



11^a
Emisión

**DIPLOMADO
Desarrollo de Sistemas
con Tecnología Java**

Módulo 1
Programación orientada a objetos con
Java

Carlos Eligio Ortiz Maldonado

carloseligio@ortizm.com



DGTIC

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Dirección General de Cómputo y de Tecnologías de información y Comunicación
Dirección de Docencia en TIC



Educación
Continua
1971 - 2021

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.

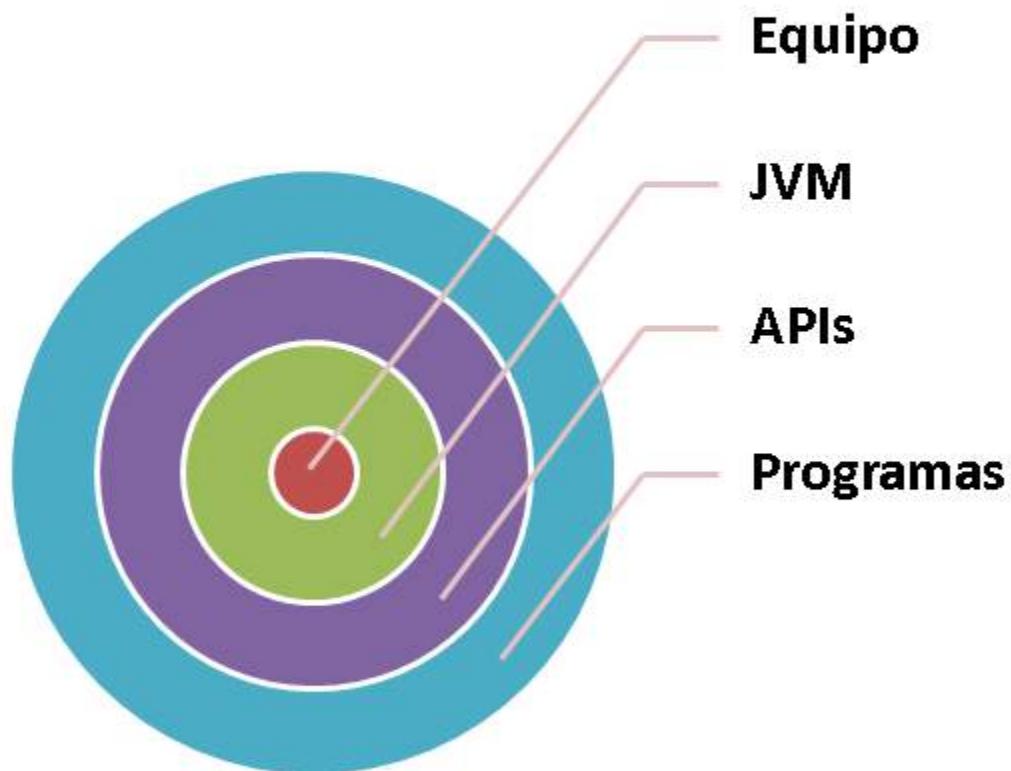
Especificaciones

Ver. 11 (LTS): <https://docs.oracle.com/javase/specs/jls/se11/html/index.html>

Ver. 18: <https://docs.oracle.com/javase/specs/jls/se18/html/index.html>

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

Java SE



Equipo

JVM

APIs

Programas

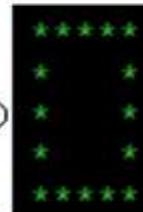
Java Development Kit (JDK)
Para desarrollo

Java Runtime Environment (JRE)
Para ejecución

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.

A. Ejercicios propuestos

1. Mostrar Hola + nombre-del-asistente. HolaNombre 
2. Desplegar un rectángulo de 5 x 4 asteriscos. Rectangulo5x4 
3. Desplegar un triángulo rectángulo de base 5 y altura 5. TrianguloR 
4. Desplegar un triángulo escaleno de altura 5. TrianguloEscaleno
5. Desplegar un menú para imprimir figuras (1-Rectángulo, 2-Triángulo rectángulo, 3-Triángulo escaleno y 9-Salir) MenuFiguras 
6. Desplegar un rectángulo, pero solo el perímetro. PerimetroRectangulo 
7. Desplegar rectángulo de asteriscos, pero usando un solo System.out.print(). Rectangulo1println
8. Reproducir el sonido de mensaje por default (carácter '\007') . Campana
9. Desplegar un árbol de navidad ArbolNavidad
10. Desplegar los datos de un equipo de futbol. Equipo

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)

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 | Iniciar con minúscula | apellidoPaterno, calificacion, volumen |
| Constante | Mayúsculas | PI, IVA, LIMITE_FALTAS |



Definición de variable

tipo nombre = valor inicial ;

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

| 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) |

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



B. Ejercicios propuestos

1. Probar si los nombres de variables propuestos son válidos o no. (Dejar como comentario las que no sean válidas).
NombresVariables
2. Definir variables (de tipo int) con nombres válidos. Imprimir los valores de cada una. Variables
3. Definir variable de nombre “class” y revisar el mensaje del compilador. VariableClass
4. Definir dos veces la misma variable y revisar el mensaje del compilador. VariableDuplicada



51 palabras reservadas en Java 15



| | | | |
|----------|------------|--------------|-----------|
| 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 | |
| open | Restringido en ciertos contextos |
| module | |
| requires | |
| transitive | |
| exports | |
| opens | |
| to | |
| uses | |
| provides | |
| with | |

Eligio Ortiz (carloselgio@ortizm.com)



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**

Clases equivalentes

| Tipo de dato primitivo | Wrapper class |
|------------------------|---------------|
| 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
```

C. Ejercicios compuestos

1. Definir —e inicializar- una variable para cada tipo de datos, desplegando al final el contenido de cada una de ellas.
TiposVariable
2. Definir —e inicializar- las variables necesarias para guardar los datos de un alumno inscrito al curso. DatosAlumno
3. Definir —e inicializar- las variables necesarias para guardar los datos de una película. DatosPelicula
4. Definir —e inicializar- las variables necesarias para guardar los datos un automóvil registrado en la CDMX.
DatosAuto
5. Definir una variable con cada una de las clases equivalentes. EjemploCE

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

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 |

D. Ejercicios propuestos

1. Capturar 3 calificaciones parciales y desplegar el promedio. **Promedio**
2. Ídem, pero recibiendo de la línea de comando las calificaciones parciales.
PromedioLineaComando
3. Intercambiar los valores de dos variables numéricas. IntercambiarVariables
4. Capturar 4 calificaciones (ejercicios, prácticas, proyecto, participación) y calcular, de acuerdo a lo mencionado al inicio de módulo, la calificación final del curso.
CalificacionFinal
5. Pedir precio de venta, cantidad de artículos y desplegar subtotal, IVA calculado y total.
Ventas
6. Pedir dos valores numéricos y desplegar: su +, -, *, / y % ¿qué pasa cuando el segundo valor es cero? OperacionesBasicas
7. Intercambiar los valores de dos variables numéricas ($a \leftrightarrow b$).
IntercambiarVariables
8. Rotar los valores de tres variables numéricas ($a \rightarrow b \rightarrow c$). Intercambiar2Variables
9. Intercambiar los valores de dos variables numéricas, **sin** utilizar una tercera.
IntercambiarVariables2

Carlos Eligio Ortiz (carloselgio@ortizm.com)



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*

E. Ejercicios propuestos

1. Recibiendo 1 valor numérico desplegar si es par o impar.
EsPar
2. Recibiendo 2 valores numéricos saber si el primero es múltiplo del segundo. EsMultiplo
3. Recibiendo 2 valores numéricos enteros, dividir el primero entre el segundo, siempre y cuando el segundo sea diferente de cero. DivisionValidada
4. Recibir la hora del día (a 24 hrs) y desplegar “Buenos días”, “Buenas tardes” o “Buenas noches” dependiendo de la hora. Saludos

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);
```

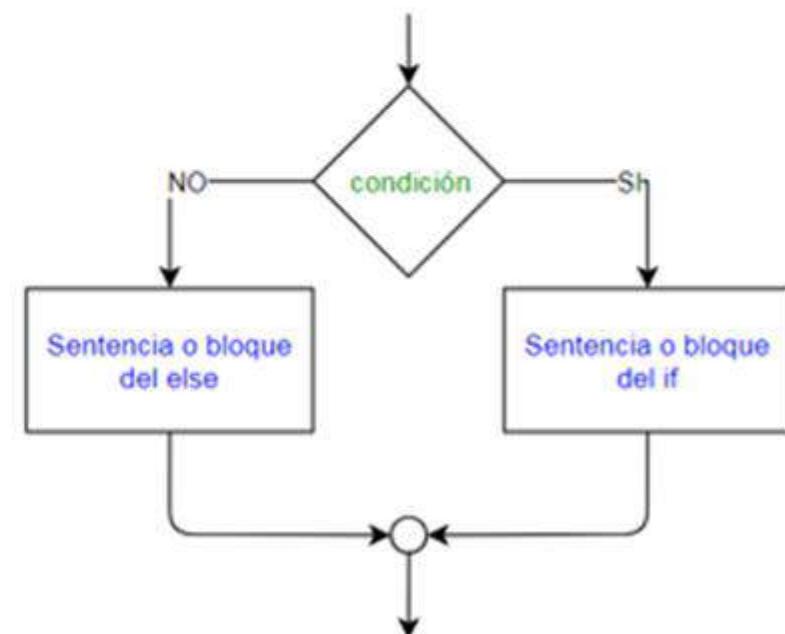
| Operators | Associativity |
|---|---------------|
| [] . () (method call) | Left to right |
| ! ~ ++ -- +(unary) -(unary) () (cast) new | Right to left |
| * / % | Left to right |
| + - | Left to right |
| << >> >>> | Left to right |
| < <= > >= instanceof | Left to right |
| == != | Left to right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

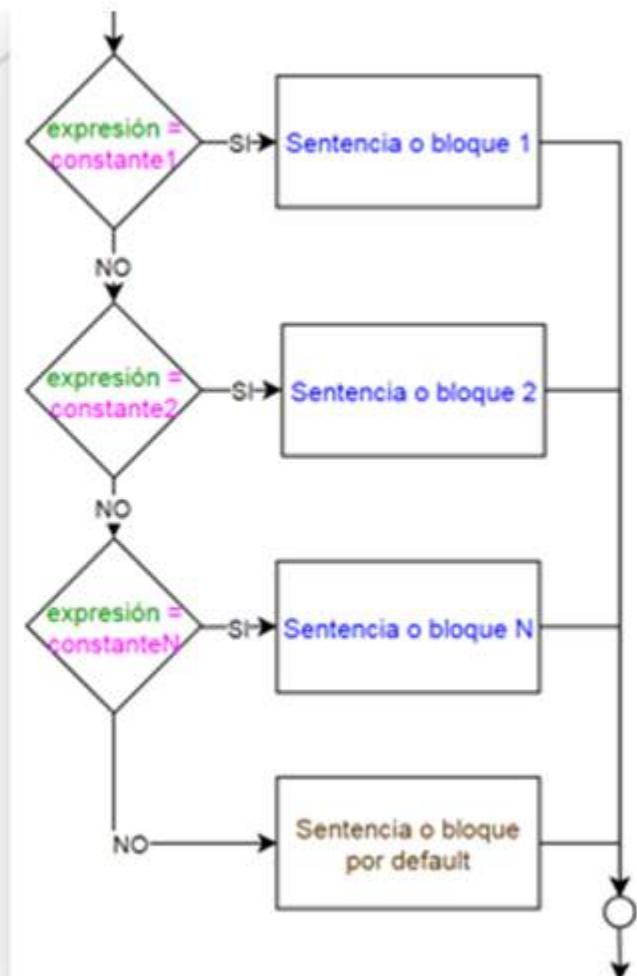
```
if (condición)
    sentencia-o-bloque
else
    sentencia-o-bloque
```



Sentencia switch

- Compara un *byte*, *short*, *int* o *char* con un valor (literal o constante) para ejecutar solo un bloque de código. Equivalente a *if's* 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;  
}
```



Carlos Eligio Ortiz (carloselgio@ortizm.com)

F. Ejercicios propuestos

1. Pedir dos valores y desplegar: su +, -, /, *. Validar que el segundo valor no sea cero. Operaciones
2. Recibir del usuario un número entero positivo e indicarle al usuario si es par o impar. EsParImpar
3. Recibir un nombre de la línea de comando, si es tu nombre, desplegar “Bienvenid@” + tuNombre; en otro caso solo decir “Hola” Bienvenido
4. Recibir de la línea de comando 3 parámetros (operando1 operación operando2) del tipo $5 + 3$, $6 - 2$, $7 * 8$ y calcular la operación solicitada
CalculadoraBasica
5. Recibir el número del mes e imprimir el nombre NombreMes
6. Ídem, pero con el día de la semana. NombreDia
7. Ídem, pero con las entidades federativas de México. NombreEntidad
8. Solicitar el num. del mes y desplegar cuántos días tiene el mes. DiasMes
9. Ejercicios 4 y 7 en uno solo. NombreDiasMes

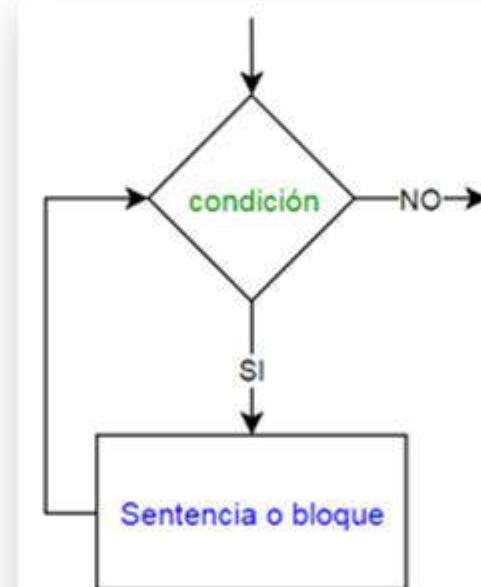
Carlos Eligio Ortiz (carloselgio@ortizm.com)



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  
}
```

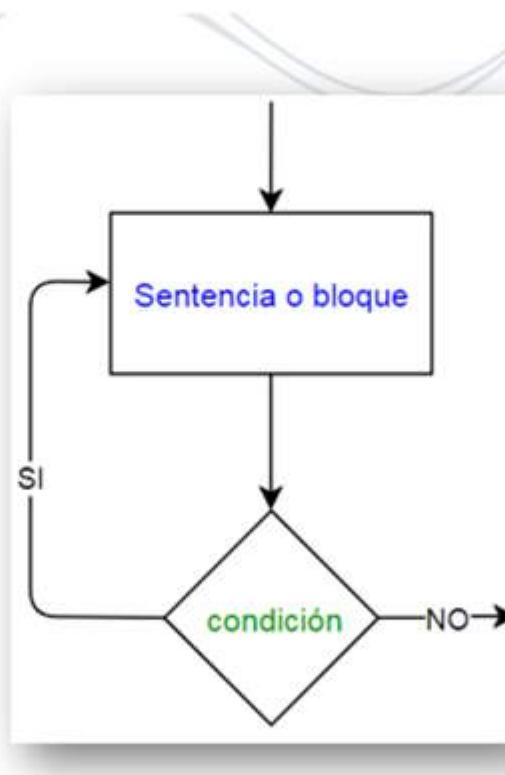


```
i=10;  
j=1;  
while ( i==10 ) {  
    System.out.println  
    j++;  
    if ( j >15 ) i=12;  
}
```

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);
```

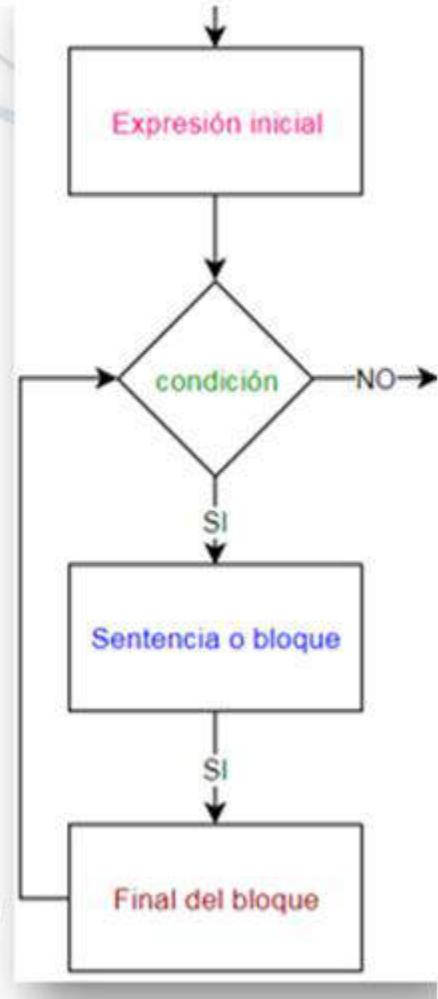
G. Ejercicios propuestos

1. Desplegar una lista de los primeros 100 números. [Primeros100](#)
2. Dado un número **n**, imprimir su tabla de multiplicar [Tabla](#)
3. Desplegar las primeras 50 decenas (10,20,30, ...) [PrimerasDecenas](#)
4. Imprimir lista (1-80) mostrando el número como entero y como carácter.
[ListaNumeroCaracter](#)
5. Retomando [MenuFiguras](#), solicite al usuario su selección, terminando al dar 9.
[MenuFiguras](#)
6. Desplegar todos los pares menores a 60. [Pares60](#)
7. Desplegar los números impares entre 1000 y 1100. [RangoImpares](#)
8. Pedir ancho y alto, e imprimir rectángulo respectivo [RectanguloWhile](#)
9. Pedir un número entero y dibujar un triángulo rectángulo de ancho y alto igual al número capturado. [TrianguloWhile](#)
10. Calcular el factorial de un número dado por el usuario. [Factorial](#)
11. Desplegar la sucesión de Fibonacci. Primeros **n** números. [Fibonacci01](#)
12. Desplegar los elementos de la sucesión de Fibonacci menor a cierto límite.
[Fibonacci02](#)

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;
```



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);  
}
```

H. Ejercicios propuestos

- Realizar un menú para desplegar un rectángulo, triángulo rectángulo o un triángulo equilátero, dependiendo de lo que seleccione el usuario. Se repetirá el ciclo hasta que el usuario escoja la opción de salir. La impresión del menú se hará por medio de una función. [MenuFiguras](#)
- Calcular el factorial de un número por medio de una función [Factorial](#)
- Calcular la constante e. [ConstanteE](#) $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$
- Calcular π con la serie de Gregory-Leibniz (1674).
Como entrada, el # de iteraciones. [Pi_GL](#) $\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots\right)$
- Calcular π con la serie de Nilakantha (s. XV).
Como entrada, el # de iteraciones [Pi_Nilakantha](#) $\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{8 \times 9 \times 10} + \dots$
- Calcular π con la serie de Wallis (1655).
Como entrada, el # de iteraciones [Pi_Wallis](#) $\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \cdot \dots = \frac{\pi}{2}$
- Determinar cuál de los tres métodos para cálculo de Pi converge más rápido.
- Hacer una función para cada operación aritmética básica (+, -, *, /, %, cuadrado, potencia). Usar cada función de acuerdo a lo que determine el usuario. [Integrar validaciones](#). [EjemploFunciones](#)
- Modificar [EsParImpar](#) y [NombreMes](#) para utilizar funciones.
- Hacer una función para pedir –con Scanner- cada uno de los tipos de datos vistos (int, char, String, etc.) recibiendo como parámetro el texto que saldrá (con `println`). [FuncionesCaptura](#)
- Calcular recursivamente la sumatoria de 1 a n. [SumatoriaRecursiva](#)
- Calcular la serie de Fibonacci utilizando recursividad [FibonacciRecursivo](#)
- Determinar cuántas veces se ejecuta la función recursiva del punto anterior para los primeros 10,20 y 30 números de la serie.

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).

Paquetes de uso común

- [java.lang](#) Default, clases equivalentes, System, Object, String, Thread, Exception.
- [java.util](#) Colecciones, Scanner, Date
- [java.desktop](#) Framework para interfaces gráficas
- [java.io](#) Clases para manejo de archivos
- [java.net](#) Clases para comunicaciones en red
- Existen paquetes de terceros para múltiples funciones:
 - [Música](#)
 - [Manejo de archivos JSON](#)
 - [Ajedrez](#)
 - Álgebra lineal (Mahout)
 - Machine Learning ([Weka](#), [Tribuo](#))

Carlos Eligio Ortiz (carloselgio@ortizm.com)



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 NombredelPaquete.*;***

```
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);
    }
} @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")
```

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



Clase `java.lang.Math`

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



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];
```



I. Ejercicios propuestos

1. Con el número de mes dado por el usuario, desplegar el nombre de mes y número de días que tiene dicho mes (que estarán almacenados en uno o más arreglos).
NombreMesArreglos
2. Capturar 5 datos enteros, guardarlos en un arreglo y luego obtener Sumatoria, Promedio, Valor mínimo, Valor máximo, Moda y Mediana.
EstadisticasConArreglos
3. Dado un arreglo de 10 enteros cualesquiera, recorrer los elementos n posiciones a la derecha en forma circular. Así, un arreglo con 1,2,3,4,5,6,7,8,9,10 quedaría, recorriéndolo 3 lugares, como 8,9,10,1,2,3,4,5,6,7. RecorrerArreglo
4. Dado un arreglo de 99 enteros, donde están todos los enteros del 1 al 100, excepto uno, ¿cómo encontrar el que falta recorriendo una sola vez el arreglo?.
5. Dado un arreglo de enteros desordenado, ordenar los datos. SortArreglo
6. Dado el arreglo del punto 3, buscar un número en particular y desplegar posición.
BusquedaArreglo



J. Ejercicios propuestos

1. Dado un arreglo bidimensional de enteros (5 filas por 3 columnas) donde cada fila son las calificaciones de un alumno, y cada columna es la calificación de cada uno de los 3 parciales desplegar

- El promedio de cada alumno
- El promedio de cada parcial

EjemploArregloBiDimensional

2. Dado un arreglo de 9x9 de enteros encontrar una configuración que represente una solución a un juego Sudoku (cualquiera). Sudo ku
3. Representar en un arreglo la configuración inicial de una partida de ajedrez.
Ajedrez
4. Realizar las operaciones básicas de matrices. OperacionesMatrices
5. Dado una matriz cuadrada de números, girar los elementos 90° . GirarMatriz

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

-->

| | | |
|---|---|---|
| 7 | 4 | 1 |
| 8 | 5 | 2 |
| 9 | 6 | 3 |

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



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
```

Sentencia class

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

//Métodos
    tipo nombreMetodo1() {

    }

    tipo nombreMetodoN() {

    }
}
```



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()
```



K. Ejercicios propuestos

- 1. Clase SerieTV con los atributos** nombre, país, temporadas, idioma y género. SerieTV.java y PruebaSerieTV.java
- 2. Clase Alumno con los atributos** nombre, edad, sexo, 4 calificaciones (ejercicios, prácticas, proyecto final y participación en clase) y promedio. Alumno.java y PruebaAlumno.java.

Métodos set y get/is

- 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;  
    }  
}
```



L. Ejercicios propuestos

1. Clase SerieTV con los atributos nombre, país, temporadas, idioma y género. Incluir métodos set y get. SerieTV.java y PruebaSerieTV.java
2. Clase Alumno con los atributos nombre, edad, sexo, 4 calificaciones (ejercicios, prácticas, proyecto final y participación en clase) y promedio. Incluir métodos set y get (en los métodos get de las calificaciones calcular el promedio). Alumno.java y PruebaAlumno.java.



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 nuevas clases a partir de las ya existentes.
- 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
```



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 super 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 ();  
}
```



Destrucción de objetos

- Automática: con el garbage collector.
- Manual:
 - Con el método `finalize()` (obsoleto desde la versión 9)
 - Implementando la interfaz `AutoCloseable` y su método `close()`

```
class Reloj
    implements AutoCloseable {
    public Reloj () {
        //Constructor que reserva recurso
        System.out.println ("... constructor ...");
    }

    public void close() {
        //Método close() (podría usarse cualquier otro)
        System.out.println ("... liberando recursos ...");
    }
}
```

```
1 class PruebaDestruccion {
2     public static void main (String[] args) {
3         Reloj uno = new Reloj ();
4         uno.close();
5     }
6 }
```

... constructor ...
... destructor ...

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**.

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

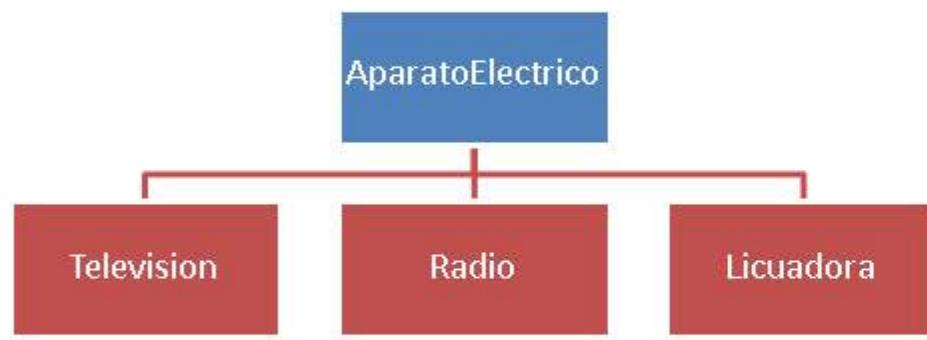
Polimorfismo

- Habilidad para que un objeto se comporte de diferentes maneras.
- Permite ejecutar métodos de acuerdo al 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

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

Interfaces

- Una interfaz define un comportamiento común a las clases que la implementen, es un concepto abstracto.
- Los métodos son *abstract* (en ese sentido se parecen a las clases abstractas) aunque se puede incluir métodos *default*.
- Los atributos definidos sólo pueden ser *static* o *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 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

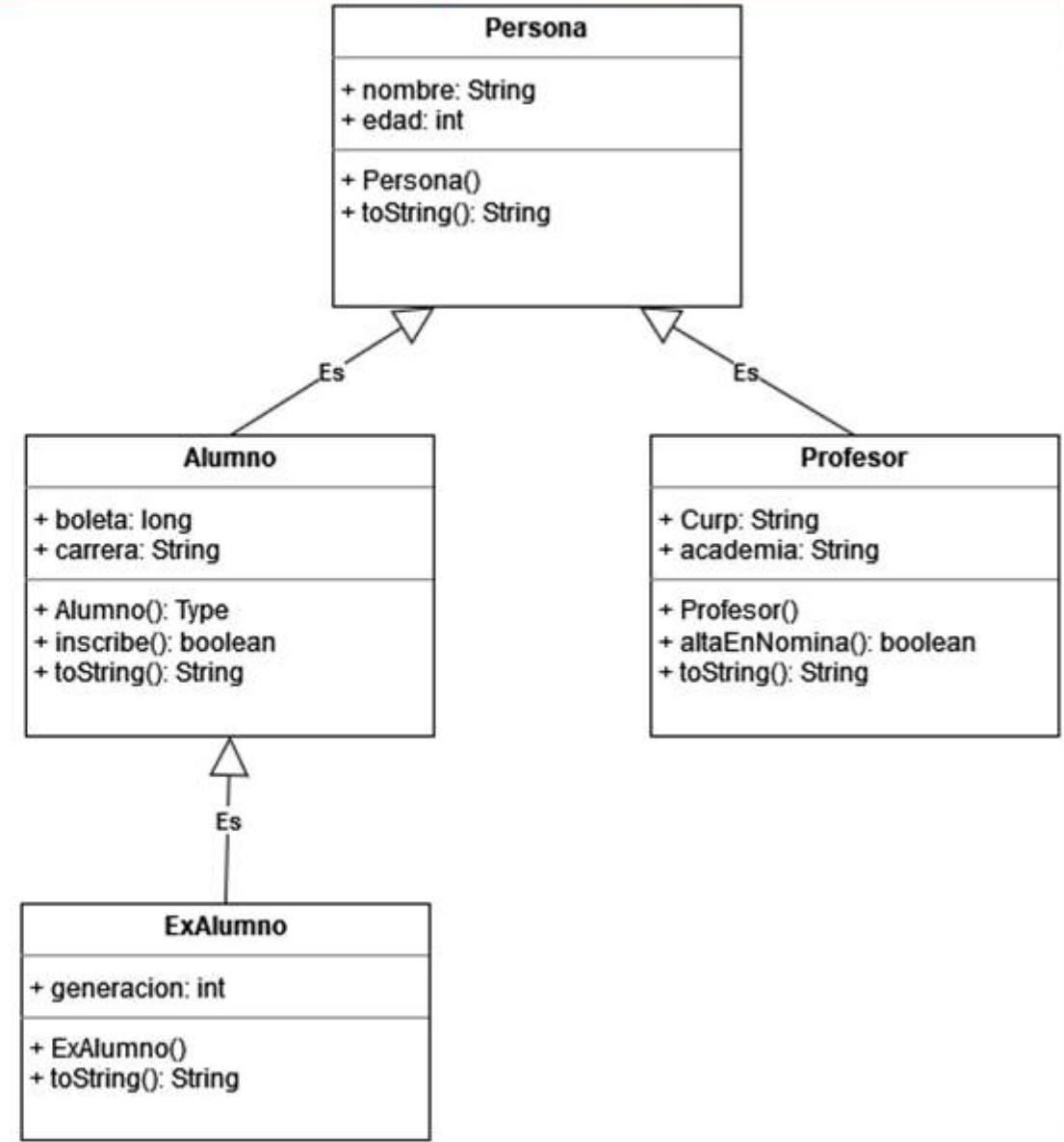
```
|public abstract interface IAjustes {  
|    public static final int VOLUMEN_MAXIMO = 100;  
|    public abstract void volumenMas ();  
|    public abstract void volumenMenos ();  
|    public abstract void encender();  
|    public abstract void apagar();  
|    public abstract void silencio();  
}  
  
|  
|    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;  
|        };  
|    }  
|}
```



Diagrama de clases

Representación gráfica de las clases y relaciones o dependencias entre ellas.

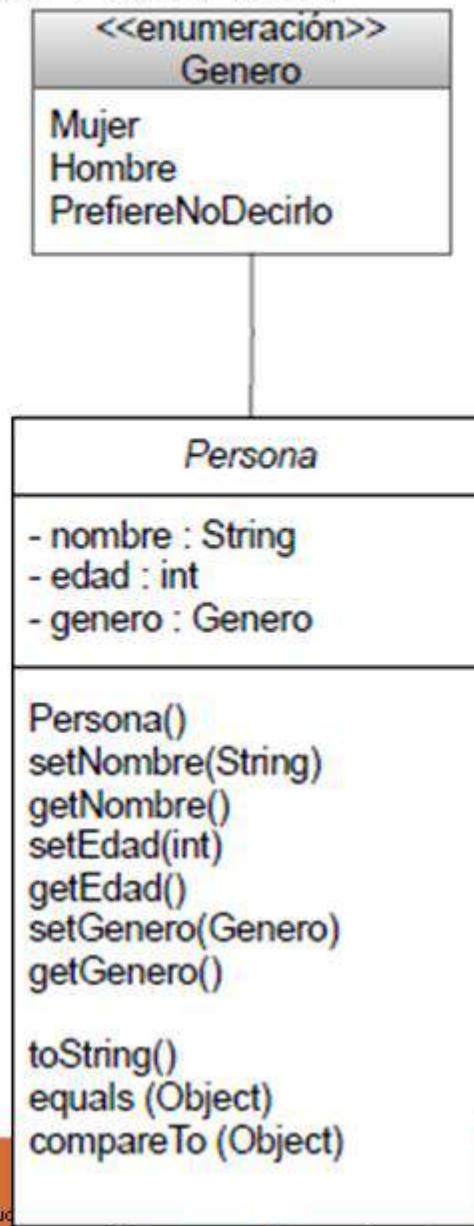
Existen diversas notaciones para su diagramación.



Relaciones entre clases y representación en un diagrama de clases

Línea continua sin punta de flecha:

- Puede usarse para ligar entre enumeración y clase donde se usa.

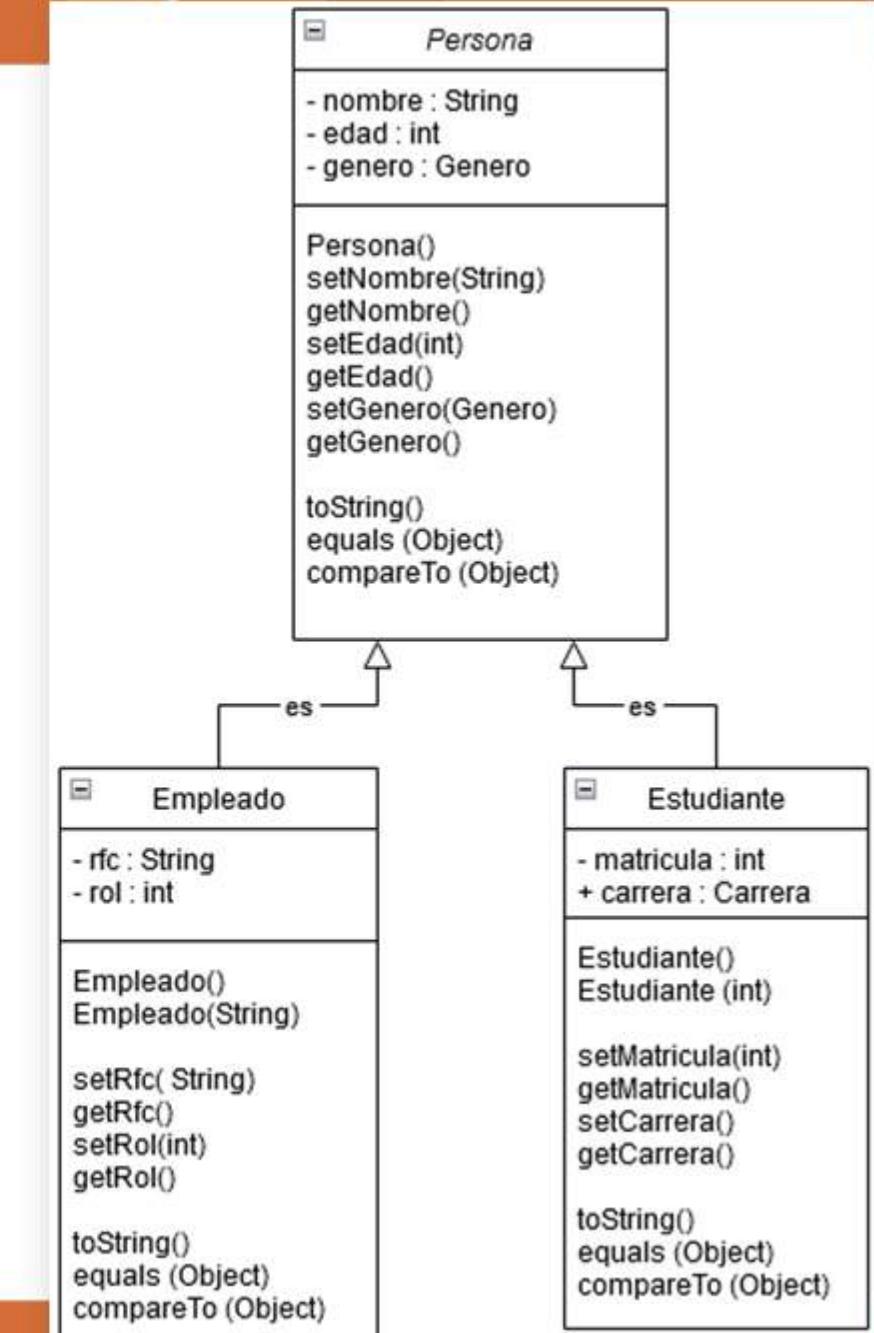


Relaciones entre clases y representación en un diagrama de clases

Herencia

Línea continua con punta de flecha:

- Suele usarse para indicar relación de herencia entre súper clase (donde inicia la línea) y la subclase (a donde apunta la punta de flecha).

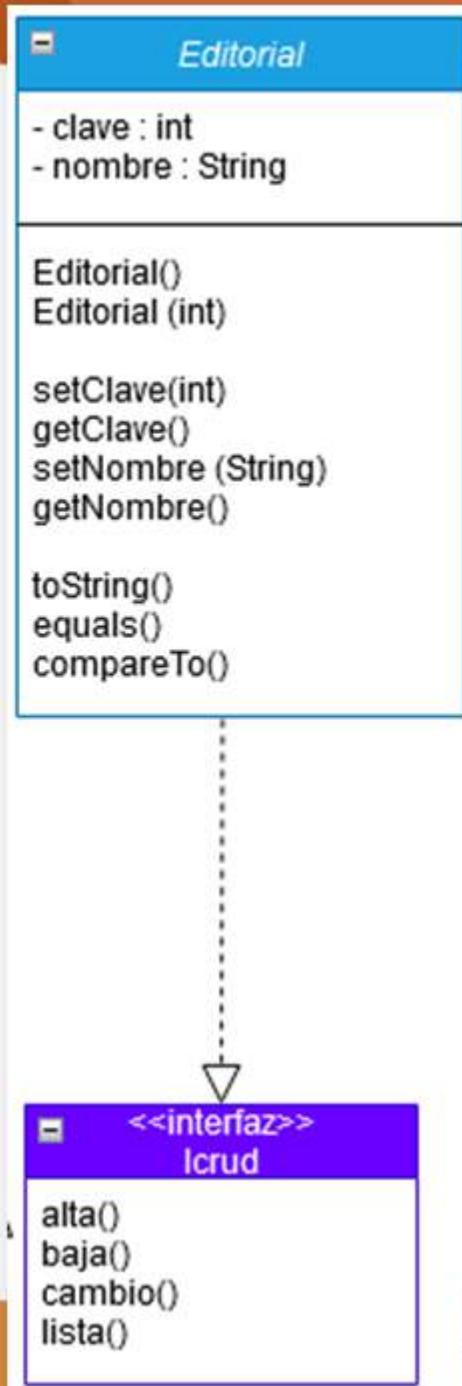


Relaciones entre clases y representación en un diagrama de clases

Implementación

Línea punteada con punta de flecha:

- Indicar implementación de en una clase (donde inicia la línea) de una interfaz (a donde apunta la punta de flecha).

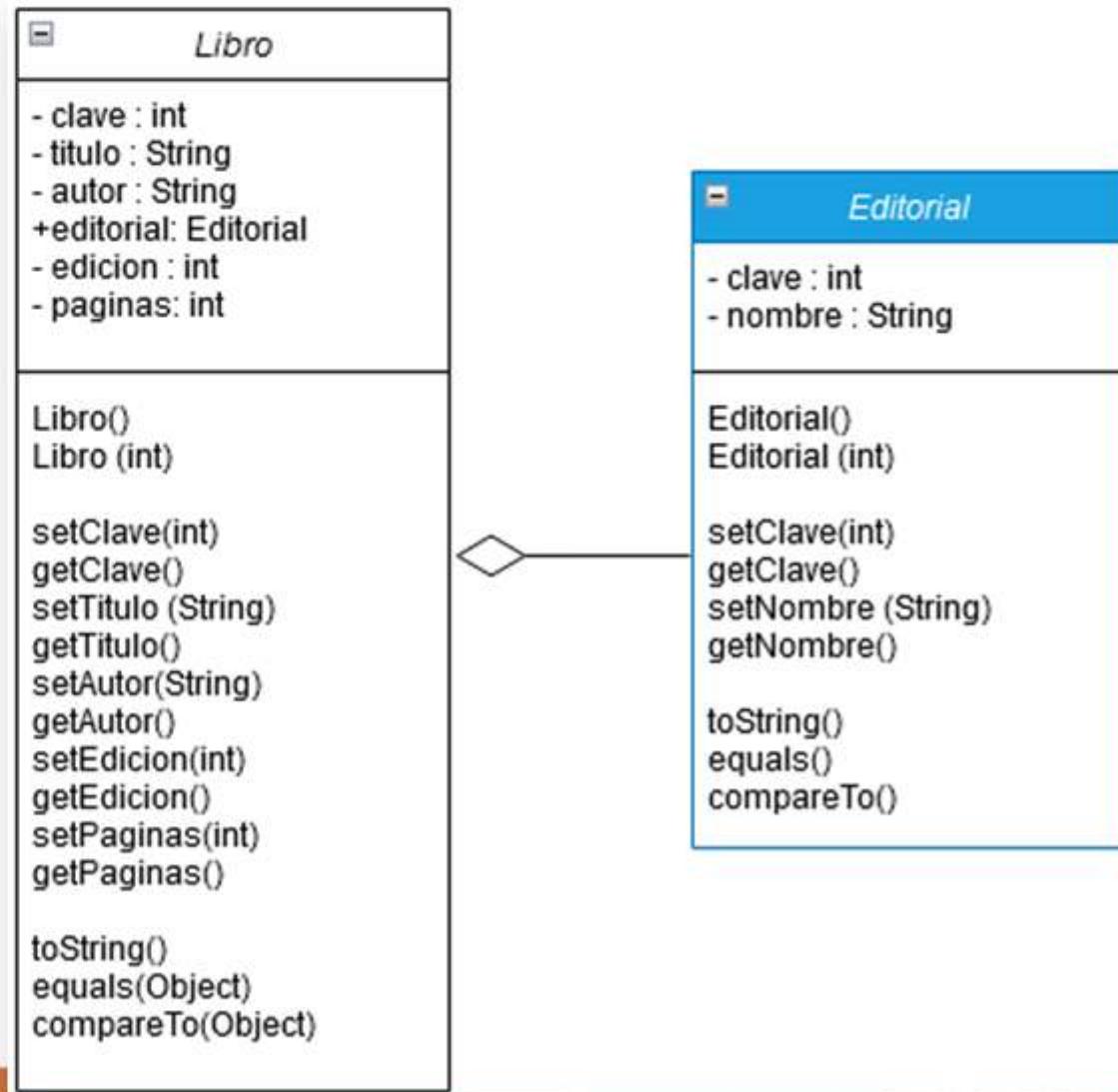


Relaciones entre clases y representación en un diagrama de clases

Agregación

Línea continua con diamante vacío:

- Relación del tipo “tiene un/a” unidireccional.
- Las clases son independientes entre sí.
- La clase a donde apunta el diamante es aquella donde se usa la otra clase.

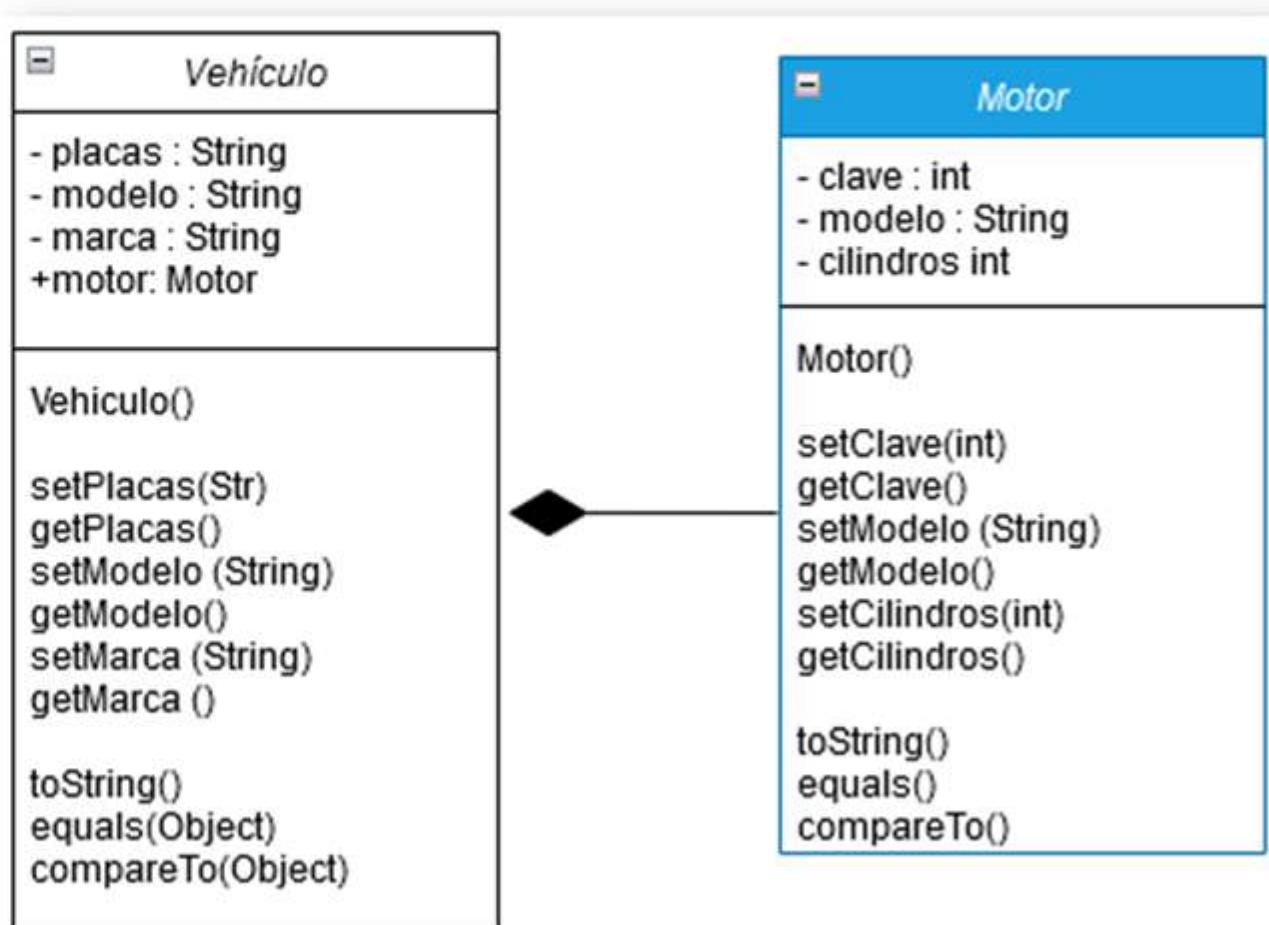


Relaciones entre clases y representación en un diagrama de clases

Composición

Línea continua con diamante relleno:

- Relación del tipo “parte de” unidireccional.
- Las clases son dependientes entre sí, una no existe sin la otra.



Clases anidadas

- Son clases definidas dentro de otra clase. Útil cuando:
 - Una clase solo será usada por otra clase específica (por seguridad, reforzar encapsulamiento)
 - Legibilidad del código

```
class ClaseRegular {  
    ...  
    class ClaseInterna {  
        ...  
    }  
    ...  
}
```

```
ClaseRegular objetoR = new ClaseRegular();  
  
//Creación de objeto interno A PARTIR de uno externo  
ClaseRegular.ClaseInterna objetoInterno = objetoR.new ClaseInterna();
```



* Casos especiales de clases no estáticas

Ejemplo de clases anidadas

```
class Principal {
    public String miembroNoEstatico="\tMiembro NO estático principal";
    static String miembroEstatico ="\tMiembro estático de principal";
    class InternaNoEstatica { //Clase DENTRO de clase Principal
        void prueba() {
            System.out.println(miembroNoEstatico);
            System.out.println(miembroEstatico);
        }
    }

    static class InternaEstatica { //Clase estática DENTRO de clases anidadas
        static void metodoPruebaEstatico () {
//ERROR        System.out.println(miembroNoEstatico);
            System.out.println("\t\tMétodo estático:"+miembroEstatico);
        }
        void metodoPruebaNOEstatico () {
//ERROR        System.out.println(miembroNoEstatico);
            System.out.println("\t\tMétodo NO estático:"+miembroEstatico);
        }
    }
}
```

```

Clase interna:
    Miembro NO estático principal
    Miembro estático de principal

Clase interna ESTÁTICA:
    Método estático de clase ESTÁTICA:
        Método estático:           Miembro estático de principal

    Métodos estático y NO estático de clase ESTÁTICA:
        Método estático:           Miembro estático de principal
        Método NO estático:        Miembro estático de principal

```

```
public class ClasesAnidadas {
    /** Ejemplo de uso de Clases anidadas ***/
    public static void main(String[] args) {
        System.out.println("Clase interna:");
        //Instancia de clase principal
        Principal principal = new Principal(); //Instancia de clase interna estática

        //Instancia de clase interna NO estática (debe existir un objeto de clase Principal
        Principal.InternaNoEstatica internaNoEstatica = principal.new InternaNoEstatica();
        internaNoEstatica.prueba();

        System.out.println("\nClase interna ESTÁTICA:");
        System.out.println("\tMétodo estático de clase ESTÁTICA:");
        //Acceso a método estático en clase estática interna
        Principal.InternaEstatica.metodoPruebaEstatico();
        //ERROR          Principal.InternaEstatica.metodoPruebaNOEstatico();

        System.out.println("\n\tMétodos estático y NO estático de clase ESTÁTICA:");
        Principal.InternaEstatica internaEstatica = new Principal.InternaEstatica();
        internaEstatica.metodoPruebaEstatico();
        internaEstatica.metodoPruebaNOEstatico();
    }
}
```

¿Qué pasa cuando se define `private` a `InternaEstatica` y `a InternaNoEstatica`?



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
```

Aserciones (assert)

Aseveraciones que deben ser verdaderas en tiempo de ejecución

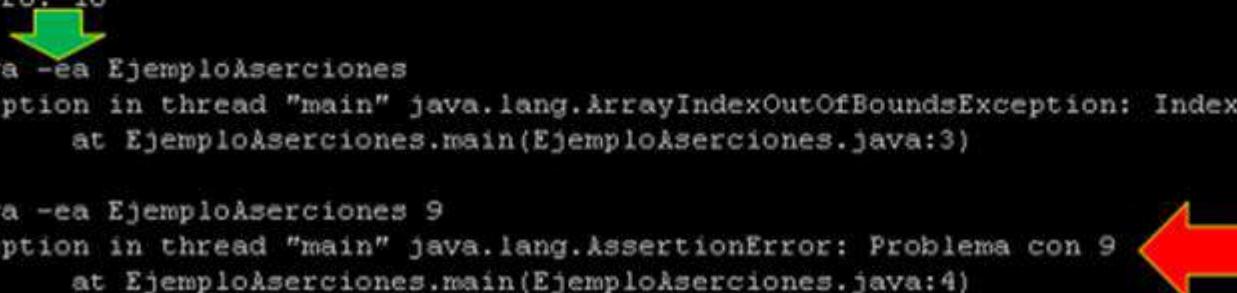
`assert condición: expresión`

- 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



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
```



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.



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,dia
        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());

        //Formato 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 |

Genéricos

- Forma que tiene Java de hacer clases, interfaces o métodos reciban un tipo (clase) como parámetros de entrada y así generalizar una solución para diferentes tipos de objetos de entrada.
- Similar a las plantillas de C++.



Ejemplo de Genéricos

```
public class Generico<T> {//T es la clase generica
    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

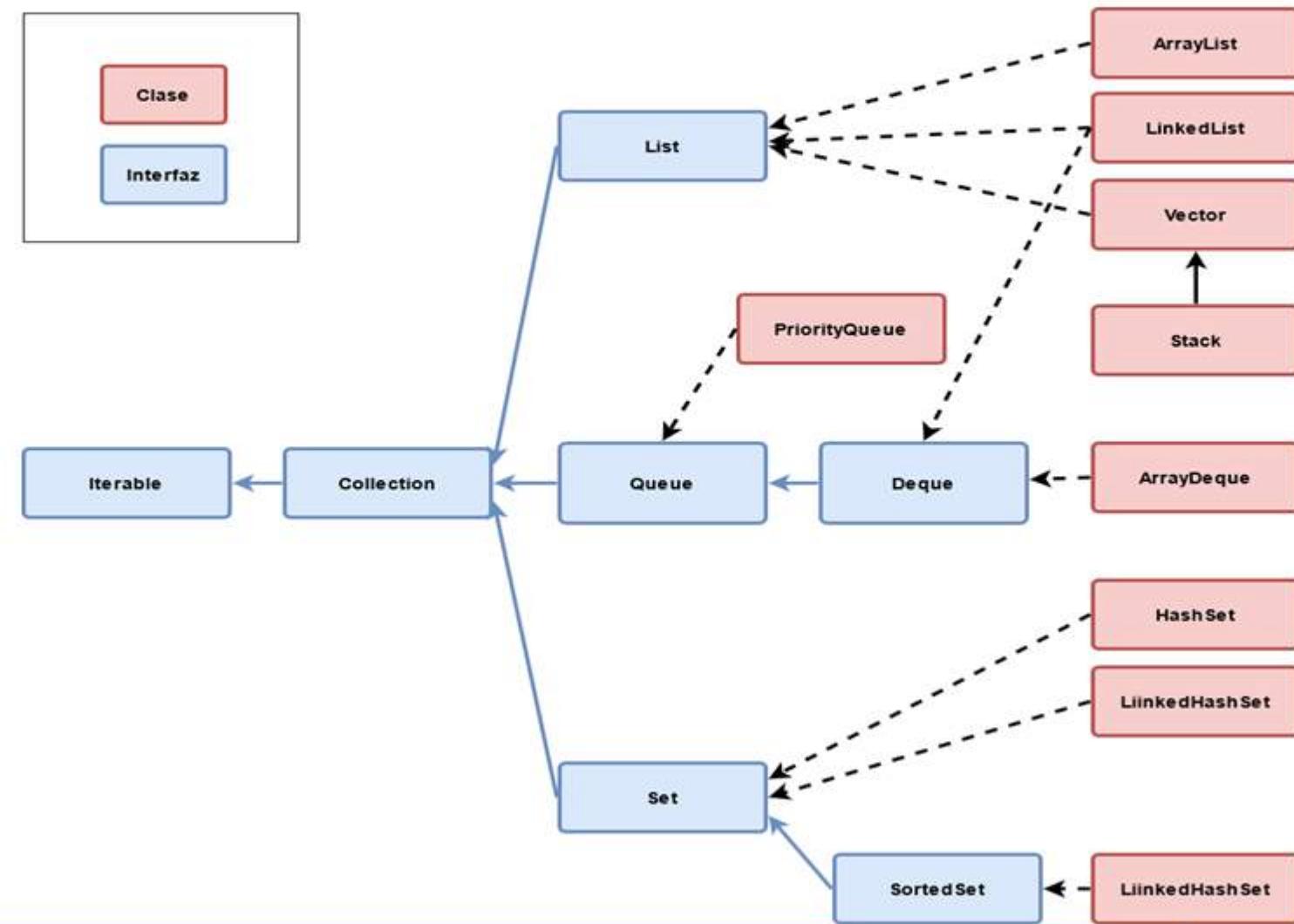
Miguel Hidalgo
Batman(100.0)



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 () | Vacía 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 índice |
| get (índice) | Regresa el elemento con dicho índice |
| indexOf (elemento) | Regresa primer posición de elemento |
| 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

| Miembro * | Descripción |
|--------------------|--|
| addFirst(elemento) | Añade un elemento al inicio de la lista |
| addLast(elemento) | Añade un elemento al final de la lista. Equivalente a add(elemento) |
| getFirst() | Regresa el primer elemento de la lista |
| getLast() | Regresa el último elemento de la lista |
| push(elemento) | Añade un elemento al final de la lista. Equivalente a add(elemento) y a addLast(elemento) |
| pop() | Regresa y elimina el elemento primer 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



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.
```

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

Autoboxing y Unboxing

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.

```
jshell> int variableEntera=10;
variableEntera ==> 10

jshell> Integer objetoEntero = 20;
objetoEntero ==> 20

jshell> variableEntera = variableEntera + objetoEntero;
variableEntera ==> 30

jshell> objetoEntero = objetoEntero -variableEntera;
objetoEntero ==> -10
```



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

