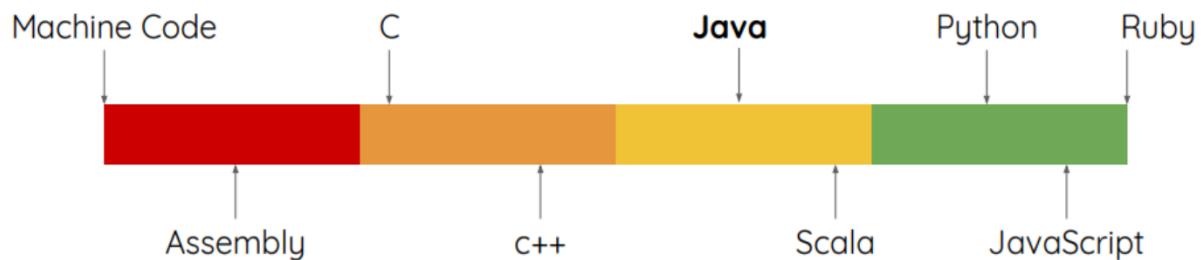


Desarrollo en Java

Java es un lenguaje de alto nivel:



Paradigmas de la programación

- Se comunican entre ellos

Programación estructurada:

- Programación secuencial
- Usa ciclos y condicionales

Programación funcional

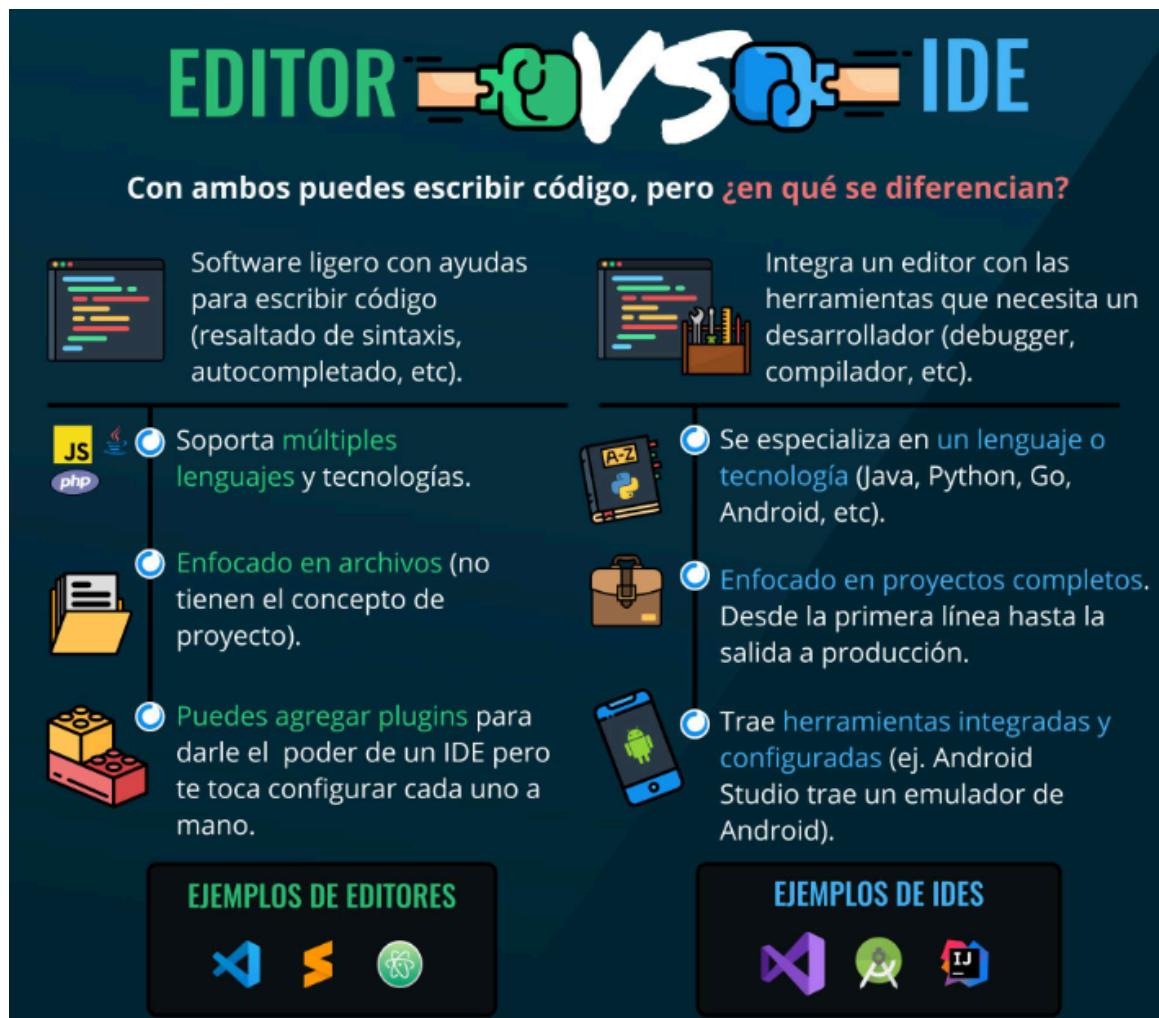
- Divide el programa en pequeños objetos
- Las tareas son ejecutadas por funciones

Programación orientada a objetos

- Divide un sistema en partes (objetos)

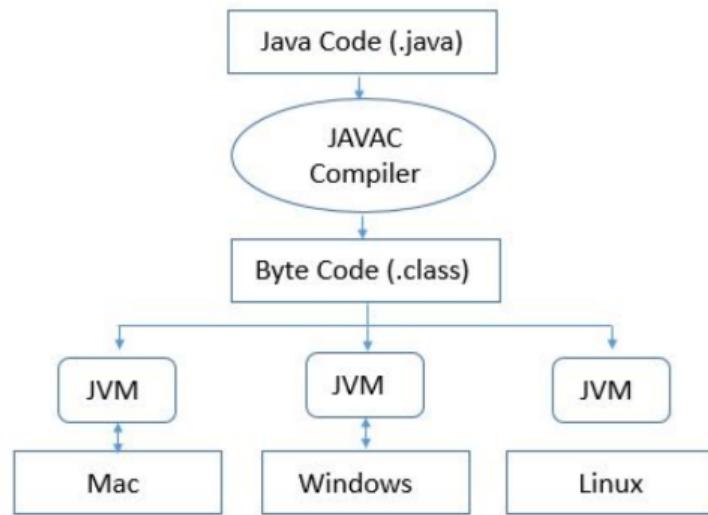
Programación reactiva

- Observa flujos de datos asincrónicos
- Reacciona frente a los cambios



El bytecode

Todo código java se compila en un lenguaje intermedio que no es 100% binario, está entremedio de la máquina y el código fuente.



¿ Que es JDK ?

JDK (Java Development Kit)

- JRE - Java Runtime Environment - (JVM)
- Compilador de java
- API de java

Palabras reservadas en java:

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

Tipos de datos primitivos:

Data Type	Size	Description
byte	1 byte	Almacena números enteros de -128 a 127
short	2 bytes	Almacena números enteros de -32,768 a 32,767
int	4 bytes	Almacena números enteros de -2,147,483,648 a 2,147,483,647
long	8 bytes	Almacena números enteros de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
float	4 bytes	Almacena números decimales con suficiente para almacenar de 6 a 7 dígitos
double	8 bytes	Almacena números decimales con suficiente para almacenar 15 dígitos
char	2 bytes	Almacena un carácter o código ASCII
boolean	1 bit	Almacena valores verdaderos o falsos (true y false)

Casteo

En la programación hay situaciones donde se necesita cambiar el tipo de dato. Genera un tipo de dato diferente al original.

Ejemplo:

```
double d = 86.45;  
int i = ( int ) d;
```

Wrapper

Java define una clase “wrapper” para cada tipo primitivo. Dan mayor funcionalidad para operaciones de comprobaciones y conversiones.

Primitivo	byte	short	int	long	float	double	char	boolean
Equivalente	Byte	Short	Integer	Long	Float	Double	Character	Boolean

```
Boolean bool1 = true;  
Character char1 = 'a';  
Double double1 = 10.98; } sin new  
  
Boolean bool2 = new Boolean (true);  
Character char2 = new Character ('a'); } con new  
Double double2 = new Double (10.98);
```

Alcance de las variables

Variables de instancia:

Los valores posibles son únicos para cada instancia.

Variables de clase:

Se declaran con el modificador “static” para indicarle al compilador que hay exactamente una copia de la variable, y es compartida por todas las instancias.

Variables de clase:

Solo es visible al método donde fue creada y no puede ser accedida desde el resto de clases.

Variables parámetros:

Se pasan entre métodos.

Clase Scanner

Permite leer datos de la consola, archivos, redes, etc.

Para utilizar la clase Scanner:

1_ Importarlo: `import java.util.Scanner;`

2_ Crearlo: `Scanner scanner = new Scanner(System.in);`

Funciones de Scanner:

- `nextLine()`: Lee una línea completa y devuelve una cadena.
- `nextInt()`: Lee un entero y devuelve un valor int.
- `nextDouble()`: Lee un número de punto flotante y devuelve un valor double.
- `nextBoolean()`: Lee un valor booleano y devuelve un valor boolean.
- `close()`: Cierra la fuente de entrada asociada con el objeto Scanner y libera cualquier recurso asociado.

Ejemplo:

```
>Main.java ×  
1 import java.util.Scanner;  
2  
3 public class Main {  
4     public static void main(String[] args) {  
5         Scanner entrada = new Scanner(System.in);  
6         System.out.print("Ingrese un número: ");  
7         int numero = entrada.nextInt();  
8         entrada.nextLine(); // Consumir el salto de linea  
9         System.out.print("Ingrese una cadena: ");  
10        String cadena = entrada.nextLine();  
11  
12        System.out.println("Número ingresado: " + numero);  
13        System.out.println("Cadena ingresada: " + cadena);  
14    }  
15}
```

Programación orientada a objetos

Está conformada por 4 pilares:

Abstracción: Permite seleccionar las características relevantes dentro de un conjunto.

Encapsulamiento: Es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos.

Herencia: Organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes.

Polimorfismo: Un mismo método (o acción) puede tener el mismo nombre pero un comportamiento diferente para el mismo o diferentes objetos.



Objetos

un objeto posee:

- Estado

- Comportamiento

- Identidad

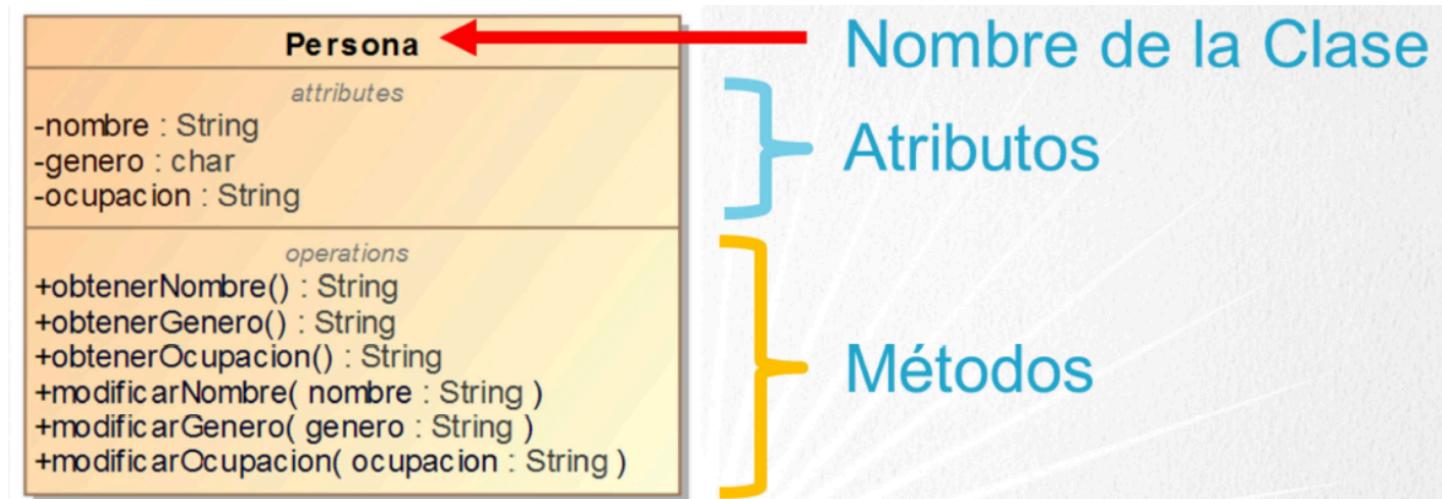
Lo que el objeto sabe. Cambia en el transcurso del tiempo. Implementado por un conjunto de propiedades.

Lo que el objeto puede hacer. Se implementa mediante métodos.

Cada objeto tiene una identidad única, incluso si su estado es idéntico al de otro objeto.

Lenguaje Unificado de Modelado (UML)

Es un lenguaje utilizado para describir un sistema, detallar las clases y artefactos que forman parte del mismo, describir las relaciones entre los mismos, y para documentar y construir.



Estructura de un método

Modificadores de acceso: Especifican la visibilidad del método y controlan su acceso desde otras partes del código. Los modificadores de acceso en Java son: public, protected, private y default.

Tipo de retorno: Especifica el tipo de dato que devolverá el método al final de su ejecución. Puede ser un tipo primitivo, un objeto, un arreglo u otro tipo de datos.

Nombre del método: Es el nombre con el que se invocará al método en el código.

Parámetros: Son los valores que se pasan al método para ser procesados. Los parámetros se definen dentro de los paréntesis y pueden ser de cualquier tipo de dato.

Cuerpo del método: Es el bloque de código que contiene las instrucciones que se ejecutan cuando se llama al método.

Palabra clave “return”: Especifica el valor que se devolverá al final del método.

Constructores

Es un **MÉTODO ESPECIAL** que se utiliza para crear e inicializar objetos de una clase.

- Es un tipo de método que se ejecuta automáticamente cuando se crea un objeto de una clase y se utiliza para inicializar los valores de los atributos de la clase.
- Tienen el mismo nombre que la clase a la que pertenecen y no tienen un tipo de retorno, ni siquiera void.
- Pueden aceptar parámetros para inicializar las variables de instancia de la clase.

Palabra "this"

"this" es una palabra clave en Java que hace referencia al objeto actual en el que se está ejecutando el método. Se utiliza para distinguir entre variables de instancia (atributos de la clase) y variables locales o parámetros con el mismo nombre.

Ejemplo de constructor por defecto:

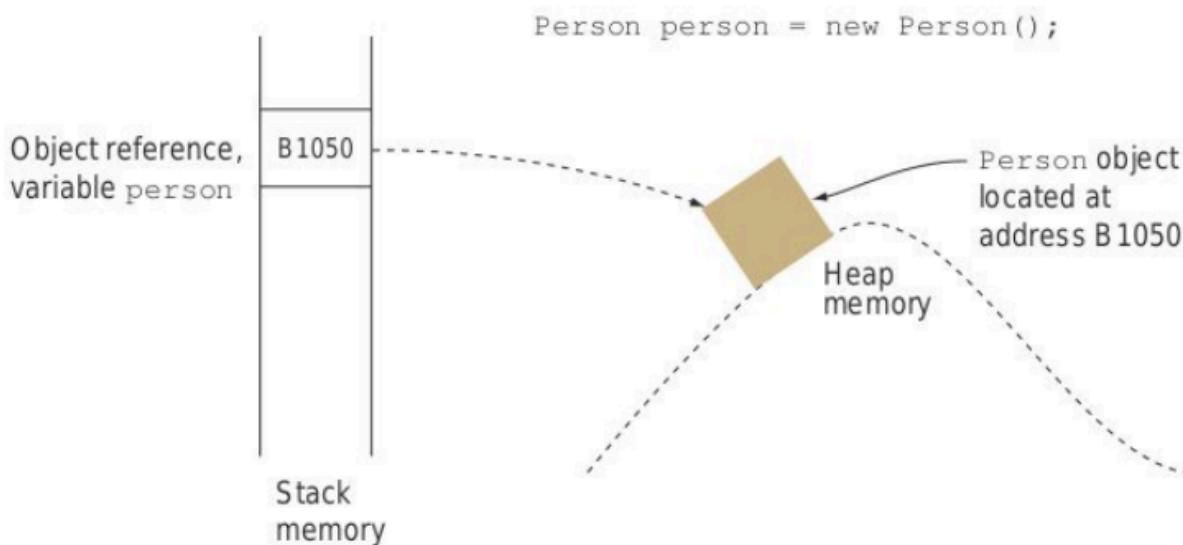
```
public class Persona {  
  
    //Atributos  
    String nombre;  
    int edad;  
  
    // constructor por defecto de la clase Persona  
    public Persona() {  
        // inicialización de los atributos por defecto  
        this.nombre = null;  
        this.edad = 0;  
    }  
  
    //Metodos  
    public void detallarInformacion() {  
  
        System.out.println("Nombre: " + nombre);  
        System.out.println("Edad: " + edad);  
    }  
}
```

Ejemplo de la palabra "this":

```
public class Persona {  
    2 usages  
    private String nombre;  
    2 usages  
    private int edad;  
  
    // Constructor que inicializa los campos nombre y edad (recibe parámetros)  
    1 usage  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre; // 'this.nombre' se refiere al campo de la clase  
        this.edad = edad; // 'this.edad' se refiere al campo de la clase  
    }  
    // Métodos getter para acceder a los campos  
    1 usage  
    public String getNombre() {  
        return nombre;  
    }  
    1 usage  
    public int getEdad() {  
        return edad;  
    }  
    public static void main(String[] args) {  
        // Crear un objeto Persona utilizando el constructor  
        Persona persona = new Persona( nombre: "Juan", edad: 25);  
        System.out.println("Nombre: " + persona.getNombre());  
        System.out.println("Edad: " + persona.getEdad());  
    }  
}
```

Creacion y destrucción de objetos

Cuando un objeto es instanciado utilizando el operador new, se retorna una dirección de memoria. La dirección en memoria contiene una referencia a una variable.



Zona de datos: Donde se almacenan las instrucciones del programa, las clases con sus métodos y constantes. No se puede modificar en tiempo de ejecución.

Stack: El tamaño se define en tiempo de compilación y es estático en tiempo de ejecución. Aquí se almacenan las instancias de los objetos y los datos primitivos (int, float, etc.)

Heap: Zona de memoria dinámica. Almacena los objetos que se crean.

Modificadores de acceso

PUBLIC: Ofrece la máxima visibilidad: una variable, método o clase será visible desde cualquier clase. Se puede acceder libremente a un miembro público mediante un código definido fuera de su clase.

PRIVATE: Cuando un método o un atributo es declarado como private, su uso queda restringido al interior de la misma clase. Se puede acceder a un miembro privado solo por otros métodos definidos por su clase .

PROTECTED: Un método o atributo declarado como protected es visible para las clases del mismo paquete y subclases.

DEFAULT: Visibilidad para clases del mismo paquete.

Visibilidad	Public	Protected	Default	Private
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase del mismo Paquete	SI	SI	SI	NO
Desde una SubClase fuera del mismo Paquete	SI	SI, a través de la herencia	NO	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

JVM - Garbage collector

Es un proceso de baja prioridad que se ejecuta dentro de la JVM. Técnica por la cual el ambiente de objetos se encarga de destruir y asignar automáticamente la memoria heap. Un objeto podrá ser “limpiado” cuando desde el stack ninguna variable haga referencia al mismo.

Getters y setters

Son métodos de acceso, lo que indica que son siempre declarados públicos, y nos sirven para dos cosas:

SETTERS: Del Inglés Set (establecer), nos sirve para asignar un valor inicial a un atributo, pero de forma explícita. El Setter nunca retorna nada, siempre es void, y solo nos permite dar acceso público a ciertos atributos que deseemos el usuario pueda modificar.

GETTERS: Del Inglés Get (obtener), nos sirve para obtener (recuperar o acceder) el valor ya asignado a un atributo y utilizarlo para cierto método.

Ejemplo:

```
public class Animal {  
    String nombre;  
  
    ... //constructor  
  
    //setters y getters  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre( String nombre ) {  
        this.nombre = nombre;  
    }  
}
```

Clase math

- Math.max() : Devuelve el valor más grande de dos números.
- Math.min() : Devuelve el valor más pequeño de dos números.
- Math.sqrt() : Devuelve la raíz cuadrada de un número.
- Math.pow() : Devuelve el resultado de elevar un número a una potencia específica.
- Math.random() : Devuelve un número aleatorio entre 0 y 1.

Clase string

Es una de las clases más utilizadas en el lenguaje, y se utiliza para representar y manipular cadenas de caracteres (letras, números y caracteres especiales).

Las variables del tipo String son objetos, por lo cual tienen acceso a sus métodos. Algunos métodos se pueden utilizar de manera estática con la clase String (tema que se verá más adelante), y otros vamos a necesitar una instancia de un objeto. Es una clase final (no puede ser heredada).

SUS OBJETOS SON INMUTABLES: Una vez que se crea un objeto String, no se puede modificar su contenido.

Formas de declarar un string:

```
String miCadena = new String("Hola mundo");
```

```
String otraCadena = "Esto es otra cadena";
```

Funciones de la clase string:

- int compareTo(String b): Compara contra la cadena del argumento.
- Operador relacional de igualdad (==): Compara por referencia.
- boolean equals(String b): Compara por valor.
- boolean equalsIgnoreCase(String b): Compara independientemente de mayúsculas o minúsculas.
- int length() : Retorna el número de caracteres.
- Para concatenar cadenas: simplemente se utiliza el operador de suma.
- toLowerCase() : devuelve una copia de la cadena actual en minúsculas.
- toUpperCase() : devuelve una copia de la cadena actual en MAYÚSCULAS.
- trim() : devuelve una copia de la cadena actual sin espacios en blanco al inicio y al final.
- int indexOf (String cadena) : Devuelve la posición en que se encuentra el carácter (o cadena) indicado por primera vez, buscando desde el principio.
- int lastIndexOf (String cadena): Indica en qué posición se encuentra el carácter (o cadena) indicado por primera vez, buscando desde el final.
- substring (int beginIndex, int endIndex) : devuelve una subcadena de la cadena actual que comienza en el índice especificado y termina en el índice especificado.
- String substring (int desde): Extrae la subcadena desde la posición indicada.
- boolean startsWith(String prefijo) Dice si la cadena comienza o no con el prefijo indicado.
- boolean endsWith(String sufijo) Dice si la cadena termina o no con el sufijo indicado.
- String[] split(String patron) Divide la cadena en varias subcadenas utilizando el patrón indicado como separador.
- char charAt(int posicion) Retorna el carácter que está en la posición indicada.
- char[] toCharArray() Convierte la cadena en un arreglo de caracteres.

Constructores de la clase String:

CONSTRUCTOR	DESCRIPCIÓN
<code>String()</code>	Constructor por defecto. El nuevo String toma el valor "" <code>String s = new String(); //crea el string s vacío.</code> Equivale a: <code>String s = "";</code>
<code>String(String s)</code>	Crea un nuevo String, copiando el que recibe como parámetro. <code>String s = "hola";</code> <code>String s1 = new String(s);</code> <code>//crea el String s1 y le copia el contenido de s</code>
<code>String(char[] v)</code>	Crea un String y le asigna como valor los caracteres contenidos en el array recibido como parámetro. <code>char [] a = {'a', 'b', 'c', 'd', 'e'};</code> <code>String s = new String(a);</code> <code>//crea String s con valor "abcde"</code>
<code>String(char[] v, int pos, int n)</code>	Crea un String y le asigna como valor los n caracteres contenidos en el array recibido como parámetro, a partir de la posición pos. <code>char [] a = {'a', 'b', 'c', 'd', 'e'};</code> <code>String s = new String(a, 1, 3);</code> <code>//crea String s con valor "bcd";</code>

StringBuilder

Los StringBuilder son similares a los String porque gestionan conjuntos de caracteres, pero se pueden modificar sin estar generando nuevas instancias en memoria como ocurre con los Strings.

```
StringBuilder sb = new StringBuilder("abc");
StringBuilder sb = "abc";
```

Funciones de StringBuilder:

.append(String s)

- Adjunta el String s al final del StringBuffer actual
- Devuelve una referencia a si mismo

.charAt(int index)

- Devuelve el carácter indicado por el índice
- Al igual que los arrays la primera posición es la 0

.setCharAt(int index, char ch)

- Se cambia el carácter de la posición del valor index al valor almacenado en ch

.delete(int start, int end)

- Elimina los caracteres desde la posición start hasta la end -1
- Devuelve una referencia a si mismo

.insert(int offset, String s)

- Inserta el string pasado en la posición especificada por offset
- Devuelve una referencia a si mismo

.reverse()

- Invierte los caracteres
- Devuelve una referencia a si mismo

Arrays

Son tipos de datos de referencia (objetos) que contienen varios elementos ordenados, una colección. Pueden contener tipos de datos primitivos u objetos, pero siempre asociado a un solo tipo de datos.

Declarar un arreglo:

```
//Arreglos con datos primitivos
int numeros [];
boolean [] banderas;

//Arreglos con datos de tipo Objeto
Persona personas [];
String [] nombres;
```

Como un Array es considerado un Objeto en Java, debe crearse una instancia de este objeto. Por ende, de forma similar al resto de los objetos, se usa el operador new. Para crear el array hay que usar el operador new, más el tipo de los elementos, más el número de elementos.

```
//Arreglos con datos primitivos
int numeros [] = new int[5];
boolean [] banderas = new boolean[2];

//Arreglos con datos de tipo Objeto
Persona personas [] = new Persona[3];
String [] nombres = new String[7];
```

length : Retorna la longitud total del arreglo (esté cargado o no).

```
int numeros [] = new int[5];
int longArray = numeros.length;
System.out.println(longArray);      //5
```

Formas de recorrer un array:

- While:

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

int numeros []= new int[5]; //Declaro e instancio el arreglo

int i=0; // inicializo el contador en 0

while(i < numeros.length) {      //numeros.length es 5
    System.out.println("Ingrese un numero en la posicion : " + i);
    numeros[i]=teclado.nextInt(); //Ingreso un valor y lo asigno a la posicion i
    i++;
}

i=0;

while(i < numeros.length) {
    System.out.println("Posicion " + i + " valor " + numeros[i]);
    //Muestra primero la posicion i y luego accede al elemento [i] para mostrarlo
    i++;
}
```

- For:

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

System.out.println("Cuantos elementos quiere cargar?");

int tamArray=teclado.nextInt(); // Ingreso por teclado el tamaño del array

int numeros []= new int[tamArray]; //Declaro e instancio el arreglo


for(int i=0 ; i < numeros.length ; i++) {
    System.out.println("Ingrese un numero en la posicion : " + i);
    numeros[i]=teclado.nextInt();
}

for(int i=0 ; i < numeros.length ; i++) {
    System.out.println("Posicion " + i + " valor " + numeros[i]);
}
```

- For each:

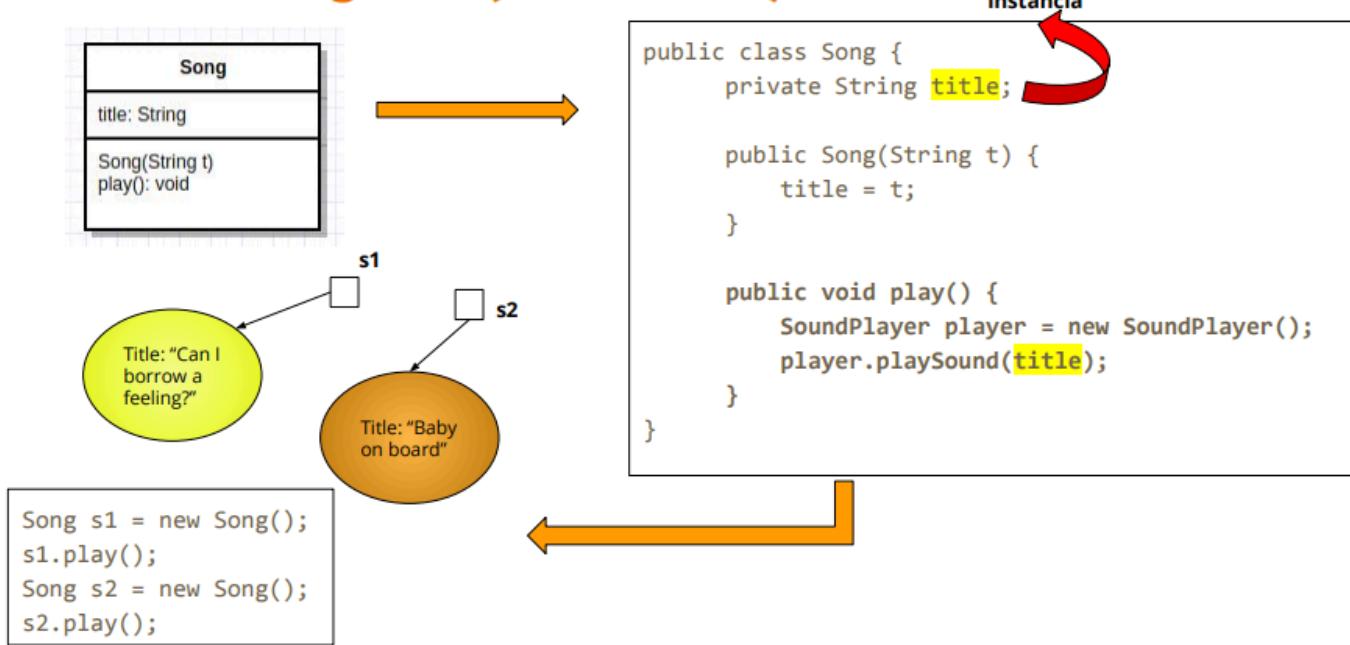
```
String [] nombres = {"Andres", "Maria", "Jose", "Isabel", "Juan"};

for(String nom: nombres) {
    System.out.println("nombre = " + nom);
}
```

Métodos regulares: Non static

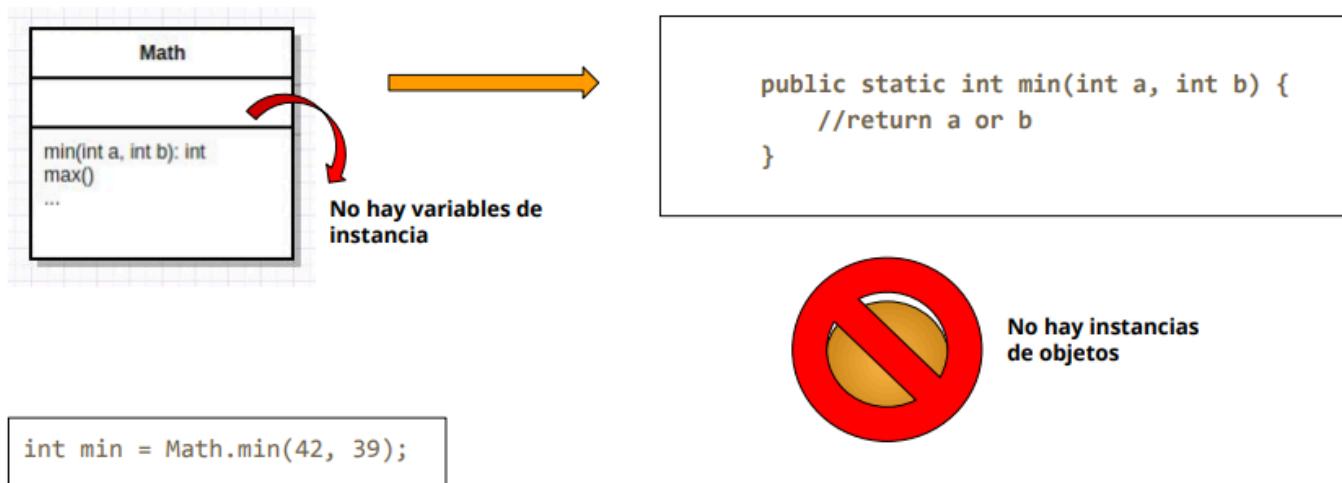
- Son los métodos tradicionales, los que vimos hasta ahora: usan valores variables de instancia.
- Es necesario crear una instancia de la clase, para poder invocarlos.

Método regular (non-static)



Métodos static

- Métodos en los que NO es necesario crear una instancia de la clase. La llamada a los métodos static se realiza mediante la clase: NombreClase.metodo(), respetando las reglas de visibilidad.
- Aunque también se pueden llamar con un objeto de la clase, no es recomendable debido a que son métodos dependientes de la clase y no de los objetos.
- Para declarar un método como static, debemos colocar la palabra static antes del tipo de valor de retorno en la declaración del método.



- Se pueden combinar métodos regulares y estáticos en la misma clase.
- Los métodos estáticos no pueden usar variables de instancia.
- Los métodos estáticos no pueden usar métodos regulares, porque usan variables de instancias
- Un método estático no puede hacer referencia a elementos no estáticos de su misma clase.
- Un método estático no puede hacer uso de la palabra reservada super y this.

Variables static

- Se declaran de igual manera que otra variable, añadiendo como prefijo la palabra reservada static;
- Las variables miembro static no forman parte de los objetos de la clase sino de la propia clase;
- Se accede a ellas de la manera habitual, simplemente con su nombre. Desde el exterior se accede con el nombre de la clase, el selector y el nombre de la variable.
- Los atributos estáticos deben llevar un modificador de acceso que permita su uso desde el exterior de la misma, este puede ser public, protected o ninguno.

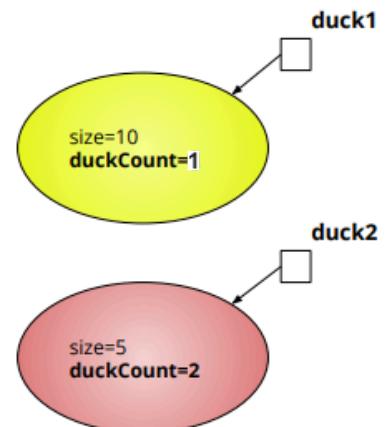
```
public class Duck {
    private int size;
    private static int duckCount = 0;

    Public Duck() {
        duckCount++;
    }

    public void setSize(int s) {
        size = s;
    }

    public int getSize() {
        return size;
    }
}
```

Se inicializa una única vez,
cuando la clase de carga por
primera vez.



Constantes

```
public static final double PI = 3.141592653589793
```

- La palabra reservada final indica que una vez inicializada, el valor de la variable no puede cambiar.
- Generalmente se establecen como public para que puedan ser accedidas desde cualquier lugar de nuestro código.
- Son estáticas para que no sea necesario crear una instancia de la clase para poder usarlas.
- El nombre de una variable debe ir en mayúsculas.

Final

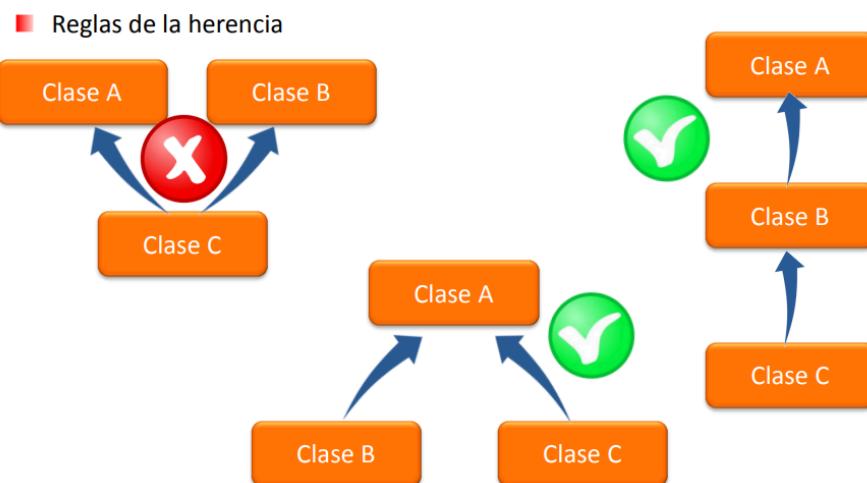
- La palabra reservada final no es sólo para variables estáticas.
- Se puede usar final para variables de instancias, variables locales, parámetros de métodos y clases.
- Indica que el valor no puede cambiar una vez que fue inicializado.
- Una variable final significa que no puede cambiar su valor.
- Un método final significa que no puede sobreescibirse.
- Una clase final significa que no puede tener subclases.

Herencia

La herencia es una forma de reutilización de software en la que se crea una nueva clase absorbiendo los miembros (atributos y métodos) de una clase existente, pudiendo al mismo tiempo añadir atributos y métodos propios.

Superclase y subclase

- El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol.
- En la estructura jerárquica, la clase padre se llama SUPERCLASE.
- La clase hija de una superclase se llama SUBCLASE.
- Las subclases no están limitadas al estado y comportamiento provisto por la superclase → pueden agregar variables y métodos además de los que ya heredan.
- Las clases hijas también pueden sobreescribir los métodos que heredan.
- Una superclase puede tener cualquier número de subclases.
- En Java, una subclase puede tener sólo una superclase (HERENCIA SIMPLE).
- Si es posible la herencia multinivel (es decir, "A" puede ser heredada por "B", y "C" puede heredar de "B").
- Por defecto, todas las clases derivan de `java.lang.Object`, a no ser que se especifique otra clase padre.



Ventajas de la implementación de herencia en POO:

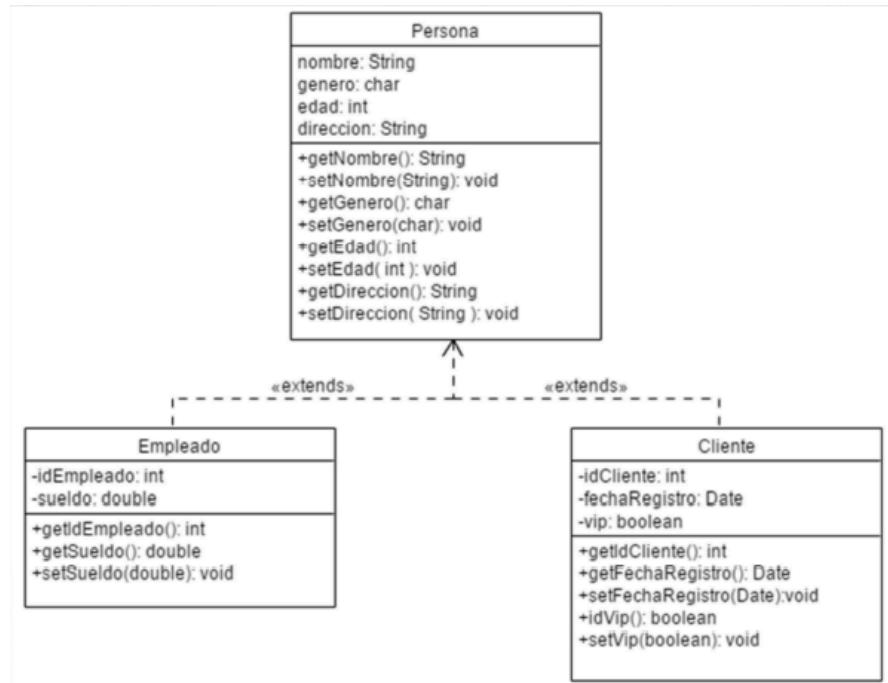
Reutilización de código: la herencia permite a las clases hijas utilizar el código existente en la clase padre, lo que permite ahorrar tiempo y esfuerzo de programación, y además evitar errores, ya que se reutiliza código ya probado.

Mantenibilidad y extensión: la herencia puede facilitar el mantenimiento del código, la actualización y la extensión, ya que cualquier cambio en la clase padre se reflejará automáticamente en las clases hijas. Además, si tenemos una clase con una determinada funcionalidad y tenemos la necesidad de ampliar dicha funcionalidad, no necesitamos modificar la clase existente, sino que podemos crear una clase que herede de la primera, adquiriendo toda su funcionalidad y añadiendo sólo las nuevas.

Modularidad: la herencia puede ayudar a organizar y estructurar el código de manera modular, lo que puede facilitar su comprensión y su mantenimiento.

API: Java proporciona gran cantidad de clases (bibliotecas) al programador, en el API (Application Programming Interface) Java.

UML con herencia:



Sobreescritura de métodos

Cuando una clase hereda de otra, el comportamiento de los métodos que hereda no siempre se ajusta a las necesidades de la nueva clase. En estos casos, la subclase puede optar por volver a reescribir el método heredado. Es lo que se conoce como sobrescritura de un método.

Es un concepto que permite que una clase hija proporcione su propia implementación de un método que ya está definido en la clase padre.

Cuando se llama al método en un objeto de la clase hija, la implementación de la clase hija se ejecutará en lugar de la implementación de la clase padre.

La firma del método (su formato, que incluye el nombre del método, los tipos de parámetros y el tipo de retorno) debe ser la misma en ambas clases.

El método sobrescrito puede tener un modificador menos restrictivo que el de la superclase. Por ejemplo, el método de la superclase puede ser protected y la versión sobrescrita en la subclase puede ser public, pero nunca uno más restrictivo.

Palabra reservada super

La palabra clave super es una llamada al constructor de la superclase.

La llamada al super debe tener los mismos parámetros que tenga el constructor de la superclase. El constructor de la superclase inicializa los campos correspondientes y le pasa el control al constructor de la subclase.

El constructor de una subclase debe tener siempre como primera sentencia una invocación al constructor de su superclase.

Si el constructor en la subclase no invoca explícitamente al constructor de la superclase, el compilador de Java inserta automáticamente una llamada al super sin parámetros.

En contra de la regla de las llamadas a super en los constructores, la llamada a super en los métodos puede ocurrir en cualquier lugar dentro de dicho método, no tiene por qué ocurrir en su primer sentencia.

Al contrario que en las llamadas a super en los constructores, no se genera automáticamente ninguna llamada a super y tampoco se requiere ninguna llamada a super, es completamente opcional. Por lo tanto, el método de una subclase podría sobrescribir y ocultar por completo la versión de la superclase del mismo método.

Derechos de acceso

Una subclase puede invocar a cualquier método público de su superclase como si fuera propio, no necesita ninguna variable.

Una subclase NO puede acceder a los miembros privados de la superclase.

Si un método de una subclase necesita acceder a un campo privado de su superclase, la superclase necesitará ofrecer los métodos apropiados (por ejemplo, los setters y getters).

Clase final

Si queremos evitar que una clase sea heredada por otra, deberá ser declarada con el modificador final delante de superclase.

```
public final class ClaseA {  
    //Código de la clase  
    :  
}
```

Polimorfismo

Es la capacidad que tienen los objetos de distintas clases de responder a un mismo método, de forma diferente.

Esto se logra a través de la herencia, en la cual una clase hereda los métodos y propiedades de otra.

→ Polimorfismo estático o sobrecarga de métodos:

- Se produce cuando una clase tiene varios métodos con el mismo nombre pero con diferentes parámetros o argumentos de entrada. (esto significa que pueden realizar acciones similares, pero con diferentes tipos de datos o número de argumentos).
- En la sobrecarga no interviene el tipo de retorno.
- El compilador de Java decide cuál de los métodos sobrecargados se debe invocar en tiempo de compilación, basándose en los tipos de argumentos que se le pasan.

La sobrecarga también podría darse en las clases hijas: una clase hija redefine un método de la clase padre, con el mismo nombre pero distinta lista de parámetros.

→ Polimorfismo dinámico o enlace dinámico:

Cuando encontramos un comportamiento que muchas clases derivadas van a realizar, la tendencia es enviar ese método a la superclase para que todas sus hijas realicen ese comportamiento. Sin embargo se pierde la especialización de ese método. Para esto existe, la sobreescritura del método de la clase padre en las clases hijas. El método tiene la misma firma en la clase padre y en las hijas, y al invocarlo, se determinará en tiempo de ejecución cuál de ellos se ejecutará.

Diferencias entre ambos:

- TIEMPO: El estático se da en tiempo de COMPILACIÓN y el dinámico en tiempo de EJECUCIÓN.
- En el ESTÁTICO los métodos tienen el MISMO NOMBRE pero DIFERENTES PARÁMETROS, y los métodos se encuentran en la MISMA CLASE o en SUBCLASES.
- En el DINÁMICO los métodos tienen la MISMA FIRMA, y se encuentran en DISTINTAS CLASES.

Ejecución dinámica de métodos

Pueden invocarse aquellos métodos del objeto que también estén definidos o declarados en la superclase, pero no aquellos que sólo existan en la clase a la que pertenece el objeto.

El método que se ejecutará se determina en tiempo de ejecución y está determinado por el tipo dinámico, no por el tipo estático (de ahí el nombre de polimorfismo dinámico).

Los métodos sobrescritos de las subclases tienen precedencia sobre los métodos de las superclases.

La búsqueda del método comienza al final de la jerarquía de herencia (comienza en la clase dinámica de la instancia), entonces la última redefinición de un método (el método que se encuentra primero) es la que se ejecuta primero.

Cuando un método está sobrescrito, sólo se ejecuta la última versión (la más baja en la jerarquía de herencia). Las versiones del mismo método en cualquier superclase no se ejecutan automáticamente.

Encapsulamiento

El encapsulamiento en Java se refiere al mecanismo de ocultar los detalles internos de una clase y proporcionar una interfaz pública para acceder a los datos y métodos de la clase.

Ventajas:

- Ayuda a mantener la integridad de los datos de la clase.
- Permite que los cambios internos de la clase sean más fáciles de realizar.
- Facilita la reutilización del código.
- Mejora la seguridad del código.

Utilizando los métodos públicos para acceder y modificar los datos :

```
Gato gato1 = new Gato("Pelusa", "Gris", 1, true);

System.out.println("El nombre del gato es: " + gato1.getNombre());
System.out.println("La edad del gato es: " + gato1.getEdad());
System.out.println("El color del gato es: " + gato1.getColor());
System.out.println("Es adoptado : " + gato1.isAdoptado());

System.out.println("Modifique el nombre del gato");
gato1.setNombre("Rocky");
System.out.println("El nombre nuevo del gato es: " + gato1.getNombre());
```

Abstracción

- Es uno de los pilares de la POO
- Consiste en la capacidad de representar un objeto en su forma más esencial, abstracta y general, sin preocuparse por los detalles específicos de su implementación.
- Permite encapsular la complejidad de un objeto y separar la implementación de sus métodos públicos, lo cual facilita la comprensión y el mantenimiento del código y permite una mayor modularidad y reutilización del mismo.

Clase abstracta

Una clase abstracta es una clase similar a una clase concreta (posee atributos y métodos), pero la gran diferencia es que es una clase que NO SE PUEDE INSTANCIAR DIRECTAMENTE (no se pueden crear objetos de una clase abstracta), sino que se utiliza como una plantilla o modelo para definir las características y el comportamiento común de un conjunto de clases relacionadas.

Son clases que NO fueron pensadas para crear instancias de ellas sino exclusivamente para servir como superclase de otra.

Es posible definir constructores en las superclases, pero no es posible crear instancias, sólo se usan para que a través de la herencia las utilicen las subclases.

Métodos abstractos

- Está precedido por la palabra clave abstract.
- No tiene cuerpo y su encabezado termina con punto y coma (es decir, se especifica su nombre, parámetros y tipo de retorno, pero no incluye código).
- Un método se define como abstracto PORQUE EN ESE MOMENTO NO SE CONOCE COMO HA DE SER SU IMPLEMENTACIÓN, y serán las subclases de la clase abstracta las responsables de darle "cuerpo" mediante la sobrescritura del mismo.
- Si un método se declara como abstracto, se debe marcar la clase como abstracta → No puede haber métodos abstractos en una clase concreta.
- Los métodos abstractos deben implementarse en las subclases concretas (subclases) mediante @Override.
- Las subclases de una clase abstracta están obligadas a sobrescribir todos los MÉTODOS ABSTRACTOS que heredan.
- En caso de que no interese sobrescribir alguno de esos métodos, la subclase deberá ser declarada también abstracta.

```
public abstract class Figura {  
    private String color;  
    public Figura(String color){  
        this.color=color;  
    }  
    public String obtenerColor(){  
        return color;  
    }  
    public abstract double area();  
}
```



Implementa y sobrescribe el
método abstracto area()
(Polimorfismo)

```
//Circulo.java  
public class Circulo extends Figura{  
    private int radio;  
    public Circulo(int radio, String color){  
        super(color);  
        this.radio = radio;  
    }  
    public double area(){  
        return Math.PI*radio*radio;  
    }  
    public int obtenerRadio(){  
        return radio;  
    }  
}
```

```
public class Triangulo extends Figura{  
    private int base;  
    private int altura;  
    public Triangulo(int base,int altura,String color){  
        super(color);  
        this.base = base;  
        this.altura = altura;  
    }  
    public double area(){  
        return (base*altura)/2;  
    }  
    public int obtenerBase(){  
        return base;  
    }  
    public int obtenerAltura(){  
        return altura;  
    }  
}
```

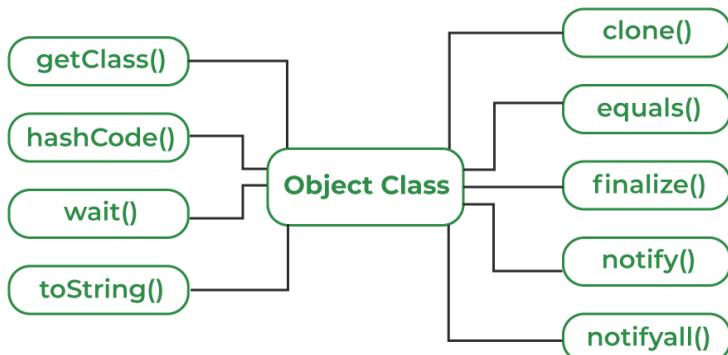


Implementación del método area()
aplicado en función a la fórmula para el
Triángulo (Polimorfismo)

Clase object

La clase Object es la CLASE PADRE DE TODAS LAS CLASES: todas las clases en Java directa o indirectamente heredan de la clase Object.

Define algunos métodos que están disponibles en todas las clases en Java:



Método equals

- La comparación que hace el equals no es la misma que proporciona el operador ==, que solamente compara si dos referencias a objetos apuntan al mismo objeto.
- El método equals() se utiliza para saber si dos objetos separados son del mismo tipo y contienen los mismos datos. Retorna true si los objetos son iguales y false en caso contrario.
- Las subclases pueden sobreescibir el método equals() para realizar la adecuada comparación entre dos objetos de un tipo que haya sido definido por el programador.

Cuando NO:

- Cada instancia de la clase es propiamente única.
- No hay necesidad para la clase de proveer una igualdad lógica.
- Una superclase ya implementó este método y el mismo es apropiado para la clase.
- La clase es privada y estamos seguros que el método equals nunca será invocado.

Cuando SI:

- Cuando una clase tiene una igualdad lógica que va más allá de la identidad del objeto. Y una superclase no haya implementado este método.

Precauciones:

Hay que comprobar que el objeto recibido no sea null.

Hay que comprobar que sean del mismo tipo con la sentencia "instanceof"

Cómo sobreescibir correctamente el método equals:

- Usar el operador == para chequear si el argumento es una referencia a este objeto. Si lo es retorna true.
 - Usar el operador instanceof para chequear si el argumento posee el tipo correcto. Si no, retorna false.
 - Castear el argumento al tipo correcto. Como el casteo se realiza después de un instanceof está garantizado que será satisfactorio.
 - Por cada atributo "significante" en la clase chequear si ese atributo es igual al atributo correspondiente de este objeto. Si todos estos son satisfactorios, retorna true. Caso contrario false.
- Para los atributos primitivos cuyo tipo no es float ni double usar el operador == para comparar.
- Para objetos, llamar al método equals recursivamente.
- Para atributos float usar el método estático compare de la clase Float. Float.compare(float1, float2)
- Para atributos double usar el método estático compare de la clase Double. Double.compare(double1, double2).

Método hashCode

Devuelve un código hash único para un objeto. El código hash es un número entero que se utiliza para identificar de manera única un objeto en una tabla hash.

- Cuando el método hashCode es invocado en un objeto repetidas veces debe retornar el mismo valor consistentemente.
- Si dos objetos son iguales de acuerdo al método equals, entonces hashCode debe retornar el mismo valor para ambos.
- Si dos objetos son distintos de acuerdo al método equals, no es necesario que hashCode produzca un valor distinto. Sin embargo esto puede mejorar la performance de una hash table.

Como hacer un método hashCode:

- Declarar una variable entera llamada resultado e inicializarla con el valor de hash code del primer atributo significante en nuestro objeto.
- Para cada atributo remanente en nuestro objeto hacer lo siguiente:
 - ◆ Si el atributo es de tipo primitivo computar Type.hashCode(atributo). Donde Type es el Tipo correspondiente a la primitiva.
 - ◆ Si el atributo es un objeto y esa clase hace uso de equals de manera recursiva para sus atributos. Usar hashCode recursivamente en el atributo. Si el valor del atributo es nulo usar 0.
 - ◆ Si el atributo es un array, tratarlo como si cada elemento fuera un atributo separado. Esto significa calcular un hashCode para cada elemento en el array y combinar los valores. Si el array no tiene elementos significantes usar una constante diferente de 0. Si TODOS los elementos son importantes usar Arrays.hashCode.
- Combinar el resultado de esta forma: resultado = 31* resultado + atributo;

Método toString

Es la manera de obtener la representación en cadena de un objeto.

Todas las clases implementan el método `toString()`, porque este método está definido en la clase `Object`, pero la implementación por omisión de `toString()` raramente es suficiente, para la mayoría de las clases creadas por el programador será deseable sobrescribir el método `toString()` y proporcionar nuestras propias representaciones en forma de cadena.

Para sobreescibir `toString()` basta simplemente con devolver un objeto `String` que contenga una cadena que describa apropiadamente al objeto de la clase.

```
public class Persona {  
  
    // Atributos  
    String nombre;  
    int edad;  
  
    //Constructor y otros metodos  
  
    @Override  
    public String toString() {  
        return "Persona [nombre=" + nombre + ", edad=" + edad + "]";  
    }  
}
```

Método getClass

Es un método final (no puede sobreescribirse).

Devuelve una representación en tiempo de ejecución de la clase del objeto. Devuelve un objeto Class al que se le puede pedir información sobre la clase, como su nombre, el nombre de su superclase y los nombres de las interfaces que implementa.

```
public void PrintClassName() {
    System.out.println("La clase del Objeto es " + getClass().getName());
}
```

Clase enum

La palabra reservada “enum” representa un tipo especial de clase que siempre extiende de java.lang.Enum y en la cual se restringe los posibles valores que puede tomar una variable.

Al restringir los posibles valores que puede tomar una variable:

- Documenta por adelantado la lista de valores aceptados.
- Hace el código más legible.
- Ayuda a reducir los errores en el código.
- Permite verificación en tiempo de compilación.
- Permite algunos usos especiales interesantes.
- Evita el comportamiento inesperado si es que se reciben valores no válidos.

```
public enum PizzaStatus {
    ORDERED,
    READY,
    DELIVERED;
}
```

Por convención, los nombres de los valores que puede tomar se escriben en letras mayúsculas, para recordarnos que son valores fijos.

Una vez declarado el tipo enumerado, todavía no existen variables hasta que no las creamos explícitamente:

```
TipoEnumerado nombreVariable; → PizzaStatus status;
```

Un tipo enumerado puede ser declarado dentro de una clase o como una clase aparte, pero NO dentro de un constructor ni cualquier otro método. Por tanto, no podemos declarar un enum dentro de un método main. Si lo hacemos, nos saltará el error de compilación “enum types must not be local” (los tipos enumerados no deben ser locales a un método).

Se dispone automáticamente de métodos especiales, como por ejemplo el método values(), que el compilador agrega automáticamente cuando se crea un enum. Este método devuelve un array conteniendo todos los valores del enumerado en el orden en que son declarados.

Un tipo enumerado puede añadir campos constantes al objeto enumerado y recuperar esos campos. Para ello se usa un constructor especial para tipos enumerados.

Usando el tipo enum nos aseguramos que sólo existe una instancia de la constante en la JVM, por lo tanto se puede usar el operador “==” para comparar dos variables del mismo tipo enum.

Seguridad en tiempo de ejecución:

- 1) if(pizza.getStatus().equals(Pizza.PizzaStatus.DELIVERED));
- 2) if(pizza.getStatus() == Pizza.PizzaStatus.DELIVERED);

En la opción 1) si el status de pizza es null, obtendremos una excepción del tipo NullPointerException. No así en la opción 2)

Interfaces

Una interfaz es un tipo de referencia similar a una clase abstracta que define un conjunto de MÉTODOS sin implementación y de CONSTANTES que una clase puede implementar.

La finalidad de las interfaces es definir comportamientos que una o varias clases necesitan implementar, definir el formato que deben de tener determinados métodos que han de implementar ciertas clases.

Normalmente una interfaz se compone de un conjunto de declaraciones de cabeceras de métodos (sin implementar, de forma similar a un método abstracto) que especifican un protocolo de comportamiento para una o varias clases. Pero también puede emplearse para declarar constantes que luego puedan ser utilizadas por otras clases.

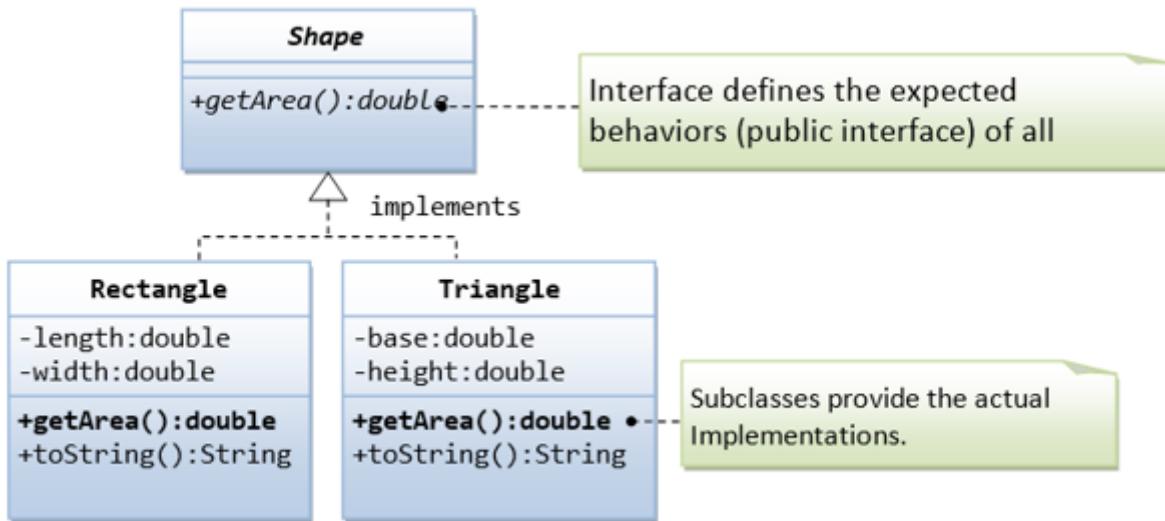
Una interfaz es similar a una clase abstracta llevada al límite, en la que todos sus métodos son abstractos.

La declaración de una interfaz es similar a una clase, aunque EMPLEA LA PALABRA RESERVADA "INTERFACE" EN LUGAR DE "CLASS", NO INCLUYE LA DECLARACIÓN DE VARIABLES, Y LOS MÉTODOS QUE SE DEFINEN EN LA INTERFAZ NO TIENEN CUERPO.

Las clases que implementan una interfaz están obligadas a proporcionar una implementación para todos los métodos definidos en la interfaz, o bien declararse como clase abstracta.

Para implementar una interfaz en una clase, se usa la palabra clave "implements" en la declaración de la clase, seguida del nombre de la interfaz.

UML de una interfaz:



Una clase abstracta pertenece a una jerarquía de clases mientras que una interfaz no pertenece a una jerarquía de clases. En consecuencia, clases sin relación de herencia pueden implementar la misma interfaz. LAS INTERFACES CAPTURAN SIMILITUDES ENTRE CLASES NO RELACIONADAS SIN FORZAR UNA RELACIÓN ENTRE ELLAS.

Por lo general, una interfaz se utiliza cuando clases no relacionadas necesitan compartir métodos y constantes comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos.

Una interfaz sólo puede tener métodos abstractos, no puede implementar ningún método.

Todos los métodos de una interfaz son abstractos y públicos de forma implícita, no hace falta declararlo así explícitamente.

Una interfaz no puede contener variables de instancia. No es una clase, y por ende no es posible crear objetos de una interfaz (no hace falta aclararlo con abstract, es implícito). Por ello, tampoco puede contener constructores. Sólo puede contener constantes y métodos sin implementar.

Una clase abstracta tampoco puede instanciarse, pero sí puede contener variables de instancia, constructores, constantes y métodos tanto concretos como abstractos, lo que permite que las subclases hereden y utilicen estos campos.

Interfaz como tipo de dato

La interfaz es un tipo de dato de referencia, por ende puede utilizarse como tipo de dato del objeto.

Cuando declaramos una interfaz, estamos definiendo un nuevo tipo de referencia.

Este tipo puede ser utilizado en cualquier lugar donde se esperaría un tipo de dato, incluyendo variables, parámetros de métodos y valores de retorno.

Ventajas:

- Se puede escribir el código de manera que opere con interfaces en lugar de con clases, en consecuencia, el código va a ser más flexible, porque en cualquier método, colección o lugar donde se espera recibir un objeto, no necesito usar un objeto de la misma clase u otra clase hija, puedo usar cualquier objeto de cualquier clase que implemente la interfaz, sin depender de la herencia.
- Se garantiza que cualquier objeto asignado a este tipo de referencia cumple con un contrato específico (es decir, implementa los métodos definidos por la interfaz).
- Este enfoque es una manifestación del principio de polimorfismo y permite que múltiples tipos de objetos que no pertenecen a una misma jerarquía de clases sean manejados de manera uniforme.

Ejemplo:

→ En este ejemplo, Músico puede ensayar con cualquier InstrumentoMusical, ya sea una Guitarra, un Piano, o cualquier otro instrumento que implemente la interfaz.

→ La referencia de tipo InstrumentoMusical garantiza que el método ensayar opere con cualquier instancia de clase que implemente InstrumentoMusical

```
interface InstrumentoMusical {
    void tocar();
}

class Guitarra implements InstrumentoMusical {
    public void tocar() {
        System.out.println("Tocando guitarra.");
    }
}

class Piano implements InstrumentoMusical {
    public void tocar() {
        System.out.println("Tocando piano.");
    }
}

class Musico {
    void ensayar(InstrumentoMusical instrumento) {
        instrumento.tocar();
    }
}
```

Las dos grandes ventajas de las interfaces:

- Compartir constantes y métodos en clases derivadas (con lo cual se aumenta la reutilización de código), pero sin el límite de la herencia simple.
- Usar las interfaces como tipos de variable (contenedores) para almacenar cualquier clase (siempre y cuando sea derivada) en una misma colección o tratarla de la misma manera.

Interfaz Comparable

La interfaz Comparable en Java es una forma de definir el orden natural de los objetos de una clase. Al implementar Comparable, una clase indica que sus instancias se pueden ordenar de acuerdo a un orden específico, generalmente en una secuencia ascendente o descendente.

La interfaz Comparable es fundamental para varias estructuras de datos de Java que dependen de un orden, como TreeSet y TreeMap, y para algoritmos de ordenación como Arrays.sort() y Collections.sort().

El método compareTo(Object o) ES EL ÚNICO MÉTODO DE LA INTERFAZ COMPARABLE. Un objeto a se compara con otro objeto b mediante a.compareTo(b).

El contrato de este método especifica que debe devolver lo siguiente:

- Entero negativo si el objeto es menor al especificado por parámetro.
- Cero si el objeto es igual al especificado por parámetro.
- Entero positivo si el objeto es mayor al especificado por parámetro

```
public interface Comparable<T> {  
    int compareTo(T t);  
}
```

ES IMPORTANTE QUE compareTo SEA CONSISTENTE CON EQUALS: si a.compareTo(b) == 0, entonces a.equals(b) debería retornar true.

Esto no es obligatorio, pero muchas clases y métodos en Java asumen esta consistencia.

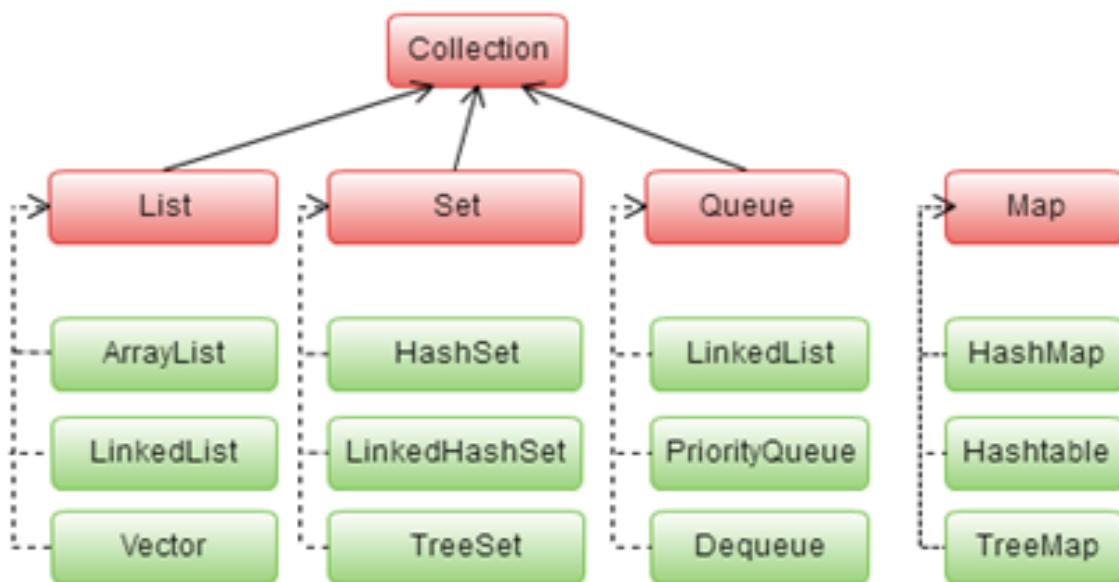
Ejemplo:

```
public class Telephone {  
    private short areaCode;  
    private short prefix;  
    private short lineNum;  
    ...  
    public int compareTo(Telephone tp) {  
        int result = Short.compare(areaCode, tp.areaCode);  
        if (result == 0) {  
            result = Short.compare(prefix, tp.prefix);  
            if (result == 0) result = Short.compare(lineNum, tp.lineNum);  
        }  
        return result;  
    }  
}
```

API collection

La API Collection es un conjunto de interfaces y clases que se utilizan para almacenar y manipular conjuntos de objetos en forma de colecciones.

En el siguiente gráfico, el color rosa representa interfaces y el color verde clases:



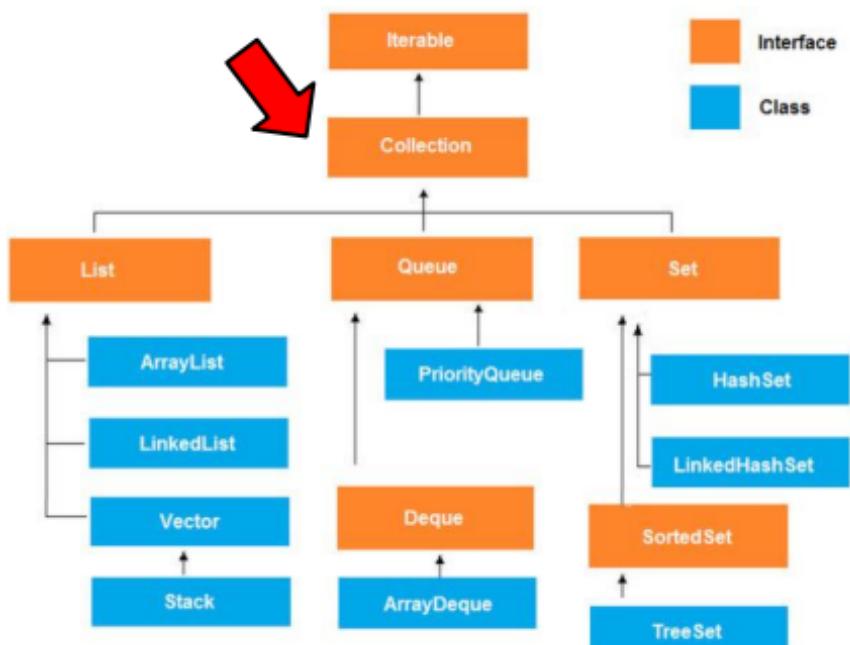
La API Collection proporciona una serie de métodos y operaciones comunes para manipular y acceder a los elementos de las colecciones, lo que permite manipularlas independientemente de los detalles de la implementación.

En consecuencia:

- Reduce los esfuerzos de programación al proveer estructuras de datos y algoritmos que no tenemos que escribir. Ofrece múltiples operaciones como búsqueda, inserción, eliminación y ordenación de elementos.
- Provee implementaciones de alta performance.
- Fomenta la reutilización del software.

Es la RAÍZ DE TODAS LAS INTERFACES Y CLASES relacionadas con colecciones de elementos SET, LIST y QUEUE:

Define operaciones básicas que obligatoriamente han de implementar las clases que implementan la interfaz Collection o cualquiera de las interfaces que hereden de Collection.



Interfaz collection

Es la manera más genérica para representar un grupo de elementos, permite una gran flexibilidad. Por ello, gracias a esta interfaz podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes para manipular cualquier colección de elementos.

Al ser una interfaz y no una clase, obviamente no se pueden construir objetos de tipo Collection.

Principales métodos de instancia:

- add(E e): Agrega un elemento a la colección.
- remove(Object o): Elimina de la lista la primera aparición del objeto especificado
- size(): Retorna el número de elementos en la colección.
- isEmpty(): Verifica si la colección está vacía.
- contains(Object o): Verifica si el objeto especificado está en la colección.
- iterator(): Retorna un iterador sobre los elementos en la colección.
- clear(): Elimina todos los elementos de la colección
- void addAll (Collection c)
- Object[] toArray()

Principales métodos estáticos:

- boolean addAll (Collection c, T .. elements)
- Collections.addAll (elementos, new CD(),new DVD());
- void copy (List listaDestino, List listaOrigen)
- int frequency (Collection c, Object obj) (*obj debe implementar equals())
- Object max (Collection c) (*los elementos tienen que implementar Comparable)
- Object min (Collection c) (*los elementos tienen que implementar Comparable)
- void reverse (List lista)

Interface List

La característica distintiva es que permite agrupar una colección de elementos en forma ORDENADA o SECUENCIAL (UNO DETRÁS DE OTRO), lo que permite el ACCESO PRECISO A SUS ELEMENTOS MEDIANTE UN ÍNDICE que representa su ubicación numérica dentro de la lista.

Además de los que hereda de la interfaz Collection (add, addAll, remove, clear, contains, size, isEmpty, etc), agrega métodos relacionados con el acceso posicional a los elementos:

- Add (int índice, E elemento): Inserta un elemento en la posición especificada por el índice, desplazando los elementos existentes hacia adelante.
- Object remove (int índice): Elimina el elemento en la posición especificada por el índice, desplazando hacia atrás los elementos subsiguientes.
- Object Get (int índice): Devuelve el elemento en la posición especificada por el índice.
- Object set (int index, Object element): Reemplaza el elemento en la posición index con element.
- int indexOf (Object objeto): Devuelve el índice de la primera aparición del objeto especificado en la lista, o -1 si no está en la lista.
- int lastIndexOf (Object objeto): Devuelve el índice de la última aparición del objeto especificado en la lista, o -1 si no está en la lista.
- subList (int inicio, int fin): Devuelve una vista de sublista de la lista original, que abarca desde el índice de inicio (inclusive) hasta el índice de fin (exclusive).
- listIterator(): Devuelve un ListIterator que permite la iteración bidireccional (hacia adelante y hacia atrás) y permite añadir, modificar y eliminar elementos mientras se itera.

ArrayList

Es una CLASE que permite almacenar datos de forma similar a un ARRAY pero de forma DINÁMICA, con lo cual hay dos ventajas:

- No es necesario declarar su tamaño inicial.
- Puede ir creciendo si es necesario.

Ventajas:

Las que ya posee por el hecho de implementar la Interfaz List → acceso directo a los elementos mediante índice → las operaciones de acceso a elementos, capacidad y saber si está vacío se realizan de forma eficiente y rápida.

Más los beneficios relacionados con que sea un arreglo DINÁMICO → No necesita definir dimensión inicial y redimensionamiento.

Desventajas:

Añadir elementos puede ser lento si se requiere redimensionar el array, especialmente si el array es grande, porque implica copiar todos los elementos al nuevo array.

Añadir y eliminar elementos en el medio de la lista es lento porque implica mover todos los elementos subsiguientes para mantener el orden.

No están sincronizados. Con lo cual, si múltiples hilos acceden a un mismo ArrayList concurrentemente, podríamos tener problemas en la consistencia de los datos. Por lo tanto, al usar ArrayList habrá que controlar la concurrencia de acceso.

Hay 2 tipos de formas declarar un arraylist:

- A) ArrayList<String> nombreArrayList = new ArrayList<String>();
- B) List<String> lista = new ArrayList<String>();

Insertar un elemento por índice: Agregar un nuevo “elemento en la posición 11”;

Solución:

Modificar un elemento por índice:

nombreArrayList.add(11, "Elemento 3");

```
nombres.set(1, "Carlos"); // Juan es reemplazado por Carlos
```

Iterar un ArrayList

Con un for

```
for(int i=0; i<arrayName.size(); i++) {  
    System.out.println(al.get(i));  
}
```

Con un for each

```
for (String nombre : lista) {  
    System.out.println(nombre);  
}
```

Con un iterador

```
ListIterator<String> iterador = nombres.listIterator();  
while (iterador.hasNext()) {  
    System.out.println(iterador.next());  
    // Puedes también moverte hacia atrás con iterador.previous()  
}
```

Vector

Es similar a un ArrayList pero sí está SINCRONIZADO.

El vector es sincronizado, mientras que ArrayList no lo es. Si no trabajamos en un entorno multihilos, conviene utilizar ArrayList.

La estructura de ambos está basada en un arreglo dinámico.

Ambos pueden crecer y reducirse en forma dinámica, sin embargo la forma en la que se redimensionan es diferente.

Formas de crearlo:

```
Vector vector = new Vector(20, 5);
```

- El vector se inicializa con una dimensión inicial de 20 elementos. Si rebasamos dicha dimensión y guardamos 21 elementos la dimensión del vector crece a 25.

```
Vector vector=new Vector(20);
```

- Si se rebasa la dimensión inicial guardando 21 elementos, la dimensión del vector se duplica.

```
Vector vector=new Vector();
```

- Se inicializa con dimensión inicial de 10 elementos. La dimensión del vector se duplica si se rebasa la dimensión inicial

Métodos:

- void addElement(Object obj)
- Object elementAt(int indice)
- void insertElementAt(Object obj, int indice)
- boolean removeElement(Object obj)
- void removeElementAt(int indice)
- void setElementAt(Object obj, int indice)
- int size()

Stack (pilas)

Representa una estructura LIFO (last in - first out).

Es una SUBCLASE DE VECTOR, por lo tanto también es sincronizada, y por lo tanto su estructura también está basada en un array.

Es un vector que ofrece métodos adicionales para trabajar con él como si fuese una pila.

- **push** → introduce un elemento en la pila.
- **pop** → saca un elemento de la pila.
- **peek** → consulta el primer elemento de la cima de la pila.
- **empty** → comprueba si la pila está vacía.
- **search** → busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella.

LinkedList

Su implementación se basa en una LISTA DOBLEMENTE ENLAZADA DE TAMAÑO ILIMITADO.

Al igual que ArrayList, también implementa la interfaz List, por lo tanto puede accederse a los elementos por posición.

Su estructura está formada por Nodos. Cada nodo contiene tres componentes: el dato, un enlace a su nodo anterior y otro enlace a su nodo siguiente. Esto permite recorrer la lista en ambos sentidos.

ArrayList esta basada en una estructura de datos del tipo arreglo, mientras que LinkedList esta basada en una lista doblemente vinculada.

0	1	2	3	4
23	3	17	9	42

¿Cuándo usar cada uno?



El ArrayList es más conveniente en aplicaciones que requieren búsqueda frecuente de elementos por su índice y en las cuales las adiciones y eliminaciones se realizan principalmente al final de la lista.

La LinkedList es más conveniente en aplicaciones que requieren inserciones y eliminaciones frecuentes.

ArrayList	<ul style="list-style-type: none">Muy rápida accediendo a los elementos.Se adapta a un gran número de escenarios.
LinkedList	<ul style="list-style-type: none">Listas enlazadas.Gran eficiencia agregando y eliminando elementos.
Vector	<ul style="list-style-type: none">Considerada como colección obsoleta.Utilizada únicamente en operaciones de concurrencia.
HashSet	<ul style="list-style-type: none">Rápida.No duplicados.No ordenacion.No acc. aleatorio.
LinkedHashSet	<ul style="list-style-type: none">Ordenación por entrada.Eficiente al acceder.No eficiente al agregar.
TreeSet	<ul style="list-style-type: none">Es ordenado.Poco eficiente.
HashMap	<ul style="list-style-type: none">No ordenacion.Eficiente.
LinkedHashMap	<ul style="list-style-type: none">Ordenado por inserción.Permite ordenacion por uso.Eficiente lectura.Poca eficiente lectura.
TreeMap	<ul style="list-style-type: none">Ordenado por clave.Poco eficiente en todas sus operaciones.
Hashtable	<ul style="list-style-type: none">Considerada como colección obsoleta.Utilizada únicamente en operaciones de concurrencia.

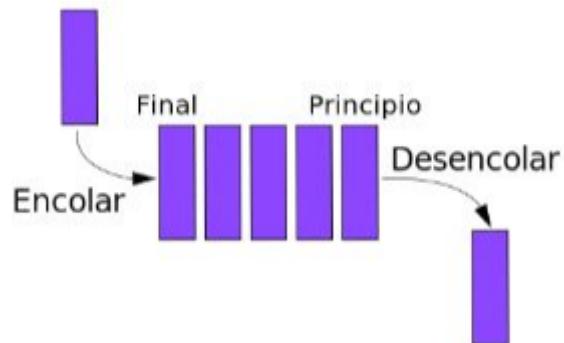
Colección Queue

Interface Queue

Representa al tipo Cola o FILA, que es una lista en la que sus elementos se introducen únicamente por un extremo (fin de la cola) y se remueven por el extremo contrario (principio de la cola).

Métodos:

- offer(E e): Inserta el elemento en la cola si es posible hacerlo inmediatamente sin violar las restricciones de capacidad.
- poll(): Recupera y elimina la cabeza de la cola, o devuelve null si esta está vacía.
- peek(): Recupera, pero no elimina, la cabeza de la cola, o devuelve null si esta está vacía



Interface Set

Se utiliza para representar una colección de elementos ÚNICOS Y NO ORDENADOS.

La interfaz Set es implementada en varias clases, como HASHSET, TREESET Y LINKEDHASHSET.

Elementos únicos: La interface Set hereda los métodos de Collection y agrega sus propias restricciones para prohibir el duplicado de elementos.

No ordenados: los elementos en un Set no tienen ningún orden en particular, lo que significa que no se puede garantizar el orden en el que se devolverán los elementos.

Acceso mediante iteración: los elementos en un Set se pueden acceder mediante un iterador. No se puede acceder a los elementos por índice, como en List.

si tenemos un objeto que tiene las mismas características (equals) y el mismo hashCode que algún objeto que ya se encuentra en el Set , no se lanza ninguna excepción, pero el elemento NO se agrega a la colección. Como el método add de la Interface Collection retorna true o false si agregó o no, en este caso nos sirve analizarlo: si el elemento a añadir no estén el conjunto y ha sido añadido retornará true, o false si el elemento aba ya se encontraba dentro del conjunto.

Para que Set funcione correctamente, es crucial que los objetos que se añaden implementen de manera adecuada los métodos equals y hashCode:

- equals(Object obj): Determina si dos objetos son iguales. Es usado por implementaciones de Set para asegurar que no se añadan duplicados.
- hashCode(): Proporciona el código hash de un objeto, que es usado por las implementaciones basadas en hash de Set (como HashSet) para almacenar objetos de manera que puedan ser recuperados rápidamente. Si dos objetos son considerados iguales según el método equals, entonces deben tener el mismo valor de código hash.

- Iterator iterator(): método que recorre los elementos de una colección.

```
Iterator it = hs.iterator();
while(it.hasNext())
{
    System.out.println(it.next());
}
```

Set hereda todos los métodos de la interfaz Collection (add, remove, contains, size, etc). No introduce nuevos métodos específicos. Su principal diferencia radica en cómo maneja la adición de elementos y la garantía de unicidad.

Cuando se comparan dos conjuntos para igualdad, se consideran iguales si contienen exactamente los mismos elementos, independientemente del orden.

Esto es diferente de las listas, donde el orden de los elementos también importa.

La igualdad entre dos Set se comprueba utilizando los métodos equals de los elementos contenidos, verificando que cada conjunto contenga todos los elementos del otro sin considerar el orden.

HashSet

→ Los objetos se almacenan en una TABLA DE DISPERSIÓN (HASH): Almacena los elementos mediante una función hash, que convierte el contenido del elemento en un código de hash utilizado para determinar en qué parte de la tabla se almacenará el elemento.

→ La clase HashSet delega casi todas sus funcionalidades a un mapa interno (HashMap).

→ Los elementos no se mostrarán necesariamente en el mismo orden que se insertaron.

La tabla hash hace que sea más eficiente que las clases que implementan la interfaz list. Sin embargo, la iteración a través de sus elementos es más costosa, ya que necesitará recorrer todas las entradas de la tabla de dispersión.

```
Creación: HashSet <String> hs = new HashSet <String>();
```

Agregar elementos:	hs.add ("Lionel"); hs.add ("Dibu");	Si el objeto a agregar ya estaba agregado, add retorna el valor anterior de esa key. Si el objeto a agregar NO estaba agregado, lo agrega y add retorna NULL.
Ejemplo:		

```
// Inicializar un HashSet
Set<String> miHashSet = new HashSet<>();

// Agregar elementos
miHashSet.add("Uno");
miHashSet.add("Dos");
miHashSet.add("Tres");

// Ver el tamaño del HashSet
System.out.println("Tamaño del HashSet: " + miHashSet.size());

// Eliminar un elemento
miHashSet.remove("Dos");

// Verificar si contiene un elemento
if (miHashSet.contains("Uno")) {
    System.out.println("El HashSet contiene el elemento 'Uno'");
}

// Recorrer el HashSet
Iterator<String> iterator = miHashSet.iterator();

while (iterator.hasNext()) {
    String elemento = iterator.next();
    System.out.println("Elemento: " + elemento);
}
```

LinkedHashSet

Es similar a HashSet, pero la tabla hash de dispersión se maneja con una lista doblemente enlazada, lo que hace que los elementos conserven el orden de inserción.

Los elementos que se inserten tendrán enlaces entre ellos. Por lo tanto, las operaciones básicas seguirán teniendo coste constante, con un ligero costo adicional por la carga extra de tener que gestionar los enlaces. Sin embargo, habrá una mejora en la iteración, ya que al establecerse enlaces entre los elementos no tendremos que recorrer todas las entradas de la tabla, el coste sólo estará en función del número de elementos insertados.

El orden de iteración será el mismo orden en el que se insertaron.

Ejemplo:

```
// Creación de un LinkedHashSet  
LinkedHashSet<String> capitales = new LinkedHashSet<>();  
  
// Añadir elementos  
capitales.add("Madrid");  
capitales.add("Londres");  
capitales.add("Nueva York");  
  
// Eliminar un elemento  
capitales.remove("Londres");  
  
// Imprimir el LinkedHashSet manteniendo el orden de inserción  
for (String capital : capitales) {  
    System.out.println(capital);  
}
```

TreeSet

```
// Creación de un TreeSet  
TreeSet<String> frutas = new TreeSet<>();  
  
// Añadir elementos  
frutas.add("Mango");  
frutas.add("Banana");  
frutas.add("Manzana");  
  
// Acceder al primer y último elemento  
String primeraFruta = frutas.first();  
String ultimaFruta = frutas.last();  
  
// Imprimir el TreeSet en orden natural  
for (String fruta : frutas) {  
    System.out.println(fruta);  
}
```

→ Almacena los elementos utilizando un árbol y ordenándolos en función de sus valores.

→ Los elementos a almacenar que pertenezcan a clases propias del sistema (String, etc) se ordenan de acuerdo a su orden natural, y a las clases propias se les deberá implementar la interfaz Comparable para que puedan ordenarse.

→ Es bastante más lento que HashSet. El TreeSet es útil cuando se necesita un ordenamiento constante de los elementos.

Métodos:

- object higher(object o): Devuelve el elemento menor de la colección, pero que sea mayor que el elemento dado. Devuelve NULL si no existe el elemento dado.
- object lower(object o): Devuelve el elemento mayor de la colección, pero que sea menor que el elemento dado. Devuelve NULL si no existe el elemento dado.
- object pollFirst(): Elimina el primer elemento de la colección. Devuelve NULL si está vacía.
- object pollLast(): Elimina el último elemento de la colección. Devuelve NULL si está vacía.

Interfaz Map

Representa una estructura de datos que asocia claves con valores, y ninguna clave puede repetirse dentro de un mismo mapa.

La clave funciona como un identificador único, y no se admiten claves duplicadas. Y para cada clave sólo habrá un valor, aunque diferentes claves pueden asociarse con el mismo valor.

Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que no heredan de la interfaz Collection.

Para iterar sobre un Map, se pueden obtener vistas de las claves, los valores o los pares clave-valor (entradas). Estas vistas son Sets para claves y entradas, y Collection para valores, lo que permite utilizar todas las operaciones de iteración proporcionadas por esas interfaces.

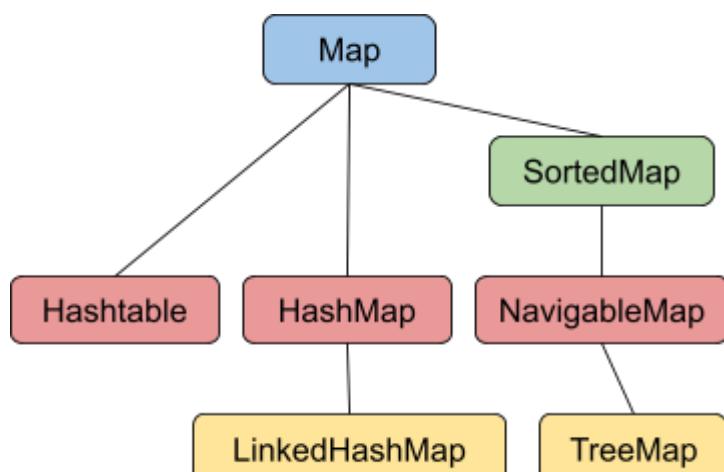
Como los valores se insertan, recuperan o eliminan usando la clave correspondiente, las operaciones de acceso son muy rápidas dependiendo de la implementación del map.

Métodos:

- Object put (K key, V value): Inserta el par clave-valor en el mapa. Si la clave ya existe, el valor anterior se reemplaza por el valor especificado.
- Object get (Object key): Devuelve el valor asociado con la clave especificada, o null si el mapa no contiene la clave.
- Object remove (Object value)
- Object containsKey (Object key): Devuelve true si el mapa contiene la clave especificada.
- boolean containsValue (Object value): Devuelve true si el mapa mapea una o más claves a este valor.
- Set keySet (): Devuelve un Set de las claves contenidas en este mapa.
- values (): Devuelve una Collection de los valores contenidos en este mapa.
- entrySet (): Devuelve un Set de las entradas (pares clave-valor) contenidas en este mapa
- boolean isEmpty ()
- int size ()

Cómo recorrerlo:

```
Iterator it = hm.entrySet().iterator();
while(it.hasNext())
{
    Map.Entry me = (Map.Entry) it.next();
    System.out.println(me.getKey()+"-"+me.getValues());
}
```



HashMap

- Los objetos se almacenan utilizando una tabla de dispersión (hash), sin orden.
- Hashing o dispersión: técnica de organización de archivos en la cual se almacenan registros en una dirección del archivo que es generada por una función que se aplica a la clave del mismo. Otorga alto rendimiento en operaciones básicas como inserción, eliminación y acceso.
- Sólo se admite una clave null.

```
// Inicializar un HashMap
Map<String, Integer> miHashMap = new HashMap<>();

// Agregar elementos
miHashMap.put("Uno", 1);
miHashMap.put("Dos", 2);
miHashMap.put("Tres", 3);

// Ver el tamaño del HashMap
System.out.println("Tamaño del HashMap: " + miHashMap.size());

// Eliminar un elemento
miHashMap.remove("Dos");

// Verificar si contiene un elemento
if (miHashMap.containsKey("Uno")) {
    System.out.println("El HashMap contiene la clave 'Uno'");
}

// Recorrer el HashMap
for (Map.Entry<String, Integer> entry : miHashMap.entrySet()) {
    System.out.println("Clave: " + entry.getKey() + ", Valor: " + entry.getValue());
}
```

Hashtable

Es similar a HashMap pero sincronizado. Además no admite claves null.

LinkedHashMap

Similar a HashMap pero con la diferencia que mantiene una LISTA DOBLEMENTE VINCULADA, además de la tabla Hash.

Al usar una lista doblemente vinculada, los elementos quedan agregados en orden de inserción. Permite iteraciones predecibles y ordenadas sin sacrificar el rendimiento significativo de HashMap.

```
Map<String, String> diccionario = new LinkedHashMap<>();
diccionario.put("brillante", "Que brilla.");
diccionario.put("efímero", "Que dura poco tiempo.");
diccionario.put("sutil", "Delicado, tenué.");

for (String key : diccionario.keySet()) {
    System.out.println(key + ": " + diccionario.get(key));
}
// Imprime las entradas en el orden en que fueron insertadas.
```

TreeMap

- Utiliza un ÁRBOL BINARIO equilibrado para implementar el mapa.
 - Permite tener un mapa ORDENADO. La iteración será en orden ascendente según sus keys.
 - TreeMap ordenará todos los valores “en el orden natural” de las claves o mediante un comparador, de forma ascendente, si trabajamos con objetos tipo String, Integer, etc. Si usamos como key clases creadas por el programador, habrá que implementar en ellas la interfaz Comparable.
 - El coste de las operaciones básicas será logarítmico con el número de elementos del mapa $O(\log n)$.

Métodos:

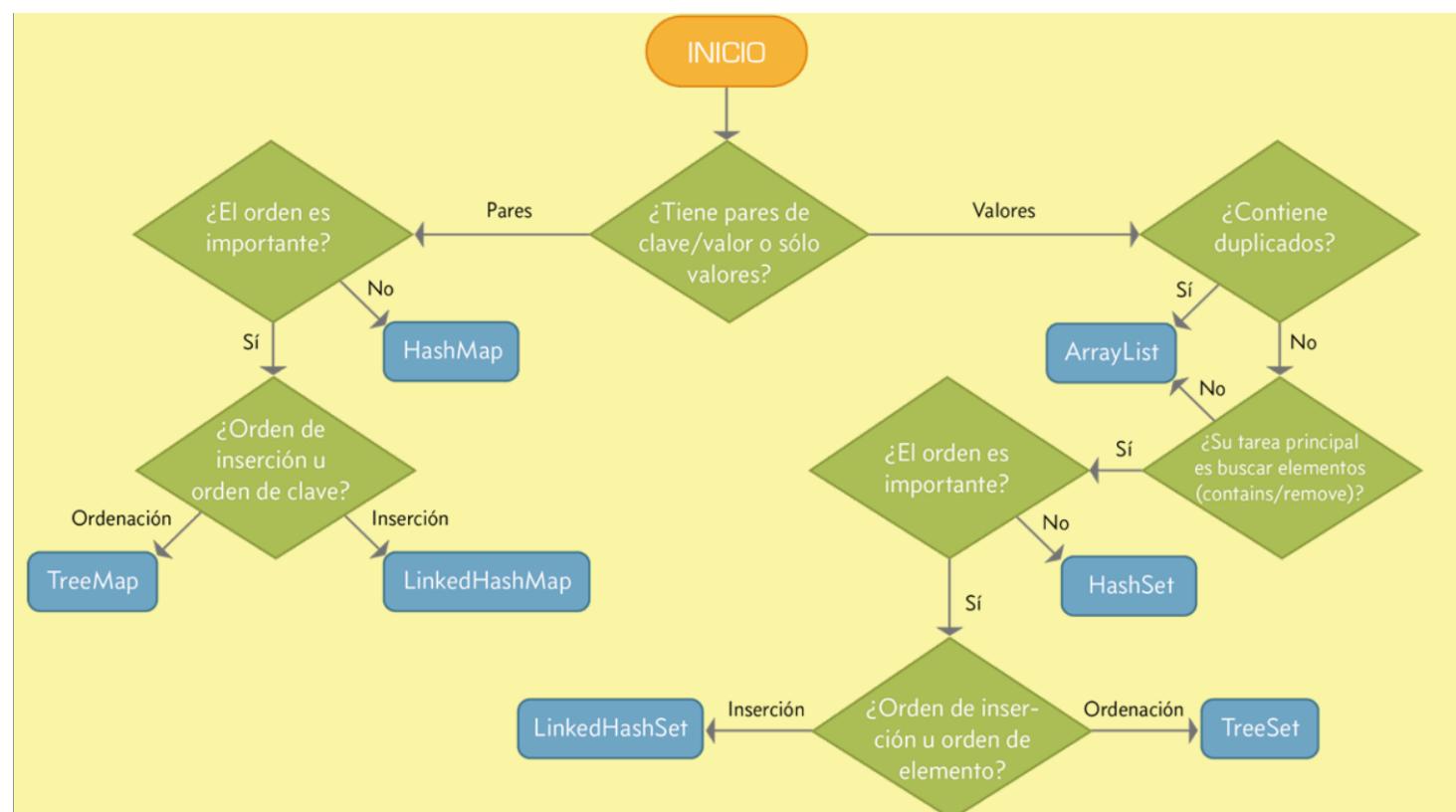
- Object ceilingKey()
 - Object firstKey()
 - Object lastKey()
 - Object lowerKey(object key)
 - Object higherKey(object key)

```
Map<String, Integer> libros = new TreeMap<>();
libros.put("Cien años de soledad", 30);
libros.put("Don Quijote", 15);
libros.put("Hamlet", 20);

for (Map.Entry<String, Integer> entrada : libros.entrySet()) {
    System.out.println(entrada.getKey() + ": " + entrada.getValue());
}
// Imprime las entradas en orden alfabético de las claves.
```

Conclusión:

- La principal diferencia entre uno y otro es la sincronización: HashTable es el único sincronizado, HashMap y TreeMap no. Por ello, para aplicaciones multihilos es preferible elegir Hashtable.
 - HashMap es mejor en cuanto a performance.
 - TreeMap es el único ordenado, HashMap y HashTable no lo son.
 - Hashtable no admite valores nulos en ninguna de sus partes, mientras que HashMap permite tener una clave null y un valor null.



Excepciones

Durante la ejecución del programa pueden producirse errores o situaciones excepcionales que interrumpen el flujo normal del programa.

Generalmente, ante estos errores, el programa se cierra. El manejo de excepciones es el mecanismo previsto por Java para permitir tratar estos errores: capturarlos, recuperarnos, mostrar un mensaje y decidir si continuamos o no.

En Java, las excepciones son objetos que encapsulan la información del error ocurrido. Convierten los errores que un método puede arrojar en una parte explícita del contrato del método.

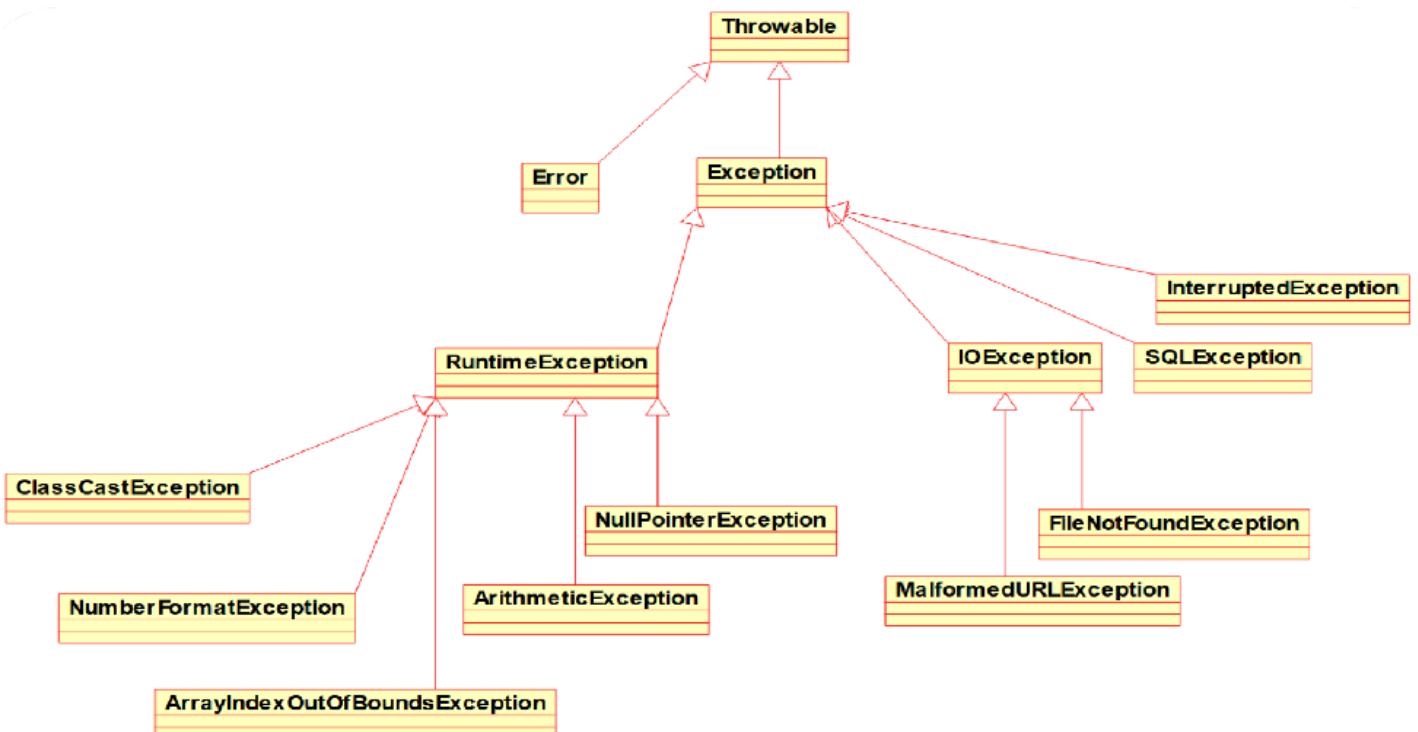
Separar el código que gestiona los errores del código principal del programa

Nos permite GESTIONAR el error y continuar con la ejecución del programa

Agrupar y diferenciar entre diferentes tipos de errores

Propagar errores hacia arriba en la pila de llamadas (StackTrace)

Las excepciones también son OBJETOS. Todos los tipos de excepción deben extender de la clase base Throwable (la superclase de todas las clases de errores y excepciones en Java) o de alguna de sus subclases. De Throwable nacen dos grandes ramas: Error y Exception.



Tipos de excepciones

ERROR: Representa aquellos errores que son irrecuperables, errores de una magnitud tal que una aplicación nunca debería intentar realizar nada con ellos. Generalmente están relacionados con problemas del entorno de ejecución de la aplicación, como quedarse sin memoria (`OutOfMemoryError`), errores de la JVM, desbordamiento de buffer.

EXCEPTION Representa aquellos errores que sí deberíamos tratar de gestionar. De `Exception` derivan clases de las que se instancian objetos para ser lanzados y luego capturados por los `catch`. Por convenio, los nuevos tipos de excepción extienden a `Exception`.

De `Exception` nacen múltiples ramas, que a su vez pueden clasificarse en excepciones checked y unchecked.

RuntimetypeException

→ Pertenecen a la clase `EXCEPTION`

→ Son excepciones que se propagan automáticamente sin necesidad de especificarlas en la cabecera de los métodos.

Excepciones checked

→ Una excepción de tipo checked representa un error del cual técnicamente podemos recuperarnos.

→ Son todas las SITUACIONES que son totalmente AJENAS AL PROPIO CÓDIGO, por ejemplo: fallo de una operación de lectura/escritura.

→ El compilador obliga a que DEBAN SER MANEJADAS EXPLÍCITAMENTE POR EL PROGRAMA (capturadas o relanzadas por el propio método para que quien lo invoque sí las capture). Si no compilará.

Excepciones unchecked

→ Representan ERRORES DE PROGRAMACIÓN.

→ No necesitan ser declaradas en un método o constructor mediante una cláusula `throws`, y el compilador no requiere que sean capturadas.

Métodos

→ `try`: es un bloque para detectar excepciones.

→ `catch`: es un manejador para capturar excepciones de los bloques `try`.

→ `throw`: es una expresión para lanzar (`throw`) una excepción en el momento que detecta el error.

→ `Throws`: indica las excepciones que puede elevar un método.

→ `Finally`: es un bloque opcional situado después de los `catch` de un `try`.

Bloque try:

El bloque `try` es donde se coloca el código que puede causar una excepción.

Este bloque intenta ejecutar dentro de él el código que es susceptible a errores. Si ocurre un error dentro del bloque `try`, la ejecución del código dentro de este bloque se detiene inmediatamente, y el control se pasa al bloque `catch` correspondiente que puede manejar ese tipo específico de excepción.

Bloque catch:

El bloque `catch` captura y maneja las excepciones especificadas. Es usado para definir una respuesta a diversas situaciones de error, permitiendo que el programa continúe su ejecución de una manera controlada o que proporcione una salida de error más comprensible para el usuario.

Puede haber múltiples bloques `catch` asociados a un solo `try`, permitiendo manejar diferentes tipos de excepciones de manera específica. Cada `catch` es diseñado para capturar un tipo de excepción en particular y contiene el código que determina cómo responder a esa situación.

Bloque finally:

Contiene código que debe ejecutarse después de que los bloques `try` y `catch` hayan terminado su ejecución, independientemente de si se lanzó una excepción o no.

Generalmente se usa para limpiar el estado interno o para liberar recursos, independientemente de si las operaciones fueron exitosas o no.

Ejemplo con varios catch:

```
try {  
String valor = JOptionPane.showInputDialog(null, "Ingresar un entero:");  
// Un valor no numérico lanzará un NumberFormatException  
int divisor = Integer.parseInt(valor);  
// Si la división es 0, esto resultará un ArithmeticException  
System.out.println(10 / divisor);  
} catch (NumberFormatException nfe) {  
[ acá manejamos el error NumberFormatException ...]  
} catch (ArithmeticException ae) {  
[acá manejamos el error ArithmeticException ...]  
}
```

Lanzamiento de excepciones

- Las excepciones se lanzan utilizando la palabra reservada THROW: throw AException
- AException debe ser un objeto Throwable, ya sea una excepción predefinida en Java o una excepción personalizada creada por el programador.
- Las excepciones son objetos, por lo tanto se debe crear una instancia antes de lanzarse.
- Las excepciones deben lanzarse en situaciones donde el método no puede completar su tarea de manera normal debido a circunstancias excepcionales. Si se lanza la excepción, no se regresa al flujo normal del programa.

Manejo de datos

- Las sentencias dentro de la cláusula try se ejecutan hasta que se lance una excepción o hasta que finalicen con éxito las instrucciones del try.
- Si se lanza una excepción, las acciones que hubiera detrás del punto donde se produjo la excepción no tienen lugar, y pasarán a examinarse sucesivamente las sentencias contenidas dentro de cada cláusula catch (en orden), y por último el bloque finally.

```
public static void main(String[] args) {  
    FileWriter fichero = null;  
    try {  
        fichero = new FileWriter(PATH_ARCHIVO);  
        fichero.write("Escribiendo...")  
    } catch(IOException e) {  
        e.printStackTrace();  
    } finally {  
        if (fichero != null) {  
            fichero.close();  
        }  
    }  
}
```

Propagación

Los métodos deben:

Capturar todas las excepciones chequeadas que puedan ser lanzadas dentro de su ámbito, o propagarlas hacia arriba en la pila de llamadas hasta que se encuentren con un bloque catch adecuado para manejarlas. Esto es especialmente útil en aplicaciones grandes, donde un error en un nivel bajo puede necesitar ser tratado en un nivel más alto de manera específica. La capacidad de propagar excepciones permite a los desarrolladores decidir si un método debe manejar un error directamente o pasarlo a su llamador para que lo maneje, basándose en el contexto de la aplicación.

Si al ocurrir un error está activa una porción de código denominado manejador de excepción, entonces el flujo de control se transfiere al manejador.

Si no existe un manejador para la excepción, ésta se propaga al método que invoca.

Si en éste tampoco se capta, la excepción se propaga al que a su vez le llamó.

Si llega al método por el que empieza la ejecución, es decir, main(), y tampoco es captada, la ejecución termina.

Throws

Throws es la palabra reservada que se utiliza en la firma de un método para declarar una o varias excepciones checked que puede lanzar el método. Es un aviso de que en el cuerpo del método hay por lo menos una sentencia que lanza una excepción checked.

Se pueden declarar varias, separadas por comas.

RuntimeException y Error son las únicas excepciones que no hace falta incluir en las cláusulas throws.

Hay muchos ejemplos (por ejemplo, los métodos de lectura de archivos). El IDE nos avisará del mismo para que lo encerremos en un bloque try-catch.

Si se invoca a un método que tiene una excepción checked en su cláusula throws, existen tres opciones:

- 1) Capturar la excepción y gestionarla.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}  
  
public void leerArchivo(String path) {  
    try {  
        abrirArchivo(path);  
    } catch(IOException e) {  
        System.out.println("Se produjo un error al abrir el archivo").  
    }  
}
```

- 2) Capturar la excepción y transformarla en una de nuestras excepciones.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}  
  
public void leerArchivo(String path) {  
    try {  
        abrirArchivo(path);  
    } catch(IOException e) {  
        throw new ArchivoNoExisteException(e);  
    }  
}
```

- 3) Declarar la excepción en la cláusula throws y hacer que otro método la gestione.

```
public void abrirArchivo(String path) throws IOException {  
    ...  
}  
  
public void leerArchivo(String path) throws IOException {  
    abrirArchivo(path);  
    ...  
}
```

No se permite que los métodos redefinidos declaren más excepciones checked en la cláusula throws que las que declara el método.

Se pueden lanzar subtipos de las excepciones declaradas, ya que podrán ser capturadas en el bloque catch correspondiente a su supertipo.

Crear nuestras propias excepciones

Además de las clases de excepciones que define Java, se pueden definir nuevas excepciones personalizadas para que las aplicaciones tengan control específico de errores, definiendo acciones concretas ante errores particulares sin tener que depender de excepciones más genéricas que podrían no ser tan descriptivas sobre la naturaleza del error.

- Aquellas que se definan, tienen que heredar de la clase base Exception.
- Normalmente, se definen varios constructores para proporcionar flexibilidad en cómo se puede lanzar la excepción, pasando mensajes de error o causas
- Cada excepción en Java puede encapsular información sobre el contexto en el que ocurrió el error, incluyendo un mensaje descriptivo y un rastro de pila (stack trace) que indica la secuencia de llamadas que llevó al error. Esto es crucial para depurar y resolver problemas en el software.

```
public class WrongPasswordException extends Exception{  
  
    public WrongPasswordException(String message) {  
        super(message);  
    }  
  
    @Override  
    public String getMessage() {  
        return "Ocurrio un error de validacion con su contrasenia: " + super.getMessage();  
    }  
}
```

Crear el método que puede llegar a lanzar esa excepción:

```
public class Authentication {  
    private String username;  
    private String password;  
  
    public Authentication(String username, String password) {  
        this.username = username;  
        this.password = password;  
    }  
  
    public boolean login(String password) throws WrongPasswordException {  
        if (!password.equals(this.password)) {  
            throw new WrongPasswordException("Contrasenia incorrecta.");  
        }  
        return true;  
    }  
}
```

Envolver el método que contiene el posible error a manejar en un bloque try/catch:

```
public static void main(String[] args) {
    Scanner teclado = new Scanner(System.in);
    boolean flag = false;

    Authentication user = new Authentication("usuario123", "1234");

    System.out.println("Ingrese su contraseña");
    String password = teclado.nextLine();

    try {

        flag = user.login(password);

    } catch (WrongPasswordException e) {
        System.out.println(e.getMessage());
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Buenas prácticas al manejar excepciones

- Capturar las excepciones más específicas antes de las más generales.
- Documentar las excepciones.
- Usar excepciones comprobadas.
- Lanzar excepciones específicas.
- Lanzar excepciones de acuerdo al nivel de abstracción en el que nos encontramos.
- Cerrar los recursos adecuadamente.
- Utilizar el constructor que acepta una causa es útil para el encadenamiento de excepciones, donde un throwable es causado por otro. Esto es especialmente útil en situaciones donde se captura una excepción y se quiere lanzar una diferente sin perder el rastreo original.
- Al crear excepciones personalizadas, extender Exception generalmente es suficiente.

Malas prácticas al manejar excepciones

- Ignorar las excepciones o capturarlas sin hacer nada.
- Capturar excepciones demasiado generales.
- Usar excepciones para controlar el flujo del programa. Es ineficiente y hace el código más difícil de leer y mantener. Las excepciones están diseñadas para manejar errores, no para controlar el flujo de ejecución en condiciones normales.
- Aunque es técnicamente posible, generalmente no se recomienda capturar y manejar objetos de tipo Error porque representan problemas graves como fallas de la JVM que una aplicación no debería intentar manejar.
- Lanzar excepciones no comprobadas.
- No cerrar los recursos.

Genericidad

La genericidad permite definir una CLASE, una INTERFAZ o un MÉTODO sin especificar el tipo de datos o parámetros de uno o más de sus miembros. Las clases, interfaces y métodos genéricos pueden trabajar con cualquier tipo de objeto, lo que los hace extremadamente reutilizables.

No es exclusiva de los lenguajes orientados a objetos, pero es en ellos en los que ha adquirido verdadera importancia y uso.

Casos de uso

- CONTENEDORES cuyas operaciones no dependen del tipo de datos almacenado.

Por ejemplo, un ArrayList o una Stack en Java puede almacenar cualquier tipo de objetos (Integer, String, etc, o cualquier objeto definido por el usuario), simplemente especificando el tipo de dato al momento de su creación.

- ALGORITMOS y operaciones aplicables a cualquier dato, independientemente de su tipo.

Los algoritmos de resolución de numerosos problemas no dependen del tipo de datos que procesa. Por ejemplo, la ordenación.

Ventajas

- Reutilización de código. Las clases y métodos genéricos pueden trabajar con cualquier tipo de objeto, lo que los hace extremadamente reutilizables
- Evita el casteo de clases , y con ello mejora el rendimiento: El casteo requiere que la JVM haga chequeo de tipos en tiempo de ejecución, en caso de que sea necesario ejecutar una ClassCastException. Con genéricos se evita el chequeo de tipos en tiempo de ejecución.
- Código más seguro, porque el chequeo de tipos se hace en tiempo de compilación. La genericidad ayuda a detectar y prevenir errores porque fuerza a verificar el tipo en tiempo de compilación. Reduce la posibilidad de fallos en tiempo de ejecución porque muchos problemas que podrían surgir de malas asignaciones o casteos de tipo inapropiadas se detectan antes de que el programa se ejecute.
- Legibilidad y mantenibilidad del código. Se pueden crear clases e interfaces más limpias y fáciles de entender.

Desventajas

- No se puede instanciar una clase genérica con tipos de datos primitivos.
- No se pueden crear objetos ni arreglos del tipo genérico.

Tipos genéricos

Ejemplo de tipo genérico:

```
public class Box<T> {  
    private T t; // T stands for "Type"  
  
    public void set(T t) {  
        this.t = t;  
    }  
  
    public T get() {  
        return t;  
    }  
}
```

Convención

Podría designarse cualquier nombre para un genérico en Java, pero para mejorar la legibilidad e interpretación del código existen convenciones para los nombres de estos tipos:

- **E** → elemento de una colección
- **K** → clave
- **N** → número
- **T** → tipo
- **V** → valor
- **S, U, V, etc.** → usado para representar otros tipos.

Restricciones

- LOS GENÉRICOS EN JAVA SÓLO PUEDEN SER INSTANCIADOS CON CLASES Y NO CON TIPOS PRIMITIVOS (como int, double, etc.).
- NO SE PUEDEN CREAR OBJETOS NI ARREGLOS DEL TIPO GENÉRICO.
- Una declaración de tipos genéricos puede tener múltiples tipos parametrizados, separados por comas, dentro de los < >. Por ejemplo:

```
public class Pair<T1, T2> { }
```
- Todas las invocaciones de clases genéricas son expresiones de una clase. Al instanciar una clase genérica no se crea una nueva clase.
- No se puede usar el tipo genérico como tipo de un campo estático o en cualquier lugar dentro de un método estático o inicializador estático.
- Dentro de una definición de clases, el tipo genérico puede aparecer en cualquier declaración no estática donde se podría utilizar cualquier tipo de datos concreto.

Declaración de una clase genérica:

```
public class MiClase<T> {

    private List<T> lista;

    public MiClase() {
        this.lista = new ArrayList<>();
    }

    public void imprimirLista() {
        for (T elemento : lista) {
            System.out.print(elemento + " ");
        }
        System.out.println();
    }

    public void agregarElemento(T elemento) {
        lista.add(elemento);
    }
}
```

Y su implementación en el main:

```
MiClase<Integer> miClaseNumeros = new MiClase<>();
miClaseNumeros.imprimirLista();
miClaseNumeros.agregarElemento(4);
miClaseNumeros.imprimirLista();

MiClase<String> miClasePalabras = new MiClase<>();
miClasePalabras.imprimirLista();
miClasePalabras.agregarElemento("!");
miClasePalabras.imprimirLista();
```

Métodos genéricos

Un método genérico o plantilla es un método que puede invocarse con una variedad de tipos de objetos diferentes, más allá de los vinculados a la clase en la que están definidos.

Un método genérico puede definirse dentro de una clase genérica o dentro de una clase ordinaria, ya que la declaración del tipo es específica para el método y no afecta a la clase entera.

Ejemplo:

```
public <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
}
```

Interfaces genéricas

Una interfaz genérica define un conjunto de métodos que operan en tipos de datos genéricos.

No se especifica con qué tipo de objeto concreto trabajarán los métodos; en su lugar, se utilizan parámetros de tipo que se sustituirán por tipos reales cuando una clase implemente esta interfaz.

Ejemplo:

```
public interface MyCollection <T>{
    void add(T objeto);
    T get();
    void sort();
}
```

```
//Implementacion Directa
public MyClass Implements
MyCollection<ClaseConcreta>{
.... //Implementaciones
}
```

```
public MyClass <T> Implements MyCollection<T>{
    void add(T object){
        ....
    }
    T get(){
        ....
    }
    void sort(){
        ....
    }
}
```

Limitaciones

- Los parámetros de tipo acotado permiten limitar los tipos que pueden ser aceptados por un parámetro de tipo en una clase o método genérico. Es decir: limitar los tipos con los que se puede parametrizar nuestra clase.
- Esto se hace especificando una clase o interfaz superior en la jerarquía de la cual el tipo de argumento debe ser una subclase o una implementación.
- Para definir un parámetro de tipo acotado, se utiliza la palabra clave extends en la declaración del parámetro de tipo (ya sean clases o interfaces).
- La limitación al tipo genérico también puede ser múltiple.
- En una limitación múltiple, los tipos pueden ser:
 - a) sólo una clase, la cual debe ser la primera, o
 - b) interfaces, las que se deseen, separadas por el ampersand.

Herencia en genericidad

- Una clase puede heredar de una clase genérica.
- La nueva clase puede
 - A) Mantener la genericidad de la clase padre
 - public class CajaSeguridad extends Contenedor
 - B) Restringir la genericidad.
 - public class CajaSeguridad <T extends Valorable> extends Contenedor
 - C) No ser genérica y especificar un tipo concreto.
 - public class CajaSeguridad extends Contenedor<Valorable>

Comodines

Los comodines en Java se utilizan con tipos genéricos para indicar incertidumbre sobre el tipo de los elementos en una colección.

Hay dos principales formas de comodines:

- ? extends Type

Se usa para declarar que el tipo desconocido representa un tipo que es una subclase de Type, incluido Type mismo. Esto se conoce como un límite superior.

```
public void printList(List<? extends Number> list) {  
    for (Number n : list) {  
        System.out.println(n);  
    }  
}
```

- ? super Type

Se usa para declarar que el tipo desconocido representa un tipo que es una superclase de Type, incluido Type mismo. Esto se conoce como un límite inferior.

```
public void addNumbers(List<? super Integer> list) {  
    list.add(123);  
}
```

Archivos

Un archivo (o fichero) es un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica.

¿Por qué usar archivos? Se usan cuando el volumen de datos no es muy elevado y tenemos que guardar los datos de nuestro programa para poderlos recuperar más adelante.

Suelen organizarse en estructuras jerárquicas de directorios. Estos directorios son contenedores de ficheros (y de otros directorios) que permiten organizar los datos del disco.

El nombre de un archivo está relacionado con su posición en el árbol de directorios que lo contiene, lo que permite no sólo identificar únicamente cada fichero, sino encontrarlo en el disco a partir de su nombre.

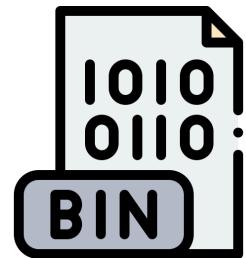
Archivos de texto

- Los datos se almacenan como caracteres ASCII o Unicode.
- Son archivos que podremos crear desde un programa Java y leer con cualquier editor de texto, o bien crear con un editor de textos y leer desde un programa Java.
- Los archivos de texto son adecuados para almacenar y procesar datos que se pueden leer y editar fácilmente.



Archivos binarios

- Los datos se almacenan en su forma binaria original (como una secuencia de bits)
- Son más eficientes en el almacenamiento y procesamiento de datos que los archivos de texto.
- Son adecuados para almacenar y procesar datos que no están diseñados para ser legibles por humanos, como imágenes, audio, video y cualquier otro tipo de archivo que requiera ser procesado por un programa.



Operaciones con archivos de texto

Para manipular archivos, siempre tendremos los mismos pasos:

- 1) Abrir el archivo → Si no abrimos el archivo, obtendremos un mensaje de error al intentar acceder a su contenido.
- 2) Escribir datos o leerlos.
- 3) Cerrar el archivo → Si no cerramos el archivo, puede que realmente no se llegue a guardar ningún dato

Clases propias de archivos de texto

- File: El objeto de la clase File contiene el nombre del archivo, la ruta y más propiedades relativas a él. Esta clase define métodos para conocer propiedades del archivo, como la última modificación, permisos de acceso, tamaño, etc.
- PrintWriter : Es una clase que permite escribir texto en un archivo de texto.
- FileWriter : Es una clase que permite escribir caracteres en un archivo de texto.
- BufferedReader : es una clase que permite leer texto de un archivo de texto.
- FileReader : es una clase que permite leer caracteres de un archivo de texto.

Crear un archivo:

```
public static void crearArchivo(String nombreArchivo) {  
    File file = new File(nombreArchivo);  
  
    try {  
  
        PrintWriter salida = new PrintWriter(file);  
        salida.close();  
  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
  
    System.out.println("El archivo se ha creado correctamente");  
}
```

Sobreescribir un archivo

```
public static void escribirArchivo(String nombreArchivo, String contenido) {  
    File file = new File(nombreArchivo);  
  
    try {  
  
        PrintWriter salida = new PrintWriter(file);  
        salida.println(contenido);  
        salida.close();  
        System.out.println("El archivo se ha escrito correctamente");  
  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
}
```

Agregar información a un archivo:

```
public static void agregarInformacion(String nombreArchivo, String contenido) {  
    File file = new File(nombreArchivo);  
  
    try {  
  
        PrintWriter salida = new PrintWriter(new FileWriter(file, true));  
        salida.println(contenido);  
        salida.close();  
        System.out.println("El archivo se ha escrito correctamente");  
  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Leer archivo:

```
public static void leerArchivo(String nombreArchivo) {  
    File file = new File(nombreArchivo);  
  
    try {  
  
        BufferedReader entrada = new BufferedReader(new FileReader(file));  
        String lineaActual = entrada.readLine();  
        while (lineaActual != null) {  
            System.out.println(lineaActual);  
            lineaActual = entrada.readLine();  
        }  
        entrada.close();  
  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Borrar contenido del archivo:

```
public static void borrarContenidoArchivo(String nombreArchivo) {  
  
    File file = new File(nombreArchivo);  
  
    try {  
  
        FileWriter salida = new FileWriter(file, false);  
        salida.write("");  
        salida.close();  
        System.out.println("El archivo se ha limpiado correctamente");  
  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Eliminar archivo:

```
public static void eliminarArchivo(String nombreArchivo) {  
  
    File file = new File(nombreArchivo);  
  
    try {  
  
        if (file.delete()) {  
            System.out.println("Archivo eliminado con éxito.");  
        } else {  
            System.out.println("No se pudo eliminar el archivo.");  
        }  
  
    } catch (Exception e) {  
        System.out.println("Ocurrió un error al intentar eliminar el archivo.");  
        e.printStackTrace();  
    }  
}
```

JSON

Es un formato de intercambio de datos que se utiliza para transmitir información estructurada entre diferentes aplicaciones.

Se basa en una sintaxis de objetos y arrays, representada en TEXTO, que permite representar datos simples y complejos de forma sencilla, liviana, e INDEPENDIENTE DEL LENGUAJE DE PROGRAMACION.

Características y ventajas

- Ligereza: Es un formato de datos muy ligero, lo que lo hace ideal para aplicaciones web y móviles donde el ancho de banda es un recurso valioso.
- Fácil de leer y escribir: Utiliza una sintaxis simple y fácil de entender y de escribir.
- Independiente del lenguaje: Es compatible con muchos lenguajes de programación y plataformas, lo que lo hace ideal para aplicaciones que necesitan comunicarse con diferentes sistemas.
- Flexibilidad: Admite estructuras de datos complejas, como arrays y objetos anidados, lo que lo convierte en un formato muy flexible y escalable..
- Es gratis, libre y documentado
- Todo ello lo convierte en un formato muy popular y aceptado por la comunidad de programadores de diversas tecnologías, plataformas y lenguajes.

Librerías JJ (Java + JSON)

En la actualidad existen varias librerías para serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y deserializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos JSON:

- Jackson
- GSON
- Boon
- JSON.org

Proporcionan métodos y mapeadores para realizar consultas y creación de objetos java desde JSON.

JSONObject y JSONArray

JSON trabaja con dos tipos estructurados: objetos y arreglos.

- UN OBJETO: es una colección no ordenada de cero o más pares de nombres/valores. Los valores pueden ser cadenas, números, booleanos, nulos y estos dos tipos estructurados.
- UN ARREGLO: es una secuencia ordenada de cero o más objetos.

JSONObject

- Sirve para crear nuestros OBJETOS JSON.
- Cuenta con un método PUT sobrecargado para todo tipo de datos.
- El método necesita como argumentos una clave (String) y un valor.
- Acepta tipos de datos del lenguaje Java, objetos JSON y arreglos JSON.
- Este método arroja JSONException por lo que debemos tener todo en un bloque try- catch.
- Cuenta con un método GET+TipoDeDatos para obtener el valor de una clave (enviada como parámetro).

JSONArray

- Sirve para representar ARREGLOS de JSONObject o cualquier tipo de dato.
- Los elementos dentro del arreglo se separan por coma (de manera automática)
- Cuenta con un método PUT para ingresar elementos al arreglo.
- Cuenta con un método GET para acceder a un elemento en particular.
- Como no se registran con clave-valor, necesitamos un índice para ubicarlo.
- Cuenta con un método LENGTH para conocer la longitud del mismo.

Escritura de datos en el JSONArray

```
JSONArray jsonArray = new JSONArray();
JSONObject jsonObj1 = new JSONObject();
JSONObject jsonObj2 = new JSONObject();
JSONObject jsonObj3 = new JSONObject();

try {
    jsonObj1.put("nombre", "Juan");
    jsonObj1.put("edad", 30);
    jsonObj1.put("soltero", true);

    jsonObj2.put("nombre", "Pedro");
    jsonObj2.put("edad", 25);
    jsonObj2.put("soltero", false);
}
```

```
jsonObj3.put("nombre", "Roberto");
jsonObj3.put("edad", 15);
jsonObj3.put("soltero", true);

jsonArray.put(jsonObj1);
jsonArray.put(jsonObj2);
jsonArray.put(jsonObj3);
} catch (JSONException e) {
    e.printStackTrace();
}
```

Creación del archivo JSON:

```
try {
    FileWriter file = new FileWriter("archivo.json");
    file.write(jsonArray.toString());
    file.close();
    System.out.println("Archivo JSON creado correctamente");
} catch (IOException e) {
    e.printStackTrace();
}
```

Grabado de la información en un archivo JSON:

Crearemos un método estático en una clase para poder reutilizar el grabado de información:

```
public static void grabar(JSONArray array) {
    try {
        FileWriter file = new FileWriter("test.json");
        file.write(array.toString());
        file.flush();
        file.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Lectura de archivos JSON:

```
public static JSONTokener leer(String archivo) {
    JSONTokener tokener=null;

    try {
        tokener = new JSONTokener(new FileReader(archivo));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    return tokener;
}
```

```

try {
    JSONArray arreglito;
    arreglito = new JSONArray(JSONUtilities.leer("test.json"));
    System.out.println(arreglito.length());

    for (int i = 0; i < arreglito.length(); i++) {

        Persona persona = new Persona();

        JSONObject jsonObject = arreglito.getJSONObject(i);
        System.out.println(jsonObject.getString("picture"));

        JSONObject objName = jsonObject.getJSONObject("name");
        persona.setNombre(objName.getString("first"));
        persona.setApellido(objName.getString("last"));

        JSONArray tags = jsonObject.getJSONArray("tags");
        for (int j = 0; j < tags.length(); j++) {
            String tag = tags.getString(j);
            System.out.println(tag);
        }
        JSONArray amigos = jsonObject.getJSONArray("friends");
        for (int j = 0; j < amigos.length(); j++) {
            JSONObject amigo = amigos.getJSONObject(j);
            System.out.println(amigo.getString("name"));
        }
    }
}

```

Buenas prácticas con JSON:

- Evitar errores comunes:

- Asegurarse de que la sintaxis del archivo JSON sea válida, esto significa que el archivo debe tener un formato de objeto JSON válido y estar bien estructurado.
- Verificar que los valores en el archivo JSON sean del tipo correcto, como cadena, número o booleano, según lo que se espera en tu aplicación.
- Asegurarse de que los nombres de las propiedades estén escritos correctamente y sean coherentes en todo el archivo JSON.

- Estándares de codificación y convenciones de nomenclatura:

- Utiliza nombres de propiedades descriptivos y consistentes en todo el archivo JSON, preferiblemente en minúsculas y separados por guiones bajos para facilitar la lectura.
- Utiliza indentación para estructurar el archivo JSON, lo que lo hace más fácil de leer y de mantener.
- Usa comillas dobles para los nombres de las propiedades y para los valores de cadena, ya que es el estándar aceptado en la mayoría de las bibliotecas JSON.

- Consideraciones de seguridad:

- Validar el archivo JSON antes de analizarlo para detectar y prevenir cualquier contenido malintencionado, ya que un archivo JSON mal formado puede ser utilizado para ejecutar ataques de inyección de código.
- No confiar en la fuente del archivo JSON, ya que los datos pueden ser manipulados o alterados antes de ser guardados en el archivo.
- Si estás trabajando con datos confidenciales, asegúrate de encriptar el archivo JSON y de protegerlo con medidas de seguridad adicionales, como autenticación y autorización.