Desarrollo de Aplicaciones JSE

INDICE

1. INTRODUCCIÓN A JAVA

El lenguaje de programación Java es un **lenguaje orientado a objetos** lo que quiere decir que todos los recursos que creamos y administramos los vamos a considerar como objetos. Una factura será un objeto, un cliente será un objeto, una conexión a una base de datos también será considerada como un objeto.

Otra característica importante de Java es que es un **lenguaje independiente de la plataforma**. En otros lenguajes de programación, por ejemplo C, el hecho de crear una aplicación consta de una serie de pasos:

1. Se crea el código fuente de la aplicación
2. Se compila para un determinado Sistema Operativo (Plataforma) para conseguir el archivo binario (0,1)
3. Se linka el código para ese S.O. para obtener el archivo ejecutable (.exe)

El resultado es que si necesitamos que nuestra aplicación esté disponible para varias plataformas hay que generar un archivo ejecutable para cada una de ellas.

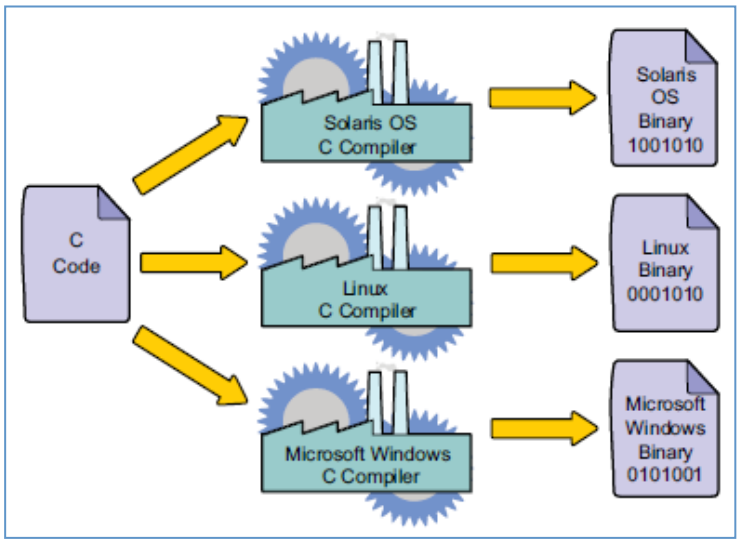


Gráfico 1. El código fuente se compila para cada plataforma

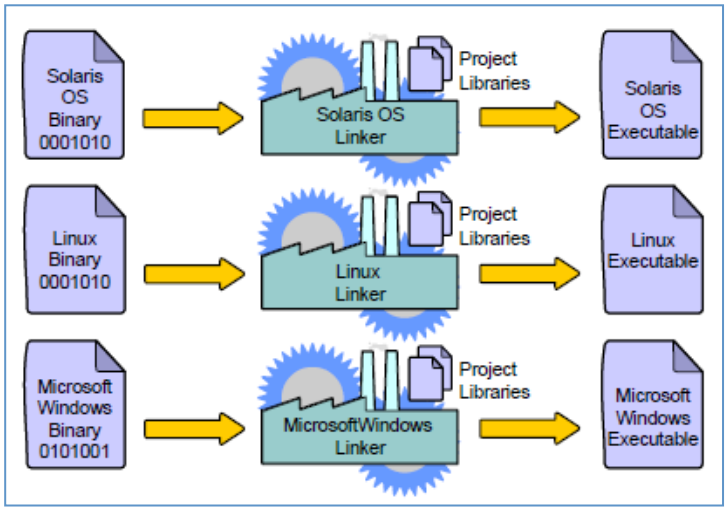


Gráfico 2. Se efectúa el linkado para cada plataforma

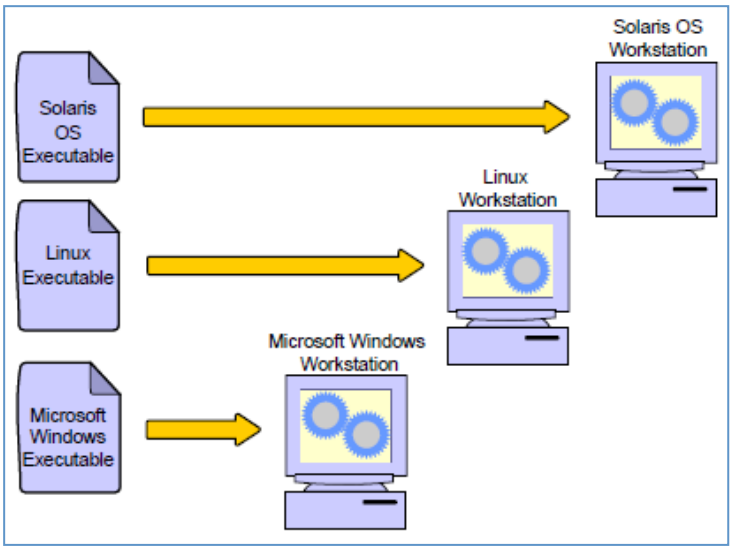


Gráfico 3. El ejecutable se distribuye a cada plataforma

Como decíamos, Java es un lenguaje independiente a la plataforma esto implica una reducción considerable de trabajo a la hora de crear una aplicación.

1. Se crea el código fuente de la aplicación (archivos .java)
2. Se compila para obtener archivos bytecode (.class)
3. No se necesita obtener un ejecutable, ya que los archivos bytecode los interpretara la maquina virtual de java (JVM Java Virtual Machine)

El archivo bytecode obtenido se puede ejecutar en cualquier plataforma, para ello tan solo es necesario tener instalada la maquina virtual de Java adecuada al S.O. instalado.

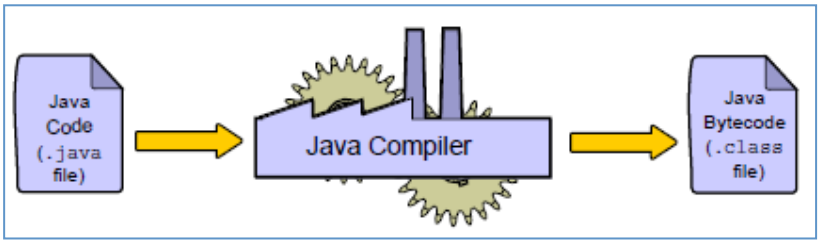


Gráfico 4. El código fuente se compila independiente de la plataforma

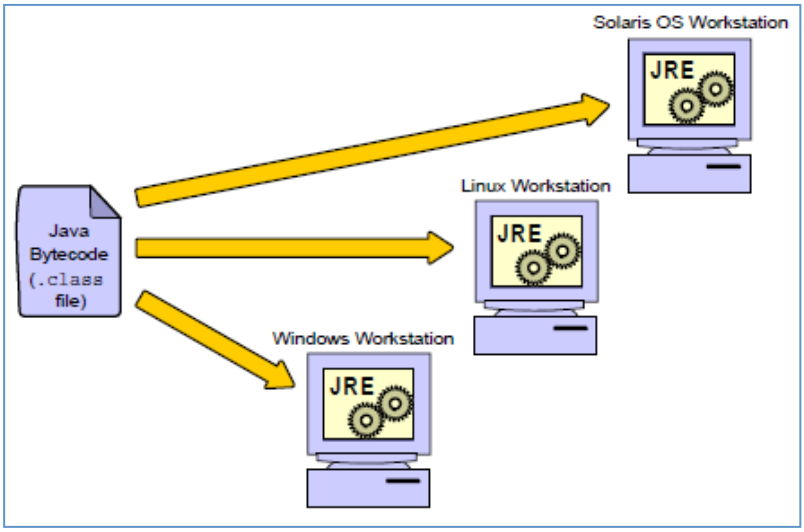


Gráfico 5. El archivo Bytecode puede ser interpretado por distintas plataformas

JRE Y JDK

Necesitamos tener una maquina virtual de Java (JVM) para poder ejecutar nuestras aplicaciones, esta se puede descargar dependiendo del perfil del usuario:

- **JRE (Java Runtime Environment)**; Sería necesario únicamente para ejecutar una aplicación. Esta sería la versión que se deben descargar los clientes, usuarios finales de nuestra aplicación. Incluye únicamente la JVM y un conjunto de librerías para poder ejecutar.
- **JDK (Java Development Kit)**; Estos son los recursos que necesitamos los desarrolladores ya que incluye lo siguiente:
    - JRE
    - Compilador de java
    - Documentación del API (todas las librerías de Java)
    - Otras utilidades por ejemplo para generar archivos .jar, crear documentación, efectuar un debug.
    - También incluye ejemplos de programas.

EDICIONES JAVA

El lenguaje Java se distribuye en tres ediciones:

- **JSE (Java Standard Edition)**; Es la edición más básica de Java pero no menos importante. Recoge los fundamentos básicos del lenguaje pero solo nos permite desarrollar aplicaciones locales, aplicaciones escritorio y applets (aplicaciones que se ejecutan en el navegador del cliente).
- **JEE (Java Enterprise Edition)**; Esta edición es la más completa de todas. Gracias a ella podremos desarrollar aplicaciones empresariales tales como aplicaciones web, aplicaciones eCommerce, aplicaciones eBusiness, ...etc.
- **JME (Java Micro Edition)**; Con esta edición podremos desarrollar aplicaciones para micro dispositivos tales como teléfonos móviles, PDAs, sistemas de navegación, ...etc.

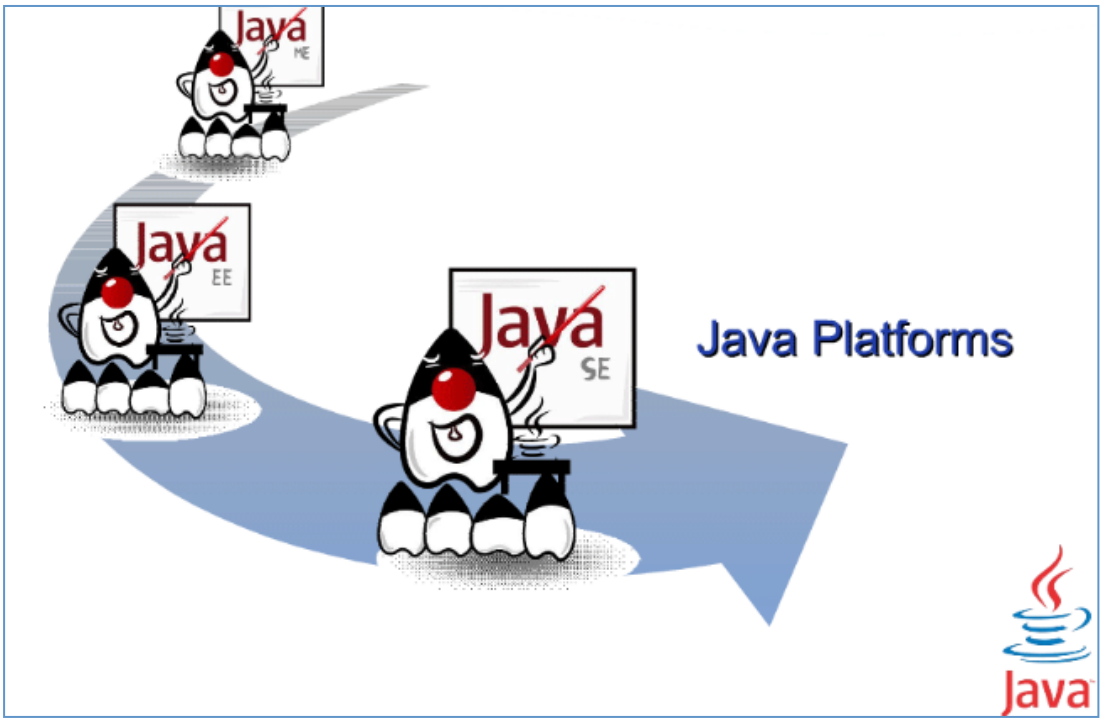


Gráfico 6. Ediciones Java

GARBAGE COLLECTOR

Cada vez que creamos un objeto este reside en la memoria de nuestro equipo. En otros lenguajes es necesario destruir dichos objetos cuando ya no son necesarios.

Java incorpora una ventaja a través de un hilo conocido como Garbage Collector, lo podríamos traducir como el recolector de basura.

Dicho proceso se lanza de vez en cuando de una forma similar a la función de salvar de un editor de texto. Este proceso recorre la memoria localizando aquellos objetos que ya no son necesarios porque no se van a utilizar más y los elimina.

El modo como selecciona dichos objetos lo estudiaremos en capítulos siguientes.

CARACTERÍSTICAS DE JAVA

Resumimos las características principales del lenguaje de programación Java:

- **Orientado a objetos**; como se mencionaba al inicio del capítulo, en Java todo se considera como un objeto. La OO facilita la reutilización de código como se verá más adelante.
- **Distribuido**; Java permite desarrollar aplicaciones distribuidas, esto significa que parte de la aplicación puede alojarse en un servidor de Madrid y otra parte puede residir en otro servidor de Barcelona por ejemplo.
- **Simple**; Aunque os parezca mentira en este momento, Java es un lenguaje muy simple de programar. Además, contamos con muchas librerías ya desarrolladas, listas para utilizarse y lo más importante una buena documentación de uso.
- **Multihilo**; El lenguaje Java funciona mediante hilos no a través de procesos como otros lenguajes. Esto hace que sea más rápida y más segura su ejecución.
- **Seguro**; En el curso iremos viendo las diversas formas de implementar seguridad en aplicaciones Java.
- **Independiente de la plataforma**; como ya comentamos anteriormente.

RECUERDA QUE...

- Java es un lenguaje multiplataforma, por lo cual solo debes instalar el jdk adecuado a tu sistema operativo.
- El lenguaje Java se distribuye en tres ediciones: JSE, JEE, JME.
- No debes preocuparte por eliminar los objetos creados, el Garbage Collector se encarga de esta tarea.

2.- CLASES Y OBJETOS

Un **objeto** es un recurso dinámico que se crea y se almacena en memoria durante un tiempo determinado. A los objetos también se les conoce con el nombre de **instancias** de clase porque realmente esto es lo que son, una copia de la clase con unos valores determinados.

Una **clase** la podríamos definir como la plantilla a partir de la cual se va a generar el objeto.

Como ejemplo, podríamos definir una clase Cliente donde declararíamos todas las propiedades y métodos comunes a todos los clientes. Pues bien, un objeto o instancia de tipo cliente sería un cliente personalizado con su propio Nif, su nombre, su dirección, ...etc.

La clase es el archivo .java para definir un cliente y los objetos son los recursos dinámicos que se generan en la memoria.

DECLARACION DE CLASES

Para declarar una clase nos debemos ajustar a la sintaxis de Java. Dicha sintaxis nos indica que el archivo en el cual declaramos la clase se debe seguir este orden.

1. **Declaración del paquete**; Un paquete cumple una función similar a una carpeta en Windows. El paquete nos sirve para organizar nuestros recursos y además poder implementar niveles de acceso como veremos más adelante.
2. **Importaciones**; En una clase podremos utilizar otras clases ya creadas. Estas pueden ser del propio API o también clases desarrolladas por terceras personas. Para poder acceder a otras clases es necesario importarlas previamente.
3. **Declaración de la clase**; Aquí se definirán los recursos de la clase.

```java
// Declaracion del paquete
package ejemplo1clasesyobjetos;

// importaciones si fuesen necesarias

// Declaracion de la clase
public class Cliente {

    // propiedades
    public String nif;
    public String nombre;
    public String direccion;

    // constructores
    public Cliente() {
    }

    // metodos
    public String mostrarDatos() {
        return "nif=" + nif + " nombre=" + nombre + " direccion=" + direccion;
    }
}
```

Gráfico 7. Ejemplo declaración de clase

Como vemos en el ejemplo, una clase puede tener los siguientes recursos:

Propiedades; Las propiedades o también llamadas atributos o campos recogen las características de la clase. Según nuestro ejemplo un cliente se representa por su nif, nombre y dirección.

Constructores; Los constructores son un tipo de método especifico que se invocará cada vez que vayamos a crear un objeto o instancia de tipo Cliente.

Métodos; Son las acciones que podremos llevar a cabo sobre un cliente. En este caso, mostrar sus datos.

A lo largo del capítulo aprenderemos a crear y gestionar estos recursos.

DECLARACION DE PROPIEDADES

Una propiedad se entiende como una variable global a la cual se puede acceder desde cualquier otro recurso dentro de la misma clase y dependiendo del modo de acceso, también se podrá acceder a ella desde otras clases.

La sintaxis para declarar una propiedad es la siguiente:

acceso tipo nombreVariable;

Ejemplos:

```
public int numero;

private String nombre;

protected double medida;

long capital;
```

DECLARACION DE METODOS

Utilizamos los métodos para introducir el código de nuestra aplicación. Podremos invocar a un método tantas veces como se quiera por lo cual nos permite una reutilización de código para aquellas tareas repetitivas.

Igual que ocurre con las propiedades, un método se puede invocar desde otro recurso declarado en la misma clase y dependiendo de su acceso, se podrá llamar desde otras clases.

La sintaxis para declarar un método es la siguiente:

1.- si el método no devuelve ningún dato:

```
acceso void nombreMetodo(){

}
```

Ejemplos:

```
public void mostrarDatos(){
        // logica de negocio
}

protected void abrirConexion(){
        // logica de negocio
}
```

2.- si el método devuelve un dato:

```
acceso tipoDevuelto nombreMetodo(){
}
```

Ejemplos:

```
public int sumar(int num1, int num2){
    return n1 + n2;
}

public String getNombre(){
    return nombre;
}
```

Como observamos en los ejemplos, cuando un método devuelve un dato incluimos la clausula **return**. Esta instrucción lo que hace es devolver un dato a la sentencia que ha invocado al método.

En el ejemplo del método sumar, se han incluido dos argumentos en el método. Los argumentos son los valores que recibe el método para procesarlos. Los argumentos no son obligatorios, es decir, hay métodos que declaran argumentos y métodos que no. En este último caso los paréntesis siguen siendo obligatorios.

También destacamos que todo el cuerpo del método (lógica de negocio) va encerrado entre llaves.

DECLARACION DE CONSTRUCTORES

Un constructor lo hemos definido como una especie de método que se invoca a la hora de crear un objeto.

La sintaxis para definir un constructor es diferente a la del método ya que nunca puede devolver un dato y tampoco se declara como tipo void.

```
acceso nombreClase(){
}
```

El constructor debe tener obligatoriamente el mismo nombre que la clase. También puede ser que reciba argumentos o no. Estos no son obligatorios al igual que los métodos.

Ejemplos:

```
public Cliente(){

}


public Cliente(String nif, String nombre, String direccion){

        // Asignamos los valores recibidos a las propiedades del objeto

        this.nif = nif;

        this.nombre = nombre;

        this.direccion = dirección;

}
```

CREACIÓN DE OBJETOS Y ADMINISTRACIÓN DE MEMORIA

A la hora de crear un objeto utilizamos la palabra new seguida del constructor.

Ejemplo: new Cliente();

Por definición, en Java, los objetos son anónimos, que significa realmente esto?

Cuando creamos una variable siempre le asignamos un nombre de esta forma podemos acceder a dicha variable a través de dicho nombre. Pues bien, cuando creamos un objeto como hemos hecho en el ejemplo anterior, no asignamos un nombre al objeto por lo cual lo único que se sabe es la dirección de memoria donde ha sido generado.

De esta forma no podremos acceder a dicho objeto. Para solucionar este problema vamos a crear una variable del tipo de la clase que vamos a instanciar y almacenaremos en ella la dirección de memoria del objeto para que nos sirva de referencia.

Ejemplo: Cliente c = new Cliente();

De esta forma ahora podremos acceder al objeto a través de la variable c.

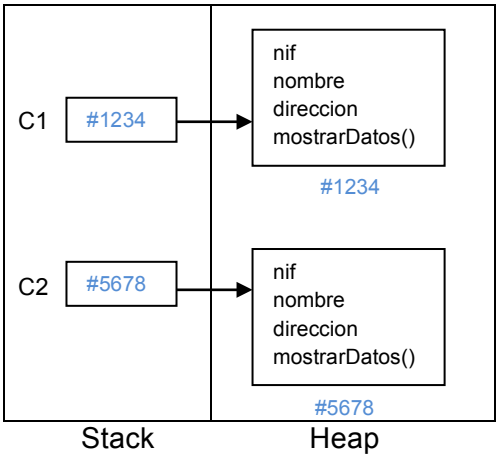
La memoria se divide en 2 partes:

- **Stack** (Pila); donde se almacenan las variables
- **Heap** (Montón); donde se almacenan los objetos

Si creamos dos objetos de tipo Cliente:

```
Cliente c1 = new Cliente();
Cliente c2 = new Cliente();
```

Este es el aspecto que presentaría la memoria:



Como vemos cada variable apunta a su propio objeto, esto permite tener varios objetos del mismo tipo en memoria.

Para acceder al objeto simplemente pondremos el nombre de la variable c1 o c2. Si queremos acceder a un recurso determinado del objeto utilizaremos el operador miembro (.).

Ejemplo:

```
c1.nombre = "Juan";
c2.nombre = "Maria";

System.out.println(c1.mostrarDatos());
System.out.println(c2.mostrarDatos());
```

El ámbito del objeto queda comprometido al ámbito de la variable. Cuando la variable pierde su ámbito por ejemplo se había declarado dentro de un método y la ejecución del método ha finalizado. La siguiente vez que se ejecute el Garbage Collector eliminará dicha variable de memoria y de esta forma el objeto perderá su referencia

por lo cual no se podrá volver a referenciar. El Garbage Collector también elimina el objeto por esta razón.

ACCESO A MIEMBROS

Para declarar el acceso a un miembro de la clase podemos utilizar los siguientes modos:

Modo de Acceso	Desde la misma clase	Desde el mismo paquete	Subclase	Resto de paquetes
public	SI	SI	SI	SI
protected	SI	SI	SI	NO
default	SI	SI	NO	NO
private	SI	NO	NO	NO

Como podemos observar están organizados de mayor a menor acceso. Expliquemos cada uno de ellos:

- **public**; un recurso definido como public se considera público por lo cual permite el acceso desde cualquier otro recurso ya sea declarado en la misma clase, en otra clase del mismo paquete, desde una subclase (herencia que se verá posteriormente) o incluso desde otras clases declaradas en otros paquetes.
- **protected**; un recurso definido como protected se considera protegido. Este modo permite el acceso desde cualquier otro recurso ya sea declarado en la misma clase, en otra clase del mismo paquete, desde una subclase. En este último caso es lo mismo si la subclase se encuentra en el mismo paquete o en otro.
- **default**; significa no establecer un modo de acceso. De esta forma solo se podrá invocar el recurso desde la misma clase o desde otra clase declarada en el mismo paquete.
- **private**; un recurso privado solo permite el acceso desde la misma clase donde ha sido declarado.

ENCAPSULACION

Una clase encapsulada es una clase que declara todas sus propiedades como privadas y solo se puede acceder a ella a través de los métodos get() y set().

Toda propiedad privada va a tener asociados dos métodos públicos (get y set) a través de los cuales vamos a poder modificar el valor de la propiedad (set) o recuperar su valor (get).

Veamos un ejemplo de una clase no encapsulada:

```
package ejemplo2encapsulacion;

public class FechaNoEncapsulada {

    public int dia;
    public int mes;
    public int anyo;

    public FechaNoEncapsulada() {
    }

    public void mostrarFecha(){
        System.out.println(dia + "/" + mes + "/" + anyo);
    }

}
```

Gráfico 8. Clase no encapsulada

```
public class Main {

    public static void main(String[] args) {

        FechaNoEncapsulada fechaErronea = new FechaNoEncapsulada();

        fechaErronea.dia = 78;
        fechaErronea.mes = 0;
        fechaErronea.anyo = -12;

        fechaErronea.mostrarFecha();
    }

}
```

Gráfico 9. Crear un objeto con datos erróneos

Al generar una instancia de una clase no encapsulada, como las propiedades son publicas puedo acceder directamente a ellas y establecer cualquier valor mientras que sea un numero entero.

El resultado al mostrar la fecha será: 78/0/-12

Para evitar la introducción de datos erróneos optamos por encapsular la clase, de esta forma tengo que acceder a las propiedades a través del método setXXX() en el cual

puedo incorporar la lógica de negocio necesaria para poder controlar si el valor introducido es correcto.

```
public class FechaEncapsulada {

    private int dia, mes, anyo;

    public void setDia(int dia) {
        // si dia es un valor correcto se asigna a la propiedad
        this.dia = dia;
    }

    public void setMes(int mes) {
        // si mes es un valor correcto se asigna a la propiedad
        this.mes = mes;
    }

    public void setAnyo(int anyo) {
        // si anyo es un valor correcto lo asigno
        this.anyo = anyo;
    }
```

Gráfico 10. Fragmento de una clase encapsulada

Si ahora intentamos introducir los valores anteriores el código no lo permitirá.

```
FechaEncapsulada fechaCorrecta = new FechaEncapsulada();

fechaCorrecta.setDia(78);
fechaCorrecta.setMes(0);
fechaCorrecta.setAnyo(-12);

fechaCorrecta.mostrarFecha();
```

Gráfico 11. Fragmento de acceso a la clase encapsulada

El resultado tras la ejecución de este ejemplo será: 0/0/0

RECUERDA QUE...

- Una clase es el código que nosotros escribimos, mientras que un objeto es una copia de esa clase en memoria.
- Una clase puede contener: propiedades, métodos y constructores.
- La encapsulación consiste en definir todas las propiedades de la clase como privadas y acceder a ellas mediante los métodos de acceso: getxxx() y setxxx().

3.- IDENTIFICADORES, PALABRAS CLAVE Y TIPOS

REGLAS PARA NOMBRAR UN IDENTIFICADOR

Consideramos un identificador al nombre de una clase, propiedad o método.

Las reglas a seguir para nombrar un identificador son las siguientes:

- Debe comenzar por una letra (mayúscula o minúscula), guión bajo (_) o símbolo de dólar ($).
- Son case sensitive por lo que se diferencias las letras mayúsculas y minúsculas. Se consideran 2 variables diferentes: nombre y Nombre.
- No existe una longitud máxima.

Ejemplos:

identifier
userName
user_name
_sys_var1
$change

PALABRAS CLAVE

Como todos los lenguajes de programación, Java define una serie de palabras clave que son reservadas para uso interno. No podremos utilizar ninguna palabra de esta lista como identificador.

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

Gráfico 12. Palabras clave

TIPOS PRIMITIVOS

Java define ocho tipos de datos primitivos. Los cuales se dividen a su vez en 4 categorías:

- Enteros: byte, short, int y long
- Decimales: float y double
- Lógico: boolean
- Textual: char

Enteros: byte, short, int y long

Los enteros primitivos pueden utilizar 3 formatos: decimal, octal y hexadecimal.

- 2 La representación decimal del numero 2.
- 077 Si el numero comienza por 0 indica un formato octal
- 0xBAAC Si el numero comienza por 0x indica un formato hexadecimal

El entero por defecto es el tipo int. Lo que quiere decir que cualquier numero entero se representará en 32 bits por defecto.

Los valores enteros de tipo long deben ir con el sufijo L o l.

Los diferentes tipos enteros representan los siguientes rangos:

Longitud	Nombre	Rango
8 bits	byte	2^7 a 2^7 -1
16 bits	short	2^{15} a 2^{15} -1
32 bits	int	2^{31} a 2^{31} -1
64 bits	long	2^{63} a 2^{63} -1

Decimales: float y double

Los decimales primitivos pueden utilizar 2 formatos: simple y exponencial.

- 3.14 Un valor simple decimal (un double)
- 6.02E23 Un numero decimal expresado con formato exponencial.

Los valores de tipo float deben ir con el sufijo F o f.

El decimal por defecto es el tipo double.

Los diferentes tipos decimales representan los siguientes rangos:

Longitud	Nombre	Rango
32 bits	float	2^{31} a 2^{31} -1
64 bits	double	2^{63} a 2^{63} -1

Otra de las novedades introducidas en Java 7 es el uso de la barra baja para separar números literales y que sea más fácil leerlos. Por ejemplo, si hasta ahora un millón se escribía así:

```
1000000
```

ahora lo escribiremos de esta forma:

```
1_000_000
```

También se introducen los literales binarios, con lo que ya no hay que convertirlos a hexadecimales y nos ahorramos un trabajo.

Lógico: boolean

Admite solo 2 literales true o false.

Textual: char

Se representa mediante 16 bits. El literal debe ir entre comillas simples ('). También admite las siguientes notaciones:

- 'a' La letra a
- '\t' Un tabulador
- '\u????' Un carácter Unicode, ????, se reemplaza por 4 dígitos hexadecimales
- Por ejemplo, '\u03A6' es la letra griega phi [Φ].

RECUERDA QUE...

- Al igual que todos los lenguajes de programación, Java reserva una serie de palabras para su uso interno. Estas no se pueden utilizar como identificadores.
- Java define ocho tipos primitivos: byte, short, int, long, float, double, char y boolean.

4.- OPERADORES Y CONTROL DE FLUJO

OPERADORES MATEMÁTICOS

Operador	Operación	Ejemplo	Comentario
+	Suma	sum = num1 + num2;	
-	Resta	diff = num1 – num2;	
*	Multiplicación	prod = num1 * num2;	
/	División	quot = num1 / num2;	La división devuelve un valor entero
%	Módulo	mod = num1 % num2;	El módulo devuelve el resto de la división.

OPERADORES INCREMENTO Y DECREMENTO

Operador	Operación	Ejemplo	Comentario

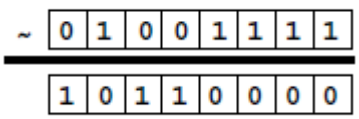
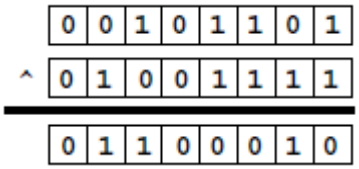
++	Pre-incremento (++variable)	int i = 6;
		int j = ++i;
		i vale 7, j vale 7
			La variable i se incrementa antes de asignar su valor a j.

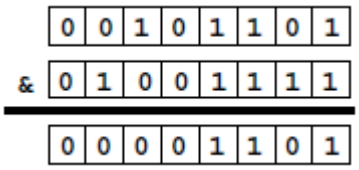
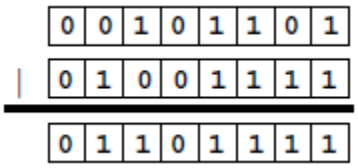
	Post-incremento (variable++)	int i = 6;
		int j = i++;
		i vale 7, j vale 6
			Primero se asigna el valor de i a j y después se incrementa i.

--	Pre-decremento (--variable)	int i = 6;
		int j = --i;
		i vale 5, j vale 5
			La variable i se decrementa antes de asignar su valor a j.

	Post-decremento (variable--)	int i = 6; int j = i--; i vale 5, j vale 6	Primero se asigna el valor de i a j y después se decrementa i.

OPERADORES BIT A BIT

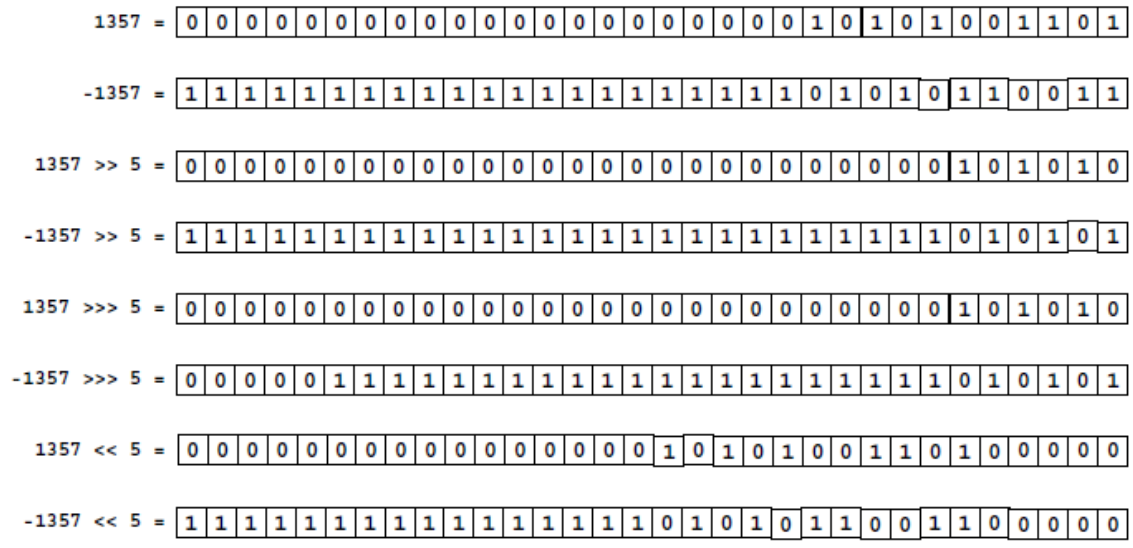
Operador	Operación	Ejemplo	Comentario
~	Complemento		Invierte valores, los 1 los convierte a 0 y viceversa.
^	Xor		Devuelve 0 si encuentra la combinación 0 y 0 ó 1 y 1. El resto devuelve 1.

&	And		Devuelve 1 si encuentra la combinación 1 y 1. En caso contrario devuelve 0.
|	OR		Devuelve 1 si encuentra algún 1 en la combinación. Si no lo encuentra devuelve 0.

OPERADORES DE DESPLAZAMIENTO

Operador	Operación	Comentario
>>	Desplazamiento derecha con signo	Desplaza a la derecha tantos bits se especifiquen. Si el numero es positivo se rellenaran los bits a 0. Si es negativo se rellenarán a 1.
>>>	Desplazamiento derecha sin signo	Desplaza a la derecha tantos bits se especifiquen sin tener en cuenta el signo. Los bits se rellenarán a 0.
<<	Desplazamiento izquierda	Desplaza a la izquierda tantos bits se especifiquen. Los bits se rellenarán a 0.

Ejemplos:



OPERADORES LÓGICOS

Operador	Operación	Ejemplo	Comentario

&&	AND	int i = 2;
		int j = 8;
		((i < 1) && (j > 6))
			Devuelve true si ambas condiciones se evalúan como ciertas.
			El resultado del ejemplo sería false.

||	OR	int i = 2;
		int j = 8;
		((i < 1) || (j > 10))
			Devuelve true si alguna de las condiciones o ambas se evalúan como ciertas.
			El resultado del ejemplo sería false.

!	NOT	int i = 2; (!(i < 3))	Devuelve lo contrario de como se evalúe la condición. El resultado del ejemplo sería false.

OPERADORES RELACIONALES

Operador	Operación	Ejemplo	Comentario
==	Igual a	int i=1; (i == 1)	El resultado es true.
!=	Diferente de	int i=2; (i != 1)	El resultado es true.
<	Menor que	int i=0; (i < 1)	El resultado es true.
<=	Menor o igual que	int i=1; (i <= 1)	El resultado es true.
>	Mayor que	int i=2; (i > 1)	El resultado es true.
>=	Mayor o igual que	int i=1; (i >= 1)	El resultado es true.

CONDICIONAL IF -ELSE

Una estructura condicional nos permite ejecutar un bloque de código u otro en función del resultado de una condición.

Sintaxis:

```
if(condición){
        // bloque de código
}
```

En este caso el bloque de código se ejecutará únicamente si se cumple la condición, dicho de otro modo, si la condición se evalúa como true.

Podemos completar este condición ofreciendo una alternativa.

```
Sintaxis:   if(condición){
                // bloque de código 1
            }else{
                //bloque de código 2
            }
```

En este otro caso, si la condición se evalúa como true se ejecutará el bloque de código 1, si la condición se evalúa como false entonces se ejecutará el bloque de código 2.

También podemos anidar condiciones.

```
Sintaxis:   if(condición 1){
                // bloque de código 1
            }else if (condición 2){
                //bloque de código 2
            }else{
                // bloque de código 3
            }
```

El bloque de código 1 se ejecutará si se cumple la condición 1. El bloque de código 2 se ejecutará si no se cumple la condición 1 y si se cumple la condición 2. Por último el bloque de código 3 se ejecutará si no se cumplen ninguna de las condiciones anteriores.

SWITCH-CASE

Esta estructura permite ejecutar un bloque de código dependiendo del valor devuelto por una expresión.

Sintaxis:

```
switch (expresión){
        case valor1:
            // bloque de código 1
            [break;]
        case valor2:
            // bloque de código 2
            [break;]
        default:
            // bloque de código 3
}
```

Si la expresión devuelve el valor1 se ejecutará el bloque de código 1. Si devuelve el valor2, se ejecutará el bloque de código 2 y si devuelve otro valor que no sea ni valor1, ni valor 2 entonces se ejecutará el bloque de código 3.

Las sentencias break son optativas. Si se pone esta sentencia se produce una ruptura lo que quiere decir que se ejecutará únicamente el bloque de código asociado a cada caso. Si no utilizamos break, entonces se ejecutarán todos los bloques de código desde el caso que se cumpla hasta el final. Por ejemplo, si la expresión devuelve el valor2 y no hemos puesto break, se ejecutará el bloque de código 2 y el bloque de código 3.

La expresión debe ser de uno de los siguientes tipos: byte, short, int, char o enumerado.

STRINGS EN SENTENCIAS SWITCH

Hasta ahora, las sentencias Switch usaban o bien tipos primitivos o bien tipos enumerados. Con Java 7 se pueden usar también strings, que antes tendrían que ir en una serie de if-else como estos:

```
private void processTrade(Trade t){
        String status = t.getStatus( );
        if(status.equalsIgnoreCase(NEW)){
            newTrade(t);
        }else if(status.equalsIgnoreCase(EXECUTE)){
```

```
                executeTrade(t);
        }else if(status.equalsIgnoreCase(PENDING)){
                pendingTrade(t);
        }
}
```

Comparadlo ahora como queda al poder usar una sentencia Switch:

```
public void processTrade(Trade t){
        String status = t.getStatus( );
        switch (status){
                case NEW : newTrade(t);
                                break;
                case EXECUTE : executeTrade(t);
                                break;
                case PENDING : pendingTrade(t);
                                break;
         }
 }
```

BUCLE FOR

Un bucle es un conjunto de instrucciones que se ejecutan varias veces dependiendo de una condición.

El bucle for se considera un bucle determinado porque sabemos de antemano el número de veces que se va a ejecutar.

Sintaxis:

```
for(inicio variable contador; condición; modificación){
        // cuerpo del bucle
}
```

Los términos utilizados los detallamos a continuación:

- **inicio variable contador**; para poder controlar el número de veces que se va a ejecutar el bucle utilizamos una variable. Esta variable puede estar declarada anteriormente o podemos declararla en el bucle. Con iniciar nos referimos a dar un valor inicial a la variable.
- **condición**; el bucle se ejecutará mientras se cumpla la condición.
- **modificación**; cada vez que se ejecute el bucle procedemos a modificar la variable contador. Lo más común suele ser incrementar o decrementar su valor.
- **cuerpo del bucle**; se denomina de esta forma al conjunto de instrucción que se ejecutan de forma repetitiva.

BUCLE FOR - EACH

A partir de Java 5 surge una mejora en el bucle for denominada for-each. Este bucle nos permite recorrer una colección de datos sin necesidad de tener una variable contador y condición.

Sintaxis:
```
for(tipo variable : coleccion){
        // cuerpo de bucle
}
```

Se define una variable del mismo tipo que los elementos de la colección. Si esta contiene números enteros se declarará una variable de tipo int, por ejemplo.

El funcionamiento, sería el siguiente, se recorre la colección comenzando por el primer elemento y terminando por el último. Cada elemento se asigna a la variable declarada y se ejecuta el cuerpo del bucle. Se pasa al siguiente elemento, el cual se almacena de nuevo en la variable y se procesa de nuevo el cuerpo del bucle. Estas operaciones se repiten hasta el último elemento de la colección.

BUCLE WHILE

El bucle while se considera un bucle indeterminado puesto que depende de una condición el número de veces que se va a ejecutar.

Sintaxis:
```
while(condición){
        // cuerpo del bucle
}
```

Si la condición se evalúa por primera vez como false no se ejecutará ninguna vez el cuerpo del bucle.

BUCLE DO-WHILE

Es una variante del anterior. También se considera un bucle indeterminado.

Sintaxis:
```
do {
        // cuerpo del bucle
} while (condición);
```

La diferencia con el bucle anterior es que esta vez la condición se evalúa al final, por lo cual si esta devuelve un resultado false, se habrá ejecutado el cuerpo del bucle una vez.

SENTENCIAS BREAK Y CONTINUE

Son instrucciones que permiten una alteración en la ejecución de un bucle.

BREAK .-

Permite finalizar, dar por terminado, totalmente la ejecución del bucle.

Sintaxis:
```
do {
        // sentencias 1
        if (condicion 1)
                break;
        // sentencias 2
} while (condición 2);
```

Si se cumple la condición 1 el bucle do-while habrá finalizado y continuaremos con la siguiente línea de código.

CONTINUE .-

Permite comenzar una nueva iteración sin terminar la actual.

Sintaxis:

```
do {
        // sentencias 1
        if (condicion 1)
                continue;
        // sentencias 2
} while (condición 2);
```

Si se cumple la condición 1 el bucle comenzará una nueva iteración sin terminar la actual, sin ejecutar las sentencias 2.

RECUERDA QUE...

- Java presenta las estructuras if-else y switch-case como condiciones
- Para generar bucles tenemos las estructuras: for, for-each, while y do-while
- Podemos romper o avanzar la ejecución de un bucle con las instrucciones: break y continue.

5.- ARRAYS

Un array es una estructura de datos que permite almacenar varios elementos. Cada uno de los elementos se almacena en una posición del array y para acceder a dicho elemento lo hacemos a través de un índice.

Los arrays tienen una serie de características:

- Se debe definir el tamaño del array al inicio. El tamaño se refiere al número de elementos máximo que puede almacenar.
- Todos los elementos han de ser del mismo tipo de datos.
- Son estructuras estáticas, lo que quiere decir que una vez que hemos fijado el tamaño del array este no se puede redimensionar.

Podemos definir arrays de una o varias dimensiones.

UNA DIMENSIÓN

DECLARAR UN ARRAY

En Java los arrays se tratan como objetos por lo cual necesitan de una variable de referencia para almacenar la dirección de memoria donde son creados.

Declarar un array significa crear una variable de tipo array. Lo que diferencia de la declaración de una variable normal a una variable de tipo array es el uso de corchetes ([ ]). Veremos más adelante que los corchetes también definen la dimensión del array, un grupo de corchetes será un array de una dimensión, dos grupos de corchetes definen un array de dos dimensiones y así sucesivamente.

Un array puede almacenar cualquier tipo de datos desde tipos primitivos como tipos de clase o sea objetos.

Veamos como declarar arrays de ambos tipos:

```
// Declarar un array de tipo primitivo
int numeros[];

// Declarar un array de objetos
Alumno alumnos[];
```

Gráfico 13. Ejemplos de declaración de arrays

Los corchetes se pueden posicionar antes o después de la variable y aunque en principio parece que es lo mismo no lo es. Vemos la diferencia con un ejemplo:

```
// Ejemplos de declaración
int num1[];
int [] num2;

int n1[], n2;
int [] n3, n4;
```

Gráfico 14 . Ejemplo de posición de los corchetes

En el primer ejemplo declaramos como variables de tipo array num1 y num2. En este caso tiene exactamente el mismo significado poner los corchetes antes o después de la variable.

En el segundo ejemplo declaramos diferente tipo de variables en función de la posición de los corchetes.

- Si ponemos los corchetes después de la variable n1 indicamos que es un array, sin embargo n2 es una variable de tipo int (no es un array).
- Si ponemos los corchetes delante de las variables n3 y n4 estamos declarando las dos variables como arrays.

En este punto, si intentamos acceder a la variable del array nos devolverá un **NullPointerException** ya que como toda variable de tipo objeto almacena un valor **null** si no apunta a ningún objeto creado.

CREAR EL ARRAY

Una vez declarado el array el siguiente paso es crear el array. En Java es obligatorio dar un tamaño al array, esto es, indicar el número de elementos máximo que puede contener.

```
// Creación de arrays
numeros = new int[5];
alumnos = new Alumno[3];
```

Gráfico 15. Ejemplo de creación de arrays

El array numeros puede contener hasta 5 números mientras que el array alumnos puede contener un máximo de 3 objetos de tipo Alumno.

ALMACENAR ELEMENTOS EN UN ARRAY

```
// Almacenar elementos en un array
numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
numeros[3] = 40;
numeros[4] = 50;

alumnos[0] = new Alumno(1, "Juan", "López", 5.6F);
alumnos[1] = new Alumno(2, "Marcos", "Saenz", 8.3F);
alumnos[2] = new Alumno(3, "Maria", "Arias", 6.3F);
```

Gráfico 16. Almacenar elementos en un array

Para poder almacenar elementos en un array es necesario hacer referencia a su índice. El índice de un array siempre comienza en 0 y termina en longitud-1. En el array de números enteros con longitud 5, el índice comienza en 0 y termina en 4. En el array de alumnos con longitud 3, el índice comienza en 0 y termina en 2.

GESTION DE MEMORIA

Vamos a representar en memoria los dos arrays creados con sus elementos.

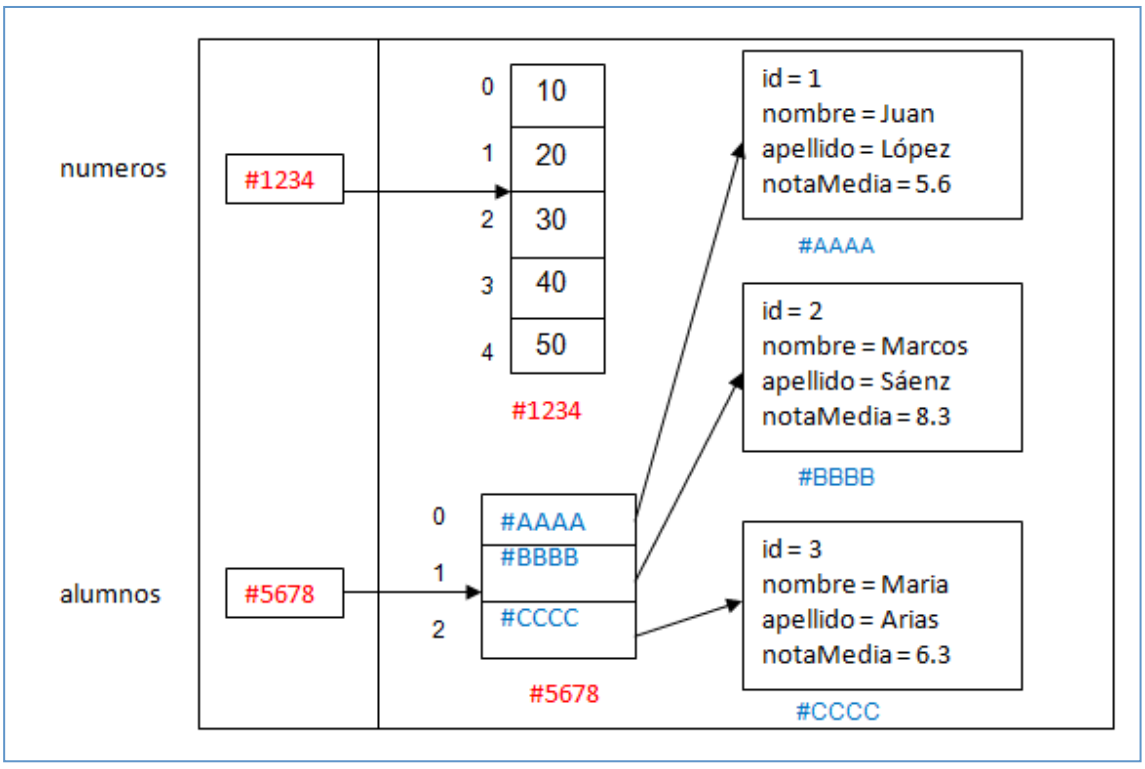


Gráfico 17. Representación en memoria de los arrays números y alumnos

Como vemos en los arrays de tipo primitivo se almacena directamente el valor, sin embargo en los arrays de tipo clase se almacenan las direcciones de memoria de los objetos almacenados.

DECLARAR, CREAR Y ALMACENAR ELEMENTOS

Hay una forma más rápida de declarar, crear y almacenar elementos en el array en una sola línea de código.

En el siguiente ejemplo se declara la variable de tipo array nums y se crea el array con los 8 números entre llaves. En este caso el array tiene una longitud de 8 elementos y sabemos que no se podrá redimensionar.

```
// Declarar, crear y almacenar elementos en una sola sentencia
int nums[] = {1,2,3,4,5,6,7,8};
```

Gráfico 18. Forma rápida de manejar arrays

ACCEDER A UN ELEMENTO

Para acceder a un elemento del array es necesario indicar su índice.

```
// Acceder a un elemento
System.out.println(numeros[3]);
System.out.println(alumnos[1].getNombre());
```

Gráfico 19. Acceso a los elementos del array

En el primer ejemplo accedemos al elemento con índice 3 en el array de numeros, el valor recuperado será el numero 40.

En el segundo ejemplo accedemos al objeto referenciado por el elemento con índice 1 del array de alumnos. Una vez que obtenemos la referencia del objeto podremos recuperar su nombre, el cual será Marcos.

RECORRER UN ARRAY

Muchas veces no nos interesa tan solo acceder a un elemento concreto del array sino más bien queremos recorrer todo el array accediendo a todos sus elementos. Pues bien, la forma más optima de llevar este proceso a cabo es con cualquiera de los dos formatos del bucle for.

- **bucle for**

La propiedad **length** de los arrays nos devuelve la longitud del mismo, o sea el número de elementos que tiene. Aprovechamos esta propiedad para establecer la condición del bucle.

```
// Recorrer el array con bucle for
for(int i=0; i<numeros.length; i++){
    System.out.println(numeros[i]);
}
```

Gráfico 20. Recorremos el array de números con el bucle for

La variable contador i nos sirve de índice, por lo cual la inicializamos a 0 que es el primer índice del array. La condición del bucle indica mientras que i sea menor que la longitud del array, hay que tener en cuenta que i nunca puede tomar el valor de la longitud. Recordamos que si un array tiene 5 elementos, su longitud es 5 y el índice empieza en 0 y termina en 4.

Si por error dejamos que i tomé el valor 5 esto producirá una excepción de tipo **ArrayIndexOutOfBoundsException** para indicar que estamos accediendo fuera del índice del array.

- **bucle for-each**

El bucle for-each es más cómodo de utilizar ya que no es preciso indicar el índice de los elementos.

```
// Recorrer el array con bucle for-each
for(Alumno a: alumnos){
    System.out.println(a);
}
```

Gráfico 21. Recorremos el array de alumnos con el bucle for-each

COPIAR ARRAYS

La clase **System** aporta un método **arraycopy** que permite copiar un array en otro. Veamos la sintaxis:

```
arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

Gráfico 22. Sintaxis del método arraycopy

A continuación detallamos cada uno de sus parámetros:

- src; El array origen, del cual vamos a copiar los datos.
- srcPos; posición desde la cual empezamos a copiar en el array de origen
- dest; El array destino, en el cual vamos a pegar los datos.
- destPos; posición desde la cual empezamos a pegar en el array de destino
- length; el número de elementos que serán copiados.

Vamos a ver un ejemplo en el cual copiamos 3 elementos del array nums comenzando en el índice 2 y los pegamos en el array números a partir de la posición con índice 1.

```
// copiar elementos de un array en otro
System.arraycopy(nums, 2, numeros, 1, 3);
```

Gráfico 23. Ejemplo del método arraycopy

El array numeros obtenido será el siguiente:

```
10 3 4 5 50
```

Gráfico 24. Resultado del array numeros

VARIAS DIMENSIONES

DECLARAR UNA MATRIZ

Los arrays de varias dimensiones solemos conocerlos con el nombre de matrices. Para declarar matrices utilizamos tantos grupos de corchetes como dimensiones queremos en el array.

En el siguiente ejemplo declaramos dos matrices de dos dimensiones.

```
// Declarar un array de tipo primitivo
int numeros[][];

// Declarar un array de objetos
Alumno alumnos[][];
```

Gráfico 25. Declaración de matrices

CREAR UNA MATRIZ

- **Matrices cuadradas**

Una matriz cuadrada es aquella que contiene el mismo número de columnas para todas las filas.

En el siguiente ejemplo creamos la matriz números con 3 filas y 2 columnas y también creamos la matriz alumnos con 2 filas y 2 columnas.

```
// Creación de matrices
numeros = new int[3][2];
alumnos = new Alumno[2][2];
```

Gráfico 26. Creación de matrices cuadradas

- **Matrices no cuadradas**

Una matriz no cuadrada es aquella que cada fila tiene un determinado número de columnas.

En el siguiente ejemplo creamos una matriz no cuadrada con 2 filas, la primera de ellas contendrá 3 columnas y la segunda contendrá 2 columnas.

```
// Creación de matrices no cuadradas
int [][] nums = new int[2][];
nums[0] = new int[3];
nums[1] = new int[2];
```

Gráfico 27. Creación de matriz no cuadrada

Siempre debemos especificar el número de filas para luego definir el numero de columnas de cada fila aparte.

Lo que no podemos hacer es definir el numero de columnas, o sea, definirlo al revés.

ALMACENAR ELEMENTOS EN UNA MATRIZ

Para almacenar elementos necesitamos acceder a sus índices fila y columna.

```
// Almacenar elementos en una matriz
numeros[0][0] = 1;
numeros[0][1] = 2;
numeros[1][0] = 3;
numeros[1][1] = 4;
numeros[2][0] = 5;
numeros[2][1] = 6;

nums[0][0] = 10;
nums[0][1] = 20;
nums[0][2] = 30;
nums[1][0] = 40;
nums[1][1] = 50;
```

Gráfico 28. Almacenar elementos en una matriz

GESTION DE MEMORIA

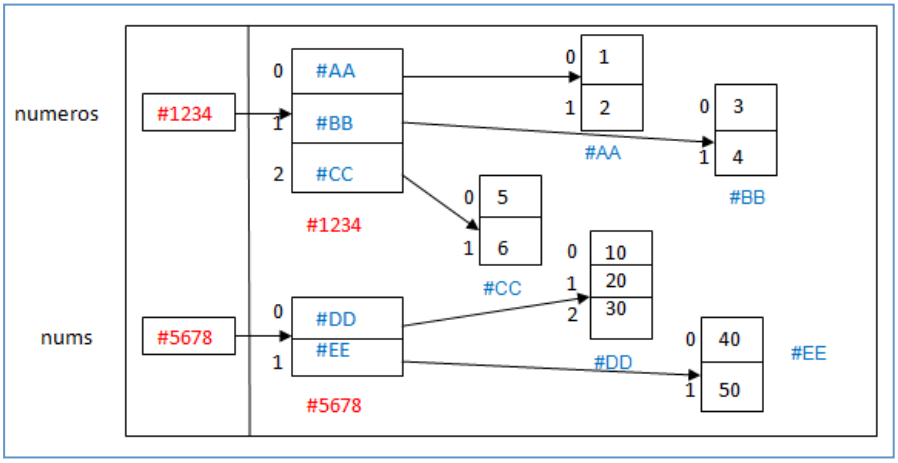


Gráfico 29. Representación en memoria de las matrices números y nums

DECLARAR, CREAR Y ALMACENAR ELEMENTOS

De la misma forma que en los arrays de una dimensión podíamos unificar todas estas tareas en una sola instrucción también lo podemos hacer con las matrices. Veamos como:

```
// Declarar, crear y almacenar elementos en una sola sentencia
int matriz1[][] = {{1,2},
                   {3,4}};

int matriz2[][] = {{1,2,3},
                   {4,5,6,7,8},
                   {9,0}};
```

Gráfico 30. Forma rápida de manejar matrices

En el primer ejemplo creamos una matriz cuadrada de 2 filas y 2 columnas. En el segundo ejemplo hemos creado un matriz no cuadrada de 3 filas: la primer fila tiene 3 elementos, la segunda fila tiene 5 elementos y la tercera fila tiene 2 elementos.

ACCEDER A UN ELEMENTO

Para acceder a un elemento de una matriz es necesario especificar la fila y columna del elemento.

```
// Acceder a un elemento
System.out.println(numeros[1][0]);
```

Gráfico 31. Acceder al elemento situado en la segunda fila y primera columna

Según nuestro ejemplo devolverá el valor 3.

RECORRER UNA MATRIZ

- **bucle for**

Necesitamos de dos bucles anidados. El primero de ellos es el encargado de recorrer el array de las filas con el índice i mientras que el segundo es quien recorre el array de las columnas. Este código es válido para recorrer matrices cuadradas y no cuadradas.

```
// Recorrer el array con bucle for
for(int i=0; i<numeros.length; i++){
    for(int j=0; j<numeros[i].length; j++){
        System.out.print(numeros[i][j] + " ");
    }
    System.out.println("");
}
```

Gráfico 32. Recorrer una matriz con el bucle for

```
1 2
3 4
5 6
```

Gráfico 33. Resultado obtenido

- **bucle for-each**

```
// Recorrer el array con bucle for-each
for(int[] columna : nums){
    for(int dato : columna){
        System.out.print(dato + " ");
    }
    System.out.println("");
}
```

Gráfico 34. Recorrer una matriz con el bucle for-each

Con el bucle for-each nuevamente necesitamos de dos bucles anidados. El primero nos devuelve los arrays de las columnas que recorremos en el bucle interno. El resultado que obtenemos es el que vemos a continuación.

```
10 20 30
40 50
```

Gráfico 35. Resultado obtenido

RECUERDA QUE...

- Un array se considera un objeto.
- Java solo reconoce los arrays de una dimensión, para manejar arrays de varias dimensiones se trabaja como un array de arrays.
- Podemos crear matrices cuadradas o no.

6.- CLASES AVANZADAS

HERENCIA

Una de las características de la programación orientada a objetos es la herencia. El termino herencia es conocido por todos, quien no ha oído hablar de una herencia en la familia. Nosotros no vamos a heredar dinero ni propiedades pero si vamos a aprender a heredar recursos de otras clases.

Observemos el diseño de las siguientes clases:

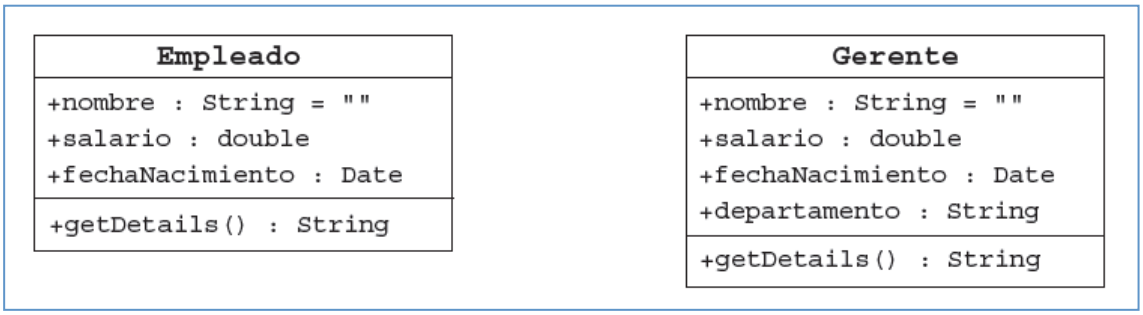


Gráfico 36. Diagramas de clase Empleado y Gerente

La implementación en código sería la siguiente:

```
public class Empleado {
  public String nombre = "";
  public double salario;
  public Date fechaNacimiento;

  public String getDetails() {...}
}
```

Gráfico 37. Implementación de la clase Empleado

```
public class Gerente {
  public String nombre = "";
  public double salario;
  public Date fechaNacimiento;
  public String departamento;

  public String getDetails() {...}
}
```

Gráfico 38. Implementación de la clase Gerente

Este ejemplo ilustra la duplicación de datos entre las clases Gerente y Empleado. Asimismo, podría haber una serie de métodos igualmente aplicables a estas dos

clases. Por tanto, necesitamos una forma de crear una clase nueva a partir de una clase existente, lo que recibe el nombre de creación de subclases.

En los lenguajes orientados a objetos, se proporcionan mecanismos especiales que permiten definir una clase sobre la base de una clase definida con anterioridad. En el siguiente gráfico puede verse un diagrama UML donde la clase Gerente es una subclase de Empleado.

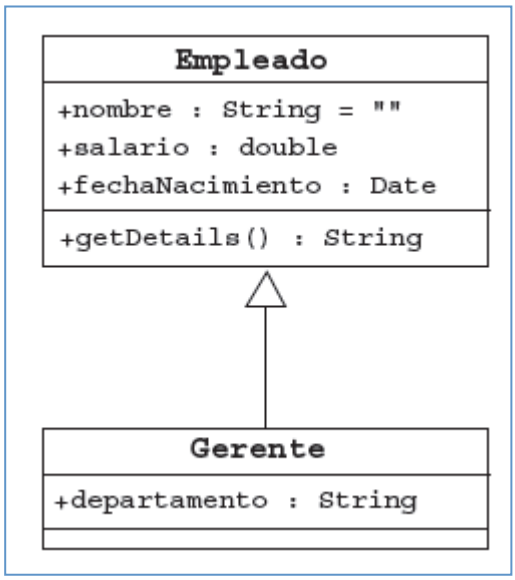


Gráfico 39. Diagramas de clase Empleado y Gerente a través de Herencia

Como se puede apreciar en el gráfico, en UML utilizamos una flecha continua para representar la herencia.

La clase Empleado se conoce como superclase, clase base o también como clase madre, mientras que la clase Gerente recibe el nombre de subclase, clase derivada o clase hija.

La implementación de la clase Gerente utilizando herencia sería la siguiente:

```
public class Gerente extends Empleado {
  public String departamento;
}
```

Gráfico 40. Implementación de la clase Gerente a través de herencia

Aunque sólo vemos la propiedad departamento declarado en la clase Gerente, de forma implícita tenemos todos los miembros heredados de la clase Empleado, nos referimos a las propiedades nombre, salario y fecha de nacimiento, así como al método getDetails().

En el lenguaje Java una clase solo puede heredar de otra, nunca de varias a la vez. Esta restricción se denomina herencia sencilla o herencia simple.

Veamos un ejemplo más ampliado de herencia.

La clase Empleado contiene tres atributos (nombre, salario y fechaNacimiento), así como un método (getDetails). La clase Gerente hereda todos estos miembros y añade un atributo más, departamento, además de usar el método getDetails. La clase Director hereda todos los miembros de Empleado y Gerente, pero además agrega el atributo vehiculoEmpresa y un nuevo método, aumentarComision.

Asimismo, las clases Ingeniero y Secretario heredan los miembros de la clase Empleado y podrían tener otros miembros específicos (no se muestran).

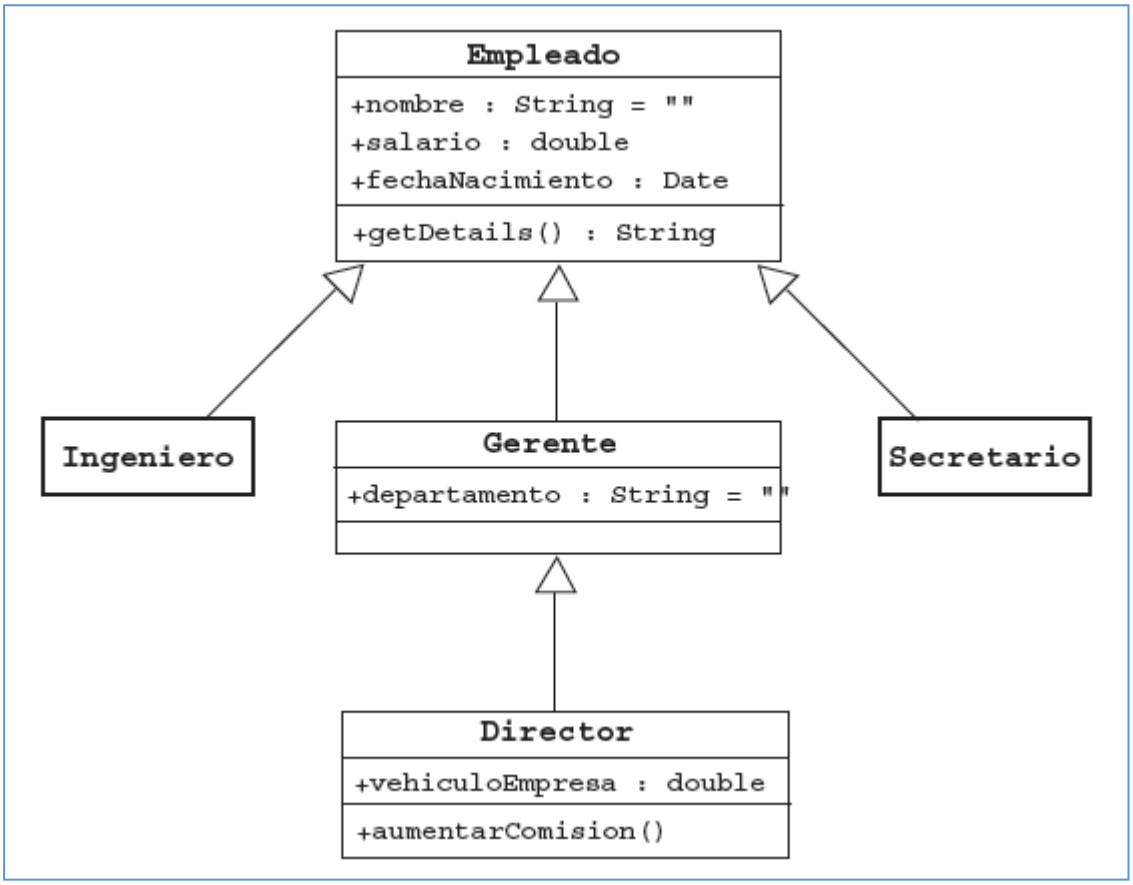


Gráfico 41. Ejemplo árbol de herencia

SOBREESCRITURA DE MÉTODOS

Además de poder agregar atributos para generar clases nuevas a partir de una clase existente, es posible modificar el comportamiento de la clase original.

Si se define un método en una subclase de tal forma que su nombre, el tipo de retorno y la lista de argumentos son idénticos a los de la superclase (la clase de nivel superior), se dice que el nuevo método sobrescribe al antiguo.

Observe estos ejemplos de métodos en las clases Empleado y Gerente:

```
public class Empleado {
  protected String nombre;
  protected double salario;
  protected Date fechaNacimiento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
           + "Salario: " + salario;
  }
}

public class Gerente extends Empleado {
  protected String departamento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
           + "Salario: " + salario + "\n"
           + "Gerente de: " + departamento;
  }
}
```

Gráfico 42. Método getDetails en la clase Empleado y Gerente

La clase Gerente contiene un método getDetails por definición, porque lo hereda de la clase Empleado. Pero el método original ha sido sustituido, sobrescrito, por la versión de la clase subordinada.

Recuerde que, para que un método pueda sustituir al de una clase de nivel superior, su nombre y el orden de sus argumentos deben ser idénticos a los del método de la superclase. Asimismo, el método sustitutivo no puede ser menos accesible que el método al que sustituye.

Analice este fragmento de código incorrecto:

```
public class Padre {
  public void hacerAlgo() {}
}

public class Hijo extends Padre {
  private void hacerAlgo() {} // no válido
}
```

Gráfico 43. Sobre escritura incorrecta de un método

REGLAS SOBREESCRITURA DE MÉTODOS

Si no cumplimos las reglas expuestas a continuación nos encontraremos con un error de compilación:

- El tipo de dato devuelto en el método sobreescrito debe ser el mismo o una subclase del tipo de dato devuelto en el método de la superclase.
- La lista de argumentos recibidos en el método sobreescrito debe ser la misma, nos referimos al mismo número, orden aunque pueden tener nombres diferentes.
- No se puede hacer el método sobreescrito menos accesible.
- Si el método original no tiene clausula throws no se la podemos poder. Lo que indica que si el método original no lanza excepciones, el sobreescrito tampoco.
- Sin embargo, si el método original tiene clausula throws, el método sobreescrito debe lanzar las mismas excepciones o una subclase de ella. También podemos quitar la clausula throws en el método sobreescrito.

LLAMADA A METODOS SOBREESCRITOS

Un método de una subclase puede llamar a un método de una superclase utilizando la palabra clave super.

La palabra super se refiere a la superclase de la clase en la que se ha utilizado esta clave. Se usa para hacer referencia a las variables miembro o a los métodos de la superclase. Se consigue utilizando la palabra clave super de la forma siguiente:

```
public class Empleado {
  private String nombre;
  private double salario;
  private Date fechaNacimiento;

  public String getDetails() {
    return "Nombre: " + nombre + "\nSalario: " + salario;
  }
}

public class Gerente extends Empleado {
  private String departamento;

  public String getDetails() {
    // llama al método de la superclase
    return super.getDetails()
           + "\nDepartamento: " + departamento;
  }
}
```

Gráfico 44. Invocar a un método sobreescrito

Una llamada con el formato super.metodo() llama al comportamiento completo, incluido cualquier efecto secundario del método al que se llamaría si el objeto hubiera sido una clase de nivel superior. El método no tiene por qué estar definido en la clase de nivel inmediatamente superior, puede heredarse de alguna clase situada más arriba en la jerarquía.

SOBRECARGA DE MÉTODOS

En algunos casos, puede que quiera escribir en la misma clase varios métodos que realizan la misma tarea básica pero con diferentes argumentos.

Pensemos en un método sencillo destinado a enviar una representación textual de su argumento. Este método se podría llamar println().

Java y otros lenguajes de programación permiten reutilizar un mismo nombre de método para varios métodos. Esto sólo funciona si las circunstancias bajo las que se hace la llamada permiten distinguir cuál es el método necesario. En el caso de los tres métodos de impresión, esta distinción se basa en el número de argumentos y su tipo.

Si se reutiliza el nombre de método, se obtienen los métodos siguientes:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

Gráfico 45. Método println sobrecargado

Cuando se escribe el código para llamar a uno de estos métodos, se elige el método adecuado en función del tipo de argumento o argumentos que se suministran.

REGLAS SOBRECARGA DE MÉTODOS

Al igual que ocurre con la sobreescritura de métodos si no cumplimos unas determinadas reglas en la sobrecarga de métodos nos encontraremos con errores de compilación. Estas reglas son las siguientes:

- Los métodos sobrecargados han de tener el mismo nombre.
- Pueden tener diferentes modificadores de acceso, lo que indica que uno puede ser público, otro privado, ...etc.
- Pueden tener diferente numero de argumentos y en diferente orden.
- El tipo devuelto puede ser diferente, pero cuidado. Si solo cambiamos esto genera un error de compilación porque el compilador piensa que estamos efectuando una sobreescritura.
- No hay restricción en cuanto al lanzamiento de excepciones.

- Los constructores no se heredan por lo que no se pueden sobreescribir, pero sí que puede haber sobrecarga.

SOBRECARGA DE CONSTRUCTORES

Cuando se instancia un objeto, el programa puede ser capaz de suministrar varios constructores basándose en los datos del objeto que se está creando.

Por ejemplo, un sistema de nóminas podría ser capaz de crear un objeto Empleado si conoce todos los datos básicos sobre la persona: nombre, salario base y fecha de nacimiento. Puede que, en alguna ocasión, el sistema no conozca el salario base o la fecha de nacimiento.

El siguiente ejemplo contiene cuatro constructores sobrecargados para la clase Empleado.

El primer constructor inicializa todas las variables de instancia.

En el segundo, no se incluye la fecha de nacimiento. La referencia this se utiliza como forma de envío de llamada a otro constructor (siempre dentro de la misma clase), en este caso, el primer constructor.

Por su parte, el tercer constructor llama al primer constructor pasando la constante de clase SALARIO_BASE.

El cuarto constructor llama al segundo constructor pasando SALARIO_BASE, que, a su vez, llama al primer constructor pasando null en lugar de la fecha de nacimiento.

```
public class Empleado {
  private static final double SALARIO_BASE = 15000.00;
  private String nombre;
  private double salario;
  private Date fechaNacimiento;

  public Empleado(String nombre, double salario, Date FdeNac) {
    this.nombre = nombre;
    this.salario = salario;
    this.fechaNacimiento = FdeNac;
  }
  public Empleado(String nombre, double salario) {
    this(nombre, salario, null);
  }
  public Empleado(String nombre, Date FdeNac) {
    this(nombre, SALARIO_BASE, FdeNac);
  }
  public Empleado(String nombre) {
    this(nombre, SALARIO_BASE);
  }
  // más código de Empleado...
}
```

Gráfico 46. Ejemplo sobrecarga de constructores

La palabra clave this en un constructor debe ser la primera línea de código del constructor. Puede haber más código de inicialización después de la llamada this, pero no antes.

Aunque las subclases heredan todos los métodos y variables de sus superclases, no heredan sus constructores.

Las clases sólo pueden obtener un constructor de dos maneras: Debe escribirlo el programador o, si éste no lo escribe, debe usar el constructor predeterminado.

LLAMADA A UN CONSTRUCTOR DE LA SUPERCLASE

Al igual que los métodos, los constructores pueden llamar a los constructores no privados de la superclase inmediatamente anterior.

A menudo se define un constructor con determinados argumentos y se quiere usar esos mismos argumentos para controlar la construcción de la parte superior de un objeto. Puede llamar a un determinado constructor de una superclase como parte de la inicialización de una subclase utilizando la palabra clave super en la primera línea del constructor de la subclase. Para controlar la llamada de ese constructor concreto, es preciso suministrar los argumentos adecuados a super(). Cuando no hay ninguna llamada a super con argumentos, se llama implícitamente al constructor de la

superclase que tenga cero argumentos. En ese caso, si no hay ningún constructor de nivel superior que tenga cero argumentos, se produce un error de compilación.

La llamada a super() puede adoptar cualquier cantidad de argumentos adecuados para los distintos constructores disponibles en la clase superior, pero debe ser la primera sentencia del constructor.

Si recordamos que la clase Empleado tiene el conjunto de constructores definidos en el gráfico 46, entonces se definirían los siguientes constructores en Gerente.

```
public class Gerente extends Empleado {
  private String departamento;

  public Gerente(String nombre, double salario, String depto) {
    super(nombre, salario);
    departamento = depto;
  }
  public Gerente(String nombre, String depto) {
    super(nombre);
    departamento = depto;
  }
  public Gerente(String depto) { // Este código da error: no hay super()
    departamento = depto;
  }
}
```

Gráfico 47. Invocar constructores de la superclase

El último constructor no es válido porque el compilador introduce una llamada implícita a la clase super() y en la clase Empleado no se ha suministrado ningún constructor sin argumentos.

Si se utilizan, es preciso colocar super o this en la primera línea del constructor. Si escribe un constructor que no contiene ninguna llamada a super(...) ni a this(...), el compilador introduce una llamada al constructor de la superclase de forma automática y sin argumentos.

Otros constructores pueden llamar también a super(...) o this(...), con lo que se hace una llamada a una cadena de constructores. Lo que ocurre al final es que el constructor de la clase de nivel superior (o quizás varios de ellos) se ejecuta antes que ningún otro constructor subordinado de la cadena.

LA CLASE OBJECT

La clase Object es la raíz de todas las clases en el lenguaje Java. Si se declara una clase sin cláusula extends, el compilador agrega automáticamente el código extends Object a la declaración, por ejemplo:

```
public class Empleado {
  // más código aquí
}
```

Equivale a:

```
public class Empleado extends Object {
  // más código aquí
}
```

Gráfico 48. Declaración de clase heredando de Object

Esto permite sobrescribir varios métodos heredados de la clase Object. A continuación se describen dos métodos importantes de Object.

EL METODO EQUALS

El operador == realiza una comparación para determinar la equivalencia de dos términos. Es decir, para dos valores de referencia x e y dados, x==y devuelve true si, y sólo si, x e y se refieren al mismo objeto.

La clase Object del paquete java.lang contiene el método public boolean equals(Object obj), que compara dos objetos para comprobar su igualdad. Si no se sobrescribe, el método equals() de un objeto devuelve true únicamente si las dos referencias comparadas se refieren al mismo objeto. No obstante, la intención de equals() es comparar el contenido de dos objetos siempre que es posible. Ésta es la razón por la que se sobrescribe con frecuencia. Por ejemplo, el método equals() en la clase String devuelve true únicamente si el argumento no es null y es un objeto String que representa la misma secuencia de caracteres que el objeto String con el que se ha llamado al método.

En conclusión, el operador == compara si dos variables contienen la misma referencia mientras que con el método equals podemos comparar las propiedades del objeto para verificar si son iguales o no.

Se debería sobrescribir el método hashCode cada vez que sobrescriba equals. Una implementación sencilla podría usar un XOR de bits en los códigos hash de los elementos cuya equivalencia se quiera comprobar.

En el siguiente ejemplo, la clase MyDate sobrescribe el método equals que compara los atributos de año, mes y día.

También sobrescribimos el método hashCode implementa un XOR de bits de los atributos de fecha. Esto hace que el código hash de los objetos iguales de MyDate tenga el mismo valor y ofrece la posibilidad de que fechas diferentes devuelvan valores diferentes.

```java
public class MyDate {

    private int day;
    private int month;
    private int year;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    @Override
    public boolean equals(Object o) {
        boolean result = false;
        if ((o != null) && (o instanceof MyDate)) {
            MyDate d = (MyDate) o;
            if ((day == d.day) && (month == d.month)
                    && (year == d.year)) {
                result = true;
            }
        }
        return result;
    }

    @Override
    public int hashCode() {
        return (day ^ month ^ year);
    }
}
```

Gráfico 49. Ejemplo sobreescritura del método equals y hashCode

El programa siguiente compara dos objetos MyDate que no son idénticos pero son iguales en cuanto a la comparación de año-mes-día.

```java
public class Main {

    public static void main(String[] args) {
        MyDate date1 = new MyDate(14, 3, 1976);
        MyDate date2 = new MyDate(14, 3, 1976);

        if (date1 == date2) {
            System.out.println("date1 es idéntica a date2");
        } else {
            System.out.println("date1 no es idéntica a date2");
        }

        if (date1.equals(date2)) {
            System.out.println("date1 es igual que date2");
        } else {
            System.out.println("date1 no es igual que date2");
        }

        System.out.println("set date2 = date1;");
        date2 = date1;

        if (date1 == date2) {
            System.out.println("date1 es idéntica a date2");
        } else {
            System.out.println("date1 no es idéntica a date2");
        }
    }
}
```

Gráfico 50. Ejemplo comparación de fechas

En la primera comparación con el operador == se compara si los dos objetos tienen la misma referencia.

En la segunda comparación con el método equals, se compara si los dos objetos son iguales, si tienen el mismo día, mes y año.

Después asignamos el objeto date1 a date2 y procedemos a una nueva comparación de las referencias de los objetos.

La ejecución de este programa de comparación genera la salida siguiente:

```
date1 no es idéntica a date2
date1 es igual que date2
set date2 = date1;
date1 es idéntica a date2
```

Gráfico 51. Resultados de la comparación de fechas

EL METODO TOSTRING

El método toString convierte un objeto en una representación de cadena o String. El compilador hace referencia a él cuando se produce una conversión automática de cadenas. Por ejemplo, la llamada System.out.println():

```
Date now = new Date();
System.out.println(now);

Equivale a:

System.out.println(now.toString());
```

Gráfico 52. Impresión de objetos

La clase Object define un método toString que devuelve el nombre de la clase y su dirección de referencia (normalmente de escasa utilidad).

Muchas clases sobrescriben toString para proporcionar información de mayor utilidad.

En el siguiente ejemplo creamos la clase Fecha sin sobreescribir el método toString.

```
public class Fecha {

    private int dia;
    private int mes;
    private int anyo;

    public Fecha() {
    }

    public Fecha(int dia, int mes, int anyo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
    }

}
```

Gráfico 53. Clase sin sobreescritura método toString

En la clase principal, creamos una instancia de Fecha e imprimimos el objeto creado.

```java
public class Main {

    public static void main(String[] args) {
        Fecha fecha = new Fecha(12, 4, 2012);
        System.out.println(fecha);

        FechaConToString otraFecha = new FechaConToString(25, 8, 2012);
        System.out.println(otraFecha);
    }

}
```

Gráfico 54. Clase Principal donde creamos e imprimimos los objetos

Al ejecutar la clase vemos como se imprime el objeto fecha. Es una representación que no nos aporta ninguna información de utilidad, sólo se muestra el paquete, el nombre de la clase y el código del objeto.

```
ejemplotostring.Fecha@19821f
25/8/2012
```

Gráfico 55. Resultados tras la impresión de los objetos

A continuación creamos la clase FechaConToString donde sobrescribimos el método toString para obtener una representación textual del objeto.

```java
public class FechaConToString {

    private int dia;
    private int mes;
    private int anyo;

    public FechaConToString() {
    }

    public FechaConToString(int dia, int mes, int anyo) {
        this.dia = dia;
        this.mes = mes;
        this.anyo = anyo;
    }

    @Override
    public String toString() {
        return dia + "/" + mes + "/" + anyo;
    }

}
```

Gráfico 56. Clase con sobreescritura del método toString

Al generar la instancia de esta última clase e imprimirlo, vemos que esta vez muestra la representación textual del objeto, aportándonos una mayor información.

CLASES ABSTRACTAS

Hemos visto como crear una subclase extendiendo de otra clase utilizando herencia. También hemos visto como podemos sobreescribir un método que hemos heredado.

Muchas veces nos vamos a encontrar con el siguiente caso: Creamos la superclase con un método que ya sabemos de antemano que se debe sobreescribir en la subclase porque su implementación debe variar. Entonces, qué sentido tiene implementar el método?

El lenguaje Java permite diseñar las clases de modo que los métodos declarados en las superclases no proporcionen ninguna implementación. Este tipo de métodos se denominan métodos abstractos. La implementación del método viene proporcionada por las subclases. Una clase que tenga uno o varios métodos abstractos se denomina clase abstracta.

Veamos el siguiente ejemplo.

Hemos creado una clase Figura donde hemos declarado dos métodos: El método mostrarPosición está implementado. Este método lo heredarán todas las subclases y su implementación no cambiará ya que todas las figuras tienen una posición en el espacio representada por los puntos x e y.

El método calcularArea es un método abstracto y por supuesto no está implementado. Observemos que los métodos abstractos terminan en punto y coma (;), si un método tiene llave de apertura y cierre ya está implementado aunque no tenga código en su interior.

El motivo por el cual el método calcularArea es abstracto es porque sabemos que las subclases han de sobreescribir este método ya que la forma de calcular el área de cada figura es diferente.

También observamos que la clase está declarada como abstracta, esto nos los indica el compilador. Si una clase tiene uno o más métodos abstractos, se debe declarar como abstracta. En caso contrario genera un error de compilación.

```java
public abstract class Figura {
    private int x;
    private int y;

    public Figura() {
    }

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

    public abstract double calcularArea();

    public String mostrarPosicion() {
        return  "[" + x + " , " + y + "]";
    }

}
```

Gráfico 57. Ejemplo de clase abstracta

A continuación vamos a ver las subclases creadas. Como el método heredado calcularArea es abstracto por definición las subclases también lo son hasta que se implemente dicho método.

No estamos obligados a implementarlo, pero si no lo hacemos debemos declarar las subclases como abstractas. Qué problema tendremos entonces? Pues que una clase abstracta realmente es una clase inacabada por lo cual no se puede instanciar.

```java
public class Rectangulo extends Figura {

    private double base;
    private double altura;

    public Rectangulo(int x, int y) {
        super(x, y);
    }

    public Rectangulo(int x, int y, double base, double altura) {
        super(x, y);
        this.base = base;
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return base * altura;
    }

}
```

Gráfico 58. Implementación del método abstracto en la clase Rectángulo

En la clase Rectangulo implementamos el método calcularArea con el algoritmo adecuado para calcular el área de los rectángulos.

A continuación vemos la implementación del mismo método en la clase Circulo.

```java
public class Circulo extends Figura{

    private double radio;

    public Circulo(int x, int y) {
        super(x, y);
    }

    public Circulo(int x, int y, double radio) {
        super(x, y);
        this.radio = radio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * radio * radio;
    }

}
```

Gráfico 59. Implementación del método abstracto en la clase Circulo

Desde la clase principal creamos tres instancias:

La primera de ellas, es una instancia de la clase Figura que genera un error de compilación puesto que dicha clase es abstracta y no se puede instanciar.

Creamos una instancia de la clase Rectangulo donde pasamos los siguientes argumentos al constructor: puntos x e y, base y altura del rectángulo.

También se crea una instancia de la clase Circulo con los siguientes datos: puntos x e y además del radio del circulo.

```
public class Main {

    public static void main(String[] args) {
        // Error de compilación
        // Figura figura = new Figura(4, 6);

        Rectangulo rectangulo = new Rectangulo(4, 6, 12, 15);
        rectangulo.mostrarPosicion();
        System.out.println("El area del rectangulo es: " +
                           rectangulo.calcularArea());

        Circulo circulo = new Circulo(3, 8, 5);
        circulo.mostrarPosicion();
        System.out.println("El area del circulo es: " +
                           circulo.calcularArea());
    }

}
```

Gráfico 60. Clase principal donde generamos las instancias

UML utiliza letras en cursiva para indicar elementos abstractos en los diagramas de clases. También se pueden marcar las clases abstractas con el indicador {abstract} en la sección de nombre.

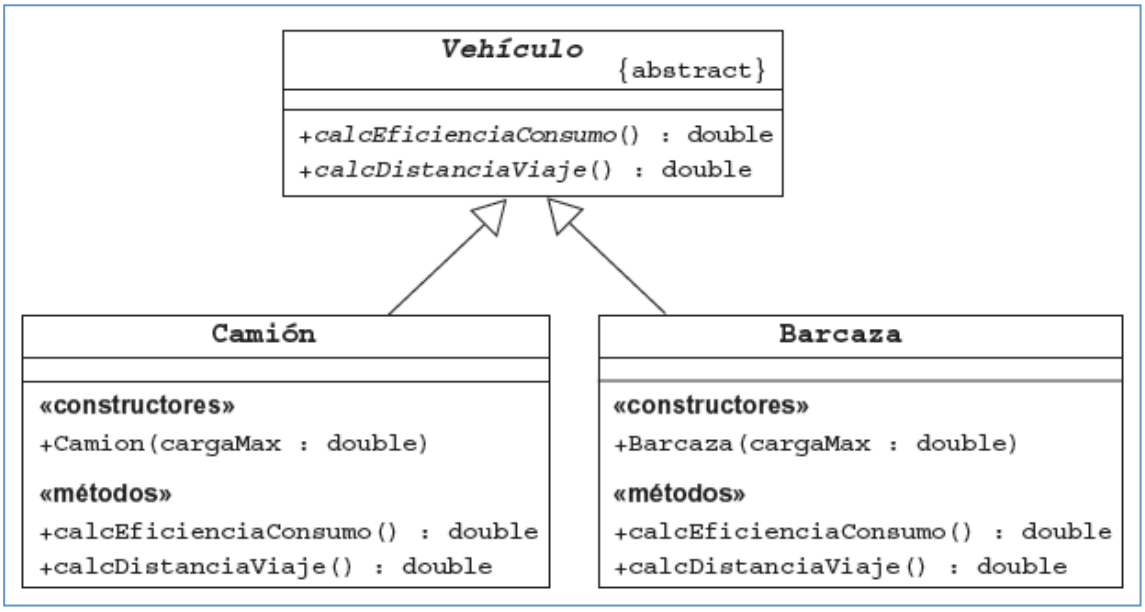


Gráfico 61. Diagrama UML clases abstractas

INTERFACES

Hemos visto que en Java solo es posible la herencia simple, el hecho de que una clase tan sólo puede heredar de otra clase, nunca de varias.

La forma de simular una herencia múltiple la tenemos en el uso de interfaces.

Una interface es una clase abstracta llevada al extremo. Una clase abstracta puede contener métodos implementados y métodos abstractos, así como propiedades y constructores. Pues bien, en una interface todos los métodos declarados son abstractos, nunca puede haber métodos implementados.

Todos los métodos declarados en una interfaz son públicos (public) y abstractos (abstract), con independencia de que mencionemos estos modificadores en el código o no. Igualmente, todos los atributos son public, static y final. En otras palabras, sólo se pueden declarar atributos constantes.

Al igual que los nombres de clases abstractas, los nombres de interfaz se utilizan como tipos de variables de referencia.

Imagine un grupo de objetos que comparten la misma habilidad: pueden volar. Puede construir una interfaz pública llamada ObjetoVolador que admite tres operaciones: despegar, aterrizar y volar.

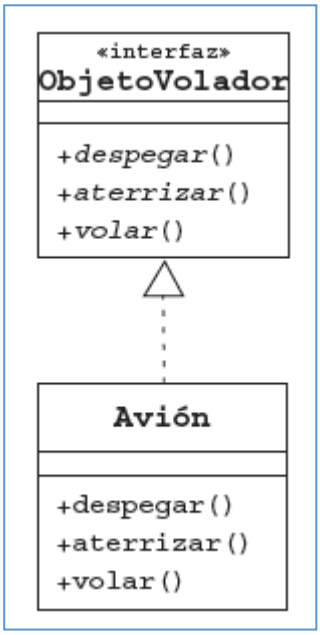


Gráfico 62. Diagrama UML con interface

Como podemos observar en el diagrama UML, las interfaces se marcan con el indicador <<interface>> en la sección de nombre.

Por otro lado, utilizamos una flecha discontinua para indicar la clase que implementa una interface.

A continuación vemos el código fuente de la interface y la clase.

```
public interface ObjetoVolador {

    public void despegar();

    public void aterrizar();

    public void volar();
}
```

Gráfico 63. Código interface ObjetoVolador

```java
public class Avion implements ObjetoVolador {

    public void despegar() {
        // acelerar hasta despegar
        // subir el tren de aterrizaje
    }

    public void aterrizar() {
        // bajar el tren de aterrizaje
        // decelerar y desplegar flaps hasta tocar tierra
        // frenar
    }

    public void volar() {
        // mantener los motores en marcha
    }
}
```

Gráfico 64. Implementación de la interface ObjetoVolador en la clase Avion

Puede haber muchas clases que implementen la interfaz ObjetoVolador. Un Avion, un ave o Superman pueden volar.

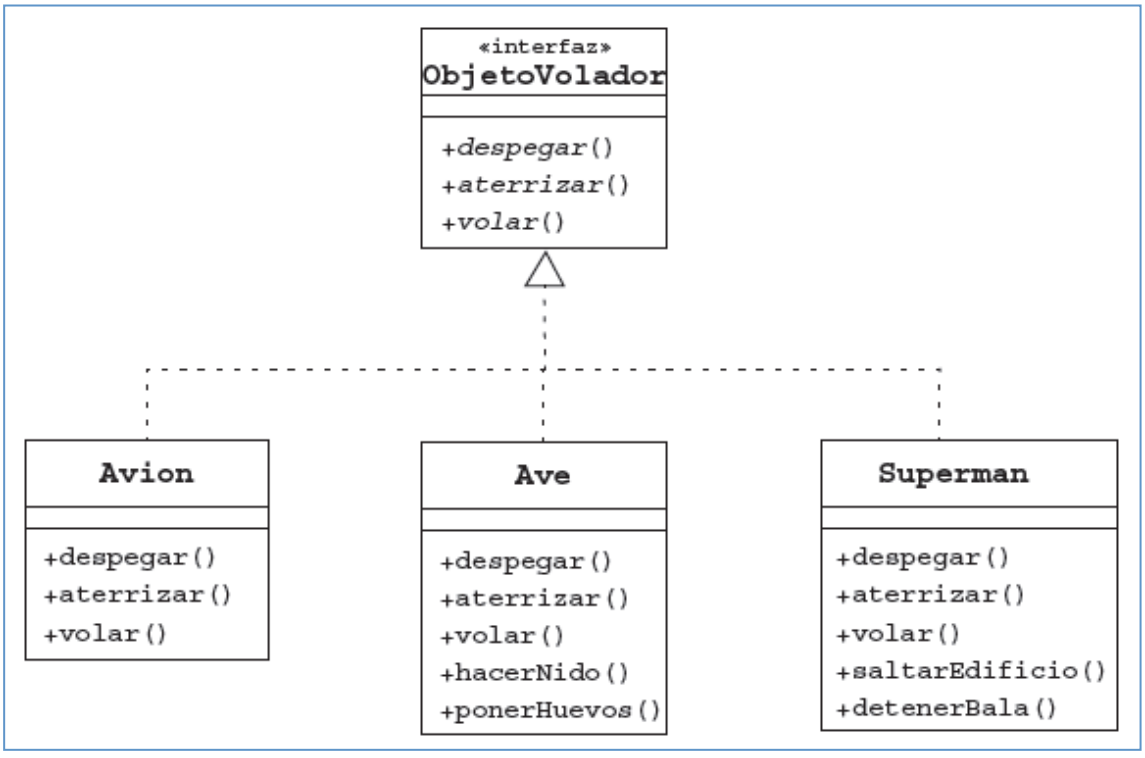


Gráfico 65. Varias clases implementan la misma interface

HERENCIA MULTIPLE

Suena como una herencia múltiple, pero no lo es. El peligro de la herencia múltiple es que una clase podría heredar dos implementaciones diferentes del mismo método. Esto no es posible con las interfaces, porque una declaración de método de una interfaz no proporciona ninguna implementación.

El secreto para similar la herencia múltiple es porque una clase sólo puede heredar de una sola clase pero puede implementar varias interfaces.

A continuación mostramos un diagrama UML donde se puede apreciar una herencia múltiple.

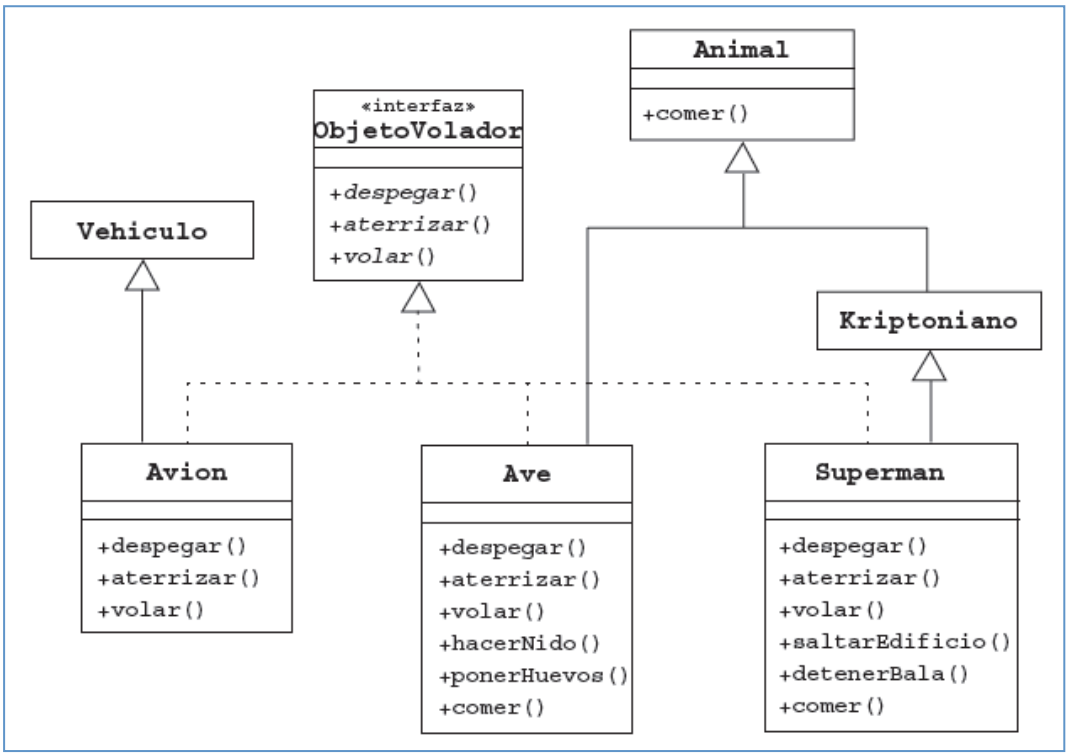


Gráfico 66. Diagrama UML con herencia múltiple

Veamos una implementación de herencia múltiple, según el diagrama la clase Ave hereda de la clase Animal y a su vez implementa la interface ObjetoVolador.

```
public class Animal {

    public void comer(){
        // los animales comen
    }

}
```

Gráfico 67. Implementación de la clase Animal

```
public class Ave extends Animal implements ObjetoVolador {

    public void despegar() { /* implementación de despegar */ }

    public void aterrizar() { /* implementación de aterrizar */ }

    public void volar() { /* implementación de volar */ }

    public void hacerNido() { /* comportamiento de nidificación */ }

    public void ponerHuevos() { /* implementación de puesta de huevos */ }

    @Override
    public void comer() { /* sobrescritura de la acción de comer */ }
}
```

Gráfico 68. Implementación de la clase Ave simulando herencia múltiple

La cláusula extends debe especificarse antes que la cláusula implements. La clase Ave puede suministrar sus propios métodos (hacerNido y ponerHuevos), así como sobrescribir los métodos de la clase Animal (comer).

MULTIPLES INTERFACES

Una clase puede implementar varias interfaces. El Hidroavion no sólo puede volar, sino también navegar. La clase Hidroavion amplía la clase Avion, con lo que hereda esa implementación de la interfaz ObjetoVolador. La clase Hidroavión también implementa la interfaz Nautico.

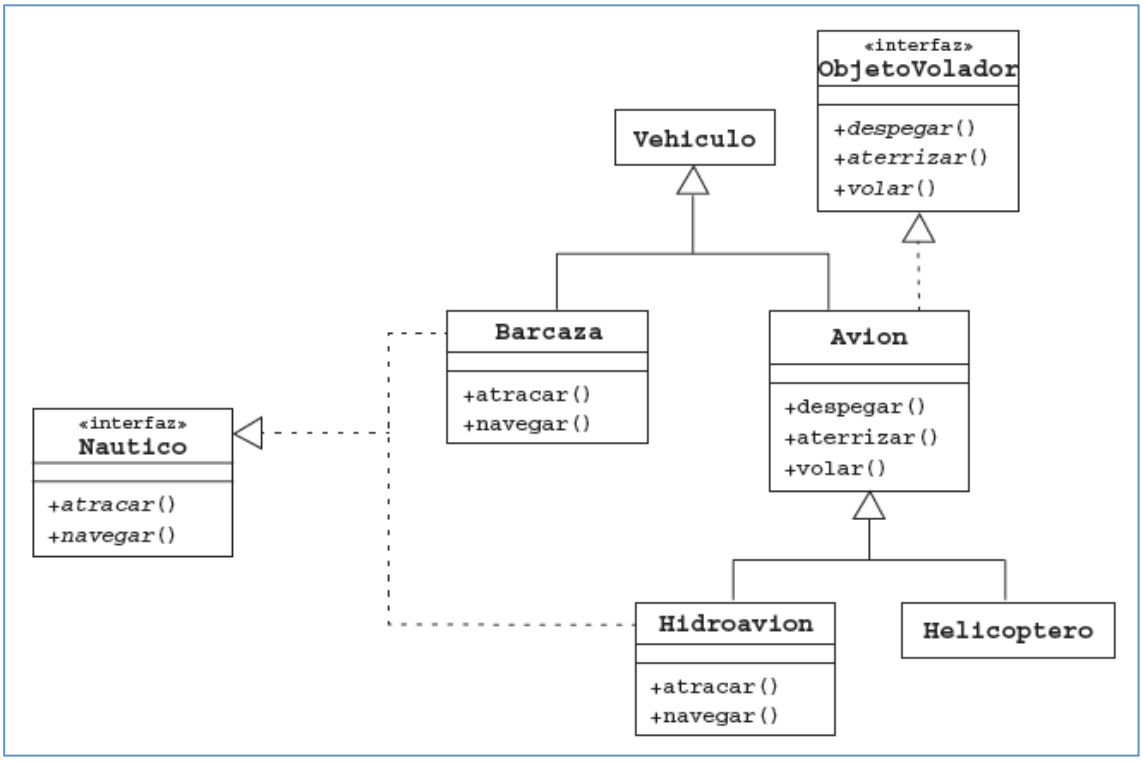


Gráfico 69. Diagrama UML con varias interfaces

USOS DE LAS INTERFACES

Las interfaces se utilizan para:

- Declarar métodos que serán implementados por una o varias clases.
- Dar a conocer la interfaz de programación de un objeto sin revelar el verdadero cuerpo de la clase (esto puede ser útil al distribuir un paquete de clases a otros desarrolladores).
- Identificar las similitudes entre clases no relacionadas sin tener que establecer ninguna relación entre ellas.
- Simular la herencia múltiple declarando una clase que implemente varias interfaces.

POLIMORFISMO

Según el siguiente diagrama de clases. La clase Gerente hereda de la clase Empleado y la clase Director hereda de la clase Gerente. También podemos decir que la clase Director hereda indirectamente de Empleado.

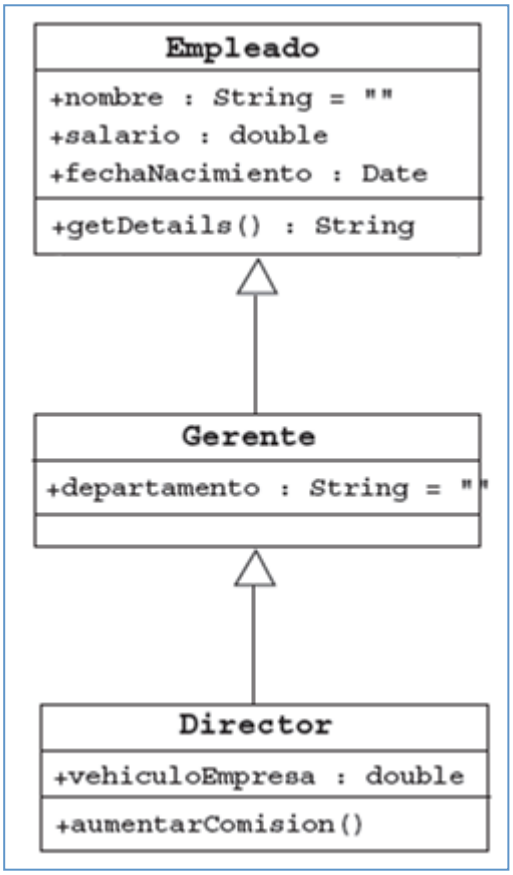


Gráfico 70. Diagrama de clases para aplicar polimorfismo

RELACION IS-A

Cuando utilizamos herencia se establece una relación is-a (traducido es-un). Según el diagrama anterior se establecen las siguientes relaciones:

- Gerente es un Empleado
- Director es un Empleado
- Director es un Gerente

Además de forma implícita, sabemos que Empleado hereda de Object por lo cual también se establecen las siguientes relaciones:

- Empleado es un Object
- Gerente es un Object
- Director es un Object

REPRESENTACION DE LOS OBJETOS CREADOS

Habéis oído hablar de las muñecas matrioskas? Son esas muñecas rusas que va una dentro de otra. Pues bien, vamos a pensar que cada objeto creado anteriormente es una muñeca. Por lo cual lo podríamos representar así.

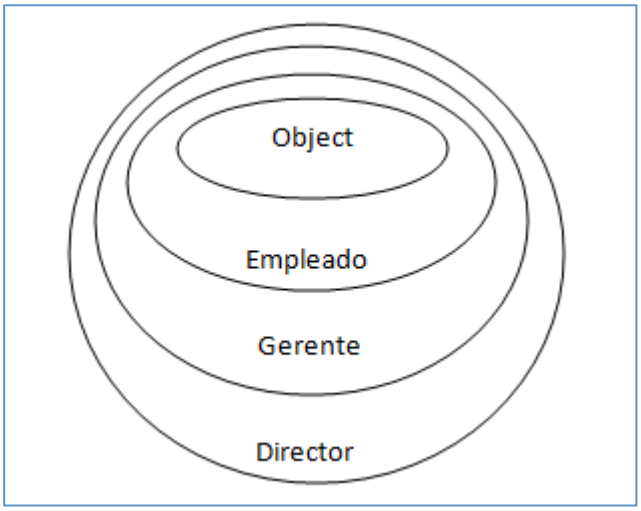


Gráfico 71. Representación de objetos por herencia

Es como decir que dentro del objeto Director hay un objeto Gerente y dentro de este un objeto Empleado y dentro de este a su vez un objeto de tipo Object.

Esta representación nos va a ayudar a entender otra de las características de la Programación Orientada a Objetos. Nos referimos al polimorfismo.

Un objeto sólo tiene una forma (aquella que se le asigna cuando se construye). Sin embargo, una variable es polimórfica porque puede hacer referencia a objetos de distintas formas. Lo que queremos decir, es que el objeto no cambia. Lo que cambia es la forma como lo vemos.

El lenguaje Java, como la mayoría de los lenguajes de programación OO, en realidad permite hacer referencia a un objeto con una variable que es uno de los tipos de una superclase. Por tanto, es posible decir:

```
Empleado e = new Gerente(); //válido
```

Gráfico 72. Ejemplo polimorfismo

Utilizando la variable e tal cual, puede acceder únicamente a las partes del objeto que son componentes de Empleado; las partes específicas de Gerente están ocultas. Esto es porque, por lo que respecta al compilador, e es un Empleado, no un Gerente. Por lo tanto, no se permite lo siguiente:

```
// Intento no permitido de asignar un atributo de Gerente
e.departamento = "Ventas";
// la variable se declara como un tipo de Empleado
// a pesar de que el objeto Gerente tiene ese atributo
```

Gráfico 73. Error en el uso de polimorfismo

LLAMADA A METODOS VIRTUALES

Supongamos que es cierto lo siguiente:

Empleado e = new Empleado();

Gerente m = new Gerente();

Si hace una llamada a e.getDetails() y a m.getDetails(), está llamando a diferentes comportamientos. El objeto Empleado ejecuta la versión de getDetails() asociada a la clase Empleado y el objeto Gerente ejecuta la versión de getDetails() asociada a la clase Gerente.

Lo que no resulta tan obvio es lo que ocurre si tenemos el código siguiente:

Empleado e = new Gerente();

e.getDetails();

O algo parecido, como un argumento de un método general o un elemento de una colección heterogénea.

De hecho, se obtiene el comportamiento asociado al objeto al que hace referencia la variable durante el tiempo de ejecución. El comportamiento no está determinado por el tipo de la variable en el momento de la compilación. Este es un aspecto del polimorfismo y un aspecto importante de los lenguajes orientados a objetos. Este comportamiento a menudo recibe el nombre de llamada a métodos virtuales.

En el ejemplo anterior, el método e.getDetails() ejecutado procede del tipo real del objeto, un Gerente.

COLECCIONES HETEROGÉNEAS

Es posible crear colecciones de objetos que tienen una clase común. Este tipo de colecciones se denominan **homogéneas**. Por ejemplo:

```
MyDate[]  dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

Gráfico 74. Ejemplo de colecciones homogéneas

El lenguaje Java incluye una clase Object que permite crear colecciones de todos los tipos de elementos porque todas las clases amplían la clase Object. Estas colecciones se denominan **heterogéneas**.

Una colección heterogénea es aquella que se compone de elementos dispares. En los lenguajes OO, es posible crear colecciones de numerosos elementos. Todas ellas tienen una clase de origen común: la clase Object. Por ejemplo:

```
Empleado [] personal = new Empleado[1024];
personal[0] = new Gerente();
personal[1] = new Empleado();
personal[2] = new Ingeniero();
```

Gráfico 75. Ejemplo de colecciones heterogéneas

Es posible escribir incluso un método de ordenación que coloque los empleados por orden de edad o salario, con independencia de que algunos de esos empleados sean gerentes.

ARGUMENTOS POLIMÓRFICOS

Es posible escribir métodos que acepten un objeto genérico (en este caso, la clase Empleado) y funcionen adecuadamente con los objetos de cualquier subclase de dicho objeto. Por ejemplo, podría generarse un método en una clase de aplicación que tome un empleado y lo compare con un determinado umbral de salario para determinar las obligaciones fiscales de ese empleado. Si se utilizan las características polimórficas, es posible hacerlo como sigue:

```
public class ServicioImpuestos {
  public TipoImpositivo hallarTipoImpositivo(Empleado e) {
    // hace los cálculos y devuelve el tipo impositivo para e
  }
}

// Mientras tanto, en otra parte de la clase de aplicación
ServicioImpuestos svcImp = new ServicioImpuestos();
Gerente m = new Gerente();
TipoImpositivo t = svcImp.hallarTipoImpositivo(m);
```

Gráfico 76. Ejemplo de argumentos polimórficos

Esto es válido porque un Gerente es un Empleado. No obstante, el método hallarTipoImpositivo sólo tiene acceso a los miembros (variables y métodos) que estén definidos en la clase Empleado.

OPERADOR INSTANCEOF

Dado que es posible pasar objetos de un lugar a otro utilizando referencias a sus superclases, a veces conviene saber de qué objetos reales se dispone. Éste es el propósito del operador instanceof.

Si recibe un objeto utilizando una referencia del tipo Empleado, puede que resulte ser un Gerente o un Ingeniero. Puede comprobarlo utilizando instanceof de la forma siguiente:

```
public void hacerAlgo(Empleado e) {
  if ( e instanceof Gerente ) {
    // Procesar un Gerente
  } else if ( e instanceof Ingeniero ) {
    // Procesar un Ingeniero
  } else {
    // Procesar cualquier otro tipo de Empleado
  }
}
```

Gráfico 77. Ejemplo operador instanceof

CONVERSIÓN DE OBJETOS

En casos en los que se ha recibido una referencia a una clase de nivel superior y se ha comprobado mediante el operador instanceof que el objeto es una subclase concreta, es posible acceder a toda la funcionalidad del objeto convirtiendo la referencia.

```
public void hacerAlgo(Empleado e) {
  if ( e instanceof Gerente ) {
    Gerente m = (Gerente) e;
    System.out.println("Éste es el gerente de "
                       + m.getDepartamento());
  }
  // resto de la operación
}
```

Gráfico 78. Ejemplo conversión de objetos

Si no realiza la conversión, el intento de ejecutar e.getDepartamento() fracasará porque el compilador no puede localizar ningún método llamado getDepartamento en la clase Empleado.

Si no realiza la comprobación con instanceof, corre el riesgo de que falle la conversión. En general, cualquier intento de convertir una referencia a un objeto se somete a diferentes comprobaciones:

- Las conversiones en dirección ascendente dentro de la jerarquía de clases siempre están permitidas y, de hecho, no precisan el operador de conversión. Se pueden hacer mediante una simple asignación.

- En el caso de las conversiones en dirección descendente, el compilador debe poder considerar que la conversión es, al menos, posible. Por ejemplo, no se permitirá ningún intento de convertir una referencia a Gerente en una referencia a Ingeniero, porque el Ingeniero no es un Gerente. La clase de destino de la conversión tiene que ser una subclase del tipo de referencia actual.
- Si el compilador permite la conversión, el tipo de objeto se comprueba durante el tiempo de ejecución. Por ejemplo, si se omite la comprobación instanceof en el código y el objeto que se va a convertir en realidad no es del tipo en el que se va a convertir, se producirá un error de tiempo de ejecución (excepción).

CLASES ENVOLVENTES

El lenguaje Java no considera los tipos de datos primitivos como objetos. Por ejemplo, los datos numéricos, booleanos y de caracteres se tratan de forma primitiva para mantener la eficiencia. A fin de poder manejar los tipos primitivos como objetos, Java proporciona las llamadas clases envoltorio o clases envolventes. Estos elementos quedan envueltos por un objeto que se crea en torno a ellos. Cada tipo primitivo en Java tiene una clase envoltorio asociada en el paquete java.lang. Cada clase envoltorio encapsula un único valor primitivo.

Las clases envolventes implementan objetos inmutables. Esto significa que, una vez inicializado el valor primitivo en su envoltorio, no hay forma de cambiar ese valor.

A continuación se muestra una tabla en la cual se recogen las diferentes clases envolventes.

Tipo de datos primitivos	Clase envoltorio
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Gráfico 79. Clases envolventes

Los objetos de clase envoltorio se construyen pasando el valor que debe encapsularse al constructor adecuado.

```
int pInt = 420;
Integer wInt = new Integer(pInt); // esto se denomina
boxing
int p2 = wInt.intValue(); // esto se denomina unboxing
```

Gráfico 80. Ejemplos de conversión de primitivos en clases envoltorio

Las clases envoltorio son útiles para convertir datos primitivos debido a la gran cantidad de métodos disponibles con este tipo de clases, por ejemplo:

```
int x = Integer.valueOf(str).intValue();

or:

int x = Integer.parseInt(str);
```

Gráfico 81. Ejemplo de uso de la clase envolvente Integer

AUTOBOXING

Si necesita cambiar los tipos de datos primitivos en su objeto equivalente (operación denominada boxing), necesita utilizar las clases envoltorio.

Igualmente, para obtener el tipo de dato primitivo a partir de la referencia del objeto (lo que se denomina unboxing), también necesita usar los métodos de las clases envoltorio. Todas estas operaciones de conversión pueden complicar el código y dificultar su comprensión. En la versión 5.0 de J2SE se ha introducido una función de conversión automática denominada **autoboxing** que permite asignar y recuperar los tipos primitivos sin necesidad de usar clases envoltorio.

El siguiente ejemplo contiene dos casos sencillos de conversión y recuperación automática de primitivos (autoboxing y autounboxing).

```
int pInt = 420;
Integer wInt = pInt; // esto se denomina autoboxing
int p2 = wInt; // esto se denomina autounboxing
```

Gráfico 82. Ejemplo de autoboxing

El compilador de J2SE versión 5.0 ahora creará el objeto envoltorio automáticamente cuando se asigne un primitivo a una variable del tipo de clase envoltorio. Asimismo, el compilador extraerá el valor primitivo cuando realice la asignación de un objeto envoltorio a una variable de tipos primitivos.

Esto puede hacerse al pasar parámetros a métodos o, incluso, dentro de las expresiones aritméticas.

RECURSOS ESTÁTICOS

La palabra **static** declara miembros (atributos, métodos y clases anidadas) que están asociados a una clase en vez de a una instancia de la clase.

ATRIBUTOS DE CLASE

A veces resulta útil tener una variable compartida por todas las instancias de una clase. Esta variable podría utilizarse, por ejemplo, como base para la comunicación entre instancias o para llevar el control del número de instancias que se han creado.

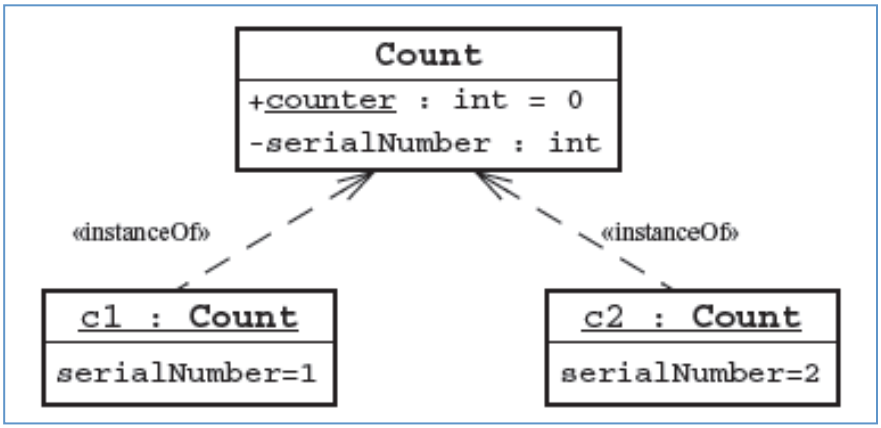


Gráfico 83. Diagrama UML de la clase Count y dos instancias únicas

El modo compartido se consigue marcando la variable con la palabra static. Este tipo de variables a veces se denominan variables de clase para distinguirlas de los miembros o las variables de instancia, que no se comparten.

```
public class Counter {

    private int serialNumber;
    public static int counter = 0;

    public Counter() {
        counter++;
        serialNumber = counter;
    }
}
```

Gráfico 84. Ejemplo de variable estática

En este ejemplo, a cada objeto que se crea se le asigna un número de serie exclusivo que empieza desde 1 y continúa en sentido ascendente. Todas las instancias comparten la variable counter por lo que, cuando el constructor de un objeto incrementa counter, el siguiente objeto que se crea recibe el valor incrementado.

Si una variable static no se marca como private, es posible acceder a ella desde fuera de la clase. Para hacerlo, no se necesita ninguna instancia de la clase, basta hacer referencia a ella mediante el nombre de la clase.

```
public class OtraClase {

    public void incrementNumber() {
        Counter.counter++;
    }
}
```

Gráfico 85. Acceso a una variable estática fuera de la clase

MÉTODOS DE CLASE

Cuando no hay disponible ninguna instancia de un determinado objeto, es necesario acceder al código del programa. Los métodos marcados con la palabra static pueden utilizarse de esta forma y algunas veces se denominan métodos de clase.

```java
public class Count2 {

    private int serialNumber;
    private static int counter = 0;

    public static int getTotalCount() {
        return counter;
    }

    public Count2() {
        counter++;
        serialNumber = counter;
    }
}
```

Gráfico 86. Ejemplo método estático

Debe acceder a los métodos estáticos utilizando el nombre de la clase en lugar de una referencia al objeto, de la forma siguiente:

```java
public class Main {

    public static void main(String[] args) {
        System.out.println("El número del contador es "
                + Count2.getTotalCount());
        Count2 count1 = new Count2();
        System.out.println("El número del contador es "
                + Count2.getTotalCount());
    }
}
```

Gráfico 87. Ejemplo invocación a un método estático

Dado que puede llamar a un método static sin necesidad de tener una instancia de la clase a la que pertenece, no existe ningún valor this. Como consecuencia, un método static no puede acceder a ninguna variable salvo a las variables locales, los atributos static y sus parámetros. Cualquier intento de acceder a atributos que no sean estáticos provoca un error de compilación.

Los atributos no estáticos se limitan a una instancia y sólo se puede acceder a ellos mediante referencias a esa instancia.

```
public class Count3 {
  private int serialNumber;
  private static int counter = 0;

  public static int getSerialNumber() {
    return serialNumber;  // ERROR DE COMPILACIÓN
  }
}
```

Gráfico 88. Error al intentar acceder a un recurso no estático

Cuando use métodos estáticos, debe tener en cuenta lo siguiente:

- No es posible sobrescribir un método static pero sí puede ocultarse. Para poder sobrescribir un método, tiene que ser no estático. La existencia de dos métodos estáticos con la misma firma en una jerarquía de clases simplemente significa que son dos métodos de clase independientes. Si se aplica un método de clase a una referencia de objeto, el método llamado será el correspondiente a la clase para la que se haya declarado la variable.
- El método main() es un método static porque JVM no crea ninguna instancia de la clase cuando lo ejecuta. Por tanto, si tiene datos de miembros, debe crear un objeto para acceder a ellos.

INICIALIZADORES ESTÁTICOS

Una clase puede contener código en bloques estáticos que no forman parte de los métodos normales. El código de los bloques estáticos se ejecuta una vez que se ha cargado la clase. Si una clase contiene varios bloques estáticos, éstos se ejecutan por orden de aparición en la clase.

```
public class Count4 {

    public static int counter;

    static {
        counter = Integer.getInteger("myApp.Count4.counter").intValue();
    }
}
```

Gráfico 89. Ejemplo inicializador estático

En el código de la clase Count4 se ha utilizado un método estático de la clase Integer getInteger(String), el cual devuelve un objeto Integer que representa el valor de una propiedad del sistema. Esta propiedad, denominada myApp.Count4.counter, se define en la línea de comandos con la opción -D. El método intValue del objeto Integer devuelve el valor como un tipo int.

```
public class TestStaticInit {

    public static void main(String[] args) {
        System.out.println("counter = " + Count4.counter);
    }
}
```

Gráfico 90. Clase principal que muestra la variable estática

El resultado es el siguiente:

```
java -DmyApp.Count4.counter=47 TestStaticInit
counter = 47
```

Gráfico 91. Ejemplo de ejecución

PALABRA CLAVE FINAL

La palabra clave final puede tener diferentes significados según el contexto en el cual se utilice.

PROPIEDAD FINAL

Cuando declaramos una propiedad final estamos creando una constante. Una constante es una variable que una vez asignado un valor, este no se puede modificar.

En el siguiente ejemplo, vemos la propiedad id declarada como final, esta se inicializa en el constructor a partir de un contador estático.

Una vez que hemos asignado un valor a la propiedad id este no podrá ser modificado.

```java
public class Persona {

    final int id;
    String nombre;
    String apellido;

    static int contador = 0;

    public Persona() {
        id = ++contador;
    }

    public Persona(String nombre, String apellido) {
        id = ++contador;
        this.nombre = nombre;
        this.apellido = apellido;
    }

}
```

Gráfico 92. Ejemplo propiedad final

METODO FINAL

Cuando declaramos un método como final, este no se puede sobrescribir.

En el siguiente ejemplo hemos agregado el método mostrarInformacion a la clase Persona. Dicho método lo hemos declarado como final lo que significa que las subclases de Persona no podrán redefinir este método.

```java
public final void mostrarInformacion(){
    System.out.println("ID: " + id +
                       " NOMBRE: " + nombre +
                       " APELLIDO: " + apellido);
}
```

Gráfico 93. Ejemplo método final

CLASE FINAL

Una clase final es una clase que no soporta herencia, esto es, no se puede crear una subclase de ella.

```java
public final class Alumno {

    private int id;
    private String nombre;
    private float notaMedia;

    public Alumno() {
    }

    public Alumno(int id, String nombre, float notaMedia) {
        this.id = id;
        this.nombre = nombre;
        this.notaMedia = notaMedia;
    }
```

Gráfico 94. Ejemplo fragmento de clase final

TIPOS ENUMERADOS

Una práctica habitual en programación es tener un número finito de nombres simbólicos que representan los valores de un atributo. Por ejemplo, sería posible crear un conjunto de símbolos que representasen los palos de una baraja de cartas: PICAS, CORAZONES, TREBOLES y DIAMANTES. Esta forma de representación se suele denominar tipo enumerado.

La versión 5.0 de Java SE incluye una modalidad de tipos enumerados que mantiene la seguridad de los tipos (typesafe enum). En el siguiente ejemplo se muestra un tipo enumerado para representar los palos de una baraja de cartas. Piense en el tipo Palo como en una clase con un conjunto finito de valores que reciben los nombres simbólicos incluidos en la definición del tipo. Por ejemplo, Palo.PICAS es del tipo Palo.

```java
public enum Palo {

    PICAS,
    CORAZONES,
    TREBOLES,
    DIAMANTES
}
```

Gráfico 95. Ejemplo de tipo enumerado

El siguiente código muestra cómo la clase NaipeBaraja utiliza el tipo Palo para el tipo de datos del atributo palo.

```
public class NaipeBaraja {
    private Palo palo;
    private int rango;

    public NaipeBaraja(Palo palo, int rango) {
        this.palo = palo;
        this.rango = rango;
    }

    public String getNamePalo() {
        String name = "";
        switch (palo) {
            case PICAS:
                name = "Picas";
                break;
            case CORAZONES:
                name = "Corazones";
                break;
            case TREBOLES:
                name = "Treboles";
                break;
            case DIAMANTES:
                name = "Diamantes";
                break;
            default:
            // No es necesario comprobar si hay errores ya que
            // el enum Palo es finito.
        }
        return name;
    }
```

Gráfico 96. Fragmento de clase que utiliza el tipo enumerado

```
public class Main {

    public static void main(String[] args) {
        NaipeBaraja naipe1 = new NaipeBaraja(Palo.PICAS, 2);
        System.out.println("el naipe1 es: " + naipe1.getRango()
                + " de " + naipe1.getNamePalo());

    }
}
```

Gráfico 97. Clase principal donde se crea una carta

TIPOS ENUMERADOS AVANZADOS

Lamentablemente, la clase NaipeBaraja sigue necesitando un método getNamePalo para mostrar en la interfaz gráfica un nombre que resulte fácil de leer al usuario. Si el programa mostrarse el verdadero valor de Palo, mostraría el nombre simbólico del valor del tipo. Por ejemplo, Palo.PICAS aparecería como PICAS. Esto resulta más reconocible para el usuario que “0”, pero no tanto como Picas.

Por otro lado, el nombre del palo no debería formar parte de la clase NaipeBaraja sino, más bien, del tipo Palo. La nueva modalidad de tipos enumerados permite usar atributos y métodos, igual que en las clases normales.

El siguiente código muestra una versión mejorada del primer tipo Palo con un atributo nombre y un método getName. Observe cómo se oculta la información de la forma adecuada con el atributo “private” y el método de acceso “public”.

```
public enum Palo {

    PICAS("Picas"),
    CORAZONES("Corazones"),
    TREBOLES("Treboles"),
    DIAMANTES("Diamantes");

    private final String name;

    private Palo(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Gráfico 98. Tipo enumerado avanzado

Un constructor enum siempre debería utilizar acceso privado. Los argumentos del constructor se suministran después de cada valor declarado. Por ejemplo, en la línea 4, la cadena "Picas" es el argumento del constructor enum para el valor PICAS. Los tipos enumerados pueden tener cualquier cantidad de atributos y métodos.

Por último, se muestra el programa de prueba modificado, que utiliza el nuevo método getName en el tipo Palo. El método getPalo devuelve un valor de Palo y el método getName devuelve el atributo nombre de ese valor de Palo.

```java
public class Main {

    public static void main(String[] args) {
        NaipeBaraja naipe1 = new NaipeBaraja(Palo.PICAS, 2);
        System.out.println("el naipe1 es: " + naipe1.getRango()
                + " de " + naipe1.getPalo().getName());

    }
}
```

Gráfico 99. Clase principal donde se usa el tipo enumerado

CONSULTA API

El API es el conjunto de todos los recursos (clases, interfaces, tipos enumerados, ...etc.) que podemos utilizar en una aplicación. Todo el API está debidamente documentado y debemos aprender como consultarlo.

Para acceder a la documentación del API abrimos un navegador y en cualquier buscador escribimos API Java 6 y nos llevará a la siguiente página:

docs.oracle.com/javase/6/docs/api/

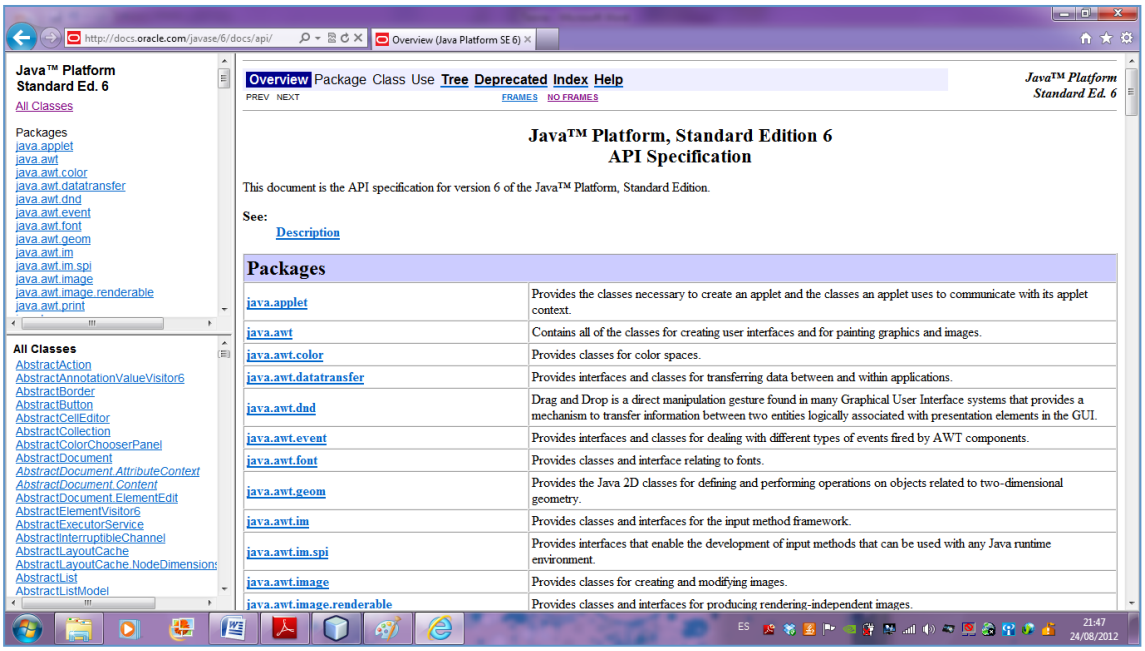


Gráfico 100. Consulta API

En la pagina vista observamos tres marcos o frames.

- En el superior izquierda podremos elegir ver todas las clases o seleccionar un paquete.
- En el marco situado en la parte inferior izquierda vemos todas las clases del paquete seleccionado anteriormente o todas las clases ordenadas alfabéticamente.
- En el marco de la derecha encontraremos toda la información de la clase seleccionada como herencia aplicada, constructores, propiedades, métodos, ...etc.

RECUERDA QUE...

- La herencia me permite heredar los recursos de una superclase siempre y cuando estos no sean constructores, miembros privados o miembros estáticos.
- El polimorfismo se basa en la capacidad de ver un objeto de múltiples formas. El objeto tiene una sola forma, un solo tipo, que será el de la clase instanciada.
- La sobreescritura de métodos me permite redefinir un método heredado.
- La sobrecarga de métodos es la capacidad de tener varios métodos con el mismo nombre. Lo importante es que el compilador pueda distinguir uno de otro.
- La sobrecarga de constructores nos permite crear un objeto con diferente numero de parámetros iniciales.
- Las clases envolventes o envoltorio permiten envolver un tipo primitivo en un objeto.
- Los recursos estáticos o también llamados recursos de clase permiten almacenar datos en la propia clase y no en el objeto.
- Los tipos enumerados permiten crear una variable para un conjunto de datos acotados.
- Una clase abstracta es aquella que tiene uno o varios métodos abstractos, esto es no implementados.
- Una interface define todos sus métodos como abstractos y publicos.
- No se puede crear una instancia de una clase abstracta o interface, pero sí que se pueden utilizar como tipo de una variable.

7.- EXCEPCIONES Y ASERCIONES

Una excepción es un error que ocurre en tiempo de ejecución haciendo que el programa termine de forma inesperada. Veamos un ejemplo:

```
public class AddArguments {
  public static void main(String args[]) {
    int sum = 0;
    for ( String arg : args ) {
      sum += Integer.parseInt(arg);
    }
    System.out.println("Sum = " + sum);
  }
}
```

Gráfico 101. Ejemplo que suma números introducidos como argumentos iniciales

Si se introducen los datos adecuados al ejecutar la clase, el resultado será el esperado.

```
java AddArguments 1 2 3 4
Sum = 10
```

Gráfico 102. Resultado con datos correctos

Ahora bien, que ocurre si no se introducen en el formato esperado? Se produce una excepción y la ejecución del programa se detiene sin llegarse a completar.

```
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

Gráfico 103. Resultado con datos erróneos

CAPTURAR EXCEPCIONES

Para evitar que el programa termine al generarse una excepción, tenemos la opción de capturarla y de esta forma la ejecución continúa.

Realmente una excepción es un objeto de tipo Exception. Cuando la máquina virtual se encuentra con un error que no puede resolver genera una instancia de una clase que hereda de Exception. El tipo de clase elegido será el del tipo de problema encontrado.

Si utilizamos un bloque **try-catch** para capturar excepciones, lo que realmente pretendemos es capturar ese objeto generado por la maquina virtual. De esta forma evitamos que se interrumpa la ejecución.

Utilizamos el bloque **try** para encerrar aquellas instrucciones "peligrosas", nos referimos a las sentencias donde se podría generar una excepción.

El bloque **catch** es donde capturamos la excepción. Veamos el ejemplo anterior mejorado.

```
public class AddArguments2 {
  public static void main(String args[]) {
    try {
      int sum = 0;
      for ( String arg : args ) {
        sum += Integer.parseInt(arg);
      }
      System.out.println("Sum = " + sum);
    } catch (NumberFormatException nfe) {
      System.err.println("One of the command-line "
                         + "arguments is not an integer.");
    }
  }
}
```

Gráfico 104. Ejemplo try-catch

La ejecución daría el siguiente resultado:

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

Gráfico 105. Resultado tras capturar excepciones

Qué está ocurriendo? Por qué no muestra el resultado de la suma? El problema es el siguiente:

Cuando se lee el primer argumento ("1") se realiza la conversión a tipo entero y se acumula a la suma. Cuando se lee el segundo argumento ("two"), al intentar hacer la conversión a tipo entero se genera una excepción de tipo NumberFormatException, en este momento se ejecuta el bloque catch para capturar dicha excepción.

El resto de los argumentos ya no se evalúan y tampoco se imprime el resultado final de la suma.

Para evitar esto, todavía podríamos mejorar más nuestro código haciendo que sólo la instrucción peligrosa forme parte del bloque try. De esta forma, se evaluarán todos los argumentos y se mostrará la suma. Veamos cómo hacerlo.

```
public class AddArguments3 {
  public static void main(String args[]) {
    int sum = 0;
    for ( String arg : args ) {
       try {
         sum += Integer.parseInt(arg);
       } catch (NumberFormatException nfe) {
         System.err.println("[" + arg + "] is not an integer"
                            + " and will not be included in the sum.");
       }
    }
    System.out.println("Sum = " + sum);
  }
}
```

Gráfico 106. Ejemplo mejorado de try-catch

La ejecución dará ahora el siguiente resultado:

```
java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the sum.
[3.0] is not an integer and will not be included in the sum.
Sum = 5
```

Gráfico 107. Resultado del ejemplo mejorado

Podemos utilizar múltiples bloques catch pero debemos seguir la regla de declararlos del más especifico al más genérico. En caso contrario se genera un error de compilación.

También podemos utilizar **try-catch-finally**. El bloque finally nos asegura que se ejecutará siempre haya excepción o no.

Un buen ejemplo de uso del bloque **finally** es cuando abrimos una conexión a la base de datos y siempre se debería de cerrar se haya producido una excepción o no. Pues bien, el cierre de la conexión iría dentro del bloque finally.

MANEJO DE EXCEPCIONES MÚLTIPLES

Se ha mejorado también el manejo de excepciones múltiples, que ahora se pueden atrapar usando un único bloque. Hasta ahora, si tenías un método con, por ejemplo, tres excepciones, tenías que atraparlas de forma individual. Con Java 7 se puede hacer lo siguiente:

```
public void newMultiCatch( ){
        try{
                methodThatThrowsThreeExceptions( );
        }catch(ExceptionOne | ExceptionTwo | ExceptionThree e){
                //maneja las excepciones
        }
}
```

Sin embargo, si tienes excepciones que pertenecen a distintos tipos que deben manejarse de distinta manera, se puede hacer lo siguiente:

```
public void newMultiMultiCatch( ){
        try{
                methodThatThrowsThreeExceptions( );
        }catch (ExceptionOne e){
                //maneja ExceptionOne
        }catch (ExceptionTwo | ExceptionThree e){
                //maneja ExceptionTwo y ExceptionThree
        }
}
```

API EXCEPTION

El API Exception nos permite ver las diferentes clases de las excepciones y la forma en que están organizadas.

Como vemos en el diagrama, la clase **Exception** hereda de la clase **Throwable** y hermana de ella tenemos la clase **Error**.

Los errores no se pueden capturar. Cuando ocurren el programa termina sin que podamos hacer nada.

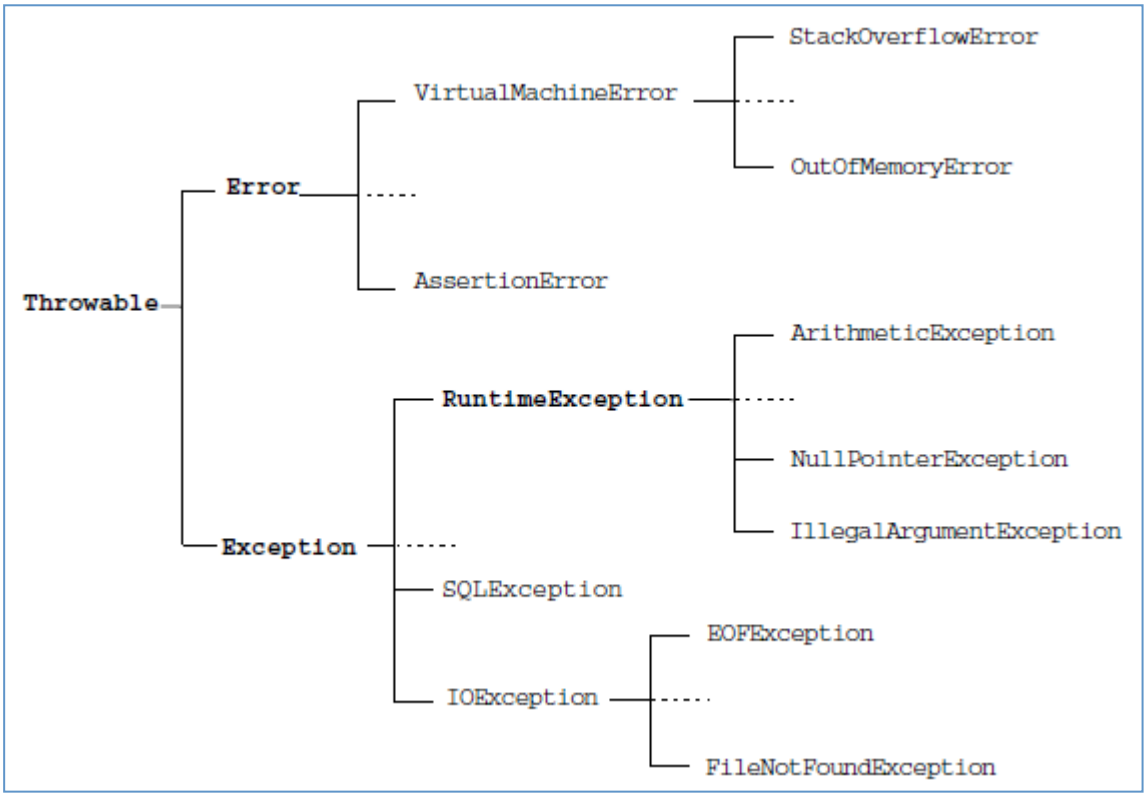


Gráfico 108. Categorías de Excepciones

PROPAGAR EXCEPCIONES

Si no queremos capturar una excepción generada tenemos otra opción que es lanzarla.

Para lanzar una excepción añadimos la clausula **throws** en el método donde se puede generar la excepción.

En este caso, la excepción se propaga a la instrucción donde se ha efectuado la llamada al método.

Veamos ejemplos:

```
void trouble() throws IOException { ... }
void trouble() throws IOException, MyException { ... }
```

Gráfico 109. Métodos que lanzan excepciones

REGLAS DE SOBREESCRITURA DE METODOS QUE LANZAN EXCEPCIONES

Si queremos sobrescribir o redefinir un método con clausula throws aquí tenemos las reglas a seguir:

- Se puede sobrescribir el método eliminado la cláusula throw
- No se puede sobrescribir para que lance más excepciones que las definidas
- Puede lanzar excepciones de una subclase de la definida originalmente.
- No puede lanzar excepciones que sean de la superclase de la definida en un principio.

EXCEPCIONES PERSONALIZADAS

A pesar de que el API Exception nos ofrecen muchas clases para manejar excepciones, muchas veces nos vamos a encontrar con la situación de querer crear nuestra propia excepción, a eso lo denominamos excepciones personalizadas o propias.

Gracias al mecanismo de herencia, para generar nuestra propia excepción es suficiente con crear una clase que herede de la clase Exception. Veamos un ejemplo:

```
public class ServerTimedOutException extends Exception {
  private int port;

  public ServerTimedOutException(String message, int port) {
    super(message);
    this.port = port;
  }

  public int getPort() {
    return port;
  }
}
```

Gráfico 110. Excepción personalizada

La maquina virtual no reconoce nuestra nueva excepción por lo cual no va a generar objetos de este tipo, lo tendremos que hacer nosotros manualmente.

Para generar una excepción personalizada es tan fácil como crear una instancia de la clase y lanzarla con la cláusula throw. Veamos como:

```
public void connectMe(String serverName)
       throws ServerTimedOutException {
  boolean successful;
  int portToConnect = 80;

  successful = open(serverName, portToConnect);

  if ( ! successful ) {
    throw new ServerTimedOutException("Could not connect",
                                      portToConnect);
  }
}
```

Gráfico 111. Creación de excepción personalizada

En este ejemplo estamos propagando la excepción con throws. También podríamos haberla capturado con un bloque try-catch o try-catch-finally.

GESTIÓN DE ASERCIONES

Una aserción es una comprobación que se hace en un punto del programa para verificar que realmente estamos utilizando los datos correctos.

La sintaxis para incluir una aserción es cualquiera de las siguientes:

assert <expresion_booleana> ;

assert < expresion_booleana > : <mensaje_error> ;

Si la expresión booleana se evalúa a false, entonces se lanza un AssertionError. Como vimos anteriormente los errores no se pueden capturar y el programa finaliza.

Veamos un ejemplo: Estamos ejecutando un código y llegados a este punto pensamos que x siempre toma un valor mayor o igual a 0. Por lo cual si la condición devuelve false pensamos que x vale 0.

```
if (x > 0) {
  // do this
} else {
  // do that
}
```

Gráfico 112. Ejemplo sin utilizar aserciones

Pues bien, el uso de las aserciones nos permite verificar esta suposición. Si x vale 0, la expresión de la aserción se evaluará como true y el programa continuará según lo previsto. Pero si x toma un valor negativo entonces la expresión devolverá false y se

generará un AssertionError haciendo que el programa se detenga. Esto es lo que realmente queremos, que el programa lance el error para poder depurarlo.

Veamos el ejemplo anterior mejorado.

```
if (x > 0) {
  // do this
} else {
  assert ( x == 0 );
  // do that, unless x is negative
}
```

Gráfico 113. Ejemplo aserciones

La ventaja de utilizar aserciones es evitar el uso de condicionales innecesarios que una vez depurado todo el código podremos desactivar y no llevarlas a producción.

BUENAS PRÁCTICAS CON ASERCIONES

Debemos saber exactamente qué consecuencias tenemos al desactivar las aserciones. No podemos comprobar mediante aserciones aquellos procesos necesarios en producción.

Se pueden utilizar aserciones para:

- Controlar los invariantes del control de flujo
- Post condiciones e invariantes de la clase
- Invariantes internas

No se deberían utilizar aserciones en los siguientes casos:

- Para chequear los parámetros de un método público.
- No usar métodos en la aserción que puedan causar efectos secundarios, como por ejemplo modificar el valor de una variable.

HABILITAR Y DESHABILITAR ASERCIONES

Por defecto, las aserciones están deshabilitadas. Para habilitarlas recurrimos a las siguientes opciones:

```
java -enableassertions MyProgram
```

o también:

```
java -ea MyProgram
```

RECUERDA QUE...

- Las excepciones son errores que surgen tiempo de ejecución.
- Estas se pueden capturar o lanzar.
- Una aserción es una comprobación que utilizamos únicamente en modo desarrollo, pudiendo habilitarlas o no antes de llevar nuestro código a producción.

8.- COLECCIONES Y GENERICOS

Al trabajar con arrays nos encontrado con una serie de inconvenientes:

- Todos los elementos del array deben ser del mismo tipo
- Al crear el array debemos poner un tamaño inicial, me refiero a especificar el número de elementos que va a contener.
- Los arrays no se pueden redimensionar ya que son estructuras estáticas.

Las colecciones son otra alternativa a los arrays que me permiten almacenar varios elementos en la misma estructura y soluciona los inconvenientes anteriores:

- Los elementos de las colecciones se almacenan como Object por lo cual me permite guardar cualquier tipo de elemento.
- No necesitamos especificar de inicio el número de elementos que va a contener.
- Son estructuras dinámicas por lo cual puedo ir añadiendo tantos elementos como quiera que la colección los admite sin problemas.

API COLLECTIONS

El API Collections contiene interfaces que permiten agrupar objetos en una de las siguientes colecciones:

- **Collection**; Un grupo de objetos denominados elementos, la implementación determina si guardan un orden específico y si se permiten elementos duplicados.
- **Set**; Es un tipo de colección donde no se garantiza que se conserve el orden de entrada de los elementos. Tampoco permite elementos duplicados.
- **List** – En esta colección si se garantiza el orden de entrada y si que se permiten los elementos duplicados.

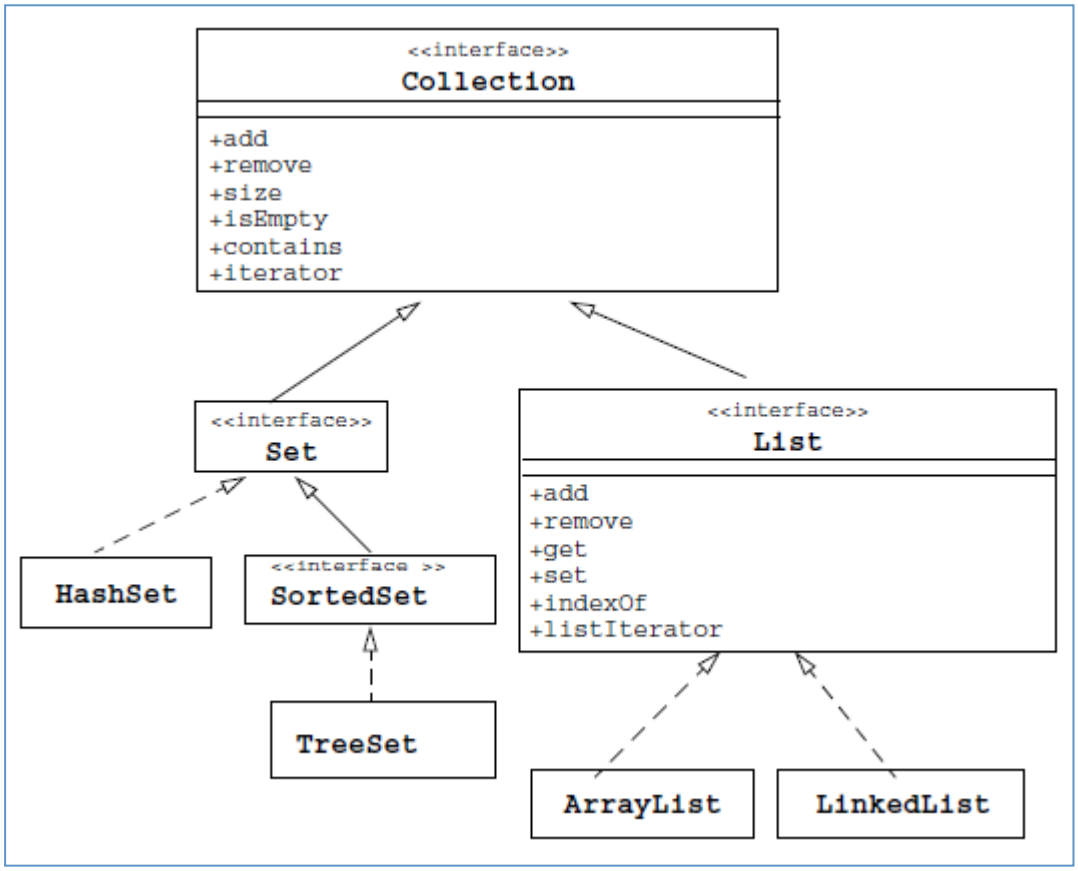


Gráfico 114. API Collections

Las colecciones de tipo SortedSet son las únicas que me permiten almacenar los elementos ordenados. Veremos más adelante como hacerlo.

PRINCIPALES COLECCIONES

A continuación presentamos una tabla donde se recogen las principales clases que implementan las colecciones.

	Hash Table	Array redimensionable	Árbol balanceado	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

SET

Recordamos que una colección de tipo Set no garantiza el orden de entrada de los elementos y tampoco permite introducir elementos duplicados.

En el siguiente ejemplo vemos como crear una colección de este tipo y añadir unos cuantos elementos.

Si intentamos introducir un elemento repetido simplemente lo ignora sin generar una excepción o error de compilación.

```
// crear una coleccion de tipo Set
Set coleccionSet = new HashSet();
coleccionSet.add("uno");
coleccionSet.add("segundo");
coleccionSet.add(new Integer(4));
coleccionSet.add(new Float(3.15));
coleccionSet.add("segundo");  // los repetidos no se añaden
System.out.println(coleccionSet);
```

Gráfico 115. Ejemplo Set

La ejecución de este ejemplo podría mostrar la siguiente salida.

```
[4, 3.15, segundo, uno]
```

LIST

Una colección de tipo List si que conserva el orden de entrada de los elementos y además permite introducir elementos duplicados.

En este ejemplo vemos como crear una colección de tipo List.

```
// crear una coleccion de tipo List
List coleccionList = new ArrayList();
coleccionList.add("uno");
coleccionList.add("segundo");
coleccionList.add(new Integer(4));
coleccionList.add(new Float(3.15));
coleccionList.add("segundo");  // los repetidos SI se añaden
System.out.println(coleccionList);
```

Gráfico 116. Ejemplo List

Tras la ejecución del código este es el resultado que veremos en la consola.

```
[uno, segundo, 4, 3.15, segundo]
```

MAP

Un mapa no se considera una colección ya que no hereda de la interface Collection.

Los elementos de un mapa se forman como clave-valor. Además tienen las siguientes restricciones:

- Las claves duplicadas no están permitidas.
- Una clave solo puede referenciar un valor, no varios.

La interface Map define una serie de métodos para manipular el mapa, los más interesantes son:

- **entrySet**; Devuelve una colección de tipo Set con todos los elementos (clave-valor).
- **keySet**; Devuelve una colección de tipo Set con todas las claves del mapa.
- **values**; Devuelve un objeto de tipo Collection con todos los valores del mapa.

El API de la interface Map es el siguiente:

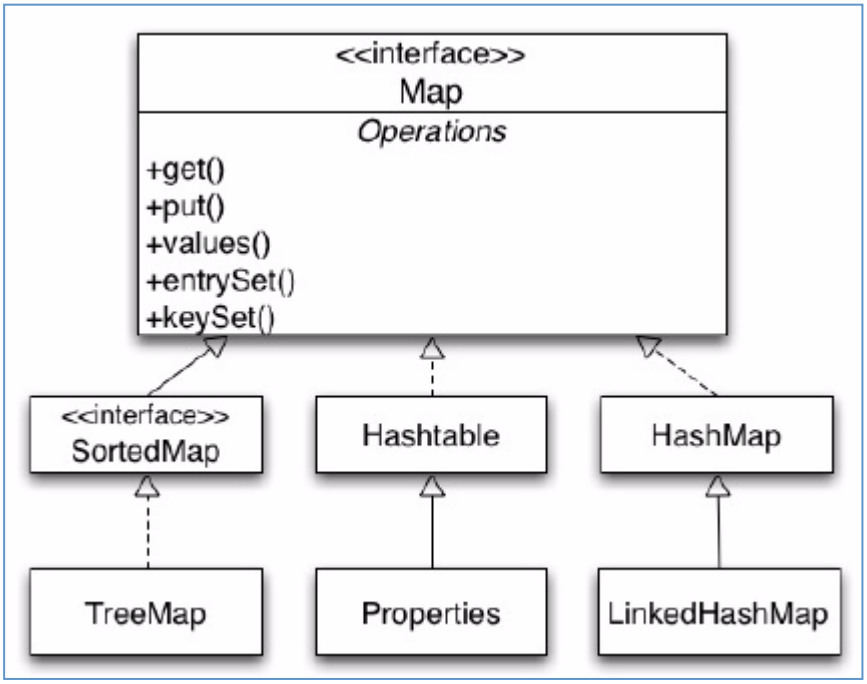


Gráfico 117. API interface Map

Vemos un ejemplo de cómo crear un mapa. Si intentamos introducir un elemento con su clave duplicada, se sobrescribe el valor del elemento.

```
// crear un Mapa
// un mapa no permite claves duplicadas, valores duplicados si.
Map mapa = new HashMap();
mapa.put("1", "uno");
mapa.put("2", "dos");
mapa.put("1", "tres"); // sobreescribe el elemento

// mostrar todas las claves (keys)
System.out.println(mapa.keySet());

// mostrar todos los valores (values)
System.out.println(mapa.values());

// mostrar todos los elementos como pares key-value
System.out.println(mapa.entrySet());
```

Gráfico 118. Ejemplo Map

El resultado tras la ejecución sería el siguiente:

```
[2, 1]
[dos, tres]
[2=dos, 1=tres]
```

ORDENAR COLECCIONES

Cuando mencionamos el concepto ordenar colecciones nos estamos refiriendo a mantener los elementos clasificados dentro de la colección, por ejemplo, alfabéticamente, cronológicamente, ascendente, ...etc.

Para lograr este objetivo utilizamos las interfaces **Comparable** y **Comparator**.

INTERFACE COMPARABLE

La interface Comparable permite definir el orden natural de los elementos. Ejemplos de clases que implementan dicha interface son:

- **String**; En los elementos de este tipo se define un orden alfabético.
- **Date**; En los elementos de este tipo se define un orden cronológico
- **Integer**; En los elementos de este tipo se define un orden numérico

La interface Comparable define un único método:

```
int compareTo(Object o)
```

Este método lo debemos implementar y el valor entero a devolver tendrá el siguiente significado:

- 0; si los objetos a comparar son iguales
- 1; si la instancia en la cual estamos es mayor que el objeto pasado como argumento.
- -1; si la instancia en la cual estamos es menor que el objeto pasado como argumento.

En el siguiente ejemplo, vemos que queremos ordenar las instancias de tipo Cliente atendiendo a su cifra de ventas.

```
public class Cliente implements Comparable{

    String nombre;
    double cifraVentas;
    String nif;

    public int compareTo(Object o) {
        Cliente otroCliente = null;
        int resultado = 0;
        if (o instanceof Cliente){
            otroCliente = (Cliente) o;
        }
        // 0 --> los objetos son iguales
        if (cifraVentas == otroCliente.cifraVentas){
            resultado = 0;
        }
        // 1 --> es mayor
        else if(cifraVentas > otroCliente.cifraVentas)
        {
            resultado = 1;
        }
        // -1 --> es menor
        else if(cifraVentas < otroCliente.cifraVentas)
        {
            resultado = -1;
        }
        return resultado;
    }
```

Gráfico 119. Clase que implementa la interface Comparable

Para mantener los elementos ordenados necesitamos una colección que implemente la interface SortedSet que como vimos antes son las únicas que permiten este propósito.

```
public static void main(String[] args) {
   // crear una coleccion clasifique los objetos --> TreeSet
   TreeSet arbol = new TreeSet();

   arbol.add(new Cliente("Juan", 1500, "12345678-A"));
   arbol.add(new Cliente("Jose", 1200, "1245778-A"));
   arbol.add(new Cliente("Maria", 1800, "2222222-A"));
   arbol.add(new Cliente("Laura", 1100, "88888888-A"));

   Object[] array = arbol.toArray();

   for (int i=0; i<array.length; i++){
       System.out.println((Cliente)array[i]);
   }

}
```

Gráfico 120. Clase principal que añade elementos al árbol para ordenarlos

El resultado de la ejecución sería el siguiente:

```
Cliente{ nombre=Laura cifraVentas=1100.0 nif=88888888-A}
Cliente{ nombre=Jose cifraVentas=1200.0 nif=1245778-A}
Cliente{ nombre=Juan cifraVentas=1500.0 nif=12345678-A}
Cliente{ nombre=Maria cifraVentas=1800.0 nif=2222222-A}
```

Gráfico 121. Resultado de ejecución

INTERFACE COMPARATOR

Al implementar la interface Comparable en la clase Cliente, estamos dando por hecho que todos los objetos Cliente siempre los vamos a ordenar por su cifra de ventas.

Ahora bien. Y si resulta que unas veces quiero ordenarlos por el nombre y otras veces por el nif por ejemplo. La solución nos la aporta la interface Comparator.

La interface Comparator define un único método:

```
int compare(Object o1, Object o2)
```

Gráfico 122. Sintaxis del método compare

También devolvemos un numero entero con el mismo significado que antes.

- 0; si los objetos a comparar son iguales
- 1; si el primer objeto es mayor que el segundo.
- -1; si el primer objeto es menor que el segundo.

Veamos un ejemplo: Ahora la clase Cliente no implementa ninguna interface.

```
public class ClienteSimple{

    String nombre;
    double cifraVentas;
    String nif;

    public ClienteSimple(String nombre, double cifraVentas, String nif) {
        this.nombre = nombre;
        this.cifraVentas = cifraVentas;
        this.nif = nif;
    }
```

Gráfico 123. Clase ClienteSimple

Generamos un comparador en una clase aparte. En este caso para ordenar por nombre.

```
public class ComparadorNombre implements Comparator{

    public int compare(Object o1, Object o2) {
        return (((ClienteSimple)o1).nombre.compareTo(((ClienteSimple)o2).nombre) );
    }

}
```

Gráfico 124. Clase ComparadorNombre

También nos podemos generar otro comparador para ordenar por el nif.

```
public class ComparadorNif implements Comparator{

    public int compare(Object o1, Object o2) {
        return (((ClienteSimple)o1).nif.compareTo(((ClienteSimple)o2).nif) );
    }

}
```

Gráfico 125. Clase ComparadorNif

A la hora de crear la colección que nos permite tener ordenados los objetos, debemos pasarle una instancia del comparador elegido. Esto especifica si queremos ordenar los elementos por el nombre o por el nif. Veamos como:

```
public static void main(String[] args) {
    // crear una coleccion clasifique los objetos --> TreeSet
    ComparadorNombre c = new ComparadorNombre();
    ComparadorNif nif = new ComparadorNif();

    // Elegimos ordenar los clientes por su nombre
    TreeSet arbol = new TreeSet(c);
    //TreeSet arbol = new TreeSet(nif);

    arbol.add(new ClienteSimple("Juan", 1500, "1111-A"));
    arbol.add(new ClienteSimple("Jose", 1200, "3333-B"));
    arbol.add(new ClienteSimple("Maria", 1800, "2222-C"));
    arbol.add(new ClienteSimple("Laura", 1100, "0000-D"));

    Object[] array = arbol.toArray();

    for (int i=0; i<array.length; i++){
        System.out.println((ClienteSimple)array[i]);
    }
}
```

Gráfico 126. Clase principal que añade elementos al árbol para luego ordenarlos

GENÉRICOS

Como hemos mencionado una colección almacena todos sus elementos como Object, por lo cual a la hora de recuperar dichos elementos debemos efectuar el casting apropiado.

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

Gráfico 127. Recuperar elementos de una colección

Utilizar un tipo genérico a la hora de crear la colección nos permite:

- Eliminar la necesidad de casting
- Controlar en tiempo de compilación que todos los elementos son del tipo genérico.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Gráfico 128. Recuperar elementos de una colección usando genéricos

Veamos un ejemplo: Creamos una colección y definimos como tipo genérico String, esto nos permite controlar que todos los elementos introducidos en la colección son de tipo String. Si no fuese así e intentásemos introducir instancias de otro tipo se produciría un error de compilación.

Como observamos en el bucle for-each ahora recuperamos todos los elementos como String y no como Object en el caso de no utilizar genéricos.

```
public static void main(String[] args) {
    Set<String> coleccionSet = new HashSet<String>();
    coleccionSet.add("uno");
    coleccionSet.add("segundo");

    // si no es de tipoString genera un error de compilacion
    // coleccionSet.add(new Integer(4));
    // coleccionSet.add(new Float(3.15));
    coleccionSet.add("segundo");  // los repetidos no se añaden
    System.out.println(coleccionSet);

    for (String elemento : coleccionSet)
            System.out.println(elemento);
}
```

Gráfico 129. Recuperar elementos de una colección usando el método for-each

Aquí tenemos otro ejemplo de cómo utilizar genéricos con mapas.

```
import java.util.*;

public class MapPlayerRepository {
  HashMap<String, String> players;

  public MapPlayerRepository() {
    players = new HashMap<String, String> ();
  }

  public String get(String position) {
    String player = players.get(position);
    return player;
  }

  public void put(String position, String name) {
    players.put(position, name);
  }
```

Gráfico 130. Genéricos con mapas

La siguiente tabla recoge como utilizar los tipos genéricos:

Category	Non Generic Class	Generic Class
Class declaration	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Constructor declaration	`public ArrayList (int capacity);`	`public ArrayList<E> (int capacity);`

Method declaration	`public void add((Object o)`
	
	`public Object get(int index)`
		`public void add(E o)`
		
		`public E get(int index)`

Variable declaration examples	`ArrayList list1;`
	
	`ArrayList list2;`
		`ArrayList <String> list1;`
		
		`ArrayList <Date> list2;`

Instance declaration examples	`list1 = new ArrayList(10);`
	
	`list2 = new ArrayList(10);`
		`list1= new ArrayList<String> (10);`
		
		`list2= new ArrayList<Date> (10);`

Gráfico 131. Tabla que recoge los usos de los tipos genéricos

El API Collections con genéricos:

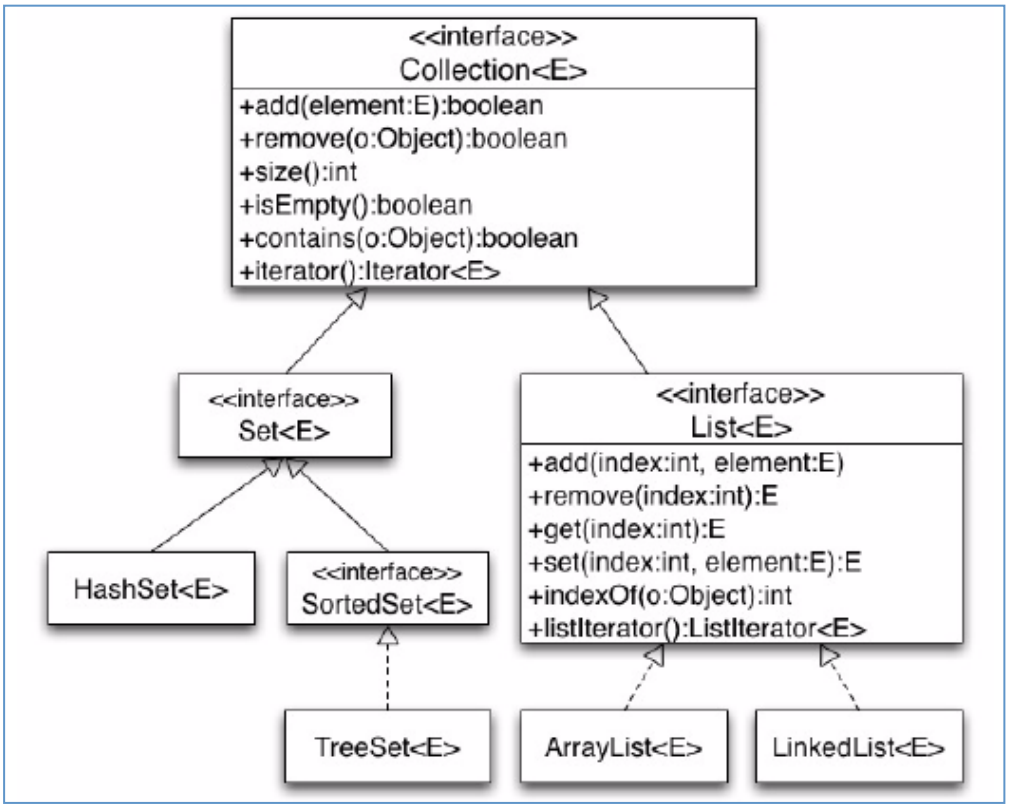


Gráfico 131. API Collections con tipos genéricos

USO DE GENERICOS CON JAVA 7

Una de las mejoras que se reclamaban en este proyecto era sobre el uso de Genéricos. Un ejemplo del uso de Genéricos hasta Java 7 sería como sigue:

```
Map<String, List<Trade>> trades = new TreeMap<String, List<Trade>>( );
```

Esto resulta un poco lioso, porque hay que declarar los tipos a ambos lados. Observad ahora como quedaría usando Java 7:

```
Map<String, List<Trade>> trades = new TreeMap<>( );
```

Mucho mejor, ¿verdad?. Ya no hay que declarar los tipos a la derecha, porque el compilador infiere de qué tipos se trata viendo los que hay a la izquierda. Incluso sería legal omitir el operador de Genéricos (diamond operator), así:

```
trades = new TreeMap( );
```

pero el compilador lanzaría advertencias de seguridad.

ITERADORES

Para recorrer colecciones podemos utilizar iteradores. Veamos un ejemplo:

```
List<Student> list = new ArrayList<Student>();
// add some elements
Iterator<Student> elements = list.iterator();
while (elements.hasNext()) {
  System.out.println(elements.next());
}
```

Gráfico 132. Creación de un iterador con tipo genérico

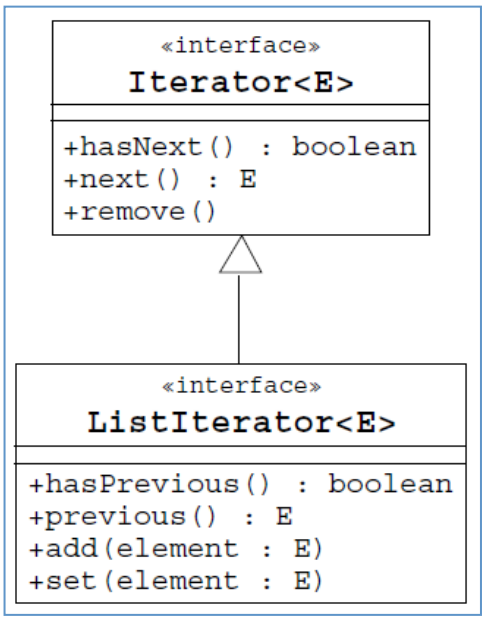


Gráfico 133. Diagrama UML de Iterator

RECUERDA QUE...

- Las colecciones es la forma de poder almacenar varios objetos de forma dinámica
- Podemos ordenar los objetos de una colección con las interfaces: Comparable y Comparator.
- Los tipos genéricos ayudan a comprobar el tipo de los objetos agregados en una colección en tiempo de compilación.

9.- NOVEDADES JAVA 8

EXPRESIONES LAMBDA

Una expresión lambda es un bloque de código que puede ser manipulado como una función sin necesidad de atarla a un identificador y que es posible manipularla en el lenguaje como se manipula cualquier otro valor u objeto.

Se usan varios términos para referirse a lo mismo, entre ellos, función anónima, literal función y constante función son de los que más abundan.

De nada serviría dar soporte en el lenguaje a esa construcción si esos bloques no pudieran ser tratados como valores: asignarlos a variables, pasarlos como argumentos a funciones, ser retornados como funciones, etc. Y entre todas esas cosas, la que sobre sale es la de que el lenguaje permita diseñar funciones de orden superior (forma funcional, funcional, functor); funciones que al menos permiten recibir como entrada funciones y devolver funciones como salida.

Una vez que se tiene ésto, es posible entonces enriquecer el diseño del programa con los famosos closures o clausuras.

Esa necesidad de pasar como argumento un bloque de código a un método de algún objeto siempre ha estado presente en las aplicaciones que se escriben con Java y cualquier otro lenguaje. El ejemplo que casi todo el mundo utiliza para hacer evidente esa necesidad es en el procesamiento de eventos y, en ese contexto casi siempre se toma a Swing como el escenario típico. Desde sus inicios, Java nos ha "obligado" a hacer eso con clases anónimas que implementan alguna interfaz funcional de manera directa o indirecta (indirecta cuando heredamos de alguna clase que ya brinda implementación por default para algunos métodos).

```
boton.addActionListener(new  ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println ("Presionaste el botón y puedes ver este mensaje en consola");
    }
});
```

Gráfico 134. Eventos de boton

Como es conocido, ActionListener es una interfaz funcional (la descripción de una acción específica que debe ser implementada por quien quiera usarla). Esa acción o código como queramos llamarla es lo que requiere el objeto boton para realizar su trabajo. Las expresiones lambda llegan en su auxilio y la porción de código anterior, podemos escribirla ahora así:

```
boton.addActionListener(event -> System.out.println("Presionaste el botón y puedes ver este mensaje en consola");
```

Gráfico 135. Eventos con Lambda

STREAMS

Los streams no son un nuevo tipo de colección son una nueva forma de recorrer las colecciones distinta a los Iterator. La ventaja de los streams es que pueden procesarse de forma serializada o paralela y proporcionan un estilo de operaciones más funcionales. Un flujo consiste un una fuente (una colección), varias operaciones intermedias (de filtrado o transformación) y una operación final que produce un resultado (suma, cuenta...). Los streams son lazy de modo que las operaciones solo se realizan cuando se llama a la operación final, también son eficientes no necesitando en algunos casos procesar todos los elementos del stream para devolver el resultado final.

Tradicionalmente en la API de colecciones la iteración sobre los elementos debíamos proporcionarla de forma externa. Con Java 8 podemos expresarla de la siguiente forma.

```
int alturaMedia = personas.stream()
    .mapToInt((Persona p) -> { return p.getAltura(); })
    .average();
```

Gráfico 136. Recorrido de colecciones con stream()

REFERENCIAS DE MÉTODOS

Todo el código desarrollado previamente a Java 8 no hace uso de las lambdas, pero con las referencias a métodos podemos usar esos métodos ya implementados como si se tratasen de funciones lambdas. Hay diferentes formas de hacer referencias a métodos:

- A métodos estáticos
- A un método de una instancia concreta
- A un método de instancia de una instancia arbitraria de un tipo
- A un constructor

```
// Método estático
Persona::compareAltura

// Método de una instancia concreta
persona::getAltura

// Método de instancia de una instancia arbitraria de un tipo
Persona::getAltura

// Constructor
Persona::new
```

Gráfico 137. Métodos de referencia

INTERFACES FUNCIONALES

Una interfaz funcional es aquella que solo tiene un método abstracto (sin implementación). Algunos ejemplos de interfaces funcionales son Runnable, ActionListener, Comparator y Callable. Para definir una interfaz funcional se puede usar la anotación @FunctionalInterface y pueden representarse con una expresión lambda. En el siguiente ejemplo puede apreciarse que con las interfaces funcionales y las lambdas podemos hacer lo mismo de forma más clara, menos verbosa y con código más legible.

```
Collections.sort(personas, new Comparator<User>() {
    public int compare(Persona p1, Persona p2) {
        return p1.getAltura().compareTo(p2.getAltura());
    }
});
```

Gráfico 138. Clase anónima

En Java 8 podemos hacer:

```
Collections.sort(personas, (Persona u1, Persona u2) -> {
    p2.getAltura().compareTo(p2.getAltura())
});
```

Gráfico 139. Interfaces funcionales

MÉTODOS POR DEFECTO

Hasta ahora las interfaces en Java solo podían definir métodos pero no sus implementaciones. El problema con las interfaces es que cuando se modifican se rompen todas las clases que las usan. Esto se ha resuelto de tal forma que se puedan añadir nuevos métodos con implementación a las interfaces y ha sido necesario para incorporar las lambdas a interfaces existentes como List. En Java 8 las interfaces podrán incorporar implementaciones para algunos de sus métodos, teniendo así algo parecido a herencia múltiple.

```
public interface Math {
    int add(int a, int b);

    default int multiply(int a, int b) {
        return a * b;
    }
}
```

Gráfico 140. Método por defecto

MÉTODOS ESTÁTICOS EN INTERFACES

Además de definir métodos por defecto en las interfaces a partir de ahora podemos definir métodos estáticos. Definiendo métodos estáticos en las interfaces evitaremos tener que crear clases de utilidad. Podremos incluir en un mismo tipo (la interfaz) todos los métodos relacionados.

```
public interface Persona {

    String getNombre();
    int getAltura();

    static String toStringDatos() {
        return getNombre() + " " + getAltura();
    }
}
```

Gráfico 141. Método estático en interfaces

RECUERDA QUE...

- Java 8 es la última versión liberada por Oracle.
- Siempre es interesante consultar la nueva API para ver sus cambios.

ÍNDICE DE GRÁFICOS