de uno en uno).

## 4.2. Estructura for/in

Es una mejora incorporada en la versión 5.0 de java.

Permite realizar recorridos sobre arrays y colecciones de objetos. Se les llama también bucle for mejorado o bucle foreach.

```
for (declaración: expresión) {
    sentencia1;
    ...
}
```

```
    sentenciaN;  
}  
}
```

- Donde expresión es un array o colección de objetos.
- Donde declaración es una declaración de una variable cuyo tipo sea compatible con la expresión. Normalmente será el tipo y el nombre de la variable a declarar.

Para cada elemento de la expresión, se guarda el elemento en la variable declarada y realiza las instrucciones del bucle. Después, en cada una de las iteraciones del bucle tendremos la variable declarada en el elemento actual de la expresión. El bucle acabará cuando se termine de recorrer el array o la colección de objetos.

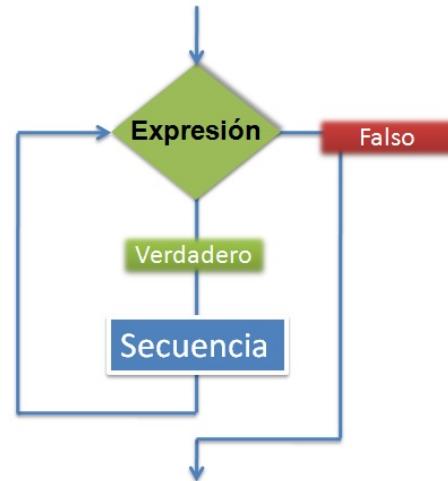
Permiten al programador despreocuparse del número de veces que se ha de iterar, pero no sabremos en qué iteración estamos, salvo que añadamos una variable contadora extra.

## 4.2. Estructura while

Es una estructura de repetición controlada por sucesos.

Su característica principal es que las instrucciones contenidas dentro de él se ejecutan al menos una vez, si la condición inicial a evaluar es verdadera. Si la evaluación inicial es falsa el cuerpo del bucle no se ejecuta.

Dentro del bucle debe realizarse alguna acción que modifique la condición que controla su ejecución, para no ejecutar un bucle infinito.



```
while (condición)  
    sentencia;  
    while (condición) {  
        sentencia1;  
        ...  
        sentenciaN;  
    }
```

La condición del bucle siempre se evalúa al principio, por tanto las instrucciones contenidas pueden no llegarse a ejecutar nunca en caso de que sea falsa.

## 4.4. Estructura do-while

Funciona exactamente igual que el while, pero la característica fundamental de esta estructura es que las instrucciones contenidas se ejecutarán al menos una vez y, a partir de ahí, se repetirá su ejecución hasta que la condición sea verdadera. Sigue siendo imprescindible realizar alguna acción que modifique la condición que controla su ejecución para no entrar en un bucle infinito.

```

do
    sentencia; //Con una sola sentencia no son necesarias las llaves

while (condición);

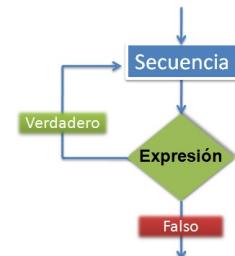
do
{
    sentencia1;

    ...
    sentenciaN;

}
while (condición);

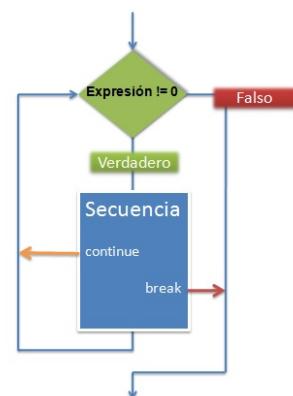
```

La condición siempre se evaluará después de una primera ejecución del bucle, por lo que siempre se ejecutarán las instrucciones contenidas en él, al menos una vez.



## 5. Estructuras de salto

Se desaconseja su uso porque pueden provocar mala estructuración de código e incremento de dificultad de mantenimiento. Java incorpora ciertas sentencias que es necesario conocer por ser útiles en algunas partes del programa: break, continue y return.



### 5.1. Sentencias break y continue

Instrucciones que permiten modificar el comportamiento de otras estructuras o sentencias de control al estar incluidas en algún punto de la secuencia.

**La sentencia break** incide en estructuras switch, while, for y do-while:

- Apareciendo dentro de una secuencia de instrucciones , la estructura se termina inmediatamente.
- Si aparece dentro de un bucle anidado finaliza solo la sentencia de la iteración más interna.

```
6  public class sentencia_break {  
7  public static void main(String[] args) {  
8      // Declaración de variables  
9      int contador;  
10  
11      //Procesamiento y salida de información  
12  
13      for (contador=1;contador<=10;contador++)  
14      {  
15          if (contador==7)  
16              break;  
17          System.out.println ("Valor: " + contador);  
18      }  
19      System.out.println ("Fin del programa");  
20      /* El bucle sólo se ejecutará en 6 ocasiones, ya que cuando  
21      * la variable contador sea igual a 7 encontraremos un break que  
22      * romperá el flujo del bucle, transfiriéndonos a la sentencia que  
23      * imprime el mensaje de Fin del programa.  
24      */  
25  }  
26 }
```

**La sentencia continue** incide sobre while, for y do-while.

- Si aparece una sentencia continue dentro de las sentencias, dicha sentencia dará por terminada la iteración actual y ejecutará una nueva.
- Si aparece en el interior de un bucle anidado solo afectará a la iteración más interna.

```
4  * Uso de la sentencia continue  
5  */  
6  public class sentencia_continue {  
7  public static void main(String[] args) {  
8      // Declaración de variables  
9      int contador;  
10  
11      System.out.println ("Imprimiendo los números pares que hay del 1 al 10... ");  
12      //Procesamiento y salida de información  
13  
14      for (contador=1;contador<=10;contador++)  
15      {  
16          if (contador % 2 != 0) continue;  
17          System.out.print(contador + " ");  
18      }  
19      System.out.println ("\nFin del programa");  
20      /* Las iteraciones del bucle que generarán la impresión de cada uno  
21      * de los números pares, serán aquellas en las que el resultado de  
22      * calcular el resto de la división entre 2 de cada valor de la variable  
23      * contador, sea igual a 0.  
24      */  
25  }  
26  
27  
28 }
```

## 5.2. Sentencia return

Modifican la ejecución de un método y se puede utilizar de dos formas:

- Para terminar la ejecución del método donde esté escrita, transfiriendo el control al punto desde donde se hizo la llamada.
- Para devolver o retornar un valor que se incluye junto a return.

Es posible utilizar return en cualquier punto del método, pero suelen aparecer al final de un método, además es lo más recomendable.

## 6. Excepciones

Las excepciones son errores no sintácticos, que se generan en tiempo de ejecución. Se deben manejar este tipo de excepciones adecuadamente. El manejo de ellos radica en:

- Que el código que se encarga de manejar errores es perfectamente identificable. Además puede estar separado del código de la aplicación.
- Que Java tiene una gran cantidad de errores estándar asociados a multitud de fallos comunes: divisiones por cero, fallo de entrada de datos, etc. y podemos gestionarlas de manera específica.

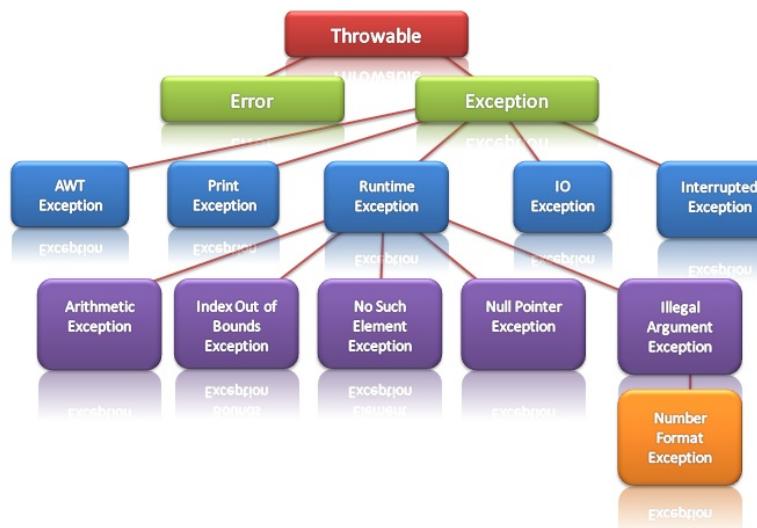
En Java se pueden preparar fragmentos de código susceptibles de provocar errores de ejecución, para ser lanzados hacia otras zonas creadas previamente para tratar dichas excepciones (throw). Si estas excepciones no se tratan, el programa probablemente se detendrá.

Las excepciones están representadas por clases. El paquete `java.lang.Exception` y sus subpaquetes contienen todos los tipos de excepciones. Todas derivan de la clase `Throwable`, como por ejemplo `Error` y `Exception`.

Cuando se produce un Error, Java genera un objeto asociado a esa excepción, este objeto es de la clase `Exception` o alguna de sus herederas, el cual se pasa al código que lo maneja. Este código puede manipular las propiedades del objeto `Exception`.

También podemos lanzar nuestras propias excepciones, derivadas de la clase `Exception`. Estas clases derivadas se ubican en dos grupos:

- Excepciones en tiempo de ejecución. Cuando el programador no ha tenido cuidado al escribir su código.
- Excepciones que indican que ha sucedido algo inesperado o fuera de control.



## 6.2. Capturar una excepción

Para ello se emplea la estructura de captura de excepciones try - catch - finally.

Se declaran bloques de código donde es posible que ocurra una excepción con try. Con catch capturamos las excepciones y las manejamos.

```
try {  
    código que puede generar excepciones;  
}  
catch (Tipo_excepcion_1 objeto_excepcion) {  
    Manejo de excepción de Tipo_excepcion_1;  
}  
catch (Tipo_excepcion_2 objeto_excepcion) {  
    Manejo de excepción de Tipo_excepcion_2;  
}  
...  
finally {  
    instrucciones que se ejecutan siempre  
}
```

La instrucción finally es opcional y solo puede aparecer una vez.

Cada catch maneja un tipo de excepción. Cuando se produce una excepción, se busca el catch que posea el manejador adecuado y será el que utilice el mismo tipo de excepción producida. Al ser la clase Exception superclase de todas, se pueden dar problemas.

El último catch debe ser el que capture excepciones genéricas y los primeros deben ser más específicos. Si se van a tratar todas las excepciones, entonces basta con un solo catch que capture objetos.

## 6.2. El manejo de excepciones

Se pueden tratar de dos formas:

- **Interrupción.** Se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la excepción se anula y no hay manera de regresar al código que la provocó, por tanto esa operación se anula.
- **Reanudación.** Se puede manejar el error y regresar de nuevo al código que lo provocó.

Java emplea la primera forma, pero se puede simular una segunda mediante un bloque try con un while en su interior. Repetirá hasta que el error deje de existir.

```

7 public static void main(String[] args){
8     boolean fueraDelimites=true;
9     int i; //Entero que tomará valores aleatorios de 0 a 9
10    String texto[] = {"uno","dos","tre","cuatro","cinco"}; //String que representa la moneda
11
12    while(fueradelimites){
13        try{
14            i= (int) Math.round(Math.random()*9); //Generamos un indice aleatorio
15            System.out.println(texto[i]);
16            fueraDelimites=false;
17        }catch(ArrayIndexOutOfBoundsException exc){
18            System.out.println("Fallo en el indice");
19        }
20    }
21}
22}
23}

```

En este código se generan números aleatorios almacenados en la variable i. Seguidamente ese valor se utiliza en el índice del array y si el valor es mayor al tamaño del mismo, se genera un error del tipo `ArrayIndexOutOfBoundsException`, el cual gestionamos con un catch. El código se repetirá gracias al while.

### 6.3. Delegación de excepciones con throws.

Cuando se produce una excepción es necesario saber quien es el encargado de solucionarla. Puede ser que sea el mismo método llamado o el código que hizo la llamada.

Cuando un método utiliza una sentencia que puede generar una excepción, pero esa excepción no es capturada y tratada por él sino que se encarga de su gestión quien llamó al método, se trata de una delegación de excepciones.

```

public class delegacion_excepciones {

    ...

    public int leeAño(BufferedReader lector) throws IOException, NumberFormatException{

        String linea = teclado.readLine();

        Return Integer.parseInt(linea);

    }

    ...
}

}

```

`IOException` y `NumberFormatException` son dos posibles excepciones del método `leeAño`, pero no las gestiona, teniendo en su cabecera un listado de excepciones que puede lanzar, separadas por comas.

## 7. Depuración de programas

Durante la etapa de implementación, los programadores deben realizar otro tipo de pruebas que verifiquen el código que escriben, a parte de los de Caja Blanca y Caja Negra.

La depuración de programas es el proceso en el cual se identifican y corrigen errores (debugging). Hay tres etapas por las que pasa un programa cuando se desarrolla y que puede generar errores:

- **Compilación.** Una vez hemos terminado un programa, solemos pasar cierto tiempo eliminando errores de compilación. El compilador de java puede sacar a la luz errores que nos e aprecian a simple vista.
- **Enlazado.** Se trata de errores que son lanzados en la etapa de compilación si se encuentran librerías que han sido mal enlazadas, por ejemplo por utilizar parámetros incorrectos en tipo y número, o si esos métodos no existen.
- **Ejecución.** Es frecuente que un programa en ejecución no funcione como esperaba. Algunos errores serán detectados automáticamente y otros simplemente darán comportamientos no esperados (bugs). Al ser errores poco claros hay que recurrir a labores de investigación para encontrar la causa.

El proceso de depuración se hace a través del entorno de desarrollo que estemos utilizando. Es una herramienta que nos ayudar a eliminar posibles errores. Podemos utilizar depuradores simples como el jdb de Java o bien utilizar un depurador existente en nuestro IDE.

Los elementos que utiliza un depurador son:

- **Breakpoints o puntos de ruptura.** Son determinados por el programador a lo largo del código fuente. La ejecución del programa se detiene en estos puntos y el depurador muestra los valores de las variables tal y como están en ese momento. Se analizan los valores que tienen y el valor que deberían tener para recoger información importante sobre el proceso de depuración.
- **Ejecución paso a paso.** Podemos ejecutar el programa línea por línea, siguiendo el proceso de ejecución y supervisar el funcionamiento del mismo. El debugger ofrece la posibilidad de no entrar en métodos que no queramos en una ejecución paso a paso.

## 8. Documentación del código

Documentar nuestro código es necesario para facilitar su mantenimiento y reutilización, debemos documentar obligatoriamente: clases, paquetes, constructores, métodos y atributos. Opcionalmente: bucles, partes de algoritmos,...

Se puede generar documentación de nuestro código a través de la herramienta Javadoc.

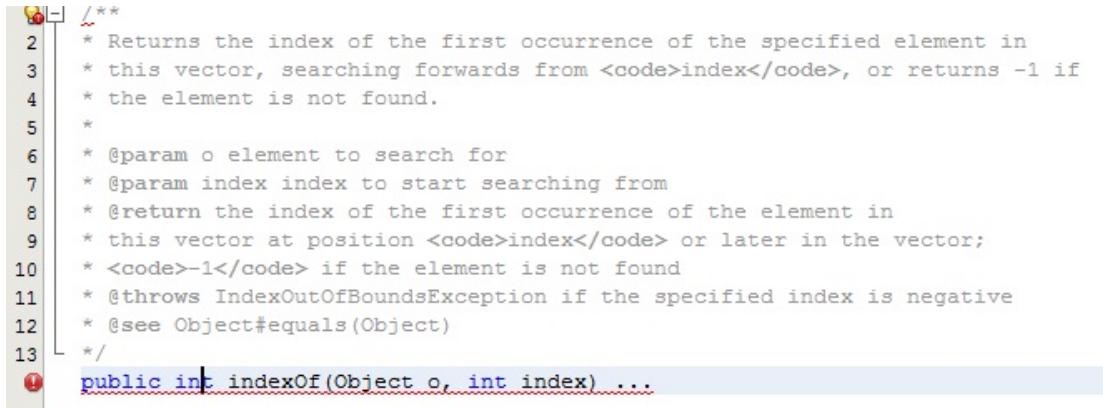
La documentación se realiza a través de unos comentarios especiales llamados comentarios de documentación, que serán tomados por Javadoc para generar archivos html que permiten posteriormente navegar por cualquier documentación con cualquier navegador web.

Estos comentarios comienzan por `/**` y terminan en `*/`. En su interior podemos encontrar dos partes diferenciadas:

```
/**
 * Descripción principal (texto/HTML)
 *
 * Etiquetas (texto/HTML)
 */
```

- Zona de descripción. En ella se escribe un comentario sobre la clase, atributo, constructor o método, se puede incluir cualquier cantidad de texto, e incluso etiquetas html que le den formato.

- **Zona de etiquetas:** Se coloca un conjunto de etiquetas a las que se asocian textos, cada etiqueta tiene un significado especial y aparece en lugares determinados de la documentación una vez generada.



```

1 /**
2  * Returns the index of the first occurrence of the specified element in
3  * this vector, searching forwards from <code>index</code>, or returns -1 if
4  * the element is not found.
5  *
6  * @param o element to search for
7  * @param index index to start searching from
8  * @return the index of the first occurrence of the element in
9  * this vector at position <code>index</code> or later in the vector;
10 * <code>-1</code> if the element is not found
11 * @throws IndexOutOfBoundsException if the specified index is negative
12 * @see Object#equals(Object)
13 */
14 public int indexOf(Object o, int index) ...

```

## 8.1. Etiquetas y posición

Existen dos tipos de etiqueta:

- **Etiquetas de bloque:** son etiquetas que solo pueden ser incluidas en el bloque de documentación, después de la descripción principal y comienzan con @.
- **Etiquetas de texto:** son etiquetas que se ponen en cualquier punto de la descripción o en cualquier parte de la documentación asociada a una etiqueta de bloque. Se definen entre llaves.

	@autor	{@code}	{@docRoot}	@deprecated	@exception	{@inheritDoc}	{@link}	{@literal}
<b>Descripción</b>	✓		✓	✓			✓	✓
<b>Paquete</b>	✓		✓	✓			✓	✓
<b>Clases e Interfaces</b>	✓		✓	✓			✓	✓
<b>Atributos</b>			✓	✓			✓	✓
<b>Constructores y métodos</b>			✓	✓	✓	✓	✓	

## 8.2. Uso de las etiquetas

Las más habituales son:

- **@autor** texto con el nombre: Se admite en clases e interfaces. No necesita formato especial, podemos incluir tantas como queramos.
- **@version** texto de la versión: El texto de la versión no necesita formato especial. Es conveniente incluir versión y la fecha. Podemos incluir varias etiquetas una detrás de otra.
- **@deprecated** texto: Indica que no debería utilizarse, indicando en el texto las causas. Se puede utilizar en todos los apartados de la documentación y se puede añadir lo que se puede utilizar en su lugar.

```
@deprecated El método no funciona correctamente. Se recomienda el uso de {@link metodoCorrecto}
```

- **@exception nombre-excepción texto:** Esta etiqueta es equivalente a **@throws**
- **@param nombre-atributo texto:** es aplicable a parámetros de constructores y métodos, describe los parámetros del constructor o método. El nombre del atributo es igual al del parámetro y seguidamente lleva una descripción.

```
@param fromIndex: El índice del primer elemento que debe ser eliminado.
```

- **@return nombre texto:** se puede omitir en métodos void, se describe explícitamente que tipo o clase de valor devuelve y sus posibles rangos de valores.

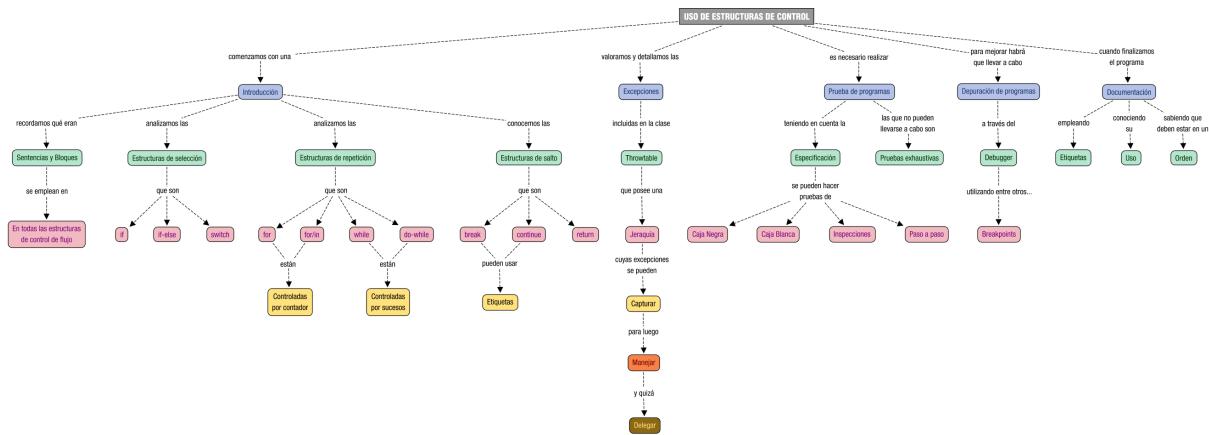
```
/**  
 * Chequea si un vector no contiene elementos.  
 *  
 * @return <code>verdadero</code>si solo si este vector no contiene componentes, esto es, su tamaño es cero;  
 * <code>falso</code> en cualquier otro caso.  
 */  
public boolean VectorVacio() {  
    return elementCount == 0;  
}
```

- **@see referencia:** se aplica a clases, interfaces, constructores, métodos, atributos y paquetes. Añade enlaces de referencia a otras partes de la documentación.

```
* @see "Diseño de patrones: La reusabilidad de los elementos de la programación orientada a objetos"  
* @see <a href="http://www.w3.org/WAI/">Web Accessibility Initiative</a>  
* @see String#equals(Object) equals
```

- **@throws nombre-excepción texto:** se indica el nombre completo de la excepción, se puede añadir una etiqueta por cada excepción que se lance por la cláusula throws, siguiendo orden alfabético. Aplicable en constructores y métodos, describiendo las posibles excepciones.

## Mapa conceptual





# 5. Desarrollo de clases.

	Autor	(X) Xerach Casanova
	Clase	Programación
	Fecha	@Jan 10, 2021 3:54 PM

## 1. Concepto de clase

[1.1. Repaso del concepto de objeto](#)

[1.2. El concepto de clase](#)

[2. Estructura y miembros de una clase](#)

[2.1. Declaración de una clase.](#)

[2.2. Cabecera de una clase](#)

[2.3. Cuerpo de una clase](#)

[2.4. Métodos estáticos o de clase](#)

## 3. Atributos

[3.1. Declaración de atributos](#)

[3.2. Modificadores de acceso](#)

[3.3. Modificadores de contenido](#)

[3.4. Atributos estáticos](#)

## 4. Métodos

[4.1. Declaración de un método](#)

[4.2. Cabecera del método](#)

[4.3. Modificadores en la declaración de un método](#)

[4.4. Parámetros en un método](#)

[4.5. Cuerpo de un método](#)

[4.6. Sobrecarga de métodos](#)

[4.7. Sobrecarga de operadores](#)

[4.8. La referencia this](#)

[4.9. Métodos estáticos](#)

## 5. Encapsulación, control de acceso y visibilidad

[5.1. Ocultación de atributos. Métodos de acceso](#)

[5.2. Ocultación de métodos](#)

[6. Utilización de los métodos y atributos de una clase](#)

[6.1. Declaración de un objeto](#)

[6.2. Creación de un objeto](#)

[6.3. Manipulación de un objeto: utilización de métodos y atributos](#)

## 7. Constructores

### 7.1. Concepto de constructor

[7.2. Creación de constructores](#)

[7.3. Utilización de constructores](#)

[7.4. Constructores de copia](#)

[7.5. Destrucción de objetos](#)

## 8. Empaquetado de clases

[8.1. Jerarquía de paquetes](#)

[8.2. Utilización de paquetes](#)

[8.3. Inclusión de una clase en un paquete](#)

[8.4. Proceso de creación de un paquete](#)

[Mapa conceptual](#)

## 1. Concepto de clase

Las clases están compuestas por atributos y métodos. Una clase especifica las características comunes de un conjunto de objetos.

Los programas a su vez están formados por un conjunto de clases, a partir de las cuales se irán creando objetos que interactúan unos con otros.

## 1.1. Repaso del concepto de objeto

Las **características fundamentales** de un objeto son: la **identidad** (los objetos son únicos y distinguibles entre sí, aunque pueda haber objetos exactamente iguales), el **estado** (los atributos que describen al objeto y los valores que tienen en cada momento) y el **comportamiento** (acciones que se pueden realizar sobre el objeto).

Los objetos tienen determinadas posibilidades de comportamientos (acciones) dependiendo de la familia a la que pertenezcan (un coche puede frenar, arrancar, cambiar de marcha, etc..., un círculo puede plantear acciones como calcular longitud de circunferencia, superficie, etc...).

También tienen sus atributos, cuyos valores cambian en función de las acciones que se realizan (en el caso del coche, ubicación o coordenadas, velocidad instantánea, kms. recorridos, etc..., en el caso de un círculo, tamaño del radio, color, etc...).

Todo objeto, por tanto, puede ser cualquier cosa que se pueda describir en término de atributos y acciones.

Un objeto no es más que la representación de cualquier entidad concreta o abstracta que puedes percibir o imaginar y que pueda resultar de utilidad para modelar los elementos del entorno del problema que deseas resolver.

## 1.2. El concepto de clase

La idea de una clase la podemos catalogar como una plantilla o modelo para cada conjunto de objetos que sean del mismo tipo, es decir, que tengan los mismos atributos y un comportamiento similar.

Una clase consiste en la definición de un tipo de objeto. Se trata de una descripción detallada de cómo van a ser los objetos que pertenezcan a esa clase, indicando qué tipo de información contendrán (atributos) y cómo se podrá interactuar con ellos (comportamiento).

Para ello se especifican:

- Atributos: comunes a todos los métodos que pertenezcan a esa clase.
- Métodos: que permiten interactuar con los objetos.

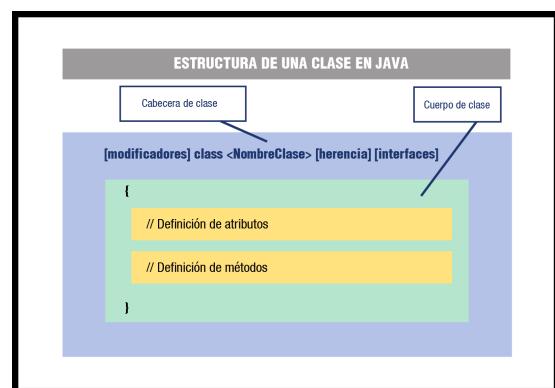
Es común confundir los términos de clase y objeto, aunque el contexto nos permite distinguir si nos referimos a una clase (definición abstracta) o a un objeto (instancia concreta de esa clase).

## 2. Estructura y miembros de una clase

La estructura de una clase consiste en:

**Cabecera:** Compuesta por una serie de modificadores de acceso, la palabra reservada class y el nombre de la clase.

**Cuerpo:** o contenido. se especifican o declaran los distintos miembros de la clase: atributos que caracterizan a los objetos y métodos que determinan el comportamiento de los mismos.



### 2.1. Declaración de una clase.

Estructura general:

```
[modificadores] class <NombreClase> [herencia] [interfaces] { // Cabecera de la clase
    // Cuerpo de la clase
    Declaración de los atributos
    Declaración de los métodos
}

/*
 *
 * Ejemplo de clase Punto 2D
 */
class Punto {
    // Atributos
    int x,y;

    // Métodos
    int obtenerX () { return x; }
    int obtenerY() { return y;}
    void establecerX (int vx) { x= vx; }
    void establecerY (int vy) { y= vy; }
}
```

Todos los distintos programas desarrollados en unidades anteriores son una clase java: se declaraban con la palabra reservada class y contenían atributos (variables) y métodos (como mínimo el método main).

Además de indicar en la cabecera de una clase el nombre de la clase y la palabra reservada class, también podemos proporcionar más información mediante modificadores y otros indicadores como el nombre de su **superclase** (si esta clase hereda de otra), o si implementa algún **interfaz**.

Cuando se implementa una clase se debe tener en cuenta:

- Por convenio los nombres de clase empiezan por letra mayúscula, si el nombre de la clase está formado por varias palabras, también tendrán su primera letra mayúscula: Coche, JugadorFutbol...
- El archivo en el que se encuentra una clase java debe tener el mismo nombre que esa clase si queremos utilizarla desde otras clases (clase principal del archivo).
- La definición e implementación de una clase se incluye en el mismo archivo .java. En otros lenguajes como C++ podrían ir separados en archivos con extensiones .h y .cpp.

## 2.2. Cabecera de una clase

La declaración de una clase puede incluir los siguientes elementos en el siguiente orden:

1. Modificadores (public, abstract, final...).
2. Nombre de la clase.
3. Nombre de su clase padre (superclase), se especifica con la palabra reservada extends (extiende o hereda de).
4. Lista separada por comas de interfaces que son implementadas por la clase con la palabra implements (implementa).
5. Cuerpo de la clase encerrado entre llaves {}

Los modificadores pueden ser:

**public:** Indica que la clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase (desde otra parte del programa). Si no se especifica ese modificador, la clase solo podrá ser utilizada desde clases del mismo paquete. Solo puede haber una clase public en un archivo .java. El resto de clases que se definan no pueden ser públicas.

**abstract:** Indica que es una clase abstracta no instanciable. No es posible crear objetos de esa clase, habrá que utilizar clases que hereden de ella.

**final:** Indica que no podrás crear clases que hereden de ellas. final y abstract son excluyentes (o se utiliza uno o se utiliza el otro).

## 2.3. Cuerpo de una clase

Una clase puede no contener en su declaración atributos o métodos, pero debe contener al menos uno de los dos (la clase no puede estar vacía).

En el ejemplo de la clase Punto teníamos los atributos x e y de tipo int. Cualquier objeto de la clase punto contendrá dichos valores, que pueden coincidir o no en el contenido. Por ejemplo, declarando 3 objetos de tipo punto:

```
Punto p1, p2, p3;
```

Cada uno de ellos contendrá un par de coordenadas, puede que esos valores coincidan con los de otros objetos de tipo punto o no, pero todos han sido creado a partir de la misma clase.

También se definían métodos:

```
int obtenerX () { return x; } //devuelve el contenido de x  
int obtenerY() { return y;} //devuelve el contenido de y  
void establecerX (int vx) { x= vx; } //cambia el contenido de x a partir del parámetro que se le pasa  
void establecerY (int vy) { y= vy; } //cambia el contenido de y a partir del parámetro que se le pasa
```

Estos métodos se pueden llamar desde cualquier objeto que sea instancia de Punto. Se trata de operaciones que permiten manipular los atributos contenidos en el objeto.

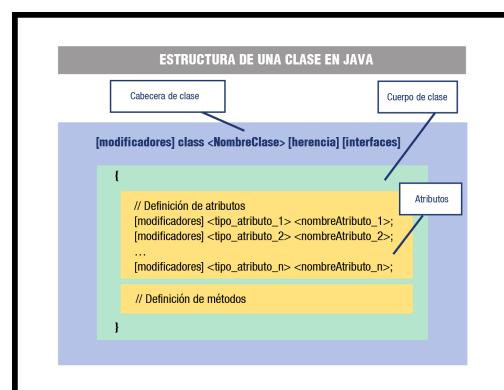
## 2.4. Métodos estáticos o de clase

Cada vez que instanciamos una clase se desencadenan una serie de procesos (se construye el objeto) que da lugar a la creación en memoria de un espacio físico que construye el objeto creado.

Por otro lado hay algunos miembros de la clase que no tienen sentido como partes del objeto, sino más bien como partes de la clase en sí. Estos miembros tienen sentido y existencia al margen de la existencia de cualquier objeto de dicha clase. Se trata de miembros estáticos o de clase y se definen con el modificador static, tanto en atributos como en métodos.

## 3. Atributos

Constituyen la estructura interna de los objetos de una clase. Es el conjunto de datos que los objetos de una determinada clase almacenan cuando son creados. Son variables cuyo ámbito de existencia es el objeto donde han sido creadas. Fuera del objeto no tienen sentido y si el objeto deja de existir, las variables también lo harán. Estos atributos son conocidos con el nombre de variables miembro o variables de objeto.



La declaración de un atributo puede contener algunos modificadores como: public, private, protected o static. Lo más normal es declarar todos o la mayoría de los atributos como privados para respetar el concepto de encapsulación. De manera que para manipular algún atributo solo se pueda realizar a través de los métodos de la clase.

### 3.1. Declaración de atributos

La sintaxis general es:

```
[modificadores] <tipo> <nombreAtributo>;
```

Ejemplos:

```
int x;  
  
public int elementoX, elementoY;  
  
private int x, y, z;  
  
static double descuentoGeneral;  
  
final boolean casado;
```

Cada vez que se crea un objeto, se crean tantas variables como atributos contenga ese objeto en su interior, definidas en la clase. Todas esas variables son encapsuladas dentro del objeto y solo tienen sentido dentro de él.

Dentro de la declaración de atributo encontramos:

- **Modificadores:** palabras reservadas que permiten modificar la utilización del atributo.
- **Tipo:** tipo de atributo, puede ser primitivo o referenciado.
- **Nombre.** Identificador único para el nombre del atributo, por convenio se utilizan minúsculas, en caso de varias palabras, se utiliza el primer carácter de cada palabra mayúscula a partir de la segunda palabra: nombre, nombreCandidato...

Los modificadores de un atributo pueden ser:

- **Modificadores de acceso:** son excluyentes entre sí e indican la forma de acceso al atributo de clase: private, protected, public
- **Modificadores de contenido:** No son excluyente y pueden aparecer varios a la vez: static, final.
- **Otros modificadores:** transient y volatile. Transient es para declarar atributos transitorios y volatile es para indicar al compilador que no se deben realizar optimizaciones sobre esa variable.

```
[private | protected | public] [static] [final] [transient] [volatile] <tipo> <nombreAtributo>;
```

### 3.2. Modificadores de acceso

Los modificadores de acceso son excluyentes. Para un atributo son:

- **Modificador de acceso por omisión (o de paquete).** Se permite el acceso a este atributo desde todas las clases del mismo paquete (package) que esta clase. No es necesario escribir ninguna palabra reservada.
- **public:** cualquier clase tiene acceso al atributo. No es habitual declarar atributos públicos (public).
- **private:** Solo se puede acceder al atributo dentro de la propia clase, el cual está oculto para cualquier otra zona de código fuera de la clase. Se recomienda que los atributos de una clase se declaren privados o

tengan acceso de paquete para garantizar la encapsulación.

- **protected:** Se permite acceder al atributo desde cualquier subclase de la clase en la que se encuentre declarado el atributo y también desde las clases del mismo paquete.

Cuadro de niveles accesibilidad a los atributos de una clase.

	Misma clase	Subclase	Mismo paquete	Otro paquete
Sin modificador (paquete)	Sí		Sí	
<code>public</code>	Sí	Sí	Sí	Sí
<code>private</code>	Sí			
<code>protected</code>	Sí	Sí	Sí	

### 3.3. Modificadores de contenido

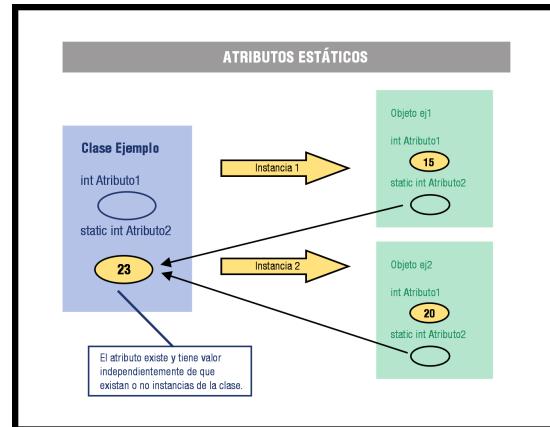
Son no excluyentes y son los siguientes:

- **static:** convierte al atributo en común para todos los objetos de una misma clase. Todas las instancias de esa clase compartirán ese atributo con ese mismo valor.
- **final.** Indica que el atributo es una constante, su valor no puede ser modificado a lo largo de la vida del objeto. Por convenio se declaran con todas las letras en mayúsculas. Ejemplo:

```
class claseGeometria {  
    // Declaración de constantes  
    public final float PI= 3.14159265;
```

### 3.4. Atributos estáticos

En el caso de los atributos estáticos, su existencia depende de la propia clase y no de los objetos, por lo tanto, solo habrá uno, independientemente del número de objetos que se creen. El atributo siempre será el mismo en todos los objetos y tendrá el mismo valor. Aunque no existan objetos de esa clase, el atributo si existe y puede contener un valor.



Un ejemplo sería un contador que indica el número de objetos de esa clase que se van creando durante la ejecución del programa, también habrá que implementar el código necesario para implementar el valor del atributo contador cada vez que se crea un objeto. Esto se realiza en el constructor. Otro ejemplo sería mantener un String con el nombre de la clase

```
class Punto {  
    // Coordenadas del punto
```

```

private int x, y;
// Atributos de clase: cantidad de puntos creados hasta el momento
public static cantidadPuntos;
public static final nombre;

```

## 4. Métodos

Los métodos nos sirven para definir el comportamiento del objeto, forman parte de la estructura interna del objeto junto con los atributos.

Se suelen declarar después de los atributos, aunque atributos y métodos pueden aparecer mezclados en el cuerpo, pero es aconsejable no hacerlo para mejorar la claridad y legibilidad del código.

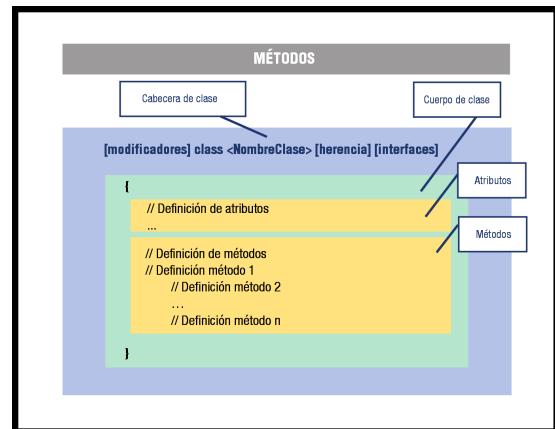
Los métodos representan la interfaz de una clase. Son la forma que tienen otros objetos de comunicarse con ellos, solicitando información o pidiendo que se lleven a cabo acciones determinadas.

### 4.1. Declaración de un método

Se compone de dos partes:

- **Cabecera.** Contiene el nombre del método junto con el tipo devuelto, un conjunto de posibles modificadores y una lista de parámetros.
- **Cuerpo.** Contiene las sentencias que implementan su comportamiento, así como variables locales.

Los elementos mínimos que deben aparecer en la declaración del método son:



- Tipo devuelto por el método
- Nombre del método
- Paréntesis (opcionalmente dentro de ellos irán los parámetros que recibe).
- Cuerpo del método entre llaves.

```

int obtenerX ()
{
    /* Cuerpo del método, donde se encuentran la declaración de variables,
    sentencias y todo tipo de estructuras de control.

    ...
}

```

### 4.2. Cabecera del método

Puede incluir los siguientes elementos ordenados:

1. **Modificadores:** public, private y algunos más. No es obligatorio.
2. **Tipo devuelto:** tipo de dato (primitivo o referencia) que el método devuelve tras ejecutarlo. void como tipo devuelto hace que el método no devuelva ningún valor.
3. **Nombre del método.** Se aplica el mismo convenio de nomenclatura que para atributos. Además, por convenio se suele utilizar un verbo o un nombre formado por varias palabras que comiencen por verbo, seguido por adjetivos, nombres, etc... que empiecen en mayúsculas: establecerValor estaVacio moverFicha SubirPalanca girarRuedaIzquierda
4. **Lista de parámetros.** Separados por coma y entre paréntesis, cada parámetro debe incluir el tipo. Si no hay parámetros los paréntesis aparecerán vacíos.
5. **Lista de excepciones.** Se utiliza la palabra reservada throws seguida de la lista de nombres de excepciones esperada por comas, no es obligatorio aunque muchas veces es conveniente.
6. **Cuerpo del método** encerrado en llaves.

```
[private | protected | public] [static] [abstract] [final] [native] [synchronized]  
<tipo> <nombreMétodo> ( [<lista_parametros>] )  
[throws <lista_excepciones>]
```

### 4.3. Modificadores en la declaración de un método

Existen los siguientes tipos de modificadores.

- **Modificador de acceso:** igual que atributos: por omisión, public, private y protected. Tienen el mismo cometido. Acceso por parte de clases del mismo paquete, por cualquier parte del programa, solo por la propia clase o también para las subclases.
- **Modificadores de contenido.** También son static y final, pero cambia su significado. -
  - **Un método static** es igual para todos los objetos de la clase y solo tiene acceso a atributos estáticos de la clase. Pueden ser llamados sin necesidad de tener un objeto de clase instanciado. Por ejemplo, la clase Maths de java tiene métodos estáticos: Maths.abs, Math.sin, Math.cos... En cualquier caso, los objetos de una clase permiten la invocación de métodos estáticos de su clase.
  - **Un método final** es un método que no permite ser sobrescrito por clases descendientes en herencia de la clase a la que pertenece el método.
- **Otros modificadores.** Solo son aplicables a métodos: abstract, native y synchronized.
  - **native** es utilizado para señalar que el método ha sido implementado en código nativo (en un lenguaje compilado a lenguaje máquina como C o C++), solo se indica la cabecera del método, pues no tiene cuerpo en java.
  - **Un método abstracto** no tiene implementación, cuerpo vacío. Es implementado por clases descendientes, solo puede ser declarado como abstract si se encuentra en una clase abstract.
  - En un método synchronized el entorno de ejecución obliga a que cuando un proceso esté ejecutando el método, el resto de procesos que lo tengan que llamar deberán esperar a que el otro proceso termine. Es útil si sabes que determinado método va a ser llamado concurrentemente por varios procesos a la vez.

**Cuadro de aplicabilidad de los modificadores.**

	Clase	Atributo	Método
<b>Sin modificador (paquete)</b>	Sí	Sí	Sí
<b>public</b>	Sí	Sí	Sí
<b>private</b>		Sí	Sí
<b>protected</b>	Sí	Sí	Sí
<b>static</b>		Sí	Sí
<b>final</b>	Sí	Sí	Sí
<b>synchronized</b>			Sí
<b>native</b>			Sí
<b>abstract</b>	Sí		Sí

## 4.4. Parámetros en un método

Se colocan entre paréntesis tras el nombre del método, cada parámetro está compuesto por el tipo de dato y el nombre, separando todos los parámetros con coma. Si no hay parámetros, tan solo aparecerán los paréntesis.

Se debe tener en cuenta:

- Se pueden incluir cualquier cantidad de parámetros.
- Pueden ser de cualquier tipo: primitivos, referencias, objetos, arrays, etc...
- Una variable local del método no puede coincidir con el nombre de un parámetro.
- No puede haber dos parámetros con el mismo nombre.
- El nombre de algún parámetro puede coincidir con algún atributo de clase, este será ocultado por el parámetro, para poder acceder al atributo se utiliza el operador this.
- En java, el paso de parámetros es siempre por valor, excepto en tipos referenciados como objetos, en cuyo caso se pasa una referencia, que no podrá ser cambiada pero sí elementos de su interior, a través de sus métodos.

Es posible utilizar una construcción especial llamada **varargs**, que permite que un método pueda tener un número variable de argumentos, para ello se colocan puntos suspensivos después del tipo del cual se va a tener una lista variable, un espacio en blanco y a continuación el nombre del parámetro que aglutina la lista de argumentos variables.

Se trata de una manera transparente de pasar un array con un número variable de elementos para no tenerlo que hacer manualmente.

## 4.5. Cuerpo de un método

Está compuesto por una serie de sentencias java:

- Declaración de variables locales del método.
- Sentencias que implementan la lógica del método: estructuras de control, utilización de métodos de otros objetos, cálculos, cadenas, etc...
- Sentencia de devolución de valor de retorno (return). Aparece al final del método permitiendo devolver la información que se le pide al método. Si el método es void no debe aparecer.

En el caso de la clase Punto teníamos los métodos de obtenerX y obtenerY. En ambos casos se devolvía un valor entero mediante la sentencia return, no recibían parámetros, ni hacían cálculos. Un método que devuelve valor de un atributo se suele llamar de tipo get o getter.

También teníamos dos métodos que servían para establecer valores a atributos del objeto (establecerX y establecerY). En este caso se le pasa un valor al método por parámetro, el cual es utilizado para variar el atributo del objeto. No se devuelve valor porque es void y no hay sentencia return. Un método que establece valores y no devuelve ninguno, se suele llamar de tipo set o setter.

## 4.6. Sobrecarga de métodos

Se pueden tener varias versiones de un mismo método gracias a la sobrecarga de métodos. Esto permite declarar en una misma clase varias versiones del mismo método con el mismo nombre. El compilador distingue entre varios métodos con el mismo nombre mediante la lista de parámetros del método. Desde que tiene distintos parámetros se consideran distintos métodos.

Por ejemplo, en una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir.

- pintarEntero(int entero)
- pintarReal(double real)
- pintarCadena (String cadena)
- pintarEnteroCadena(int entero, String cadena)

Podemos utilizar la sobrecarga de métodos de esta manera:

- pintar (int entero)
- pintar (double real)
- pintar (String cadena)
- pintarEnteroCadena (int entero, String cadena)

Como único dará error el compilador es si se utiliza otro método con el mismo número y tipo de datos de otro ya existente. Tampoco se permitiría intentar distinguirlos por el tipo de dato que devuelven.

## 4.7. Sobrecarga de operadores

Aunque java no lo permite, otros lenguajes de programación permiten la sobrecarga de operadores, para darle otro significado dependiendo del tipo de objetos con el que se va a operar.

+ x +	= +
- x -	= +
+ x -	= -
- x +	= -
+ : +	= +
- : -	= +
+ : -	= -
- : +	= -

En algunos casos puede resultar útil, ya que estos operadores resultan intuitivos.

## 4.8. La referencia this

la palabra referencia this consiste en una referencia al objeto actual. Resulta útil para evitar ambigüedad entre el nombre del parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo.

this es una referencia a la propia clase en la que te encuentras, por tanto te puedes referir a un atributo del objeto con: this.atributo.

En el ejemplo de la clase Punto, this podría utilizarse si el parámetro pasado en el método establecerX se llamase igual que el atributo.

```
void establecerX (int x)
{
    this.x= x;
}
```

## 4.9. Métodos estáticos

Un método estático puede ser usado directamente desde la clase, sin necesidad de crear una instancia del método. Son conocidos como métodos de clase (frente a los métodos de objeto).

Los métodos estáticos no manipulan atributos de instancias, solo atributos estáticos y suelen ser utilizados para realizar operaciones comunes a todos los objetos.

Por ejemplo.

- Acceso a atributos específicos de clase: incremento o decremento de contadores de clase...
- Operadores genéricas relacionadas con la clase, por ejemplo en una clase NIF que permite trabajar con el DNI y la letra, que proporciona funciones adicionales para calcular la letra de un número de DNI, este método puede ser interesante para ser usado desde fuera de la clase de manera independiente a la existencia de objetos tipo NIF.

En la biblioteca Java se encuentran multitud de clases que proporcionan métodos estáticos, que son útiles para cálculos auxiliares, conversiones de tipos, etc... La mayoría de clases del paquete java.lang ofrecen métodos estáticos para hacer conversiones:

static String valueOf(int i). Devuelve en formato string un valor int.

```
String enteroCadena = String.valueOf(23);
```

static int parseInt(String s). Método estático de la clase Integer, analiza la cadena pasada por parámetro y la transforma en un int.

```
int cadenaEntero = Integer.parseInt("-12");
```

Este tipo de clases se utilizan como una especie de caja de herramientas que contienen métodos que pueden ser utilizados desde cualquier parte, suelen ser métodos públicos.

## 5. Encapsulación, control de acceso y visibilidad

Dentro de la POO es muy importante el concepto de ocultación. Los modificadores de acceso en java permiten especificar el ámbito de visibilidad de los miembros de una clase.

En función de la visibilidad que se desee que tengan los objetos o miembros de los objetos, se eligen los distintos modificadores.

Los modificadores de acceso determinan si una clase puede utilizar miembros de otra clase. Existen dos niveles:

- **A nivel general (nivel de clase):** visibilidad de la propia clase: público (public) o privada al paquete (sin modificador)
- **A nivel de miembros:** especificación, miembro por miembro de su nivel de visibilidad: público, privado al paquete, privado (private) y protegido protected).

### 5.1. Ocultación de atributos. Métodos de acceso

Los atributos de una clase se suelen declarar como privados o como mucho protected, pero no como public. Así evitamos que se puedan manipular inadecuadamente desde otro objeto.

Para poder manipular estos atributos creamos métodos públicos que permiten acceder a esos atributos de manera controlada: métodos getter o setter. Muchos programadores utilizan el nombre en inglés para estos métodos: setNombre, getNombre, setX, setY...

Por otro lado también podemos tener métodos que no devuelven el atributo en sí, sino un cálculo realizado con él. Por ejemplo, podemos tener getDNI, al que se le pasa el número del DNI y nos devuelve el DNI junto con la letra, calculando en el método la propia letra.

También podríamos tener un método setDNI, que deje almacenar el DNI junto con la letra, siempre y cuando el cálculo de la letra a partir de los números sea el correcto. El código que comprueba que es un DNI correcto se encuentra dentro del método setDNI y si no es correcto devuelve un error de validación (por ejemplo, lanzando una excepción). En cualquier caso, la letra del DNI no tiene por qué almacenarse nunca, al ser un cálculo

## 5.2. Ocultación de métodos

Los métodos de una clase pertenecen a su interfaz y es lógico que se declaren como públicos, pero se puede dar el caso de necesitar algunos métodos privados a la clase. Son métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí son públicos. Se suelen declarar como privados porque no son de interés fuera del contexto del propio objeto.

En el ejemplo del DNI, la propia comprobación de la letra del DNI se podría calcular en un método privado que luego utilice el método get o set para interactuar con el exterior.

## 6. Utilización de los métodos y atributos de una clase

Una vez implementada una clase con sus atributos y método, podemos proceder a utilizarla instanciando objetos de esa clase o interaccionar con ellos. Esto se hace con el operador new.

### 6.1. Declaración de un objeto

Se declara exactamente igual que una variable de cualquier tipo.

```
Punto p1;
Rectangulo r1, r2;
Coche cocheAntonio;
String palabra;
```

Todas estas variables en realidad son referencias (punteros o direcciones de memoria), que apuntan a un objeto (zona de memoria) de la clase indicada en la declaración.

Un objeto recién declarado (referencia recién creada) no apunta a nada, la referencia está vacía o es una referencia nula (de hecho la variable del objeto contiene el valor null). Solamente es una variable que existe y está preparada para guardar una dirección de memoria donde se encuentra el objeto al que hace referencia. **El objeto aún no existe (no ha sido creado o instanciado).**

Para que esta variable o referencia apunte realmente a un objeto (contenga una referencia o dirección de memoria que apunte a una zona de memoria en la que hay espacio reservado para un objeto), es necesario crear o instanciar el objeto. Para ello se utiliza el operador new.

### 6.2. Creación de un objeto

Para crear un objeto o instancia de clase, es necesario utilizar el operador new, seguido del nombre del constructor de la clase y sus paréntesis con los parámetros. En el caso de no tener parámetros se pueden omitir los paréntesis.

```
p1= new Punto ();
r1= new Rectangulo ();
r2= new Rectangulo();
cocheAntonio= new Coche();
palabra= String;
```

El constructor de una clase es un método especial que toda clase tiene y coincide con el nombre de la clase. Se encarga de crear y construir el objeto, solicitando la reserva de memoria necesaria para los atributos e inicializarlos a algún valor si es necesario.

El entorno de ejecución de java es el que se encarga de reservar la memoria para la estructura del objeto a partir de ejecutar un método constructor. Entre estas tareas se encuentra: solicitar una zona de memoria disponible, reservar memoria para los atributos, enlazar la variable objeto en esa zona, etc...).

El método constructor no es necesario implementarlo si no se quiere hacer, ya que Java se encarga de dotar de un constructor por omisión o por defecto, pero el constructor por omisión no tiene parámetros ni realiza tareas adicionales.

Se puede declarar un objeto e instanciarlo en la misma línea:

```
Punto p1 = new Punto();
```

### 6.3. Manipulación de un objeto: utilización de métodos y atributos

Una vez declarado e instanciado el método, ya existe en el entorno de ejecución y puede ser manipulado en el programa, haciendo uso de sus atributos o método.

Para acceder al miembro de un objeto se utiliza el operador punto:

```
<nombreObjeto>.<nombreMiembro>
```

Ejemplos:

```
Punto p1, p2, p3;  
p1= new Punto();  
p1.x= 5; //esto no es recomendable porque implica declarar el atributo como público dentro de la clase.  
p1.y= 6;  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.x, p1.y);  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());  
p1.establecerX(25);  
p1.establecerX(30);  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());
```

## 7. Constructores

En el ciclo de vida de un objeto se distinguen las fases de:

- Constructor del objeto.
- Manipulación y utilización del objeto accediendo a sus miembros.
- Destrucción del objeto.

El constructor es un método especial con el mismo nombre de la clase que se encarga de construir el objeto.

### 7.1. Concepto de constructor

Es un método que tiene el mismo nombre de la clase a la que pertenece y no devuelve ningún valor. Su única función es proporcionar el mecanismo de creación de instancias u objetos de clase.

Un constructor es, al fin y al cabo, una especie de método (algo especial) y como java soporta la sobrecarga de métodos, también se permite la sobrecarga de constructores (disponer de varios constructores).

Toda clase tiene al menos un constructor, que si no se define, el propio compilador lo hará por nosotros. El constructor por defecto se encarga de inicializar los atributos de la clase a sus valores por defecto (0 para numéricos, null para referencias, false para boolean...)

Una vez se incluye un constructor personalizado a una clase, el compilador ya no incluye constructor por defecto y no se podrá utilizar. Si quieras que tu clase tenga constructor sin parámetros tendrás que escribir el código.

## 7.2. Creación de constructores

Los constructores indican:

- El tipo de acceso.
- El nombre de la clase (el mismo que el de un constructor).
- Lista de parámetros que puede aceptar.
- Si lanza o no excepciones.
- Cuerpo del constructor (bloque de código).

Si se crea un constructor con parámetros y no se ha implementado el constructor por defecto, el intento de utilización del constructor por defecto lanza un error de compilación.

Ejemplo de constructor:

```
public Punto (int x, int y)
{
    this.x= x;
    this.y= y;
    cantidadPuntos++; // Suponiendo que tengamos un atributo estático cantidadPuntos
}
```

En este caso el constructor recibe dos parámetros y reserva espacio para los atributos, también asigna valores iniciales a los dos atributos y por último incrementa un atributo estático llamado cantidadPuntos

## 7.3. Utilización de constructores

La forma de utilizar un constructor es igual que la del constructor por defecto, pero teniendo en cuenta que si hemos declarado parámetros en el método constructor, tendremos que asignar valores para esos parámetros:

```
Punto p1
p1 = new Punto(10,7);
```

En este caso estamos utilizando el constructor implementado y no el constructor por defecto.

## 7.4. Constructores de copia

Una forma de iniciar un objeto puede ser mediante copia de los valores de atributos de otro objeto ya existente.

Si generamos dos objetos exactamente iguales, basados en la misma clase, puedes hacer que el segundo objeto se genere con los mismos valores que el otro que ya existe. A este proceso de clonación se le llama constructor copia o constructor de copia.

Este constructor es un método constructor que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar:

```

public Punto (Punto p)
{
    this.x= p.obtenerX();
    this.y= p.obtenerY();
}

```

El constructor recibe como parámetro un objeto del mismo tipo (clase Punto), inspecciona el valor de sus atributos x e y, y los reproduce en los atributos del objeto en proceso de construcción. Para utilizar este constructor podríamos hacer lo siguiente:

```

Punto p1, p2;
p1= new Punto (10, 7);
p2= new Punto (p1);

```

## 7.5. Destrucción de objetos

Cuando un objeto deja de ser utilizado, los recursos usados por él: memoria, acceso a archivos, conexiones bbdd... Deben ser liberados para ser utilizados por otros procesos.

La destrucción de objetos es trabajo del recolector de basura (garbage collector). Este proceso busca periódicamente objetos que ya no están referenciados y los marca para ser eliminados. Luego los elimina cuando considera oportuno.

En java también es posible implementar un método destructor de clase `finalize();`

El método `finalize` es llamado por el recolector de basura cuando se va a destruir el objeto (esto no se sabe cuando va a suceder. Si ese método no existe se ejecuta un destructor por defecto).

Se recomienda que si un objeto utiliza determinados recursos de los cuales no hay garantía que el entorno de ejecución los vaya a eliminar, se implemente el método `finalize` en tus clases. Si lo único que utilizan nuestras clases es espacio en memoria para los atributos, estos si son liberados sin problemas.

Si en un momento dado es necesario garantizar el proceso de finalización, se puede recurrir al método `runFinalization()` de la clase `System` para forzarlo, ya que el entorno de ejecución de java solo marca los objetos para ser borrados pero no lo tiene por qué hacer en ese instante.

Este método se encarga de llamar a todos los métodos marcados por el recolector para ser destruidos.

Para implementar un destructor se debe tener en cuenta:

- El nombre del método se debe llamar `finalize();`
- No recibe parámetros
- Solo puede haber uno.
- No devuelve valores (debe ser `void`)

## 8. Empaquetado de clases

La encapsulación de la información dentro de las clases permite llevar a cabo el proceso de ocultación. Conforme aumenta la complejidad de una aplicación, es probable que se necesite que algunas clases puedan tener acceso a parte de la implementación de otras debido a la relación que se establezca entre ellas cuando diseñas tu modelo de datos.

Se suele hablar de un nivel de superior de encapsulamiento llamado empaquetado.

Un paquete es un conjunto de clases relacionadas entre sí y agrupadas bajo el mismo nombre. Se encuentran en el mismo paquete todas esas clases que forman una biblioteca o que reúnen algún tipo de característica

común.

## 8.1. Jerarquía de paquetes

Los paquetes en java se organizan jerárquicamente de manera similar a la estructura de carpetas de un dispositivo de almacenamiento.

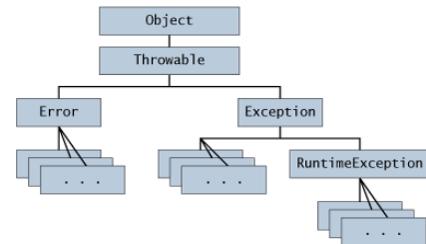
- Las clases serían los archivos
- Cada paquete es una carpeta que contiene esas clases.
- Cada paquete puede contener otros paquetes.
- Para hacer referencia a una clase dentro de una estructura de paquetes hay que indicar la trayectoria completa desde el paquete raíz hasta el paquete en el que se encuentra la clase.

La estructura de paquetes de java permite organizar y clasificar las clases, evitando conflictos de nombres y facilitando la ubicación de una clase dentro de una estructura jerárquica.

También permite el control de acceso a miembros de las clases desde otras clases del mismo paquete gracias a los modificadores.

Las clases proporcionadas por java en sus bibliotecas son miembros de distintos paquetes organizados jerárquicamente. Ese conjunto de paquetes forman la API de java.

Las clases básicas se encuentran en `java.lang`, las de entrada y salida se encuentran en `java.io`, en `java.math` se encuentran clases para trabajar con números grandes y de precisión.



## 8.2. Utilización de paquetes

Es posible acceder a cualquier clase de cualquier paquete, siempre que esté disponible en nuestro sistema, para ello debemos usar la estructura jerárquica del paquete. Esto se hace utilizando el operador punto para especificar cada subpaquete

```
java.lang.String;  
java.util.regex.Patern.
```

Existe la posibilidad de indicar si se desea trabajar con las clases de uno o varios paquetes, de manera que cuando se vaya a utilizar una clase de esos paquetes no es necesario indicar toda la trayectoria. Esto se realiza con `import`.

```
import java.lang.String; //este es un ejemplo, pero este paquete no es necesario importarlo ya que es el paquete por defecto.  
import java.util.regex.Patern.
```

Una vez importado el paquete solo se tiene que utilizar el nombre de la clase y no toda su trayectoria para trabajar con ella.

Si vamos a trabajar con varias clases de un mismo paquete podemos utilizar el símbolo asterisco para indicar que queremos importar todas las clases del mismo paquete.

```
import java.util.*;  
import java.util.regex.*;
```

## 8.3. Inclusión de una clase en un paquete

Se realiza mediante la palabra reservada package seguida del nombre del paquete en la cabecera del programa.

```
package paqueteEjemplo;  
class claseEjemplo {  
    ...  
}
```

Esta sentencia se debe incluir en cada archivo fuente de cada clase. Si en un archivo fuente hay distintas clases definidas, todas formarán parte del mismo paquete.

Si no se incluye ninguna sentencia package, el compilador considera que esas clases forman parte del paquete por omisión (paquete sin nombre asociado al proyecto).

Dentro de un mismo paquete no pueden haber clases con el mismo nombre para evitar ambigüedad, pero si pueden existir en paquetes distintos.

Al igual que una clase debe tener el mismo nombre que el archivo fuente, la estructura de los paquetes debe corresponder al mismo nivel de directorios y carpetas para que el compilador sea capaz de localizar todos los paquetes.

El compilador debe tener conocimiento de donde comienza la estructura de carpetas definida por los paquetes en la cual se encuentran las clases. Para ello se utiliza el ClassPath.

## 8.4. Proceso de creación de un paquete

Se recomienda seguir los siguientes pasos.

- Poner nombre al paquete: suele ser habitual utilizar el dominio de internet de la empresa que lo ha creado. Para el caso de miempresa.com se podría utilizar com.miempresa.
  - Crear una estructura jerárquica de carpetas equivalente a la estructura de subpaquetes. La ruta de la raíz de esa estructura debe estar especificada en el ClassPath de java.
  - Especificar a que paquete pertenecen la clase o clases del archivo .java usando la sentencia package.

## Mapa conceptual



# 6. Estructuras de almacenamiento de información.

	Autor	(X) Xerach Casanova
	Clase	Programación
	Fecha	@Feb 27, 2021 5:42 PM

1. Introducción a las estructuras de almacenamiento

2. Cadenas de caracteres

    2.1. Operaciones avanzadas con cadenas de caracteres

        2.1.1. Operaciones avanzadas con cadenas de caracteres (I)

        2.1.1. Operaciones avanzadas con caracteres de cadena (II)

        2.1.2. Operaciones avanzadas con cadenas de caracteres (III)

        2.1.3. Opciones avanzadas con cadenas de caracteres (IV)

        2.1.4. Operaciones avanzadas con cadenas de caracteres (V)

    2.2. Expresiones regulares

        2.2.1. Expresiones regulares (I)

        2.2.1. Expresiones regulares (II)

        2.2.2. Expresiones regulares (III)

3. Creación de arrays

    3.1. Uso de arrays unidimensionales

    3.2. Inicialización

4. Arrays multidimensionales

    4.1. Uso de arrays multidimensionales

    4.2. Inicialización de arrays multidimensionales

Mapa conceptual

Cuando un programa maneja datos compuestos (datos compuestos a su vez de datos simples)

Los datos compuestos son un tipo de estructura de datos. Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos. El objeto o instancia de una clase sería un dato compuesto. A veces los datos tienen estructuras aún más complejas y necesitan otro tipo de soluciones.

## 1. Introducción a las estructuras de almacenamiento

A la hora de utilizar estructuras de datos del mismo tipo que varían en tamaño de forma dinámica se utilizan estructuras de datos. Las clases son una evolución de un tipo de estructuras conocidas como datos compuestos, también llamadas registros).

Las estructuras de almacenamiento se pueden clasificar atendiendo a:

- **Estructuras con capacidad de almacenar datos del mismo tipo:** varios números, varios caracteres, etc... Arrays, cadenas de caracteres, listas y conjuntos.
- **Estructuras con capacidad de almacenar varios datos de distinto tipo:** fechas, cadenas de caracteres, etc... Todo dentro de una misma estructura (las clases son un ejemplo de ello).

Atendiendo a la forma que los datos se ordenan en la estructura podemos diferenciar:

- **Estructuras que no se ordenan de por sí** y debe ser el programador el encargado de hacerlo si es necesario.
- **Estructuras ordenadas.** Se trata de estructuras que incorporan un dato nuevo a los ya existentes, este se almacena en una posición concreta e irá en función del orden.

## 2. Cadenas de caracteres

Son estructuras de almacenamiento que permiten almacenar secuencias de caracteres de casi cualquier longitud. Un carácter se codifica como secuencias de bits que representan los símbolos usados en la comunicación humana: letras, números, símbolos matemáticos, ideogramas y pictogramas.

La forma más habitual de ver escrita una cadena de caracteres es como un literal de cadena, que consiste simplemente en una secuencia de caracteres entre comillas dobles.

Los literales de cadena en Java son instancias de la clase String. Cada vez que escribimos un literal, se crea una instancia de la clase String. Esto da flexibilidad pero a la vez consume memoria.

**String cad = "Ejemplo de cadena";**

En este caso, el literal de cadena a la derecha es una instancia de la clase String y la variable cad se convierte en una referencia a ese objeto ya creado. Otra forma de hacerlo sería:

**String cad= new String ("Ejemplo de cadena");**

En este caso, se realiza una copia en memoria de la cadena pasada por parámetro, la nueva instancia de la clase String hará referencia a la copia de la cadena y no a la original.

## 2.1. Operaciones avanzadas con cadenas de caracteres

### 2.1.1. Operaciones avanzadas con cadenas de caracteres (I)

La operación avanzada más sencilla es la concatenación con el signo suma

```
String cad = "Bien"+ "venido";
System.out.println(cad);
```

En la operación anterior se crea una nueva cadena a partir de dos cadenas.

Otra forma de usar la concatenación sería usando el método concat del objeto String

```
String cad = "Bien".concat("venido");
System.out.printf(cad);
```

En ambas expresiones participan tres instancias de la clase String, una contiene el texto "Bien", otra el texto "venido" y otra que contiene "Bienvenido", la cual se crea al realizar la operación de concatenación, sin que las otras dos hayan desaparecido. El recolector de basura se encargará de borrar las otras dos cadenas cuando detecte que ya no se usan.

Además, en el ejemplo se puede observar que se puede invocar un método de la clase String poniendo el método al literal de cadena, esto demuestra que escribiendo el literal se está creando un objeto inmutable String.

Con el método `toString()` podemos concatenar cadenas con literales numéricos e instancias de otros objetos.

**El método `toString()` es un método disponible en todas las clases de Java, permite la conversión de una instancia de clase en cadena de texto. Convertir no siempre es posible, hay clases fácilmente convertibles en texto, como Integer y otras que el resultado de invocar `toString()` es información relativa a la instancia.**

La ventaja del método `toString()` es que se invoca automáticamente sin especificarlo:

```
Integer i1 = new Integer(1223);
System.out.println("Número: " + i1);
```

En el ejemplo anterior se ha invocado automáticamente `i1.toString()`;

### 2.1.1. Operaciones avanzadas con caracteres de cadena (II)

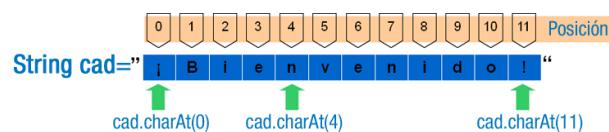
Partiendo de que en todos los siguientes ejemplos, la cadena `cad` contiene la cadena ¡Bienvenidos!

- **int Length()**. Retorna un número entero que contiene la longitud de la cadena, incluyendo los espacios.

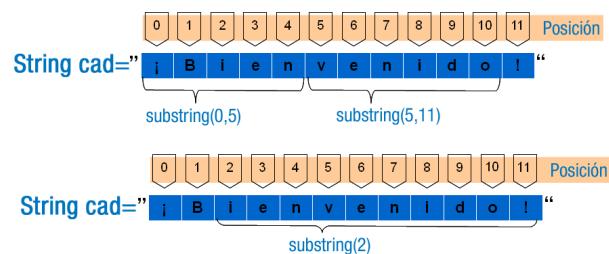
**String cad=" i B i e n y e n j d o ! "**

**Longitud = 12**

- **char charAt(int pos).** Retorna el carácter ubicado de la posición pasada por parámetro. El carácter obtenido se almacena en un tipo de dato char. Las posiciones empiezan a contar desde el 0.



- **String substring(int beginIndex, int endIndex)**. Permite extraer una subcadena de otra de mayor tamaño entre la posición que se indica en el parámetro beginIndex y la de endIndex -1. cad.substring(0,5) devuelve el valor ¡Bien



- **String substring(int beginIndex).** Si solo se le proporciona un parámetro al método substring, se extraerá una cadena que comience a partir de la posición beginIndex

```
String subcad = cad.substring(2);
System.out.println(subcad);
//Se devuelve "ienvenido!"
```

Otra operación habitual es convertir números a cadenas y cadenas a números. Esto ayuda a evitar errores, haciendo que el usuario siempre inserte cadenas, aunque el dato que vaya a insertar sea un número. Seguidamente se convierte el número en cadena.

Los números almacenados en memoria se hace en números binarios, no se debe confundir tipos de datos numéricos (int, short, long, float y double) con secuencias de caracteres.

La conversión de numéricos es fácil gracias al método `toString`:

```
String cad2 = "Número cinco: " + 5;  
System.out.println(cad2);
```

Esto es posible sin errores gracias a que Java convierte el número 5 a su clase envoltorio (wrapper class) correspondiente: Integer, Float, Double... y después ejecuta el método `toString` de dicha clase.

### 2.1.2. Operaciones avanzadas con cadenas de caracteres (III)

Para hacer operaciones numéricas con cadenas de caracteres que contienen números, primero se deben convertir esas cadenas a tipos numéricos. El método que ofrece java para ello es `valueOf`, existente en todas las clases envoltorio descendientes de Number: Integer, Long, Short, Float y Double.

```
String c="1234.5678"; //Cadena que contiene un número flotante  
  
double n; //declaramos una variable double  
try {  
  
    n = Double.valueOf(c).doubleValue();  
    /* Con el método valueOf convertimos el contenido de  
     la variable de cadena c en el tipo primitivo double. */  
  
} catch (NumberFormatException e) {  
    //código a ejecutar si no se puede convertir.  
}
```

Con el formateado de cadenas conseguimos darle el formato que queramos a tipos numéricos. En Java podemos formatear cadenas a través del método estático `format`, disponible en el objeto `String`.

```
float precio = 3.3f;  
  
String salida = string.format("El precio es: %.2f €", precio));  
  
System.out.println(salida);
```

El formato de salida, también denominado "cadena de formato" es el primer argumento del método `format`, la variable `precio` es la variable que se proyectará en la salida siguiendo un formato concreto. `%.2f` es un especificador de formato.

### 2.1.3. Opciones avanzadas con cadenas de caracteres (IV)

Java ofrece muchas más operaciones sobre cadenas de caracteres. En la siguiente tabla, `cad1`, `cad2` y `cad3` son cadenas ya existentes, la variable `num` es un número entero mayor o igual a cero.

`cad`

Método.	Descripción
<code>cad1.compareTo(cad2)</code>	Permite comparar dos cadenas entre sí lexicográficamente. Retornará 0 si son iguales, un número menor que cero si la cadena (cad1) es anterior en orden alfabético a la que se pasa por argumento (cad2), y un número mayor que cero si la cadena es posterior en orden alfabético.
<code>cad1.equals(cad2)</code>	Cuando se comparan si dos cadenas son iguales, no se debe usar el operador de comparación "==" , sino el método <code>equals</code> . Retornará <code>true</code> si son iguales, y <code>false</code> si no lo son.
<code>cad1.compareToIgnoreCase(cad2)</code> <code>cad1.equalsIgnoreCase(cad2)</code>	El método <code>compareToIgnoreCase</code> funciona igual que el método <code>compareTo</code> , pero ignora las mayúsculas y las minúsculas a la hora de hacer la comparación. Las mayúsculas van antes en orden alfabético que las minúsculas, por lo que hay que tenerlo en cuenta. El método <code>equalsIgnoreCase</code> es igual que el método <code>equals</code> pero sin tener en cuenta las minúsculas.
<code>cad1.trim()</code>	Genera una copia de la cadena eliminando los espacios en blanco anteriores y posteriores de la cadena.
<code>cad1.toLowerCase()</code>	Genera una copia de la cadena con todos los caracteres a minúscula.
<code>cad1.toUpperCase()</code>	Genera una copia de la cadena con todos los caracteres a mayúsculas.
<code>cad1.indexOf(cad2)</code> <code>cad1.indexOf(cad2,num)</code>	Si la cadena o carácter pasado por argumento está contenida en la cadena invocante, retorna su posición, en caso contrario retornará -1. Opcionalmente se le puede indicar la posición a partir de la cual buscar, lo cual es útil para buscar varias apariciones de una cadena dentro de otra.
<code>cad1.contains(cad2)</code>	Retornará <code>true</code> si la cadena pasada por argumento está contenida dentro de la cadena. En caso contrario retornará <code>false</code> .
<code>cad1.startsWith(cad2)</code>	Retornará <code>true</code> si la cadena comienza por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.endsWith(cad2)</code>	Retornará <code>true</code> si la cadena acaba por la cadena pasada como argumento. En caso contrario retornará <code>false</code> .
<code>cad1.replace(cad2,cad3)</code>	Generará una copia de la cadena <code>cad1</code> , en la que se reemplazarán todas las apariciones de <code>cad2</code> por <code>cad3</code> . El reemplazo se hará de izquierda a derecha, por ejemplo: reemplazar "zzz" por "xx" en la cadena "zzzzz" generará "xxzz" y no "zzxx".

## 2.1.4. Operaciones avanzadas con cadenas de caracteres (V)

El principal problema de las cadenas de caracteres es su alto consumo de memoria.

Cuando un programa realiza muchas operaciones con cadenas es conveniente optimizar el uso de memoria.

En Java String es un objeto inmutable, esto significa que cuando creamos un String o un literal de String, se crea un objeto no modificable. Java proporciona la clase StringBuilder, mutable, que permite una mayor optimización de memoria, así como la clase StringBuffer, pensada para aplicaciones multihilo.

La clase StringBuilder permite modificar la cadena que contiene, mientras que String no.

Por ejemplo:

```
StringBuilder strb = new StringBuilder("Hoal Muundo");
```

Con los métodos append (insertar al final), insert (insertar una cadena o carácter en una posición específica), delete y replace podemos rectificar la cadena anterior:

1. **strb.delete(6, 8);** eliminamos las 'uu' que sobran en la cadena, la primera u está en la posición 6 empezando desde 0 y la última está en la posición 7 (hay que pasar como parámetro la posición contigua al carácter a eliminar como en el método substring).
2. **strb.append("!");** añadimos al final de la cadena el cierre de exclamación.
3. **strb.insert(0, "¡");** añadimos al principio el símbolo de apertura de exclamación.
4. **strb.replace(3,5, "la");** Reemplazamos los caracteres 'al' situados en la posición inicial 3 y la posición final 4, por 'la', al igual que en el método delete y substring se pasa como parámetro la posición contigua al carácter final (5)

## 2.2. Expresiones regulares

### 2.2.1. Expresiones regulares (I)

La función de las expresiones regulares es permitir comprobar si una cadena sigue o no un patrón preestablecido. Son un mecanismo que describe esos patrones y se construyen de una forma relativamente sencilla.

Existen librerías distintas para trabajar con expresiones regulares y casi todas siguen una sintaxis más o menos similar con ligeras variaciones.

Esta sintaxis permite indicar el patrón de forma cómoda, como si se tratase de una cadena de texto, en la que determinados símbolos tienen un significado especial. Las reglas generales para construir expresiones regulares son:

- Se puede indicar que una cadena contiene un conjunto de símbolos fijo, poniendo esos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán el código de escape. Por ejemplo el patrón "aaa" admitirá cadenas que contengan tres aes.
- "[xyz]". Entre corchetes indicamos opcionalidad, solo uno de los símbolos podrá aparecer en el lugar donde están los corchetes. Por ejemplo "aaa[xy]" admitirá como válidas las cadenas "aaax" y "aaay". Los corchetes representan una posición de la cadena que puede tomar uno o varios valores.
- "[a-z]" "[A-Z]" "[a-zA-Z]" indicamos que el patrón admite cualquier carácter entre la letra inicial y final.
- "[0-9]" igual que con caracteres pero con numéricos.

Con estas reglas podemos indicar el conjunto de símbolos que admite el patrón y su orden. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, la cadena no encajará en el patrón.

- "a?" El interrogante indica que un símbolo puede aparecer una vez o ninguna.
- "a\*" El asterisco indica que un símbolo puede aparecer una, muchas o ninguna vez.
- "a+" El símbolo suma indica que otro símbolo debe aparecer al menos una vez.
- "a{1,4}" Usando llaves se indica el número mínimo y máximo de veces que el símbolo puede repetirse.
- a{2,} El símbolo aparece un mínimo de veces sin determinar el máximo
- a{5} Sin la coma el símbolo debe aparecer exactamente las veces que se indica.
- "[a-z]{1,4}[0-9]+" Los indicadores de repetición también se pueden utilizar entre corchetes. En este ejemplo se permiten de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

## 2.2.1. Expresiones regulares (II)

Para usar las expresiones regulares, Java ofrece las clases Pattern y Matcher contenidas en el paquete `java.util.regex.*`.

La clase Pattern se utiliza para procesar la expresión regular y compilarla. Verifica que es correcta y la deja lista para su utilización. Matcher sirve para comprobar si la cadena sigue o no un patrón.

```
Pattern p = Pattern.compile("[01]");
Matcher m = p.matcher("00001010");
if (m.matches()) System.out.println("Sí, contiene el patrón");
else System.out.println("No, no contiene el patrón");
```

El método estático compile de la clase Pattern crea un patrón, el cual compila la expresión regular pasada por parámetro y genera una instancia de Pattern (p). El patrón p se utilizará siempre que quieras para verificar si una cadena coincide o no con el patrón.

Esta comprobación se hace con el método marcher, que combina el patrón con la cadena de entrada y genera una instancia de la clase Matcher (m). La clase Matcher contiene el resultado de dicha comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- **m.matches()** devuelve true si toda la cadena (de principio a fin) encaja con el patrón o false en caso contrario.
- **m.lookingAt()** devuelve true si el patrón se encontró al principio de la cadena.
- **m.find()** devuelve true si el patrón existe en algún lugar de la cadena y false en caso contrario puede haber más de una coincidencia. Para obtener la posición exacta donde se produjo la coincidencia usamos m.start() y m.end, una segunda invocación del método find() irá a la segunda coincidencia y así sucesivamente. podemos volver a comenzar invocando al método m.reset();

Algunas construcciones adicionales que pueden ayudarnos a especificar expresiones más complejas:

- "**[^abc]**" Cuando ^ se pone justo detrás del corchete de apertura significa negación. En este caso se admite cualquier símbolo distinto a, b o c.
- "**^[01]+\$**" Cuando el símbolo ^ se pone al comienzo, permite indicar comiendo de línea o de entrada, y \$ indica fin de línea o fin de entrada. Útil trabajando con modo multilínea y con el método find().
- **\d** Un dígito numérico (equivale a "[0-9]").
- **\D** Cualquier cosa excepto un dígito numérico (equivale a "[^0-9]").
- **\s** Un espacio en blanco (incluye tabulaciones, saltos de línea y otras formas de espacio).
- **\S** Cualquier cosa excepto espacio en blanco.
- **\w** Cualquier carácter que podrías encontrar en una palabra, equivale a "[a-zA-Z\_0-9]".

## 2.2.2. Expresiones regulares (III)

Los paréntesis tienen un significado especial, permiten indicar repeticiones en un conjunto de símbolos. Por ejemplo: "**(#[01])\{2,3}**". La expresión #[01] admite cadenas como #0 o #1, pero al ponerlo entre paréntesis e indicar los contadores de repetición, decimos que la misma secuencia se debe repetir entre 2 y 3 veces, con lo que las cadenas admitidas serían #0#1 o #0#1#0.

Además los paréntesis llevan una función adicional, permite definir grupos. Los grupos permiten acceder de forma cómoda a las diferentes partes de la cadena cuando coincide con una expresión regular.

```
Pattern p = Pattern.compile("[XY]?([0-9]{1,9})(A-Za-z]");
Matcher m = p.matcher("X123456789Z Y00110011M 999999T");
while (m.find()) {
    System.out.println("Letra inicial opcional: " + m.group(1));
    System.out.println("Número: " + m.group(2));
    System.out.println("Letra NIF: " + m.group(3));
}
```

Usando los grupos obtenemos por separado el contenido de cada uno de los grupos.

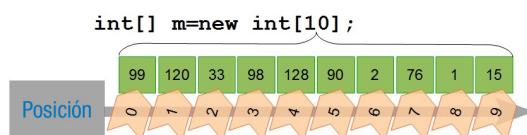
El primer grupo es 1 y no 0, si se pone m.group(0) se obtiene una cadena con toda la concurrencia o coincidencia del patrón en la cadena, es decir, se obtiene la secuencia entera de símbolos que coincide con el patrón.

Si en el ejemplo anterior se usara el método find, este buscaría una a una cada una de las ocurrencias del patrón en la cadena. Cada vez que se invoca, busca la siguiente ocurrencia del patrón y devuelve true. Si no encuentra en una iteración ninguna ocurrencia es porque no existen más y retornará false saliendo del bucle.

### Secuencias de escape.

Sirven para indicar en una cadena que habrá un paréntesis, una llave, o un corchete. Dado que esos símbolos se usan en los patrones. Para ello se antepone al símbolo \\, excepto para las comillas, en las que se utiliza una sola barra \".

## 3. Creación de arrays



Los arrays permiten almacenar una colección de objetos o datos al mismo tiempo. Su utilización se realiza de la siguiente manera:

- **Declaración.** Sigue la estructura "tipo[] nombre". El tipo será el tipo de dato que almacene o una clase ya existente, de la cual se almacenan varias unidades.
- **Creación del array.** Consiste en indicar el tamaño que tendrá. "nombre = new tipo[dimension]". El array no puede cambiar de tamaño una vez creado.

```
int [] n;
n = new int[10]
```

```
int[] m = new int[10]; //declaración y creación en la misma línea.
```

Una vez declarado y creado podemos almacenar valores en cada una de las posiciones del array, indicando en el interior de los corchetes la posición, siendo cero la primera.

### 3.1. Uso de arrays unidimensionales

Los tres ámbitos donde se utilizan arrays son: modificación de una posición, acceso a una posición y paso de parámetros.

La **modificación de una posición** se realiza con una simple asignación:

```
int[] Numeros = new int[3];  
  
Numeros[0] = 99;  
Numeros[1] = 120;  
Numeros[2] = 33;
```

El acceso a un valor ya existente se consigue poniendo el nombre del array y la posición entre corchetes

```
int suma = Numeros [0] + Numeros[1] + Numeros[2];
```

Los arrays son como objetos en Java y disponen de la propiedad `length`, que permite saber el tamaño de cualquier array:

```
System.out.println("Longitud del array: " + Numeros.length);
```

El **paso de parámetros** a una función o método se hace de la siguiente manera:

```
int sumaarray (int[] j) {  
  
    int suma = 0;  
  
    for (int i=0; i<j.length; i++)  
  
        suma = suma+h[i];  
  
    return suma;  
}
```

En este ejemplo pasamos el array `j` como argumento al método, dentro de él se recorre con un bucle `for` y se suma el contenido de cada una de las posiciones del array dentro de la variable `suma`.

Para pasar un array ya creado como argumento se pasa simplemente poniendo el nombre:

```
int suma = sumaarray (Numeros);
```

## 3.2. Inicialización

La forma más habitual de llenar un array es a través de un método que lleve a cabo la creación y llenado del array, seguidamente el método retorna el array indicando en la declaración que el valor retornado es tipo[].

```
static int[] arrayConNumerosConsecutivos (int totalNumeros) {  
    int [] r = new int [totalNumeros];  
    for (int i=0; i<totalNumeros; i++) r[i] = i;  
    return r;  
}
```

En este ejemplo se crea dentro de un método que crea un array con una serie de números consecutivos, el tamaño del array se pasa por parámetro al método. El método finalmente devuelve el array inicializado con los valores que hemos introducido en el bucle for.

Otra forma de inicializar es en la propia declaración del array, utilizando llaves.

```
int[] array = {10, 20, 30};  
  
String[] diassemana = {"lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"};
```

Este tipo de inicialización funciona solo con datos primitivos o Strings y algunos casos más, pero no con cualquier objeto.

En una creación de array de objetos, la inicialización inicial será null. Al crear un array de objetos no significa que se haya creado las instancias de los objetos, las cuales hay que crearlas para cada posición del array.

```
StringBuilder[] j= new StringBuilder[10];  
  
for (int i=0; i<j.length; i++) j[i] = new StringBuilder("cadena "+i);
```

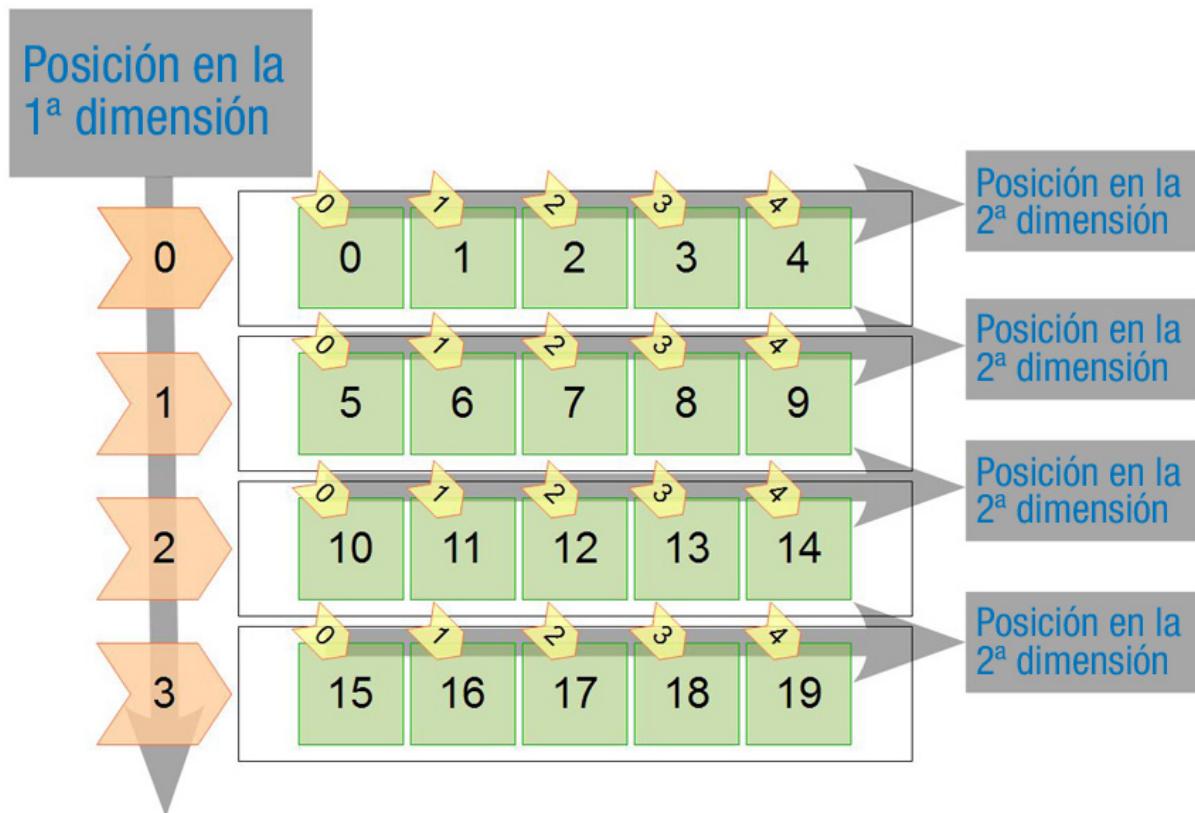
En este ejemplo creamos un array de objetos de la clase StringBuilder, seguidamente utilizamos un bucle for para inicializar cada una de las posiciones del array

## 4. Arrays multidimensionales

Los arrays multidimensionales en Java se crean de la siguiente manera:

```
int[][] a2d=new int[4][5];
```

Se crea un array de dos dimensiones, que contendrá 4 arrays de 5 números cada uno:



Se pueden hacer arrays de las dimensiones que queramos y de cualquier tipo. Todos deben ser del mismo tipo y la declaración comienza especificando el tipo de clase de los elementos, después ponemos tantos corchetes como dimensiones tenga el array y después el nombre del array.

### 4.1. Uso de arrays multidimensionales

```
int[][] a2d = new int[4][5];
```

Para acceder a cada uno de los elementos del array anterior se indica la posición en las dos dimensiones, teniendo en cuenta que los índices de ambas empiezan en 0.

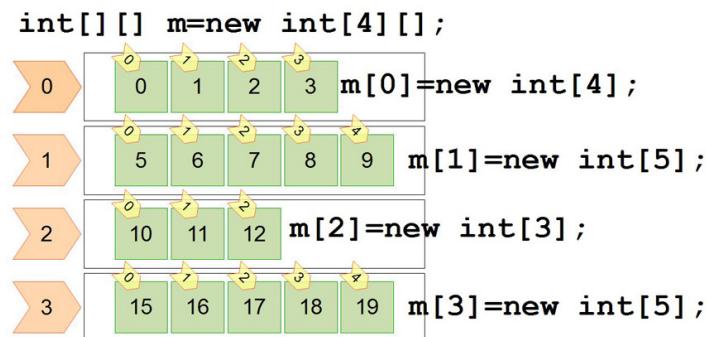
```
a2d[0][0] = 3;
```

También se pueden pasar como parámetros a métodos:

```
static int sumaarray2d(int[][] a2d){  
  
    int suma = 0;  
  
    for (int i1=0; i1 < a2d.length; i1++)  
        for (int i2 = 0; i2 < a2d[i1].length; i2++)  
            suma += a2d[i1][i2];  
  
    return suma;  
}
```

Aplicando length directamente sobre el array nos permite saber el tamaño de la primera dimensión (`a2d.length`). Para saber el tamaño de una dimensión superior tenemos que poner el o los índices entre corchetes seguidos de `length` (`a2d[i1].length`).

Gracias a esto podemos tener arrays multidimensionales irregulares:



Esto significa que los arrays de la segunda dimensión pueden ser de distinto tamaño entre sí. Para hacer esto se hace de la siguiente manera:

- Declaramos el array sin especificar la segunda dimensión: `irregular = new int[3][];`
- Después creamos cada uno de los arrays unidimensionales del tamaño que queramos y lo asignamos a la posición correspondiente del array anterior:

```
irregular[0] = new int[7];  
irregular[1] = new int[15];  
irregular[2] = new int[9];
```

## 4.2. Inicialización de arrays multidimensionales

Para que una función retorne un array multidimensional se hace igual que en arrays multidimensionales, simplemente hay que poner el tipo de dato seguido del número de corchetes correspondiente.

```

int[][] inicializarArray (int n, int m) {
    int[][] ret=new int[n][m];
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            ret[i][j]=n*m;
    return ret;
}

```

También se puede inicializar usando llaves, poniendo después de la declaración del array un símbolo de igual, encerrando entre llaves los valores del array separados por comas, pero poniendo llaves nuevas en cada nueva dimensión:

```

int[][] a2d={{0,1,2},{3,4,5},{6,7,8},{9,10,11}};
int[][][] a3d={{{0,1},{2,3}},{{0,1},{2,3}}};

```

El primer array es de  $4 \times 3$  y el segundo array es de  $2 \times 2 \times 2$ . Con esta notación también se pueden inicializar arrays irregulares:

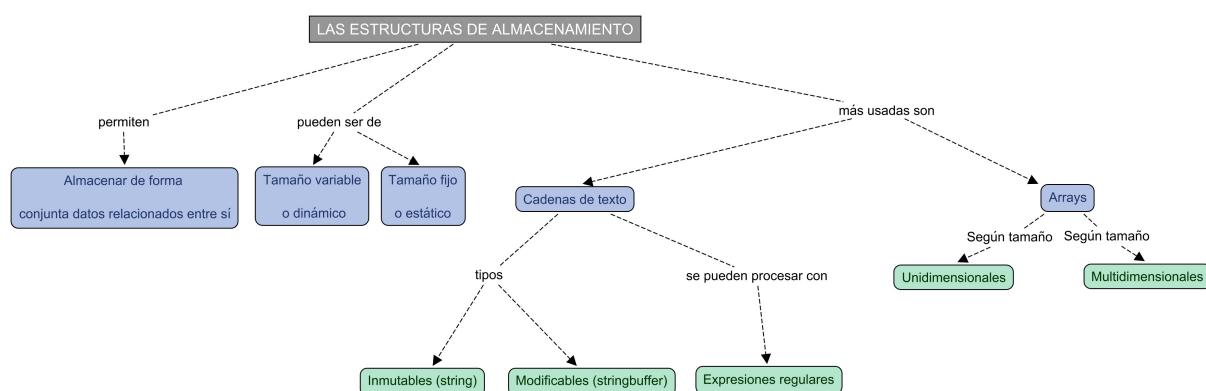
```

int[][] i2d={{0,1},{0,1,2},{0,1,2,3},{0,1,2,3,4},{0,1,2,3,4,5}};
int[][][] i3d={ { {0,1},{0,2} } , { {0,1,3} } , { {0,3,4},{0,1,5} } };

```

Por último se ha de saber que los arrays también reciben el nombre de **arreglos**, **vectores** o **matrices**.

## Mapa conceptual





# 7. Utilización avanzada de clases.

	Autor	(X) Xerach Casanova
	Clase	Programación
	Fecha	@Mar 6, 2021 1:36 PM

## 1. Relaciones entre clases

- 1.1. Composición
- 1.2. Herencia
- 1.3. ¿Herencia o composición?

### Composición

- 2.1. Sintaxis de la composición
- 2.2. Uso de la composición(I). Preservación de la ocultación.
- 2.3. Uso de la composición (II). Llamadas a constructores

### 3. Herencia

- 3.1. Sintaxis de la herencia
- 3.2. Acceso a miembros heredados
- 3.3. Utilización de miembros heredados (I). Atributos.
  - 3.3.1. Utilización de miembros heredados (II). Métodos
- 3.4. Redefinición de métodos heredados
- 3.5. Ampliación de métodos heredados
- 3.6. constructores y herencia
- 3.7. Cración y utilización de clases derivadas
- 3.8. La clase Object en java
- 3.9. Herencia múltiple

### 4. Clases abstractas

- 4.1. Declaración de una clase abstracta
- 4.2. Métodos abstractos
- 4.3. Clases y métodos finales

### 5. Interfaces

- 5.1. Concepto de interfaz
  - 5.1.1. ¿Clase abstracta o interfaz?
- 5.2. Definición de interfaces
- 5.3. Implementación de interfaces
- 5.4. Simulación de la herencia múltiple mediante el uso de interfaces
- 5.5. Herencia de interfaces

### 6. Polimorfismo

- 6.1. Concepto de polimorfismo
- 6.2. Ligadura dinámica
- 6.3. Limitaciones de la ligadura dinámica
- 6.4. Interfaces y polimorfismo
- 6.5. Conversión de paquetes

Anexo I - Elaboración de los constructores de la clase Rectángulo

Anexo II - Métodos para las clases heredadas Alumno y Profesor

Mapa Conceptual

# 1. Relaciones entre clases

A la hora de diseñar un conjunto de clases para modelar el conjunto de información a automatizar, es importante establecer apropiadamente las clases relacionales que existen entre ellas.

Una clase puede ser una especialización de otra, o bien una generalización. Una clase puede contener objetos en el interior de otra, o utiliza otra clase. Se pueden distinguir distintos tipos de relaciones entre clases:

- **Clientela.** Cuando una clase utiliza objetos de otra, por ejemplo al pasarlo como parámetros a través de un método. Es la relación fundamental y más habitual entre clases. Por ejemplo, usando objetos de tipo String en una clase, esta clase será cliente de la clase String.
- **Composición.** Cuando alguno de los atributos de una clase es un objeto de otra clase. Por ejemplo, si describes una clase donde uno de sus atributos es de tipo String, tu clase está compuesta por un objeto de tipo String.
- **Anidamiento.** Cuando se definen clases en el interior de otra clase. Es la menos habitual, implica declarar clases dentro de otras clases, en algunos casos puede resultar útil para tener un nivel más de encapsulamiento y ocultación.
- **Herencia.** Cuando una clase comparte determinadas características de otra (clase base), añadiéndole funcionalidad específica (especialización).

Podría decirse que la composición y la anidación son casos particulares de clientela, ya que en realidad en todos los casos una clase está haciendo uso de otra, al contener atributos que son objetos de otra clase, al definir clases dentro de otras clases, al utilizar objetos en paso de parámetros, al declarar variables locales utilizando otras clases, etc...

## 1.1. Composición

La composición se da cuando una clase contiene algún atributo que es una referencia a un objeto de otra clase. Es decir, el objeto de la clase A contiene uno o varios objetos de la clase B.

Por ejemplo, si describes una entidad País compuesta por una serie de atributos, entre los cuales se encuentra una lista de comunidades autónomas, podrías decir que los objetos de la clase País contienen varios objetos de la clase ComunidadAutonoma.

La composición se puede encadenar todas las veces que sea necesario hasta llegar a objetos básicos del lenguaje o hasta tipos primitivos, que ya no contendrán objetos en su interior. Es una manera eficiente y sencilla de gestionar la reutilización de todo el código ya escrito.

La forma más sencilla de plantearte si una relación existe entre dos clases A y B es de composición es usando la expresión "tiene un" entre las dos clases.

- Un coche tiene un motor y tiene cuatro ruedas.
- Una persona tiene una cuenta bancaria asociada para ingresar nómina.

## 1.2. Herencia

El concepto de herencia es algo básico pero potente. Cuando se define una nueva clase y ya existen clases que de alguna manera implementan parte de la funcionalidad que necesita, es posible crear una nueva clase derivada de la que ya tienes. Esto posibilita la reutilización de todos los atributos y métodos de la clase padre o superclase, sin necesidad de tener que escribirlos de nuevo.

Una subclase hereda todos los miembros de la clase padre: atributos, métodos y clases internas.

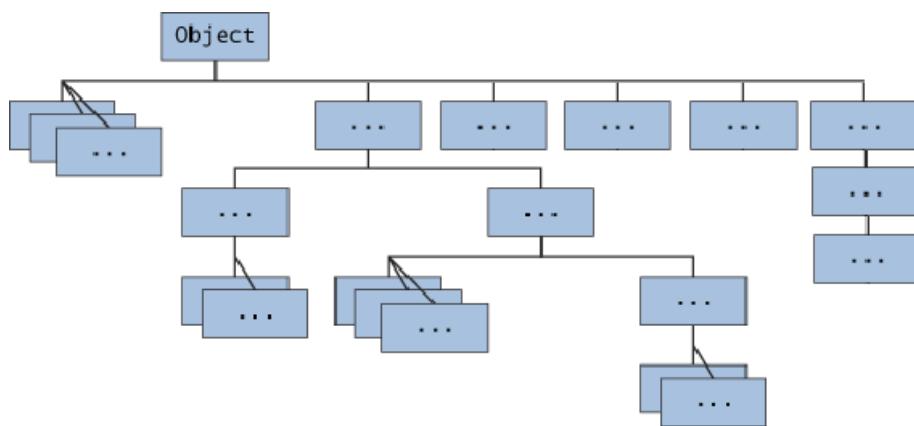
Los constructores no se heredan, pero se pueden invocar desde la subclase.

- Un coche es un vehículo (hereda atributos como velocidad máxima, o métodos como parar y arrancar).
- Un empleado es una persona (hereda atributos como nombre o fecha de nacimiento).
- Un rectángulo es una figura geométrica (hereda métodos como el cálculo de la superficie, o su perímetro).

La expresión idiomática para plantear si existe herencia entre dos clases A y B es "es un". La clase A "es un" tipo específico de la clase B (especialización), o visto de otro modo: la clase B es un caso general de la clase A.

Java implementa la herencia mediante la utilización de la palabra reservada `extends`. La clase `Object`, dentro del paquete `java.lang`, define e implementa el comportamiento común de todas las clases, incluyendo las propias. Cualquier clase deriva en última instancia de la clase `Object`.

Todas las clases tienen una clase padre, que a su vez posee una superclase y así llegar hasta la clase `Object`.



### 1.3. ¿Herencia o composición?

Cuando escribimos nuestras propias clases se debe tener claro cuando utilizar la composición y cuando la herencia.

En el caso de la composición, una clase puede estar formada por objetos de otras clases, pero no necesariamente tienen que compartir características. Estos objetos incluidos no son más que atributos miembros de la clase que estamos definiendo.

En el caso de la herencia, una clase cumple todas las características de otra y además las suyas propias (especialización, particularización, extensión o restricción). O lo que es lo mismo, una clase base es una generalización de las clases derivadas.

#### Ejemplo:

Por ejemplo, imagina que dispones de una clase `Punto` (ya la has utilizado en otras ocasiones) y decides definir una nueva clase llamada `Círculo`. Dado que un punto tiene como atributos sus coordenadas en plano (`x1, y1`), decides que es buena idea aprovechar esa información e incorporarla en la clase `Círculo` que estás escribiendo. Para ello utilizas la herencia, de manera que al衍生 la clase `Círculo` de la clase `Punto`, tendrás disponibles los atributos `x1` e `y1`. Ahora solo

faltaría añadirle algunos atributos y métodos más como por ejemplo el radio del círculo, el cálculo de su área y su perímetro, etc.

En principio parece que la idea pueda funcionar pero es posible que más adelante, si continúas construyendo una jerarquía de clases, observes que puedas llegar a conclusiones incongruentes al suponer que un círculo es una especialización de un punto (un tipo de punto). ¿Todas aquellas figuras que contengan uno o varios puntos deberían ser tipos de punto? ¿Y si tienes varios puntos? ¿Cómo accedes a ellos? ¿Un rectángulo también tiene sentido que herede de un punto? No parece muy buena idea. En este caso habría resultado mejor establecer una relación de composición. Analízalo detenidamente: ¿cuál de estas dos situaciones te suena mejor?

1. **Un círculo es un punto** (su centro)", y por tanto heredará las coordenadas **x1** e **y1** que tiene todo punto. Además tendrá otras características específicas como el **radio** o métodos como el cálculo de la **longitud** de su perímetro o de su **área**.
2. "**Un círculo tiene un punto** (su centro)", junto con algunos atributos más como por ejemplo el **radio**. También tendrá métodos para el cálculo de su **área** o de la longitud de su **perímetro**.

Parece que en este caso la composición refleja con mayor fidelidad la relación que existe entre ambas clases. **Normalmente suele ser suficiente con plantearse las preguntas “¿A es un tipo de B?” o “¿A contiene elementos de tipo B?”.**

## Composición

### 2.1. Sintaxis de la composición

En la composición no es necesaria ninguna sintaxis especial. cada uno de esos objetos es un atributo:

```
class Rectangulo {  
    private Punto vertice1;  
    private Punto vertice2;  
}
```

Los métodos de esta clase deben tener en cuenta que ya no hay cuatro atributos de tipo double, sino dos atributos de tipo punto, cada uno de los cuales contiene en su interior dos atributos de tipo double.

#### Ejercicio resuelto:

Intenta escribir los siguientes los métodos de la clase Rectangulo teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase Punto, en lugar de cuatro elementos de tipo double):

1. **Método calcularSuperficie()**, que calcula y devuelve el área de la superficie encerrada por la figura.
2. **Método calcularPerimetro()**, que calcula y devuelve la longitud del perímetro de la figura.

En ambos casos la interfaz no se ve modificada en absoluto (desde fuera su funcionamiento es el mismo), pero internamente deberás tener en cuenta que ya no existen los atributos `x1`, `y1`, `x2`, `y2`, de tipo `double`, sino los atributos `vertice1` y `vertice2` de tipo `Punto`.

```
public double calcularSuperficie () {  
    double area, base, altura; // Variables locales  
  
    base= vertice2.obtenerX () - vertice1.obtenerX (); // Antes era x2 - x1  
    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1  
    area= base * altura;  
    return area;  
}  
  
public double CalcularPerimetro () {  
    double perimetro, base, altura; // Variables locales  
  
    base= vertice2.obtenerX () - vertice1.obtenerX (); // Antes era x2 - x1  
    altura= vertice2.obtenerY () - vertice1.obtenerY (); // Antes era y2 - y1  
    perimetro= 2*base + 2*altura;  
    return perimetro;  
}
```

## 2.2. Uso de la composición(I). Preservación de la ocultación.

Cuando se escriben clases que contienen objetos de otras clases, se debe tener precaución con aquellos métodos que devuelven información de atributos de la clase (métodos `get`).

Los atributos suelen ser privados o protegidos, para ocultarlos a los posibles clientes de la clase. Para que otros objetos puedan acceder a la información contenida en los atributos, deben hacerlo a través de los métodos que sirvan de interfaz (métodos `getter` y `setter`).

Al igual que con los atributos primitivos, podemos devolver un objeto completo, pero **se debe tener en cuenta que si en un método de la clase devuelves un objeto que es atributo, se está ofreciendo directamente una referencia a un objeto atributo que probablemente se ha definido como privado**. De esta forma estás volviendo a hacer público un atributo que inicialmente era privado.

Para evitar estas situaciones, se opta por diversas alternativas, procurando evitar la devolución directa de un atributo que sea un objeto.

- Devolver siempre tipos primitivos.
- Dado que esto siempre no es posible, otra posibilidad es crear un nuevo objeto que sea copia del atributo a devolver y utilizar ese objeto como valor de retorno.

También se debe tener en cuenta que es posible que en algunos casos sí se necesite realmente la referencia al atributo original (algo habitual en atributos estáticos). En tales casos no hay problema

en devolver directamente el atributo para que el código llamante (cliente) haga uso de él.

#### Ejercicio resuelto:

**Dada la clase Rectangulo, escribe sus nuevos métodos obtenerVertice1 y obtenerVertice2 para que devuelvan los vértices inferior izquierdo y superior derecho del rectángulo (objetos de tipo Punto), teniendo en cuenta su nueva estructura de atributos (dos objetos de la clase Punto, en lugar de cuatro elementos de tipo double):**

Los métodos de obtención de vértices devolverán objetos de la clase Punto:

```
public Punto obtenerVertice1 ()  
{  
    return vertice1;  
}  
  
public Punto obtenerVertice2 ()  
{  
    return vertice2;  
}
```

Esto funcionaría perfectamente, pero deberías tener cuidado con este tipo de métodos que devuelven directamente una referencia a un objeto atributo que probablemente has definido como privado. Estás de alguna manera haciendo público un atributo que fue declarado como privado.

Para evitar que esto suceda bastaría con crear un nuevo objeto que fuera una copia del atributo que se desea devolver (en este caso un objeto de la clase **Punto**).

Aquí tienes algunas posibilidades:

```
public Punto obtenerVertice1 () // Creación de un nuevo punto extrayendo sus atributos  
{  
    double x, y;  
  
    Punto p;  
  
    x= this.vertice1.obtenerX();  
    y= this.vertice1.obtenerY();  
  
    p= new Punto (x,y);  
  
    return p;  
}  
  
public Punto obtenerVertice1 () // Utilizando el constructor copia de Punto (si es que está definido)  
{  
    Punto p;  
  
    p= new Punto (this.vertice1); // Uso del constructor copia  
  
    return p;  
}
```

## 2.3. Uso de la composición (II). Llamadas a constructores

A la hora de escribir clases que contienen objetos de otras clases como atributos, es su comportamiento cuando se instancian. Durante el proceso de creación del objeto de la clase contenedora, también se debe tener en cuenta la creación de aquellos objetos contenidos.

El constructor de la clase contenedora debe invocar a los constructores de las clases de los objetos contenidos.

Además, hay que tener cuidado con las referencias a objetos pasados como parámetros para llenar el contenido de los atributos. **Es conveniente hacer una copia de esos objetos y utilizar esas copias para los atributos**, ya que si se utiliza la referencia que se ha pasado como parámetro, el código cliente de la clase puede tener acceso a ella sin necesidad de pasar por la interfaz de la clase y volveríamos a dejar abierta una puerta pública a algo que quizás sea privado.

Si el objeto parámetro que se pasa al constructor formaba parte de otro objeto, esto podría ocasionar un efecto colateral si esos objetos son modificados desde el código cliente de la clase. Podemos sin querer compartir esos objetos con otras partes del código sin ningún tipo de control de acceso.

**En el fondo los objetos no son más que variables de tipo referencia a la zona de memoria en la que se encuentra toda la información del objeto en sí mismo. Esto es, puedes tener un único objeto y múltiples referencias a él. Pero sólo se trata de un objeto, y cualquier modificación desde una de sus referencias afectaría a todas las demás, pues estamos hablando del mismo objeto.**

Hay que recordar que solo se crean objetos cuando se llama a un constructor, y que si se realizan reasignaciones o pasos de parámetros, no se están copiando o pasando copias de los objetos, sino referencias de ellas.

En este caso, el objeto se está adueñando de un parámetro objeto que pertenece a otro y que es posible que en un futuro haga uso de él.

## 3. Herencia

La clase de la que se hereda se llama clase base, clase padre o superclase, la clase hereda es la clase hija, derivada o subclase.

Una clase derivada puede ser clase padre de otra que herede de ella, dando lugar a jerarquía de clases.

Una clase hija no tiene acceso a miembros privados de su clase padre, tan solo a sus públicos o a sus protegidos, a los que solo tienen acceso clases derivadas y las del mismo paquete. Los miembros que sean privados de la clase base son también heredados, pero el acceso a ellos es restringido al propio funcionamiento de la superclase y solo se puede acceder a ellos si la superclase dejó algún medio indirecto para hacerlo. Por ejemplo, a través de un método.

Los miembros de la superclase son heredados por la subclase, algunos pueden ser redefinidos o sobreescritos (overriden), y también se pueden añadir nuevos miembros (especialización).

### 3.1. Sintaxis de la herencia

En java la herencia se indica mediante la palabra reservada extends.

```
[modificador] class ClasePadre {
    // Cuerpo de la clase
    ...
}

[modificador] class ClaseHija extends ClasePadre {
    // Cuerpo de la clase
    ...
}
```

Ejemplo. Una clase Persona contiene atributos: nombre, apellido y fecha de nacimiento.

```
public class Persona {
    String nombre;
    String apellidos;
    GregorianCalendar fechaNacim;
    ...
}
```

Pero podemos necesitar la clase Alumno, que comparte esos atributos, pero además tiene características propias (especialización).

```
public class Alumno extends Persona {
    String grupo;
    double notaMedia;
    ...
}
```

A partir de ahora, un objeto de la clase Alumno contendrá los atributos grupo y notaMedia (propios de la clase Alumno), pero también nombre, apellidos y fechaNacim (propios de su clase base Persona y que por tanto ha heredado).

### 3.2. Acceso a miembros heredados

No es posible acceder a miembros privados de una superclase. Para ello existe el modificador **protected**, el cual permite el acceso a todas las clases heredadas, pero no desde fuera de las clases (estrictamente hablando, desde fuera del paquete, que serían como miembros privados).

**Cuadro de niveles accesibilidad a los atributos de una clase**

	Misma clase	Subclase	Mismo paquete	Otro paquete
<b>Sin modificador (paquete)</b>	X		X	X
<b>public</b>	X	X	X	X
<b>private</b>	X			
<b>protected</b>	X	X	X	

Al definir la clase Alumno como heredera de Persona, no habrías tenido acceso a esos atributos si declaramos los atributos como protected, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar esos atributos como protected o bien sin modificador (para que también tengan acceso a ellos otras clases del mismo paquete, si es que se considera oportuno):

```
public class Persona {
    protected String nombre;
    protected String apellidos;
    ...
}
```

### 3.3. Utilización de miembros heredados (I). Atributos.

Los atributos heredados por una clase son a efectos prácticos iguales que aquellos definidos en la clase derivada.

En el ejemplo anterior la clase Persona disponía de tres atributos y la clase Alumno, que heredaba de ella, añadía dos atributos más. Desde un punto de vista funcional podrías considerar que la clase Alumno tiene cinco atributos: tres por ser Persona (nombre, apellidos, fecha de nacimiento) y otros dos más por ser Alumno (grupo y nota media).

#### 3.3.1. Utilización de miembros heredados (II). Métodos

Los métodos también se heredan en la clase derivada, sumándose a los que se implementen en ella.

En el ejemplo de la clase Persona, si dispusiéramos de métodos get y set para cada uno de sus tres atributos (nombre, apellidos, fechaNacim), tendrías seis métodos que podrían ser heredados por sus clases derivadas. Podrías decir entonces que la clase Alumno, derivada de Persona, tiene diez métodos:

- Seis por ser Persona (getNombre, getApellidos, getFechaNacim, setNombre, setApellidos, setFechaNacim).
- Otros cuatro más por ser Alumno (getGrupo, setGrupo, getNotaMedia, setNotaMedia).

Sin embargo, sólo tendrías que definir esos cuatro últimos (los específicos) pues los genéricos ya los has heredado de la superclase.

### 3.4. Redefinición de métodos heredados

Una clase puede redefinir algunos métodos que ha heredado de su clase base, este nuevo método (especializado) sustituye al heredado. También se conoce como sobreescritura de métodos.

Igualmente, se puede acceder al método original con la referencia super, aunque haya sido sobreescrito o redefinido, pero solo se podrá hacer eso con los métodos heredados de la clase padre inmediatamente superior.

Los métodos redefinidos pueden ampliar su accesibilidad con respecto a la del método original, pero nunca la restringen. Un método declarado protected o de paquete en la clase padre, puede redefinirse como public en la clase derivada.

Los métodos estáticos o de clase no pueden ser sobreescritos. Los originales de la clase base permanecen inalterables.

Cuando sobrescribas un método heredado en Java puedes incluir la anotación @Override. Esto indicará al compilador que tu intención es sobre escribir el método de la clase padre. De este modo, si te equivocas (por ejemplo, al escribir el nombre del método) y no lo estás realmente sobre escribiendo, el compilador producirá un error y así podrás darte cuenta del fallo. En cualquier caso, no es necesario indicar @Override

En el ejemplo de la clase Alumno, podrían redefinirse algunos de los métodos heredados. Por ejemplo, imagina que el método getApellidos devuelva la cadena "Alumno:" junto con los apellidos del alumno. En tal caso habría que rescribir ese método para realizar esa modificación:

```
@override  
public String getApellidos () {  
    return "Alumno: " + apellidos;  
}
```

#### Ejercicio resuelto

Dadas las clases Persona, Alumno y Profesor que has utilizado anteriormente, redefine el método getNombre para que devuelva la cadena "Alumno: ", junto con el nombre del alumno, si se trata de un objeto de la clase Alumno o bien "Profesor ", junto con el nombre del profesor, si se trata de un objeto de la clase Profesor.

Clase Alumno.

Al heredar de la clase Persona tan solo es necesario escribir métodos para los nuevos atributos (métodos especializados de acceso a los atributos especializados), pues los métodos genéricos (de acceso a los atributos genéricos) ya forman parte de la clase al haberlos heredado. Esos son los métodos que se implementaron en el ejercicio anterior (getGrupo, setGrupo, etc.).

Ahora bien, hay que escribir otro método más, pues tienes que redefinir el método getNombre para que tenga un comportamiento un poco diferente al getNombre que se hereda de la clase base Persona:

```
// Método getNombre  
@Override  
public String getNombre (){
```

```
        return "Alumno: " + this.nombre;  
    }
```

En este caso podría decirse que se “renuncia” al método heredado para redefinirlo con un comportamiento más especializado y acorde con la clase derivada.

Clase Profesor.

Seguimos exactamente el mismo procedimiento que con la clase Alumno (redefinición del método `getNombre`).

```
// Método getNombre  
  
@Override  
  
public String getNombre (){  
  
    return "Profesor: " + this.nombre;  
}
```

### 3.5. Ampliación de métodos heredados

En algunas ocasiones, podemos ampliar el comportamiento de un método en vez de ampliarlo.

Para poder preservar el comportamiento del método de la superclase y añadir el nuevo se puede invocar desde el método ampliador de la clase derivada al método ampliado, usando la referencia `super`.

Esta referencia es una referencia a la clase padre de la que te encuentres en cada momento.

Por ejemplo, imagina que la clase Persona dispone de un método que permite mostrar el contenido de algunos datos personales de los objetos de este tipo (nombre, apellidos, etc.). Por otro lado, la clase Alumno también necesita un método similar, pero que muestre también su información especializada (grupo, nota media, etc.). ¿Cómo podrías aprovechar el método de la superclase para no tener que volver a escribir su contenido en la subclase?

```
public void mostrar () {  
  
    super.mostrar (); // Llamada al método "mostrar" de la superclase  
  
    // A continuación mostramos la información "especializada" de esta subclase  
  
    System.out.printf ("Grupo: %s\n", this.grupo);  
  
    System.out.printf ("Nota media: %.2f\n", this.notaMedia);  
}
```

Este tipo de ampliaciones de métodos resultan especialmente útiles por ejemplo en el caso de los constructores, donde se podría ir llamando a los constructores de cada superclase encadenadamente hasta el constructor de la clase en la cúspide de la jerarquía (el constructor de la clase `Object`).

#### Ejercicio resuelto

Dadas las clases Persona, Alumno y Profesor, define un método mostrar para la clase Persona, que muestre el contenido de los atributos (datos personales) de un objeto de la clase Persona. A continuación, define sendos métodos mostrar especializados para las clases Alumno y Profesor que “amplíen” la funcionalidad del método mostrar original de la clase Persona.

Método mostrar de la clase Persona.

```
public void mostrar () {  
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");  
  
    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());  
  
    System.out.printf ("Nombre: %s\n", this.nombre);  
  
    System.out.printf ("Apellidos: %s\n", this.apellidos);  
  
    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);  
}
```

Método mostrar de la clase Profesor.

Llamamos al método mostrar de su clase padre (Persona) y luego añadimos la funcionalidad específica para la subclase Profesor:

```
public void mostrar () {  
    super.mostrar (); // Llamada al método “mostrar” de la superclase  
  
    // A continuación mostramos la información “especializada” de esta subclase  
  
    System.out.printf ("Especialidad: %s\n", this.especialidad);  
  
    System.out.printf ("Salario: %7.2f euros\n", this.salario);  
}
```

Método mostrar de la clase Alumno.

Llamamos al método mostrar de su clase padre (Persona) y luego añadimos la funcionalidad específica para la subclase Alumno:

```
public void mostrar () {  
    super.mostrar ();  
  
    // A continuación mostramos la información “especializada” de esta subclase  
  
    System.out.printf ("Grupo: %s\n", this.grupo);  
  
    System.out.printf ("Nota media: %5.2f\n", this.notaMedia);  
}
```

## 3.6. constructores y herencia

Un constructor de una clase puede llamar a otro constructor de la misma clase a través de la referencia this. En estos casos la utilización de this solo puede hacerse en la primera línea del

código constructor.

Un constructor de una clase derivada puede hacer algo parecido para llamar al constructor de su clase base con la palabra super. El constructor de una clase derivada puede llamar primero al constructor de su clase base para que inicialice los atributos heredados y después inicializar los atributos específicos de la clase.

Esta llamada también debe ser la primera sentencia de un constructor (a excepción de que exista una llamada a otro constructor de la clase mediante this).

Si no se incluye la llamada a super() en el constructor, el compilador incluye automáticamente una llamada al constructor por defecto de clase base. Esto da lugar a una llamada en cadena de constructores de superclase hasta llegar a la jerarquía más alta (Object).

En el caso del constructor por defecto de la subclase si el programador no ha escrito ninguno, antes de la inicialización de los atributos a sus valores por defecto, se hace una llamada al constructor de la clase base mediante super.

Cuando se destruye un objeto (método finalize), es importante llamar a los finalizadores en orden inverso a como fueron llamados los constructores. Primero se liberan los recursos de la clase derivada y después los de la clase base mediante super.finalize()

Si la clase Persona tuviera un constructor de este tipo:

```
public Persona (String nombre, String apellidos, GregorianCalendar fechaNacim) {  
    this.nombre= nombre;  
    this.apellidos= apellidos;  
    this.fechaNacim= new GregorianCalendar (fechaNacim);  
}
```

Puedes llamarlo desde un constructor de una clase derivada (Alumno) de la siguiente forma:

```
public Alumno (String nombre, String apellidos, GregorianCalendar fechaNacim, String grupo, double notaMedia) {  
    super (nombre, apellidos, fechaNacim);  
    this.grupo= grupo;  
    this.notaMedia= notaMedia;  
}
```

### 3.7. Creación y utilización de clases derivadas

La idea de la herencia es simplificar los programas al máximo y procurar que haya que escribir la menor cantidad posible de código repetitivo para facilitar la realización de cambios.

En este enlace se muestran ejemplos de utilización de herencia y clases derivadas:

<https://www.youtube.com/watch?v=Nu2ziz9Sq0g>

### 3.8. La clase Object en java

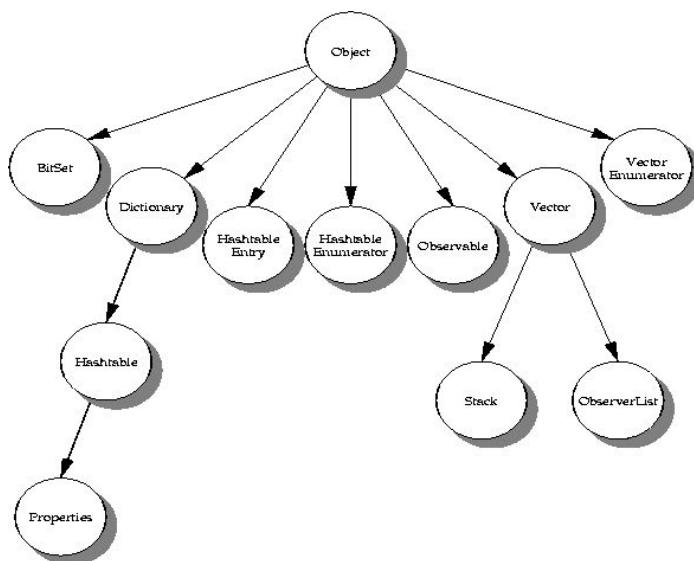
Todas las clases en Java son descendentes de la clase `Object`, la cual define estados y comportamientos básicos que deben tener los objetos. Entre ellos se encuentra:

- La posibilidad de compararse.
- La capacidad de convertirse a cadenas.
- La habilidad de devolver la clase del objeto.

### Principales métodos de la clase Object

Método	Descripción
<code>Object ()</code>	Constructor.
<code>clone ()</code>	Método <b>clonador</b> : crea y devuelve una copia del objeto ("clona" el objeto).
<code>boolean equals (Object obj)</code>	Indica si el objeto pasado como parámetro es igual a este objeto.
<code>void finalize ()</code>	Método llamado por el <b>recolector de basura</b> cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
<code>int hashCode ()</code>	Devuelve un código <b>hash</b> para el objeto.
<code>toString ()</code>	Devuelve una representación del objeto en forma de <code>String</code> .

La clase `Object` representa la superclase que se encuentra en la cúspide de la jerarquía de herencia en Java.



## 3.9. Herencia múltiple

Se podría considerar la posibilidad de necesitar heredar de más de una clase y disponer de los miembros de dos o más clases disjuntas. El problema en estos casos es la posibilidad de producir ambigüedades, por ejemplo, si tenemos miembros con el mismo identificador en clases distintas, ¿qué miembro se hereda? Los compiladores suelen solicitar que ante casos de ambigüedad, se especifique de manera explícita la clase de la cual se quiere determinar un miembro ambiguo.

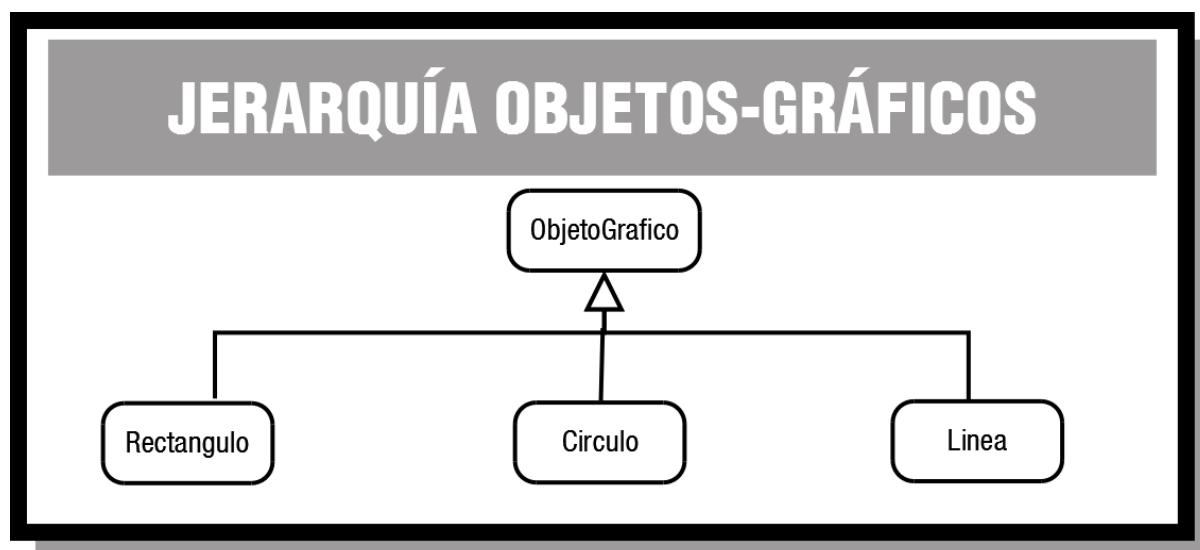
No todos los lenguajes de programación soportan la herencia múltiple y java es uno de ellos.

## 4. Clases abstractas

En algunos casos puede resultar útil disponer de clases que no van a ser instanciadas, pero que proporcionan un marco o modelo a seguir por sus clases derivadas dentro de la jerarquía de herencia.

Permiten dar unas líneas generales de cómo es una clase sin tener que implementar todos sus métodos, o implementando solo algunos de ellos. De esta manera, las clases derivadas podrán usar los mismos métodos de la clase abstracta, especificando su implementación para cada subclase.

Ejemplo: Imagina que estás trabajando en un entorno de manipulación de objetos gráficos y necesitas trabajar con líneas, círculos, rectángulos, etc. Estos objetos tendrán en común algunos atributos que representen su estado (ubicación, color del contorno, color de relleno, etc.) y algunos métodos que modelen su comportamiento (dibujar, llenar con un color, escalar, desplazar, rotar, etc.). Algunos de ellos serán comunes para todos ellos (por ejemplo la ubicación o el desplazamiento) y sin embargo otros (como por ejemplo dibujar) necesitarán una implementación específica dependiendo del tipo de objeto. Pero, en cualquier caso, todos ellos necesitan esos métodos (tanto un círculo como un rectángulo necesitan el método dibujar, aunque se lleven a cabo de manera diferente). En este caso resultaría muy útil disponer de una clase abstracta objeto gráfico donde se definirían las líneas generales (algunos atributos concretos comunes, algunos métodos concretos comunes implementados y algunos métodos genéricos comunes sin implementar) de un objeto gráfico y más adelante, según se vayan definiendo clases especializadas (líneas, círculos, rectángulos), se irán concretando en cada subclase aquellos métodos que se dejaron sin implementar en la clase abstracta.



### 4.1. Declaración de una clase abstracta

Una clase abstracta no se puede instanciar. La idea es permitir a otras clases que heredan de ella, proporcionar un modelo genérico y algunos métodos de utilidad general.

```
[modificador_acceso] abstract class nombreClase [herencia] [interfaces] {  
    ...  
}
```

Una clase puede contener en su interior métodos abstractos, de los cuales solo se indica la cabecera pero no su implementación. En ese caso, la clase necesariamente será **abstract** y esos métodos serán implementados en sus clases derivadas.

Cuando se trabaja con clases abstractas se debe tener en cuenta:

- Solo se puede usar para crear clases derivadas. No se puede hacer un new de una clase abstracta.
- Puede contener métodos totalmente definidos y métodos abstractos.

### Ejercicio resuelto

**Basándote en la jerarquía de clases de ejemplo (Persona, Alumno, Profesor), que ya has utilizado en otras ocasiones, modifica lo que consideres oportuno para que Persona sea, a partir de ahora, una clase abstracta (no instanciable) y las otras dos clases sigan siendo clases derivadas de ella, pero sí instanciables.**

En este caso lo único que habría que hacer es añadir el modificador abstract a la clase Persona. El resto de la clase permanecería igual y las clases Alumno y Profesor no tendrían porqué sufrir ninguna modificación.

```
public abstract class Persona {  
    protected String nombre;  
    protected String apellidos;  
    protected GregorianCalendar fechaNacim;  
    ...  
}
```

## 4.2. Métodos abstractos

Su implementación no se define, se declara únicamente su interfaz o cabecera y su cuerpo será implementado en la clase derivada.

Un método se declara como abstracto con el modificador abstract.

```
[modificador_acceso] abstract <tipo> <nombreMetodo> ([parámetros]) [excepciones];
```

Estos deben ser obligatoriamente definidos en las clases derivadas. Si se deja algún método abstracto sin implementar, esa clase derivada también será una clase abstracta.

Con métodos abstracto debemos tener en cuenta:

- implica que la clase a la que pertenece debe ser abstracta, pero no que todos los métodos tengan que serlo.

- Un método abstracto no puede ser privado ya que no se podría implementar.
- No pueden ser estáticos ya que los métodos estáticos no pueden ser redefinidos.

### Ejercicio resuelto

**Basándose en la jerarquía de clases Persona, Alumno, Profesor, crea un método abstracto llamado mostrar para la clase Persona. Dependiendo del tipo de persona (alumno o profesor) el método mostrar tendrá que mostrar unos u otros datos personales (habrá que hacer implementaciones específicas en cada clase derivada).**

**Una vez hecho esto, implementa completamente las tres clases (con todos sus atributos y métodos) y únalas en un pequeño programa de ejemplo que cree un objeto de tipo Alumno y otro de tipo Profesor, los rellene con información y muestre esa información en la pantalla a través del método mostrar.**

Dado que el método mostrar no va a ser implementado en la clase Persona, será declarado como abstracto y no se incluirá su implementación:

```
protected abstract void mostrar();
```

Recuerda que el simple hecho de que la clase Persona contenga un método abstracto hace que sea clase sea abstracta (y deberá indicarse como tal en su declaración): public abstract class Persona.

En el caso de la clase Alumno habrá que hacer una implementación específica del método mostrar y lo mismo para el caso de la clase Profesor.

#### 1. Método mostrar para la clase Alumno.

```
// Redefinición del método abstracto mostrar en la clase Alumno
public void mostrar () {

    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");

    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());

    System.out.printf ("Nombre: %s\n", this.nombre);

    System.out.printf ("Apellidos: %s\n", this.apellidos);

    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);

    System.out.printf ("Grupo: %s\n", this.grupo);

    System.out.printf ("Grupo: %.2f\n", this.notaMedia);

}
```

#### 2. Método mostrar para la clase Profesor.

```
// Redefinición del método abstracto mostrar en la clase Profesor
public void mostrar () {

    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");

    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());
```

```

        System.out.printf ("Nombre: %s\n", this.nombre);
        System.out.printf ("Apellidos: %s\n", this.apellidos);
        System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);
        System.out.printf ("Especialidad: %s\n", this.especialidad);
        System.out.printf ("Salario: %7.2f euros\n", this.salario);
    }
}

```

### 4.3. Clases y métodos finales

El modificador final en clases y métodos tienen un comportamiento diferente al que tienen en atributos y variables.

Una clase declarada como final no puede ser heredada (no puede tener clases derivadas. Un método declarado como final no podrá ser redefinido en la clase derivada.

```

[modificador_acceso] final class nombreClase [herencia] [interfaces]
[modificador_acceso] final <tipo> <nombreMetodo> ([parámetros]) [excepciones]

```

El modificador final también puede ir acompañando a un parámetro de un método, en ese caso no se puede modificar el parámetro dentro del código del método:

```
public final metodoEscribir (int par1, final int par2).
```

## 5. Interfaces

Podemos llegar a tener una clase abstracta donde todos sus métodos sean abstractos. De este modo, solo damos a las subclases el marco de comportamiento, sin ningún método implementado. La idea de la interfaz (o interface) es disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación, no necesariamente jerárquica.

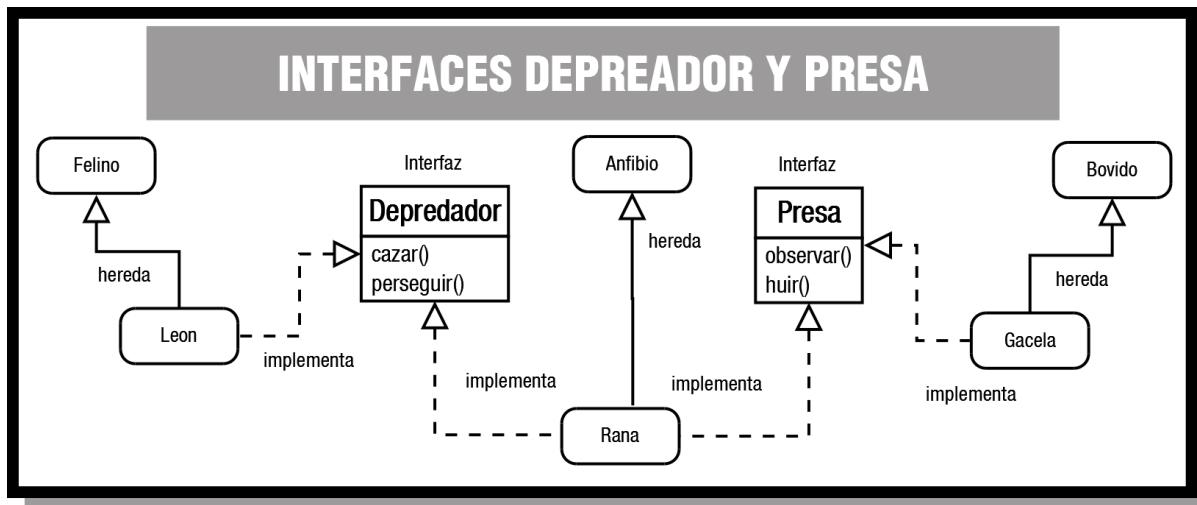
Una interfaz es una lista de declaraciones de métodos sin implementar que caracterizan un determinado comportamiento. Si se desea que una clase tenga ese comportamiento, debe implementar todos los métodos de la interfaz.

La diferencia entre herencia e interfaz es que en la herencia, la clase A es una especialización de B y en el caso de la interfaz A implementa el comportamiento o los métodos establecidos en B.

Ejemplo:

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieras que lleven a cabo están relacionadas con el hecho de que algunos animales sean depredadores (por ejemplo: observar una presa, perseguirla, comérsela, etc.) o sean presas (observar, huir, esconderse, etc.). Si creas la clase León, esta clase podría implementar una interfaz Depredador, mientras que otras clases como Gacela implementarían las acciones de la interfaz Presa. Por otro lado, podrías tener también el caso de la clase Rana, que implementaría las acciones de la interfaz Depredador (pues es cazador de

pequeños insectos), pero también la de Presa (pues puede ser cazado y necesita las acciones necesarias para protegerse).



## 5.1. Concepto de interfaz

Una interfaz se encarga de establecer que comportamientos hay que tener (qué métodos), pero no dice como se deben llevar a cabo (implementación).

Los métodos de la interfaz son públicos y se deben definir todos y cada uno de ellos en las subclases.

En definitiva, una interfaz se encarga de establecer unas líneas generales sobre los comportamientos de todos los objetos que implementan esa interfaz, pero no indican lo que el objeto es, que de eso se encarga la misma clase y sus superclases. Solo indica las acciones que el objeto debe ser capaz de realizar. Por eso en java muchas interfaces terminan con el sufijo -able, -or, o -ente (capacidad o habilidad): configurable, serializable, modifiable, administrador, servidor, buscador... dando así la idea de que se tiene que llevar a cabo el conjunto de acciones especificadas en la interfaz.

Una clase puede adoptar distintos modelos de comportamiento establecidos en diferentes interfaces. Es decir una clase puede implementar varias interfaces.

Ejemplo:

Imagínate por ejemplo la clase **Coche**, subclase de **Vehículo**. Los coches son **vehículos a motor**, lo cual implica una serie de acciones como, por ejemplo, **arrancar el motor** o **detener el motor**. Esta acción no la puedes heredar de **Vehículo**, pues no todos los vehículos tienen porqué ser a motor (piensa por ejemplo en una clase **Bicicleta**), y no puedes heredar de otra clase pues ya heredas de **Vehículo**. Una solución podría ser crear una **interfaz Arrancable**, que proporcione los métodos típicos de un **objeto a motor** (no necesariamente vehículos). De este modo la clase **Coche** sigue siendo subclase de **Vehículo**, pero también implementaría los comportamientos de la interfaz **Arrancable**, los cuales podrían ser también implementados por otras clases, hereden o no de **Vehículo** (por ejemplo una clase **Motocicleta** o bien una clase **Motosierra**). La clase **Coche** implementará su método **arrancar** de una manera, la clase **Motocicleta** lo hará de otra (aunque bastante parecida) y la clase **Motosierra** de otra forma probablemente muy diferente, pero todos tendrán su propia versión del método **arrancar** como parte de la interfaz **Arrancable**.

Según esta concepción, podrías hacerte la siguiente pregunta: **¿podrá una clase implementar varias interfaces?** La respuesta en este caso sí es afirmativa.

### 5.1.1. ¿Clase abstracta o interfaz?

Existe un parecido formal entre una clase abstracta y una interfaz, pudiéndose en ocasiones utilizar de manera indistinta para el mismo fin, pero existen algunas diferencias muy importantes.

- Una clase no puede heredar varias clases, aunque sean abstractas (herencia múltiple), pero si puede implementar una o varias interfaces y además seguir heredando de una clase.
- Una interfaz no puede definir métodos, tan solo declara o enumera.
- Una interfaz puede hacer que dos clases tengan un mismo comportamiento, aunque no hereden de la misma superclase.
- Las interfaces permiten establecer un comportamiento sin apenas dar detalles, ya que esos detalles dependen de como cada clase decida implementar la interfaz.
- Las interfaces tienen su propia jerarquía, diferente e independiente a la jerarquía de clases.

Una clase abstracta proporciona una interfaz disponible solo a través de la herencia, es decir, solo quienes hereden de esa clase abstracta dispondrán de esa interfaz.

Por tanto, si quisieramos erróneamente que una clase que no hereda de una clase abstracta implemente sus métodos solo se podría realizar volviendo a escribir la jerarquía de clases (redundante) o convertir esa clase en clase derivada de la clase abstracta aunque no tengan nada que ver.

En cambio, una interfaz al poder ser implementada por cualquier clase, nos dejará compartir un determinado comportamiento sin tener que forzar una relación de herencia que no existe.

Así pues, la idea de compartir un determinado comportamiento o interfaz es otro tipo de relación entre clases. Dos clases pueden tener en común un determinado comportamiento sin tener que forzar una relación de herencia.

#### Recomendación:

Si sólo vas a proporcionar una lista de métodos abstractos (interfaz), sin definiciones de métodos ni atributos de objeto, suele ser recomendable definir una interfaz antes que clase abstracta. Es más, cuando vayas a definir una supuesta clase base, puedes comenzar declarándola como interfaz y sólo cuando veas que necesitas definir métodos o variables miembro, puedes entonces convertirla en clase abstracta (no instanciable) o incluso en una clase instanciable.

## 5.2. Definición de interfaces

En java es similar a la declaración de una clase, con algunas variaciones.

- Se utiliza la palabra reservada interface en lugar de class.
- Puede utilizarse el modificador public. Si se incluye, la interfaz debe tener el mismo nombre que el archivo .java en el que se encuentra, si no se indica, el acceso será por omisión o de paquete como ocurre con las clases.
- Los miembros de la interfaz son public de manera implícita, no es necesario indicar el modificador pero se puede hacer.

- Todos los atributos son de tipo final y public, tampoco es necesario especificarlo. Hay que darles valor inicial.
- Todos los métodos son abstractos de manera implícita, no hace falta indicarlo. No tienen cuerpo, tan solo cabecera, se utiliza punto y coma para terminar de definirlas.

```
[public] interface <NombreInterfaz> {
    [public] [final] <tipo1> <atributo1>= <valor1>;
    [public] [final] <tipo2> <atributo2>= <valor2>;
    ...
    [public] [abstract] <tipo_devuelto1> <nombreMetodo1> ([lista_parámetros]);
    [public] [abstract] <tipo_devuelto2> <nombreMetodo2> ([lista_parámetros]);
    ...
}
```

Ejemplo:

```
public interface Depredador {
    void localizar (Animal presa);
    void cazar (Animal presa);
    ...
}
```

### Ejercicio resuelto

**Crea una interfaz en Java cuyo nombre sea Imprimible que contenga un método útil para mostrar el contenido de una clase:**

Método devolverContenidoString, que crea un String con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves: "{<nombre\_atributo\_1>=<valor\_atributo\_1>, ..., <nombre\_atributo\_n>=<valor\_atributo\_n>}".

```
public interface Imprimible {
    String devolverContenidoString ();
}
```

## 5.3. Implementación de interfaces

Todas las clases que implementan una determinada interfaz están obligadas a proporcionar una definición o implementación de los métodos de la interfaz.

Dada una interfaz, cualquier clase puede especificar dicha interfaz mediante la palabra reservada implements:

```
class NombreClase implements NombreInterfaz {
```

Es posible indicar varias interfaces separándolas por comas:

```
class NombreClase implements NombreInterfaz1, NombreInterfaz2, ... {
```

Al redefinir los métodos se debe hacer con acceso público o se produce error de compilación. De modo que ni se pueden restringir los permisos de acceso a la herencia de clases, ni tampoco a la implementación de interfaces.

**Recomendación:**

Para añadir la implementación de los métodos de una clase que implementa una interfaz, podemos utilizar la funcionalidad de Netbeans de agregarlos de forma automática exactamente igual que hicimos con los métodos abstractos. Agiliza la inserción de código pues nos ahorramos escribir la cabecera de todos los métodos a los que tenemos que dar implementación.

Ejemplo:

```
class Leon implements Depredador {  
    void localizar (Animal presa) {  
  
        // Implementación del método localizar para un león  
  
        ...  
  
    }  
}
```

En el caso de que se pudiera ser depredador y presa se deben implementar ambas interfaces.

```
class Rana implements Depredador, Presa {
```

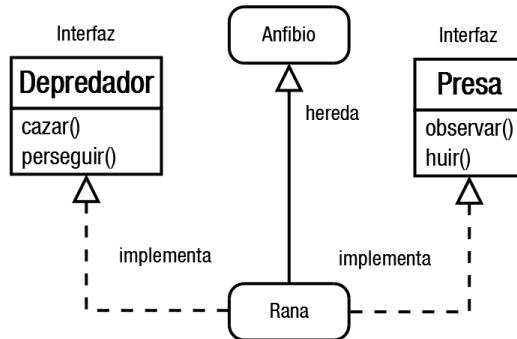
## 5.4. Simulación de la herencia múltiple mediante el uso de interfaces

Una interfaz no tiene espacio de almacenamiento asociado, es decir, no tiene implementación.

En java no se permite la herencia múltiple, pero se puede simular definiendo interfaces que indiquen los comportamientos o métodos que debería tener según pertenezca a una supuesta clase, pero sin implementar ningún método concreto ni atributos de objeto, solo interfaz.

Ejemplo: Una clase X puede implementar las interfaces A, B y C, que la dotan de comportamientos que deseaba heredar de las clases A, B y C, a su vez puede heredar de otra clase Y, que le proporciona sus características dentro de su jerarquía de objeto.

## IMPLEMENTACIÓN DE LAS INTERFACES DEPREDADOR Y PRESA



De este modo, con una herencia y varias interfaces se consiguen resultados similares a la herencia múltiple.

Puede darse el caso de la colisión de nombres cuando se implementan dos interfaces con un método llamado igual.

- Si los dos métodos tienen distintos parámetros ocurrirá sobrecarga de métodos y no habrá problema.
- Si tienen un valor de retorno de un tipo diferente se produce error de compilación.
- Si los dos métodos son iguales de parámetro y tipo devuelto, solo se podrá implementar uno de los dos métodos.

## 5.5. Herencia de interfaces

Las interfaces permiten herencia, se hace con la palabra reservada extends, pero en este caso si se permite herencia múltiple de interfaces. Si se hereda más de una interfaz se separan por comas.

```
public interface InterfazCompleja extends InterfazUno, InterfazDos {  
    // Métodos y constantes de la interfaz compleja  
}
```

## 6. Polimorfismo

Es otro de los grandes pilares en los que se sustenta la POO y también establece diferencias entre interfaz e implementación (entre el qué y el como).

El polimorfismo es fundamental a la hora de manipular muchos objetos de clases distintas como si fueran de la misma clase.

Te permite mejorar la organización y legibilidad del código, así como desarrollar aplicaciones fáciles de ampliar e incorporar nuevas funcionalidades.

### 6.1. Concepto de polimorfismo

Consiste en poder referenciar un objeto de una determinada clase como si fuera de otra (que sea subclase).

Un método polimórfico ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en tiempo de ejecución en lugar de en tiempo de compilación. Para eso es necesario utilizar métodos que pertenecen a una superclase y que están implementados en las subclases de una forma particular.

Solo en tiempo de ejecución (una vez instanciada una u otra clase) se conoce realmente qué método de qué subclase es invocado.

Esto ayuda a desentenderte del tipo de objeto específico para centrarte en el objeto genérico y se pueden manipular objetos hasta cierto punto desconocidos en tiempo de compilación.

El polimorfismo ofrece la posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una de sus subclases. Esto te va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus superclases. Se puede llevar a cabo tanto con superclases (abstractas o no), como con interfaces.

Ejemplo:

Imagina que estás trabajando con las clases Alumno y Profesor y que en determinada zona del código podrías tener objetos, tanto de un tipo como de otro, pero eso sólo se sabrá según vaya discurriendo la ejecución del programa. En algunos casos, es posible que un determinado objeto pudiera ser de la clase Alumno y en otros de la clase Profesor, pero en cualquier caso serán objetos de la clase Persona. Eso significa que la llamada a un método de la clase Persona (por ejemplo devolverContenidoString) en realidad será en unos casos a un método (con el mismo nombre) de la clase Alumno y, en otros, a un método (con el mismo nombre también) de la clase Profesor. Esto será posible hacerlo gracias a la ligadura dinámica.

## 6.2. Ligadura dinámica

La conexión que tiene durante una llamada a un método se suele llamar ligadura, vinculación o enlace (binding). Si al vinculación se realiza en compilación se llama ligadura estática o vinculación temprana.

En POO existe la ligadura dinámica, conocida como vinculación tardía, enlace tardío o late binding.

Hace posible que sea el tipo de objeto instanciado, obtenido mediante el constructor utilizado para crear el objeto y no el tipo de la referencia, lo que determine qué versión de método será invocado. El tipo de objeto al que apunta la variable de referencia solo podrá ser conocido en la ejecución del programa.

### Ejercicio resuelto

**Imagínate una clase que represente a instrumento musical genérico (Instrumento) y dos subclases que representen tipos de instrumentos específicos (por ejemplo Flauta y Piano). Todas las clases tendrán un método tocarNota, que será específico para cada subclase.**

**Haz un pequeño programa de ejemplo en Java que utilice el polimorfismo (referencias a la superclase que se convierten en instancias específicas de subclases) y la ligadura dinámica (llamadas a un método que aún no están resueltas en tiempo de compilación) con estas clases que representan instrumentos musicales. Puedes implementar el método tocarNota mediante la escritura de un mensaje en pantalla.**

La clase Instrumento podría tener un único método (tocarNota):

```
public abstract class Instrumento {  
    public void tocarNota (String nota) {  
        System.out.printf ("Instrumento: tocar nota %s.\n", nota);  
    }  
}
```

En el caso de las clases Piano y Flauta puede ser similar, heredando de Instrumento y redefiniendo el método tocarNota:

```
public class Flauta extends Instrumento {  
    @Override  
    public void tocarNota (String nota) {  
        System.out.printf ("Flauta: tocar nota %s.\n", nota);  
    }  
}  
  
public class Piano extends Instrumento {  
    @Override  
    public void tocarNota (String nota) {  
        System.out.printf ("Piano: tocar nota %s.\n", nota);  
    }  
}
```

A la hora de declarar una referencia a un objeto de tipo instrumento, utilizamos la superclase (Instrumento):

```
Instrumento instrumento1; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
```

Sin embargo, a la hora de instanciar el objeto, utilizamos el constructor de alguna de sus subclases (Piano, Flauta, etc.):

```
if (<condición>) {  
    // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)  
    instrumento1= new Piano ();  
}  
  
else if (<condición>) {  
    // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)  
    instrumento1= new Flauta ();
```

```
    } else {  
        ...  
    }
```

Finalmente, a la hora de invocar el método tocarNota, no sabremos a qué versión (de qué subclase) de tocarNota se estará llamando, pues dependerá del tipo de objeto (subclase) que se haya instanciado. Se estará utilizando por tanto la ligadura dinámica:

```
// Interpretamos una nota con el objeto instrumento1  
  
// No sabemos si se ejecutará el método tocarNota de Piano o de Flauta  
  
// (dependerá de la ejecución)  
  
instrumento1.tocarNota ("do"); // Ejemplo de ligadura dinámica (tiempo de ejecución)
```

## 6.3. Limitaciones de la ligadura dinámica

Existe una importante restricción en el uso de esta capacidad, pues el tipo de referencia limita cuáles son los métodos a utilizar y los atributos a los que acceder.

No se puede acceder a los miembros específicos de una subclase a través de una referencia a una superclase. Solo se puede acceder a los miembros declarados en la superclase, aunque la definición sea la de la subclase.

Ejemplo:

En el ejemplo de las clases Persona, Profesor y Alumno, el polimorfismo nos permitiría declarar variables de tipo Persona y más tarde hacer con ellas referencia a objetos de tipo Profesor o Alumno, pero no deberíamos intentar acceder con esa variable a métodos que sean específicos de la clase Profesor o de la clase Alumno, tan solo a métodos que sabemos que van a existir seguro en ambos tipos de objetos (métodos de la superclase Persona).

### Ejercicio resuelto

Si tuviéramos diferentes variables referencia a objetos de las clases Alumno y Profesor tendrías algo así:

```
Alumno obj1;  
  
Profesor obj2;  
  
...  
  
// Si se dan ciertas condiciones el objeto será de tipo Alumno y lo tendrás en obj1  
System.out.printf ("Nombre: %s\n", obj1.getNombre());  
  
// Si se dan otras condiciones el objeto será de tipo Profesor y lo tendrás en obj2  
System.out.printf ("Nombre: %s\n", obj2.getNombre());
```

Pero si pudieras tratar de una manera más genérica la situación, podrías intentar algo así:

```
Persona obj;
```

```
// Si se dan ciertas condiciones el objeto será de tipo Alumno y por tanto lo instanciarás como tal  
obj = new Alumno (<parámetros>);  
  
// Si se otras condiciones el objeto será de tipo Profesor y por tanto lo instanciarás como tal  
obj = new Profesor (<parámetros>);
```

De esta manera la variable obj obj podría contener una referencia a un objeto de la superclase Persona de subclase Alumno o bien de subclase Profesor (polimorfismo).

Esto significa que independientemente del tipo de subclase que sea (Alumno o Profesor), podrás invocar a métodos de la superclase Persona y durante la ejecución se resolverán como métodos de alguna de sus subclases:

```
//En tiempo de compilación no se sabrá de qué subclase de Persona será obj.  
  
//Habrá que esperar la ejecución para que el entorno lo sepa e invoque al método adecuado.  
  
System.out.printf ("Contenido del objeto usuario: %s\n", stringContenidoUsuario);
```

Por último recuerda que debes de proporcionar constructores a las subclases Alumno y Profesor que sean "compatibles" con algunos de los constructores de la superclase Persona, pues al llamar a un constructor de una subclase, su formato debe coincidir con el de algún constructor de la superclase (como debe suceder en general con cualquier método que sea invocado utilizando la ligadura dinámica).

## 6.4. Interfaces y polimorfismo

Es posible llevar a cabo el polimorfismo usando interfaces.

Un objeto puede tener una referencia cuyo tipo sea una interfaz, pero para que el compilador lo permita, la clase cuyo constructor se utilice para crear el objeto debe implementar esa interfaz, ya sea por si misma o porque la implemente alguna superclase.

Un objeto cuya referencia es de tipo interfaz, solo puede utilizar aquellos métodos definidos en la interfaz, no podrá utilizar atributos y métodos específicos de su clase.

Estas referencias permiten unificar de una manera estricta la forma de utilizarse objetos que pertenecen a clases muy diferentes pero que implementan la misma interfaz.

De esta manera podemos hacer referencias a objetos sin relación jerárquica utilizando la misma variable.

Ejemplo:

Si tenías una variable de tipo referencia a la interfaz Arrancable, podrías instanciar objetos de tipo Coche o Motosierra y asignarlos a esa referencia (teniendo en cuenta que ambas clases no tienen una relación de herencia). Sin embargo, tan solo podrás usar en ambos casos los métodos y los atributos de la interfaz Arrancable (por ejemplo arrancar) y no los de Coche o los de Motosierra (sólo los genéricos, nunca los específicos).

Otro ejemplo:

En el caso de las clases Persona, Alumno y Profesor, podrías declarar, por ejemplo, variables del tipo Imprimible:

```
Imprimible obj; // Imprimible es una interfaz y no una clase
```

Con este tipo de referencia podrías luego apuntar a objetos tanto de tipo Profesor como de tipo Alumno, pues ambos implementan la interfaz Imprimible:

```
obj= new Alumno (nombre, apellidos, fecha, grupo, nota); // Polimorfismo con interfaces  
...  
// En otras circunstancias podría suceder esto:  
obj= new Profesor (nombre, apellidos, fecha, especialidad, salario); // Polimorfismo con interfaces  
...
```

```
String contenido;  
contenido= obj.devolverContenidoString(); // Ligadura dinámica con interfaces
```

## 6.5. Conversión de paquetes

Si deseamos tener acceso a todos los métodos y atributos específicos del objeto subclase, podemos realizar una conversión explícita (casting), logrando que convierta la referencia más general en la del tipo específico.

Para ello es obligatorio que exista herencia entre ellas. Se realizará conversión implícita automática siempre que sea necesario, pues un objeto de tipo subclase contendrá toda la información necesaria para ser considerada un objeto superclase.

La conversión de superclase a subclase se debe hacer de forma explícita porque podría dar lugar a errores por falta de información (atributos) o de métodos con ClassCastException.

Ejemplo:

imagina que tienes una clase A y una clase B, subclase de A:

```
class ClaseA {  
    public int atrib1;  
}  
  
class ClaseB extends ClaseA {  
    public int atrib2;  
}
```

A continuación declaras una variable referencia a la clase A (superclase) pero sin embargo le asignas una referencia a un objeto de la clase B (subclase) haciendo uso del polimorfismo:

```
A obj; // Referencia a objetos de la clase A  
obj= new B (); // Referencia a objetos clase A, pero apunta realmente a objeto clase B (polimorfismo)
```

El objeto que acabas de crear como instancia de la clase B (subclase de A) contiene más información que la que la referencia obj te permite en principio acceder sin que el compilador genere un error (pues es de clase A). En concreto los objetos de la clase B disponen de atrib1 y atrib2, mientras que los objetos de la clase A sólo de atrib1. Para acceder a esa información adicional de la clase especializada (atrib2) tendrás que realizar una conversión explícita (casting):

```
// Casting del tipo A al tipo B (funcionará bien porque el objeto es realmente del tipo B)  
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib2);
```

Sin embargo si se hubiera tratado de una instancia de la clase A y hubieras intentado acceder al miembro atrib2, se habría producido una excepción de tipo ClassCastException:

```
A obj; // Referencia a objetos de la clase A  
  
obj= new A (); // Referencia a objetos de la clase A, y apunta realmente a un objeto de la clase A  
  
// Casting del tipo A al tipo B (puede dar problemas porque el objeto es realmente del tipo A):  
  
// Funciona (la clase A tiene atrib1)  
  
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib1);  
  
// ¡Error en ejecución! (la clase A no tiene atrib2). Producirá una ClassCastException.  
  
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib2);
```

## Anexo I - Elaboración de los constructores de la clase Rectángulo

Intenta redactar los constructores de la clase Rectangulo teniendo en cuenta ahora su nueva estructura de atributos (dos objetos de la clase Punto, en lugar de cuatro elementos de tipo double):

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros, x1, y1, x2, y2, que cree un rectángulo con los vértices (x1, y1) y (x2, y2).
3. Un constructor con dos parámetros, punto1, punto2, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros, base y altura, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor copia.

Durante el proceso de creación de un objeto (constructor) de la clase contenedora (en este caso Rectangulo) hay que tener en cuenta también la creación (llamada a constructores) de aquellos objetos que son contenidos (en este caso objetos de la clase Punto).

En el caso del primer constructor, habrá que crear dos puntos con las coordenadas (0,0) y (1,1) y asignarlos a los atributos correspondientes (vertice1 y vertice2):

```

public Rectangulo ()
{
    this.vertice1= new Punto (0,0);
    this.vertice2= new Punto (1,1);
}

```

Para el segundo constructor habrá que crear dos puntos con las coordenadas x1, y1, x2, y2 que han sido pasadas como parámetros:

```

public Rectangulo (double x1, double y1, double x2, double y2)
{
    this.vertice1= new Punto (x1, y1);
    this.vertice2= new Punto (x2, y2);
}

```

En el caso del tercer **constructor** puedes utilizar directamente los dos puntos que se pasan como parámetros para construir los vértices del rectángulo:

Ahora bien, esto podría ocasionar un efecto colateral no deseado si esos objetos de tipo Punto son modificados en el futuro desde el código cliente del constructor (no sabes si esos puntos fueron creados especialmente para ser usados por el rectángulo o si pertenecen a otro objeto que podría modificarlos más tarde).

Por tanto, para este caso quizás fuera recomendable crear dos nuevos puntos a imagen y semejanza de los puntos que se han pasado como parámetros. Para ello tendrías dos opciones:

1. Llamar al constructor de la clase Punto con los valores de los atributos (x, y).Llamar al constructor copia de la clase Punto, si es que se dispone de él.
2. Llamar al **constructor copia** de la clase **Punto**, si es que se dispone de él.

Aquí tienes las dos posibles versiones:

Constructor que “extrae” los atributos de los parámetros y crea nuevos objetos:

```

public Rectangulo (Punto vertice1, Punto vertice2)
{
    this.vertice1= vertice1;
    this.vertice2= vertice2;
}

```

Constructor que crea los nuevos objetos mediante el constructor copia de los parámetros:

```

public Rectangulo (Punto vertice1, Punto vertice2)
{
    this.vertice1= new Punto (vertice1.obtenerX(), vertice1.obtenerY() );
    this.vertice2= new Punto (vertice2.obtenerX(), vertice2.obtenerY() );
}

```

```
}
```

En este segundo caso puedes observar la utilidad de los **constructores de copia** a la hora de tener que **clonar** objetos (algo muy habitual en las inicializaciones).

Para el caso del **constructor** que recibe como parámetros la base y la altura, habrá que crear sendos vértices con valores (0,0) y (0 + base, 0 + altura), o lo que es lo mismo: (0,0) y (base, altura).

```
public Rectangulo (Punto vertice1, Punto vertice2)
{
    this.vertice1= new Punto (vertice1 );
    this.vertice2= new Punto (vertice2 );
}
```

Quedaría finalmente por implementar el constructor copia:

```
// Constructor copia
public Rectangulo (Rectangulo r) {
    this.vertice1= new Punto (r.obtenerVertice1() );
    this.vertice2= new Punto (r.obtenerVertice2() );
}
```

En este caso nuevamente volvemos a clonar los atributos vertice1 y vertice2 del objeto r que se ha pasado como parámetro para evitar tener que compartir esos atributos en los dos rectángulos.

## Anexo II - Métodos para las clases heredadas Alumno y Profesor

Dadas las clases Alumno y Profesor que has utilizado anteriormente, implementa métodos get y set en las clases Alumno y Profesor para trabajar con sus cinco atributos (tres heredados más dos específicos).

```
public class Alumno extends Persona {
    protected String grupo;
    protected double notaMedia;
    // Método getNombre
    public String getNombre (){
        return nombre;
    }
}
```

```

// Método getApellidos

public String getApellidos (){
    return apellidos;
}

// Método getFechaNacim

public GregorianCalendar getFechaNacim (){
    return this.fechaNacim;
}

// Método getGrupo

public String getGrupo (){
    return grupo;
}

// Método getNotaMedia

public double getNotaMedia (){
    return notaMedia;
}

// Método setNombre

public void setNombre (String nombre){
    this.nombre= nombre;
}

// Método setApellidos

public void setApellidos (String apellidos){
    this.apellidos= apellidos;
}

// Método setFechaNacim

public void setFechaNacim (GregorianCalendar fechaNacim){
    this.fechaNacim= fechaNacim;
}

// Método setGrupo

public void setGrupo (String grupo){
    this.grupo= grupo;
}

// Método setNotaMedia

public void setNotaMedia (double notaMedia){
    this.notaMedia= notaMedia;
}

```

```
}
```

Si te fijas, puedes utilizar sin problema la referencia `this` a la propia clase con esos atributos heredados, pues pertenecen a la clase: `this.nombre`, `this.apellidos`, etc.

```
public class Profesor extends Profesor {  
  
    String especialidad;  
  
    double salario;  
  
    // Método getNombre  
  
    public String getNombre (){  
  
        return nombre;  
  
    }  
    // Método getApellidos  
  
    public String getApellidos (){  
  
        return apellidos;  
  
    }  
  
    // Método getFechaNacim  
  
    public GregorianCalendar getFechaNacim (){  
  
        return this.fechaNacim;  
  
    }  
  
    // Método getEspecialidad  
  
    public String getEspecialidad (){  
  
        return especialidad;  
  
    }  
  
    // Método getSalario  
  
    public double getSalario (){  
  
        return salario;  
  
    }  
  
    // Método setNombre  
  
    public void setNombre (String nombre){  
  
        this.nombre= nombre;  
  
    }  
  
    // Método setApellidos  
  
    public void setApellidos (String apellidos){  
  
        this.apellidos= apellidos;  
  
    }  
  
    // Método setFechaNacim
```

```

public void setFechaNacim (GregorianCalendar fechaNacim){

    this.fechaNacim= fechaNacim;

}

// Método setSalario

public void setSalario (double salario){

    this.salario= salario;

}

// Método setEspecialidad

public void setEspecialidad (String especialidad){

    this.especialidad= especialidad;

}

}

```

Una conclusión que puedes extraer de este código es que has tenido que escribir los métodos get y set para los tres atributos heredados, pero ¿no habría sido posible definir esos seis métodos en la clase base y así estas dos clases derivadas hubieran también heredado esos métodos? La respuesta es afirmativa y de hecho es como lo vas a hacer a partir de ahora. De esa manera te habrías evitado tener que escribir seis métodos en la clase Alumno y otros seis en la clase Profesor. Así que recuerda: se pueden heredar tanto los atributos como los métodos.

Aquí tienes un ejemplo de cómo podrías haber definido la clase Persona para que luego se hubieran podido heredar de ella sus métodos (y no sólo sus atributos):

```

public class Persona {

    protected String nombre;

    protected String apellidos;

    protected GregorianCalendar fechaNacim;

    // Método getNombre

    public String getNombre (){

        return nombre;

    }

    // Método getApellidos

    public String getApellidos (){

        return apellidos;

    }

    // Método getFechaNacim

    public GregorianCalendar getFechaNacim (){

        return this.fechaNacim;

    }
}

```

```
}

// Método setNombre

public void setNombre (String nombre){

    this.nombre= nombre;

}

// Método setApellidos

public void setApellidos (String apellidos){

    this.apellidos= apellidos;

}

// Método setFechaNacim

public void setFechaNacim (GregorianCalendar fechaNacim){

    this.fechaNacim= fechaNacim;

}

}
```

## Mapa Conceptual



# 8. Colecciones de datos

	Autor	(X) Xerach Casanova
	Clase	Programación
	Fecha	@Mar 14, 2021 9:39 PM

- [1. Introducción a las colecciones](#)
- [2. Clases y métodos genéricos \(I\)](#)
  - [2.1. Clases y métodos genéricos \(II\)](#)
- [3. Conjuntos \(I\)](#)
  - [3.1. Conjuntos \(II\)](#)
  - [3.2. Conjuntos \(III\)](#)
  - [3.3. Conjuntos \(IV\)](#)
  - [3.5. Conjuntos \(V\)](#)
- [4. Listas \(I\)](#)
  - [4.1. Listas \(II\)](#)
  - [4.2. Listas \(III\)](#)
  - [4.3. Listas \(IV\)](#)
- [5. Conjuntos de pares clave/valor](#)
- [6. Iteradores \(I\)](#)
  - [6.1. Iteradores \(II\)](#)
- [7. Algoritmos](#)
  - [7.1. Algoritmos \(II\)](#)
  - [7.2. Algoritmos \(III\)](#)

[Mapa conceptual](#)

## 1. Introducción a las colecciones

Una colección a nivel software es un grupo de elementos almacenados de forma conjunta en una misma estructura. Una colección o contenedor es un objeto que agrupa elementos múltiples en un objeto simple. Las colecciones se usan para almacenar, recuperar y manipular datos. Un array o vector no está incluido en el framework Collections.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permite manejar los grupos de objetos que a priori están relacionados entre sí (aunque no es obligatorio) y pueden trabajar con cualquier tipo de objeto (por eso se emplean genéricos).

Además permiten realizar algunas operaciones útiles sobre elementos almacenados, como búsqueda u ordenación. En algunos casos, los objetos almacenados deben cumplir

algunas condiciones implementando algunas interfaces para hacer uso de esos algoritmos.

En java, el uso de las colecciones es bastante más sencillo que en otros lenguajes. Parten de una serie de interfaces básicas que definen un modelo de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados. La interfaz inicial, a partir de la cual están creadas todas las colecciones es `java.util.Collection` y define todas las operaciones comunes de las colecciones derivadas.

Hay que tener en cuenta que una colección es una interfaz genérica donde "`<E>`" es el parámetro de tipo.

Las operaciones más importantes de cualquier colección son:

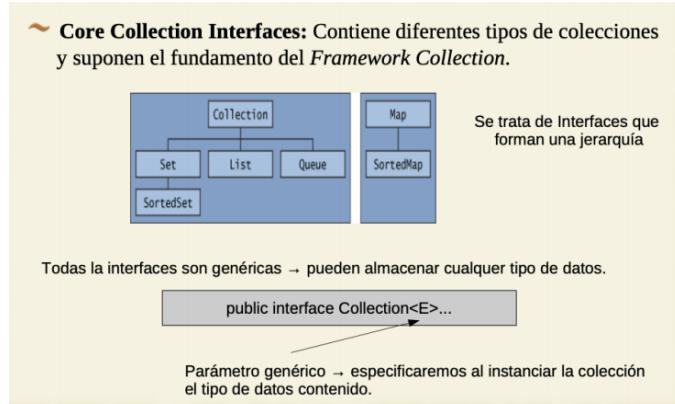
- Método **int size()**: retorna el número de elementos de la colección.
- Método **boolean isEmpty()**: retornará verdadero si la colección está vacía.
- Método **boolean contains (Object element)**: retornará verdadero si la colección tiene el elemento pasado como parámetro.
- Método **boolean add(E element)**: permitirá añadir elementos a la colección.
- Método **boolean remove (Object element)**: permitirá eliminar elementos de la colección.
- Método **Iterator <E> iterator()**: permitirá crear un iterador para recorrer los elementos de la colección.
- Método **Object[] toArray()**: permite pasar la colección a un array de objetos tipo Object.
- Método **containsAll(Collection<?> c)**: permite comprobar si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
- Método **addAll (Collection<? extends E> c)**: permite añadir todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base).
- Método **boolean removeAll(Collection<?> c)**: si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
- Método **boolean retainAll(Collection<?> c)**: si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
- Método **void clear()**: vaciar la colección.

Las ventajas de utilizar las colecciones son:

1. Reducen el esfuerzo de programación.
2. Incrementan la calidad de las aplicaciones.
3. Incrementan la interoperabilidad, muchas APIs utilizan clases o interfaces de colecciones como argumentos en sus métodos.

4. Reducen el esfuerzo de aprender nuevas APIs

5. Contribuyen a la reusabilidad del código.



## 2. Clases y métodos genéricos (I)

Las clases y métodos genéricos son un recurso de programación disponible en muchos lenguajes, con el objetivo de facilitar la reutilización de software, creando distintos métodos y clase que pueden trabajar con diferentes tipos de objetos, evitando la conversión de tipos.

Ejemplo de versión genérica y no genérica de un método compararTamano.

### Versión no genérica.

```
public class util {  
  
    public class util {public static int compararTamano(Object[] a, Object[] b) {  
        return a.length-b.length;  
    }  
  
}
```

### Versión genérica.

```
public class util {  
    public static <T> int compararTamano (T[] a, T[] b) {  
        return a.length-b.length;  
    }  
}
```

Los dos métodos permiten y comprueban si un array es mayor que otro. Retornan 0 si son iguales, mayor de cero si el array b es mayor y menor de cero si el array es mayor. La versión genérica incluye la expresión `<T>` antes del tipo retornado por el método. Esta es la definición de una variable o parámetro formal de tipo de la clase o método genérico (parámetro de tipo o parámetro genérico) y se puede utilizar a lo largo de todo el método

o clase haciendo referencia a cualquier clase con la que nuestro algoritmo tiene que trabajar.

Para invocar un método genérico, solo hay que realizar una invocación de tipo genérico, olvidándonos de las conversiones de tipo. Consiste en indicar qué clases o interfaces concretas se utilizarán en lugar de cada parámetro genérico "<t>" para después ejecutar ese algoritmo pasándole los argumentos correspondientes.

Cada clase o interfaz la podemos denominar tipo o tipo base y se da por sentado que los argumentos pasados al método genérico serán también de dicho tipo base.

### Invocación genérica y no genérica de un método Integer.

Invocación del método no genérico.	Invocación del método genérico.
<pre>Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.compararTamano ((Object[])a, (Object[])b);</pre>	<pre>Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.&lt;Integer&gt;compararTamano (a, b);</pre>

## 2.1. Clases y métodos genéricos (II)

Las clases genéricas permiten definir un parámetro de tipo o genérico que se puede utilizar a lo largo de toda la clase, facilitando crear clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre de la clase.

```
public class Util<T> {  
    T t1;  
    public void invertir(T[] array) {  
        for (int i = 0; i < array.length / 2; i++) {  
            t1 = array[i];  
            array[i] = array[array.length - i - 1];  
            array[array.length - i - 1] = t1;  
        }  
    }  
}
```

En el ejemplo anterior, la clase Util contiene el método invertir cuya función es invertir el orden de los elementos de cualquier array, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de dicha clase especificando el tipo base entre <>, justo detrás del nombre de la clase. Veamos un ejemplo:

```
Integer[] numeros={0,1,2,3,4,5,6,7,8,9};  
Util<Integer> u= new Util<Integer>();  
u.invertir(numeros);  
for (int i=0;i<numeros.length;i++) System.out.println(numeros[i]);
```

Simplemente a la hora de crear una instancia genérica se especifica el tipo, tanto en la definición (`Util<integer> u`) como en la creación (`new Util<Integer>()`).

Los parámetros de tipo de las clases genéricas no pueden ser datos primitivos: `int`, `short`, `double`, etc...

## 3. Conjuntos (I)

Los conjuntos son un tipo de colección que no admite duplicados.

La interfaz `java.util.Set` define cómo deben ser los conjuntos y extiende la interfaz `Collection`, aunque no añade ninguna operación nueva. Las implementaciones más usadas son las siguientes:

- **java.util.HashSet**. Conjunto que almacena los objetos usando tablas hash.
  - Ventajas: rapidez en el acceso a objetos.
  - Desventajas: no almacenan de forma ordenada y necesitan bastante memoria.
- **java.util.LinkedHashSet**. Conjunto que almacena objetos combinando tablas hash y listas enlazadas. El orden de almacenamiento es el de la inserción.
  - Ventajas: rapidez en acceso a objetos y almacena de forma ordenada.
  - Desventajas: necesitan bastante memoria y es algo más lenta que HashSet.
- **java.util.TreeSet**. Conjunto que almacena objetos usando estructuras conocidas como árboles rojo-negro.
  - Ventajas: los datos almacenados se ordenan por valor aunque se almacenen de manera desordenada.
  - Desventajas: son más lentas que las dos anteriores.

Para crear un conjunto se crea el HashSet se debe importar el paquete `java.util.HashSet` indicando el tipo de objeto que se va a almacenar, dado que es una clase genérica que puede trabajar cualquier tipo de datos.

```
HashSet<Integer> conjunto=new HashSet<Integer>();
```

Después podemos almacenar objetos dentro del conjunto con el método `add` (definido por la interfaz `Set`). Los objetos deben ser siempre del tipo especificado al crear el conjunto:

```
Integer n=new Integer(10);
if (!conjunto.add(n)) System.out.println("Número ya en la lista.");
```

Si el elemento ya está en el conjunto, el método add retorna false y podemos indicar que no se pueden insertar duplicados.

### 3.1. Conjuntos (II)

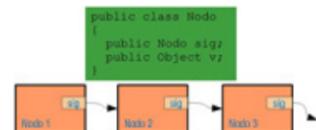
Para obtener los elementos almacenados en un conjunto se utilizan iteradores que permiten obtener elementos del conjunto uno a uno de forma secuencial. La forma más transparente de usarlo es a través de un for-each:

```
for (Integer i: conjunto) {  
    System.out.println("Elemento almacenado:"+i);  
}
```

### 3.2. Conjuntos (III)

La diferencia entre LinkedHashSet, TreeSet y HashSet radica en su funcionamiento interno.

**LinkedHashSet** es una estructura que funciona como lista enlazada, pero también usa tablas hash para acceder rápidamente a los elementos. Una lista enlazada es una estructura compuesta por nodos o elementos que forman la lista y que se van enlazando entre sí.



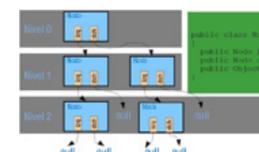
Un nodo contiene el dato almacenado y el siguiente nodo de la lista. Si no hay nodo, la variable que contiene el siguiente nodo es nula.

Las listas enlazadas tienen un montón de operaciones asociadas como son: inserción de un nodo al final, al principio, entre dos nodos, etc.

```
LinkedHashSet <Integer> t;  
t=new LinkedHashSet<Integer>();  
t.add(new Integer(4));  
t.add(new Integer(3));  
t.add(new Integer(1));  
t.add(new Integer(99));  
for (Integer i:t) System.out.println(i);
```

Salida por pantalla: 4 3 1 99 (los valores salen ordenados según el momento de inserción en el conjunto)

**Treeset** utiliza internamente árboles, que son como listas pero más complejos, en vez de tener un único elemento siguiente, puede tener dos o más, formando estructuras organizadas y jerárquicas.



Los nodos se diferencian en nodos padre y nodos hijos. Un nodo padre puede tener varios hijos asociados, dando lugar a una estructura de árbol invertido.

Puesto que un nodo hijo puede ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el del primer nivel que no tiene padre.

Los árboles son estructuras complejas de manejar con operaciones muy sofisticadas. Los árboles utilizados en TreeSet son árboles rojo-negro, que son auto-ordenados. Al insertar un elemento, se queda ordenado por su valor y al recorrer el árbol los elementos salen ordenados. También tiene operaciones a nivel interno como son inserción de nodos, eliminación, búsqueda de valor, etc.

La creación de un TreeSet es similar a la de un HashSet, solo sustituyendo su nombre. Tampoco admite duplicados y se utilizan los métodos vistos antes, existentes en la interfaz Set.

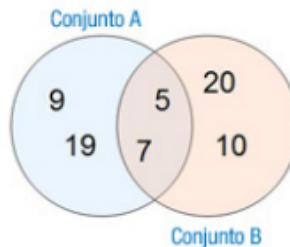
```
TreeSet <Integer> t;  
t=new TreeSet<Integer>();  
t.add(new Integer(4));  
t.add(new Integer(3));  
t.add(new Integer(1));  
t.add(new Integer(99));  
for (Integer i:t) System.out.println(i);
```

Salida por pantalla: 1 3 4 99 (el resultado sale ordenado por valor).

### 3.3. Conjuntos (IV)

Los conjuntos y las colecciones facilitan operaciones para poder combinar datos de varias colecciones.

**Ejemplo:**

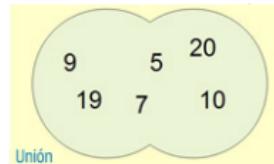


```
TreeSet<Integer> A= new TreeSet<Integer>();  
A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7  
  
LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();  
B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

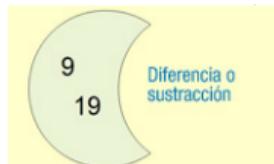
En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio Integer sin tener que hacer nada.

### Tipos de combinaciones:

- **Unión. A.add(B).** Añadir todos los elementos del conjunto B con el A. Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20.



- **Diferencia. A.removeAll(B).** Eliminar los elementos del conjunto B que puedan estar en el conjunto A. Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.



- **Intersección. A.RetainAll(B).** Retiene los elementos comunes de ambos conjuntos. Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.



## 3.5. Conjuntos (V)

Los TreeSet ordenan sus elementos de forma ascendente, pero tienen un conjunto de operaciones adicionales además de las que incluye por el hecho de ser conjunto, que permite, por ejemplo, cambiar la forma de ordenar los elementos. Es muy útil para tipos de objetos complejos. TreeSet es capaz de ordenar tipos básicos, como números, cadenas o fechas, pero otros tipo de objetos no se puede con tanta facilidad.

Para indicar a un TreeSet cómo tiene que ordenar los elementos debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello podemos utilizar la interfaz genérica java.util.Comparator, la cual se usa en general en algoritmos de ordenación, creando una clase que implementa dicha interfaz. Esta interfaz solo requiere un método que debe calcular si un objeto pasado por parámetro es mayor, menor o igual que otro del mismo tipo.

```
class ComparadorDeObjetos implements Comparator<Objeto> {  
    public int compare(Objeto o1, Objeto o2) { ... }  
}
```

La interfaz Comparator obliga a implementar un método (compare), el cual tiene dos parámetros, que son los dos elementos a comparar.

- Si el objeto (o1) es menor que el objeto (o2), debe retornar un número entero negativo. otra forma de verlo: Si el primer objeto (o1) debe ir antes que el segundo objeto (o2).
- Si el objeto (o1) es mayor que el objeto (o2), debe retornar un número positivo. otra forma de verlo: Si el primer objeto (o1) debe ir después que el segundo objeto (o2).
- Si ambos son iguales, debe retornar 0.

Una vez creado el comparador solo tenemos que pasarlo como parámetro en el momento de la creación al TreeSet y los datos internamente mantendrán dicha ordenación.

### Ejemplo:

Tenemos un objeto en una clase como la siguiente:

```
class Objeto {
    public int a;
    public int b;
}
```

Al añadirlos en un TreeSet, estos se tienen que ordenar de forma que la suma de sus atributos (a y b) sea descendente, ¿como sería el comparador?

```
class ComparadorDeObjetos implements Comparable<Objeto> {
    @Override
    public int compare(Objeto o1, Objeto o2) {

        int sumao1=o1.a+o1.b;
        int sumao2=o2.a+o2.b;

        if (sumao1<sumao2) return 1;
        else if (sumao1>sumao2) return -1;
        else return 0;
    }
}
```

## 4. Listas (I)

Las listas son elementos de programación algo más avanzados que los conjuntos. Amplían el conjunto de operaciones de colecciones añadiendo operaciones extra:

- Pueden almacenar duplicados, si no queremos duplicados se debe verificar manualmente antes de la inserción.
- Acceso posicional. Se puede acceder a un elemento indicando su posición en la lista.
- Búsqueda. Se puede buscar y obtener posición. En las colecciones solo se puede saber si un conjunto contenía o no un elemento, devolviendo true o false.

- **Extracción de sublistas.** Podemos obtener una lista que contenga solo una parte de los elementos.

Java, para las listas dispone de la interfaz `java.util.List` y dos implementaciones: `java.util.LinkedList` y `java.util.ArrayList`. Todos los métodos de la interfaz `List` están presentes en ambas interfaces y son:

- **E get(int index).** Permite obtener un elemento partiendo de su posición.
- **E set(int index, E element).** Permite cambiar el elemento almacenado en una posición de la lista por otro que se le pase por parámetro.
- **void add (int index, E Element).** Se añade el elemento en la lista en una posición concreta, desplazando los siguientes.
- **E remove (int index).** Permite eliminar un elemento indicando su posición.
- **boolean addAll(int index, Collection <?extends E> c).** Permite insertar una colección pasada por parámetro en una posición de la lista desplazando al resto de elementos siguientes.
- **int indexOf(Object o).** Permite conocer la posición de un elemento y si no está retorna -1.
- **int lastIndexOf(Object o).** Permite obtener la última ocurrencia del objeto en la lista.
- **List<E> subList(int from, int to).** Genera una sublista con los elementos correspondidos entre la posición inicial incluida y la posición final no incluida.

Todos los elementos de una lista comienzan a enumerarse por 0.

## 4.1. Listas (II)

Hay que recordar que para hacer uso de `LinkedList` y `ArrayList` hay que importar los paquetes correspondientes.

En el siguiente ejemplo creamos un `LinkedList` con números enteros. El contenido de la lista al final será: 2, 3 y 5

```
LinkedList<Integer> t=new LinkedList<Integer>(); // Declaración y creación del LinkedList de enteros.

t.add(1); // Añade un elemento al final de la lista.
t.add(3); // Añade otro elemento al final de la lista.
t.add(1,2); // Añade en la posición 1 el elemento 2.
t.add(t.get(1)+t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.
t.remove(0); // Elimina el primer elemento de la lista.

for (Integer i: t) System.out.println("Elemento:" + i); // Muestra la lista.
```

En este otro ejemplo creamos un `ArrayList`, el resultado final será 10 y 12.

```
ArrayList<Integer> al=new ArrayList<Integer>(); // Declaración y creación del ArrayList de enteros.
```

```
al.add(10); al.add(11); // Añadimos dos elementos a la lista.  
al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego lo reemplazam
```

En el ejemplo siguiente utilizamos sublistas. Con el método size se obtiene el tamaño de la lista. El método subList extrae una sublista de la lista desde la posición indicada por parámetro hasta la posición indicada por parámetro. El método addAll añade todos los elementos de la sublista al arraylist anterior.

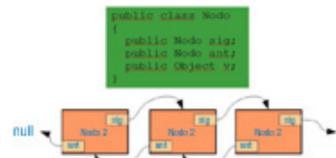
```
al.addAll(0, t.subList(1, t.size()));
```

Se debe tener en cuenta que si se realiza una operación sobre una sublista, también repercute sobre la lista original. En el siguiente ejemplo, borrando de la sublista también se borra de la lista.

```
al.subList(0, 2).clear();
```

## 4.2. Listas (III)

La diferencia entre LinkedList y ArrayList es que los LinkedList utilizan listas doblemente enlazadas, que son listas enlazadas que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos, estos nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento.



Al tener un doble enlace significa que cada nodo almacena la información del siguiente y del anterior. Si un nodo no tiene siguiente o anterior se almacena null.

En el caso de los ArrayList se implementan utilizando arrays que se van redimensionando de forma transparente conforme se necesita más o menos espacio. Los ArrayList son más rápidos en cuanto a acceso a los elementos, ya que en una lista doblemente enlazada hay que recorrer la lista. En cambio eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada.

Sabiendo lo anterior, llegamos a la conclusión que si se van a realizar muchas operaciones de eliminación sobre una lista, se debe utilizar LinkedList, y si la mayoría de operaciones consisten en insertar o consultar por posición, conviene usar ArrayList.

LinkedList implementa las interfaces java.util.Queue y java.util.Deque, las cuales permiten utilizar las listas como si fueran una cola de prioridad o una pila respectivamente.

Las colas son conocidas como colas de prioridad (FIFO, primero que llega, primero en ser atendido).

En una LinkedList normal tenemos estos tres métodos:

- **boolean add(E e)** y **boolean offer(E e)**, retornarán true si se ha podido insertar el elemento al final de la LinkedList.
- **E poll()** retornará el primer elemento de la LinkedList y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará null si la lista está vacía.
- **E peek()** retornará el primer elemento de la LinkedList pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía.

Las pilas son todo lo contrario a las listas. Mientras que en una lista se añade y se elimina al final, en una pila se añade y se elimina al principio. Para ello se utilizan tres métodos:

- push: meter al principio de la pila.
- pop: sacar y eliminar al principio de la pila.
- peek: igual que si se usara la lista pero en una cola.

## 4.3. Listas (IV)

Cuando usamos colecciones, no es lo mismo usarlos con objetos inmutables como Strings, Integer, etc... que con objetos mutables.

Los objetos mutables no se pueden modificar después de su creación, cuando se incorporan a la lista a través del método add se pasan por copia. En cambio los objetos mutables (objetos propios) no se copian y puede producir efectos no deseados.

**Ejemplo:**

```
class Test
{
    public Integer num;
    Test (int num) {
        this.num=new Integer(num);
    }
}
```

Esta clase es mutable, no se pasa copia a la lista.

Si en el siguiente código se crea una lista que utilice este tipo de objetos:

```
Test p1=new Test(11); // Se crea un objeto Test donde el entero que contiene vale 11.

Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene vale 12.

LinkedList<Test> lista=new LinkedList<Test>(); // Creamos una lista enlazada para objetos tipo Test.

lista.add(p1); // Añadimos el primero objeto test.

lista.add(p2); // Añadimos el segundo objeto test.

for (Test p:lista) System.out.println(p.num); // Mostramos la lista de objetos
```

En el código anterior se mostrarían los números 10 y 11.

Si modificamos el valor de uno de los números de los objetos test:

```
p1.num=44;  
for (Test p:lista) System.out.println(p.num);
```

El resultado de imprimir la lista será 44 y 12.

El número ha sido modificado sin tener que volver a insertarlo en la lista, ya que en la lista está almacenado el apuntador al objeto original. Solo existe un objeto al que se hace referencia desde distintos lugares.

## 5. Conjuntos de pares clave/valor

Los conjuntos de pares son arrays asociativos, los cuales permiten almacenar pares de valores conocidos como clave/valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En java existe `java.util.Map`, que define los métodos que deben tener los mapas. Existen tres implementaciones principales: `java.util.HashMap`, `java.util.TreeMap` y `java.util.LinkedHashMap`. Cada una de las tiene características similares a `HashSet`, `TreeSet` y `LinkedHashSet`, tanto en funcionamiento como en rendimiento.

Los mapas utilizan clases genéricas para dar extensibilidad y flexibilidad y permiten definir un tipo base para la clave y otro para el valor.

```
HashMap<String, Integer> t=new HashMap<String, Integer>();
```

### Métodos principales de los mapas:

- **V put(K key, V value);** Inserta un par de objetos llave (key) y valor (value) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará null.
- **V get(Object key);** Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará null.
- **V remove(Object key);** Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe.
- **boolean containsKey(Object key);** Retornará true si el mapa tiene almacenada la llave pasada por parámetro, false en cualquier otro caso.
- **boolean containsValue(Object value);** Retornará true si el mapa tiene almacenado el valor pasado por parámetro, false en cualquier otro caso.
- **int size();** Retornará el número de pares llave y valor almacenado en el mapa.
- **boolean isEmpty();** Retornará true si el mapa está vacío, false en cualquier otro caso.

## 6. Iteradores (I)

Los iteradores nos permiten recorrer las colecciones de dos formas: bucles for-each y a través de un bucle normal creando un iterador.

Para crear un iterador se invoca el método iterator() de cualquier colección. Por ejemplo:

```
Iterator<Integer> it=t.iterator();
```

Se especifica un parámetro para el tipo de dato genérico en el iterador poniendo <Integer> después de Iterator, ya que también son clases genéricas. Si no se especifica el tipo base también se permite recorrer la colección pero retornará objetos tipo Object y nos vemos forzados a conversión de tipo.

Para recorrer y gestionar la colección el iterador ofrece tres métodos:

- **boolean hasNext();** Devuelve true si quedan más elementos a la colección por visitar.
- **E next().** Devuelve el siguiente elemento de la colección, si no existe lanza una excepción (NoSuchElementException). Conviene chequear primero si existe el elemento.
- **remove().** Elimina de la colección el último elemento retornado de la invocación next. Si next no ha sido invocado lanza una excepción.

Por tanto para recorrer una colección con estos métodos se utiliza un while:

```
while (it.hasNext()) // Mientras que haya un siguiente elemento, seguiremos en el bucle.  
{  
    Integer t=it.next(); // Escogemos el siguiente elemento.  
    if (t%2==0) it.remove(); //Si es necesario, podemos eliminar el elemento extraído de la lista.  
}
```

### 6.1. Iteradores (II)

Los inconvenientes de usar los iteradores sin especificar el tipo de objeto es la obligatoriedad de tener que usar la operación de conversión de tipos. Usando genéricos se aportan grandes ventajas.

Ejemplo indicando el tipo de objeto del iterador:

```
ArrayList <Integer> lista=new ArrayList<Integer>();  
  
for (int i=0;i<10;i++) lista.add(i);  
  
Iterator<Integer> it=lista.iterator();  
  
while (it.hasNext()) {  
    Integer t=it.next();  
    if (t%2==0) it.remove();  
}
```

Ejemplo sin indicar el tipo de objeto del iterador:

```
ArrayList <Integer> lista=new ArrayList<Integer>();  
  
for (int i=0;i<10;i++) lista.add(i);  
  
Iterator it=lista.iterator();  
  
while (it.hasNext()) {  
    Integer t=(Integer)it.next();  
    if (t%2==0) it.remove();  
}
```

Para recorrer mapas con iteradores se utiliza el método entrySet que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien el método keySet para generar un conjunto con las llaves existentes en el mapa:

```
HashMap<Integer, Integer> mapa=new HashMap<Integer, Integer>();  
  
for (int i=0;i<10;i++) mapa.put(i, i); // Insertamos datos de prueba en el mapa.  
  
for (Integer llave:mapa.keySet()) // Recorremos el conjunto generado por keySet, contendrá las llaves  
{  
    Integer valor=mapa.get(llave); //Para cada llave, accedemos a su valor si es necesario.  
}
```

El conjunto generado por KeySet no tiene el método add para añadir elementos al mismo, eso se tiene que hacer a través del mapa.

## 7. Algoritmos

Los algoritmos basados en colecciones nos sirven para:

- Ordenar listas y arrays
- Desordenar listas y array
- Búsqueda binaria en listas y arrays
- Conversión de arrays a listas y de listas a arrays
- Partir cadenas y almacenar el resultado en un array.

La mayoría de estos algoritmos están recogidos como métodos estáticos en java.util.Collections y java.util.Arrays, salvo los referentes a cadenas.

Los algoritmos de ordenación lo hacen ordenando de manera natural siempre que java sepa como hacerlo. Si no, hay que facilitar el mecanismo para producir la ordenación.

Los tipos ordenables son enteros, cadenas y fechas en orden ascendente.

La clase Collections y la clase Arrays facilitan el método sort que permite ordenar listas y arrays. Los siguientes ejemplos ordenan números de forma ascendente.

### Ejemplo de ordenación de array de números:

```
Integer[] array={10, 9, 99, 3, 5};  
Arrays.sort(array);
```

### Ejemplo de ordenación de una lista con números

```
ArrayList<Integer> lista=new ArrayList<Integer>();  
  
lista.add(10);  
lista.add(9);  
lista.add(99);  
lista.add(3);  
lista.add(5);  
  
Collections.sort(lista)
```

## 7.1. Algoritmos (II)

En java hay dos maneras para cambiar la forma en que se ordenan los elementos.

Imagina que hay artículos almacenados en una lista llamada "articulos" y que cada artículo se almacena en la siguiente clase, teniendo en cuenta que el código es un String.

```
class Articulo {  
  
    public String codArticulo; // Código de artículo  
    public String descripcion; // Descripción del artículo.  
    public int cantidad; // Cantidad a proveer del artículo.  
}
```

La primera forma es crear una clase que implemente la interfaz `java.util.Comparator` e implementar el método `compare` definido en esa interfaz.

```
class comparadorArticulos implements Comparator<Articulo> {  
    @Override  
    public int compare( Articulo o1, Articulo o2) {  
        return o1.codArticulo.compareTo(o2.codArticulo);  
    }  
}
```

Una vez creada la clase, simplemente se pasa como segundo parámetro una instancia del comparador creado.

```
Collections.sort(articulos, new comparadorArticulos());
```

La segunda forma consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz `java.util.Comparable`. Todos los objetos que implementan esta interfaz son ordenables y se puede invocar el método `sort` sin indicar un comparador para ordenarlos. La interfaz `comparable` solo requiere implementar el método `compareTo`.

```
class Articulo implements Comparable<Articulo>{
    public String codArticulo;
    public String descripcion;
    public int cantidad;
    @Override

    public int compareTo(Articulo o) {
        return codArticulo.compareTo(o.codArticulo);
    }
}
```

La interfaz `Comparable` es genérica y para que funcione es conveniente indicar el tipo de base sobre el que se permite la comparación. El método `compareTo` solo admite un parámetro y es el que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método `compareTo` es el mismo que el método `compare` de `Comparator` (-1, 0, 1).

```
Collections.sort(articulos);
```

## 7.2. Algoritmos (III)

`java.util.Collections` y `java.util.Arrays` ofrecen, además los siguientes métodos.

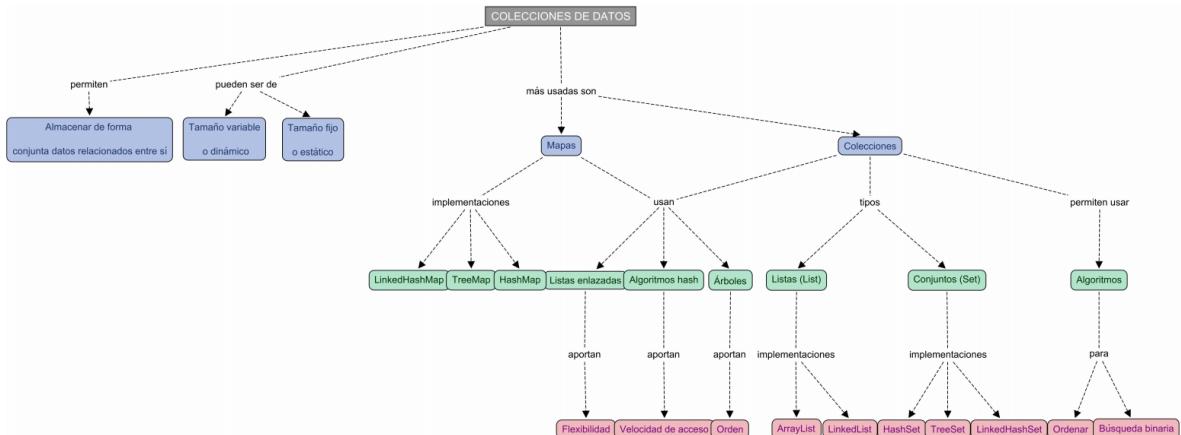
- **Desordenar una lista. `Collections.shuffle(lista)`**; Desordena una lista, este método no está disponible para arrays.
- **Rellenar una lista o array. `Collections.fill(lista, elemento)`; `Array.fill(array, elemento)`**; Rellena una lista o array copiando el mismo valor en todos los elementos del array ,o lista. Útil para reiniciar una lista o array.
- **Búsqueda binaria: `Collections.binarySearch(lista, elemento)`; `Arrays.binarySearch(array, elemento)`**; Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito.
- **Convertir un array a lista `List lista = Arrays.asList(array)`**; Si el tipo de dato almacenado en el array es conocido es conveniente especificarlo: `List<Integer>lista = Arrays.asList(array)`; Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es `ArrayList` ni `LinkedList`), solo se especifica que retorna una lista que implementa la interfaz `java.util.List`.

- **Convertir una lista a array Integer[] array = new Integer[list.size()];**  
**lista.toArray(array);** Permite convertir una lista a array. Esto se puede realizar en todas las colecciones, y no es un método de la clase Collections, sino propio de la interfaz Collection. Es conveniente que sepas de su existencia.
- **Dar la vuelta. Collections.reverse(lista);** Da la vuelta a una lista, poniéndola en orden inverso al que tiene.

Otra operación es la de dividir texto por partes. Cuando una cadena está formada por trozos de texto delimitados por un separador que se repite es posible obtener cada uno de los trozos de texto por separado en un array de cadenas. Para poder realizar esta operación se utiliza split de la clase String.

```
String texto="Z,B,A,X,M,O,P,U";
String []partes=texto.split(",");
Arrays.sort(partes);
```

## Mapa conceptual





# 9. Almacenando datos

	Autor	(X) Xerach Casanova
	Clase	Programación
	Fecha	@Mar 29, 2021 3:26 PM

## 1. Introducción

### 1.1. Excepciones

## 2. Concepto de flujo

### 3. Clases relativas a flujos

#### 3.1. Ejemplo comentado de una clase con flujos

## 4. Flujos

### 4.1. Flujos predefinidos. Entrada y salida estándar.

### 4.2. Flujos predefinidos. Entrada y salida estándar. Ejemplo.

### 4.3. Flujos basados en bytes

### 4.4. Flujos basados en caracteres

### 4.5. Rutas de los ficheros

## 5. Trabajando con ficheros

### 5.1. Escritura y lectura de información de ficheros.

### 5.2. Ficheros binarios y ficheros de texto (I)

#### 5.2.1. Ficheros binarios y ficheros de texto (II)

### 5.3. Modos de acceso. Registros

### 5.4. Acceso secuencial

### 5.5. Acceso aleatorio

## 6. Aplicaciones del almacenamiento de información en ficheros

## 7. Utilización de los sistemas de ficheros

### 7.1. Clase File

### 7.2. Interface FilenameFilter

### 7.3. Creación y eliminación de ficheros y directorios

## 8. Almacenamiento de objetos en ficheros. Persistencia. Serialización

### 8.1. Serialización: utilidad

## 9. Mapa conceptual

# 1. Introducción

El almacenamiento en variables o vectores es temporal, los datos se pierden en las variables cuando están fuera de su ámbito o si el programa termina. Los

ordenadores utilizan ficheros para guardar los datos. Solemos llamar a los datos que se guardan en ficheros "datos persistentes".

A las operaciones que constituyen un flujo de información del programa con el exterior se les conoce como Entrada/salida (E/S) y se distinguen dos tipos: E/S estándar que se realiza con el terminal del usuario y E/S a través de ficheros, que trabaja con ficheros de disco.

Todas estas operaciones vienen proporcionadas por la API de java en el paquete `java.io` e incorpora interfaces, clases y excepciones.

El contenido de un archivo puede interpretarse como campos y registros (grupos de campos), dándole un significado al conjunto de bits que posee en realidad.

## 1.1. Excepciones

Trabajar con archivos puede generar errores y para manejarlos debemos usar excepciones. Las dos más comunes son:

- **FileNotFoundException** cuando no encuentra el archivo.
- **IOException** si no se tienen permisos de lectura o escritura, o si el archivo está dañado.

El esquema básico de uso de la captura y tratamiento de excepciones de un programa podría ser este

```
public static void main(String args[]){
    try{
        //Se ejecuta algo que permite producir una excepción.

    } catch (FileNotFoundException e) {
        //manejo de una excepción cuando no encuentra un archivo.

    } catch (IOException e) {
        //manejo de una excepción de entrada/salida.

    } catch (Exception e) {
        //manejo de una excepción cualquiera
    }
}
```

## 2. Concepto de flujo

La clase Stream representa un flujo de corriente de datos o un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de E/S, memoria, conector TCP/IP, etc...

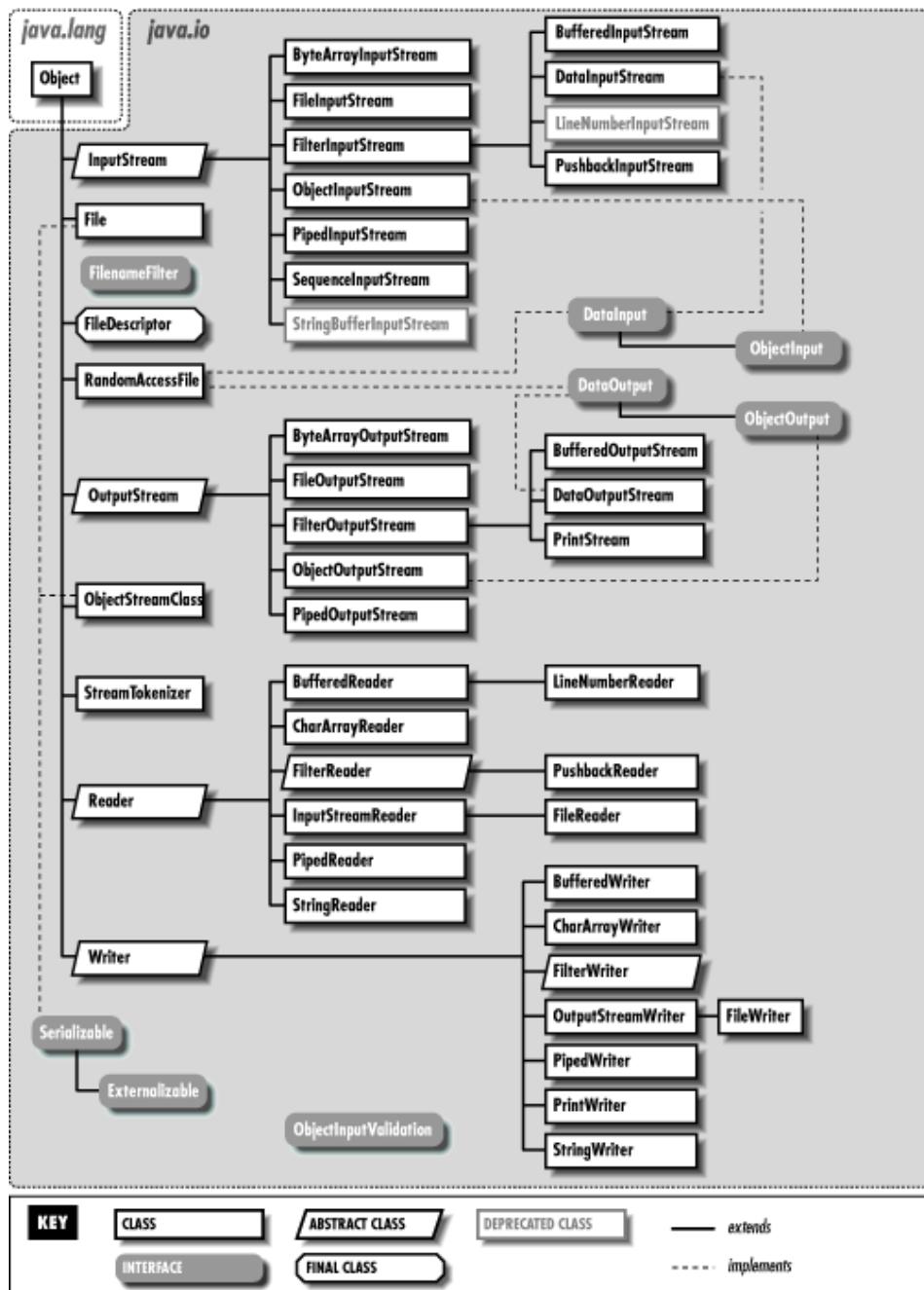
Cualquier programa realizado en Java que necesita llevar operaciones E/S lo hace a través de stream. Un flujo o stream es una abstracción de aquello que produzca o consuma información.

Las clases y métodos de E/S que necesitamos son las mismas en todos los dispositivos que usemos, el núcleo de Java es el encargado de determinar si trata con el teclado, el monitor, un sistema de archivos o un socket de red, de esta manera no tenemos que cambiar nada en nuestra aplicación, ya que ésta será independiente de los dispositivos físicos de almacenamiento y del Sistema Operativo sobre el que se ejecuta.

### 3. Clases relativas a flujos

Existen dos tipos de flujos:

- **Flujos de caracteres (16 bits)** se usan para manipular datos legibles por humanos, como por ejemplo un fichero de texto. Usan dos clases abstractas: Reader y Writer, las cuales manejan flujos de caracteres Unicode. De ellas derivan subclases que implementan métodos definidos destacando read() y write().
- **Flujos de bytes (8 bits)**, se usan para manipular datos binarios legibles por la máquina, como el fichero de un programa. El tratamiento del flujo de bytes viene dado por las dos clases abstractas InputStream y OutputStream, las cuales definen los métodos que implementan sus subclases entre las que destacan read() y write().



Entre las clases del paquete `java.io` destacamos:

- **BufferedInputStream**: permite leer datos a través de un flujo con un buffer intermedio.
- **BufferedOutputStream**: implementa métodos para escribir en un flujo a través de un buffer.
- **FileInputStream**: permite leer bytes de un fichero.
- **FileOutputStream**: permite escribir bytes en un fichero o descriptor.

- **StreamTokenizer**: recibe un flujo de entrada, lo analiza (parse) y divide en pedazos (tokens), permitiendo leer uno en cada momento.
- **StringReader**: es un flujo de caracteres cuya fuente es un String.
- **StringWriter**: es un flujo de caracteres cuya salida es un buffer de cadena de caracteres que se puede utilizar para construir un String.

Hay clases que se montan sobre otros flujos para modificar la forma de trabajar con ellos, por ejemplo, con `BufferedInputStream` podemos añadir un buffer a un flujo `FileInputStream`, mejorando la eficiencia de acceso a dispositivos donde se almacena el fichero.

### 3.1. Ejemplo comentado de una clase con flujos

La clase `StreamTokenizer` obtiene un flujo de entrada y lo divide en tokens. El flujo `tokenizer` puede reconocer identificadores, números y otras cadenas.

El ejemplo q siguiente , muestra cómo utilizar la clase `StreamTokenizer` para contar números y palabras de un fichero de texto. Se abre el flujo con ayuda de la clase `FileReader`, y puedes ver cómo se "monta" el flujo `StreamTokenizer` sobre el `FileReader`, es decir, que se construye el objeto `StreamTokenizer` con el flujo `FileReader` como argumento, y entonces se empieza a iterar sobre él.

En este código, se aprecia que se iterará hasta llegar al fin del archivo, para cada token, se mira su tipo y según su tipo se incrementa el contador de palabras o números.

```
package token;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;

public class Token {

    public static void contarPalabryNumeros(String nombre_fichero) {

        StreamTokenizer sTokenizer = null;
        int contapal = 0, numNumeros = 0;

        try {

            sTokenizer = new StreamTokenizer(new FileReader(nombre_fichero));

            while (sTokenizer.nextToken() != StreamTokenizer.TT_EOF) {

                if (sTokenizer.ttype == StreamTokenizer.TT_WORD)
                    contapal++;


```

```

        else if (sTokenizer.ttype == StreamTokenizer.TT_NUMBER)
            numNumeros++;
    }

    System.out.println("Número de palabras en el fichero: " + contapal);
    System.out.println("Número de números en el fichero: " + numNumeros);

} catch (FileNotFoundException ex) {
    System.out.println(ex.getMessage());
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    contarPalabryNumeros("c:\\datos.txt");
}
}

```

El método `nextToken` devuelve un `int` que indica el tipo de token leído. Hay una serie de constantes que determinan el tipo de token.

- **TT\_WORD** indica que el token es una palabra.
- **TT-NUMBER** indica que el token es un número.
- **TT-EOL** indica que se ha leído el fin de línea.
- **TT-EOF** indica que se ha llegado al fin del flujo de entrada.

## 4. Flujos

### 4.1. Flujos predefinidos. Entrada y salida estándar.

Tradicionalmente Unix, Linux y MS-DOS han utilizado un tipo de E/S estándar. El fichero de entrada estándar (`stdin`) es típicamente el teclado y el de salida (`stdout`) es típicamente la pantalla o la ventana del terminal. El fichero de salida de error estándar (`stderr`) también se dirige normalmente a la pantalla, pero se implementa como otro fichero para distinguirlo de la salida normal.

Java tiene acceso a ellos a través de la clase `System`. Los tres ficheros que se implementan son:

- **Stdin.** Es un objeto de tipo `InputStream` y está definido en la clase `System` como flujo de entrada estándar. Por defecto es el teclado, pero se puede redirigir para cada host o usuario.

- **Stdout. System.out.** Implementa stdout como una instancia de la clase PrintStream. Se pueden utilizar print() y println() con cualquier tipo básico Java como argumento.
- **Stderr.** Es un objeto de tipo PrintStream. Es un flujo de salida definido por la clase System y representa la salida de error estándar, pero es posible redirigirlo a otro dispositivo de salida.

Para la entrada, se usa el método read para leer de la entrada estándar:

- **int System.in.read();**
  - Lee el siguiente byte (char) de la entrada estándar.
- **int System.in.read(byte[] b);**
  - Lee un conjunto de bytes de la entrada estándar y lo almacena en un vector.

Para leer valores numéricos se toma el valor de la entrada estándar en forma de cadena y usamos métodos para transformar el texto a números.

- **byte Byte.parseByte(String):** Convierte una cadena en un número entero de un byte.
- **short Short.parseShort(String):** Convierte una cadena en un número entero corto.
- **int Integer.parseInt(String):** Convierte una cadena en un número entero.
- **long Long.parseLong(String):** Convierte una cadena en un número entero largo.
- **float Float.parseFloat(String):** Convierte una cadena en un número real simple.
- **double Double.parseDouble(String):** Convierte una cadena en un número real doble.
- **boolean Boolean.parseBoolean(String):** Convierte una cadena en un valor lógico.

## 4.2. Flujos predefinidos. Entrada y salida estándar. Ejemplo.

En este ejemplo, se lee por teclado hasta pulsar la tecla retorno, en ese momento el programa acaba imprimiendo por la salida estándar la cadena leída.

Para construir la cadena con caracteres leídos usamos la clase StringBuffer, que permite almacenar cadenas que cambiarán la ejecución del programa, o StringBuilder, similar pero no es síncrona. Para la mayoría de aplicaciones donde

solo se ejecuta un hilo, supone una mejora de rendimiento sobre StringBuffer. Todo el proceso debe estar en un bloque try catch

```
import java.io.IOException;

public class leeEstandar {
    public static void main(String[] args) {
        // Cadena donde iremos almacenando los caracteres que se escriban
        StringBuilder str = new StringBuilder();
        char c;
        // Por si ocurre una excepción ponemos el bloque try-catch
        try{
            // Mientras la entrada de teclado no sea Intro
            while ((c=(char)System.in.read())!='\n'){
                // Añadir el carácter leído a la cadena str
                str.append(c);
            }
        }catch(IOException ex){
            System.out.println(ex.getMessage());
        }

        // Escribir la cadena que se ha ido tecleando
        System.out.println("Cadena introducida: " + str);
    }
}
```

## 4.3. Flujos basados en bytes

Este tipo de flujos es idóneo para el manejo de entradas y salidas de bytes, su uso está orientado a lectura y escritura de datos binarios.

Para el tratamiento de flujos de bytes se usan las clases abstractas InputStream y OutputStream. Cada una de las cuales tiene subclases que controlan las diferencias entre distintos dispositivos E/S que se pueden usar.

```
class FileInputStream extends InputStream {

    FileInputStream (String fichero) throws FileNotFoundException;
    FileInputStream (File fichero) throws FileNotFoundException;
    ...
}

class FileOutputStream extends OutputStream {

    FileOutputStream (String fichero) throws FileNotFoundException;
    FileOutputStream (File fichero) throws FileNotFoundException;
    ...
}
```

OutputStream e InputStream y todas sus subclases reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Por ejemplo, podemos copiar el contenido de un fichero en otro:

```
void copia (String origen, String destino) throws IOException {
    try{
        // Obtener los nombres de los ficheros de origen y destino
        // y abrir la conexión a los ficheros.
        InputStream fentrada = new FileInputStream(origen);
        OutputStream fsalida = new FileOutputStream(destino);
        // Crear una variable para leer el flujo de bytes del origen
        byte[] buffer= new byte[256];
        while (true) {
            // Leer el flujo de bytes
            int n = fentrada.read(buffer);
            // Si no queda nada por leer, salir del bucle
            if (n < 0)
                break;
            // Escribir el flujo de bytes leidos al fichero destino
            fsalida.write(buffer, 0, n);
        }
        // Cerrar los ficheros
        fentrada.close();
        fsalida.close();
    }catch(IOException ex){
        System.out.println(ex.getMessage());
    }
}
```

## 4.4. Flujos basados en caracteres

Las clases orientadas al flujo de bytes ayudan a realizar cualquier tipo de operación de entrada y salida, pero no trabajan directamente con caracteres Unicode, los cuales están representados por dos bytes y por eso usamos las clases orientadas al flujo de caracteres, para lo cual Java tiene dos clases abstractas: Reader y Writer.

Reader, Writer y todas sus subclases reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de E/S. El flujo de datos se debe cerrar cuando no se necesita.

Existen muchos tipos de flujos dependiendo de la utilidad de los datos que extraemos de los dispositivos.

Un flujo se puede envolver por otro para tratarlo de manera más cómodo. Así, un BufferedWriter permite manipular flujos de datos como un buffer, pero si lo envolvemos en un PrintWriter podemos escribir con más funcionalidades para este tipo de datos.

```

try{
    PrintWriter out = null;
    out = new PrintWriter(new FileWriter("c:\\\\salida.txt", true));

    BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    while (!(s = br.readLine()).equals("salir")){
        out.println(s);
    }
    out.close();
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}

```

## 4.5. Rutas de los ficheros

Cuando se opera con rutas de ficheros, el carácter separador entre directorios cambia dependiendo del S.O.

Para evitar problemas, se recomienda usar File.separator y se puede hacer una función que al pasarle la ruta nos devuelva la adecuada según el S.O. actual.

```

static String substFileSeparator(String ruta) {
String separador = "\\\\";

try{
    // Si estamos en Windows
    if ( File.separator.equals(separador) )
        separador = "/" ;
    // Reemplaza todas las cadenas que coinciden con la expresión
    // regular dada oldSep por la cadena File.separator
    return ruta.replaceAll(separador, File.separator);
} catch(Exception e){
    // Por si ocurre una java.util.regex.PatternSyntaxException
    return ruta.replaceAll(separador + separador, File.separator);
}
}

```

## 5. Trabajando con ficheros

Se debe tener en cuenta que la manera de proceder con los ficheros es:

- Abrir o crear el fichero.
- Hacer operaciones.

- Cerrar el fichero.

Se debe tener control de las excepciones para evitar fallos en tiempo de ejecución.

## 5.1. Escritura y lectura de información de ficheros.

Debemos tener en cuenta las clases que vamos a emplear y determinar si es más eficiente emplear clases de Reader Writer o las de InputStream y OutputStream.

También debemos considerar el acceso al fichero. Ya que podemos acceder de manera secuencia y recorrer el fichero entero hasta llegar a la posición, o de manera aleatoria para posicionarnos en una posición del fichero.

La idea del uso de las clases que usan buffer es que cuando una aplicación necesita leer datos de un fichero, debe esperar a que el disco en el que está el fichero le proporcione la información, consiguiendo reducir el número de accesos al fichero y mejorando la eficiencia de la aplicación, ya que un dispositivo de memoria masiva es más lento que la CPU del ordenador.

Para ello asociamos al fichero una memoria intermedia o buffer y cuando se necesita leer el byte del archivo, se trae hasta el buffer asociado al flujo.

Cualquier operación E/S puede generar una IOException y deben estar dentro de un bloque try/catch.

## 5.2. Ficheros binarios y ficheros de texto (I)

Los ficheros se utilizan para guardar información en soporte y distinguimos entre ficheros de texto y binarios. Los ficheros de texto están codificados en Unicode o en ASCII.

Podemos averiguar la codificación de un fichero con el método getEncoding()

```
FileInputStream fichero;
try {
    // Elegimos fichero para leer flujos de bytes "crudos"
    fichero = new FileInputStream("c:\\\\texto.txt");
    // InputStreamReader sirve de puente de flujos de byte a caracteres
    InputStreamReader unReader = new InputStreamReader(fichero);
    // Vemos la codificación actual
    System.out.println(unReader.getEncoding());
} catch (FileNotFoundException ex) {
}
```

Los archivos de texto se pueden abrir usando la clase FileReader, que nos proporciona métodos para leer caracteres y, cuando nos interese leer líneas

completas podemos usar la clase BufferedReader a partir de FileReader.

```
File arch = new File ("C:\\fich.txt");
FileReader fr = new FileReader (arch);
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

Para escribir en archivos de texto lo hacemos teniendo en cuenta

```
FileWriter fich = null;
PrintWriter pw = null;
fich = new FileWriter("/fich2.txt");pw = new PrintWriter(fichero);
pw.println("Linea de texto");
```

Si el fichero que queremos escribir existe y solo queremos añadir información pasamo el segundo parámetro como true.

```
FileWriter("/fich2.txt",true)
```

### 5.2.1. Ficheros binarios y ficheros de texto (II)

Los ficheros binarios almacenan la información en bytes codificada en binario, pueden contener fotografías, números, letras, archivos ejecutables, etc.

Los archivos binarios guardan una representación de los datos en el fichero en código UTF-8.

Para leer un fichero binario se usa la clase FileInputStream, que trabaja con bytes y se leen desde el flujo asociado a un fichero.

Ejemplo de lectura de fichero de bytes:

```
package leerconbuffer;

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

/**
 *
 * @author
 */
public class LeerConBuffer {

    public static void main(String[] args) {
        int tama ;
```

```

try{
    // Creamos un nuevo objeto File, que es la ruta hasta el fichero desde
    File f = new File("C:\\apuntes\\test.bin");

    // Construimos un flujo de tipo FileInputStream (un flujo de entrada desde
    // fichero) sobre el objeto File. Estamos conectando nuestra aplicaciÃ³n
    // a un extremo del flujo, por donde van a salir los datos, y "pidiendo"
    // al Sistema Operativo que conecte el otro extremo al fichero que indica
    // la ruta establecida por el objeto File f que habÃ¡amos creado antes. De
    FileInputStream flujoEntrada = new FileInputStream(f);
    BufferedInputStream fEntradaConBuffer = new BufferedInputStream(flujoEntrada);

    // Escribimos el tamaÃ±o del fichero en bytes.
    tama = fEntradaConBuffer.available();
    System.out.println("Bytes disponibles: " + tama);

    // Indicamos que vamos a intentar leer 50 bytes del fichero.
    System.out.println("Leyendo 50 bytes....");

    // Creamos un array de 50 bytes para llenarlo con los 50 bytes
    // que leamos del flujo (realmente del fichero)*/
    byte bytarray[] = new byte[50];

    // El mÃ©todo read() de la clase FileInputStream recibe como parÃ¡metro un
    // array de byte, y lo llena leyendo bytes desde el flujo.
    // Devuelve un nÃºmero entero, que es el nÃºmero de bytes que realmente se
    // han leÃ±do desde el flujo. Si el fichero tiene menos de 50 bytes, no
    // podrÃ¡ leer los 50 bytes, y escribirÃ¡ un mensaje indicÃ¡ndolo.
    if (fEntradaConBuffer.read(bytarray) != 50)
        System.out.println("No se pudieron leer 50 bytes");

    // Usamos un constructor adecuado de la clase String, que crea un nuevo
    // String a partir de los bytes leÃ±ados desde el flujo, que se almacenaron
    // en el array bytarray, y escribimos ese String.
    System.out.println(new String(bytarray, 0, 50));

    // Finalmente cerramos el flujo. Es importante cerrar los flujos
    // para liberar ese recurso. Al cerrar el flujo, se comprueba que no
    // haya quedado ningÃºn dato en el flujo sin que se haya leÃ±ado por la aplicaciÃ³n. */
    fEntradaConBuffer.close();

    // Capturamos la excepciÃ³n de Entrada/Salida. El error que puede
    // producirse en este caso es que el fichero no estÃ© accesible, y
    // es el mensaje que enviamos en tal caso.
}catch (IOException e){
    System.err.println("No se encuentra el fichero");
}
}
}
}

```

Para escribir datos a un fichero binario la clase FileOutputStream nos permite usar el fichero para escritura de bytes en él, La filosofía es la misma pero en dirección contraria, desde la aplicación que hace de fuente de datos, hasta el fichero que los consumen.

En la siguiente presentación puedes ver un esquema de cómo utilizar buffer para optimizar la lectura de teclado desde consola, por medio de las envolturas, podemos usar métodos como `readline()`, de la clase `BufferedReader`, que envuelve a un objeto de la clase `InputStreamReader`.



### 5.3. Modos de acceso. Registros

En java no se impone una estructura en un fichero y no existe el concepto de registro en los ficheros creados con Java.

Los programadores deben definir su registro con el número de bytes que les interesen, moviéndose por el fichero teniendo en cuenta ese tamaño definido.

Un fichero es de acceso directo u organización directa cuando para acceder a un registro n cualquiera, no se tiene que pasar por los n-1 registros anteriores.

Podemos trabajar con ficheros secuenciales y con ficheros de acceso aleatorio. En los secuenciales, para recuperar la información se hace en el mismo orden en que se introdujo, por lo que para acceder a un registro determinado se deben leer los anteriores.

Si se trata de un fichero de acceso aleatorio se puede acceder directamente.

### 5.4. Acceso secuencial

En el siguiente ejemplo vemos cómo se escriben datos en un fichero secuencial: el nombre y apellidos de una persona utilizando el método `writeUTF()` que proporciona `DataOutputStream`, seguido de su edad que la escribimos con el método `writeln()` de la misma clase. A continuación escribimos lo mismo para una segunda persona y de nuevo para una tercera. Después cerramos el fichero. Y ahora lo abrimos de nuevo para ir

leyendo de manera secuencial los datos almacenados en el fichero, y escribiéndolos a consola.

```
package escylee;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 *
 * @author JJBH
 */
public class EscyLee {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declarar un objeto de tipo archivo
        DataOutputStream archivo = null ;
        DataInputStream fich = null ;
        String nombre = null ;
        int edad = 0 ;
        try {
            // Creando o abriendo para añadir el archivo
            archivo = new DataOutputStream( new FileOutputStream("c:\\\\secuencial.dat",true) );

            // Escribir el nombre y los apellidos
            archivo.writeUTF( "Antonio LÃ³pez PÃ©rez      " );
            archivo.writeInt(33) ;
            archivo.writeUTF( "Pedro Piqueras PeÃ±aranda" );
            archivo.writeInt(45) ;
            archivo.writeUTF( "JosÃ© Antonio Ruiz PÃ©rez " ) ;
            archivo.writeInt(51) ;
            // Cerrar fichero
            archivo.close();

            // Abrir para leer
            fich = new DataInputStream( new FileInputStream("c:\\\\secuencial.dat") );
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            fich.close();

        } catch(FileNotFoundException fnfe) { /* Archivo no encontrado */ }
    }
}
```

```

        catch (IOException ioe) { /* Error al escribir */ }
        catch (Exception e) { /* Error de otro tipo*/
            System.out.println(e.getMessage());}

    }

}

```

Al ver el código escribimos las cadenas de caracteres del mismo tamaño para saber luego el tamaño de registro que hay que leer.

Para buscar información en un fichero secuencial se debe abrir el fichero e ir leyendo registros hasta encontrar el que buscamos, Si se quiere eliminar un registro de un fichero secuencial no se puede quitar el registro y reordenar. Una manera de hacerlo sería crear un nuevo fichero copiando cada uno de los registros excepto el que queremos borrar.

## 5.5. Acceso aleatorio

Podemos acceder al fichero como si fuera una base de datos, donde se accede a un registro concreto del fichero, con la clase `RandomAccessFile`, que permite utilizar un fichero de acceso aleatorio en el que el programador define el formato de los registros.

```
RandomAccessFile objFile = new RandomAccessFile( ruta, modo );
```

`modo` es la dirección física en el sistema de archivos y puede ser `r` para sólo lectura y `rw` para lectura y escritura.

```
RandomAccessFile in = new RandomAccessFile("input.dat", "r");
RandomAccessFile in = new RandomAccessFile("input.dat", "rw");
```

Esta clase implementa las interfaces `DataInput` y `DataOutput`.

Esta clase permite leer y escribir sobre el fichero, no se necesitan dos clases distintas. Hay que especificar el modo de acceso al construir un objeto. Dispone de métodos específicos como `seek` y `skipBytes` para moverse de un registro a otro o posicionarse en una posición concreta.

En este enlace hay más información sobre el acceso aleatorio:

<http://puntocomnoesunlenguaje.blogspot.com/2013/06/java-ficheros-acceso-aleatorio.html>

# 6. Aplicaciones del almacenamiento de información en ficheros

La diversidad de ficheros que existe según la información que se guarda es muy extensa: fotos, música, ficheros de S.O., código fuente de lenguajes de programación, ficheros de películas, etc.

## 7. Utilización de los sistemas de ficheros

### 7.1. Clase File

La clase File proporciona una representación abstracta de ficheros y directorios y permite examinar y manipular, archivos y directorios de cualquier plataforma.

Las instancias de la clase File representan nombres de archivo y no los archivos en sí mismo.

El archivo correspondiente a un nombre dado puede no existir y hay que controlar las posibles excepciones.

Las rutas pueden ser:

- Relativas al directorio actual
- Absolutas si la ruta que pasamos empieza por:
  - Barra en Unix, Linux (/).
  - Letra de unidad en Windows (C:, D:)
  - UNC (Universal naming convention) en Windows como por ejemplo.

```
File miFile=new File("\\\\\\mimaquina\\\\download\\\\prueba.txt");
```

Dado un objeto file, podemos hacer las siguientes operaciones con él:

- Renombrar con el método renameTo().-
- Borrar con el método delete() o deleteOnExit() (se borra cuando finaliza la ejecución de la máquina virtual).
- Crear un fichero con nombre único con el método estático createTempFile, crea un nuevo fichero temporal y devuelve un objeto File que apunta a él.
- Establecer fecha y hora de modificación de archivo con setLastModified(), por ejemplo new File("prueba.txt").setLastModified(new Date().getTime());

- Crear un directorio con el método mkdir() y mkdirs() si los directorios superiores no existen.
- Listar contenido de directorio con list() y listfiles().
- Listar los nombres de archivo de la raíz del sistema de archivos mediante listRoots()

## 7.2. Interface FilenameFilter

La interface FilenameFilter se usa para crear filtros que establezcan criterios de filtrado relativo al nombre de los ficheros y deben definir e implementar el método accept.

```
boolean accept(File dir, String nombre)
```

```
public class Filtro implements FilenameFilter {
    String extension;
    Filtro(String extension){
        this.extension=extension;
    }
    @Override
    public boolean accept(File dir, String name){
        return name.endsWith(extension);
    }

    public static void main(String[] args) {
        try {
            File fichero=new File("c:\\datos\\");
            String[] listadeArchivos;
            listadeArchivos = fichero.list(new Filtro(".odt"));
            int numarchivos = listadeArchivos.length ;

            if (numarchivos < 1)
                System.out.println("No hay archivos que listar");
            else
            {
                for (String listadeArchivo : listadeArchivos) {
                    System.out.println(listadeArchivo);
                }
            }
        catch (Exception ex) {
            System.out.println("Error al buscar en la ruta indicada");
        }
    }
}
```

Launchpad

## 7.3. Creación y eliminación de ficheros y directorios

### **Se puede crear un fichero del siguiente modo:**

- Se crea el objeto que encapsula el fichero

```
File fichero = new File("c:\\\\prueba\\\\miFichero.txt");
```

- Con el objeto File creamos el fichero físicamente, con la siguiente instrucción:

```
fichero.createNewFile()
```

### **Para borrar un fichero:**

podemos usar la clase File, comprobando que existe previamente:

- Fijamos el nombre de la carpeta y del fichero con:

```
File fichero = new File("C:\\\\prueba", "agenda.txt");
```

- Comprobamos si existe con exists() y si es así lo borramos con

```
fichero.delete()
```

### **Para crear directorios:**

```
11 /**
12 * @author JJBH
13 */
14 public class CrearDir {
15
16     public static void main(String[] args) {
17         try {
18             // Declaración de variables
19             String directorio = "C:\\\\micarpeta";
20             String varios = "carpeta1/carpeta2/carpeta3";
21
22             // Crear un directorio
23             boolean exito = (new File(directorio)).mkdir();
24             if (exito) {
25                 System.out.println("Directorio: " + directorio + " creado");
26             }
27             // Crear varios directorios
28             exito = (new File(varios)).mkdirs();
29             if (exito) {
30                 System.out.println("Directorios: " + varios + " creados");
31             }
32         } catch (Exception e) {
33             System.err.println("Error: " + e.getMessage());
34         }
35     }
36
37 }
38
```

### Para borrar un directorio

Borramos cada uno de los ficheros y directorios que contenga, se puede recorrer recursivamente el directorio y borrar los ficheros. podemos listar el directorio con:

```
File[] ficheros = directorio.listFiles();
```

y luego ir borrando. Sabemos si un elemento es directorio con el método isDirectory

## 8. Almacenamiento de objetos en ficheros. Persistencia. Serialización

La serialización es el proceso por el que un objeto se convierte en una secuencia de bytes para más tarde poder reconstruir el valor de sus variables. Permite guardar un objeto en un archivo. Para lo cual.

- El objeto debe implementar la interfaz `java.io.Serializable`, la cual no tiene métodos, solo informa al JVM que es un objeto serializable.
- Todos los objetos incluidos en él tienen que implementar dicha interfaz.

Los tipos primitivos en Java son serializables por defecto, al igual que los array y otros muchos tipos estándar.

Para leer objetos serializados:

```
FileInputStream fich = new FileInputStream("str.out");
ObjectInputStream os = new ObjectInputStream(fich);
Object o = os.readObject();
```

`readObject` lee un flujo de entrada `fitch`, cuando se leen objetos desde un flujo se tiene en cuenta que tipo de objetos se esperan en el flujo y se han de leer en el mismo orden en que se guardaron.

## 8.1. Serialización: utilidad

Su desarrollo viene de la necesidad de convertir los parámetros necesarios a enviar a un objeto en una máquina remota o devolver valores desde ella en forma de flujos de bytes. Enviar primitivos es más fácil pero objetos complejos no.

El método `writeObject` guarda un objeto a través de un flujo de salida. El objeto pasado a `writeObject` implementa la interfaz `Serializable`.

```
FileOutputStream fisal = new FileOutputStream("cadenas.out");
ObjectOutputStream oos = new ObjectOutputStream(fisal);
oos.writeObject();
```

La serialización de objetos se emplea también con JavaBean. Las clases bean se cargan en herramientas de construcción de software visual, como NetBeans.

Con la paleta de diseño se puede personalizar el bean asignando fuentes, tamaños, texto y otras propiedades.

Una vez que se ha personalizado el bean, para guardarla, se emplea la serialización: se almacena el objeto con el valor de sus campos en un fichero con extensión `.ser`, que suele emplazarse dentro de un fichero `.jar`.

## 9. Mapa conceptual

