



UNIVERSIDAD NACIONAL DE INGENIERIA
FACULTAD DE INGENIERIA INDUSTRIAL Y DE SISTEMAS
SISTEMAS UNI

PROGRAMADOR JAVA



FUNDAMENTOS DE JAVA

INTRODUCCIÓN

Java es un lenguaje de programación orientado a objetos desarrollado por SUN cuya sintaxis está basada en C++, por lo que todos aquellos que estén acostumbrados a trabajar en este lenguaje encontrarán que la migración a Java se produce de forma sencilla y podrán verlo como su evolución natural: un lenguaje con toda la potencia de C++ que elimina todas las estructuras que inducían a confusión y que aumentaban la complejidad del código y que cumple todos los requerimientos actualmente del paradigma de programación orientada a objetos.

BREVE HISTORIA

Java no surgió inicialmente como un lenguaje de programación orientado a la web. Los orígenes se remontan al año 1991 cuando Mosaic (uno de los primeros browsers) o la World Wide Web no eran más que meras ideas interesantes. Los ingenieros de Sun Microsystems estaban desarrollando un lenguaje capaz de ejecutarse sobre productos electrónicos de consumo tales como electrodomésticos.

Simultáneamente James Gosling, el que podría considerarse el padre de Java, estaba trabajando en el desarrollo de una plataforma software de bajo costo e independiente del hardware mediante C++. Por una serie de razones técnicas se decidió crear un nuevo lenguaje, al que se llamó Oak, que debía superar algunas de las deficiencias de C++ tales como problemas relacionados con la herencia múltiple, la conversión automática de tipos, el uso de punteros y la gestión de memoria.

El lenguaje Oak se utilizó en ciertos prototipos de electrónica de consumo, pero en un principio no tuvo el éxito esperado dado que la tecnología quizás era demasiado adelantada a su tiempo. No obstante, lo positivo de estos primeros intentos fue que se desarrollaron algunos de los elementos precursores de los actuales componentes Java; componentes tales como el sistema de tiempo de ejecución y la API.

En 1994 eclosionó el fenómeno web y Oak fue rebautizado como Java. En un momento de inspiración, sus creadores decidieron utilizar el lenguaje para desarrollar un browser al que se llamó WebRunner, que fue ensayado con éxito, arrancando en ese momento el proyecto Java/HotJava. HotJava fue un browser totalmente programado en Java y capaz así mismo de ejecutar código Java.

A lo largo de 1995 tanto Java, su documentación y su código fuente como HotJava pudieron obtenerse para múltiples plataformas al tiempo que se introducía soporte para Java en la versión 2.0 del navegador Netscape.

La versión beta 1 de Java despertó un inusitado interés y se empezó a trabajar para que Java fuera portable a todos los sistemas operativos existentes. En diciembre de 1995 cuando se dio a conocer la versión beta 2 de Java y Microsoft e IBM dieron a conocer su intención de solicitar licencia para aplicar la tecnología Java, su éxito fue ya inevitable.

El 23 de enero 1996 se publicó oficialmente la versión Java 1.0 que ya se podía obtener descargándola de la web. A principios de 1997 aparece la versión 1.1 mejorando mucho la primera versión. Java 1.2 (Java 2) apareció a finales de 1998 incorporando nuevos elementos. Según Sun esta era la primera versión realmente profesional. En mayo del 2000 se lanza la versión 1.3 del J2SE (Java 2 Standar Edition), luego tenemos la versión 1.4, 1.5 (Java 5.0), 1.6 (Java 6), 1.7 (Java 7) y 1.8 (Java 8).

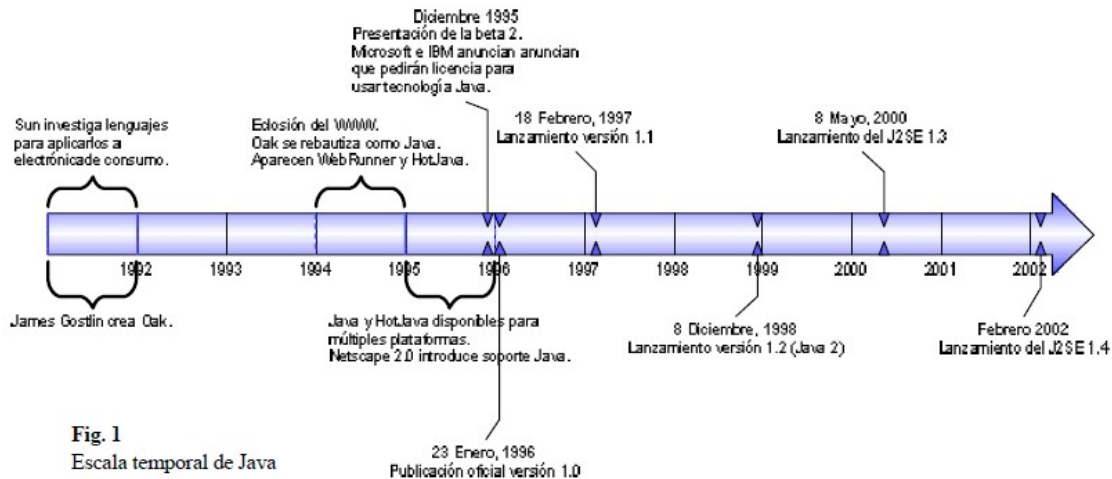


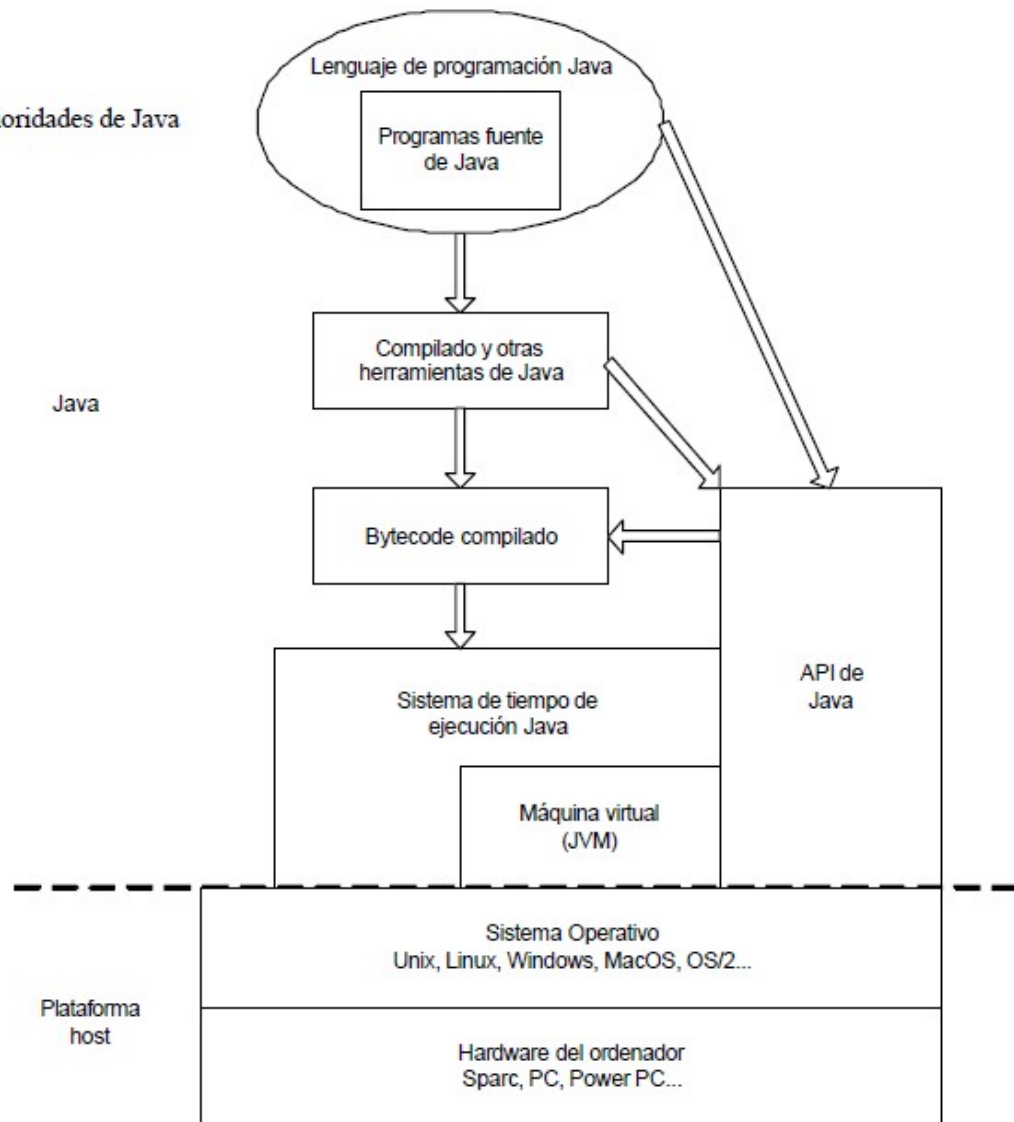
Fig. 1
Escala temporal de Java

¿QUÉ ES JAVA?

Java no es sólo un lenguaje de programación, Java es además un sistema de tiempo de ejecución, un juego de herramientas de desarrollo y una interfaz de programación de aplicaciones (API). Todos estos elementos, así como las relaciones establecidas entre ellos se esquematizan en la figura 2.

El desarrollador de software escribe programas en el lenguaje Java que emplean paquetes de software predefinidos en la API. Luego compila sus programas mediante el compilador Java y el resultado de todo ello es lo que se denomina bytecode compilado. Este bytecode es un archivo independiente de la plataforma que puede ser ejecutado por máquina virtual Java. La máquina virtual puede considerarse como un microprocesador que se apoya encima de la arquitectura concreta en la que se ejecuta, interactuando con el sistema operativo y el hardware, la máquina virtual es por tanto dependiente de la plataforma del host pero no así el bytecode. Necesitaremos tantas máquinas virtuales como plataformas posibles pero el mismo bytecode podrá ejecutarse sin modificación alguna sobre todas ellas.

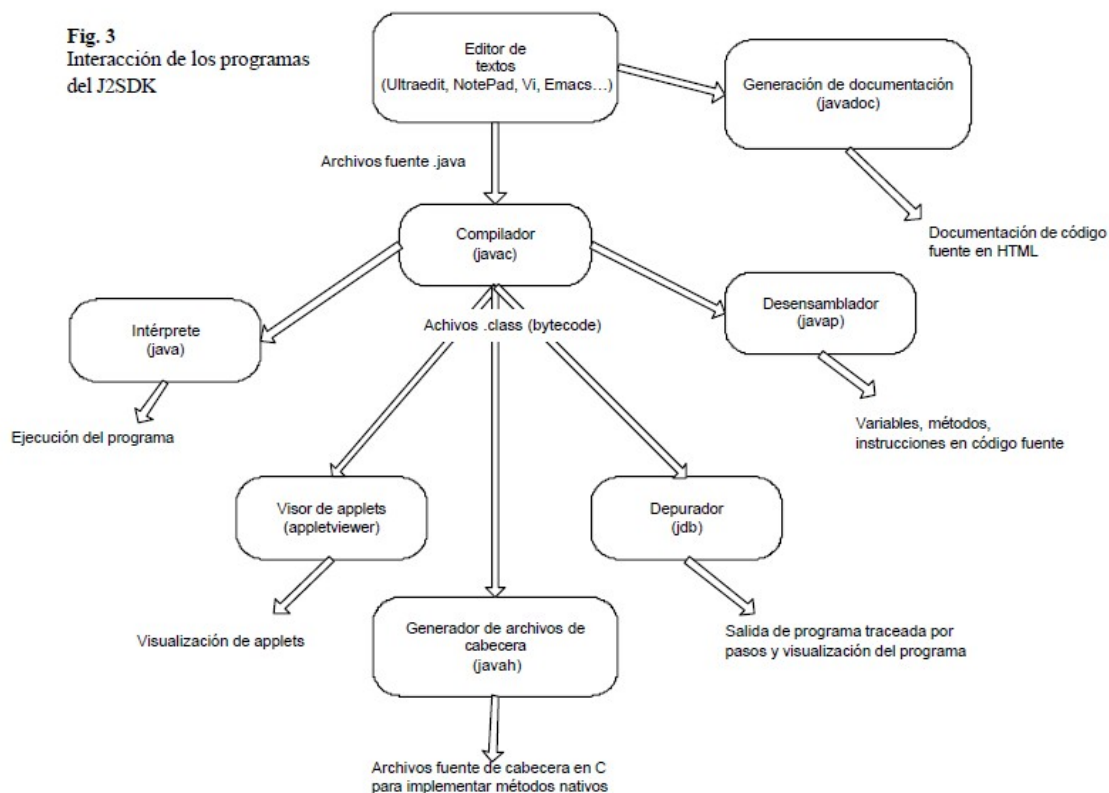
Fig. 2
Las interioridades de Java



¿QUÉ APLICACIONES HAY EN EL SDK?

El entorno de SDK es de tipo línea de comando. Constituye todas las herramientas necesarias para desarrollar aplicaciones Java, así pues, consta del compilador/linkador, del intérprete de aplicaciones, de un debugger, de un visor de applets y de un programa de generación automática de documentación, entre otros. Existen entornos de desarrollo Java ajenos a Sun como pueden ser JCreator, NetBeans, Eclipse, JDeveloper entre otros.

Fig. 3
Interacción de los programas
del J2SDK



COMPILACIÓN: javac

Para generar el archivo .class, que es el bytecode que recibe el entorno de ejecución, se usa el compilador javac que recibe el código fuente en un archivo con extensión .java. Para compilar una aplicación basta con compilar la clase principal, aquella donde está el método main, y después de forma automática se va llamando para compilar todas las clases que necesite. Estas clases deben ser accesibles a través de la variable de entorno CLASSPATH.

Vamos a hacer un pequeño ejemplo para probar el compilador. Abrir el NotePad y copiar las siguientes líneas y a continuación guardar el archivo como HolaUNI.java

```

/**
 *
 * @teacher
 * @blog
 * @email
 */
public class HolaUNI {
    public static void main(String[] args) {
        System.out.println("Hola UNI!");
    }
}
  
```

Después en la línea de comando ejecuta el compilador de la siguiente manera:

```
C:\ProgJava\src>javac HolaUNI.java [Enter]
```

EJECUCIÓN DE APLICACIONES: java

Ahora que ya hemos compilado nuestro flamante HolaUNI.java y estamos ansiosos por ver el resultado de nuestro primer programa, usaremos el intérprete Java para ejecutarlo.

Nada más sencillo que digitar el siguiente comando:

```
C:\ProgJava\src>java HolaUNI [Enter]
Hola UNI!
```

CONCEPTOS GENERALES

En Java todos los programas se construyen a partir de clases, dentro de esas clases encontramos declaraciones de variables (instrucciones atómicas) y procedimientos o funciones (conjuntos de instrucciones).

Comentarios

Los comentarios son cadenas de texto que el programa usa para entender y hacer inteligible su código a otros. Los comentarios son ignorados por el compilador. Java tiene tres tipos de comentarios como los que ilustraremos en el siguiente programa:

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class HolaUNI {

    /* Método main que convierte a una clase en ejecutable */
    public static void main(String[] args) {
        // Envía un mensaje por la salida estándar
        System.out.println("Hola UNI!");
    }

}
```

Identificadores

Los identificadores se usan para nombrar y referirnos a las entidades del lenguaje Java, entidades tales como clases, variables o métodos. Java es sensible a la diferenciación de mayúsculas y minúsculas, así pues una variable contador no es la misma que otra

Contador y pueden, aunque no sea aconsejable, coexistir sin problemas. Un identificador tampoco puede ser igual a una palabra reservada.

Palabras reservadas

Las palabras reservadas por el lenguaje Java son las que se muestran a continuación. Al final de este curso conoceremos el significado de la mayoría de ellas:

abstract, default, if, this, implements, package, throw, boolean, double, import, private, break, else, byte, extends, instanceof, public, try, case, final, int, cast, finally, return, void

Tipos de datos primitivos

Java es un lenguaje fuertemente tipificado, lo que quiere decir toda variable debe ser declarada de un tipo. De los ocho tipos primitivos, hay seis numéricos (cuatro enteros y dos en coma flotante), otro es el carácter y el último es el booleano. La portabilidad de Java garantiza todos los tipos tendrán el mismo tamaño independientemente de la plataforma.

✓ Enteros

Tabla 1. Tipos enteros

Tipo	Tamaño (en bytes)
int	4
short	2
long	8
byte	1

✓ Tipos en coma flotante

Tabla 2. Tipos en coma flotante

Tipo	Tamaño (en bytes)	Cifras significativas
float	4	7
double	8	15

✓ Caracteres: char

Los caracteres se almacenan en el tipo char. Java utiliza el código Unicode de 16 bits (diferenciándose de la mayor parte de lenguajes clásicos, como C/C++, que utilizan ASCII de 8 bits). Unicode es un superconjunto de ASCII, es decir ASCII está contenido en Unicode, pero esto último proporciona muchísimos más caracteres (los dos bytes permiten tener $2^{16}=65.536$ caracteres diferentes frente a los $2^8=256$ caracteres del ASCII extendido). Los caracteres se encierran entre comillas sencillas (") y no dobles ("). Los caracteres de escape, al igual que en C/C++, se preceden de la barra invertida (\). Tenemos los siguientes códigos de escape:

Tabla 4. Códigos de escape

Código de escape	Carácter	Cifra hexadecimal
\b	retroceso	\u0008
\t	tabulación	\u0009
\n	avance de línea	\u000a
\f	avance de papel	\u0006
\r	retroceso de carro	\u000d
\"	comillas normales	\u0022
\'	comillas sencillas	\u0027
\\	barra invertida	\u005c

✓ Booleanos: boolean

A diferencia de C/C++ los valores cierto y falso que se utilizan en expresiones lógicas, no se representan con un entero que toma los valores 1 y 0, sino que existe un tipo destinado a tal efecto, el tipo boolean que toma valores true y false.

DECLARACIONES DE VARIABLES

Una variable es una estructura que se referencia mediante un identificador que nos sirve para almacenar los valores que usamos en nuestra aplicación. Para usar una variable debemos declararla previamente de un tipo. Veamos algunos ejemplos:

```
boolean b;
int numero
float decimal=43.32e3f
int contador=0;
char c='a';
```

Como vemos la forma de declarar variables tiene la siguiente sintaxis:

```
tipoVariable identificadorVariable [= valorInicial];
```

Donde primero indicamos el tipo al que pertenece la variable, a continuación el identificador o nombre de la variable, opcionalmente un valor inicial y finalmente acabamos con un punto y coma (;). A diferencia de otros lenguajes, Java permite declarar las variables en cualquier parte de un bloque de instrucciones (no tiene por qué ser al principio).

CONSTANTES

El concepto de constante no puede entenderse estrictamente en el paradigma de orientación a objetos y por ello en Java no existen las constantes propiamente dichas. Esto se explicará con más detalle cuando pasemos al capítulo de orientación a objetos, por el momento diremos que lo más parecido que tenemos a una constante es una variable de clase no modificable que se declara de la siguiente manera:

```
static final tipo indentificador = valor;
```

Por ejemplo:

```
static final float pi = 3.141592F;
```


ASIGNACIONES

El operador de asignación, como ya hemos visto en las inicializaciones de variables, es el "=", por ejemplo:

```
bool javaEsGenial;  
javaEsGenial=true;  
int edad;  
edad=22;
```

Es posible hacer asignaciones entre tipos numéricos diferentes. Esto es posible de una manera implícita en el sentido de las flechas (no puede haber pérdida de información):

```
byte -> short -> int -> long -> float -> double
```

También puede hacerse en forma explícita cuando tiene que ser al revés mediante un casting, siendo responsabilidad del programador la posible pérdida de información ya que Java no producirá ningún error. Por ejemplo:

```
float radio;  
radio = 5; // no es necesaria la conversión explícita de int a float  
int perimetro;  
perimetro = radio * 2 * PI; // error! no se puede guardar en un int  
                        // el resultado de una operación con float  
perimetro = (int) (radio * 2 * PI) // Ahora es correcto porque el  
                        // casting fuerza la conversión
```

El casting de carácter a un tipo numérico también es posible, aunque no demasiado recomendable. En cambio no es posible la conversión entre booleanos y tipos numéricos.

STRINGS

Ante todo dejar claro que los strings no son tipos primitivos en Java. Son una clase implementada en la biblioteca estándar aunque con ciertas funcionalidades que hacen que su uso sea comparable al de los tipos primitivos en ciertos aspectos. Podemos declarar e inicializar strings como:

```
String st = ""; // String vacío  
String st1 = "Hola";  
String st2 = "cómo estás?";  
String st3 = st1 + " " + st2; // st3 vale "Hola, cómo estás?"
```

La cadena st3 contiene una concatenación de las cadenas st1, un espacio en blanco y st2. Para concatenar Strings basta con operarlos mediante el símbolo "+". Cuando concatenamos un String con algo que no lo es, este valor es automáticamente convertido a String.

```
String st = "numero: " + 3; // st vale "numero: 3"
```

Además de la concatenación, también podemos usar otras operaciones de la clase String para trabajar con ellos. Algunos ejemplos pueden ser (veremos la forma general de llamar operaciones que actúan sobre objetos en el próximo capítulo):

```
String st = "String de ejemplo";  
String st2 = "String de ejemplo";
```

- ✓ **st.length()** Devuelve la longitud de la cadena st.
- ✓ **st.substring(a, b)** Devuelve la subcadena a partir de la posición a (incluida) hasta la b (no incluida).
- ✓ **st.charAt(n)** Devuelve el carácter en la posición n.
- ✓ **st.equals(st2)** Devuelve un booleano que evalúa a cierto si los dos String st y st2 son iguales, en este caso valdrá true.

Cabe destacar que cuando consideramos posiciones dentro de un String, si la longitud es n, las posiciones válidas van de la 0 a la n-1. Otro aspecto a tener en cuenta es que una variable de tipo String es un puntero a una zona de memoria (ya veremos que ocurre en general con todos los objetos). Así pues, (si no hay ciertas optimizaciones por parte del compilador) la comparación `st == st2` evalúa a falso.

OPERADORES

Java define operadores aritméticos, relacionales, lógicos de manipulación de bits, de conversión de tipo, de clase, de selección y de asignación. No debe preocuparse si ahora no queda claro el significado de alguno de ellos

Tipo de operador	Operador	Descripción	Ejemplo
Aritmético	+	Suma	a+b
	-	Resta	a-b
	*	Multipliación	a*b
	/	División	a/b
	%	Módulo	a%b
Relacional	>	Mayor que	a>b
	<	Menor que	a=	Mayor o igual que	a>=b
	<=	Menor o igual que	a<=b
	!=	Diferente	a!=b
	==	Igual	a==b
Lógico	!	No	!a
	&&	Y	a&&b
		O	a o
Asignación	~	Complemento	~a
	&	Y bit a bit	a&b
		O bit a bit	a b
	=	Asignación	a=b
	++	Incremento y asignación	a++
	--	Decremento y asignación	a--
	+=	Suma y asignación	a+=b
	-=	Resta y asignación	a-=b
	=	Multipliación y asignación	a=b
	/=	División y asignación	a/=b
	%=	Módulo y asignación	a%=b
	=	O y asignación	a =b
	&=	Y y asignación	a&=b
	^=	O exclusiva y asignación	a^=b
	<<=	Desplazamiento a la izda. y asignación	a<<=b
	>>=	Desplazamiento a la dcha. y asignación	a>>=b
	>>>=	Desplazamiento a la dcha. rellenando ceros y asignación	a>>>=b
Conversión de tipo (casting)	(tipo)	Convertir tipo	(char)b
Instancia	instanceof	¿Es instancia de clase)	a instanceof b

ESTRUCTURAS DE CONTROL DE FLUJO

Antes de ingresar a fondo con las estructuras de control de flujo debemos tener claro qué es un bloque. Los bloques consisten en secuencias de declaraciones e instrucciones de variables locales. Se escribe como sigue:

```
{
    bloqueCuerpo
}
```

Donde `bloqueCuerpo` es una secuencia de declaraciones e instrucciones de variables locales. Un bloque también puede consistir en una instrucción única sin tener que estar entre llaves. La sentencia vacía consiste en un solo punto y coma (;) y no realiza proceso alguno.

Condicionales

Permiten desviar el flujo de ejecución por una rama u otra dependiendo de la evaluación de una condición. Existen dos estructuras condicionales, el *if* :

```
if (condición1)
    {bloque1}
[else if (condición2)
    {bloque2}
...]
[else
    {bloqueN}]
```

Que puede tener una o más ramas que evalúen condiciones y opcionalmente un *else* cuyo bloque se ejecutará sólo si no se ha cumplido ninguna de las condiciones anteriores. Toda condición (en estas estructura y en adelante) debe evaluarse a un valor booleano no pudiéndose utilizar valores enteros como en C/C++. La otra estructura condicional es el *switch* que permite elegir un bloque en función del valor de una variable de referencia:

```
switch (variableReferencia) {
    case valor1:
        {bloque1}
    [case valor2:
        {bloque2}
    ...]
    [default:
        {bloqueN}
    ]
}
```

variableReferencia: sólo puede ser una expresión que evalúe a los tipos primitivos enteros o `char`. Se evalúa la expresión y sucesivamente se va comprobando que coincida con alguno de los valores, en caso de que así sea se ejecuta el bloque correspondiente. Hay que tener en cuenta que la última instrucción de cada bloque debe ser un `break` porque en caso contrario el flujo del programa seguiría por la primera instrucción del siguiente bloque y así sucesivamente (esto puede ser útil en alguna circunstancia pero no es deseable generalmente). Si `variableReferencia` no coincide con ninguna de los valores del `case`, se ejecutará, caso de existir, el bloque correspondiente al `default`; en caso contrario, simplemente se seguiría con la próxima instrucción después del `switch`.

BUCLES

Los bucles son estructuras iterativas que ejecutan un cierto bucle mientras se da una cierta condición. Java dispone de tres tipos de bucles. Tenemos en primer lugar el *while*:

```
while (condicion){  
  
    bloque  
  
}
```

Que ejecutará el código del bloque mientras condición se evalúe a cierto. La sentencia *do ..while*:

```
do{  
  
    bloque  
  
}while (condicion)
```

Es muy similar sólo que garantiza que el bloque se ejecutará al menos una vez, ya que la evaluación de la condición se produce después de la ejecución del bloque. Finalmente tenemos el clásico *for*:

```
for(inicializacion;condicion;incremento)  
{  
    bloque  
}
```

Que de forma análoga permite ejecutar un bloque mientras se da una cierta condición. El *for* tiene de particular que la inicialización (generalmente un contador) y el incremento queda encapsulado en la misma estructura. Es útil para recorrer estructuras de datos secuenciales. La palabra reservada *break* en cualquiera de los tres bucles fuerza la salida del bucle pasándose a ejecutar la siguiente instrucción después del mismo. La sentencia *continue* fuerza el abandono de la iteración actual haciendo que se ejecute la siguiente; también para todos los tipos de bucle.

EL MÉTODO main

El método *main* es la primera operación que se ejecuta en un programa Java, el que se encarga de poner todo en marcha, y sólo puede haber uno. Su declaración es como sigue: `public static void main(String[]args)` y siempre debe ser exactamente así, ya que si modificamos (u olvidamos) alguno de los modificadores Java no lo interpreta como el método *main* y cuando intentáramos ejecutar la aplicación obtendríamos un error que precisamente nos diría esto más o menos: "no puedo ejecutar el programa porque no existe un método *main*". Como puede observar, el método *main* siempre recibe un parámetro que es un array de *String*.

Este array contiene los parámetros que opcionalmente le hemos podido pasar por la línea de comandos. ¿Qué significa esto? Habitualmente cuando ejecutamos un programa lo hacemos de la siguiente manera:

```
> java HolaUNI [Enter]
```

Pero también habríamos podido ejecutarlo pasándole al programa información adicional:

```
> java HolaUNI p1 p2 p3 [Enter]
```

En este caso hemos llamado a nuestra aplicación y le hemos pasado tres parámetros, p1, p2 y p3, que se almacenan precisamente en el parámetro args del método main. Para acceder a ellos nada más fácil que hacer:

```
public static void main(String[]args) {  
    ...  
    String s = args[0];  
    ...  
}
```

Tener en cuenta que args es un array de String's por lo que aunque nosotros pasemos desde la línea de parámetros números, estos son interpretados como String y si queremos el int, por ejemplo, correspondiente a esa representación tendremos que aplicar el método de conversión adecuado.

El número de parámetros que se han pasado por la línea de comandos puede consultarse mirando la longitud del array:

```
int numParam = args.length;
```


TRABAJANDO CON ARREGLOS

Se estudiara el uso de arreglos y cadenas en la solución de problemas en Java.

DEFINICIÓN

Un arreglo es una colección de variables del mismo tipo. La longitud de un arreglo es fija cuando se crea.

Elemento 0	[0]	Valor = 1	1
Elemento 1	[1]	Valor = 2	2
Elemento 2	[2]	Valor = 4	4
Elemento 3	[3]	Valor = 8	8

Pasos de la creación de arreglos de primitivas

Se declara el arreglo Inicialmente la variable referencia un valor nulo <code>int[] potencias; //forma más usada</code> <code>int potencias[];</code>	<code>potencias-> null</code>
Se crea el arreglo Se requiere la longitud del arreglo <code>potencias = new int[4];</code> En caso de variables primitivas se inician en 0 o false. Las primitivas no se pueden operar con el valor null. En caso de variables referencia se inician en null. No referencian a ningún objeto.	<code>potencias-></code> 0 0 0 0
Se inicia los valores del arreglo. Se asignan valores elemento por elemento <code>potencias[0] = 1;</code> <code>potencias[1] = 2;</code> <code>potencias[2] = 4;</code> <code>potencias[3] = 8;</code> Los arreglos pueden ser creados e iniciados al mismo tiempo <code>int[] potencias = {1,2,4,8};</code>	<code>potencias-></code> 1 2 4 8

Los arreglos son muy usados para buscar datos, especialmente si se conocen sus valores cuando se crean.

```
int[] diasMesesAnioBisiesto = {31,29,31,30,31,30,31,31,30,31,30,31};
```

Arreglos multidimensionales

Se trata de arreglos de arreglos y se declara de la siguiente forma:

```
tipo[][] nombreArreglo = new tipo[cantidadFilas][cantidadColumnas];
```

Ejemplo:

```
int[][] tabla = new int[4][2];
tabla[0][0] = 1;
tabla[0][1] = 7;
tabla[1][0] = 3;
tabla[1][1] = 5;
tabla[2][0] = 4;
tabla[2][1] = 8;
```

Tabla	>	[0]	tabla[0]	>	1	7
		[1]	tabla[1]	>	3	5
		[2]	tabla[2]	>	4	8
		[3]	tabla[3]	>	0	0

Definir una matriz de enteros colocando como valores la suma de su número de fila y columna en la matriz

```
{
    int n= 10;
    int p= 20;
    double[][] m = new double[n][p];
    for (int i= 0; i<n; i++)
        for (int j= 0; j<p; j++)
            m[i][j]= i+j;
}
```

CADENAS

Una cadena es una secuencia de caracteres. La librería String (o clase String) se usa para definir todas las cadenas en Java. Las cadenas se delimitan con comillas dobles.

```
System.out.println("Hola Mundo.");
String camara = "Camara";
String luces = camara + " Accion";
String vacio = "";
```

Construcción de cadenas.

También se puede usar la siguiente sintaxis para construir cadenas.

```
// Con una constante
String nombreDocente = new String("Instructor Java");

// Con una cadena vacía
String inicio = new String();
// Copiando una cadena
String copiaEmpleado = new String(nombreEmpleado);

// Con un arreglo de char
char[] vocales = {'a','e','i','o','u'};
String cadenaVocales = new String(vocales);
```

Concatenación

Para concatenar cadenas puede usar lo siguiente:

```
// Usando el operador +
System.out.println(" Nombre = " + nombreDocente );

// Puede concatenar primitivas y cadenas.
int edad = 22;
System.out.println(" Edad = " + edad );

// Mediante la función concat()
String nombre = "Alberto ";
String apellidos = "Abad Rosales";
String nombreCompleto = nombre.concat(apellidos);
```

Operaciones con cadenas

✓ Longitud de una cadena

```
String blog = ""; int longitud =
blog.length();
```

✓ Ubicar un carácter mediante un índice

```
String nombre = "Instructor Java";
char c = nombre.charAt(0);
```

✓ **Extraer una subcadena**

```
//          012345678901234567890123
String nombre = "Instructor
Java";
String subCadena =
nombre.substring(8,12);
System.out.println(subCadena);
```

✓ **Convertir a mayúsculas o minúsculas**

```
String titulo = ""; String mayusculas =
titulo.toUpperCase(); String minusculas =
titulo.toLowerCase();
```

✓ **Eliminar espacios del inicio y el final de la cadena**

```
String titulo = "  Desarrolla Software  ";
String resaltar = "*" + titulo.trim()+"*";
```

✓ **Ubicar una subcadena desde una ubicación**

```
//          01234567890123456789012345
String blog = ""; int posicion1 =
blog.indexOf("soft");
int posicion2 = blog.indexOf("r",7);
int posicion3 = blog.indexOf("t");
```

✓ **Comparando dos cadenas**

```
String password = "secreto";
if password.equals("Secreto")
    System.out.println("Correcto!");
else
    System.out.println("Error!");

String password = "secreto";
if password.equalsIgnoreCase("Secreto")
    System.out.println("Correcto!");
else
    System.out.println("Error!");
```

✓ **Comparando regiones de una cadena**

```
String url = "http://"; if (url.endsWith(".pe"))
    System.out.println("Pagina Nacional");
else
    System.out.println("Pagina Extranjera");

String parametro = "ip=192.100.51.2";
if (parametro.startsWith("ip"))
    System.out.println("La direccion " + parametro);
else
    System.out.println("El parámetro no es una ip");
```

✓ **Obtener cadenas desde las primitivas**

Se utilizan funciones de la librería String.

```
String seven = String.valueOf(7);
String unoPuntoUno = String.valueOf(1.1);
float pi = 3.141592;
String piString = String.valueOf(pi);
```

✓ **Obtener primitivas desde las cadenas**

En este ejemplo se utilizan métodos de las clases Integer y Float.

```
String alfa = "1977";
int alfaInteger = Integer.parseInt(alfa);

String beta = "19.77";
float betaFloat = Float.parseFloat(beta);
```

Arreglos de cadenas

Un arreglo de cadenas también sigue los pasos de creación de arreglos.

Declaración	String [] categorías;
Creación	categorías = new String[3];
Iniciación.	categorías[0] = "Drama";

A continuación, se tiene un ejemplo:

```
// Creando una arreglo de cadenas vacías.  
String [] arreglo = new String [4];  
for ( int i = 0; i < arreglo.length; i++ ) {  
    arreglo[i] = new String();  
}  
  
// Creando e iniciando un arreglo.  
String [] categorias = {"Drama", "Comedia", "Acción"};  
  
// Accesando los elementos del arreglo  
String [] categorias = {"Drama", "Comedia", "Accion"};  
System.out.println("Comedia = " + categorias[1].length() );
```



PROGRAMACIÓN ORIENTADA A OBJETOS

UN NUEVO PARADIGMA

- ✓ OO es un nuevo paradigma de diseño y programación.
- ✓ OO se basa en modelar objetos del mundo real.
- ✓ OO crea componentes de software reusables.
- ✓ Los objetos contienen en sí mismo información (atributos y datos) y comportamiento (métodos).

OBJETO

Definición

- ✓ **Definición filosófica:** Es una entidad que se puede reconocer.
- ✓ **Para la tecnología de objetos:** Es una abstracción de un objeto del mundo real.
- ✓ **En términos de negocios:** Es una entidad relevante al dominio del negocio.
- ✓ **En términos de software:** Es una estructura que asocia datos y funciones.

Algunos ejemplos de objetos son: Cliente, Factura, Contrato, Película.

Un Cliente tiene atributos, por ejemplo: nombre, dirección, crédito. Un Cliente tiene comportamiento, por ejemplo: alquilar una película, pagar una factura, devolver una película.

LOS OBJETOS REALIZAN OPERACIONES

Un objeto es útil si tiene algún método o comportamiento.

Cada comportamiento se denomina operación.

Objeto: **Mi lapicero azul**

Objeto: **El cajero automático**

Operación: **Escribir**

Operación: **Entregar dinero**

LOS OBJETOS TIENEN VALORES.

Los objetos conocen cuál es su estado actual. Cada conocimiento del objeto se denomina atributo.

Objeto: **Mi lapicero azul**

Objeto: **El cajero automático**

Operación: **Cantidad de tinta**

Operación: **Disponible en soles**

LOS OBJETOS SON UNA ABSTRACCIÓN

Todo depende del contexto. Cuando se modela un objeto, solo se requiere modelar las operaciones y atributos que son relevantes para el problema

Objeto: **Mi lapicero azul**

Operación: **Apuntar una pizarra**

Atributos: **Longitud, Marca**

ENCAPSULAMIENTO

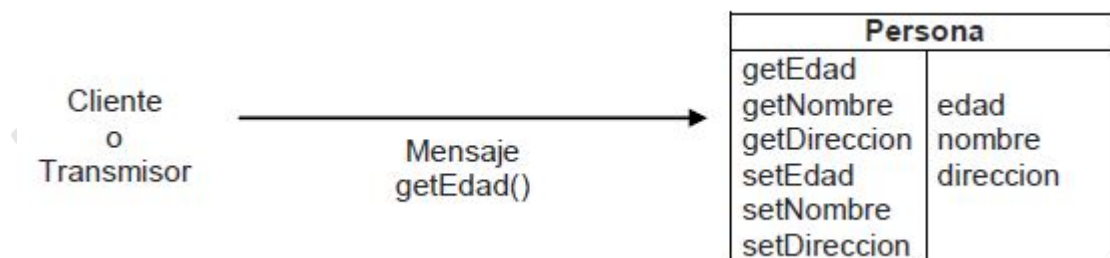
El encapsulamiento oculta como las cosas funcionan dentro de un objeto. Solo nos comunicamos con el objeto a través sus métodos. Los métodos son una interfaz que permite ignorar como están implementados. No es posible evadir el encapsulamiento en OO

El cajero automático es un objeto que entrega dinero.

El cajero encapsula todas las operaciones a los tarjetahabientes.

RELACION ENTRE OBJETOS

Los objetos se comunican unos con otros enviando mensajes. El trasmisor del mensaje pide que el receptor realice una operación. El receptor ejecuta el método correspondiente. En programación estructurada se invocan funciones o procedimientos. En OO se envía un mensaje a un objeto antes que invocarse un procedimiento. Los métodos presentan polimorfismo (varias formas)



ASOCIACIÓN ENTRE OBJETOS

Para que un objeto envíe un mensaje a otro, el receptor debe ser visible para el transmisor. Esta visibilidad se da a través de enlaces (métodos públicos). Un objeto envía mensajes a otro invocando sus métodos.



COMPOSICIÓN DE OBJETOS

Los objetos están compuestos de otros objetos. Los objetos son partes de otros objetos. Esta relación entre objetos se conoce como Agregación o Composición, según sea el caso.

El Banco de la Nación es un objeto	El cajero automático es un objeto. El cajero es del Banco de la Nación.	El cajero automático esta compuesto por objetos como: El teclado El dispensador de billetes El lector de la tarjeta
------------------------------------	--	--

CLASES





Una clase es un molde para crear objetos. Para definir una clase se tiene que indicar las operaciones y atributos. Los objetos son instancias de la clase. Cuando se crea un cajero automático no se requiere indicar sus operaciones y atributos.

Solo se requiere indicar a que clase pertenece.

Clase	Cliente	Película
Atributos	int edad	String titulo
Métodos	String nombre String dirección cambiarDireccion()	double precio String categoria cambiarPrecio()

HERENCIA

Entre diferentes clases puede haber similitudes. La herencia es una relación entre clases donde una es padre de otra. Las propiedades comunes definen la superclase. Clase padre. Las subclases heredan estas propiedades. Clase hija. Un objeto de una clase hija es un-tipo-de una superclase. Un Helicóptero es un tipo de Nave Aérea.

Nave Aérea (padre)			
			
Jumbo	Helicóptero	Globo	Dirigible

POLIMORFISMO

Significa que la misma operación se realiza en diferentes tipos de objetos de diferente forma. Estas operaciones tienen el mismo significado, comportamiento. Internamente cada operación se realiza de diferente forma.

Abordar pasajeros		
 <p>Barco</p>	 <p>Tren</p>	 <p>Helicóptero</p>

MIEMBROS DE CLASE

OPERACIONES CON CLASES

Definición de clase

Una clase es un molde para crear múltiples objetos que encapsulan datos y comportamiento.

PAQUETES

Un paquete es un contenedor (agrupador) de clases que están relacionadas lógicamente. Un paquete tiene sus clases en un mismo directorio. Varias clases pueden tener el mismo nombre pero en diferente paquete.

MODIFICADORES DE ACCESO

Java controla el acceso a las variables y métodos a través de modificadores de acceso como: private, public y protected. Un elemento público puede invocarse en cualquier clase. Un elemento sin modificador solo se puede invocar desde el mismo paquete. Un elemento protegido solo se invoca en clases heredadas. Un elemento privado no puede ser invocado por otra clase.

CREACIÓN DE OBJETOS

Cada objeto es una instancia de alguna clase.

Pelicula
private String titulo;
private String tipo;
Public void mostrarDetalles()
public void obtenerTitulo()



Los objetos se crean con el operador new.

```
Pelicula pelicula1 = new Pelicula();
Pelicula pelicula2 = new Pelicula();
```

El operador new realiza las siguientes acciones:

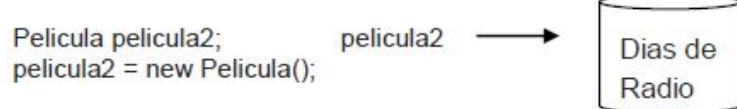
- ✓ Separa memoria para el nuevo objeto
- ✓ Invoca el método de inicio de la clase llamado constructor.
- ✓ Retorna la referencia a un nuevo objeto.



Las variables primitivas almacenan valores.



Los objetos almacenan referencias



LA REFERENCIA NULL

<ul style="list-style-type: none"> Los objetos inician en null Se puede comparar un objeto con null Se puede liberar la referencia con null. 	<pre>Pelicula pelicula1 = null; if (pelicula1 == null) pelicula1 = new Pelicula(); pelicula1 = null;</pre>
---	--

ASIGNANDO REFERENCIAS

Se puede asignar una variable referencia a otra, el resultado es dos referencias al mismo objeto.

```
Pelicula pelicula1 = new Pelicula("Betty Blue");

Pelicula pelicula2 = pelicula1;
```

Las variables de instancia se declaran en la clase. Estos son atributos de la clase.

```
public class Pelicula {

    public String titulo;
    public String tipo;

}
```

Las variables públicas de instancia se acceden con el operador punto.

```
Pelicula pelicula1 = new Pelicula();
pelicula1.titulo = "Kramer vs Kramer";
if (pelicula1.titulo.equals("El planeta de los simios"))
    pelicula1.tipo = "Ciencia Ficción";
```

MÉTODOS

Definición

Los métodos definen el comportamiento de los objetos.

Los métodos solo se definen dentro de una clase.

```
modificador tipoDeRetorno nombreMetodo (ListaDeArgumentos) {  
  
    // desarrollo del método;  
  
}
```

Argumentos

En la definición del método se indica el tipo y el nombre de los argumentos del método.

```
public void setTipo (String nuevoTipo) {  
    tipo = nuevoTipo;  
}
```

Si el método tiene varios argumentos, estos se separan por comas.

```
public void setDatos (String nuevoTitulo, String nuevoTipo) {  
    tipo = nuevoTipo;  
    titulo = nuevoTitulo;  
}
```

Si el método no tiene argumentos, se deja solo los paréntesis

```
public void mostrarDetalles() {  
    System.out.println("El titulo es " + titulo);  
    System.out.println("El tipo es " + tipo);  
}
```

Se usa la sentencia return para salir del método retornando un valor. No se requiere return si el tipo de retorno es void.

```
public class pelicula {  
    private String tipo;  
    //...  
    public String obtenerTipo () {  
        return tipo;  
    }  
}
```

Invocando métodos

Se utiliza el operador punto para invocar el método de una clase, si el método es de la misma clase, no se requiere el calificador de la clase.

Pelicula.java

```
public class Pelicula {
    private String titulo, tipo;
    //...
    public String getTipo () {
        return tipo;
    }
    public void setTipo (String nuevoTipo) {
        tipo = nuevoTipo;
    }
}
```

PruebaPeliculas.java

```
public class PruebaPeliculas {
    public static void main (String[] args) {
        Pelicula pelicula1 = new Pelicula();
        pelicula1.setTipo("Comedia");
        if (pelicula1.getTipo().equals("Drama")) {
            System.out.println("La película es un drama");
        } else {
            System.out.println("La película no es un drama");
        }
    }
}
```

ENCAPSULAMIENTO

Las variables de instancia de una clase deberían ser declaradas como privadas. Solo un método debería acceder a las variables privadas. No se debe acceder a las variables de instancia directamente, sino a través de un método.

```
Pelicula pelicula1 = new Pelicula();
if( pelicula1.getTitulo().equals("Los doce del patibulo") ) {
    pelicula1.setTipo("Acción");
}
```

CÓDIGO DE UNA CLASE.**Pelicula.java**

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Pelicula {

    private String titulo;
    private String tipo;

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

}
```

Paso de variables a métodos

Cuando el argumento es de tipo primitivo, se genera una copia de la variable para el método.

TestPrimitivas.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestPrimitivas {

    public static void main(String[] args) {
        int numero = 150;
        unMetodo(numero);
        System.out.println(numero);
    }

    private static void unMetodo(int argumento) {
        if (argumento < 0 || argumento > 100) {
            argumento = 0;
            System.out.println(argumento);
        }
    }
}
```

Aquí tenemos su ejecución:

```
run:
0
150
```

Cuando se pasa como argumento un objeto referencia, no se genera copia. Se pasa la referencia del objeto original.

TestReferencias.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestReferencias {

    public static void main(String[] args) {
        Pelicula pelicula1 = new Pelicula();
        pelicula1.setTitulo("El Resplandor");
        pelicula1.setTipo("Drama");
        System.out.println(pelicula1.getTipo());
        unMetodo(pelicula1);
        System.out.println(pelicula1.getTipo());
    }

    public static void unMetodo(Pelicula referencia) {
        referencia.setTipo("Terror");
    }

}
```

A continuación, tenemos su ejecución:

```
run:
Drama
Terror
```

SOBRE CARGA DE METODOS

SOBRECARGA DE MÉTODOS

Algunos métodos en una clase pueden tener el mismo nombre. Estos métodos deben contar con diferentes argumentos. El compilador decide que método invocar comparando los argumentos. Se generara un error si los métodos solo varían en el tipo de retorno.

```
public class Pelicula {

    private float precio;

    public void setPrecio() {
        precio = 3.50;
    }

    public void setPrecio(float nuevoPrecio) {
        precio = nuevoPrecio;
    }

}
```

INICIACIÓN DE VARIABLES DE INSTANCIA

Las variables de instancia se pueden iniciar en la declaración. Esta iniciación ocurre cuando se crea un objeto.

```
public class Pelicula {
    private String titulo;           // implicito
    private String tipo = "Drama";  // explicito
    private int numeroDeOscars;     // implicito
}
```

Las variables de tipos primitivos se inician implícitamente como se ilustra a continuación:

Tipo	Valor
char	'0' 0
byte, short, int, long	false
boolean	0.0
float, double	null
Referencia a Objeto	

En forma explícita se puede indicar un valor inicial.

```
private String tipo = "Drama";
```

Un constructor provee una iniciación de variables de instancia más compleja.

CONSTRUCTORES

Para una adecuada iniciación de variables de instancia, la clase debe tener un constructor. Un constructor se invoca automáticamente cuando se crea un objeto. Se declaran de forma pública. Tiene el mismo nombre que la clase. No retorna ningún valor. Si no se codifica un constructor, el compilador crea uno por defecto sin argumentos que solo inicia las variables de instancia.

Pelicula.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Pelicula {

    private String titulo;
    private String tipo = "Drama";

    public Pelicula() {
        titulo = "Pelicula sin definir.";
    }

    public Pelicula(String titulo) {
        this.titulo = titulo;
    }

    public Pelicula(String titulo, String tipo) {
        this.titulo = titulo;
        this.tipo = tipo;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
}
```

TestConstructores.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestConstructores {

    public static void main(String[] args) {
        Pelicula pelicula1 = new Pelicula();
        Pelicula pelicula2 = new Pelicula("La lista de Schindler.");
        Pelicula pelicula3 = new Pelicula("El dormilon.", "Comedia");
        System.out.println(pelicula1.getTitulo() + pelicula1.getTipo());
        System.out.println(pelicula2.getTitulo() + pelicula2.getTipo());
        System.out.println(pelicula3.getTitulo() + pelicula3.getTipo());
    }
}
```

A continuación, tenemos el resultado:

```
run:
Pelicula sin definir.Drama
La lista de Schindler.Drama
El dormilon.Comedia
```

La referencia: this

Los métodos de instancia reciben el argumento `this` implícitamente que se refiere al mismo objeto.

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Pelicula {

    private String titulo;

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

}
```

Se puede compartir código entre constructores usando la referencia this. Un constructor invoca a otro pasándole los argumentos que requiere.

Pelicula.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Pelicula {

    private String titulo;

    public Pelicula(){
        this("Pelicula sin definir");
    }

    public Pelicula (String titulo) {
        this.titulo = titulo;
    }

}
```

TestThis.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestThis {

    public static void main(String[] args) {
        Pelicula pelicula1 = new Pelicula();
        Pelicula pelicula2 = new Pelicula("Todo sobre mi madre");
        System.out.println(pelicula1.getTitulo());
        System.out.println(pelicula2.getTitulo());
    }

}
```

A continuación, tenemos el resultado de su ejecución:

```
run:
Pelicula sin definir.
Todo sobre mi madre
```

VARIABLES DE CLASE

Las variables de clase comparten un único valor entre todas las instancias de la clase. Se declaran con el calificador static.

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public Class Pelicula {

    // Iniciación por defecto
    private static double precioMinimo;
    private String titulo, tipo;

}
```

Las variables de clase se pueden iniciar en la declaración. La iniciación ocurre cuando se carga la clase en memoria. Para una iniciación compleja se usara un bloque static

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public Class Pelicula {

    // Iniciación explícita
    private static double precioMinimo = 3.29;

}
```

MÉTODOS DE CLASE

Estos métodos son compartidos por todas las instancias. Se usan estos métodos principalmente para manipular variables de clase. Se les declara con el calificador static. Se invoca a este método de clase con el nombre de la clase o con el nombre de una instancia.

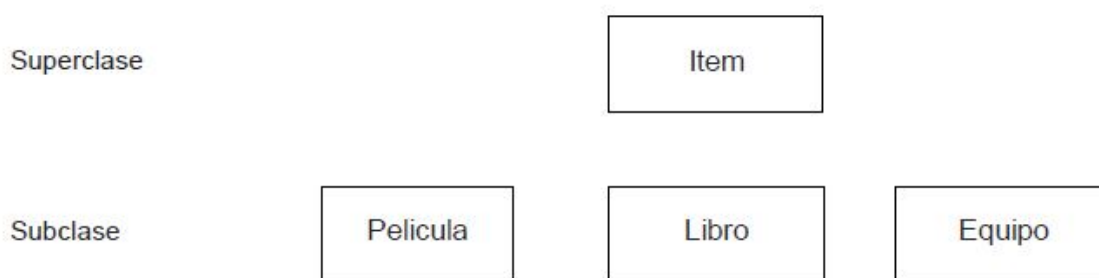
HERENCIA

HERENCIA Y POLIMORFISMO

Se estudia el uso de la herencia y el polimorfismo en el reúso de clases.

Herencia

Permite a una clase compartir la misma estructura de datos y comportamiento de otra clase. La herencia minimiza la necesidad de duplicar código. El Polimorfismo permite utilizar el método de acuerdo al objeto heredado.



¿Qué atributos tienen en común las subclases?

¿Qué atributos no tienen en común las subclases?

¿Qué método no tienen en común las subclases?

LA HERENCIA EN JAVA

Una subclase se define indicando a que superclase extiende.

```

public class Item {
    // Definición de la superclase Item.
}

public class Pelicula extends Item {
    // Atributos y métodos adicionales para distinguir una
    // pelicula de otros tipos de item
}
  
```

Una subclase hereda todas las variables instancia de la superclase. Las variables de instancia deben ser private para que instancias de la subclase hereden sus valores.

```
public class Item {
    protected float precio = 0;
    protected String estado = "Excelente";
}

public class Pelicula extends Item {
    private String titulo = "";
    private int duracion = 0;
}
```

Una subclase no hereda ningún constructor de la superclase, debe declararse explícitamente. Solo en caso no se declare explícitamente, se ejecutarán los constructores por defecto de las superclases y finalmente de la subclase.

```
Pelicula pelicula = new Pelicula ();
// Inicia variables de la clase Item. Constructor por defecto.
// Inicia variables de la clase Pelicula. Constructor por defecto.
```

LA REFERENCIA SUPER

Se refiere a la clase padre. Se usa para invocar constructores de la clase padre. Debe ser la primera sentencia del constructor de la clase hijo. Esta referencia también se usa para invocar cualquier método del padre.

```
public class Item {
    protected float precio = 0;

    Item (float precio) {
        this.precio = precio;
    }
}

public class Pelicula extends Item {
    private String titulo = "";

    Pelicula (float precio, String titulo) {
        super(precio);
        this.titulo = titulo;
    }
}
```

MÉTODOS

La superclase define los métodos para todas las subclases. La subclase puede especificar métodos propios.

Item0.java

```
public class Item0 {  
  
    protected float precio = 0;  
  
    Item0 (float precio) {  
        this.precio = precio;  
    }  
  
    public float getPrecio() {  
        return precio;  
    }  
  
}
```

Pelicula0.java

```
public class Pelicula0 extends Item0 {  
  
    private String titulo = "";  
  
    Pelicula0 (float precio, String titulo) {  
        super(precio);  
        this.titulo = titulo;  
    }  
  
    public String getTitulo(){  
        return titulo;  
    }  
  
}
```

TestSuper.java

```
public class TestSuper {  
  
    public static void main (String[] args) {  
  
        Item0 item = new Item0(1.1f);  
        System.out.println( item.getPrecio() );  
  
        Pelicula0 pelicula = new Pelicula0(2.2f,"Zelig");  
        System.out.println( pelicula.getPrecio() );  
        System.out.println( pelicula.getTitulo() );  
  
    }  
  
}
```


¿Qué diferencia existe entre this y super?.

¿Se puede reemplazar super(precio); por this.precio = precio;?.

¿Qué métodos puede invocar una subclase?.

La subclase hereda todos los métodos del padre. La subclase puede re-escribir un método del padre.

Item1.java

```
public class Item1 {  
  
    public float calcularImporte(int cliente) {  
        return 50;  
    }  
  
}
```

Pelicula1.java

```
public class Pelicula1 extends Item1 {  
  
    public float calcularImporte(int cliente) {  
        if (cliente < 500)  
            return 10;  
        else  
            return 30;  
    }  
  
}
```

TestSobrescribir.java

```
public class TestSobrescribir {  
  
    public static void main (String[] args) {  
  
        Item1 item1 = new Item1();  
        System.out.println( item1.calcularImporte(599) );  
  
        Pelicula1 pelicula1 = new Pelicula1();  
        System.out.println( pelicula1.calcularImporte(399) );  
        System.out.println( pelicula1.calcularImporte(599) );  
  
    }  
  
}
```

¿Cuál es la diferencia entre sobre-carga de métodos y sobre-escritura de métodos?

LA REFERENCIA SUPER

Si una subclase sobrescribe un método de la superclase; el método de la superclase se puede invocar con la referencia super.

Item2.java

```
public class Item2 {  
  
    public float calcularImporte(int cliente) {  
        return 50;  
    }  
  
}
```

Equipo2.java

```
public class Equipo2 extends Item2 {  
  
    public float calcularImporte(int cliente) {  
        float seguroEquipo = 25;  
        float alquiler = super.calcularImporte(cliente);  
        return seguroEquipo + alquiler;  
    }  
  
}
```

TestSuper2.java

```
public class TestSuper2 {  
  
    public static void main (String[] args) {  
  
        Item2 articulo = new Item2();  
        System.out.println( articulo.calcularImporte(599) );  
  
        Equipo2 vhs = new Equipo2();  
        System.out.println( vhs.calcularImporte(599) );  
  
    }  
  
}
```

POLIMORFISMO

Permite efectuar una misma operación dependiendo del tipo de objeto.

TacoraFilms inicia sus operaciones alquilando únicamente películas. Tres meses después amplía el alquiler a equipos, juegos y libros. El alquiler de una película es 2 soles por día de alquiler. El alquiler de un equipo consta de un seguro de 50 soles además de 5 soles por día. El alquiler de juegos depende del fabricante. PlayStation 2

soles/día, Nintendo 1 sol/día. Los libros no se alquilan, se prestan uno a la vez, mientras sean clientes de la tienda.

Explique por qué se obtienen los resultados

Alquiler3.java

```
public class Alquiler3 {

    private int dias;

    public Alquiler3(int dias) {
        this.dias = dias;
    }

    public int getDias () {
        return dias;
    }

}
```

Item3.java

```
public class Item3 {

    protected float calcularImporte(Alquiler3 contrato) {
        return 0;
    }

}
```

Pelicula3.java

```
public class Pelicula3 extends Item3 {

    protected float calcularImporte(Alquiler3 contrato) {
        int importe = 2*contrato.getDias();
        return importe;
    }

}

public class Equipo3 extends Item3 {

    protected float calcularImporte(Alquiler3 contrato) {
        int seguroEquipo = 50;
        int importe = seguroEquipo + 5*contrato.getDias();
        return seguroEquipo + importe;
    }

}
```

Juego3.java

```
public class Juego3 extends Item3 {

    String fabricante;

    public Juego3(String fabricante) {
        this.fabricante = fabricante;
    }

    public String getFabricante() {
        return fabricante;
    }

    protected float calcularImporte(Alquiler3 contrato) {
        String fabricante = this.fabricante;
        int tasa = 0;
        if (fabricante.equals("PlayStation"))    tasa = 2;
        if (fabricante.equals("Nintendo"))      tasa = 1;
        int importe = tasa*contrato.getDias();
        return importe;
    }

}
```

Libro3.java

```
public class Libro3 extends Item3 {

    protected float calcularImporte(Alquiler3 contrato) {
        return 0;
    }

}
```

TestPolimorfismo.java

```
public class TestPolimorfismo {  
  
    public static void main (String[] args) {  
  
        Alquiler3 contrato = new Alquiler3(10);  
  
        Pelicula3 oscar = new Pelicula3();  
        System.out.println( oscar.calcularImporte(contrato) );  
  
        Equipo3 vhs = new Equipo3();  
        System.out.println( vhs.calcularImporte(contrato) );  
  
        Juego3 mu = new Juego3("Nintendo");  
        System.out.println( mu.calcularImporte(contrato) );  
  
        Libro3 agua = new Libro3();  
        System.out.println( agua.calcularImporte(contrato) );  
  
    }  
}
```

EL OPERADOR INSTANCEOF Y CAST

El operador instanceof permite determinar la clase de un objeto en tiempo de ejecución.
La operación cast permite modificar la clase de un objeto.

```
public class TestOperador {

    public static void main (String[] args) {
        Pelicula3 oscar = new Pelicula3();
        Equipo3   vhs   = new Equipo3();
        Juego3    mu    = new Juego3("Nintendo");
        Libro3    agua  = new Libro3();

        testOperador(oscar);
        testOperador(vhs);
        testOperador(mu);
        testOperador(agua);
    }

    public static void testOperador (Item3 articulo) {
        if (articulo instanceof Juego3) {
            Juego3 juego = (Juego3) articulo;
            System.out.println(juego.getFabricante());
        } else {
            System.out.println("No tiene Fabricante");
        }
    }
}
```

ALCANCE DE LA CLASE

ATRIBUTOS, MÉTODOS Y CLASES FINAL

Variables finales

Una variable final es una constante. Una variable final no puede ser modificada. Una variable final debe ser iniciada. Una variable final por lo general es pública para que pueda ser accesada externamente.

```
public final static String NEGRO = "FFFFFF";
public final static float PI = 3.141592f;
public final static int MAXIMO_ITEMS = 10;
```

Métodos finales

Un método puede ser definida como final para evitar la sobre-escritura en una subclase. Un método final no se puede redefinir en una clase hijo.

```
public final static String getBlanco() {
    return "000000";
}

public final boolean verificarPassword(String password) {
    if (password.equals(...
    }
}
```

Clases finales

Una clase final no puede ser padre de otra clase. Una clase puede ser definida como final para evitar la herencia. El compilador es más eficiente con definiciones final por qué no buscare estas clases o métodos al tratar clases heredadas.

```
public final class Color {

    public final static String NEGRO = "FFFFFF";

    public final static String getBlanco() {
        return "000000";
    }

}
```

EL MÉTODO finalize()

Cuando todas las referencias de un objeto se pierden, se marcan para que el Garbage Collector los recoja y libere ese espacio en memoria.

```
Pelicula pelicula = new Pelicula("Zelig");  
pelicula = null;
```

El objeto "Zelig" que estaba referenciado por pelicula ha perdido todas sus referencias. Luego el Garbage Collector liberará el espacio ocupado por "Zelig". El método finalize es llamado justo antes que el Garbage Collector libere la memoria. En este instante se puede aprovechar para realizar otras operaciones.

```
public class Pelicula4 {  
  
    private String titulo;  
  
    public Pelicula4(String titulo) {  
        this.titulo = titulo;  
    }  
  
    public void finalize() {  
        System.out.println("Se acabo "+titulo);  
    }  
}  
  
public class TestFinalize {  
  
    public static void main (String[] args) {  
        Pelicula4 globo = new Pelicula4("Zelig");  
        globo = null;  
    }  
}
```


CLASES ABSTRACTAS E INTERFACES

Se estudiará el uso de clases abstractas, métodos abstractos, interfaces, implementación de interfaces.

CLASES ABSTRACTAS

Clases abstractas

Sirven para modelar objetos de alto nivel, no contienen código, sino solo declaraciones. Todos sus métodos deben existir en sus clases hijas. Una clase abstracta no puede ser instanciada (Crear un objeto a partir de ella).

Métodos abstractos

Estos métodos son parte de clases abstractas. Un método abstracto debe ser redefinido en las subclases. Cada subclase puede definir el método de manera diferente. Las clases abstractas pueden contener métodos que no son abstractos, estos son métodos concretos.

Item.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public abstract class Item {

    protected String titulo;
    protected float precio = 5.0f;

    // Método abstracto
    public abstract boolean esAlquilable();

    // Método concreto
    public float getPrecio() {
        return precio;
    }
}
```

Pelicula.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Pelicula extends Item{

    // Implementación del método abstracto
    @Override
    public boolean esAlquilable() {
        return true;
    }

}
```

Libro.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Libro extends Item {

    // Implementación de método abstracto
    @Override
    public boolean esAlquilable() {
        return false;
    }

    // Redefinición de método getPrecio.
    @Override
    public float getPrecio() {
        this.precio = 0.0f;
        return precio;
    }

}
```

Abstracto.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Abstracto {

    public static void main(String[] args) {

        Pelicula pelicula = new Pelicula();
        Libro libro = new Libro();

        System.out.println("PELICULA");
        System.out.println("Alquilable:" + pelicula.esAlquilable());
        System.out.println("Precio:" + pelicula.getPrecio());
        System.out.println("LIBRO");
        System.out.println("Alquilable:" + libro.esAlquilable());
        System.out.println("Precio:" + libro.getPrecio());

    }






}
```

A continuación, tenemos el resultado de su ejecución:

```
run: PELICULA
Alquilable:true
Precio:5.0
LIBRO
Alquilable:false
Precio:0.0
```

INTERFACES

Una interface es totalmente abstracta; todos sus métodos son abstractos, sus atributos son públicos estáticos y final. Una interface define los métodos que otras clases pueden implementar pero no provee ninguna línea de código. Una clase solo puede heredar de una superclase. Una clase puede implementar muchas interfaces; por lo que las interfaces permiten herencia múltiple.

Conducible			
No Conducible			

Las interfaces describen la conducta que requiere muchas clases. El nombre de una interface por lo general es un adjetivo como Conducible, Ordenable, Ubicable.

Aquí se diferencia de una clase que usualmente es un sustantivo como Pelicula, Cliente, Alquiler. Las clases implementadas por una interface pueden no tener ninguna relación unas con otras. A diferencia de las clases heredadas de una superclase tiene similitudes. Las clases que implementan una interface deben definir todos los métodos de la interface.

Conducible.java

```
public interface Conducible {

    public static final int MAXIMO_GIRO = 45;

    public abstract void girarIzquierda(int grados);
    public abstract void girarDerecha(int grados);

}
```

También se puede definir la interface sin los calificadores public static final abstract, puesto que son implícitos. Para declarar que una clase implementa una interface se usa implements.

Conducible.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public interface Conducible {

    // Máximo angulo de giro
    int MAXIMO_GIRO = 90;

    // Girar a la izquierda
    void girarIzquierda(int grados);

    // Girar a la derecha
    void girarDerecha(int grados);

}
```

NaveArea.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class NaveAerea {

    protected char direccion;
    protected int altura;

    public void setDireccion(char direccion) {
        this.direccion = direccion;
    }

    public char getDireccion() {
        return this.direccion;
    }

}
```

Globo.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Globo extends NaveAerea implements Conducible {

    private int volumenGas;

    public void setVolumenGas(int volumenGas) {
        this.volumenGas = volumenGas;
    }

    public int getVolumenGas() {
        return this.volumenGas;
    }

    // Implementación de método abstracto de Conducible
    @Override
    public void girarIzquierda(int grados) {
        if (getDireccion() == 'N' && grados == 90) {
            setDireccion('O');
        }
    }

    // Implementación de método abstracto de Conducible
    @Override
    public void girarDerecha(int grados) {
        if (getDireccion() == 'N' && grados == 90) {
            setDireccion('E');
        }
    }
}
```

Patin.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Patin implements Conducible {

    // Implementación de método abstracto de Conducible
    @Override
    public void girarIzquierda(int grados) {
        System.out.println("Giro de " + grados + " grados a la izquierda");
    }

    // Implementación de método abstracto de Conducible
    @Override
    public void girarDerecha(int grados) {
        System.out.println("Giro de " + grados + " grados a la derecha");
    }

}
```

TestInterface.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestInterface {

    public static void main(String[] args) {

        Globo zepelin = new Globo();
        zepelin.setDireccion('N');
        zepelin.girarDerecha(90);
        System.out.println("GLOBO");
        System.out.println("Dirección: " + zepelin.getDireccion());

        Patin patin = new Patin();
        System.out.println("PATIN");
        patin.girarDerecha(45);

    }

}
```

A continuación, tenemos el resultado de su ejecución:

```
run: GLOBO
Dirección: E
PATIN
Giro de 45 grados a la derecha
```

LA CLASE Object

Todas las clases en java heredan de la clase java.lang.Object. Los siguientes métodos se heredan de la clase Object

boolean equals(Object obj)	Indica si el objeto es igual a otro.
int hashCode()	Proporciona un valor hashcode para el objeto (id)
String toString()	Proporciona una representación en cadena del objeto

Otros métodos se encuentran en:

<http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

CONVERTIR DATOS PRIMITIVOS EN REFERENCIAS

Los datos primitivos no se comparan con null. Los datos primitivos no pueden integrar una Colección.

Para cada dato primitivo java provee una clase para convertirlo en referencia y tratarlo como un objeto y resolver los problemas previos.


```

/**
 *
 * @teacher
 * @blog
 * @email
 */
public class PruebaReferencia {

    static int precio;

    public static void main(String[] args) {

        System.out.println("A.- " + precio);
        precio = 2;
        System.out.println("B.- " + (precio + 5));

        Integer objetoPrecio;
        objetoPrecio = new Integer(precio);
        System.out.println("C.- " + objetoPrecio.toString());
        System.out.println("D.- " + objetoPrecio);
        System.out.println("E.- " + (objetoPrecio + 5));
        System.out.println("F.- " + objetoPrecio.intValue());
        System.out.println("G.- " + (objetoPrecio.intValue() + 5));

    }

}

```

A continuación, tenemos el resultado de su ejecución:

```

run:
A.- 0
B.- 7
C.- 2
D.- 2
E.- 7
F.- 2
G.- 7

```

Análisis del resultado:

- ✓ En la fila A se muestra el valor por defecto del variable precio.
- ✓ En la fila B se muestra la suma del valor del variable precio y el valor 2.
- ✓ En la fila C esta mostrando la representación en cadena del objeto objetoPrecio.
- ✓ La fila D es similar a la fila C, por defecto se muestra el resultado del método toString().

- ✓ En la fila E, existe una conversión implícita del objeto objetoPrecio a int para que pueda efectuarse la suma con el valor 2.
- ✓ En la fila F se esta mostrando el valor del objeto objetoPrecio, pero como un valor entero.
- ✓ La fila G es similar a la fila E, solo que en este caso la conversión es explícita.



ARREGLOS Y COLECCIONES

COLECCIONES

Es un conjunto librerías para manipular y almacenar datos. Estas librerías se llaman colecciones.

Las colecciones se organizan en:

- ✓ Interfaces: Manipulan los datos independientemente de los detalles de implementación.
- ✓ Clases: Implementan las interfaces.

Para programar con colecciones se debe:

- ✓ Elegir una interface adecuada a la funcionalidad requerida.
- ✓ Elegir una clase que implemente la interfaz
- ✓ Extender la clase si fuera necesario.

Reto:

¿En qué se diferencia una clase de una interface?

ARQUITECTURA

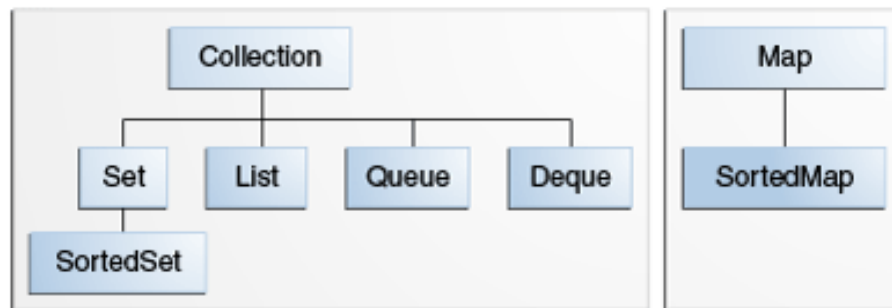
Referencia:

<https://docs.oracle.com/javase/tutorial/collections/intro/index.html>

Las interfaces y las clases están relacionadas en un armazón (framework) de colecciones para facilitar su uso.

- ✓ Interfaces de colecciones que son definiciones abstractas de los tipos de colecciones.
- ✓ Clases que implementan las interfaces.
- ✓ Clases abstractas que implementan parcialmente las interfaces.
- ✓ Métodos estáticos que son algoritmos (por ejemplo, ordenar una lista).
- ✓ Interfaces de soporte para las colecciones. (una infraestructura).

Interfaces de colecciones



Collection Representa un grupo de objetos. Sin implementaciones directas, agrupa la funcionalidad general que todas las colecciones ofrecen.

Set Colección que no puede tener objetos duplicados.

SortedSet Set que mantiene los elementos ordenados

List Colección ordenada que puede tener objetos duplicados

Map Colección que enlaza claves y valores; no puede tener claves duplicadas y cada clave debe tener al menos un valor.

SortedMap Map que mantiene las claves ordenadas.

LA INTERFACE COLLECTION

add(Object) Añade el objeto en la colección

addAll(Collection) Añade la colección.

clear() Quita todos los elementos de la colección.

contains(Object) ¿El objeto se encuentra en la colección?

containsAll(Collection) ¿Todos esos elementos están en la colección?

equals(Object) ¿Es igual esta colección al argumento?

isEmpty() ¿La colección está vacía?

Iterator iterator() Devuelve un iterador para la colección

remove(Object) Elimina una aparición del objeto

removeAll(Collection) Elimina todos esos objetos

retainAll(Collection) Se queda sólo con los objetos del argumento

size()	Número de elementos en la Colección
toArray()	Devuelve un arreglo con los objetos de la colección.

LA INTERFACE LIST

Colecciones ordenadas (secuencias) en las que cada elemento ocupa una posición identificada por un índice. El primer índice es el 0. Las listas admiten duplicados.

add(int, Object)	Añade el objeto en la posición indicada
add(Object)	Añade el objeto al final de la lista
addAll(int, Collection)	Añade la colección en la posición
addAll(Collection)	Añade la colección al final
clear()	Quita todos los elementos de la lista.
contains(Object)	¿El objeto se encuentra en la lista?
containsAll(Collection)	¿Todos esos elementos están en la lista?
equals(Object)	¿Es igual la lista con el argumento?
get(int)	Devuelve el objeto en la posición.
indexOf(Object)	Devuelve la 1ra posición en la que está el objeto
isEmpty()	¿La lista está vacía?
Iterator iterator()	Devuelve un iterador para la colección.
lastIndexOf(Object)	Devuelve la última posición del objeto ListIterator
listIterator()	Devuelve un iterador de lista
ListIterator listIterator(int)	Devuelve un iterador de lista para la sublista que inicia en int
remove(int)	Quita de la lista el objeto en esa posición
remove(Object)	Elimina una aparición del objeto en la lista
removeAll(Collection)	Elimina todos esos objetos
retainAll(Collection)	Se queda sólo con los objetos del argumento
set(int, Object)	Reemplaza el objeto en esa posición por el objeto que se proporciona
size()	Número de elementos en la Colección

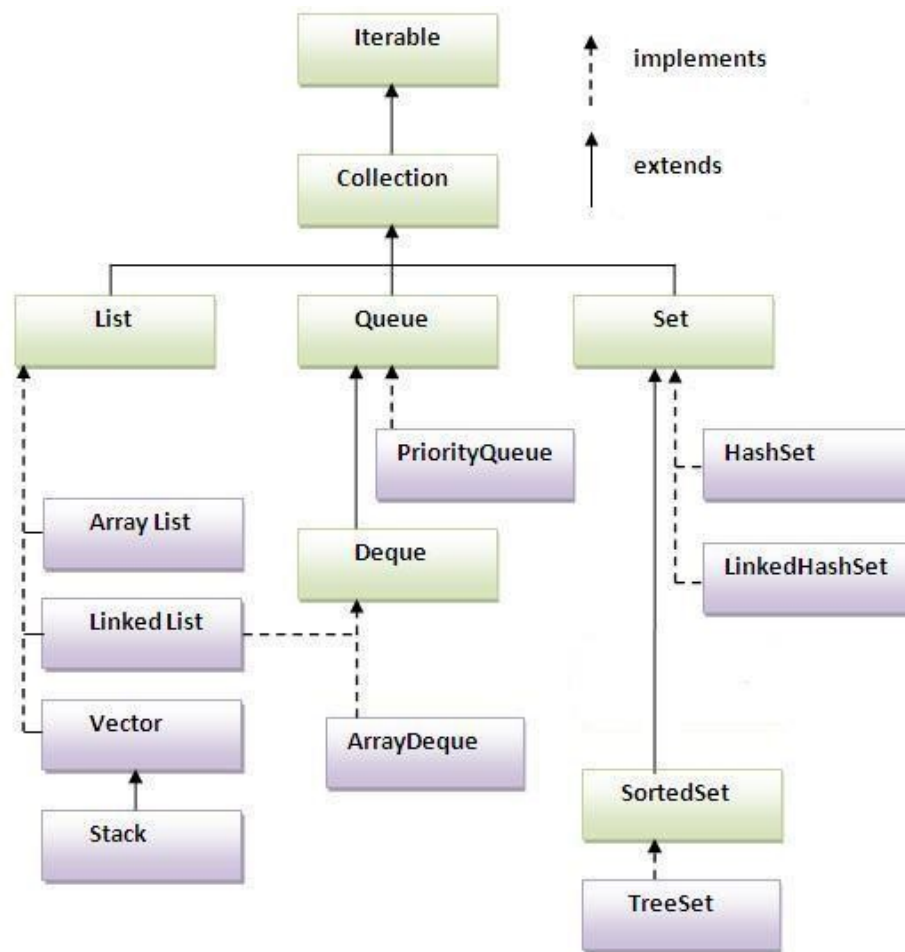
List subList(int, int)	Devuelve la sublista que comienza en el índice del primer argumento hasta el índice del segundo argumento.
toArray()	Devuelve un arreglo con los objetos de la colección.

LA INTERFACE MAP

Son pares de datos (clave, valor). No puede haber claves duplicadas y cada clave se corresponde con al menos un valor.

clear()	Elimina todos los pares del mapa
containsKey(Object)	¿La clave proporcionada se encuentra en el mapa?
containsValue(Object)	¿Hay algún par con el valor proporcionado?
equals(Object)	¿Es igual este mapa y el proporcionado?
get(Object)	Devuelve el objeto que se corresponde con la clave dada.
isEmpty()	¿La lista está vacía?
put(Object clave, Object valor)	Asocia la clave proporcionada con el valor proporcionado
putAll(Map)	Agrega los pares de ese mapa
remove(Object)	Quita la clave del mapa junto con su correspondencia.
size()	Devuelve el número de pares que hay en el mapa.

CLASES IMPLEMENTADAS



Las interfaces **List**, **Set** y **SortedSet** son descendientes de la interface **Collection**

El concepto de Polimorfismo aplica para todas las clases que implementan estas interfaces.

- ✓ Las clases que implementan la interface **List** son: **ArrayList** y **LinkedList**
- ✓ Las clases que implementan la interface **Set** son: **HashSet** y **LinkedHashSet**
- ✓ La clase que implementa la sub-interface **SortedSet** es: **TreeSet**.

Definiendo una clase

Para manipular las colecciones usaremos la clase **Producto** compuesta por dos atributos, un constructor y un método **get**.

Producto.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Producto {

    private String nombre;
    private int cantidad;

    public Producto() {
    }

    public Producto(String nombre, int cantidad) {
        this.nombre = nombre;
        this.cantidad = cantidad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

}
```

MOSTRAR LOS ELEMENTOS DE UNA COLECCIÓN: ARRAYLIST

En el ejemplo crearemos una lista del mercado y mostraremos sus elementos.

- ✓ Primero se importan las librerías de java.util.* donde se concentran la gran mayoría de las Clases del "Collection Framework".
- ✓ Se inicia la declaración de la clase seguido de su método principal main.
- ✓ Se definen 5 instancias con el constructor de la clase Producto.

- ✓ Agregamos estas instancias al ArrayList con el método add
- ✓ Mostramos el número de objetos de la colección mediante el método size.
- ✓ Se declara una instancia Iterator la cual facilita la extracción de objetos de la colección.
- ✓ Se extrae los objetos del ArrayList y se muestran en pantalla.
- ✓ Se elimina el objeto con índice 2. Se muestra la lista nuevamente.
- ✓ Se eliminan todos los objetos mediante el método clear.

```
import java.util.ArrayList;
import java.util.Iterator;

/**
 *
 * @teacher
 * @blog
 * @email
 */
public class MercadoLista {

    public static void main(String args[]) {

        // Se definen 5 instancias de la Clase Producto
        Producto pan = new Producto("Pan", 6);
        Producto leche = new Producto("Leche", 2);
        Producto manzanas = new Producto("Manzanas", 5);
        Producto brocoli = new Producto("Brocoli", 2);
        Producto carne = new Producto("Carne", 2);

        // Se define un ArrayList
        ArrayList lista = new ArrayList();
        // Se agregan a la lista las instancias de la clase Producto.
        lista.add(pan);
        lista.add(leche);
        lista.add(manzanas);
        lista.add(brocoli);

        // Indica el índice de insercion
        lista.add(1, carne);
        lista.add(carne);

        // Imprimir contenido de ArrayLists
        System.out.println("Lista del mercado con " + lista.size() + " productos");
```

```
// Definir Iterator para extraer e imprimir los objetos
Iterator it = lista.iterator();
while (it.hasNext()) {
    Object objeto = it.next();
    Producto producto = (Producto) objeto;
    System.out.println(producto);
}

// Eliminar elemento de ArrayList
lista.remove(2);
System.out.println("Lista del mercado con " + lista.size() + " productos");

// Definir Iterator para extraer e imprimir valores
Iterator it2 = lista.iterator();
while (it2.hasNext()) {
    Producto producto = (Producto) it2.next();
    System.out.println(producto.getNombre() + " - " + producto.getCantidad());
}

// Eliminar todos los valores del ArrayList
lista.clear();
System.out.println("Lista del mercado con " + lista.size() + " productos");
}
}
```

A continuación, tenemos el resultado de su ejecución:

```
run:
Lista del mercado con 6 productos
colecciones.Producto@2a139a55
colecciones.Producto@15db9742
colecciones.Producto@6d06d69c
colecciones.Producto@7852e922
colecciones.Producto@4e25154f
colecciones.Producto@15db9742
Lista del mercado con 5 productos
Pan - 6
Carne - 2
Manzanas - 5
Brocoli - 2
Carne - 2
Lista del mercado con 0 productos
```

Objetos Duplicados

Debe notar que un objeto esta duplicado, se está considerando dos veces.

EVITAR OBJETOS DUPLICADOS: HASHSET

- ✓ Primero modificamos la clase Producto agregando dos métodos equals y hashCode
- ✓ El método equals desarrolla como se comparan dos objetos
- ✓ El método hashCode devuelve un identificador único.

Producto.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Producto {

    private String nombre;
    private int cantidad;

    public Producto() {
    }

    public Producto(String nombre, int cantidad) {
        this.nombre = nombre;
        this.cantidad = cantidad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }
}
```

```
@Override
public boolean equals(Object objeto) {
    boolean rpta = false;
    if (objeto != null) {
        Producto producto = (Producto) objeto;
        if (this.getNombre().equals(producto.getNombre())) {
            return true;
        }
    }
    return rpta;
}

@Override
public int hashCode() {
    return this.getNombre().hashCode();
}
```

- ✓ Aun cuando se agregaron 6 elementos, la lista solo cuenta con 5. Set no permite duplicados.
- ✓ La evaluación de duplicidad de objetos se realiza mediante los métodos equals y hashCode.
- ✓ Un Set no cuenta con índice, por lo que para eliminar un elemento se indica el objeto.

MercadoHashSet.java

```
import java.util.HashSet;
import java.util.Iterator;

/**
 *
 * @teacher
 * @blog
 * @email
 */
public class MercadoHashSet {

    public static void main(String args[]) {

        // Definir 5 instancias de la Clase Producto
        Producto pan = new Producto("Pan", 6);
        Producto leche = new Producto("Leche", 2);
        Producto manzanas = new Producto("Manzanas", 5);
        Producto brocoli = new Producto("Brocoli", 2);
        Producto carne = new Producto("Carne", 2);
        Producto res = new Producto("Carne", 3);
```

```
// Definir un HashSet
HashSet lista = new HashSet();
lista.add(pan);
lista.add(leche);
lista.add(manzanas);
lista.add(brocoli);
lista.add(carne);
lista.add(res);

// Imprimir contenido de HashSet
// Aunque son insertados 6 elementos, el HashSet solo contiene 5
// Se debe a que un Set no permite elementos duplicados.
System.out.println("Lista del mercado con "
    + lista.size() + " productos");

// Definir Iterator para extraer e imprimir valores
for (Iterator it = lista.iterator(); it.hasNext();) {
    Object objeto = it.next();
    Producto producto = (Producto) objeto;
    System.out.println(producto);
}

// No es posible eliminar elementos por indice
// En un HashSet solo se elimina por valor de Objeto
lista.remove(manzanas);
System.out.println("Lista del mercado con "
    + lista.size() + " productos");
for (Iterator it2 = lista.iterator(); it2.hasNext();) {
    Producto producto = (Producto) it2.next();
    System.out.println(producto.getNombre() + " - " + producto.getCantidad());
}

// Eliminar todos los valores del ArrayList
lista.clear();
System.out.println("Lista del mercado con "
    + lista.size() + " productos");
}
}
```

A continuación, tenemos el resultado de su ejecución:

```
run:
Lista del mercado con 5 productos
colecciones.Producto@6c2e9d68
colecciones.Producto@3ddf86b
colecciones.Producto@45e6467
colecciones.Producto@1387d
colecciones.Producto@c90fb9f
Lista del mercado con 4 productos
Brocoli - 2
Carne - 2
Leche - 2
Pan - 6
Lista del mercado con 0 productos
```

MANEJAR COLECCIONES ORDENADAS: TREESET

- ✓ Primero modificamos la clase Producto implementando un comparador para el ordenamiento.
- ✓ La clase implementa la interface Comparable.
- ✓ El método compareTo de la interfase Comparable indica que atributos se usaran para comparar.

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Producto implements Comparable {

    private String nombre;
    private int cantidad;

    public Producto() {

    }

    public Producto(String nombre, int cantidad) {
        this.nombre = nombre;
        this.cantidad = cantidad;
    }

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

```

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public int getCantidad() {
    return cantidad;
}

public void setCantidad(int cantidad) {
    this.cantidad = cantidad;
}

@Override
public boolean equals(Object objeto) {
    boolean rpta = false;
    if (objeto != null) {
        Producto producto = (Producto) objeto;
        if (this.getNombre().equals(producto.getNombre())) {
            return true;
        }
    }
    return rpta;
}

@Override
public int hashCode() {
    return this.getNombre().hashCode();
}

@Override
public int compareTo(Object objeto) {
    // Indica en base a que atributos se compara el objeto
    // Devuelve +1 si this es > que objeto
    // Devuelve -1 si this es < que objeto
    // Devuelve 0 si son iguales
    Producto producto = (Producto) objeto;
    String nombreObjeto = producto.getNombre().toLowerCase();
    String nombreThis = this.getNombre().toLowerCase();
    return (nombreThis.compareTo(nombreObjeto));
}
}

```

- ✓ Un TreeSet no permite elementos duplicados.
- ✓ Un TreeSet mantiene la lista ordenada.
- ✓ El elemento a comparar debe contar con métodos equals, hashCode y compareTo.

MercadoTreeSet.java

```
import java.util.Collection;
import java.util.TreeSet;

/**
 *
 * @teacher
 * @blog
 * @email
 */
public class MercadoTreeSet {

    public static void main(String args[]) {

        // Definir 5 instancias de la Clase Producto
        Producto pan = new Producto("Pan", 6);
        Producto leche = new Producto("Leche", 2);
        Producto manzanas = new Producto("Manzanas", 5);
        Producto brocoli = new Producto("Brocoli", 2);
        Producto carne = new Producto("Carne", 2);
        Producto res = new Producto("Carne", 3);

        // Definir un TreeSet
        TreeSet lista = new TreeSet();
        lista.add(pan);
        lista.add(leche);
        lista.add(manzanas);
        lista.add(brocoli);
        lista.add(carne);
        lista.add(res);

        // Imprimir contenido de TreeSet
        // Aunque se agregan 6 elementos, el TreeSet solo contiene 5
        // TreeSet no permite elementos duplicados,
```



```
// TreeSet detecta que el elemento "Carne" esta duplicado
// Notese que el orden del TreeSet refleja un orden ascendente
mostrarLista(lista);

// No es posible eliminar elementos por indice
// Un TreeSet solo elimina por valor de Objeto
lista.remove(manzanas);
mostrarLista(lista);

// Eliminar todos los valores del TreeSet
lista.clear();
mostrarLista(lista);
}

public static void mostrarLista(Collection<Producto> lista) {
    System.out.println("Lista del mercado con "
        + lista.size() + " productos");
    for (Producto producto : lista) {
        System.out.println(producto.getNombre() + " - " + producto.getCantidad());
    }
}
}
```

A continuación, tenemos el resultado de su ejecución:

```
run:
Lista del mercado con 5 productos
Brocoli - 2
Carne - 2
Leche - 2
Manzanas - 5
Pan - 6
Lista del mercado con 4 productos
Brocoli - 2
Carne - 2
Leche - 2
Pan - 6
Lista del mercado con 0 productos
```

ORDENAR Y BUSCAR EN COLECCIONES: COLLECTIONS

- ✓ La clase Collections (que no es la interface Collection) nos permite ordenar y buscar elementos en listas.
- ✓ Se usaran los métodos sort y binarySearch
- ✓ Los objetos de la lista deben tener métodos equals, hashCode y compareTo adecuados.

Producto.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class Producto implements Comparable {

    private String nombre;
    private int cantidad;

    public Producto() {
    }

    public Producto(String nombre, int cantidad) {
        this.nombre = nombre;
        this.cantidad = cantidad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    @Override
    public boolean equals(Object objeto) {
        boolean rpta = false;
        if (objeto != null) {
            Producto producto = (Producto) objeto;
            if (this.getNombre().equals(producto.getNombre())) {
                return true;
            }
        }
        return rpta;
    }
}
```

```

@Override
public int hashCode() {
    return this.getNombre().hashCode();
}

@Override
public int compareTo(Object objeto) {
    // Indica en base a que atributos se compara el objeto
    // Devuelve +1 si this es > que objeto
    // Devuelve -1 si this es < que objeto
    // Devuelve 0 si son iguales

    // Dependera del argumento como comparar los atributos.
    String nombreObjeto;
    if (objeto instanceof Producto) {
        Producto producto = (Producto) objeto;
        nombreObjeto = producto.getNombre().toLowerCase();
    } else if (objeto instanceof String) {
        String producto = (String) objeto;
        nombreObjeto = producto.toLowerCase();
    } else {
        nombreObjeto = "";
    }

    String nombreThis = this.getNombre().toLowerCase();
    return (nombreThis.compareTo(nombreObjeto));
}
}

```

MercadoCollections.java

```

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class MercadoCollections {

```

```
public static void main(String args[]) {
    // Definir 5 instancias de la Clase Producto
    Producto pan = new Producto("Pan", 6);
    Producto leche = new Producto("Leche", 2);
    Producto manzanas = new Producto("Manzanas", 5);
    Producto brocoli = new Producto("Brocoli", 2);
    Producto carne = new Producto("Carne", 2);

    // Definir un ArrayList
    ArrayList lista = new ArrayList();

    // Colocar Instancias de Producto en ArrayList
    lista.add(pan);
    lista.add(leche);
    lista.add(manzanas);
    lista.add(brocoli);
    lista.add(1, carne);

    // Imprimir contenido de ArrayList
    mostrarLista(lista);

    // Ordenar elemntos de ArrayList
    Collections.sort(lista);

    // Imprimir contenido de ArrayList
    mostrarLista(lista);

    // Buscar un elemento que se compare con Pan de tipo String
    String buscar = "Pan";
    int indice = Collections.binarySearch(lista, buscar);

    System.out.println(buscar + " es el elemento " + indice);
}

public static void mostrarLista(ArrayList<Producto> lista) {
    System.out.println("Lista del mercado con "
        + lista.size() + " productos");
    for (int i = 0; i < lista.size(); i++) {
        Producto producto = lista.get(i);
        System.out.println(i + ".- " + producto.getNombre()
            + "|" + producto.getCantidad());
    }
}

}
```

A continuación tenemos el resultado de su ejecución:

```
run:
Lista del mercado con 5 productos
0.- Pan|6
1.- Carne|2
2.- Leche|2
3.- Manzanas|5
4.- Brocoli|2
Lista del mercado con 5 productos
0.- Brocoli|2
1.- Carne|2
2.- Leche|2
3.- Manzanas|5
4.- Pan|6
Pan es el elemento 4
```

EJEMPLO DE CLASES IMPLEMENTADAS: MAP

- ✓ La clase que implementan la interfase **Map** es **HashMap**.
- ✓ La clase que implementa la sub-interfase **SortedMap** es **TreeMap**.

EJEMPLO DE HASHMAP

- ✓ Se define una instancia de la clase HashMap
- ✓ Se colocan 9 pares clave-valor con el método put
- ✓ Se muestra el contenido mediante un iterador que extrae los valores del HashMap
- ✓ Se define un arreglo de String con tres claves para eliminar de la agenda
- ✓ Se elimina las claves de la agenda
- ✓ Verifique si permite duplicados

AgendaHashMap.java

```
import java.util.HashMap;
import java.util.Map;

/**
 *
 * @teacher
 * @blog
 * @email
 */
public class AgendaHashMap {

    public static void main(String args[]) {
        // Definir un HashMap
        HashMap<String, String> agenda = new HashMap();

        // Agregar pares "clave"- "valor" al HashMap
        agenda.put("Doctor", "(+52)-4000-5000");
        agenda.put("Casa", "(888)-4500-3400");
        agenda.put("Hermano", "(575)-2042-3233");
        agenda.put("Tio", "(421)-1010-0020");
        agenda.put("Suegros", "(334)-6105-4334");
        agenda.put("Oficina", "(304)-5205-8454");
        agenda.put("Abogado", "(756)-1205-3454");
        agenda.put("Papa", "(55)-9555-3270");
        agenda.put("Tienda", "(874)-2400-8600");

        // Definir Iterator para extraer/imprimir valores
        mostrarMapa(agenda);

        // Definir un arreglo con valores determinados
        String personas[] = {"Tio", "Suegros", "Abogado"};

        // Eliminar los valores contenidos en el arreglo
        for (int i = 0; i < personas.length; i++) {
            agenda.remove(personas[i]);
        }
        mostrarMapa(agenda);
    }

    public static void mostrarMapa(Map<String, String> agenda) {
        System.out.println("AGENDA CON " + agenda.size() + " TELEFONOS");
        for (String key : agenda.keySet()) {
            String value = (String) agenda.get(key);
            System.out.println(key + " : " + value);
        }
    }
}
```

A continuación tenemos el resultado de su ejecución:

run:

AGENDA CON 9 TELEFONOS

Casa : (888)-4500-3400
Tienda : (874)-2400-8600
Papa : (55)-9555-3270
Hermano : (575)-2042-3233
Doctor : (+52)-4000-5000
Tio : (421)-1010-0020
Oficina : (304)-5205-8454
Suegros : (334)-6105-4334
Abogado : (756)-1205-3454

AGENDA CON 6 TELEFONOS

Casa : (888)-4500-3400
Tienda : (874)-2400-8600
Papa : (55)-9555-3270
Hermano : (575)-2042-3233
Doctor : (+52)-4000-5000
Oficina : (304)-5205-8454

EJEMPLO DE TREEMAP

- ✓ En un TreeMap los elementos están ordenados por la clave
- ✓ Luego se definen dos referencias de la interfase SortedMap
- ✓ En la primera se colocan las claves que se encuentran entre A y O. Metodo submap("A", "O")
- ✓ La segunda almacena las claves desde la P hacia el final.
- ✓ Estas comparaciones se han hecho con la clase String.
- ✓ Otras clases deberán definir sus propios métodos compareTo, equals y hashCode.

AgendaTreeMap.java

```
import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;

/**
 *
 * @teacher
 * @blog
 * @email
 */
public class AgendaTreeMap {

    public static void main(String args[]) {

        // Definir un TreeMap
        TreeMap<String, String> agenda = new TreeMap();

        // Agregar pares "clave"- "valor" al HashMap
        agenda.put("Doctor", "(+52)-4000-5000");
        agenda.put("Casa", "(888)-4500-3400");
        agenda.put("Hermano", "(575)-2042-3233");
        agenda.put("Tio", "(421)-1010-0020");
        agenda.put("Suegros", "(334)-6105-4334");
        agenda.put("Oficina", "(304)-5205-8454");
        agenda.put("Abogado", "(756)-1205-3454");
        agenda.put("Papa", "(55)-9555-3270");
        agenda.put("Tienda", "(874)-2400-8600");

        // Notese que el orden del TreeMap refleja un orden ascendente
        // en sus elementos independientemente del orden de insercion.
        // Debido al uso de String se refleja un orden alfabetico
        mostrarMapa(agenda);
        // Definir dos TreeMap nuevos
        SortedMap<String, String> agendaAO = agenda.subMap("A", "O");
        SortedMap<String, String> agendaPZ = agenda.tailMap("P");

        System.out.println("---- Agenda A-O ----");
        mostrarMapa(agendaAO);
        System.out.println("---- Agenda P-Z ----");
        mostrarMapa(agendaPZ);
    }

    public static void mostrarMapa(Map<String, String> agenda) {
```



```

        System.out.println("AGENDA CON " + agenda.size() + " TELEFONOS");
        for (String key : agenda.keySet()) {
            String value = (String) agenda.get(key);
            System.out.println(key + " : " + value);
        }
    }
}

```

A continuación, se muestra el resultado de su ejecución:

run:

run:

```

AGENDA CON 9 TELEFONOS
Abogado : (756)-1205-3454
Casa : (888)-4500-3400
Doctor : (+52)-4000-5000
Hermano : (575)-2042-3233
Oficina : (304)-5205-8454
Papa : (55)-9555-3270
Suegros : (334)-6105-4334
Tienda : (874)-2400-8600
Tio : (421)-1010-0020

```

---- Agenda A-O ----

```

AGENDA CON 4 TELEFONOS
Abogado : (756)-1205-3454
Casa : (888)-4500-3400
Doctor : (+52)-4000-5000
Hermano : (575)-2042-3233

```

---- Agenda P-Z ----

```

AGENDA CON 4 TELEFONOS
Papa : (55)-9555-3270
Suegros : (334)-6105-4334
Tienda : (874)-2400-8600
Tio : (421)-1010-0020

```

EXCEPCIONES

Son eventos que interrumpen la ejecución de un programa, por ejemplo:

- ✓ Usar un índice fuera de los límites de un arreglo
- ✓ Dividir entre cero
- ✓ ejecutar métodos de objetos nulos.

¿QUÉ ES UN ERROR?

En java es cuando la situación es irrecuperable y termina el programa.

- ✓ No hay memoria para correr JVM
- ✓ Errores internos en JVM

¿CUÁL ES LA DIFERENCIA?

Una excepción se puede controlar en el programa. Un error no.

CARACTERÍSTICAS DE JAVA

Cuando ocurre una excepción en un método, Java lanza (throw) una excepción (Exception).

El objeto Exception generado contiene el tipo de excepción, y el estado del programa cuando ocurrió el error.

El manejo de excepciones en java permite separarlos del algoritmo principal.

El resultado es un código más legible y menos propenso a errores de programación.

MANEJO TRADICIONAL DE ERRORES

```
int leerRegistroArchivo() {  
    int errorCode = 0;  
    abrirArchivo();  
    if (errorAbrirArchivo) {  
        errorCode = OPEN_ERROR;  
    }else {  
        leerArchivo();  
        if (errorLeerArchivo) {  
            errorCode = READ_ERROR;  
        }  
        cerrarArchivo();  
        if (errorCerrarArchivo) {  
            errorCode = CLOSE_ERROR;  
        }  
    }  
    return errorCode;  
}
```

MANEJO DE EXCEPCIONES EN JAVA

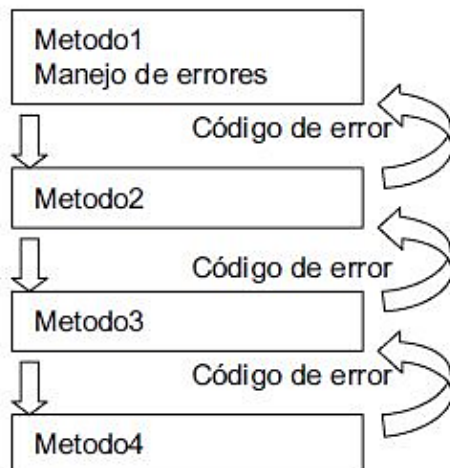
```
leerRegistroArchivo() {  
    try {  
        abrirArchivo();  
        leerArchivo();  
        cerrarArchivo();  
    }catch (errorAbrirArchivo) {  
        manejarErrorAbrirArchivo;  
    }catch (errorLeerArchivo) {  
        manejarErrorLeerArchivo;  
    }catch (errorCerrarArchivo) {  
        manejarErrorCerrarArchivo;  
    }  
}
```

Java separa los detalles del manejo de errores del código principal, obteniéndose un código más legible y menos propenso a errores de codificación.

EXCEPCIONES

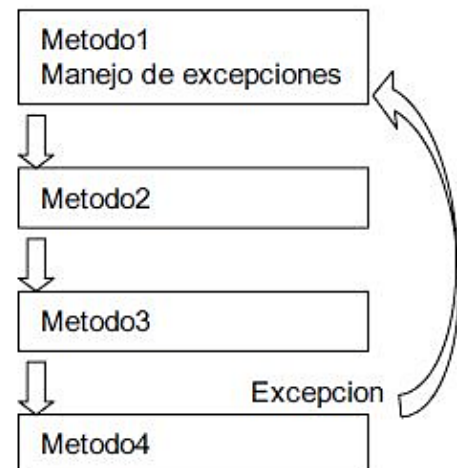
Devolviendo la excepción hasta el manejador de excepciones. No se requiere que cada método invocado maneje la excepción, sino únicamente lo hará el primer manejador de excepciones de la lista de métodos invocados.

MANEO TRADICIONAL DE ERRORES



El método4 retorna el código de error al método3.
El método3 retorna el código de error al método2
El método2 retorna el código de error al método1
El método1 maneja el error.

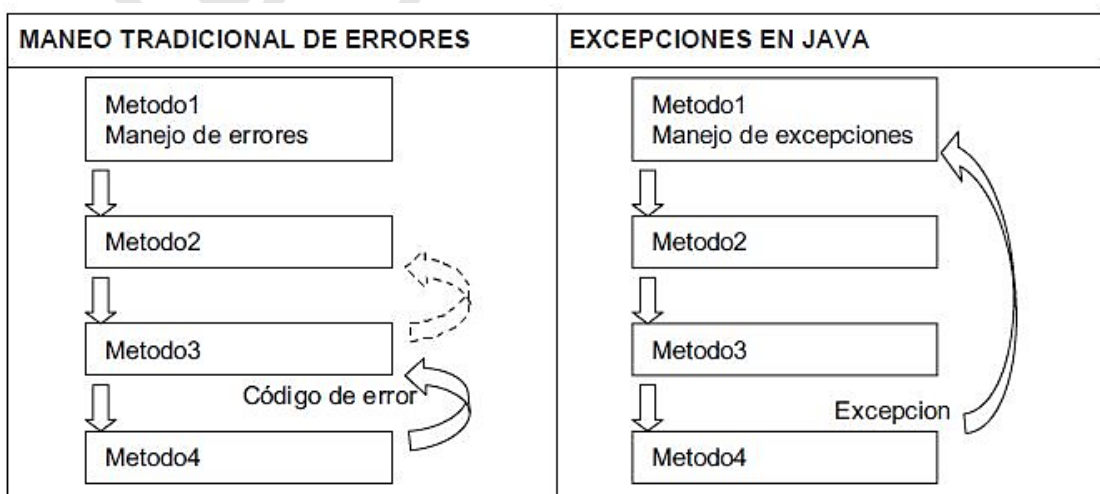
EXCEPCIONES EN JAVA



El método4 lanza una excepción
El método1 captura la excepción y la maneja

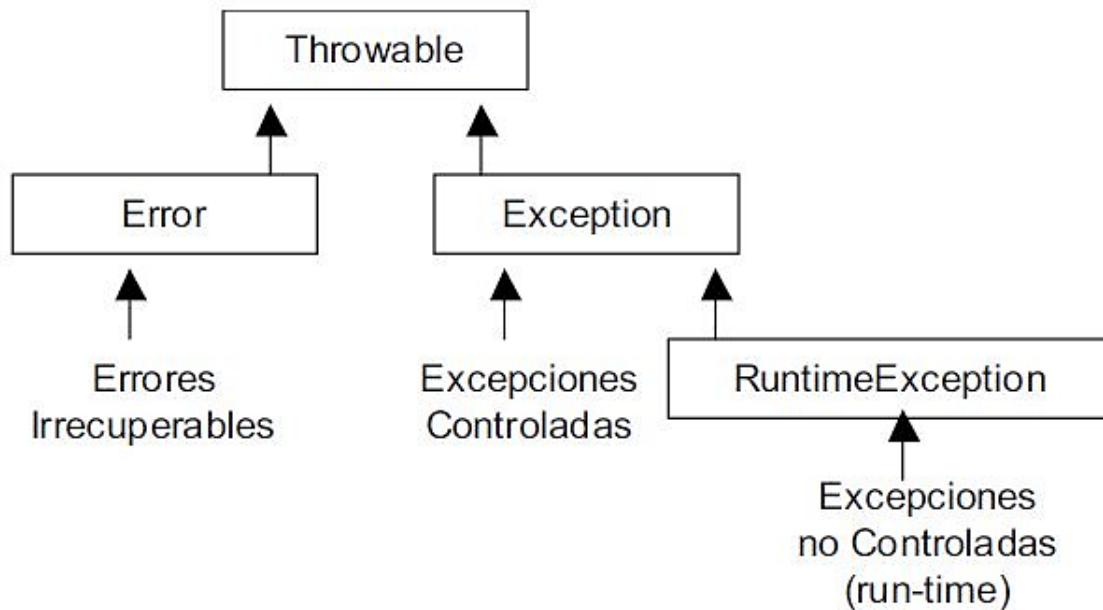
LAS EXCEPCIONES NO PUEDEN IGNORARSE

Una vez que un método lanza un error no puede ignorarse a diferencia de la programación tradicional donde se debe controlar en cada punto de invocación.



THROWABLE

Todos los errores y excepciones heredan de la clase Throwable



ERRORES

Heredan de la clase **Error**; estos se generan cuando ocurren errores fatales para el programa como, por ejemplo: cuando la memoria está llena o cuando es imposible encontrar una clase requerida.

EXCEPCIONES NO CONTROLADAS

Heredan de la clase **RuntimeException**. Estas ocurren cuando por ejemplo se divide entre cero o se intenta acceder a un elemento del arreglo fuera de sus límites. Mediante código se puede capturar, manipular o ignorar estas excepciones. Si no maneja el error, Java terminará el programa indicando el estado del error.

EXCEPCIONES CONTROLADAS

Heredan de la clase **Exception**. Estas deben ser capturadas y manejadas en algún lugar de la aplicación. Las excepciones creadas por el programador serán heredadas de la clase **Exception**.

¿Qué se puede hacer con una Excepción?

- ✓ Capturar la excepción y manipularla
- ✓ Pasar la excepción al método invocador.
- ✓ Capturar la excepción y lanzar una excepción diferente.

EXCEPCIONES NO CONTROLADAS

No necesitan ser controladas en el código.

El JVM terminara el programa cuando las encuentra.

Si no desea que el programa termine tendrá que manejarlas.

<ul style="list-style-type: none"> ✓ Encierre el código del método en un bloque try ✓ Maneje cada exception en un bloque catch. ✓ Cualquier proceso final realícelo en un bloque finally. Este bloque siempre se ejecutara, ocurra o no una excepción. 	<pre>try { // código del método } catch (exception1) { // manejar la excepción1 } catch (exception2) { // manejar la excepción2 } ... finally { // cualquier otro proceso final }</pre>
---	---

La documentación del java indicara que excepciones lanza los métodos del java.

```
Clase : java.io.FileInputStream
Método: public FileInputStream(String name) throws FileNotFoundException
Método: public int read() throws IOException
```

CAPTURANDO UNA EXCEPCIÓN

Este ejemplo convierte una cadena en un entero

```
int cantidad;
String cadena = "5";
try {
    cantidad = Integer.parseInt(cadena);
}
catch ( NumberFormatException e) {
    System.err.println(cadena + " no es un entero");
}
```

Pero si la cadena no representa un número entero, se generará la excepción `NumberFormatException`.

CAPTURANDO MÚLTIPLES EXCEPCIONES

Este ejemplo convierte una cadena en un entero y realiza una división.

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestMultiException {

    public static void main(String[] args) {
        int cantidad = 0;
        int divisor = 0;
        String cadena = "5";
        try {
            cantidad = Integer.parseInt(cadena);
            System.out.println(cantidad);
            int resultado = cantidad / divisor;
            System.out.println(resultado);
        } catch (NumberFormatException e) {
            System.err.println(cadena + " no es un entero");
        } catch (ArithmeticException e) {
            System.err.println("Error en " + cantidad + "/" + divisor);
        }
    }
}
```

A continuación, tenemos el resultado de su ejecución:

```
run:
5
Error en 5/0
```

El error se ha presentado al momento de realizar la división.

Reto:

Realice otras pruebas cambiando el valor de las variables.

EJECUCIÓN DEL BLOQUE FINALLY

El bloque finally siempre se ejecuta dependiendo como se termina el bloque try.

- ✓ Hasta terminar la última sentencia del bloque try.
- ✓ Debido a sentencias return o break en el bloque try.
- ✓ Debido a una excepción.

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestFinally {

    public static void main(String[] args) {
        int cantidad = 0;
        int divisor = 0;
        String cadena = "5";
        try {
            if (cadena.equals("5")) {
                return;
            }
            cantidad = Integer.parseInt(cadena);
            System.out.println(cantidad);
            int resultado = cantidad / divisor;
            System.out.println(resultado);
        } catch (NumberFormatException e) {
            System.err.println(cadena + " no es un entero");
        } catch (ArithmeticException e) {
            System.err.println("Error en " + cantidad + "/" + divisor);
        } finally {
            System.out.println("Se trabajó con " + cadena + " y " + divisor);
        }
    }
}
```

A continuación, tenemos el resultado de su ejecución:

```
run:
Se trabajó con 5 y 0
```

Reto:

Realice otras pruebas cambiando el valor de las variables.

COMO PASAR LA EXCEPCIÓN AL MÉTODO INVOCADO

Para pasar la excepción al invocador, se declara la cláusula throws. La excepción se propaga al método que lo invoca. En el ejemplo, las excepciones de los métodos se manejan en el método invocador.

```
public void metodoInvocador() {
    try {
        miMetodo();
        getResultado();
    }
    catch ( . . . ) {
    }
    finally {
    }
}

public int miMetodo() throws Exception {
    // Código que podría lanzar la Exception
}

public int getResultado() throws NumberFormatException {
    // Código que podría lanzar la exception NumberFormatException
}
```

TestThrows.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class TestThrows {

    public static void main(String[] args) {
        String cadena = "abcde";
        int posicion = 6;
        char letra = ' ';
        try {
            letra = getLetra(cadena, posicion);
            System.out.println(letra);
        } catch (IndexOutOfBoundsException e) {
            System.err.println("Error en " + cadena + " " + posicion);
        }
    }

    public static char getLetra(String cadena, int posicion)
        throws IndexOutOfBoundsException {
        char c = cadena.charAt(posicion);
        return c;
    }
}
```


El resultado de su ejecución es:

```
run:
Error en abcde 6
```

El error se debe a que el índice 6 está fuera del tamaño de la cadena.

Reto:

Realice otras pruebas cambiando el valor de la variable posición.

COMO LANZAR UNA EXCEPCIÓN

Para lanzar una excepción use las declaraciones throws y throw. Puede crear excepciones para manejar posibles problemas en el código.

```
public String getValor(int indice) throws IndexOutOfBoundsException {
    if (indice < 0 || indice > 100) {
        throw IndexOutOfBoundsException();
    }
}
```

COMO CREAR UNA EXCEPCIÓN

Para crear su propia excepción tiene que heredarla de la clase Exception. Puede ser usado en caso quiera tratar cada problema en el código en forma diferente. Por ejemplo el manejo del archivo1 puede tratarlo con una excepción1. El manejo de un archivo2 puede tratarlo con una excepcion2. Cada archivo tendría su propia excepción.

UserFileException.java

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public class UserFileException extends Exception {

    public UserFileException (String mensaje) {
        super(mensaje);
    }

}
```

COMO CAPTURAR UNA EXCEPCIÓN Y LANZAR OTRA DIFERENTE

En el ejemplo si ocurre un problema de IO (entrada/salida) se captura la excepción y se lanza una excepción propia.

```
/**
 *
 * @teacher
 * @blog
 * @email
 */
public void ReadUserFile throws UserFileException {

    try {
        // Código que manipula un archivo
    }
    catch (IOException e) {
        throw new UserFileException(e.toString());
    }

}
```

B U E N A S P R Á C T I C A S

PROGRAMACIÓN EN CAPAS

Introducción

Cuando se construye software como producto empresarial o comercial, se llevan a cabo varias técnicas de manera que el desarrollo se haga en forma ordenada y así poder asegurar un avance continuo del proyecto, un producto final de calidad, y además realizar posteriores mejoras sea una tarea más fácil.

Existen muchas prácticas de programación, dependiendo del tipo de software que se va a desarrollar y de la disciplina o disciplinas de programación que se utilicen en el desarrollo del producto.

Una de las más utilizadas se llama la programación por capas, que consiste en dividir el código fuente según su funcionalidad principal.

La programación para lograr sacarle el mayor provecho a la programación por capas se necesita seguir una serie de pasos complejos los cuales primeramente deben ser definidos para cada proyecto específico, luego deben ser revisados para asegurarse de que el modelo adoptado cumpla con las normas necesarias para que la aplicación sea del agrado del usuario, y por último debe ser implementado por el grupo de desarrollo encargado para tal fin, los cuales siguiendo el modelo propuesto obtienen una herramienta útil para facilitar la labor de programación dividiendo la aplicación en módulos y capas fáciles de pulir.

CARACTERÍSTICAS DE LA PROGRAMACIÓN EN CAPAS

La programación por capas es una técnica de ingeniería de software propia de la programación orientada por objetos, éstos se organizan principalmente en 3 capas: la capa de presentación o frontera, la capa de lógica de negocio o control, y la capa de datos.

Siguiendo el modelo, el desarrollador se asegura avanzar en la programación del proyecto de una forma ordenada, lo cual beneficia en cuanto a reducción de costos por tiempo, debido a que se podrá avanzar de manera más segura en el desarrollo, al ser dividida la aplicación general en varios módulos y capas que pueden ser tratados de manera independiente y hasta en forma paralela.

Por otra parte, otra característica importante de recalcar es la facilidad para las actualizaciones de la aplicación. En este aspecto, la programación en capas juega un papel de suma importancia ya que sigue un estándar conocido en el ambiente de desarrollo de aplicaciones, lo cual da al programador una guía para hacer mejoras a la aplicación sin que esto sea una tarea tediosa y desgastante, siguiendo el estándar establecido para tal fin y dividiendo las tareas en partes específicas para cada capa del proyecto.



Las principales capas que siempre deben estar en este modelo son:

- ✓ Capa de Presentación o Frontera
- ✓ Capa de Lógica de Negocio
- ✓ Capa de Datos

Capa de Presentación o Frontera

La presentación del programa ante el usuario, debe manejar interfaces que cumplan con el objetivo principal de este componente, el cual es facilitar al usuario la interacción con la aplicación. Para esto se utilizan patrones predefinidos para cada tipo de aplicación y para cada necesidad del usuario. La interfaz debe ser amigable y fácil de utilizar, ya que el usuario final es el que se va a encargar de utilizar el sistema y de dar retroalimentación al equipo de desarrollo en caso de que haya algo que mejorar.

Las interfaces deben ser consistentes con la información que se requiere, no se deben utilizar más campos de los necesarios, así como la información requerida tiene que ser especificada de manera clara y concisa, no debe haber más que lo necesario en cada formulario y por último, las interfaces deben satisfacer los requerimientos del usuario, por lo cual no se debe excluir información solicitada por el usuario final y no se debe incluir información no solicitada por el mismo.

Dentro de la parte técnica, la capa de presentación contiene los objetos encargados de comunicar al usuario con el sistema mediante el intercambio de información, capturando y desplegando los datos necesarios para realizar alguna tarea. En esta capa los datos se procesan de manera superficial por ejemplo, para determinar la validez de su formato o para darles algún orden específico.

Esta capa se comunica únicamente con la Capa de Lógica de Negocio.

Capa de Lógica de Negocio o Control

Es llamada capa de reglas de negocio porque en esta se definen todas las reglas que se deben cumplir para una correcta ejecución del programa.

Es aquí donde se encuentra toda la lógica del programa, así como las estructuras de datos y objetos encargados para la manipulación de los datos existentes, así como el procesamiento de la información ingresada o solicitada por el usuario en la capa de presentación.

Representa el corazón de la aplicación ya que se comunica con todas las demás capas para poder llevar a cabo las tareas. Por ejemplo, mediante la capa de presentación obtiene la información ingresada por el usuario, y despliega los resultados. Si la aplicación se comunica con otros sistemas que actúan en conjunto, lo hace mediante esta capa. También se comunica con la capa de datos para obtener información existente o ingresar nuevos datos.

Recibe los datos que ingresó el usuario del sistema mediante la capa de presentación, luego los procesa y crea objetos según lo que se necesite hacer con estos datos; esta acción se denomina encapsulamiento.

Al encapsular los datos, el programa asegura mantener la consistencia de los mismos, así como obtener información precisa de las bases de datos e ingresar en las mismas, solamente la información necesaria, asegurando así no tener datos duplicados ni en las bases de datos, ni en los reportes solicitados por el usuario.

Capa de Datos

Es la encargada de realizar transacciones con bases de datos y con otros sistemas para obtener o ingresar información al sistema.

El manejo de los datos debe realizarse de forma tal que haya consistencia en los mismos, de tal forma los datos que se ingresan así como los que se extraen de las bases de datos, deben ser consistentes y precisos.

Es en esta capa donde se definen las consultas a realizar en la base de datos, tanto las consultas simples como las consultas complejas para la generación de reportes más específicos.

Esta capa envía la información directamente a la capa de reglas de negocio para que sea procesada e ingresada en objetos según se necesite, esta acción se denomina encapsulamiento.

Ventajas y Desventajas

La programación en capas no es una técnica rígida que debe implementarse solamente de una forma, sino que los desarrolladores de proyectos tienen múltiples maneras de implementarla según las tecnologías y tendencias que se utilicen.

La satisfacción de los requerimientos del usuario es la base para escoger el modelo de implementación a seguir. La tendencia a utilizar el modelo de programación en capas es grande cuando se trata principalmente de aplicaciones empresariales donde se deben manejar gran cantidad de subsistemas y módulos, así como generar reportes lo suficientemente complejos como para necesitar un orden estricto a la hora de desarrollar el proyecto.

Dentro del concepto de programación en capas, existen dos términos esenciales para el mejor entendimiento de los conceptos relativos a esta metodología, es aquí donde radica la importancia de la cohesión y el acoplamiento dentro de una aplicación generada mediante este método.

COHESIÓN

Este término es utilizado para describir el comportamiento que deben tener los módulos y objetos de un sistema o subsistema, comportamiento que describe la forma en que deben trabajar los objetos y módulos entre sí, con alta cohesión para que trabajando en conjunto los módulos y objetos puedan alcanzar un solo propósito de manera más eficaz y rápida.

Determina que las operaciones de un objeto deben trabajar en conjunto para alcanzar un propósito común. Es deseable que haya alta cohesión.

ACOPLAMIENTO

Se refiere al grado de dependencia que existe entre los módulos. Este grado de dependencia debe ser considerablemente bajo ya que el trabajo se divide en módulos para que cada uno tenga un funcionamiento específico y puede ser más factible la implementación por separado de cada uno. En caso de haber alto acoplamiento entre módulos no se estaría alcanzando el principal objetivo de este modelo, el cual es dividir una tarea grande en varias pequeñas, ya que los módulos actuarían como uno solo al estar altamente acoplados entre sí y se perdería el objetivo primordial de dividir el proyecto.

Ventajas

Al implementar este modelo de programación, se asegura un trabajo de forma ordenada y separada, debido a que sigue el principio de “divide y vencerás”.

Cada capa está dividida según su funcionalidad cuando se quiere modificar el sistema basta con cambiar un objeto o conjunto de objetos de una capa. Esto se llama modularidad.

Desventajas

Cuando se implementa un modelo de programación en capas, se debe llegar a un balance entre el número de capas y subcapas que componen el programa. Este debe ser el necesario y suficiente para realizar un trabajo específico con eficiencia y ser lo más modular posible.

De lo contrario se tiene una serie de desventajas como: pérdida de eficiencia, realización de trabajo innecesario o redundante entre capas, gasto de espacio de la aplicación debido a la expansión de las capas, o bien una alta dependencia entre los objetos y capas que contradice el objetivo principal del modelo.

Conclusiones

La programación en capas ha sido una de las últimas tendencias en cuanto a software comercial se refiere, es una tendencia que bien aplicada puede resultar en un desarrollo de software eficiente.

Sin embargo, no todos los equipos ni empresas desarrolladoras usan un sistema rígido, ni existe una forma estricta en la que tenga que implementarse el modelo de capas, cada quién debe hacerlo según sus necesidades, alcances y lo más importante; las tendencias y nuevas tecnologías que vayan surgiendo.

Es importante tener en cuenta que no importa el lenguaje de programación usado, o el tipo de implementación que se le dé al modelo; se debe buscar una alta cohesión y un bajo acoplamiento dentro de los objetos y capas para lograr que la aplicación sea fácilmente desarmable y sea más sencillo realizar mejora y actualizaciones al sistema.

PRINCIPIOS SOLID

SOLID es un acrónimo inventado por Robert C. Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.



El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo. Los costes de mantenimiento pueden abarcar el 80% de un proyecto de software por lo que hay que valorar un buen diseño.

S-Responsabilidad simple (Single responsibility)

Este principio trata de destinar cada clase a una finalidad sencilla y concreta. En muchas ocasiones estamos tentados a poner un método reutilizable que no tienen nada que ver con la clase simplemente porque lo utiliza. En ese momento pensamos "Ya que estamos aquí, para que voy a crear una clase para realizar esto. Directamente lo pongo aquí".

El problema surge cuando tenemos la necesidad de utilizar ese mismo método desde otra clase. Si no se refactoriza en ese momento y se crea una clase destinada para la finalidad del método, nos toparemos a largo plazo con que las clases realizan tareas que no deberían ser de su responsabilidad.

Con la anterior mentalidad nos encontraremos, por ejemplo, con un algoritmo de formateo de números en una clase destinada a leer de la base de datos porque fue el primer sitio donde se empezó a utilizar. Esto conlleva a tener métodos difíciles de detectar y encontrar de manera que el código hay que tenerlo memorizado en la cabeza.

O-Abierto/Cerrado (Open/Closed)

Principio atribuido a Bertrand Meyer que habla de crear clases extensibles sin necesidad de entrar al código fuente a modificarlo. Es decir, el diseño debe ser abierto para poderse extender pero cerrado para poderse modificar. Aunque dicho parece fácil, lo complicado es predecir por donde se debe extender y que no tengamos que modificarlo. Para

conseguir este principio hay que tener muy claro cómo va a funcionar la aplicación, por donde se puede extender y cómo van a interactuar las clases.

El uso más común de extensión es mediante la herencia y la reimplementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una interface de manera que podemos ejecutar cualquier clase que implemente ese interface. En todos los casos, el comportamiento de la clase cambia sin que hayamos tenido que tocar código interno.

Como ya he comentado llega un momento en que las necesidades pueden llegar a ser tan imprevisibles que nos topemos que con los métodos definidos en el interface o en los métodos extensibles, no sean suficientes para cubrir las necesidades. En este caso no habrá más remedio que romper este principio y refactorizar.

L-Sustitucion Liskov (Liskov substitution)

Este principio fue creado por Barbara Liskov y habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base. Cuando creamos clases derivadas debemos asegurarnos de no reimplementar métodos que hagan que los métodos de la clase base no funcionasen si se tratasen como un objeto de esa clase base.

I-Segregacion del interface (Interface segregation)

Este principio fue formulado por Robert C. Martin y trata de algo parecido al primer principio. Cuando se definen interfaces estos deben ser específicos a una finalidad concreta. Por ello, si tenemos que definir una serie de métodos abstractos que debe utilizar una clase a través de interfaces, es preferible tener muchos interfaces que definan pocos métodos que tener un interface con muchos métodos.

El objetivo de este principio es principalmente poder reaprovechar los interfaces en otras clases. Si tenemos un interface que compara y clona en el mismo interface, de manera más complicada se podrá utilizar en una clase que solo debe comparar o en otra que solo debe clonar.

D-Inversión de dependencias (Dependency inversion)

También fue definido por Robert C. Martin. El objetivo de este principio es conseguir desacoplar las clases. En todo diseño siempre debe existir un acoplamiento pero hay que evitarlo en la medida de lo posible. Un sistema no acoplado no hace nada pero un sistema altamente acoplado es muy difícil de mantener.

El objetivo de este principio es el uso de abstracciones para conseguir que una clase interactúe con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben conocer las clases de nivel inferior. Dicho de otro modo, no debe conocer los detalles. Existen diferentes patrones como la inyección de dependencias o service locator que nos permiten invertir el control.

SIMPLICIDAD DEL CODIGO

KISS

Kiss es un acrónimo en inglés que significa “Keep it simple, stupid”, que en español vendría siendo “Manténlo simple, estúpido”. Una traducción más suave que también se usa es “Keep it super simple”, es decir: “Manténlo súper simple”. El principio fue formulado por un ingeniero del ejército estadounidense en 1960, y propone que cualquier sistema que implementemos, un producto, un servicio, un sistema informático, las soluciones a un problema, etc.; son mucho mejores si se basan en la simplicidad y sencillez.

Pero la simplicidad no se debe confundir con falta de características o por implementaciones incompletas. Debe entenderse como un producto, servicio o solución que es completa justo con lo necesario. No más, no menos; sin florituras, adornos o extras innecesarios.

Kiss plantea que usted debe evitar cualquier complejidad que no agregue valor a la solución, y reducir la tendencia de muchas personas de “sobre complicar” las cosas.

El principio Kiss si bien ha sido aplicado principalmente en el desarrollo de software, también podría ser de utilidad para cualquier negocio en marcha o cualquier emprendimiento. Piense en el principio Kiss cuando ande en busca de una idea millonaria, o identifique que puede resolver una necesidad a través de un nuevo producto o servicio; o bien, cuando piense en mejorar un proceso actual.

La búsqueda de la solución perfecta y puede acarrear pérdidas de tiempo y dinero, con soluciones con mayores funcionalidades o características a las que el cliente necesita.

El principio Kiss implica entender a profundidad la verdadera necesidad del cliente así como las características y elementos esenciales y no esenciales de una solución, producto o servicio. Así podrá desechar aquello que no aporte valor. Pregúntese siempre si lo que está añadiendo es algo imprescindible para atender sus necesidades o las del cliente.

BENEFICIOS DEL PRINCIPIO KISS

Algunos de los beneficios que pueden obtener del principio Kiss para su negocio son los siguientes:

- ✓ Resolverá los problemas más rápido.
- ✓ Podrá lanzar sus productos y servicios con mayor rapidez y agilidad.
- ✓ Reducirá costos de desarrollo.
- ✓ El mantenimiento de los productos que lo requieran será más sencillo de ejecutar y a menor costo.
- ✓ Los productos o servicios serán de mayor calidad, ya que puede invertir más dinero en lo esencial, en lugar de invertir en cosas innecesarias.

Así que recuerde, la próxima vez que vaya a realizar cualquier trabajo o emprendimiento, ¡manténlo sencillo! No complique las cosas más allá de lo estrictamente necesario.

La perfección se alcanza, no cuando no hay nada más que añadir, sino cuando ya no queda nada más que quitar.

Antoine Saint Euxpéry

YAGNI

YAGNI significa "You Aren't Gonna Need It" (No vas a necesitarlo) Es uno de los principios del Extreme Programming que indica que un programador no debe agregar funcionalidades extras hasta que no sea necesario.

Aplicar siempre las cosas cuando realmente los necesita, no cuando lo que prevén que los necesita.

Ron Jeffries

La relación con el principio de "Last Responsible Moment" es clara, ya que nos debemos basar en hechos y no en suposiciones. Así evitamos sobre diseñar sistemas, una tendencia que retrasa y entorpece el alcance inicial de la tarea que estamos desarrollando.

NAVAJA DE OCCAM

La Navaja de Occam se remonta a un principio metodológico y filosófico, perfectamente aplicable al desarrollo de software, según el cual, "en igualdad de condiciones, la explicación más sencilla suele ser la correcta".

Este principio lo podemos usar tanto en el momento de implementar una solución como a la hora de encontrar el causante a un bug. ¿Cuántas veces nos habrá pasado que la metedura de pata más tonta es la causante del problema aunque hayamos estado comprobando lo más complejo?

SIMPLE NO SIGNIFICA IGNORAR LA COMPLEJIDAD REAL

Cada vez que se habla de simplicidad se cae en la trampa del exceso de simplicidad. Los principios anteriormente expuestos no significan que tengamos que ignorar la complejidad de los hechos reales. Un diseño de software simple es el que se centra en los requisitos actuales, pero no olvida necesidades futuras como el mantenimiento de software, extensibilidad o la reutilización. Al fin y al cabo, los diseños que buscan la simplicidad del código son más fáciles de adaptar a las necesidades futuras que los complejos.

REFLEXIÓN CUIDADOSA

Tanto el principio KISS como la Navaja de Occam pueden ser recursos incómodos para quienes gustan de ver el mundo en polaridades de bueno o malo, correcto o incorrecto. Son maneras de pensar instrumentales, de tránsito; herramientas para encarar la ambigüedad y seguir adelante sin entraparse en tediosas discusiones bizantinas sobre

suposiciones, gustos o creencias. También hay que aclarar que estos principios exigen una mentalidad abierta y crítica para testear los hechos con frecuencia y someterlos a revisión.

El principio KISS puede convertirse en una filosofía o estilo de vida, aplicable en cualquier ámbito.

PRINCIPIO DRY



En diseño orientado a objetos, El principio No te repitas (en inglés Don't Repeat Yourself o DRY, también conocido como Una vez y sólo una) es una filosofía de definición de procesos que promueve la reducción de la duplicación especialmente en programación. Según este principio toda pieza de información nunca debería ser duplicada debido a que la duplicación incrementa la dificultad en los cambios y evolución posterior, puede perjudicar la claridad y crear un espacio para posibles inconsistencias. Por "pieza de información" podemos entender, en un sentido amplio, desde datos almacenados en una base de datos pasando por el código fuente de un programa de software hasta llegar a información textual o documentación.

Cuando el principio DRY se aplica de forma eficiente los cambios en cualquier parte del proceso requieren cambios en un único lugar. Por el contrario, si algunas partes del proceso están repetidas por varios sitios, los cambios pueden provocar fallos con mayor facilidad si todos los sitios en los que aparece no se encuentran sincronizados.

Cada vez que repites un trozo de código (reglas, procesos, rutinas) debes acordarte de dónde las pones. Si posteriormente cambian las especificaciones, o encuentras una mejor forma de hacer las cosas, o te diste cuenta que hiciste algo que no debías, tienes que cambiar el código en todos los lugares donde existe. Si no lo haces, vas a tener un sistema que se comporta de una forma en unas ocasiones, y de otra forma en otras, porque utiliza estas representaciones inconsistentes, tendrás un sistema inconsistente.

Obsesiónate con este principio, cada vez que uses el editor de texto, y veas que estás repitiendo algo, haciendo mucho "copy & paste" de una serie de estructuras IF para ponerlos en dos o tres métodos diferentes, o si varias clases son muy similares, tal vez es momento de detenerte y hacer una refactorización para colocar todo eso en un sólo lugar y reutilizar toda esa lógica, debes utilizar la herencia, clases abstractas o cualquier recurso que te permita concentrar todo eso en un sólo sitio.

¿Cómo surge la duplicación?

- ✓ La duplicación puede parecer impuesta. El desarrollador puede pensar que el sistema parece requerir duplicación.
- ✓ La duplicación puede ser involuntaria. El desarrollador puede no darse cuenta de que está duplicando información.
- ✓ La duplicación por impaciencia. El desarrollador puede tener flojera y duplicar porque parece lo más fácil. Tarde o temprano se estará lamentando.
- ✓ La duplicación por falta de coordinación. Múltiples desarrolladores en un equipo o diferentes equipos pueden duplicar información.

Con respecto al código, no se trata de que los pedazos de código NUNCA se deban repetir, a veces es necesario y menos complejo dejar que algunas líneas se repitan en diferentes clases; donde en realidad lo debes aplicar es cuando tratas de hacer representaciones funcionales de tu aplicación: Ejemplo, si tienes una rutina que va a recuperar los datos de cliente y hacer cierta transformación a esos datos y lo harás a menudo en la aplicación, vale la pena hacer una función llamada (por ejemplo) "recuperaClienteInfo".

En muchas ocasiones, encontramos que una función existente hace "casi" lo mismo que queremos, pero necesitamos que haga algo extra o que no lo haga. Si te encuentras en este caso, en lugar de duplicar la función, mejor añádele un parámetro, y utiliza la información de ese parámetro para que la función haga el trabajo que necesitas. Una vez hecha esta operación, el problema que se presenta es rastrear el código existente y modificar por todos lados para añadir ese parámetro; puedes entonces tener varias alternativas:

- ✓ Revisa si tu lenguaje de programación detecta el número de parámetros recibidos y mediante esa función lee el nuevo parámetro y condiciona la nueva lógica.
- ✓ Hacer algo de "refactorización".
- ✓ Aquí tendrías que renombrar las funciones y adaptar tu código, es doloroso, pero a la larga te beneficiarás, sobre todo si planeas que tu programa sobreviva el devenir de los años y la modernización obligada en el futuro.
- ✓ Si usas algún lenguaje de Programación Orientada a Objetos haz uso de la sobrecarga de funciones.

UN EJEMPLO.

Supongamos que estamos haciendo una parte de un sistema en el cual se almacenará un catálogo de productos, digamos de frutas. Normalmente lo primero que haremos es crear la base de datos. En la base de datos se tendrían (por ejemplo) estos campos: id, código, nombre, precio. En la misma base de datos agregamos ciertas restricciones al producto: el código es único y es una cadena de 10 caracteres, el nombre no puede ir vacío, el precio no puede ser cero (observemos que incluso los tipos de esos datos radican en la creación de la tabla en la base de datos).

Una vez creada la base de datos, nos dirigimos a nuestro IDE para crear el código que trabajará con esos datos. Los enfoques varían mucho, y las maneras de trabajar con esos datos también. Pero en este caso vamos a trabajar con una capa de persistencia basada en un ORM (Object-Relational Mapping). En algún lado de la configuración del ORM estableceremos que la tabla fruta se mapeará al objeto Fruta. Que el objeto Fruta tendrá un id, un nombre, un código y un precio.

Con nuestro mapeo funcionando, nos dirigimos a crear nuestro objeto Fruta. Nuestro objeto tendrá un BigInteger para el id, tendrá un String para el código, otro para el nombre y un BigDecimal para el precio. Ahora crearemos los métodos de accesos con su set y su get. (Propiedades en Java). Si la capa de persistencia nos echa una mano no tendremos que verificar que el código sea único, pero si no, tendremos que codificarlo nosotros.

Ahora, si queremos que nuestro código sea escalable y pueda mantenerse a largo plazo, utilizaremos alguna implementación del patrón MVC (Model-View-Controller, o Modelo-Vista-Controlador). En cada uno pondremos los respectivos campos de la fruta: el id, el código, el nombre y el precio. Haremos un listado de las frutas que hay almacenadas, y una forma para agregar y/o editar entradas. En la vista le diremos al controlador qué desplegar, y haremos una tabla en el View. El model ya lo implementamos, que es nuestro objeto Fruta. Para el formulario haremos lo mismo, pondremos los campos de fruta y lo guardaremos. Para proteger nuestro modelo haremos algunas validaciones del lado del servidor, pero para darle una experiencia interactiva al usuario, también haremos validaciones en el lado del cliente. Usaremos, claro está, uno de esos bonitos frameworks que ahora nos hacen la vida más fácil con javascript. La validación del código único requerirá una llamada de Ajax, y las demás serán simples validaciones con Javascript.

Ahora estamos a punto con el manejo de frutas en el sistema. Tenemos un diseño robusto, altamente escalable, muy legible. Pasamos nuestro trabajo a control de calidad. Control de calidad nos informa que el máximo de caracteres de código es 10 y en el javascript del cliente nosotros pusimos 12. Bueno, un teclazo de más a cualquiera le pasa, ¿no?. Pasado un tiempo nos dicen que en el mapeo del ORM no pusimos correctamente el precio, y permitimos que sea vacío. Cosas, la prisa hizo que se nos pasara por alto. Pero pasado todo eso, todo bien.

Pasados unos días, nos mandan un requerimiento del cliente que pide que el código en vez de ser de 10 caracteres va a ser de 15 (cambiaron un par de cosas del negocio), y que hará falta otro código: el código de bodega que será de 20 caracteres y también hará falta el peso unitario, que será un decimal opcional.

En nuestra mente comienza a correr el tedioso proceso que tenemos que realizar: hacer los cambios en la base de datos, hacer los cambios en el mapeo, hacer los cambios en el objeto fruta, hacer los cambios en el controller y en el view para la lista, hacer los cambios en el controller y en el view (que incluye dos validaciones de javascript). Además qué fácil será confundirnos.

A todo esto la documentación estaba ya lista, y habrá que cambiarla, pero por falta de tiempo creemos que se quedará tal y como está.

He aquí lo que nos pasa cuando violamos el principio DRY. El principio DRY establece que la información, o el conocimiento debe estar almacenado en UN SOLO LUGAR, para evitar lo engorroso que es hacer cambios en mil lados cuando hacen, además de

ahorrarnos la tendencia a cometer errores cuando hay que cambiar muchas cosas que significan lo mismo.

Analicemos los cambios: agregar un campo y modificar otro. Este conocimiento, en gran parte está almacenado en la base de datos (aunque no todo, por ejemplo en la base de datos no podemos hacer ciertas validaciones para los códigos, o cosas parecidas).

Nadie negará la utilidad de patrones como el MVC, o de ideas como el ORM. Pero a veces esas soluciones crean una sobre ingeniería tan grande que no sé si compensan sus soluciones.



PATRONES DE SOFTWARE

INTRODUCCIÓN

Los Patrones de Software son recomendaciones probadas y sirven como modelo para dar soluciones a problemas comunes en el desarrollo del software. Son considerados en la actualidad como buenas prácticas en la ingeniería de software.

Un patrón define una posible solución correcta para un problema de software dentro de un contexto dado, describiendo las cualidades invariantes de todas las soluciones.

En 1979 el arquitecto Christopher Alexander aportó al mundo de la arquitectura el libro "The Timeless Way of Building"; donde, proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad. Aquí escribió la siguiente frase "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez."

Sin embargo, no fue hasta principios de los 90's cuando los patrones de diseño de software tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro "Design Patterns", escrito por GoF (Gang of Four) y compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. El libro plantea 23 patrones de diseño comunes (Factory, Singleton, Adapter, Facade, Command, Iterator entre otros)

Hoy en día existe una variedad de patrones que permite uniformizar y diseñar soluciones estándar de software, y que muy bien pueden ser aplicados en el desarrollo de aplicaciones empresariales utilizando Java.

Los patrones bastante usados en el desarrollo de aplicaciones son MVC (Model View Controller), DAO (Data Access Object) y DTO (Data Transfer Object).

SINGLETON

CONTEXTO

Existen escenarios donde la aplicación sólo necesita emplear una única instancia (objeto) de una clase, y esta debe ser accesible desde los diferentes componentes del sistema.

Las clases que cumplen estas características suelen tener alguna de las siguientes funciones:

- ✓ Controlar el acceso a un recurso que por su naturaleza no admite el acceso concurrente (impresora, socket, fichero).
- ✓ Obtener referencias de clases que actúan como servidoras de recursos (pool de conexiones).

- ✓ Ejecutar código carente de estado (si el estado no cambia solo necesitamos una instancia).
- ✓ Mantener un estado que debe ser globalmente único.

Por ejemplo, un sistema de log se ajusta a estas características porque:

- ✓ Controla el acceso al recipiente de la información de log.
- ✓ Toda la información de log se envía a través de un punto único.
- ✓ Su único estado es la configuración, que es global y no suele variar.

PROBLEMA

Necesitamos crear una instancia de una clase que sea accesible globalmente, y que sea única.

SOLUCIÓN

El patrón Singleton proporciona la siguiente solución:

- ✓ Hacer que la clase provea una instancia de sí misma.
- ✓ Permitir que otros objetos obtengan esa instancia, mediante la llamada a un método de la clase.
- ✓ Declarar el constructor como privado, para evitar la creación de otros objetos.

El diagrama UML correspondiente es muy simple y se muestra en la Figura 19.

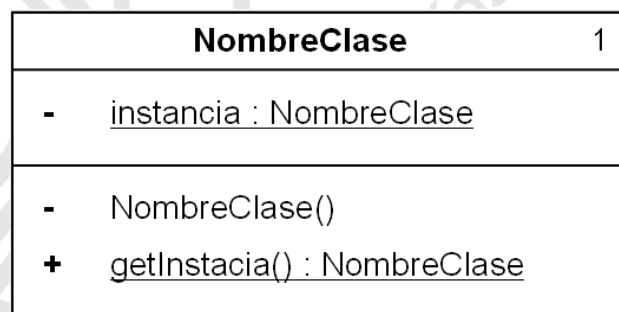


Diagrama de Clases del Patrón Singleton.

Este diagrama UML muestra cómo se implementa el patrón Singleton en una clase. Contiene una propiedad estática (el subrayado indica que es un método de clase), y que el método `getInstancia()` retorna un objeto de la misma clase.

Según la notación UML el número 1 en la esquina superior derecha, indica que solamente habrá una instancia de esta clase. El signo "-" en el constructor, lo señala como privado. Esto consigue que nadie aparte de la propia clase, pueda crear una instancia.

A continuación, tenemos un ejemplo de la implementación del patrón Singleton en una clase de nombre Demo.

```
public class Demo{

    private static Demo instancia = null;

    public static Demo getInstancia(){

        if( instancia == null ){
            instancia = new Demo();
        }
        return instancia;

    }

    private Demo() {
    }

    . . . . .

}
```

MVC (MODEL VIEW CONTROLLER)

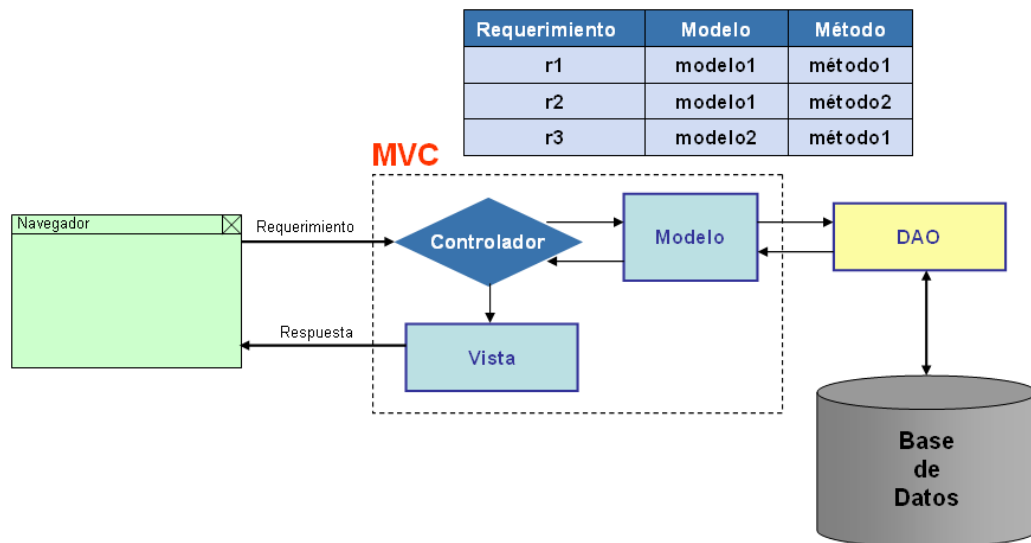
CONTEXTO

Aplicaciones interactivas con interfaces humano-computador cambiantes y flexibles.

PROBLEMA

Es muy frecuente que se solicite cambios a la interfaz de usuario. Los cambios a la interfaz deberían ser fáciles y no tener consecuencias para el núcleo del código de la aplicación.

SOLUCIÓN



Patrón de Diseño MVC – Modelo 2.

El sistema se divide en tres partes o tipos de componentes:

Modelo

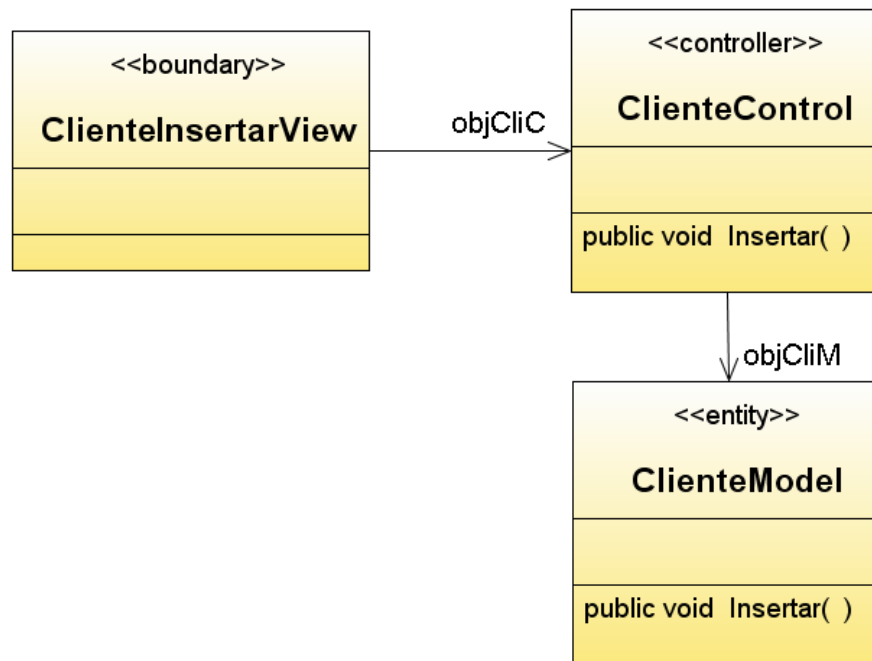
Encapsula los datos y la funcionalidad de la aplicación (lógica que da solución al problema).

Vista

Interfaz de Usuario para el ingreso y presentación de datos generados por el modelo.

Controlador

Es el que controla el flujo de la aplicación y administra quién es el responsable de dar solución a un problema; se comunica con el modelo y la vista.



Modelo del patrón MVC

El usuario interactúa con el modelo solamente a través de controladores.

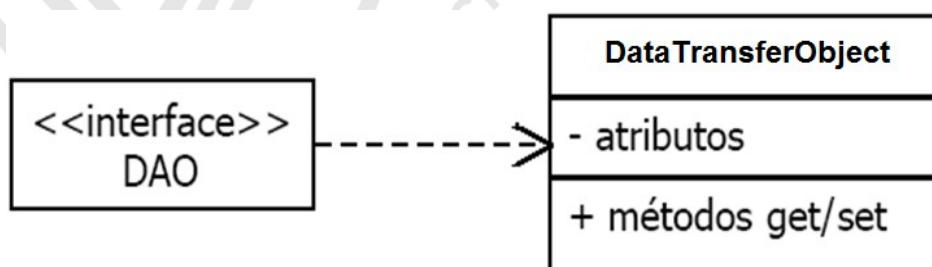
DTO (DATA TRANSFER OBJECT)

CONTEXTO

En general, los componentes de un sistema necesitan intercambiar datos.

PROBLEMA

Se requiere enviar y recuperar un conjunto de datos entre un objeto y otro.



Patrón de Diseño Data Transfer Object - Diagrama de Clases.

SOLUCIÓN

Crear una clase que agrupe un conjunto de datos que se desea transferir y por lo general implemente solo métodos set y get para asignar y recuperar los datos.

ClienteDTO
<pre>private Int CliID private String CliNom private float CliCredito</pre>
<pre>public int getCliID () public void setCliID (int val) public String getCliNom () public void setCliNom (String val) public float getCliCredito () public void setCliCredito (float val)</pre>

Clase Data Transfer Object (DTO) antes llamado Value Object (VO).

DAO (DATA ACCESS OBJECT)

CONTEXTO

El contexto para las fuentes de datos presenta muchas características que hacen difícil tener componentes estándares y escalables. A continuación se enumeran algunas características de este entorno:

- ✓ El acceso a los datos varía dependiendo de la fuente de datos.
- ✓ El acceso al almacenamiento persistente, tal como Bases de Datos, varia bastante dependiendo del tipo de almacenamiento.
- ✓ Tenemos Bases de Datos relacionales, orientadas a objetos y multidimensionales, archivos planos, XML, NoSQL, etc.
- ✓ Diferentes implementaciones de los proveedores.

PROBLEMA

Todo sistema necesita una capa de persistencia de datos, por lo tanto, podemos afirmar que no se tiene un problema, realmente se tienen varios problemas que se describen a continuación:

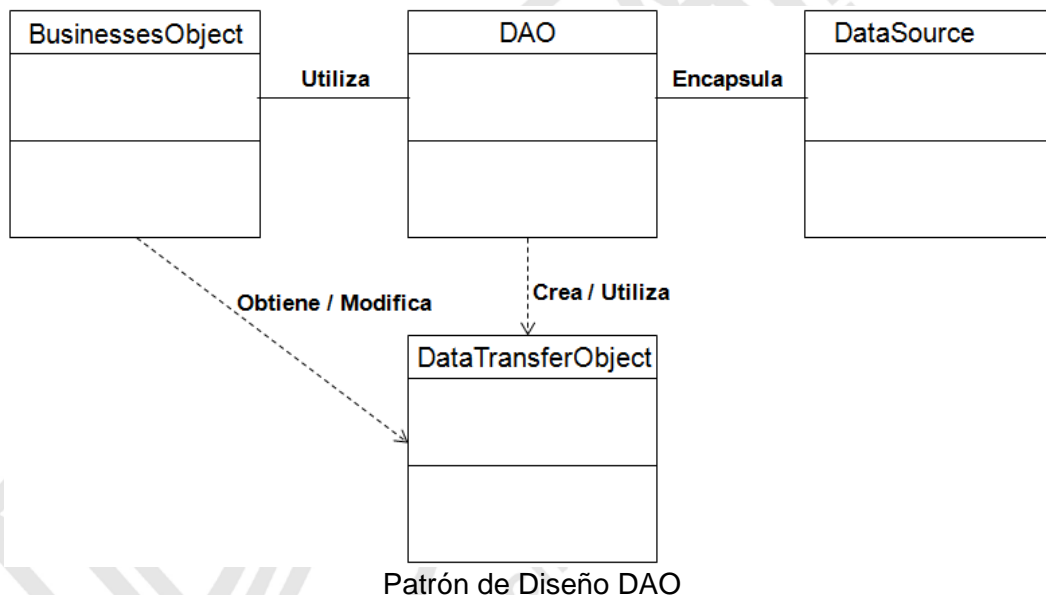
- ✓ Necesidad de acceso a una fuente de datos.
- ✓ Se tiene diferentes API para acceso a mecanismos de almacenamiento persistente.
- ✓ Las API varían entre RDBMS (Sintaxis SQL, formato de datos, etc.).
- ✓ Se tienen API propietarias.

- ✓ Difícil migración/portabilidad entre diferentes fuentes de datos.
- ✓ Dependencias de código.

SOLUCIÓN

La solución la tenemos en el patrón Data Access Object (DAO) para abstraer y encapsular todo el acceso a la fuente de datos, cuyas características se describen a continuación:

- ✓ DAO administra las conexiones con la fuente de datos, recupera y almacena información en la fuente de datos.
- ✓ Independiente del mecanismo de almacenamiento persistente.
- ✓ DAO actúa como adaptador entre los componentes de negocio y la fuente de datos.



La Figura anterior muestra el Diagrama de Clases. El objeto **DataTransferObject** encapsula una unidad de información de la fuente de datos, y sirve para la comunicación de datos entre el **BusinessesObject** y el objeto **DAO**.

Los sistemas tienen la necesidad de persistir datos en algún momento, ya sea serializándolos, guardándolos en una base de datos relacional, o una base de datos orientada a objetos, etc. Para hacer esto, la aplicación interactúa con la base de datos. El "cómo interactúa" NO debe ser asunto de la capa de lógica de negocio de la aplicación, ya que para eso está la capa de lógica de persistencia, que es la encargada de interactuar con la base de datos. Sabiendo esto, podemos decir que DAO es un patrón de diseño utilizado para crear esta capa de lógica de persistencia.

Ahora, imaginemos que hicimos una aplicación para la empresa donde trabajamos. Utilizando como motor de base de datos Oracle. Pero NO tenemos dividida la capa de lógica de negocio de la de capa de lógica de persistencia. Por lo que la interacción con la base de datos se hace directamente desde la capa de lógica de negocio. Nuestra

aplicación consiste en cantidad considerable de clases, y gran parte de ellas interactúan con la base de datos (conectándose a la base de datos, guardando y recuperando datos, etc.).

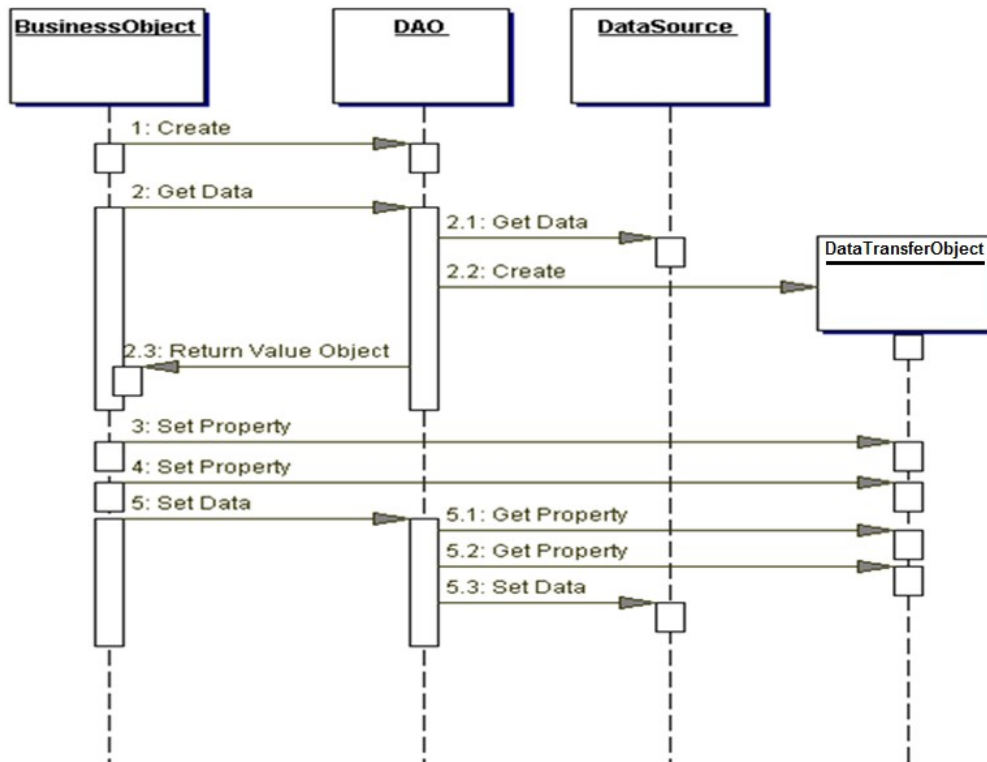
Nuestra aplicación va de maravilla, cuando de pronto, se acerca el jefe del proyecto y nos comenta que por diferentes razones se va a cambiar el motor de la base de datos a MySQL Server. Automáticamente nos imaginamos que hacer ese cambio es sumamente difícil y tomaría mucho tiempo.

Si hubiéramos tenido por separado la capa de lógica de negocio de la capa de lógica de persistencia, habría sido suficiente con modificar la capa de lógica de persistencia para que la aplicación pudiera utilizar el nuevo motor de base de datos, sin tener que modificar nada de la capa de lógica de negocio. Pero como en el ejemplo anterior NO

usamos una capa de lógica persistencia, sino que interactuamos con la base de datos directamente desde la capa de lógica de negocio, entonces vamos a tener que modificar todas las clases, cambiando todas las consultas SQL, la manera de acceder a la base de datos, etc. para adecuarse al nuevo motor de la base de datos.

Bien, ahora que sabemos porque es importante tener separadas las capas de lógica de negocio y de persistencia, vamos a ver cómo utilizar el patrón de diseño DAO para crear nuestra propia capa de lógica de persistencia.

Como se explicó párrafos anteriores, DAO encapsula el acceso a la base de datos. Por lo que cuando la capa de la lógica de negocio necesite interactuar con la fuente de datos, va a hacerlo a través de la API que le ofrece DAO. Generalmente esta API consiste en métodos CRUD (Create, Read, Update y Delete). Por ejemplo, cuando la capa de lógica de negocio necesite guardar un dato en la base de datos, va a llamar a un método create(). Lo que haga este método, es problema del DAO y depende de cómo el DAO implemente el método create(), puede que lo implemente de manera que los datos se almacenen en una base de datos relacional, como puede que lo implemente de manera que los datos se almacenen en ficheros de texto. Lo importante es que la capa de lógica de negocio no tiene por qué saberlo, lo único que sabe es que el método create() va a guardar los datos, así como el método delete() va a eliminarlos, el método update() actualizarlos, etc. Pero no tiene idea de cómo interactúa DAO con la fuente de datos.



Patrón DAO – Diagrama de Secuencia.

DAO consiste básicamente en una clase que es la que interactúa con la fuente de datos. Los métodos de esta clase dependen de la aplicación y de lo que queramos hacer. Pero generalmente se implementan los métodos CRUD para realizar las “4 operaciones básicas” de una base de datos.

Los DTO (Data Transfer Object) o también denominados TO (TransferObject), son utilizados por el DAO para transportar los datos desde la base de datos hacia la capa de la lógica de negocio y viceversa. Por ejemplo, cuando la capa de lógica de negocio llama al método create(), ¿qué es lo que hace DAO? inserta un nuevo registro que la capa de la lógica de negocio le pase como parámetro a través de un DTO.

Podría decirse que un DTO es un objeto simple, que tiene como atributos los datos del modelo, con sus correspondientes métodos getters y setters.

INDICE

FUNDAMENTOS DE JAVA.....	1
INTRODUCCIÓN.....	1
BREVE HISTORIA.....	1
¿QUÉ ES JAVA?	2
¿QUÉ APLICACIONES HAY EN EL SDK?	3
COMPILACIÓN: javac	4
EJECUCIÓN DE APLICACIONES: java	5
CONCEPTOS GENERALES	5
Comentarios	5
Identificadores	5
Palabras reservadas.....	6
Tipos de datos primitivos	6
DECLARACIONES DE VARIABLES	7
CONSTANTES	7
ASIGNACIONES.....	8
STRINGS	8
OPERADORES	9
ESTRUCTURAS DE CONTROL DE FLUJO.....	10
Condicionales	11
Bucles.....	12
EL MÉTODO main.....	12
TRABAJANDO CON ARREGLOS	14
Definición.....	14
Arreglos multidimensionales	15
CADENAS.....	15
Construcción de cadenas.	16
Concatenación	16
Operaciones con cadenas	16
Arreglos de cadenas.....	18
PROGRAMACIÓN ORIENTADA A OBJETOS.....	20
UN NUEVO PARADIGMA	20
OBJETO.....	20
Definición	20
Los objetos realizan operaciones.....	20
Los objetos tienen valores.....	20
Los objetos son una abstracción.....	21
Encapsulamiento.....	21
Relación entre objetos.....	21
Asociación entre objetos	21
Composición de objetos	22
CLASES	22
HERENCIA.....	22
POLIMORFISMO	23

MIEMBROS DE CLASE	24
OPERACIONES CON CLASES.....	24
Definición de clase	24
Paquetes	24
Modificadores de acceso.....	24
Creación de objetos.....	24
La referencia null	25
Asignando referencias.....	25
MÉTODOS.....	26
Definición	26
Argumentos.....	26
Invocando métodos.....	27
ENCAPSULAMIENTO	27
Código de una clase.....	28
Paso de variables a métodos.....	28
SOBRE CARGA DE METODOS	31
SOBRECARGA DE MÉTODOS.....	31
INICIACIÓN DE VARIABLES DE INSTANCIA.....	31
CONSTRUCTORES	32
La referencia: this.....	33
VARIABLES DE CLASE	35
MÉTODOS DE CLASE	35
HERENCIA.....	36
HERENCIA Y POLIMORFISMO	36
Herencia.....	36
La herencia en Java	36
La referencia super.....	37
Métodos.....	37
La referencia super.....	40
Polimorfismo.....	40
El operador instanceof y cast	43
ALCANCE DE LA CLASE.....	45
ATRIBUTOS, MÉTODOS Y CLASES FINAL	45
Variables finales.....	45
Métodos finales.....	45
Clases finales.....	45
EL MÉTODO finalize().....	46
CLASES ABSTRACTAS E INTERFASES.....	47
CLASES ABSTRACTAS	47
Clases abstractas.....	47
Métodos abstractos.....	47
INTERFACES.....	49
LA CLASE Object.....	54
CONVERTIR DATOS PRIMITIVOS EN REFERENCIAS	54

ARREGLOS Y COLECCIONES	57
COLECCIONES	57
Arquitectura	57
Interfaces de colecciones	58
La interface Collection.....	58
La interface List.....	59
La interface Map.....	60
Clases implementadas	61
Definiendo una clase	61
Mostrar los elementos de una colección: ArrayList.....	62
Evitar objetos duplicados: HashSet.....	65
Manejar colecciones ordenadas: TreeSet.....	68
Ordenar y buscar en Colecciones: Collections	71
Ejemplo de clases implementadas: Map.....	75
Ejemplo de HashMap.....	75
Ejemplo de TreeMap.....	77
EXCEPCIONES.....	79
¿QUÉ ES UN ERROR?	80
¿CUÁL ES LA DIFERENCIA?.....	80
CARACTERÍSTICAS DE JAVA.....	80
EXCEPCIONES.....	81
Las excepciones no pueden ignorarse.....	81
Throwable.....	82
Errores.....	82
Excepciones no controladas	82
Excepciones controladas	82
EXCEPCIONES NO CONTROLADAS	83
Capturando una excepción	83
Capturando múltiples excepciones	83
Ejecución del bloque finally.....	84
Como pasar la excepción al método invocado.....	85
Como lanzar una excepción.....	87
COMO CREAR UNA EXCEPCIÓN.....	87
COMO CAPTURAR UNA EXCEPCIÓN Y LANZAR OTRA DIFERENTE	88
BUENAS PRÁCTICAS.....	89
PROGRAMACIÓN EN CAPAS.....	89
Introducción	89
Características de la Programación en Capas.....	89
Capa de Presentación o Frontera.....	90
Capa de Lógica de Negocio o Control	91
Capa de Datos	91
Ventajas y Desventajas	92
Cohesión	92
Acoplamiento.....	92
Ventajas.....	92
Desventajas	93
Conclusiones	93

PRINCIPIOS SOLID	93
S-Responsabilidad simple (Single responsibility)	94
O-Abierto/Cerrado (Open/Closed)	94
L-Sustitucion Liskov (Liskov substitution)	95
I-Segregacion del interface (Interface segregation)	95
D-Inversión de dependencias (Dependency inversion)	95
SIMPLICIDAD DEL CODIGO.....	96
KISS	96
Beneficios del principio Kiss	96
YAGNI	97
Navaja de Occam	97
Simple no significa ignorar la complejidad real	97
Reflexión cuidadosa	97
PRINCIPIO DRY	98
¿Cómo surge la duplicación?.....	99
Un ejemplo.	99
PATRONES DE SOFTWARE.....	102
INTRODUCCIÓN.....	102
SINGLETON.....	102
Contexto	102
Problema	103
Solución.....	103
MVC (MODEL VIEW CONTROLLER)	104
Contexto	104
Problema	104
Solución.....	105
DTO (DATA TRANSFER OBJECT).....	106
Contexto	106
Problema	106
Solución.....	106
DAO (DATA ACCESS OBJECT).....	107
Contexto	107
Problema	107
Solución.....	108