



DIPLOMADO
**Desarrollo de sistemas con
tecnología Java**

Módulo 1

Programación orientada a objetos con java

Carlos Eligio Ortiz

Lenguaje Java



Origen

- Lenguaje de programación OO de propósito general.
- Nacido en 1994-1995
- Similar a C / C++ (sin funciones de bajo nivel).
- De James Gosling en *Sun*.
- Adquirido por *Oracle*.

Importancia

- Sintaxis más simple que C++
- Multiplataforma.
- Aplicaciones *standalone*, servidores, clientes, teléfonos, autos, etc.
- 9 millones de programadores
- Gratuito.

Especificaciones

Ver. 20: <https://docs.oracle.com/en/java/javase/20/index.html>

Ver 17 (LTS): <https://docs.oracle.com/en/java/javase/17/>

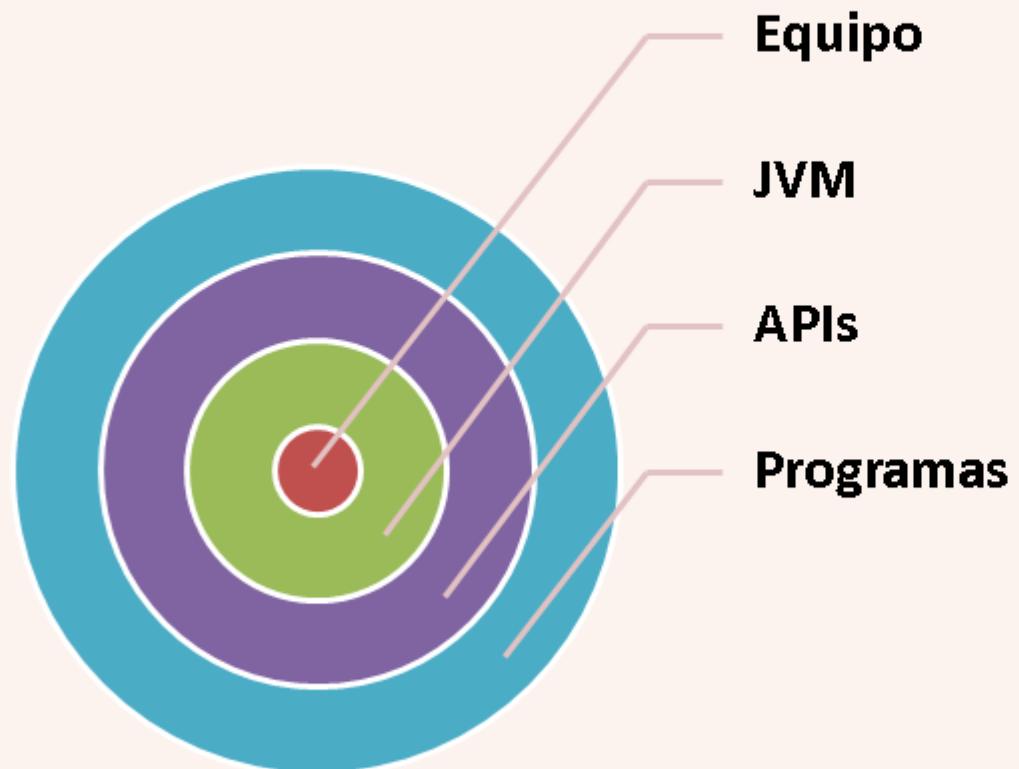
Documentación

Ver. 20: <https://docs.oracle.com/javase/specs/jls/se20/html/index.html>

Ediciones

- **Java SE. Standard edition.** Desarrollo en escritorio, servidores
- **Java EE. Enterprise edition.** Aplicaciones más robustas
- **Java ME/Embedded.** Micro edition/Incrustado. Para aplicaciones en dispositivos de recursos limitados, *IoT*. Dispositivos
- **Java Card.** Aplicaciones en tarjetas inteligentes y dispositivos similares. Énfasis en la seguridad
- **Java TV.** Basado en ME, especializado en TVs y similares.

Java SE



Java Development Kit (JDK)
Para desarrollo

Java Runtime Environment (JRE)
Para ejecución

Características del lenguaje

- De propósito general.
- Orientado a Objetos.
- Compilado e interpretado (portable).
- Seguro (Sandbox, Garbage collector).
- Multitareas

Orientado a objetos

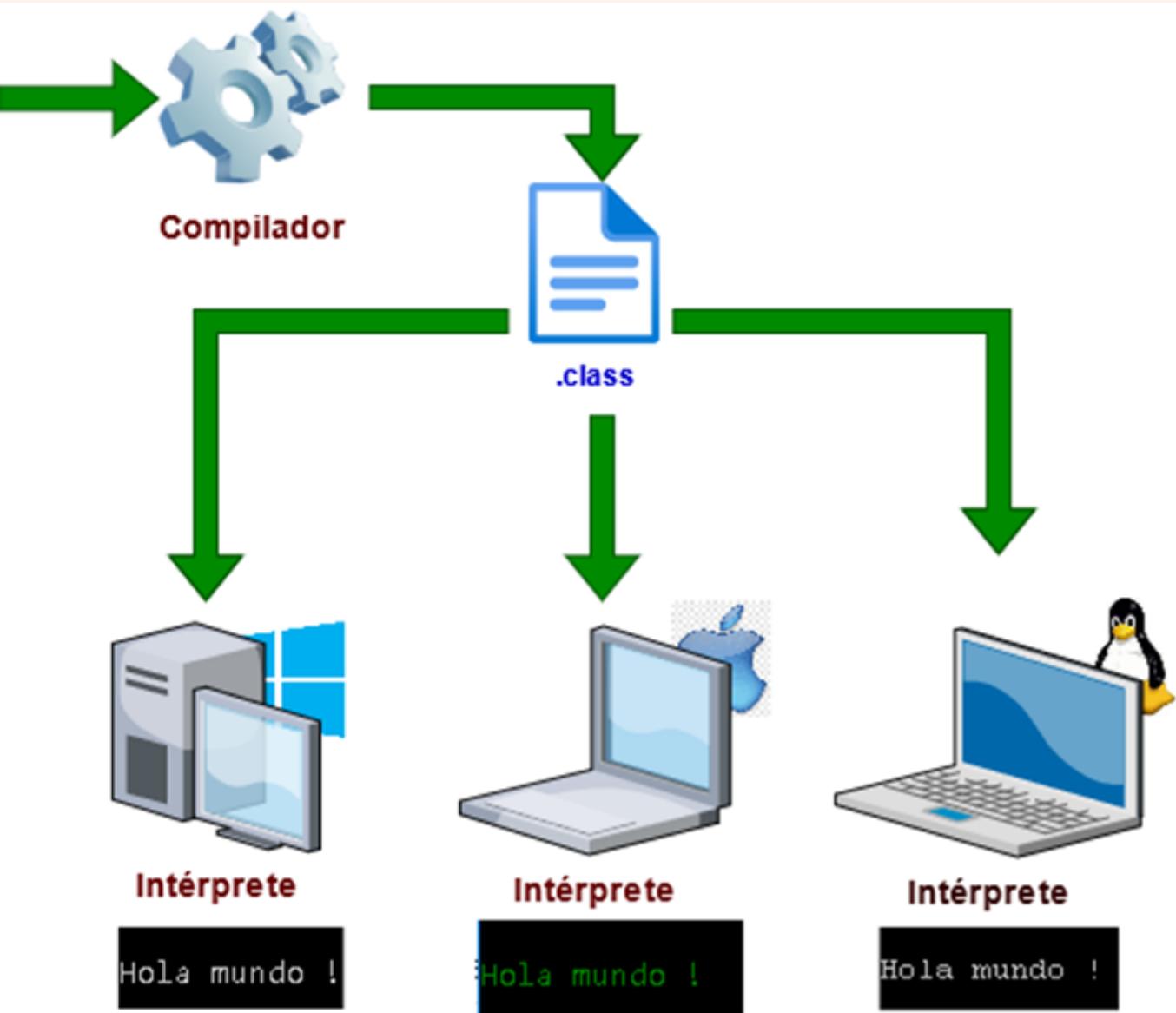
- Modelado e implementación más realista.
- Los objetos tienen propiedades (atributos) y comportamiento (métodos)
- Primero clases, después flujo del programa.
- Posible usar clases preexistentes, crear nuevas clases, extender las existentes

Compilado e Interpretado

```
class HolaMundo {  
    public static void main (String[] args) {  
        System.out.println ("Hola mundo !");  
    } // Fin del main()  
} // Fin de la clase
```

.java

El compilador toma el programa fuente (.java) y genera el *bytecode* (.class), que luego es interpretado por la máquina virtual de la plataforma.



Aplicaciones seguras

- Sandbox.
- Garbage collector.
- No hay apuntadores, no hay manejo de archivos locales en un applet, p. ej.
- En los últimos años se han encontrado vulnerabilidades en aplicaciones java principalmente por falta de actualización.

Versatilidad

- Portable
- Multitareas

Análisis de HolaMundo.java

```
1 //Mi primer programa en Java
2
3 class HolaMundo {
4     public static void main (String[] args) {
5         System.out.println ("Hola mundo !");
6     } //Fin del main()
7 } // Fin de la clase
```

1. Archivo que contiene la clase.
2. Comentario
3. Creación de la clase principal (3 a la 7)
4. Creación de método principal con argumentos de entrada (4 a la 6)
5. Despliegue en pantalla de frase “Hola Mundo!”

Carlos Eligio Ortiz (carloselgio@ortizm.com)

Despliegue de información

- La salida por default es System.out.
- Existen varios métodos para desplegar:
`System.out.println(expresión) ;`
`System.out.print (expresión) ;`
- Despliega la *expresión* ya evaluada.

Clase principal

- Cada archivo Java define una clase, y la clase tendrá el **mismo nombre** que el archivo .java que lo contiene *preferentemente*.
- Para definir una clase se usa la sentencia class y un clasificador de acceso (public en el ejemplo del HolaMundo).
- Dentro de la clase se definen sus métodos y atributos.
- El método main es el que se ejecuta en una aplicación standalone. Es el único método obligatorio (en app, standalone)

Identificadores

- Nombres de clases, variables, atributos, funciones, métodos, constantes, paquetes, etc.
- Deben comenzar con **letra (a-z o A-Z por default, otros caracteres dependiendo del código de caracteres que utilice el .java)**, **_**, o **\$**
- Pueden contener números (**0-9**)
- Sin espacios intermedios.
- No se pueden utilizar palabras reservadas.
- Recomendable usar nombres descriptivos.

Convenciones de nombres

Tipo	Convención	Ejemplos
Clase	Iniciar con mayúscula	String, Empleado, Alumno
Método	Iniciar con minúscula y luego inicial mayúscula cuando comience palabra	getNombre(), setCalificacion(), subirVolumen()
Variable/ Atributo / Paquete	Iniciar con minúscula	apellidoPaterno, calificacion, volumen
Constante	Mayúsculas	PI, IVA, LIMITE_FALTAS

Variables

- Lugar en memoria para guardar algo (número, nombre, domicilio, etc.)
- Debe tener un nombre que lo identifique de manera única en su ámbito.
- **Se definen, se inicializan y se usan.**

Definición de variable

tipo nombre = valor inicial ;

```
int contador = 13;  
double precio=34.6;  
double cantidad=6.5;  
boolean bandera=true;
```

```
int i; //No se inicializó  
int j=5; //se inicializó con 5  
int k =j; // se inicializa k con 5
```

Tipo	Descripción
boolean	Valores lógicos (true/false)
char	Un carácter
int	Número entero
double	Número real
String	Textos (no es un tipo de dato primitivo)



51 palabras reservadas en Java 20

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

false	Literales booleanas
true	
null	Literal nula
var	Identificadores restringidos
yield	
exports	
module	
non-sealed	
open	
opens	
permits	
provides	
record	
requires	
sealed	
to	
transitive	
uses	
with	
yield	Restringido en ciertos contextos

Tipos de datos

- **boolean** (falso/verdadero)
- **char** (2 bytes)
- **byte** (1 byte)
- **short** (2 bytes)
- **int** (4 bytes)
- **long** (8 bytes)
- **float** (4 bytes)
- **double** (8 bytes)

**Siempre tienen
este tamaño,
independiente
de la
plataforma**

Literales numéricas

- Si comienza en dígito (1-9), es base 10 (decimal), si además no tiene punto decimal es int
- Si comienza en 0x es hexadecimal
- Si comienza con 0 es octal
- Si comienza en 0B es binario
- Si termina en L es long
- Si termina en F es float
- Si termina en D es double
- Si tiene punto decimal es double

```
1 class LiteralesNumericas {
2     public static void main (String[] args) {
3         //Enteros
4         byte b = 0x10; //Hexadecimal
5         short s = 010; //Octal
6         int i = 0B10; //Binario
7         long l = 10L; //Long
8         System.out.println ("ENTEROS\nbyte\t" + b);
9         System.out.println ("short\t" + s);
10        System.out.println ("int\t" + i);
11        System.out.println ("long\t" + l);
12
13        //Reales
14        float f = 100.0F;
15        double d = 100.0;
16        System.out.println ("REALES\nfloat\t" + f);
17        System.out.println ("double\t" + d );
18    }
19 }
```

ENTEROS	
byte	16
short	8
int	2
long	10
REALES	
float	100.0
double	100.0

Uso de _

En una literal numérica se puede utilizar el _ para mejorar la legibilidad:

```
int numero = 1_000_000; // = 1000000
int binario = 0b10000001_11100110;
int hexadecimal = 0x7fa5_00dc__54bc;
final doble constante = 3.14_16; // = 3.1416
double decimal = 0.000_000_019;
```

Secuencias de escape

- Son caracteres especiales que representan caracteres no imprimibles o funciones especiales.

CARACTER	SECUENCIA DE ESCAPE
Retorno de carro	\r
Nueva línea	\n
Tabulador	\t
Backspace	\b
Formfeed o salto de hoja	\f
Comilla simple	'
Comilla doble	"
Diagonal inversa	\\"

Cadenas de caracteres

- ¿Cómo guardar cadenas de caracteres si solo se cuenta con **char**?
- En Java las cadenas de caracteres como una *clase específica*, no como tipo de dato primitivo.
- Las cadenas se encierran en comillas dobles ("") en lugar de las simples ('') como en los caracteres.
- Text blocks en Java 15 ("""") como los de Python

Expresiones

- Elementos que, al ser evaluadas o resueltas, regresan un valor.
- Formadas por:
 - Literales (65, 'P', -53.8, true)
 - Variables (edad, nombre, direccion)
 - Operadores (edad+5, "Hola" + "mundo", 7 * (5-2))
 - Llamado a funciones o métodos (calcula(), suma(7, 9))
 - Etc.
- En Java, una asignación es una expresión.

Sentencia

- Es la expresión independiente más pequeña que Java puede trabajar.
- Termina con punto y coma (;
- Están formadas por expresiones.
- Una instrucción puede estar formada por subexpresiones.
- Hay diferentes tipos de sentencias

```
int i;           //Declaración
i=12;           //Asignación
System.out.println("Hola"); //Llamada a método
```

Bloques

- Hay instrucciones que pueden ser agrupadas por medio de llaves {} para formar un bloque.
- Es recomendable aumentar un tabulador al inicio de cada sentencia dentro de un bloque (*indentar*).
- Una instrucción sencilla siempre puede agruparse con llaves.

```
class Sueldo { //La definición de una clase es un Bloque
    public static void main (String[] args) { //La definición de un método es un Bloque
        //Sueldo de $40,000, 5 faltas, ISR de 35%
        int iFaltas=5;
        float fSueldo=40000, fSueldoAPagar;
        float fPorcentajeAumento=30;
        float fPorcentajeISR = 35;
        fSueldoAPagar = 0; //Sueldo + Aumento - 5 faltas - ISR
        System.out.println ("Sueldo a pagar después de impuestos:\t$" + (fSueldoAPagar));
        if (fSueldoAPagar > 3000) { //Los bloques se utilizan para agrupar instrucciones en if, for, while,etc.
            ...
        } else {
            ...
        }
    }
}
```

Ámbito en los bloques

- Cada vez que se abre un nuevo bloque, se define un nuevo ámbito (scope) donde se pueden definir variables y objetos que “viven” sólo en ese espacio específico.

```
class Precedencia {  
    public static void main (String[] args) {  
  
        //Scope  
  
        int i=1;  
        {  
            int j=2;  
            j=i; //Ok  
        };  
        j=10; //Error por que aquí ya no existe j  
    }  
}
```

Clases equivalentes

Tipo de dato primitivo	Clase equivalente
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character

Ejemplo de clases equivalentes

```
1 class EjemploWC {  
2     public static void main (String[] args) {  
3         Integer entero; //Define objeto tipo Integer (no int)  
4         Double doble; //Define objeto tipo Double (no double)  
5  
6         entero = 345; //Inicializa Integer  
7         entero = Integer.parseInt ("867");  
8         doble=201.9; //Inicializa Double  
9  
10        System.out.println ("entero =" + entero);  
11        System.out.println ("doble =" + doble);  
12        System.out.print ("entero como doble =" + entero.doubleValue() );  
13    } // Fin de main()  
14 } //Fin de class
```

Autoboxing y Unboxing

Tipo de dato primitivo -> Clase equivalente

Autoboxing

- Proceso automático que convierte un tipo de dato primitivo a su clase equivalente (int a Integer por ejemplo)

Unboxing

- Proceso inverso a autoboxing donde se convierte de un objeto de clase equivalente al tipo de dato primitivo que le corresponde.

Clase equivalente -> Tipo de dato primitivo

```
jshell> int variableEntera=10;  
variableEntera ==> 10  
  
jshell> Integer objetoEntero = 20;  
objetoEntero ==> 20  
  
jshell> variableEntera = variableEntera + objetoEntero;  
variableEntera ==> 30  
  
jshell> objetoEntero = objetoEntero -variableEntera;  
objetoEntero ==> -10
```

Promoción

- Es una conversión automática, sucede cuando el tipo de datos de la izquierda (en una asignación) es mayor (en tamaño de memoria) al de la derecha:

```
long<- int  
int <- short  
float <- int
```

- En general, aplica para cualquier expresión

Casting

- Es una conversión manual, se tiene que hacer de manera explícita para forzar al compilador a hacer una conversión de datos

```
float f=2.5F; //2.5 es double  
i = (int)(f); //Convierte f a int
```

Conversión entre tipos

	DESTINO									
	boolean	char	byte	short	int	long	float	double	String	
ORIGEN	boolean	X	X	X	X	X	X	X		Boolean.toString(b) String.valueOf(b)
	char	X		C	C	P	P	P		Character.toString(c) String.valueOf(c)
	byte	X	C		P	P	P	P		Byte.toString(b) String.valueOf(b)
	short	X	C	C		P	P	P		Short.toString(sh) String.valueOf(sh)
	int	X	C	C	C		P	P		Integer.toString(i) String.valueOf(i)
	long	X	C	C	C	C		P		Long.toString(l) String.valueOf(l)
	float	X	C	C	C	C			P	Float.toString(f) String.valueOf(f)
	double	X	C	C	C	C	C			Double.toString(d) String.valueOf(d)
	String	Boolean.parseBoolean(s)	s.charAt(0)	Byte.parseByte(s)	Short.parseShort(s)	Integer.parseInt(s)	Long.parseLong(s)	Float.parseFloat(s)	Double.parseDouble(s)	

Entrada de datos desde el teclado

- Scanner (Java 6 y posteriores)

```
import java.util.*;  
class CapturaTeclado {  
    public static void main (String[] args) {  
        Scanner teclado = new Scanner(System.in); //Siempre  
  
        String cadena = teclado.nextLine();           //String  
        int entero = teclado.nextInt();              //int  
        double promedio = teclado.nextDouble();       //double  
        char caracter = teclado.next().charAt(0);     //char  
    } //Fin del main()  
} //Fin de class
```

Operadores aritméticos

Operador	Función	Ejemplo	Valor de regreso
+	Suma	$5 + 2 (=7)$	La suma de las expresiones
-	Resta	$5 - 2 (=3)$	La resta de las expresiones
-	Menos (operador unario)	-68	El valor negativo de la literal numérica
*	Multiplicación	$5 * 2 (=10)$	La multiplicación de las expresiones
/	División	$10.5 / 3 (=3.5)$	La división de las expresiones
%	Módulo (residuo)	$5 \% 2 (=1)$ $100 \% 8 (=4)$	El residuo de la división entera de las expresiones
=	Asignación	$i = 5+4$	El valor asignado

Operadores aritméticos. Continuación

Operador	Función	Ejemplo	Valor que regresa
<code>+=</code>	Asignación compuesta	<code>i += 3; //i=i + 3</code>	La suma de las expresiones
<code>-=</code>		<code>i -= 3; //i=i - 3</code>	
<code>*=</code>		<code>i *= 3; //i=i * 3</code>	
<code>/=</code>		<code>i /= 3; //i=i / 3</code>	
<code>%=</code>		<code>i %= 3; //i=i % 3</code>	
<code>&=</code>		<code>i &= 3; //i=i & 3</code>	
<code> =</code>		<code>i = 3; //i=i 3</code>	
<code>++</code>	Incremento en 1	<code>i++; //i=i + 1 (1)</code> <code>++i; (2)</code>	1.- El valor de la variable 2.- El valor de la variable incrementada
<code>--</code>	Decremento en 1	<code>i--; //i=i - 1 (1)</code> <code>--i; (2)</code>	1.- El valor de la variable 2.- El valor de la variable decrementada

Incremento / Decremento en 1

- Una operación muy común es incrementar o decrementar en un 1 a una variable (sumarle 1 o restarle 1).
- `i++` Usar el valor y después incrementa
- `++i` Incrementa y después de usa el valor
- **¿Qué se obtiene con la siguiente línea?
(probar)**

`i = i++;`

Operadores de comparación

Operador	Uso	Verdadero cuando ...
>	$\text{exp1} > \text{exp2}$	exp1 es mayor que exp2
\geq	$\text{exp1} \geq \text{exp2}$	exp1 es mayor o igual a exp2
<	$\text{exp1} < \text{exp2}$	exp1 es menor que exp2
\leq	$\text{exp1} \leq \text{exp2}$	exp1 es menor o igual a exp2
\equiv	$\text{exp1} \equiv \text{exp2}$	exp1 y exp2 son iguales
\neq	$\text{exp1} \neq \text{exp2}$	exp1 y exp2 son diferentes

Operadores lógicos

Operador	Uso	Verdadero cuando ...
<code>&&</code>	<code>exp1 && exp2</code>	<code>exp1</code> y <code>exp2</code> son ambos verdaderos. Condicionalmente evalúa <code>exp2</code>
<code>&</code>	<code>exp1 & exp2</code>	<code>exp1</code> y <code>exp2</code> son ambos verdaderos. Siempre evalúa <code>exp1</code> y <code>exp2</code>
<code> </code>	<code>exp1 exp2</code>	<code>exp1</code> o <code>exp2</code> son verdaderos (cualquiera). Condicionalmente evalúa <code>exp2</code>
<code> </code>	<code>exp1 exp2</code>	<code>exp1</code> o <code>exp2</code> son verdaderos (cualquiera). Siempre evalúa <code>exp1</code> y <code>exp2</code>
<code>!</code>	<code>! exp1</code>	<code>exp1</code> es falso. Es la negación

Tabla de
verdad del
operador

	Expresión 1	Expresión 2	Resultado
	true	true	
	true	false	
	false	true	
	false	false	

Operador ternario

```
condición? Expresión1 : Expresión2;
```

- Evalúa *condición*, si es verdadera entonces regresa *Expresión1*, de lo contrario regresará *Expresión2*

Operadores a nivel de bit

- Trabajan a nivel de bits (cuando cada bit puede tener un significado diferente, dependiendo de la posición).
- Existen dos tipos:
 - Corrimiento o desplazamiento de bits
 - De lógica de bits

Corrimiento o desplazamiento de bits

Operador	Uso	Verdadero cuando ...
<code>>></code>	<code>exp1 >> exp2</code>	Desplaza los bits de <code>exp1</code> a la derecha <code>exp2</code> veces, respetando el bit de signo
<code><<</code>	<code>exp1 << exp2</code>	Desplaza los bits de <code>exp1</code> a la izquierda <code>exp2</code> veces
<code>>>></code>	<code>exp1 >>> exp2</code>	Desplaza los bits de <code>exp1</code> a la derecha <code>exp2</code> veces, insertando un cero en el bit más a la izquierda

- Trabaja sobre valores numéricos y obtiene de resultado otro valor numérico, moviendo los bits de `exp1` hacia la izquierda o derecha `exp2` veces:

`20 >> 2 (10100 >> 2) produce 5 (00101)`

`20 << 2 (10100 << 2) produce 80 (1010000)`

- Cada corrimiento a la derecha es una división por 2.
- Cada corrimiento a la izquierda es una multiplicación por 2.

Carlos Eligio Ortiz (carloselijo@ortizm.com)

Operadores de lógica de bits

- Permite trabajar con hacer operaciones sobre los bits de dos números, y obtiene un tercero.
- Puede sumar bits (`|`), multiplicarlos (`&`), hacer XOR (`^`) o hacer un complemento (`~`)

Operador	Uso	Operación que realiza
<code>&</code>	<code>exp1 & exp2</code>	AND
<code> </code>	<code>exp1 exp2</code>	OR
<code>^</code>	<code>exp1 ^ exp2</code>	XOR (OR exclusivo)
<code>~</code>	<code>~exp1</code>	Complemento

$$\begin{array}{r}
 25 \\
 3
 \end{array}
 \begin{array}{l}
 \& \begin{array}{r}
 1 & 1 & 0 & 0 & 1 \\
 0 & 0 & 0 & 1 & 1 \\
 \hline
 0 & 0 & 0 & 0 & 1 \\
 \quad (1)
 \end{array} \\
 \mid \begin{array}{r}
 1 & 1 & 0 & 0 & 1 \\
 0 & 0 & 0 & 1 & 1 \\
 \hline
 1 & 1 & 0 & 1 & 1 \\
 \quad (27)
 \end{array}
 \end{array}$$

$$\begin{array}{r}
 25 \\
 3
 \end{array}
 \begin{array}{l}
 \wedge \begin{array}{r}
 1 & 1 & 0 & 0 & 1 \\
 0 & 0 & 0 & 1 & 1 \\
 \hline
 1 & 1 & 0 & 1 & 0 \\
 \quad (26)
 \end{array}
 \end{array}$$

Precedencia de operadores

¿Cuándo es la mitad de dos más dos?

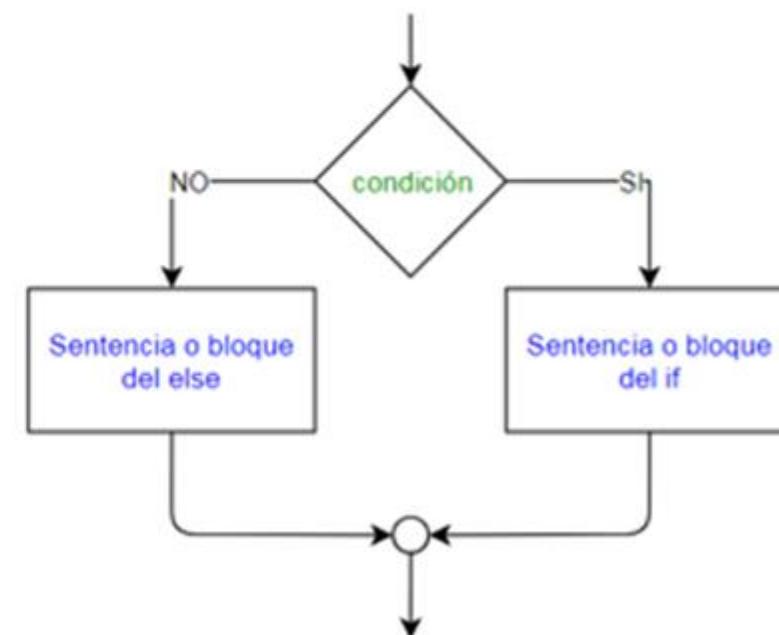
```
int i=0;  
i = 2 + 2 / 2;  
System.out.println (i);  
  
i=20;  
i = i++ / 2 + ++i / 2;  
System.out.println (i);
```

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?:	Right to left
= += -= *= /= %= &= = ^= <<= >>= >>>=	Right to left

Sentencia if-else

- Permite la ejecución de un bloque de sentencias u otro, dependiendo de la evaluación de una expresión booleana

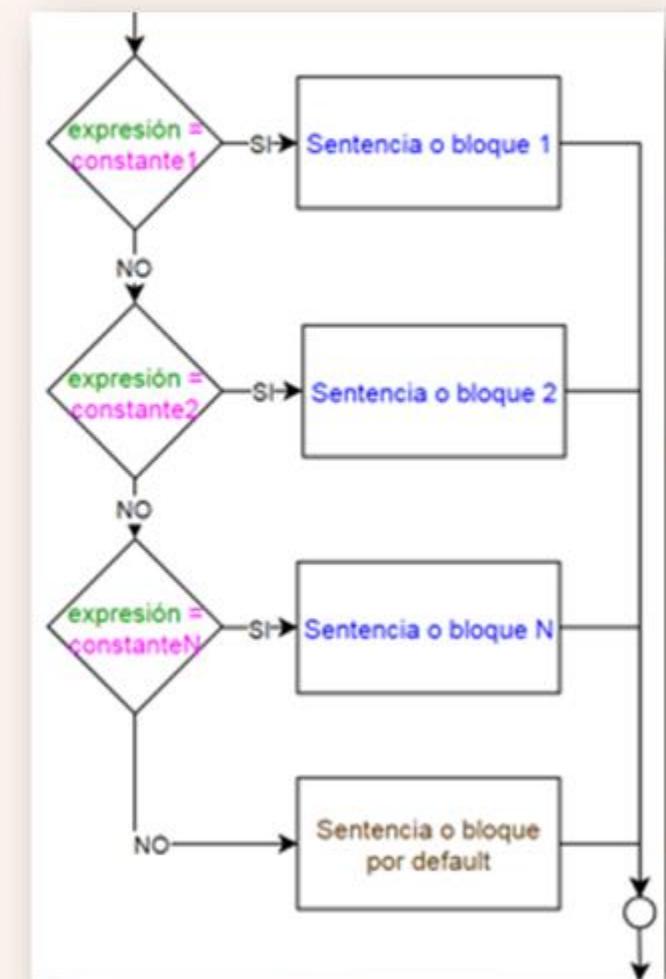
```
if (condición)
    sentencia-o-bloque
else
    sentencia-o-bloque
```



switch como sentencia

Permite comparar una expresión contra diversas constantes para ejecutar un solo bloque de código. Equivalente a if-else anidados.

```
switch (expresión) {  
    case constante1:  
        sentencia-o-bloque-1;  
        break;  
    case constante2:  
        sentencia-o-bloque-2;  
        break;  
    case constanteN:  
        sentencia-o-bloque-N;  
        break;  
    default:  
        sentencia-o-bloque-por-default;  
        break;
```



Ejemplos de *switch* como sentencia

```
public static void main (String[] args) {  
    // Con valor numérico  
    int numero=Integer.valueOf(args[0]);  
    int variableInt = 6;  
    final int CONSTANTE_INT=5;  
    switch (numero) {  
        case 0: // literal  
            System.out.println ("Es cero");  
            break;  
        case CONSTANTE_INT: // constante  
            System.out.println ("Es "+CONSTANTE_INT);  
            break;  
        case CONSTANTE_INT + 3:  
            System.out.println ("Es "+(CONSTANTE_INT + 3));  
            break;  
        case (variableInt): //Constant expression required  
            System.out.println ("Es "+ (variableInt));  
            break;  
        default:  
            System.out.println ("Es un valor ENTERO no listado");  
    }  
}
```

```
// Con valor String  
String texto=args[1];  
String variableStr = "b";  
final String CONSTANTE_STR = "HOLA";  
switch (texto) {  
    case "a":  
        System.out.println ("Es \'a\'");  
        break;  
    case CONSTANTE_STR:  
        System.out.println ("Es "+CONSTANTE_STR);  
        break;  
    case "10":  
        System.out.println ("Es diez");  
        break;  
    default:  
        System.out.println ("Es un valor String no listado");  
}
```

```
switch (mes){  
    case 1,3,5,7,8,10,12: // Lista de valores  
        System.out.println ("Mes de 31 días");  
        break;  
  
    case 4: // Diferentes case's en secuencia  
    case 6:  
    case 9:  
    case 11:  
        System.out.println ("Mes de 30 días");  
        break;  
  
    default:  
        System.out.println ("Es Febrero (28 o 29 días)");  
}
```

```
>java EjemploSwitch 1 bye  
Ejemplo de switch  
Es un valor ENTERO no listado  
Es un valor String no listado  
  
>java EjemploSwitch 8 HOLA  
Ejemplo de switch  
Es 8  
Es HOLA
```

switch como expresión (v. 14 o posterior)

```
class ExpresionSwitch{
    public static void main (String[] args) {
        String dia="desconocido";
        dia = switch (args[0]) {
            case "L", "l" -> "Lunes";
            case "M", "m" -> "Martes";
            case "Mi", "mi", "MI" -> "Miércoles";
            //...
            default -> "Día desconocido";
        }; //Fin de la expresión switch
        System.out.println (dia);
    }
}
```

```
>java ExpresionSwitch L
Lunes

>java ExpresionSwitch m
Martes

>java ExpresionSwitch MI
Miércoles

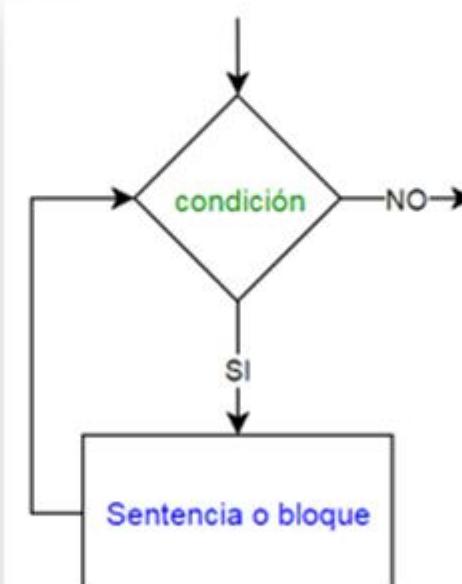
>java ExpresionSwitch Mi
Miércoles

>java ExpresionSwitch 123
Día desconocido
```

Ciclo while

- Repite una sentencia o bloque mientras una condición sea verdadera (puede ser que no se ejecute ni una sola vez)
- Cuidar no caer en ciclos infinitos

```
while(condición) {  
    sentencia-o-bloque  
}
```

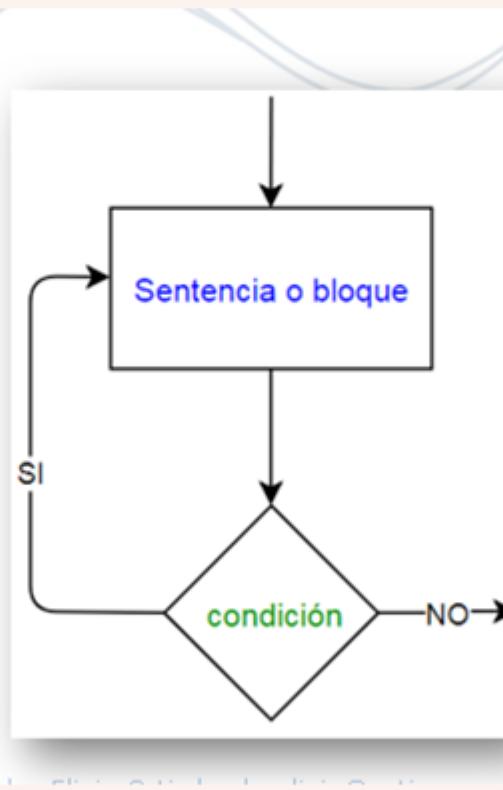


```
i=10;  
j=1;  
while ( i==10 ) {  
    System.out.println  
    j++;  
    if ( j >15 ) i=12;  
}
```

Ciclo do-while

- Repite una sentencia o bloque mientras una condición sea *verdadera*.
- La evaluación de la condición se hace **después** de la ejecución de la sentencia o bloque.

```
do {  
    sentencia-o-bloque  
} while(condición);
```



```
i=1;  
j=10;  
do {  
    System.out.println (j);  
    j++;  
    if (j >15) i=12;  
} while (i==10);
```

break

- Permite salirse de un bloque (for, do, while) o un case (switch).
- Es similar a un go to

```
i=10;  
j=1;  
while (true) {  
    System.out.println (j);  
    j++;  
    if (j >15) break;  
}
```

```
int iEntidad=9;  
switch (iEntidad) {  
    case 1:  
        System.out.println ("Aguascalientes");  
        break;  
    case 2:  
        System.out.println ("Baja California");  
        break;  
  
    case 32:  
        System.out.println ("Zacatecas");  
        break;  
    default:  
        System.out.println ("Entidad desconocida");  
        break;
```

continue

- Permite concluir una iteración de un ciclo (for, do, while)
- Es similar a un go to , pero sólo para una iteración

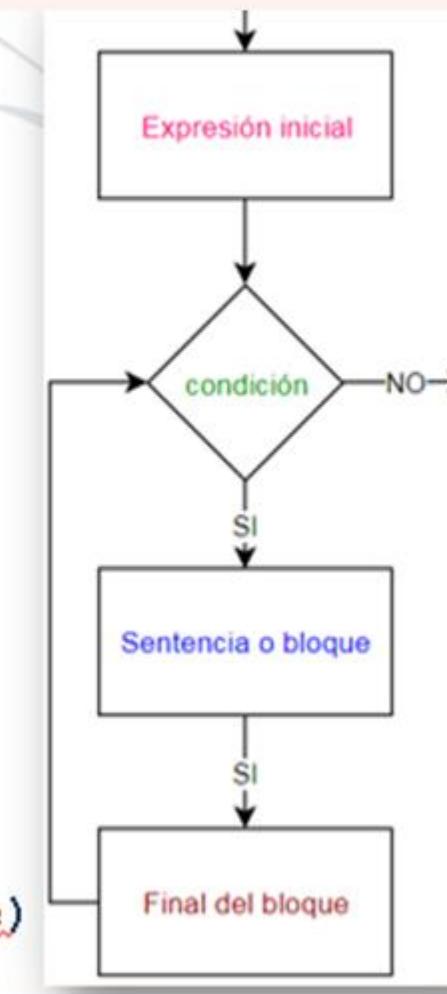
```
System.out.println ("Ejemplo de continue");
for (i=0;i<20;i++) {
    //...
    if ((i%2) != 0) { //Impar
        continue;
    }
    System.out.println (i);
}
```

Ejemplo de continue
0
2
4
6
8
10
12
14
16
18
C:\Java>_

Ciclo for

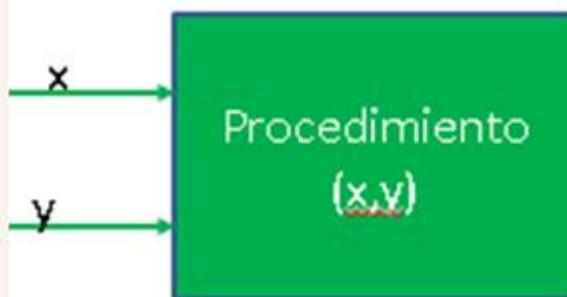
- Repite una sentencia –o bloque– mientras una condición sea verdadera.
- Usada comúnmente para recorrer un arreglo o repetir un ciclo un número determinado de veces.

```
for (expresionInicial; condición; FinalDelBloque)  
    sentencia-o-bloque;
```



Funciones y procedimientos en Java

- En Java no existen funciones o procedimientos (todos son métodos) pero se puede tener el mismo comportamiento con métodos que regresan valor (**funciones**) y métodos con valor de regreso void (**procedimientos**).



Función

- Método que puede, o no, recibir parámetros de entrada y puede regresar un valor de un tipo definido con la instrucción `return`.
- Se manda llamar como parte de otra expresión.

```
int suma (int sumando1, int sumando2) {  
    int resultado=0;  
    resultado = sumando1 + sumando2;  
    return (resultado);  
}
```

Procedimiento

- Método que puede recibir parámetros de entrada y no regresa valor a quien lo llama (es de tipo void y no tiene instrucción return).
- Se manda llamar en una línea por separado;

```
void imprime (String nombre) {  
    System.out.print ("Hola " + nombre);  
}
```

Recursividad

- Algunos lenguajes permiten ejecutar una función dentro de su misma definición, es decir, una función que se llama a sí misma.
- Es muy útil para cierto tipos de problemas.
- Hay que tener cuidado en implementar una condición de parada.
- En general, el uso de recursos puede incrementarse rápidamente.

Problemas que pueden solucionarse con recursividad

- La sumatoria de 1 a N se puede definir como:
 - $N + \text{la sumatoria de } 1 \text{ a } N-1$
- El elemento X de la serie de *Fibonacci* se puede calcular como:
 - El elemento X-1 + elemento X-2



Sumatoria recursiva

$$\text{sumatoria}(5) = 5 + \text{sumatoria}(4) \rightarrow 15$$

$$\text{sumatoria}(4) = 4 + \text{sumatoria}(3)$$

$$\text{sumatoria}(3) = 3 + \text{sumatoria}(2)$$

$$\text{sumatoria}(2) = 2 + \text{sumatoria}(1)$$

$$\text{sumatoria}(1) = 1$$

Carlos Eligio Ortiz (carloselgio@ortizm.com)

Factorial recursivo

$$n! = n \cdot (n - 1)!$$

Donde $0! = 1$

```
1 public class FactorialRecursivo{
2     public static double factorial (double n) {
3         double resultado=0;
4         resultado = (n==0)? 1: n*factorial(n-1);
5         return (resultado);
6     }
7
8     public static void main (String[] args) {
9         int numero = Integer.parseInt(args[0]);
10        System.out.println (numero+"!"+"+"+factorial(numero));
11    }
12 }
```

```
$java FactorialRecursivo 5
5!=120.0
```

```
$java FactorialRecursivo 10
10!=3628800.0
```

Paquete

- Es una colección de clases e interfaces relacionadas entre sí.
- Sirve para modularizar, controlar el acceso y manejar funcionalidades relacionadas.
- Hace más fácil encontrar y usar las clases.
- Los paquetes tienen nombres usando una estructura jerárquica (tienen identificadores separados por puntos).

Formas de usar paquetes en Java

1. Utilizando el nombre completo del elemento (clase o interfaz), incluyendo el nombre del paquete (*fully qualified name*) cada vez que se requiere.
2. Importando **una** clase o interfaz específica al inicio del código.
3. Importando **todo** el contenido de un paquete al inicio del código.

Sólo los miembros públicos de un paquete se pueden utilizar.

Eligio Ortiz (carloselgio@ortizm

Fully qualified name

- Se utiliza nombrando al elemento incluyendo el paquete donde reside o donde está definido.

```
//Usar la clase Vector que está en java.util  
java.util.Vector miVector = new java.util.Vector();
```

- A veces es útil cuando se quiere evitar una ambigüedad (Date en java.util y en java.sql).
- Sólo se va a usar un miembro del paquete.
- Es más lento (hay que escribir más)
- Si se cambia de paquete, hay que modificar todo.

Importar un miembro específico

- ***import*** permite importar un miembro específico de un paquete.
- ***import NOMBREdePAQUETE.Identificador;***

```
import java.util.Vector;  
import java.util.Hashtable;  
Vector miVector = new Vector();  
Hashtable miTablaH = new Hashtable();
```

- Útil cuando se quiere usar varias veces un elemento.
- Ya no se tiene que hacer el nombrado completo (*fully qualified*).

Importar todos los miembros de un paquete

- ***import*** permite importar todos los miembros de un paquete.
- ***import NOMBREDEPAQUETE.*;***

```
import java.util.*;  
  
Vector miVector = new Vector();  
  
Hashtable miTablaH = new Hashtable();
```

- Útil cuando se quiere usar varios elementos de un paquete.
- No hace daño importar todo (*), sólo hace el código un poco menos legible.

java.lang

Elemento	Uso
Byte, Character, Integer, Float, Double, Boolean, Long, SmallInt	Clases equivalentes
Object	Clase padre de todas las demás
Exception	Ídem, para Exception
Thread	Programación concurrente
Error	Errores
String, StringBuffer, StringBuilder	
System	
Math	Funciones matemáticas comunes

Clase java.lang.System

Elemento	Descripción	Uso (System._____)
in	La entrada estándar (teclado)	System.in
out	La salida estándar (monitor)	System.out
exit(código)	Termina ejecución con código. %ERRORLEVEL% en Windows \$? en Linux	System.exit(0)
getProperties()	Propiedades del equipo	System.getProperties()
getProperty(s)	Propiedad	System.getProperty("os.name")

```
class CódigoSalida {  
    public static void main (String[] args) {  
        System.exit (-10);  
    }  
}
```

Código de salida -10
"Fue un valor <= 0"

```
@ECHO OFF  
REM $? para Linux en lugar de %errorlevel%  
java CódigoSalida  
ECHO Código de salida %errorlevel%  
  
IF %errorlevel% GTR 0 (echo Salió con un valor positivo) ELSE (echo "Fue un valor <= 0")
```

Clase java.lang.Math

Elemento	Descripción	Uso (Math._____)
abs(n)	Valor absoluto	abs(-5) -> 5
sqrt(n), cbrt(n)	Raíz cuadrada y cúbica	sqrt (64) -> 8, cbrt(-27) -> -3
sin(n), cos(n), tan(n)	Seno, coseno, tangente	
ceil(d), floor(d)	Techo y Piso	ceil(4.5) -> 5, floor(4.5) -> 4
log(n), log10(n)	Logaritmo (natural y base 10)	
max (a,b), min(a,b)	Máximo y Mínimo de dos números	max (7,9) -> 9, min (7,9) -> 7
pow (a,b)	a^b	pow (4,3) -> 64
<u>random()</u>	Aleatorio entre 0 y 1	random()
E	Valor de e	E -> 2.718281828...
PI	Valor de π	PI -> 3.141592653...

Creación de paquetes

- En el archivo con la *clase* o *interface*, se coloca una instrucción **package** como primer línea.
- La clase/interfaz, los constructores y todo aquello que deba ser visible deberá ser **public** explícitamente.

```
package java.util;  
public class Vector {  
    public Vector() {  
        //...  
    }  
    //...  
}
```

Convenciones para nombrar paquetes

- Para evitar colisiones, se recomienda nombrar a las clases usando el dominio de internet (invertido)

```
package mx.unam.paquete;
```

Ejemplo de uso de paquetes

The screenshot shows a Java development environment with two code editors and a file browser.

Code Editor 1 (Left): Displays the `Television.java` file. This class defines a television with attributes `marca` (String), `volumen` (int), and `encendido` (boolean). It includes constructors, methods for setting and getting volume and brand, turning it on/off, and a `toString` method.

```

1 package mx.unam.entretenimiento;
2
3 public class Television {
4     private String marca; //Valores: "LG", "Samsung", "Sony", "Sin Marca"
5     private int volumen; //Rango de 0-100
6     private boolean encendido;
7
8     //Constructores
9     public Television() {
10        //No hace nada
11    };
12    public Television(boolean prendida) {
13        this(); //Llama a otro constructor
14        enciende();
15    }
16    //Permite llenar el atributo volumen, validando los valores entrantes
17    public void setVolumen (int nuevoVolumen) {
18        if (nuevoVolumen >= 0 && nuevoVolumen<=100) {
19            volumen = nuevoVolumen;
20        }
21    }
22    //Regresa el valor del atributo
23    public int getVolumen () {
24        return volumen;
25    }
26
27    public void setMarca (String nuevaMarca) {
28        if (nuevaMarca.equals("LG") || nuevaMarca.equals("Samsung") || ...
29    }
30
31    public String getMarca () {
32        return marca;
33    }
34    public void enciende () {
35        encendido = true;
36    }
37    public void apaga () {
38        encendido = false;
39    }
40    public boolean getEncendido () {
41        return encendido;
42    }
43
44
45    @Override public String toString() {
46        return "Marca: " + this.getMarca() + ", Volumen: " + this.getVolumen()
47    }

```

Code Editor 2 (Right): Displays the `PruebaTelevision.java` file. This is a test program that creates two `Television` objects, sets their attributes, and prints them to the console.

```

1 import mx.unam.entretenimiento.Television;
2 /* Pruebas de Television (Atributos) */
3 class PruebaTelevision {
4     public static void main (String[] argumentos) {
5         Television unaTelevision = new Television(true);
6         Television otraTelevision = new Television(false);
7
8         unaTelevision.setVolumen(10);
9         otraTelevision.setVolumen(20);
10        unaTelevision.setMarca ("LG");
11
12        unaTelevision.setVolumen(34);
13        System.out.println ("Televisión 1. Encendido: " + unaTelevision.getEncendido());
14        System.out.println ("Televisión 2. Volumen: " + otraTelevision.getVolumen());
15        System.out.println ("una Television "+ unaTelevision);
16    }
17
18

```

File Browser: Shows the project structure in the bottom right corner. The `Java` folder contains several exercise subfolders (ejercicio1.1, ejercicio2.1, etc.) and the `entretenimiento` package, which contains the `Television` class and its source file.

- Escritorio
- + Mis documentos
- + Mi PC
 - + Disco local (C:)
 - + Archivos de programa
 - + dem_11
 - + Documents and Settings
 - + Java
 - ejercicio1.1
 - ejercicio2.1
 - ejercicio2.2
 - ejercicio2.3
 - ejercicio2.4
 - ejercicio3.1
 - ejercicio3.2
 - + ejercicio3.5
 - mx
 - + unam
 - entretenimiento

Arreglos en Java

- Permite agrupar un conjunto de elementos del mismo tipo, haciendo referencia a todos ellos por medio de un nombre en común.
- Se pueden crear elementos de cualquier tipo primitivo, de objetos o de otros arreglos.

```
int[] miArreglo; //Arreglo de int's  
int miArreglo[] ; //Forma alterna de declarar
```

(sólo se crean las referencias, no el contenido)

Declaración y creación de arreglos

- Se debe usar new para crear la instancia

```
int[] miArreglo = new int[10];  
int miArreglo[] = new int[10];  
int[] canales = {2,4,5,7,9,11,13,22,28,34,40};  
TV[] miTV = {new TV(), new TV(34)};
```

- En general la forma de declarar es:

```
Tipo[] NombreDeArreglo = new Tipo[tamaño]; //Recomendable  
Tipo NombreDeArreglo[] = new Tipo[tamaño];
```

- El acceso es por medio del índice:

```
miArreglo[3] = 11; // Asigna el valor el 11 a la casilla con índice 3  
int ultimoCanal = canales[5]; // Extrae el valor de la casilla con índice 5
```

```
DIPLO jshell> int[] arreglo = new int[10];
De: arreglo ==> int[10] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }

jshell> arreglo[5]=98;
$22 ==> 98

jshell> arreglo[0]=-23
$23 ==> -23

jshell> arreglo
arreglo ==> int[10] { -23, 0, 0, 0, 0, 98, 0, 0, 0, 0 }

jshell> arreglo[99]=11
| Exception java.lang.ArrayIndexOutOfBoundsException: Index 99 out of bounds for length 10
|     at (#25:1)
```

```
arreglo.length
10

arreglo.getClass()
class [I
```

```
jshell> int[] nuevoArreglo
nuevoArreglo ==> null

jshell> nuevoArreglo = arreglo
nuevoArreglo ==> int[10] { -23, 0, 0, 0, 0, 98, 0, 0, 0, 0 }

jshell> arreglo
arreglo ==> int[10] { -23, 0, 0, 0, 0, 98, 0, 0, 0, 0 }

jshell> nuevoArreglo[2]=22;
$31 ==> 22

jshell> nuevoArreglo[7]=77;
$32 ==> 77

jshell> nuevoArreglo
nuevoArreglo ==> int[10] { -23, 0, 22, 0, 0, 98, 0, 77, 0, 0 }

jshell> arreglo
arreglo ==> int[10] { -23, 0, 22, 0, 0, 98, 0, 77, 0, 0 }
```

```
jshell> nuevoArreglo = arreglo.clone()
nuevoArreglo ==> int[10] { -23, 0, 22, 0, 0, 98, 0, 77, 0, 0 }

jshell> arreglo
arreglo ==> int[10] { -23, 0, 22, 0, 0, 98, 0, 77, 0, 0 }

jshell> nuevoArreglo
nuevoArreglo ==> int[10] { -23, 0, 22, 0, 0, 98, 0, 77, 0, 0 }

jshell> arreglo[4]=44;
$38 ==> 44

jshell> nuevoArreglo[9]=99
$39 ==> 99

jshell> arreglo
arreglo ==> int[10] { -23, 0, 22, 0, 44, 98, 0, 77, 0, 0 }

jshell> nuevoArreglo
nuevoArreglo ==> int[10] { -23, 0, 22, 0, 0, 98, 0, 77, 0, 99 }
```

Arreglos multidimensionales

- Cada elemento de un arreglo puede ser un valor primitivo, un objeto, e incluso otro arreglo:

```
int[][] miArreglo=new int[3][3]; //De 3x3 int's  
int[][] miArreglo = {{1,2,3},{4,5,6},{7,8,9}};
```

1	2	3
4	5	6
7	8	9

- No todas las dimensiones tienen que ser del mismo tamaño.
En general, la forma de declarar es:

```
int[][] miArreglo = new int[3][];  
miArreglo[0] = new int[3];  
miArreglo[1] = new int[2];  
miArreglo[2] = new int[1];
```

1	2	3
4	5	
7		

java.util.Arrays

- Arrays permite operaciones con arreglos, como ordenamientos o búsquedas. Algunos de sus métodos más comunes son:

```
Arrays.fill (miArreglo, 10); //llena miArreglo con 10's
```

```
Arrays.equals (unArreglo,otroArreglo); //Regresa true si todos los  
elementos son iguales
```

```
Arrays.sort (miArreglo); //Ordena ascendentemente todos los  
elementos
```

```
Arrays.binarySearch (miArreglo, 6); //regresa la ubicación de 6
```

Orientación a objetos

- Organiza sistemas alrededor de objetos, en lugar de hacerlo alrededor de funciones o procesos.
- Se comienza con el modelado de objetos.
- **Objeto:** es la representación de un objeto, de un concepto, una abstracción o algo.
- Cada objeto tiene un tipo. En Java se utiliza la instrucción *class* para crear nuevos tipos
- Es más fácil reutilizar código, por que se reutilizan los objetos, no las funciones

¿De qué se compone un objeto?

- Identificador que lo hace único
- Propiedades (atributos)
- Métodos (interfaz) que es la manera de comunicarse con el exterior.

Clase

- Es el tipo de un objeto, es la definición de cómo se van a comportar y componer todos los objetos de ese tipo.
- Es el *formato* que tendrán todos los objetos de su clase.
- Una clase no es un objeto, es sólo una estructura o plano para cada objeto de ese tipo.
- Cuando se genera un nuevo objeto, se hace una nueva *instancia*, se hace una *instanciación*



Creación de objetos (*instanciación*)

- Cada una de las instancias de una clase, un objeto es un elemento con el que podemos trabajar.
- Ya no es un concepto abstracto, sino que es un elemento tangible que podemos utilizar.

```
Casa miCasa = new Casa();
```

- No se puede crear una referencia a un objeto y asignarle un objeto de un tipo diferente

```
Alumno unAlumno = new Casa(); //Error
```

Referencia al objeto y Objeto en sí mismo

- La declaración de una variable crea la referencia (algo similar a una dirección donde estará el objeto).
- El comando new crea el objeto y su dirección se la asigna a la referencia



Sentencia class

```
modifAcceso modifClase class NombreClase
    [extends SuperClase] [implements NombreInterfaz] {
//Atributos
    tipo nombreAtributo1;
    tipo nombreAtributoN;

//Métodos
    tipo nombreMetodo1() {
        }

    tipo nombreMetodoN() {
        }
}
```

Sentencia class

- **nombreClase** es el nombre que tendrá la clase.
- **modifAcceso** (*public, private, protected*)
- **modifClase**
 - *abstract*. Le falta definir el código de al menos un método, lo tendrá que hacer la clase derivada. No puede ser instanciada
 - *final*. Ya no puede ser derivada
 - *extends*. Es una clase derivada de otra (default *Object*)
 - *implements* (utiliza una interfaz)

Ejemplo de definición y uso de clase

```
1 public class Casa {  
2     public int pisos=0;  
3     public double precio=0;  
4     public String color="S/C";  
5  
6     public String toString() {  
7         return "Casa de color " + this.color  
8             + ", de " + this.pisos  
9             + " pisos y con valor de $" + this.precio;  
10    }  
11}  
  
$jshell  
| Welcome  
| For an  
|  
jshell> /  
| Command. ,  is ambiguous. /list, /edit, /drop, /save, /open, /vals, /methods, /types, /imports, /exit  
, /env, /reset, /reload, /history, /debug, /help, /set, /?, /!  
| Type /help for help.  
  
jshell> /open Casa.java  
  
jshell> Casa primera = new Casa();  
primera ==> Casa de color S/C, de 0 pisos y con valor de $0.0  
  
jshell> primera.color="Rojo"  
$3 ==> "Rojo"  
  
jshell> primera.pisos=3  
$4 ==> 3  
  
jshell> primera.precio=123.45  
$5 ==> 123.45  
  
jshell> primera  
primera ==> Casa de color Rojo, de 3 pisos y con valor de $123.45
```

Atributos y Métodos

- Atributos son variables que tiene cada objeto.
- Dos objetos pueden tener todos sus atributos iguales, pero no por eso son el mismo objeto.
- Los métodos son la forma de comunicarse con el objeto

¿Qué pasa cuando se asigna un objeto a otro?

```
Casa miCasa = new Casa();  
Casa otraCasa = miCasa;
```

Nombre	Referencia
miCasa	Calle 1 # 2
otraCasa	Calle 1 # 2



null

- Se puede inicializar una referencia de un objeto a *null*

```
Casa miCasa = null;  
if (miCasa == null) ... //dará true
```

Nombre	Referencia
miCasa	

Atributos o variables de instancia

- Cada objeto tiene una copia de esos atributos. Si no hay constructor, o el constructor no lo hace, las variables de instancia se llenan con valores por default

```
class Casa {  
    int numero = 0;  
    int pisos=2;  
// ...  
- - - - -  
Casa miCasa = new Casa();  
Casa otraCasa = new Casa();
```

Nombre	Referencia
miCasa	Dirección de memoria
otraCasa	Otra dirección



Carlos Eligio Ortiz (carloselgio@ortizm.

Constructores

- Se llama al ejecutar el *new*, tiene el mismo nombre de la clase y puede haber varios (con diferentes parámetros de entrada), mismos que estarán *sobrecargados* (se verá más adelante).
- Nos permiten crear, con diferentes fuentes, el mismo objeto

```
Usuario miUsu = new Usuario();           //usuario vacío
Usuario miUsu = new Usuario(7823);      //con ID = 7823
Usuario miUsu = new Usuario("elopez");  //con usuario "elopez"
```

Métodos

- Cada objeto tiene una copia de sus métodos que, al ejecutarse, operan sobre el objeto al que pertenecen.

```
class Casa {  
    // Atributos  
    public void calculaPredial () {  
        //Código de calculaPredial()  
    }  
}
```

```
Casa miCasa = new Casa();  
Casa otraCasa = new Casa();  
...  
miCasa.calculaPredial();  
...  
otraCasa.calculaPredial();  
//En general será objeto.método()
```

Métodos *set* y *get/is*

- También llamados métodos mutadores y accesores.
- Para tener mayor control sobre los atributos, se recomienda crear siempre métodos para cambiarlos (*setAtributo*) y leerlos (*getAtributo* o *isAtributo*).
- Para no dejar “visible” el atributo (*objeto.atributo*) se utilizará el modificador *private* al definir el atributo

```
class Television {  
    private String marca; //Valores: "LG", "Samsung", "Sony", "Sin Marca"  
    private int volumen; //Rango de 0-100  
  
    //Permite llenar el atributo volumen, validando los valores entrantes  
    public void setVolumen (int nuevoVolumen) {  
        if (nuevoVolumen >= 0 && nuevoVolumen<=100) {  
            volumen = nuevoVolumen;  
        }  
    }  
    //Regresa el valor del atributo  
    public int getVolumen () {  
        return volumen;  
    }  
}
```

Sobrecarga de métodos

- Métodos con el mismo nombre.
- Los métodos deben variar en los parámetros de entrada (en la cantidad y/o en el tipo de los parámetros).
- Podrían tener diferente valor (tipo) de regreso.
- *println* es un ejemplo, ya que puede recibir un *String*, un entero, un *char*, un *long*, etc. como parámetro de entrada.

equals() y toString()

Comparación con equals()

- Determina cuándo dos objetos son iguales

```
public boolean equals (Object otro){  
    boolean sonIguales = false;  
    if (this == otro) //Refieren al mismo objeto  
        sonIguales = false;  
    else  
        if (otro == null) //El segundo no referencia a objeto  
            sonIguales = false;  
        else  
            if ( this.getClass() != otro.getClass() ) //Son clases diferentes  
                sonIguales = false;  
                //Si es necesario, usar instanceof  
            else {  
                Pelicula nueva = (Pelicula)otro; //otro lo pasa a Pelicula  
                //Criterio para determinar si dos peliculas son iguales  
                if (this.nombre.equals (nueva.nombre) &&  
                    this.duracion == nueva.duracion)  
                    sonIguales = true;  
                else  
                    sonIguales = false;  
            }  
    return (sonIguales);  
}
```

toString()

- Regresa la representación en String del objeto.

```
public String toString(){  
    return (this.nombre + " de " + this.duracion + " minutos "+ this.estrenada);  
}
```

Comparación con compareTo()

- Determina la “posición” de dos objetos (si es menor, es igual o es mayor uno con respecto al otro). Regresa valor numérico.

```
public int compareTo (Pelicula otra) {  
    int regreso=0; //Son iguales por default  
    if (this.duracion > otra.duracion) {  
        regreso = 1;  
    } else {  
        if (this.duracion < otra.duracion) {  
            regreso = -1;  
        }  
    }  
    return (regreso);  
}
```

Ejemplo de `toString()`, `equals()` y `compareTo()`

```
Pelicula p1 = new Pelicula();
Pelicula p3 = new Pelicula();

p1.duracion = 10;
p3.duracion = Integer.parseInt(args[0]);
//Uso de toString()
System.out.println (p1);
System.out.println (p3);
//Uso de compareTo()
System.out.println ("Comparación: " +p1.compareTo(p3));
//Uso de equals()
System.out.println ("¿Con iguales?: " +p1.equals(p3));
```

```
>java EjemploInstanciacion -10
desconocido de 10 minutos true
desconocido de -10 minutos false
Comparación: 1
¿Con iguales?: false

>java EjemploInstanciacion 10
desconocido de 10 minutos true
desconocido de 10 minutos false
Comparación: 0
¿Con iguales?: true

>java EjemploInstanciacion 20
desconocido de 10 minutos true
desconocido de 20 minutos false
Comparación: -1
¿Con iguales?: false
```

Otro ejemplo de `compareTo()` de la interfaz Comparable

Después de implementar la interfaz Comparable:

```
@Override
public int compareTo(Object otro){
    int regreso=0; // igual
    if (this != otro)
        if (otro == null) // otro es nulo
            regreso = 1; // Es mayor
        else
            if (this.getClass() != otro.getClass()) // No son de la misma clase
                regreso = 1; // Es mayor
            else {
                ClaseA otroClaseA = (ClaseA)otro;
                if (this.atributoPublico > otroClaseA.atributoPublico)
                    regreso = 1; // Es mayor
                if (this.atributoPublico < otroClaseA.atributoPublico)
                    regreso = -1; // Es menor
            }
    return regreso;
}
```

Control de acceso

- Java provee modificadores de acceso para controlar la visibilidad de clases, atributos, métodos: *public*, *protected* y *private*.
- Se aplican por igual a métodos y a atributos.
- Los modificadores dan la posibilidad de prevenir/permitir el acceso a variables y métodos de las clases.
- Asegura que se oculta/expone correctamente el objeto (encapsulamiento) exponiendo a diferentes niveles los elementos adecuados.

private

- Un elemento *private* sólo es accesible (visible) dentro de la clase que lo definió.
- Sólo los métodos definidos dentro de la clase pueden acceder al campo.
- Es una buena práctica hacer todos los atributos privados, y acceder a ellos por medio de los métodos *get* y *set*.

public

- Un elemento *public* es accesible (visible) dentro y fuera de la clase que lo definió.
- Cualquiera que pueda ver la clase, podrá ver sus elementos públicos.
- Es muy común para los métodos que definen el comportamiento de la clase.

protected

- Los miembros definidos en una clase como protected solo son accesibles dentro del paquete donde está definida la clase (como si fueran públicos), pero no son accesibles fuera del paquete (comportándose en ese caso como privados) a menos que sea como parte de una subclase (el acceso lo da la herencia de clase).

Elementos estáticos

- static no es un modificador de acceso, es una directiva que permite definir algún elemento (atributo o método) como estático.
- Un atributo o método estático puede referenciarse **sin necesidad de instanciar la clase** (como `Integer.parseInt()` o `Integer.MAX_VALUE`)

Herencia

- La herencia es usada para衍生新的类从现有的类。
- La clase derivada (hija) es, regularmente, más especializada que la clase de la deriva (padre).

Persona -> Empleado -> Funcionario

- Java utiliza la palabra `extends` para indicar una herencia

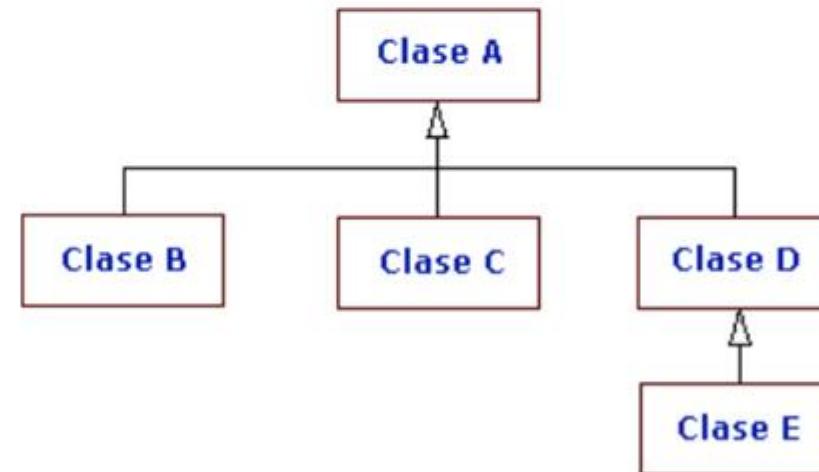
```
class NuevaClase extends ClaseOriginal  
class Empleado extends Persona
```

Herencia

- Java sólo permite herencias simples (un padre) y se puede ver a las clases heredadas como una estructura jerárquica

Una superclase puede tener cualquier número de subclases.

Una subclase puede tener sólo una superclase.



- A es la superclase de B, C y D.
- D es la superclase de E.

- B, C y D son subclases de A.
- E es una subclase de D.

Herencia: modificadores de acceso

- **public** Al heredar una clase, todos los elementos públicos de la superclase están visibles en la subclase.
- **private** Sólo se usa en la clase, y no en sus subclases.
- **protected** Visible en la clase y subclases (dentro del mismo paquete).

Palabras reservadas usadas en la herencia

- `extends` especifica la herencia
- `this` en la clase, hace referencia a elementos (atributos, métodos) de la misma clase
- `this()` constructor de la clase
- `super` hace referencia a elementos (atributos, métodos) de la clase padre.
- `super()` constructor de la súper clase.

Clases abstractas

- Similar a las interfaces, una superclase puede definir un método abstracto, de tal manera que sus subclases están obligadas a escribir el código para su implementación.
- Si una clase tiene al menos un método abstracto, toda la clase tiene que definirse como abstracta.
- Una clase con métodos abstractos no puede usarse para instanciar objetos (aún está “incompleta”).

```
public class abstract Superclase {  
    ...  
    public abstract void unMetodo ();  
}
```

Clases finales

- Cuando deseamos que una clase ya no pueda ser usada para generar subclases, se deberá definir como **final**, de esta manera ya no puede ser utilizada junto con **extends**.
- Útil cuando el código de nuestras clases ya no queremos que sea reutilizada para otros fines (recomendable para temas de seguridad)

```
public final class Clase {  
    ...  
    public void unMetodo () {  
        }  
    }
```

Métodos finales

- En ocasiones no es necesario definir como **final** a la clase, sólo a algún método, para lo cual también se utiliza el modificador **final**.
- Un método **final** no se puede sobreescribir en la subclase.

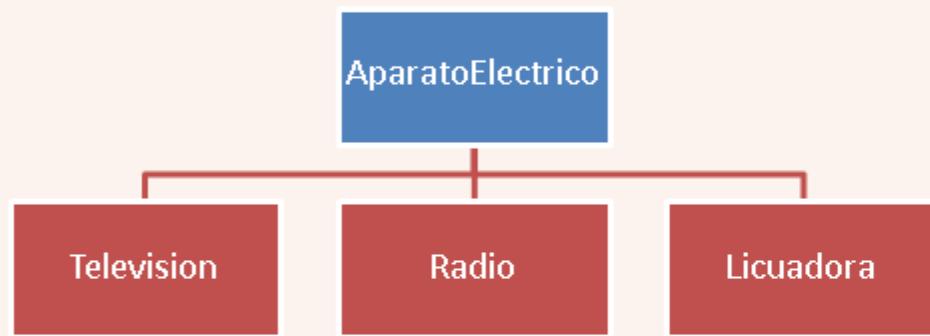
```
public class Superclase {  
    ...  
    public final void unMetodo () {  
    }  
}
```

Polimorfismo

- Habilidad para que un objeto se comporte de diferentes maneras.
- Permite ejecutar métodos de acuerdo con el tipo de objeto al momento de ejecución, y no al tipo al momento de la creación.
- Referirse a un objeto de diferentes maneras.
- Se puede hacer referencia a un objeto como su clase o como sus superclases.

Superclase miObjeto=new **Subclase()**; ✓

Ejemplo de Polimorfismo



- AparatoElectrico es la superclase y Television, Radio y Licuadora son las subclases, pero todos se pueden comportar como AparatoElectrico.
- El siguiente código no produce error:

```
AparatoElectrico[] aparatos = new AparatoElectrico[4];
aparatos[0] = new AparatoElectrico();
aparatos[1] = new Radio();
aparatos[2] = new Television();
aparatos[3] = new Licuadora();
```

Interfaces

- Una interfaz un comportamiento común a las clases que la implementen, es un concepto abstracto.
- Los métodos son *public* y *abstract* (en ese sentido se parecen a las clases abstractas) aunque se puede incluir métodos *default*.
- Los atributos definidos sólo pueden ser *public*, *static* y *final*.
- Regularmente se implementan la misma interfaz en clases que no pertenecen a la misma jerarquía.
- Solo se puede heredar de una clase, pero se pueden implementar múltiples interfaces (lo que permite hacer algo similar a una herencia múltiple).
- Un *Radio* (clase) puede ser tener métodos de un *AparatoElectrico* (interfaz) como *apagar()* y *prender()*. Una *Licuadora* (clase) puede tener la misma interfaz de *AparatoElectrico*.

Definición de Interfaces

```
public abstract interface IAjustesGenerales {  
    public static final int VOLUMEN_MINIMO=0; //Variables siempre public static final  
    public static final int VOLUMEN_MAXIMO=100;  
  
    public abstract void volumenMas () ; //Métodos siempre public abstract  
    public abstract void volumenMenos() ; //Y sin código, sólo el punto y come ()  
    public abstract void encender();  
    public abstract void apagar();  
    public abstract void silencio();  
}
```

Implementación de interfaces

- La clase que utilizará una interfaz deberá indicarlo al momento de la declaración de dicha clase

```
class Radio implements IAjustesGenerales { ... }
```

- Deberá definir cada uno de los métodos nombrados en la interfaz.

Ejemplo de uso de interfaz

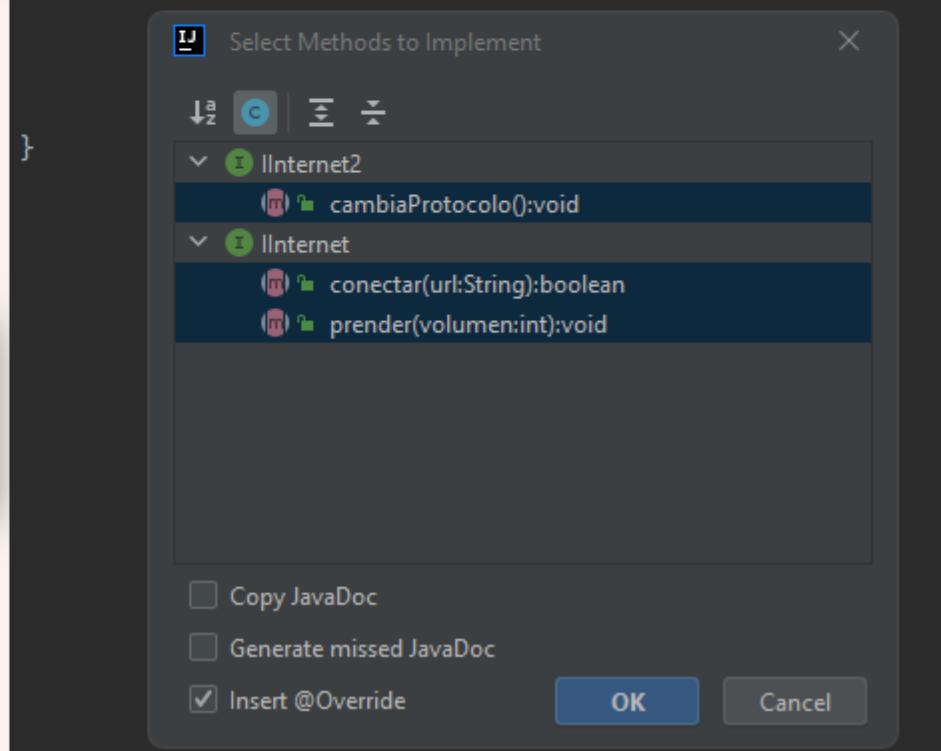
```
|public abstract interface IAjustes {  
|    public static final int VOLUMEN_MAXIMO = 100;  
|    public abstract void volumenMas ();  
|    public abstract void volumenMenos ();  
|    public abstract void encender();  
|    public abstract void apagar();  
|    public abstract void silencio();  
}|  
  
|    public class Radio implements IAjustes {  
|        private int volumen=0;  
|        private boolean encendido=false;  
|        public void volumenMas (){  
|            volumen=(volumen==VOLUMEN_MAXIMO) ? VOLUMEN_MAXIMO : volumen+1;  
|        };  
|        public void volumenMenos (){  
|            volumen=(volumen==0) ? 0:volumen-1;  
|        };  
|        public void encender(){  
|            encendido=true;  
|        };  
|        public void apagar(){  
|            encendido=false;  
|        };  
|        public void silencio(){  
|            volumen=0;  
|        };  
|    };
```

Herencia de interfaces

```
public interface IIInternet {  
    public abstract boolean conectar (String url);  
    public abstract void prender(int volumen);  
}
```

```
public interface IIInternet2 extends IIInternet{  
    public abstract void cambiaProtocolo();  
}
```

```
public class Celular extends Telefono implements IIInternet2{  
    public String banda;  
    public String sistemaOperativo;  
    private boolean encendido;
```



Herencia de clase VS Implementación de interfaz

- La herencia de clase permite heredar atributos no estáticos y comportamiento (métodos), a diferencia de la herencia de interfaz que solo obliga a implementar lo especificado en la interfaz.
- Se puede heredar de múltiples interfaces, pero solo de una clase (herencia simple).
- Si diversas interfaces implementan el mismo método, pero con diferentes parámetros (número o tipo) se aplica la sobrecarga.
- Si diversas interfaces implementan el mismo método y no es aplicable la sobrecarga, se genera una colisión.

Método **default** en interfaces

- En una interfaz, define un método con código para no tener que implementar el método de las clases que ya la implementan.
- Se definen con el modificador default.
- Automáticamente están disponibles para su ejecución en las clases que implementan la clase.
- Se utiliza para **incrementar** la funcionalidad de las clases que implementan la interfaz.
- No puede hacer uso de elementos no estáticos de la interfaz.
- La clase que la implementa puede sobreescribir el método.

Ejemplo de método *default* en interfaces

```
public interface Inflable {  
    public static String sustancia="aire";  
    default void infla(){  
        System.out.println ("Se está inflando");  
    }  
}  
  
public class Llanta implements Inflable {  
    public int rin=0; //En pulgadas  
    public String superficie = "nieve";  
    public String color="blanca";  
    public int calidad=10; //10 es nueva, 0 para desechar  
  
    public void vulcaniza (){  
        this.calidad=7;  
    }  
  
    public void infla () {  
        System.out.println ("Se infla la llanta...");  
    }  
}
```

```
public class Pelota implements Inflable {  
    public int diametro=0; //En pulgadas  
    public String color="blanca";  
    public String material="plástico";  
  
    public void pinta (String nuevoColor){  
        this.color = nuevoColor;  
    }  
}  
Welcome to JShell -- Version 16.0.2  
For an introduction type: /help intro  
jshell> /open Inflable.java  
jshell> /open Pelota.java  
jshell> /open Llanta.java  
jshell> Pelota futbol = new Pelota()  
futbol ==> Pelota@2328c243  
  
jshell> Llanta deportiva = new Llanta()  
deportiva ==> Llanta@7a4f0f29  
  
jshell> futbol.infla()  
Se est  inflando  
  
jshell> deportiva.infla()  
Se infla la llanta...
```

Excepciones y Errores

- Son utilizadas para notificar al código que se generó una condición inusual.
- Son objetos (las excepciones no son errores, son eventos para manejar errores) que contiene información sobre la excepción presentada.
- La clase de la excepción indica el tipo de excepción.
- Provocan un salto o interrupción del flujo normal.
- La interrupción puede ser controlada por código.

Manejo de excepciones

- Permite a un programa “atrapar” las excepciones, atenderlas y continuar con la ejecución del código sin detener la ejecución.

```
class Errores {
    public static void main (String[] argumentos) {
        int i;

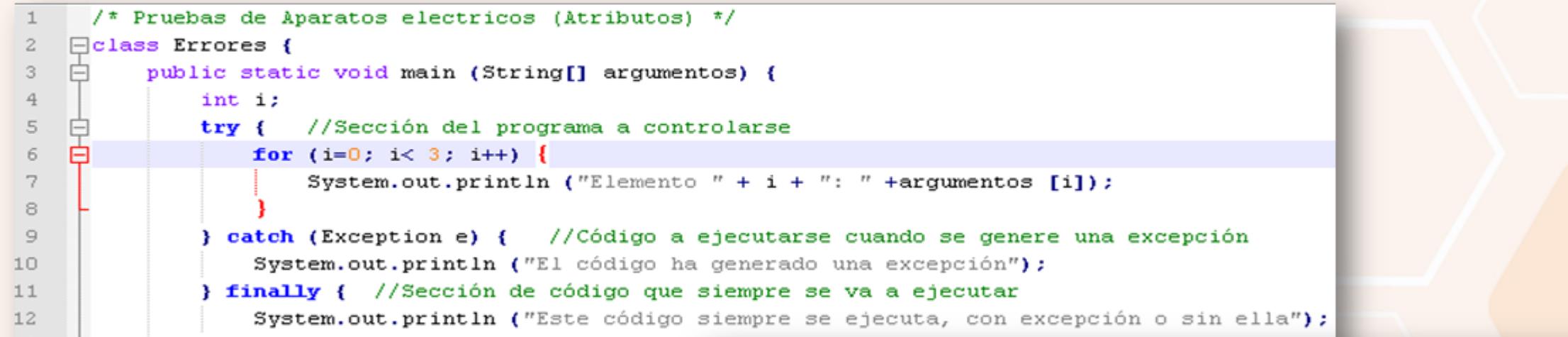
        for (i=0; i< 5; i++) {
            System.out.println ("Elemento " + i + ":" + argumentos [i]);
        }
    }
}
```

C:\WINDOWS\system32\cmd.exe

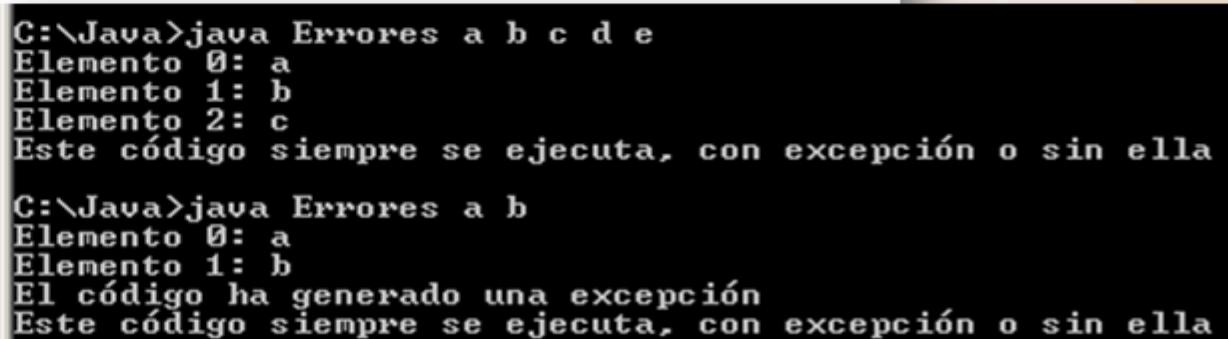
```
C:\Java>java Errores
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
at Errores.main(Errores.java:7)
```

try- catch-finally

- Para poder “atrapar” excepciones es necesario indicar la sección del código a controlar, por medio de la instrucción **try-catch-finally**.



```
1  /* Pruebas de Aparatos electricos (Atributos) */
2  class Errores {
3      public static void main (String[] argumentos) {
4          int i;
5          try { //Sección del programa a controlarse
6              for (i=0; i< 3; i++) {
7                  System.out.println ("Elemento " + i + ": " +argumentos [i]);
8              }
9          } catch (Exception e) { //Código a ejecutarse cuando se genere una excepción
10             System.out.println ("El código ha generado una excepción");
11         } finally { //Sección de código que siempre se va a ejecutar
12             System.out.println ("Este código siempre se ejecuta, con excepción o sin ella");
13         }
14     }
15 }
```



```
C:\Java>java Errores a b c d e
Elemento 0: a
Elemento 1: b
Elemento 2: c
Este código siempre se ejecuta, con excepción o sin ella

C:\Java>java Errores a b
Elemento 0: a
Elemento 1: b
El código ha generado una excepción
Este código siempre se ejecuta, con excepción o sin ella
```

try- catch-finally estructura general

```
try {
    //código potencialmente problemático
}
catch (tipoDeExcepcion1 e) {
    //manejo de tipoDeExcepcion1
}
catch (tipoDeExcepcion2 e) {
    //manejo de tipoDeExcepcion2
}
...
catch (Exception e) {
    //manejo de la excepción más general
}
finally {
    //código que siempre se ejecuta, aún sin presentar excepción
}
```

Jerarquía de excepciones

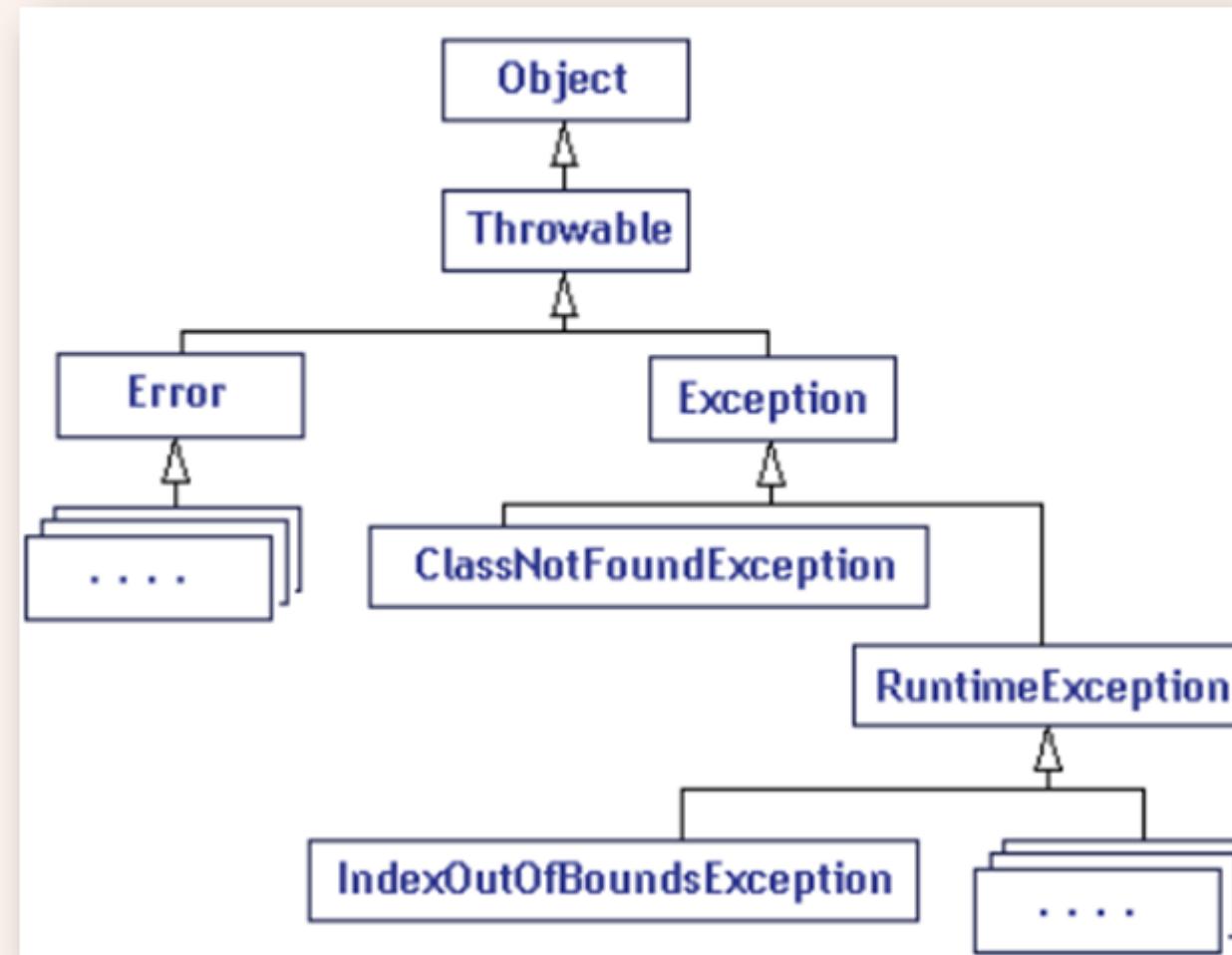
```
class Errores {  
    private static void provocaError (String[] argumentos) {  
        int i;  
        try { //Sección del programa a controlarse  
            for (i=0; i< 3; i++) {  
                System.out.println ("Elemento " + i + ": " +argumentos [i]);  
                //i = 5 / i; //Error cuando i=0  
            }  
        } catch (ArrayIndexOutOfBoundsException e) {//Excepción de indice fuera de rango  
            System.out.println ("** Índice fuera de rango en Arreglo ");  
            e.printStackTrace();  
        } catch (Exception e) { //Código a ejecutarse cuando se genere una excepción  
            System.out.println ("** El código ha generado una excepción");  
            System.out.println ("\t"+e.toString());  
            e.printStackTrace();  
        }finally { //Sección de código que siempre se va a ejecutar  
            System.out.println ("** Este código siempre se ejecuta, con o sin excepción");  
        }  
    }  
    public static void main (String[] argumentos) {  
        provocaError (argumentos);  
    }  
}
```

```
C:\Java>java Errores a b  
Elemento 0: a  
Elemento 1: b  
** Índice fuera de rango en Arreglo  
java.lang.ArrayIndexOutOfBoundsException: 2  
    at Errores.provocaError(Errores.java:7)  
    at Errores.main(Errores.java:23)  
** Este código siempre se ejecuta, con excepción o sin ella
```

Jerarquía de excepciones

- Las excepciones, como las clases, tiene su propia jerarquía. Los tipos principales son
- **Error**: Excepciones de problemas muy graves, casi siempre irrecuperables (disco lleno, p. ej.).
- **Exception**: Excepciones controlables, detectables fuera del tiempo de ejecución.
- **RuntimeException**: Se detectan en tiempo de ejecución.

Jerarquía de excepciones



Encadenamiento de excepciones

- Cuando se dispara una excepción intencionalmente (regularmente al atrapar una excepción original, aunque no necesariamente).
- Se hace uso de throw

```
class EjemploExcepcionesEncadenadas{
    public static void divide (String texto) throws Exception {
        try {
            int numero = Integer.parseInt(texto);
            System.out.println ("10/" +numero+"="+10/numero);
            numero = 10 / numero;
            System.out.println (numero);
        }
        catch (ArithmaticException e) {
            System.out.println ("Excepción en divide() "+e);
        }
        catch (Exception e) {
            // Dispara excepción de manera manual
            throw new SecurityException ("Otra excepción", e);
        }
    }

    public static void main (String[] args) {
        try {
            divide (args[0]);
            try {
                int a = 10 / 0;
            } catch (Exception e) {
                // Entra a este catch solo si hay error en el int a= 10/ 0
                System.out.println ("División por cero en el main()");
            }
        } catch (Exception e) {
            System.out.println ("Excepción en main() "+e);
        }
    }
}
```

java EjemploExcepcionesEncadenadas
Excepción en main() java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0

java EjemploExcepcionesEncadenadas 0
Excepción en divide() java.lang.ArithmaticException: / by zero
División por cero en el main()

java EjemploExcepcionesEncadenadas AAAAAA
Excepción en main() java.lang.SecurityException: Otra excepción

Aserciones (assert)

Aseveraciones que deben ser verdaderas en tiempo de ejecución

`assert condición: expresión`

- Si la `condición` es falsa entonces se dispara la excepción `AssertionError` y se despliega la `expresión`.
- Suele usarse en dos situaciones:
 1. Para definir pre o postcondiciones al ejecutar cierto código.
 2. En tiempo de desarrollo para asegurarse que lo que se asume sobre los datos es válido siempre.
- A pesar de disparar excepciones no se debe confundir su utilidad con un `try-catch` tradicional.
- Se activa de dos maneras:
 1. Al ejecutar el intérprete (`java -ea Clase`)
 2. Configurando el IDE

Run Configurations

Create, manage, and run configurations

Run a Java application



Habilitar en Eclipse

Run/Run configurations/VM arguments

Name: Excepciones

Main (x)= Arguments JRE Dependencies Source Environment Common

Program arguments:

VM arguments:
-ea

Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching
 Use @argfile when launching

Working directory:
 Default: \${workspace_loc:Semana03}
 Other: _____

Workspace... File System... Variables...

Show Command Line Revert Apply

Run Close

?

Filter matched 50 of 54 items

Ejemplo de aserciones

```
class EjemploAserciones {
    public static void main (String[] args) {
        int numero= Integer.parseInt (args[0]);
        assert numero==10 : "Problema con "+numero;
        System.out.println ("numero: "+numero);
        numero++;
    }
}

>java EjemploAserciones
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
    at EjemploAserciones.main(EjemploAserciones.java:3)

>java EjemploAserciones 9
numero: 9

>java EjemploAserciones 10
numero: 10
    ↓
>java -ea EjemploAserciones
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
    at EjemploAserciones.main(EjemploAserciones.java:3)

>java -ea EjemploAserciones 9
Exception in thread "main" java.lang.AssertionError: Problema con 9
    at EjemploAserciones.main(EjemploAserciones.java:4)
    ↙
>java -ea EjemploAserciones 10
numero: 10
```

Expresiones lambda

- También llamadas funciones lambda, función literal o función anónima, son funciones que no tienen nombre.
- Definidas y usadas en un punto y solo un punto del código.
- Se requieren 3 elementos:
 - Parámetros de entrada
 - ->
 - Expresión de salida o bloque (con `return` dentro si es necesario)

Tipo	Ejemplo
Parámetro sin paréntesis, una expresión	<code>x -> 2*x;</code>
Parámetro entre paréntesis, una expresión	<code>(x) -> x*x;</code>
Bloque de código con valor de regreso	<code>(x) -> {int resultado=0; resultado = 3 * x; return (resultado);};</code>

Ejemplo de expresiones lambda

```
public interface FuncionEntera {  
    //De la forma y=f(x), donde y es int  
    public int f(int x);  
}  
  
class Graficadora {  
    //Solo implementa la tabulacion  
    public void tabula (int xInicio, int xFin, FuncionEntera funcion){  
        System.out.println ("\n x\t\tf(x)");  
        for (int x=xInicio; x<=xFin; x++) {  
            System.out.println (x+"\t\t"+funcion.f(x));  
        }  
    }  
}
```

Ejecución de ejemplo de expresiones lambda

```
class EjemploLambda {  
    //Define varias funciones con expresiones lambda  
    public static FuncionEntera doble = x -> 2*x;  
    public static FuncionEntera cuadrado = (x) -> x*x;  
    public static FuncionEntera triple = (x) -> {int resultado=0;  
        resultado = 3 * x;  
        return (resultado);};  
  
    public static void main (String[] args) {  
        Graficadora g = new Graficadora();  
        g.tabula (10,15,doble);  
        g.tabula (-3,+3,cuadrado);  
        g.tabula (0,5,triple);  
    }  
}
```

x	f(x)
10	20
11	22
12	24
13	26
14	28
15	30
x	f(x)
-3	9
-2	4
-1	1
0	0
1	1
2	4
3	9
x	f(x)
0	0
1	3
2	6
3	9
4	12
5	15

Java Date/Time API (java.time)

```
import java.time.*;  
import java.time.format.*;  
public class EjemploDateTime {  
    public static void main(String[] args) {  
        LocalDate fecha1 = LocalDate.now(); //hoy  
        LocalDate fecha2 = LocalDate.of(1945,9,4); //año,mes,día  
        LocalDate fecha3 = LocalDate.parse("1935-12-01"); //Con texto (aaaa-mm-dd)  
  
        System.out.println ("Fecha actual"+ fecha1);  
        System.out.println ("Fecha 2: "+ fecha2);  
        System.out.println ("Fecha 3: "+ fecha3);  
  
        //Año, Mes, Día por separado  
        System.out.println ("Año:" + fecha2.getYear());  
        System.out.println ("Mes:" + fecha2.getMonthValue());  
        System.out.println ("Día (del mes):" + fecha2.getDayOfMonth());  
  
        //Formateo de fechas  
        DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy");  
        //Creación de fecha usando String con cierto formato  
        LocalDate fecha4 = LocalDate.parse("01/02/2003", formato);  
        System.out.println ("Fecha 4: "+ fecha4);  
  
        // Formateo de fecha con estructura diferente a aaaa-mm-dd  
        System.out.println ("Fecha formateada: " + fecha4.format(formato));  
    }  
}
```

Fecha actual 2021-09-27
Fecha 2: 1945-09-04
Fecha 3: 1935-12-01
Año: 1945
Mes: 9
Día (del mes): 4
Fecha 4: 2003-02-01
Fecha formateada: 01/02/2003

Hora

```
Hora actual22:54:15.532282100
Hora 2: 01:02:03
Hora 3: 10:15:30
Hora:1
Minuto:2
Segundo:3
Hora 4: 10:11
Hora formateada: 10:11
```

```
import java.time.*;
import java.time.format.*;
public class EjemploDateTime {
    public static void main(String[] args) {
        LocalTime hora1 = LocalTime.now();
        LocalTime hora2 = LocalTime.of(1,2,3); //h,r,s
        LocalTime hora3 = LocalTime.parse("10:15:30"); //Con texto()

        System.out.println ("Hora actual"+ hora1);
        System.out.println ("Hora 2: "+ hora2);
        System.out.println ("Hora 3: "+ hora3);

        //Hora, Min, Seg por separado
        System.out.println ("Hora:" + hora2.getHour());
        System.out.println ("Minuto:" + hora2.getMinute());
        System.out.println ("Segundo:" + hora2.getSecond());

        //Formateo de fechas
        DateTimeFormatter formato = DateTimeFormatter.ofPattern("HH:mm");
        //Creación de fecha usando String con cierto formato
        LocalTime hora4 = LocalTime.parse("10:11", formato);
        System.out.println ("Hora 4: "+ hora4);

        // Formateo de fecha con estructura diferente a HH:mm
        System.out.println ("Hora formateada: " + hora4.format(formato));
    }
}
```

LocalDate

```
Date fechaD = new Date();  
LocalDate fechaLD1 = LocalDate.now(); //Día actual  
LocalDate fechaLD2 = LocalDate.parse("2000-01-01"); //formato ISO  
LocalDate fechaLD3 = LocalDate.of(2000,12,31);  
  
System.out.println ("Hoy (Date):\t\t" +fechaD);  
System.out.println ("Hoy (LocalDate):\t" + fechaLD1);  
System.out.println ("1/Ene/2000 (LocalDate):\t" + fechaLD2);  
System.out.println ("31/Dic/2000 (LocalDate):" + fechaLD3);
```

Hoy (Date):	Thu May 26 19:22:57 CDT 2022
Hoy (LocalDate):	2022-05-26
1/Ene/2000 (LocalDate):	2000-01-01
31/Dic/2000 (LocalDate):	2000-12-31

Diferencia entre fechas

```
System.out.println("Ejemplo de horas con LocalTime");
LocalTime hora1 = LocalTime.parse("07:15:34");
LocalTime hora2 = LocalTime.now();
System.out.println(hora1+"--"+hora2);
System.out.println("Diferencia entre las horas:");
System.out.println("\tEn horas:" +ChronoUnit.HOURS.between (hora1, hora2));
System.out.println("\tEn minutos:" +ChronoUnit.MINUTES.between (hora1, hora2));
System.out.println(hora2);
System.out.println("\t+ 25 horas:" +hora2.plus(25, ChronoUnit.HOURS));
System.out.println("\t- 2 minutos:" +hora2.minus(2, ChronoUnit.MINUTES));
```

```
System.out.println("Ejemplo de fecha-hora con LocalDateTime");
LocalDateTime fechaHora1 = LocalDateTime.now();
LocalDateTime fechaHora2 = LocalDateTime.parse("2000-02-28T05:04:02");
System.out.println(fechaHora1+"--"+fechaHora2);
System.out.println(fechaHora1.getHour());

System.out.println("Diferencia entre "+fechaHora1 + " y "+fechaHora2);
System.out.println("\tEn años:" +ChronoUnit.YEARS.between(fechaHora1,fechaHora2));
System.out.println("\tEn meses:" +ChronoUnit.MONTHS.between(fechaHora1,fechaHora2));
System.out.println(fechaHora1);
System.out.println("\t+ 5 meses: " + fechaHora1.plus(5, ChronoUnit.MONTHS));
System.out.println("\t- 5 horas: " + fechaHora1.minus(5, ChronoUnit.HOURS));
```

Genéricos

- Forma que tiene Java de hacer clases, interfaces o métodos que generalicen una solución para diferentes tipos de objetos (clases) de entrada.
- Similar a las plantillas de C++.

Ejemplo de Genéricos

```
public class Generico<T> {//T es la clase genérica
    T atributo;

    public Generico () {
    }

    public Generico (T valorInicial){
        this.atributo = valorInicial;
    }

    public void imprime (T objeto1, T objeto2) {
        System.out.println ("\t"+objeto1+"\t\t"+objeto2);
    }
}
```

```
public class GenericoNumerico<T extends Number> {
    public double suma (T objeto1, T objeto2) {
        double resultado = 0;
        resultado = objeto1.doubleValue();
        resultado += objeto2.doubleValue();
        return (resultado);
    }

    public double sumatoria (T[] numeros) {
        double acumulado=0;
        for (T elemento : numeros)
            acumulado += elemento.doubleValue();
        return (acumulado);
    }
}
```

Ejemplo de Genéricos. Continuación

```
public class EjemploGenericos {  
    public static void main (String[] args) {  
        System.out.println ("Ejemplo de Genéricos de objetos");  
        //Creación de genéricos de objetos e impresión  
        Generico<String> generico1 = new Generico<String>();  
        Generico<Personaje> generico2 = new Generico<Personaje>();  
        //Uso de método en clase genérica  
        generico1.imprime("Magdalena Contreras", "Miguel Hidalgo");  
        generico2.imprime(new Personaje(), new Personaje("Batman"));  
  
        System.out.println ("\n\nEjemplo de números Genéricos ");  
        //Creación de genérico solo de objetos numéricos  
        GenericoNumerico<Integer> genericoNE = new GenericoNumerico<Integer>();  
        GenericoNumerico<Double> genericoNR = new GenericoNumerico<Double> ();  
        Integer[] enteros = {10, 20, 30};  
        Double [] reales = {1.1, 2.2, 3.3 };  
  
        System.out.println ("Sumatoria de enteros: " + genericoNE.sumatoria (enteros));  
        System.out.println ("Sumatoria de reales: " + genericoNR.sumatoria (reales));  
    }  
}
```

Ejemplo de Genéricos de objetos
Magdalena Contreras
Sin nombre(100.0)

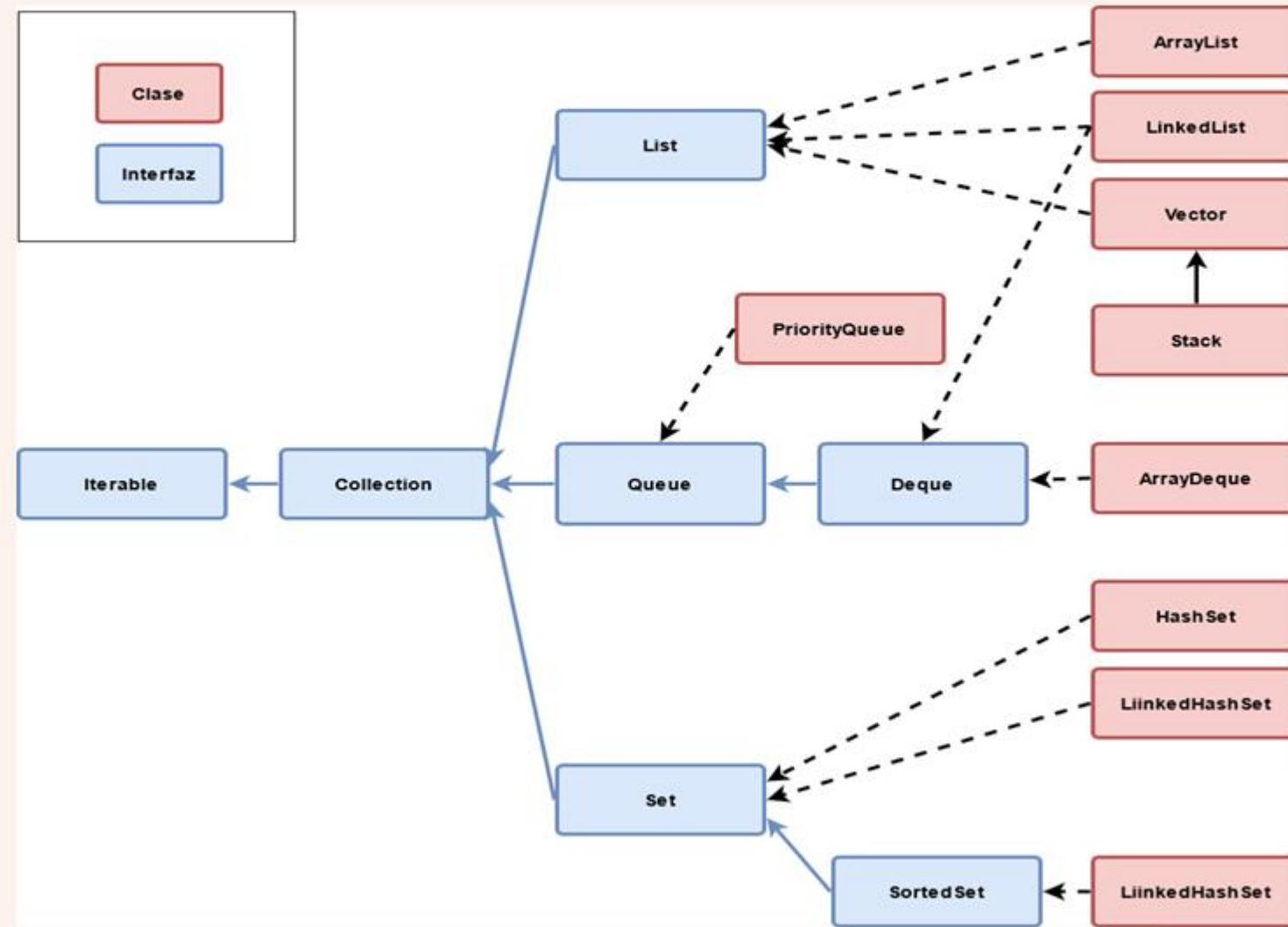
Ejemplo de números Genéricos
Sumatoria de enteros: 60.0
Sumatoria de reales: 6.6

Miguel Hidalgo
Batman(100.0)

Colecciones

- **Clases** (*ArrayList*, *Vector*, *LinkedList*, *PriorityQueue*, *HashSet*, *LinkedHashSet*, *TreeSet*) e **interfaces** (*Set*, *List*, *Queue*, *Deque*) que permiten **agrupar** objetos para su almacenamiento y extracción de diversas maneras.
- Implementadas en ***java.util***
- Framework:
 - Interfaces
 - Implementaciones (Clases)
 - Algoritmos comunes (búsqueda, ordenamiento, etc.)

Jerarquía de colecciones



Métodos más comunes de Iterable

Método *	Descripción
iterator ()	Regresa un iterador (objeto) de la colección. El iterador tiene algunos métodos para recorrer (en un sentido) como: <ul style="list-style-type: none">• hasNext()• next()• remove()
forEach ((elemento) -> { acciones })	Realiza las acciones a cada elemento del Iterable

* Opcionales dependiendo de la implementación

Métodos más comunes de Collection

Método *	Descripción
add (objeto)	Añade el objeto a la colección
clear ()	Vacia la colección
contains (objeto)	Regresa verdadero si objeto está en la colección
isEmpty ()	Regresa true si la colección está vacía
remove (objeto)	Elimina objeto de la colección
size ()	Regresa número de elementos existentes
toArray ()	Regresa un arreglo con los elementos

* Opcionales dependiendo de la implementación

Métodos más comunes de List

Método *	Descripción
add (índice, elemento)	Añade el elemento después del <i>índice</i>
get (índice)	Regresa el elemento con dicho <i>índice</i>
indexOf (elemento)	Regresa primera posición de <i>elemento</i>
lastIndexOf (elemento)	Última posición de elemento
listIterator ()	Regresa un iterador de lista (objeto) de la lista. El iterador tiene algunos métodos (para recorrer en cualquier sentido y modificar) como: <ul style="list-style-type: none">• add(), hasNext(), hasPrevious(), next(), nextIndex(), previous(), previousIndex(), remove(), set()
set (índice, elemento)	Cambia el valor en el índice con elemento
subList(inicio, fin)	Regresa una lista con los elementos de [inicio, fin] de la lista original

* Opcionales dependiendo de la implementación

ArrayList

```
import java.util.*; //Necesario para usar colecciones

public class ColeccionesArrayList {
    public static void imprime (String mensaje, Collection<String> colección) {
        //Recibe como parámetro un ArrayList de Strings
        System.out.print (mensaje);
        for (String elemento : colección){ //for tradicional
            System.out.print ("\t" + elemento); };
        System.out.println();
        //Implementación de Iterable
        // forEach ((elemento) -> { acciones })
        System.out.print ("forEach: ");
        colección.forEach ((sobrino) -> {
            System.out.print ("\t" + sobrino.toUpperCase()); });
        System.out.println ();
        // iterator()
        System.out.print ("iterator : ");
        Iterator<String> cursor = colección.iterator();
        while (cursor.hasNext()) { //hasNext() regresa true si hay elementos
            System.out.print ("\t" + cursor.next().toLowerCase()); //next() regresa el sig. elem
        }
        System.out.println();
    };
}
```

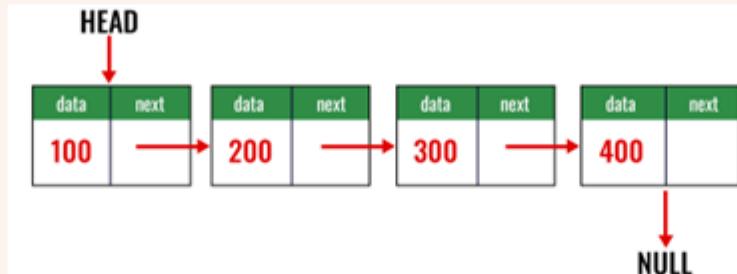
ArrayList. Continuación

```
public static void main (String[] args) {
    ArrayList<String> sobrinos = new ArrayList<>(); //Creación de un ArrayList de String's
    System.out.println (sobrinos);
    //Implementación de Collection
    sobrinos.add ("Hugo"); //add añade un elemento a la colección
    sobrinos.add ("Paco");
    sobrinos.add ("Paco"); //permite duplicados
    imprime ("add:", sobrinos);

    System.out.println(sobrinos.isEmpty()+"\t"+sobrinos.size()+"\t" + sobrinos.contains("Hugo"));
    sobrinos.remove ("Paco"); //remove() elimina el elemento enviado
    imprime ("remove:", sobrinos);
    //Implementación de List
    sobrinos.add (0, "Luis"); //Añade en una posición específica
    imprime ("add index:", sobrinos);
    System.out.println ("Posición de Luis: " + sobrinos.indexOf ("Luis"));
    System.out.println ("Elemento 0: " + sobrinos.get(0));
    //Ordenamiento
    Collections.sort (sobrinos);
    imprime ("sort:", sobrinos);

    sobrinos.clear();
    imprime ("clear:", sobrinos);
}
```

Miembros más comunes de LinkedList (lista enlazada)



Miembro *	Descripción
<code>addFirst(elemento)</code>	Añade un elemento al inicio de la lista
<code>addLast(elemento)</code>	Añade un elemento al final de la lista. Equivalente a <code>add(elemento)</code>
<code>getFirst()</code>	Regresa el primer elemento de la lista
<code>getLast()</code>	Regresa el último elemento de la lista
<code>push(elemento)</code>	Añade un elemento al final de la lista. Equivalente a <code>add(elemento)</code> y a <code>addLast(elemento)</code>
<code>pop()</code>	Regresa y elimina el elemento primer elemento de la lista
<code>peek(), peekFirst(), peekLast()</code>	Regresa, pero no elimina el elemento de la lista

* Métodos opcionales dependiendo de la implementación

Ejemplo de LinkedList

```
import java.util.*; //Necesario para usar colecciones

public class ColeccionesLinkedList {
    public static void imprime (String mensaje, Collection<String> colección) {
        //Recibe como parámetro un LinkedList de Strings
        System.out.print (mensaje);
        colección.forEach ((elemento) -> {
            System.out.print ("\t" + elemento);      });
        System.out.println ();
    }

    public static void main (String[] args) {
        LinkedList<String> beatles = new LinkedList<>(); //Creación de un LinkedList de String's
        System.out.println (beatles);

        //Implementación de Collection
        beatles.add ("John"); //add añade un elemento a la colección
        beatles.add ("Paul");
        beatles.add ("Paul"); //permite duplicados
        imprime ("Lista:", beatles);

        System.out.println(beatles.isEmpty()+"\t"+beatles.size()+"\t"+beatles.contains("Ringo"));
        beatles.remove ("Paul"); //remove() elimina el elemento enviado
        imprime ("remove:", beatles);

        //Implementación de List
        System.out.println ("Posición de John: " + beatles.indexOf ("John"));
    }
}
```

Ejemplo de LinkedList. Continuación

```
//Implementación de LinkedList
beatles.addFirst ("Ringo");
beatles.addLast ("George");
imprime ("add index:", beatles);
System.out.println ("Elementos 2:( " + beatles.get(2) +
    "), Primero:" +beatles.getFirst() + ", Último:" +beatles.getLast() + ")");
System.out.println ("Elemento eliminado: " + beatles.pop()); //Elimina el primer elemento

beatles.push("John Lennon"); //Agrega un elemento al final de la lista
beatles.push("Paul Mcartney"); //Agrega un elemento al final de la lista
imprime ("Push", beatles);
System.out.println ("Elemento removido: " + beatles.remove()); //Elimina el primer elemento

//Ordenamiento
Collections.sort (beatles);
imprime ("sort:", beatles);

beatles.clear();
imprime ("clear:", beatles);
}

}
}

List: John      Paul      Paul
false   3          false
remove: John      Paul
Posición de John: 0
add index:      Ringo     John     Paul     George
Elementos 2:(Paul), Primero:(Ringo), Último:George
Elemento eliminado: Ringo
Pop      John      Paul      George
Push      Paul Mcartney  John Lennon  John      Paul      George
Elemento removido: Paul Mcartney
sort:    George    John      John Lennon  Paul
clear:
```

Miembros más comunes de Vector

Miembro *	Descripción
Vector (tamInicial, crecimiento)	Especifica tamaño inicial y número de crecimiento (0 duplica el tamaño).
capacity ()	Tamaño del Vector.
add (índice, elemento) insertElementAt (elemento, índice)	Agrega, en la posición existente –índice- el elemento
addElement (elemento)	Agrega al final el elemento
set (índice, elemento)	Cambia el valor del índice con el elemento
get (índice) elementAt (índice)	Regresa el elemento en la posición índice
trimToSize()	Ajusta el tamaño al número de elementos actual
removelf (elem -> (exprBool))	Elimina los elementos cuya exprBool sea true

* Métodos opcionales dependiendo de la implementación

Ejemplo de Vectores

```
import java.util.*; //Necesario para usar colecciones

public class ColeccionesVector {
    public static void imprime (String mensaje, Vector<String> colección) {
        //Recibe como parámetro un LinkedList de Strings
        System.out.print (mensaje+"("+colección.capacity()+" ) ");
        colección.forEach ((elemento) -> {
            System.out.print ("\t" + elemento);//+"("+elemento.compareTo ("G")+")");
        });
        System.out.println ();
    }

    public static void main (String[] args) {
        Vector<String> elementos = new Vector<>(3,4); //Creación de un Vector de String's
                                                        //Tamaño inicial de 3 y crecimiento de 4
        System.out.println (elementos+": "+elementos.capacity());
        //Implementación de Collection
        elementos.add ("Hidrógeno"); //add añade un elemento a la colección
        elementos.add ("Litio");
        elementos.add ("Sodio"); //permite duplicados
        elementos.add ("Sodio"); //permite duplicados
        elementos.add ("Sodio"); //permite duplicados
        imprime ("Vector:", elementos);

        System.out.println (elementos.isEmpty() + "\t" +
                           elementos.size() + "\t" + elementos.contains ("Oxígeno"));
        elementos.remove ("Sodio"); //remove() elimina el elemento enviado
    }
}
```

Ejemplo de Vectores. Continuación

```
//Implementación de List
    //imprime ("add index:", elementos);
    System.out.println ("Posición de Litio: " + elementos.indexOf ("Litio"));

//Implementación de Vector
    elementos.add (4,"Cesio"); //Agrega en la posición ya existente
    elementos.addElement ("Francio"); //Agrega al final e incrementa tamaño
    elementos.addElement ("Ununnenio"); //Agrega al final e incrementa tamaño
    imprime ("adds:", elementos);
    elementos.insertElementAt ("Rubidio",4); //Agrega al final e incrementa tamaño
    imprime ("insert:", elementos);
    elementos.set (3, "Potasio");
    elementos.trimToSize();
    imprime ("set y trim:", elementos);

    System.out.println ("Elementos 2:( " + elementos.elementAt(2) +
        "), Primero:( " + elementos.firstElement() + "), Último:( " + elementos.lastElement() + " )");
    elementos.removeIf (elemento -> (elemento.compareTo ("H") < 0)); //Mayores a H
    //string.compareTo(otro) regresa <0 cuando s<otro, >0 cuando s>otro, 0 cuando ==
    imprime ("removeIf:", elementos);
//Crecimiento
    elementos.addElement ("Berilio");
    elementos.addElement ("Magnesio");
    imprime ("crecim:", elementos);
//Ordenamiento
    Collections.sort (elementos);
    imprime ("sort:", elementos);
    elementos.clear();
    imprime ("clear:", elementos);
}
}
```

Miembros más comunes de Stack (LIFO)

Miembro *	Descripción
empty ()	Regresa true si la pila está vacía
peek ()	Regresa el elemento encima de la pila, sin eliminarlo
pop ()	Regresa el elemento encima de la pila, eliminándolo de la pila
push (elemento)	Añade el elemento arriba en la pila
search (elemento)	Regresa la posición del elemento buscado

* Métodos opcionales dependiendo de la implementación

Ejemplo de Stack

```
public static void main (String[] args) {  
    Stack<String> elementos = new Stack<> (); //Creación de un Stack de String's  
    //Cuando se requiere crecimiento se duplica el tamaño del stack  
    System.out.println (elementos+": "+elementos.capacity());  
  
    //Implementación de Collection  
    elementos.add ("home"); //add añade un elemento a la colección  
    elementos.add ("google");  
    elementos.add ("gmail"); //permite duplicados  
    elementos.add ("gmail"); //permite duplicados  
    elementos.add ("youtube");  
    imprime (mensaje: "Stack:", elementos);  
  
    System.out.println (elementos.isEmpty() + "\t" +  
        elementos.size() + "\t" + elementos.contains ("gmail"));  
    elementos.remove (o: "gmail"); //remove() elimina el elemento enviado  
    imprime (mensaje: "remove:", elementos);  
  
    //Implementación de List  
    //imprime ("add index:", elementos);  
    System.out.println ("Posición de gmail: " + elementos.indexOf ("gmail"));
```

```
[:]: 10  
Stack:(10) home google gmail gmail youtube  
false 5 true  
remove:(10) home google gmail youtube  
Posición de gmail: 2
```

```
//Implementación de Vector
elementos.add ( index: 4, element: "unam"); //Agrega en la posición ya existente
elementos.addElement ( obj: "UNAM"); //Agrega al final e incrementa tamaño
elementos.addElement ( obj: "google"); //Agrega al final e incrementa tamaño
imprime ( mensaje: "adds:", elementos);
elementos.insertElementAt ( obj: "hotmail", index: 4); //Agrega al final e incrementa tamaño
imprime ( mensaje: "insert:", elementos);
elementos.set (3, "twitter");
elementos.trimToSize();
imprime ( mensaje: "set y trim:", elementos);

System.out.println ("Elementos 2:( " + elementos.elementAt( index: 2) +
    ", Primero:( "+elementos.firstElement()+" ), Último:( "+elementos.lastElement()+" )"
elementos.removeIf (elemento -> (elemento.compareTo ("j") > 0)); //Mayores a d
//string.compareTo(otro) regresa <0 cuando s<otro, >0 cuando s>otro, 0 cuando ==
imprime ( mensaje: "removeIf:", elementos);

//Implementación de Stack
Stack<String> nombre = new Stack<> (); //Creación de un Stack de String's
//Cuando se requiere crecimiento se duplica el tamaño del stack
System.out.println ("\nAcciones propias de Stack");
nombre.push ( item: "Carlos"); //Equivalente a addElement ("Universidad"); (al final)
nombre.push ( item: "Eligio");
nombre.push ( item: "Ortiz");
imprime ( mensaje: "push :", nombre);

System.out.println ("Elemento eliminado: " + nombre.pop() );
imprime ( mensaje: "pop :", nombre);

System.out.println ("Elemento extraido: " + nombre.peek() );
imprime ( mensaje: "peek :", nombre);
```

```
adds:(10) home google gmail youtube unam UNAM google
insert:(10) home google gmail youtube hotmail unam UNAM google
set y trim:(8) home google gmail twitter hotmail unam UNAM google
Elementos 2:(gmail), Primero:(home), Último:(google)
removeIf:(8) home google gmail hotmail UNAM google
Acciones propias de Stack
push :(10) Carlos Eligio Ortiz
Elemento eliminado: Ortiz
pop :(10) Carlos Eligio
Elemento extraido: Eligio
peek :(10) Carlos Eligio
sort:(10) Carlos Eligio
¿Dónde está Eligio? 1
¿Dónde está Maldonado? -1
clear:(0)
¿Vacio? true
```

HashMap (Llave-valor)

```
import java.util.HashMap;
public class Mapas {
    public static void main(String[] args) {
        // Creación de un mapa, con un par (Llave-Valor)
        HashMap<Integer, String> meses = new HashMap<Integer, String>();
        meses.put(1, "Enero"); // Agrega un nuevo par Llave:Valor
        meses.put(5, "May");
        meses.put(3, "Mar");
        System.out.println(meses+ ". " +meses.size()+" elementos.");

        // Extraer conociendo la llave
        System.out.println(meses.get(5)); // get(llave) regresa el valor
        System.out.println(meses.get(15)); // Si no existe, regresa null

        // Saber si una llave o valor existen en el HashMap
        System.out.println(meses.containsKey(5)); // ¿Existe una llave?
        System.out.println(meses.containsValue("Marzo")); // ¿Existe

        meses.put(1, "Ene"); // Cambia un Valor
        meses.put(2, "Feb"); // Agrega un nuevo par Llave:Valor
        System.out.println(meses+ ". " +meses.size()+" elementos.")
    } // Fin del main()
} //Fin de la clase
```

{1=Enero, 3=Mar, 5=May}. 3 elementos.
May
null
true
false
{1=Ene, 2=Feb, 3=Mar, 5=May}. 4 elementos.

Conjuntos con HashSet

```
import java.util.*; // Para HashSet y Arrays
public class EjemploHS {
    public static void main(String[] args) {
        System.out.println("Ejemplo de Conjunto con HashSet");
        HashSet<String> conjunto1 = new HashSet<String>(Arrays.asList ("c","a","r","l","o","s"));
        System.out.println("Conjunto 1: "+conjunto1);

        HashSet<String> conjunto2 = new HashSet<String>(Arrays.asList ("e","l","i","g","i","o"));
        System.out.println("Conjunto 2: "+conjunto2);      // No permite duplicados --^

        HashSet<String> union = new HashSet<String>(conjunto1.stream().toList()); // Copia
        union.addAll(conjunto2);
        System.out.println("Unión "+union);

        HashSet<String> interseccion = new HashSet<String>(conjunto1.stream().toList()); // Copia
        interseccion.retainAll(conjunto2);
        System.out.println("Intersección "+interseccion);

        HashSet<String> diferencia = new HashSet<String>(conjunto1.stream().toList()); // Copia
        diferencia.removeAll(conjunto2);
        System.out.println("Diferencia 1-2 "+diferencia);

        diferencia = new HashSet<String>(conjunto2.stream().toList()); // Copia
        diferencia.removeAll(conjunto1);
        System.out.println("Diferencia 2-1 "+diferencia);
    }
}
```

```
Ejemplo de Conjunto con HashSet
Conjunto 1: [a, r, c, s, l, o]
Conjunto 2: [e, g, i, l, o]
Unión [a, r, c, s, e, g, i, l, o]
Intersección [l, o]
Diferencia 1-2 [a, r, c, s]
Diferencia 2-1 [e, g, i]
```

Ejemplo de PriorityQueue

```
[]  
Colección:      A      D      G      U      M      N  
false  6  false  
peek:  A  [A, D, G, U, M, N]  
element:A  [A, D, G, U, M, N]  
Extraen y eliminan  
remove: A  [D, M, G, U, N]  
poll:  D  [G, M, N, U]  
remove: G  [M, U, N]  
remove U: true  [M, N]  
clear:
```

```
//Ejemplo de PriorityQueue  
import java.util.*; //Necesario para usar colecciones  
public class ColeccionesPriorityQueue {  
    public static void imprime (String mensaje, Collection<String> colección) {  
        //Recibe como parámetro un LinkedList de Strings  
        System.out.print (mensaje);  
        colección.forEach ((elemento) -> {  
            System.out.print ("\t" + elemento);      });  
        System.out.println ();  
    };  
    public static void main (String[] args) {  
        //Creación de un PriorityQueue de String's  
        PriorityQueue<String> letras = new PriorityQueue<String>();  
        System.out.println (letras);  
        //Implementación de Collection  
        letras.add ("U"); //add añade un elemento a la colección  
        letras.add ("N");  
        letras.add ("A");  
        letras.add ("M");  
        letras.add ("D");  
        letras.add ("G");  
        imprime ("Colección:", letras);  
        System.out.println(letras.isEmpty()+"\t"+letras.size()+"\t"+letras.contains("R"));  
  
        // Implementación de Queue  
        // Los elementos se retiran por "prioridad" (en orden)  
        // No remueven  
        System.out.println ("peek: "+letras.peek()+"\t"+letras); // null si está vacía  
        System.out.println ("element:"+letras.element()+"\t"+letras);  
        //Remueven  
        System.out.println ("Extraen y eliminan");  
        System.out.println ("remove: "+letras.remove()+"\t"+letras);  
        System.out.println ("poll: "+letras.poll()+"\t"+letras); // null si está vacía  
        System.out.println ("remove: "+letras.remove)+"\t"+letras);  
        System.out.println ("remove U: "+letras.remove("U")+"\t"+letras);  
        // clear()  
        letras.clear();  
        imprime ("clear:", letras);  
    }  
}
```

Enumeraciones

- Lista de valores fijos. Regularmente para valores discretos

```
enum Suscripcion {GRATUITO, BRONCE, PLATA, ORO, PLATINO}; |  
  
public class Enumeraciones {  
    public static void main(String[] args) {  
        Suscripcion nivel = Suscripcion.GRATUITO;  
        System.out.println ("Suscripción individual: " + nivel+"\n");  
  
        //values() regresa todos los elementos del enum  
        //ordinal() regresa la posición dentro del enum  
        Suscripcion[] niveles = Suscripcion.values();  
        for (int i=0;i<niveles.length;i++)  
            System.out.println ("Suscripción: ("+niveles[i].ordinal()+"") "+ niveles[i]);  
  
        String nivelTexto = "oro";  
        //valueOf(s) convierte a enum el valor texto de entrada  
        nivel = Suscripcion.valueOf(nivelTexto.toUpperCase());  
        System.out.println ("\nSuscripción individual: " + nivel);  
    }  
}
```

Suscripción individual: GRATUITO

Suscripción: (0) GRATUITO

Suscripción: (1) BRONCE

Suscripción: (2) PLATA

Suscripción: (3) ORO

Suscripción: (4) PLATINO

Suscripción individual: ORO

Constructores, atributos y métodos en enumeraciones

- Es posible, como si se tratara de una clase, crear atributos y métodos (incluyendo constructores) en una enumeración.

```
enum Suscripcion {  
    GRATUITO(0), BRONCE(100), PLATA(300), ORO(500), PLATINO(700);  
    //Atributos  
    private int costo;  
  
    //Métodos  
    Suscripcion (int costoxSuscripcion) {  
        this.costo = costoxSuscripcion;  
    }  
  
    public int getCosto() {  
        return (this.costo);  
    }  
};  
  
public class Enumeraciones {  
    public static void main(String[] args) {  
        Suscripcion nivel = Suscripcion.PLATA;  
        System.out.println ("Suscripción: " + nivel+ "$"+nivel.getCosto());  
  
        nivel = Suscripcion.ORO;  
        System.out.println ("Suscripción: " + nivel+ "$"+nivel.getCosto());  
    }  
}
```

Suscripción: PLATA\$300
Suscripción: ORO\$500

Anotaciones

Elementos que se incluyen en las clases para proporcionar información útil que se puede utilizar en tiempo de compilación o de ejecución, pero que no cambia el flujo del código.

Anotaciones comunes

- `@Override` (indica que un método se está sobreescribiendo)
- `@Deprecated` (el método, atributo o clase está descontinuado)
- `@SuppressWarnings` indica al compilador que no considere los mensajes warning

@Override

- Es una anotación que se usa para indicar que un método se va a sobreescribir y por tanto el compilador debe verificar que se esté sobreescribiendo un método –o sobrecarga- existente.

```
@Override
public String toString (int parametro1) {
    //Sin el @Override no genera error de compilación
    String regreso;
    regreso = "Atributo " + String.valueOf(atributo) +
    ", Otro atributo = " + String.valueOf(otroAtributo);
    return regreso;
}
```

F:\UNAM\Curso Java básico\2018\Código>javac Padre.java
Padre.java:8: error: method does not override or implement a method from a super type
 @Override
 ^
1 error

Contacto

Lic. Carlos Eligio Ortiz Maldonado

carloseligio@ortizm.com