Tecnoforby SNGULAR

Java

Tecnofor by SNGULAR

Temario:

- Introducción a Java
- Programación Orientada a Objetos
- Colecciones y Genéricos
- Streams y Lambdas
- Acceso a bases de datos
- Patrones de diseño: Singleton y Factory Builder
- Spring Boot
- JSON y XML
- Servicios REST

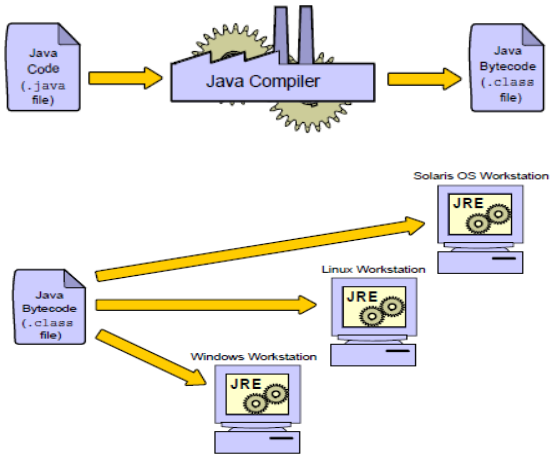
Tecnofor by SNGULAR

Introducción a Java

C/ Manuel Tovar, 31 28034-Madrid

Lenguaje Java

- El lenguaje de programación Java es un lenguaje orientado a objetos.
- Lenguaje independiente de la plataforma



JRE Y JDK

- JRE (Java Runtime Environment); Sería necesario únicamente para ejecutar una aplicación. Esta sería la versión que se deben descargar los clientes, usuarios finales de nuestra aplicación. Incluye únicamente la JVM y un conjunto de librerías para poder ejecutar.

- JDK (Java Development Kit); Estos son los recursos que necesitamos los desarrolladores ya que incluye lo siguiente:
    - JRE
    - Compilador de java
    - Documentación del API (todas las librerías de Java)
    - Otras utilidades por ejemplo para generar archivos .jar, crear documentación, efectuar un debug.
    - También incluye ejemplos de programas.

Ediciones JAVA

- JSE (Java Standard Edition); Es la edición más básica de Java pero no menos importante. Recoge los fundamentos básicos del lenguaje pero solo nos permite desarrollar aplicaciones locales, aplicaciones escritorio y applets (aplicaciones que se ejecutan en el navegador del cliente).

- JEE (Java Enterprise Edition); Esta edición es la más completa de todas. Gracias a ella podremos desarrollar aplicaciones empresariales tales como aplicaciones web, aplicaciones eCommerce, aplicaciones eBusiness, ...etc.

- JME (Java Micro Edition); Con esta edición podremos desarrollar aplicaciones para micro dispositivos tales como teléfonos móviles, PDAs, sistemas de navegación, ...etc.

Variables

- Una variable sirve para identificar mediante un nombre a un espacio de memoria en el que se sitúa un dato antes de ser utilizado por el procesador.

- En Java hay un conjunto de variables convencionales para gestionar datos numéricos, booleanos o de tipo carácter, a las que se denominan variables primitivas, y un conjunto especial de variables que identifican objetos, que se conocen como variables de referencia.

Tipos de variables .-

- Numéricas:
  - Enteros
    - byte; –128 a 127 (byte)
    - short; -32768 a 32767
    - int; -2^{31} a $2^{31}-1$ (int)
    - long; -2^{63} a $2^{63}-1$ L
  - Reales
    - float; +-3,4 x 10^{38} a +-1,4 x 10-45 F(float)
    - double; +-1,8 x 10^{308} a +-4,9 x 10-324

Tipos de variables

- Booleanas:
  - boolean; true o false (sin comillas)

- Carácter:
  - char; un solo carácter. (el dato se introduce entre comillas simples). Se usa también para caracteres de control.
  - String; un conjunto de caracteres. (la “S” debe ser mayúscula), (el dato se introduce entre comillas dobles)

Caracteres de control

'\n'	Línea siguiente
'\t'	Tabulador
'\b'	Retroceso
'\a'	Alarma (beep)
'\r'	Retorno de carro
'\''	Comillas simples
'\\'	Contrabarra (Backslash)
'\"'	Dobles comillas
'\xxx'	Carácter en octal
'\0'	Carácter nulo (null)
'\uxxxx'	Carácter en hexadecimal Unicode

Nombre de variables

- Para otorgar nombre a una variable hay que seguir una serie de normas:
  - El primer carácter del nombre debe ser una letra mayúscula o minúscula, “_” o “$”.
  - No pueden utilizarse como nombres de variables las palabras reservadas de JAVA.
  - Los nombres deben ser continuos, es decir, sin espacios en blanco.
  - Los identificadores de variables son sensibles a las mayúsculas y a las minúsculas.

Palabras reservadas

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

Crear una variable

- Declarar variables primitivas:

  tipo_de_primitiva nombre_variable;

  ejemplo: int numero;

- Declarar variables de referencia a objeto:

  Nombre_de_clase nombre_objeto;

  ejemplo: String nombre;

Crear una variable

- Podemos declarar más de una variable al mismo tiempo, siempre y cuando sean del mismo tipo

```
int num1, num2, num3, num4;
```

- También se puede declarar e inicializar más de una variable a la vez.

```
int num1 = 3, num2 = 5, num3 = 7, num4 = 8;
```

Comentarios

- Varias líneas: /* */

  Ejemplo:

```
/* Este es un ejemplo
de un comentario
que ocupa varias líneas */
```

- Una sola línea: //

  Ejemplo: // Este es de una sola línea

Operadores

- Aritméticos
- Relaciónales
- Lógicos
- Asignación
- Prioridad

Operadores Aritméticos

*	(multiplicación)
/	(división)
%	(resto de la división)
+	(suma)
-	(resta)
++	(incremento)
--	(decremento)
-	(cambio de signo)

Operadores Relacionales

>	(mayor que)
>=	(mayor o igual que)
<	(menor que)
<=	(menor o igual que)
==	(igual que)
!=	(distinto de)

Operadores Lógicos

&&	(AND)
||	(OR)
!	(negación)

Operadores de Asignación

=	(asignación normal)
+=	(con incremento)
-=	(con decremento)
/=	(con división)
%=	(con resto)

Prioridad de los operadores

Expresión:	() []
Unitarios:	- ! ++ --
Multiplicativos:	* / %
Aditivos:	+ -
Relacionales:	< <= > >= == !=
AND lógico:	&&
OR lógico:	||
Condicionales:	?:
Asignación:	= *= /= %= += -=

Fundamentos String

- Un objeto de la clase String en una cadena de caracteres inmutable, no se pueden modificar
- Los métodos que operan con String devuelven una copia de la cadena modificada

- Se pueden crear:

```
String n1=new String("mi cadena");
```

- O también:

```
String n1="mi cadena";
```

En este caso puede reutilizar cadenas de un pool

Métodos String

- int length(). Devuelve la longitud de la cadena
- String toLowerCase(), toUpperCase(). Devuelven la cadena convertida a minúsculas y mayúsculas, respectivamente

```
String n1="cadena";
System.out.println(n1.toUpperCase()); //muestra: CADENA
System.out.println(n1); //muestra: cadena, no ha cambiado
```

- String substring(int a, int b). Devuelve un trozo de cadena comprendido entre las posiciones a y b-1

```
String n1="esto es un texto";
System.out.println(n1.substring(3,9)); //muestra: o es u
```

Métodos String

- char charAt(int pos). Devuelve el carácter que ocupa la posición indicada

```
String n1="esto es un texto";
System.out.println(n1.charAt(0)); //muestra: e
System.out.println(n1.charAt(20)); //StringIndexOutOfBoundsException
```

- int indexOf(String cad). Devuelve la posición de la cadena parámetro. Si no existe, devuelve -1

```
String n1="esto es un texto";
System.out.println(n1.indexOf("un")); //muestra: 8
```

- String replace(CharSequence c1, CharSequence c2). Devuelve la cadena resultante de reemplazar la subcadena c1 por c2.

```
String n1="esto es un texto";
System.out.println(n1.replace("es","de")); //muestra: deto de un texto
```

Métodos String

- boolean startsWith(String s), endsWith(String s). Indica si la cadena empieza o termina, respectivamente, por el texto recibido:

```
String n1="esto es un texto";
System.out.println(n1.endsWith("to")); //muestra: true
System.out.println(n1.startsWith("eso")); //muestra: false
```

- String trim(). Devuelve la cadena resultante de eliminar espacios al principio y al final de la misma

```
String n1="  cade prueba nueva  ";
System.out.println(n1.trim().length()); //muestra: 17
```

- String concat(String s). Mismo efecto que aplicar el operador +
- boolean isEmpty(). Devuelve true si es una cadena vacía. Equivale:

```
cad.equals("")
```

if ... else

- Permite tomar decisiones en función de unas determinadas condiciones que se impongan para ejecutar uno u otro código según lo que indiquen dichas condiciones. Su sintaxis es la siguiente:

```
if (condición)
{
        código si cierto;
}
else
{
        código si falso;
}
```

- Comprueba la validez o falsedad de una condición. Si esta condición es verdadera se ejecuta el código contenido entre las primeras llaves. Si es falsa, se interpreta el código tras las segundas llaves.

if ... else

- La cláusula else no es obligatoria. Si la condición fuese falsa no se llevaría a cabo ninguna acción. Además si no ponemos else y el código de las primeras llaves es tan solo de una línea podemos utilizar este código:

```
if (condición) sentencia;
```

if ... else abreviado

- Con los símbolos ? y : podemos construir un condicional de forma abreviada de la siguiente manera:

condición ? sentencia_v : sentencia_f;

ejemplo: a = (c < d) ? c*2 : 27 ;

es lo mismo que:

```
if (c<d) {
        a=c*2;
} else {
        a=27;
}
```

if ... else anidados

```
if (condición1) {
        código si cierto;
} else if (condición2) {
        código si cierto;
} else if (condición3) {
        código si cierto;
} else {
        código si falso;
}
```

switch ... case

- Durante la ejecución de un programa puede darse el caso de necesitar tomar una decisión a partir de múltiples posibilidades, pero donde sólo una de ellas va a ser posible en cada ocasión. Su sintaxis es:

```
switch (expresión)
{
        case valor1_expresion:
                codigo1;
                break;
        case valor2 _expresion :
                codigo 2;
                break;
        default:
                codigo por omisión;
}
```

La expresión debe ser de tipo **char, byte, short, int o String**.

Bucle for

- Un bucle es aquella operación o conjunto de operaciones que se repite cierto número de veces hasta llegar a un estado que le obliga a detenerse, permitiendo que continúe la ejecución normal hacia delante del código. A las operaciones que se repiten en cada vuelta se las denomina conjuntamente Cuerpo del bucle.

- Cuando se sabe de antemano el número de veces que se va a repetir el cuerpo del bucle antes de detenerse, se dice que el bucle es determinado. Si no se pude saber de antemano cuándo se detendrá, se dice que es un bucle indeterminado.

Bucle for

- La sintaxis es la siguiente:

```
for (contador=valor_inicial; condición del bucle; incremento)
{
        cuerpo del bucle;
}
```

- La condición de bucle es una expresión que indica si el cuerpo del bucle se debe ejecutar una vez más o no. Mientras esta condición sea verdadera se repetirá el bucle. En el momento en que sea false se detendrá. Si inicialmente ya es falsa, el cuerpo del bucle no se ejecutará nunca.

- Los bucles for se pueden anidar.

Bucle while

- Itera mientras la condición de final se cumpla. Comprueba la condición al principio. Si es falsa no se ejecuta

```
while (condición) {
        cuerpo del bucle;
}
```

Bucle do ... while

- Itera mientras la condición de final se cumpla. Comprueba después, al menos se ejecuta una vez.

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

Break y Continue

- Continue:
  - Hace que se termine la iteración actual del bucle y que se pase a la siguiente. Esta sentencia sólo se puede utilizar dentro de un bucle si no es así genera error de compilación.

- Break:
  - Interrumpe del bucle definitivamente. Se puede utilizar en un bucle o en un switch.

Arrays o Matrices

- Un array es un conjunto de elementos colocados de forma adyacente en la memoria de manera que nos podemos referir a ellos con un solo nombre común mientras que, por otro lado, no se pierde la independencia de los mismos.

- Es un modo de agrupar datos para que se guarden ordenadamente y sean más cómodos de manejar y gestionar. Todo elemento pertenece a un array lleva asociado un índice de forma unívoca: siempre se puede acceder a un elemento determinado en cualquier array si se conoce su índice.

Declaración de arrays

- En Java todos los arrays deben ser declarados y construidos antes de ser usados.

```
int vector[];                  //una dimensión
int [] vector;
String alumnos[][];                        //dos dimensiones
String [][] alumnos;
```

- Cuando se define un array de referencias a objetos, es decir, de objetos, estos toman por defecto null hasta que se le asigne una referencia a un objeto existente. En el caso de variables primitivas toman el valor cero si son numéricas y false, si son de tipo boolean.

Definición de arrays

- Un array una vez declarado debe ser definido, proceso que también se puede identificar como construcción, en el cual se reserva el espacio necesario para almacenar los datos o los objetos que se va a contener.

```
vector = new int[30];
nombres = new String[var_nombres];
alumnos = new String [3][2];
```

Definición de arrays

- Un array de dos índices, puede tener en cada fila componentes de distinta longitud, de tal forma que se pueden construir, por ejemplo, matrices triangulares:

```
int matriz [][];
matriz = new int [3][];
        matriz[0] = new int [2];
        matriz[1] = new int [4];
        matriz[2] = new int [5];
```

Declaración y definición

- Es posible declarar y definir el array en una misma instrucción.

```
int vector[] = new int[30];
boolean resultados[]=new boolean[10];
String nombres[]=new String[var_nombres];
String alumnos[][]=new String[3][2];
```

Declaración, definición e inicialización

- Es posible hacer las tres cosas en una misma sentencia:

```
int vector[] = {2,3,4,5,6};
String nombres[] = {“Juan”,”Lola”,”Luis”,”Ana”};
String alumnos[][]={{“alumn00”,”alumn01”,”alumn02”},
                    {“alumn10”,”alumn11”,”alumn12”}};
```

Longitud de un array

- Para investigar la longitud de un array utilizamos el atributo **length** del array.

- Con dicho atributo obtenemos el máximo de elementos posibles y no el número de elementos ocupados.

```
int nFilas = matriz.length;
int nColumnas1 = matriz[0].length;
```

Programación Orientada a Objetos

C/ Manuel Tovar, 31 28034-Madrid

Clases y objetos

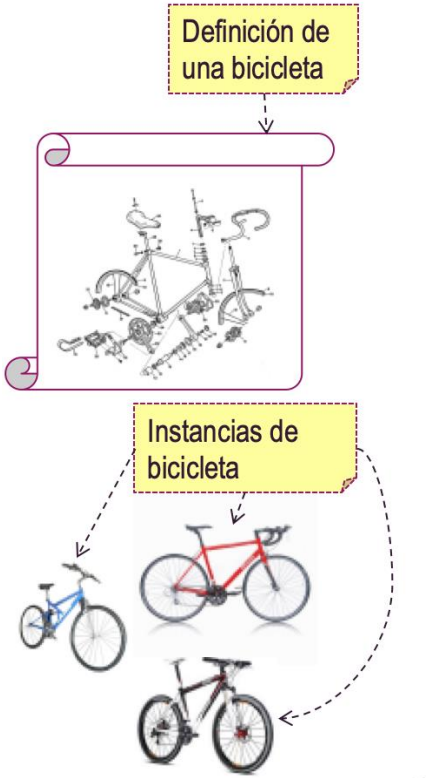
Clase:

Una clase es la definición de las características concretas de una determinada tipología de objetos. Es decir, cuáles son los atributos y métodos de los que van a disponer todos los objetos de ese tipo. Equivale a un tipo de dato.

Objeto:

Un objeto es una agregado de atributos (información) y métodos (comportamiento), creado según la definición de una clase.

A un objeto concreto que pertenece a una clase se le suele llamar instancia.



Declaración de clases

El archivo en el cual declaramos la clase se debe seguir este orden.

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

Ejemplo

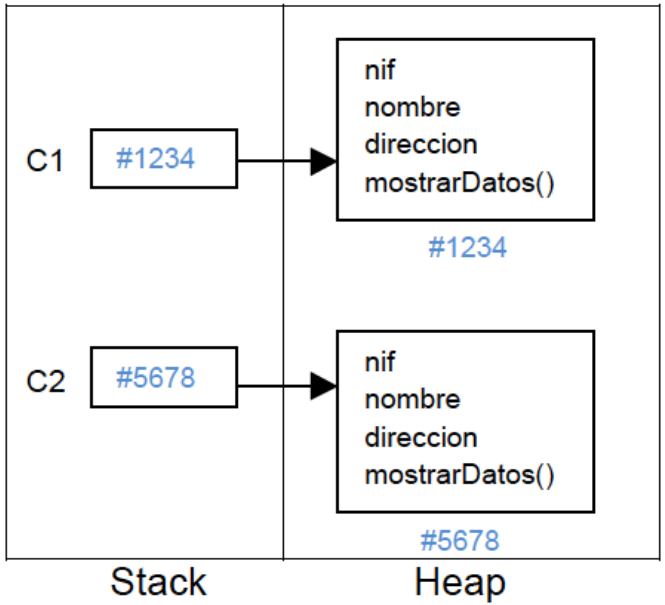
```java
// Declaracion del paquete
package ejemplo1clasesyobjetos;

// importaciones si fuesen necesarias

// Declaracion de la clase
public class Cliente {

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

    // constructores
    public Cliente() {
    }

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

Creación de objetos

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

Ejemplo: new Cliente();

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



Características de la P.O.O

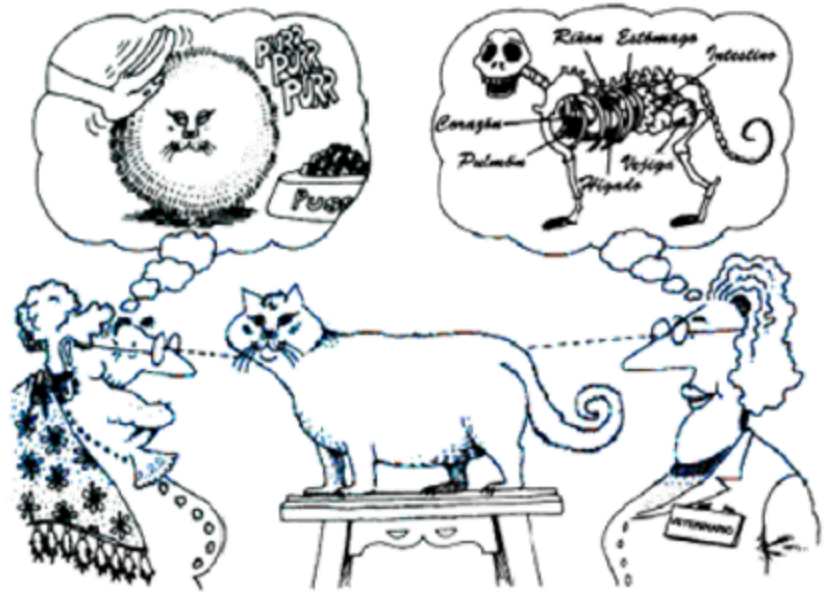
- Abstracción
- Encapsulamiento
- Herencia
- Polimorfismo

Abstracción

- Es la acción de separar las características esenciales de algo sin incluir detalles irrelevantes.

- Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

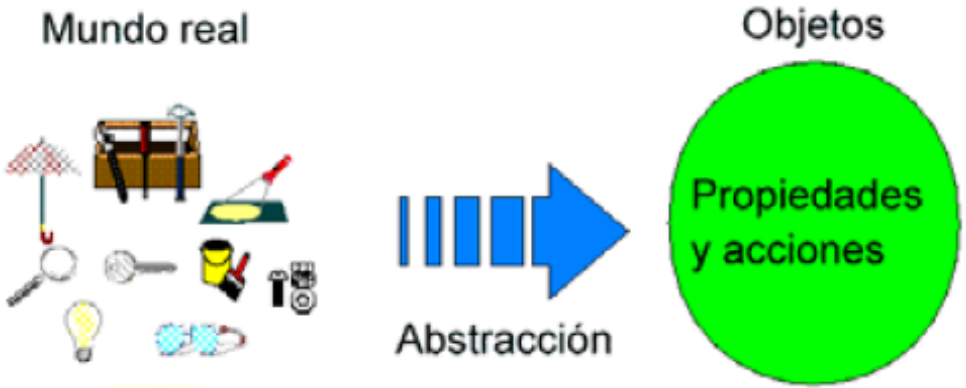
Abstracción



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

Abstracción

- Nos fijaremos en el comportamiento de los objetos para definir las acciones u operaciones que son capaces de realizar.
- Ejemplo de objetos: una factura ,un usuario, un albarán…
- A partir de objetos que tienen unas propiedades y acciones comunes se deben abstraer las clases. En las clases se definen las propiedades y operaciones comunes de los objetos.



Encapsulamiento

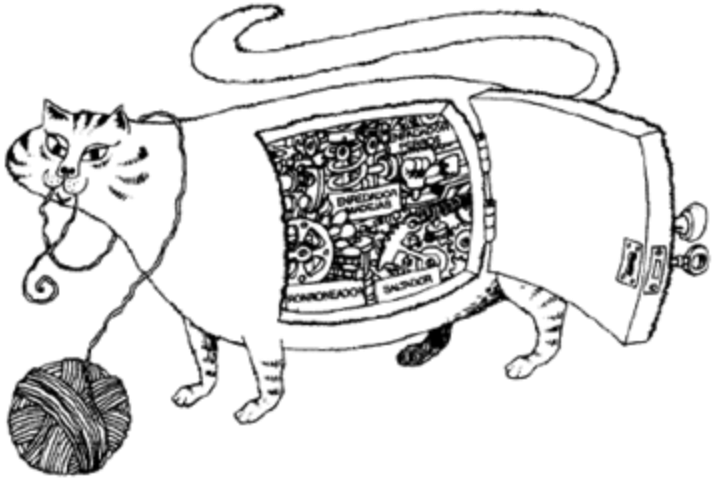
- Ocultación de información

- Es la acción de incluir dentro de un objeto todo lo que necesita, de tal forma que ningún otro objeto necesita conocer nunca su estructura interna.

- El objeto se vería como una caja negra, en la que se ha metido de alguna manera toda la información relacionada con dicho objeto.

Encapsulamiento

53



El encapsulamiento oculta los detalles de la implementación de un objeto.

Encapsulación - Ocultación de datos

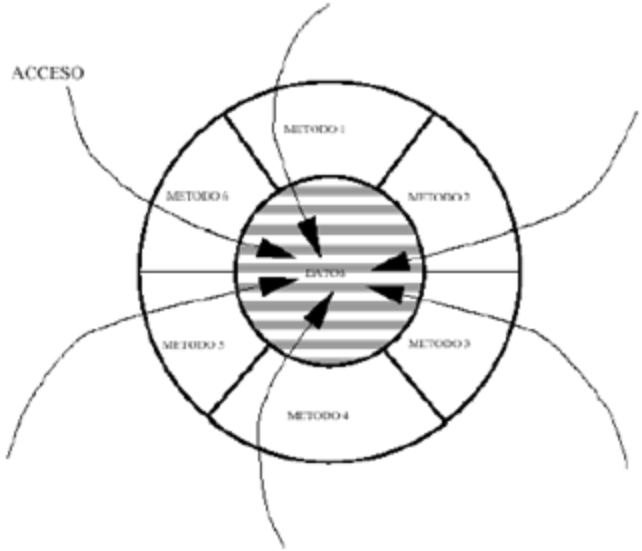
- La palabra reservada private permite una accesibilidad total desde cualquier método de la clase, pero no desde fuera de ésta.

```
public class MyDate
{
     private int day, month, year;

     public void tomorrow ()
     {
        this.day = this.day + 1;
     }
}
```

```
public class DataUser
 {
     public static void main (String args[])
     {
        MyDate mydate = new Date();
        mydate.day = 21; //Incorrecto
     }
}
```

Encapsulamiento

- Los datos o propiedades son privados y se accede a ellos a través de los métodos



Encapsulamiento

- Como los datos son inaccesibles, la única manera de leerlos o escribirlos es a través de los métodos de la clase. Todo atributo privado va a tener asociado dos métodos públicos (get y set) a través de los cuales vamos a poder modificar el valor del atributo (set) o recuperar su valor (get).

	Sintaxis	Sintaxis
GET	public (tipo de retorno) **get**Atributo() { return expresión; }	public int **get**Day() { return day; }
SET	public void **set**Atributo(argumento) { atributo=argumento; }	public void **set**Day(int hoy) { **//aquí lógica de verificación** day=hoy; }

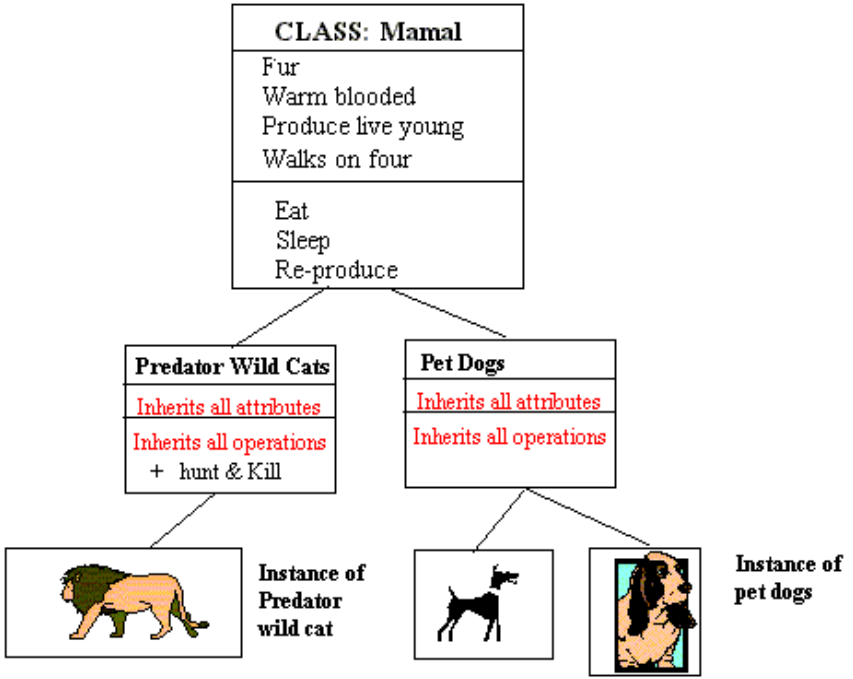
Encapsulamiento

- Esto proporciona consistencia y calidad.
- Si creamos una clase que permite acceso libre podríamos tener este tipo de errores:

```
MyDate d = new MyDate();
d.day = 32;                    Incluir días no válidos
d.month = 2;
d.day = 30;                    Posible pero incorrecto
d.month = d.month +1;          Salir del rango de meses.
```

Herencia

- La herencia es un mecanismo para compartir automáticamente métodos y atributos entre clases y subclases.
- Esta característica nos permite la reutilización del código.
- Una clase derivada puede añadir nuevos atributos y métodos y/o redefinir las variables y métodos heredados



Polimorfismo

- Es un mecanismo que permite a un método realizar distintas acciones al ser aplicado sobre distintos tipos de objetos que son instancias de una misma jerarquía de clases.
- El polimorfismo significa “ muchas formas ”
- El polimorfismo se realiza en tiempo de ejecución gracias a la ligadura dinámica.
- No se debe confundir polimorfismo con sobrecarga:
  - Sobrecarga se resuelve en tiempo de compilación, dado que los métodos se deben diferenciar en el tipo o en el número de parámetros.
  - Polimorfismo se resuelve en tiempo de ejecución, todos los métodos tienen los mismos parámetros, las acciones cambian en función del objeto al que se le aplica.

Polimorfismo

- El polimorfismo es una habilidad de tener varias formas; por ejemplo, la clase Jefe tiene acceso a los métodos de la clase Empleado.
- Un objeto tiene sólo un forma.
- Una variable tiene muchas formas, puede apuntar a un objeto de diferentes maneras.
- En Java hay una clase que es la clase padre de todas las demás: java.lang.Object.
- Un método de esta clase (por ejemplo: toString() que convierte cualquier elemento de Java a cadena de caracteres), puede ser utilizada por todos.

Interfaces

- Es una colección de:
  - declaraciones de métodos (sin definirlos)
  - declaraciones de constantes.

- Las clases que implementen (**implements**) el interface han de definir obligatoriamente las funciones declaradas en él.

- Una clase puede implementar uno ó más interfaces.

Interfaces

- Una interfaz se puede comparar a una clase abstracta que tiene todos sus métodos abstractos, ya que en una interfaz no puede existir ningún método con código.
- Una interfaz puede heredar de otra.
- Una clase puede implementar varias interfaces.

Interfaces

- Sintaxis para crearla:

```
public interface nombreInterfaz [extends interface1]
```

- Sintaxis para implementarla:

```
public class nombreClase implements interf1, interf2, ...
```

Colecciones y Genericos

C/ Manuel Tovar, 31 28034-Madrid

API Collections

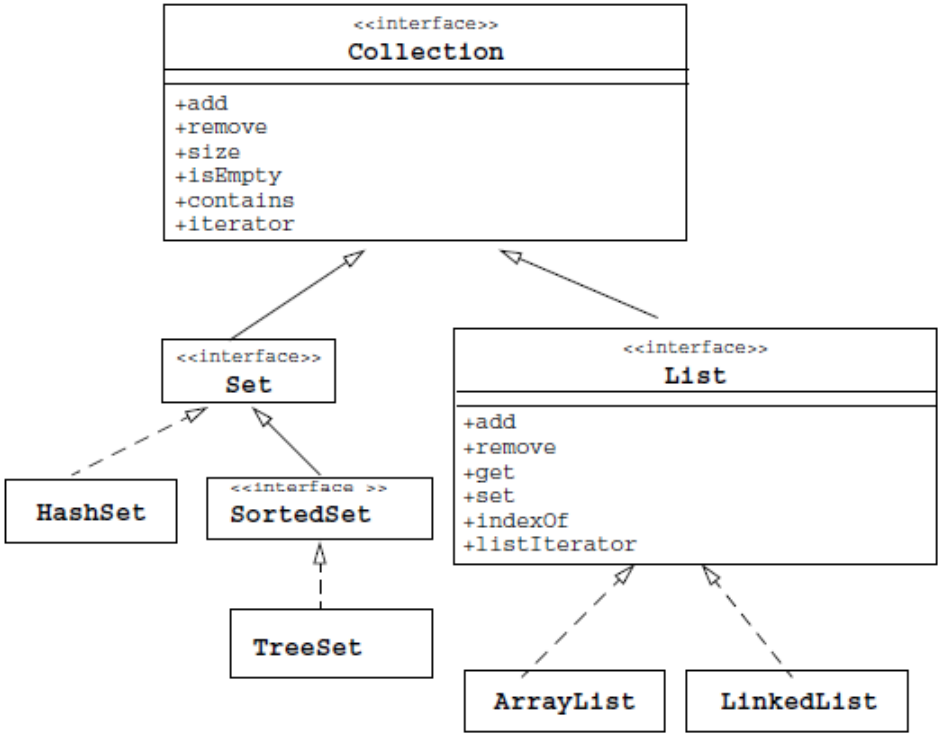
- El API Collections contiene interfaces que permiten agrupar objetos en una de las siguientes colecciones:

- **Collection**; Un grupo de objetos denominados elementos, la implementación determina si guardan un orden específico y si se permiten elementos duplicados.

- **Set**; Es un tipo de colección donde no se garantiza que se conserve el orden de entrada de los elementos. Tampoco permite elementos duplicados.

- **List** – En esta colección si se garantiza el orden de entrada y si que se permiten los elementos duplicados.

API Collections



Principales Colecciones

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

Set

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

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

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

List

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

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

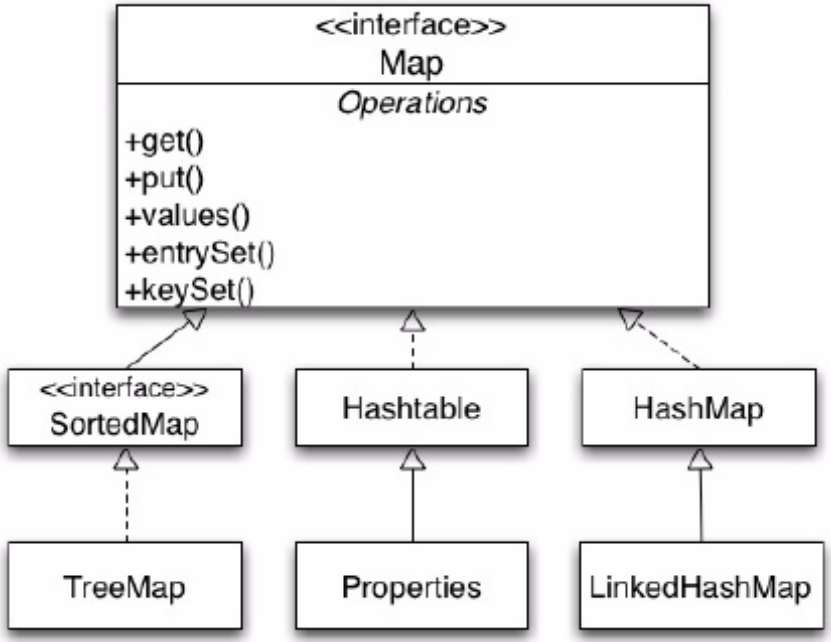
```
[uno, segundo, 4, 3.15, segundo]
```

Map

- Un mapa no se considera una colección ya que no hereda de la interface Collection.
- Los elementos de un mapa se forman como clave-valor. Además tienen las siguientes restricciones:
    - Las claves duplicadas no están permitidas.
    - Una clave solo puede referenciar un valor, no varios.

- La interface Map define una serie de métodos para manipular el mapa, los más interesantes son:
    - entrySet; Devuelve una colección de tipo Set con todos los elementos (clavevalor).
    - keySet; Devuelve una colección de tipo Set con todas las claves del mapa.
    - values; Devuelve un objeto de tipo Collection con todos los valores del mapa.

Map


<<interface>>
Map
Operations
+get()
+put()
+values()
+entrySet()
+keySet()
<<interface>>
SortedMap
Hashtable
HashMap
TreeMap
Properties
LinkedHashMap

Map

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

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

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

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

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

Genéricos

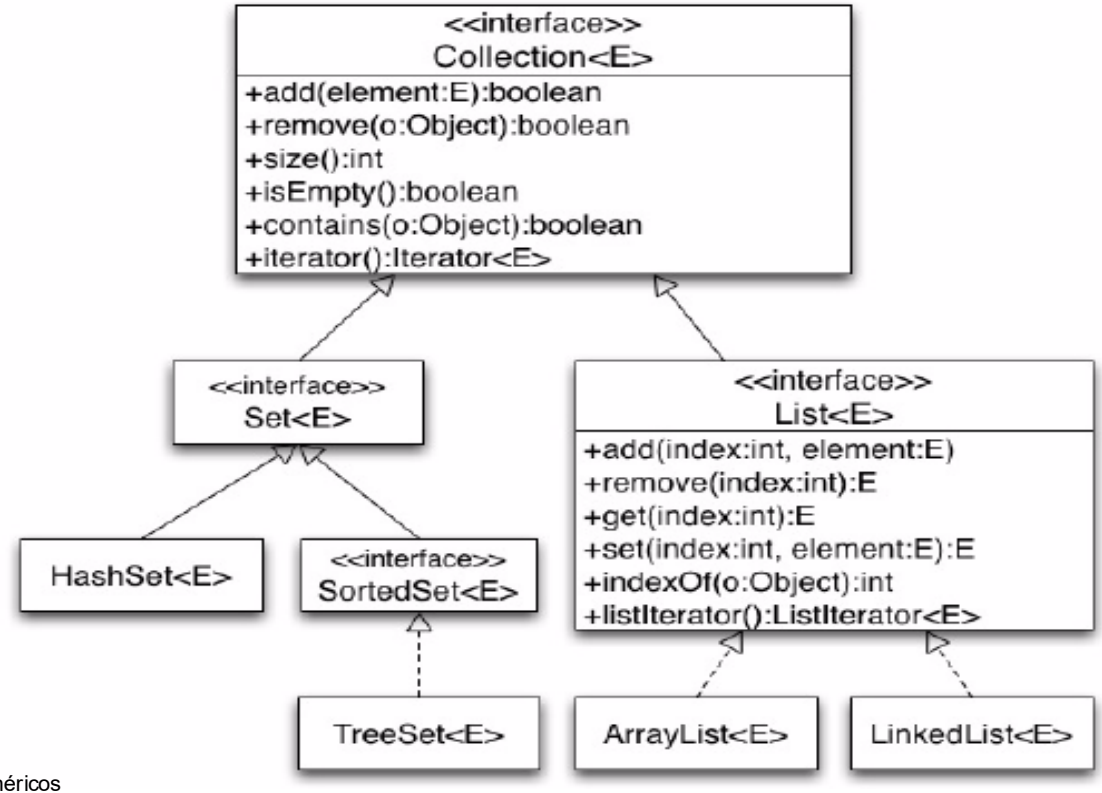
- Utilizar un tipo genérico a la hora de crear la colección nos permite:
- Eliminar la necesidad de casting
- Controlar en tiempo de compilación que todos los elementos son del tipo genérico.

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

Genéricos

Category	Non Generic Class	Generic Class
Class declaration	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Constructor declaration	`public ArrayList (int capacity);`	`public ArrayList<E> (int capacity);`
Method declaration	`public void add((Object o)` `public Object get(int index)`	`public void add(E o)` `public E get(int index)`
Variable declaration examples	`ArrayList list1;` `ArrayList list2;`	`ArrayList <String> list1;` `ArrayList <Date> list2;`
Instance declaration examples	`list1 = new ArrayList(10);` `list2 = new ArrayList(10);`	`list1= new ArrayList<String> (10);` `list2= new ArrayList<Date> (10);`

Genéricos


<<interface>>
Collection<E>
+add(element:E):boolean
+remove(o:Object):boolean
+size():int
+isEmpty():boolean
+contains(o:Object):boolean
+iterator():Iterator<E>
<<interface>>
Set<E>
<<interface>>
List<E>
+add(index:int, element:E)
+remove(index:int):E
+get(index:int):E
+set(index:int, element:E):E
+indexOf(o:Object):int
+listIterator():ListIterator<E>
HashSet<E>
<<interface>>
SortedSet<E>
TreeSet<E>
ArrayList<E>
LinkedList<E>

Streams y Lambdas

C/ Manuel Tovar, 31 28034-Madrid

Visión general de Lambdas

- Una interface funcional es una interfaz que proporciona un único método abstracto

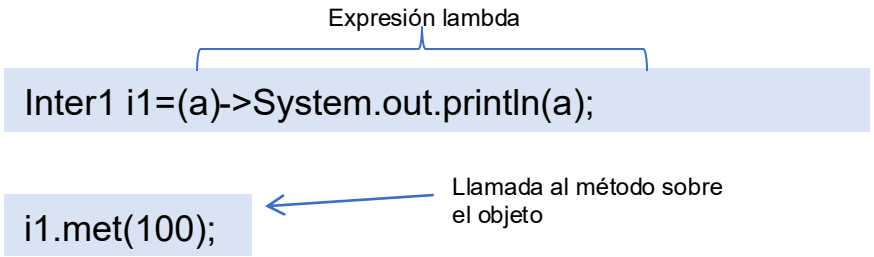
```
public interface Runnable{
    void run();
}
```

```
public interface Inter2{
    boolean process(int n, String pt);
    static void print(){}
}
```

```
public interface Inter1{
    void met(int data);
    default int res(){return 1;}
}
```

Visión general de Lambdas

- Que es una expresión lambda?
- Implementación de una interfaz funcional
- Proporciona el código del único método abstracto de la interfaz, a la vez que genera un objeto que implementa la misma



Visión general de Lambdas

- Una expresión lambda tiene dos partes, la lista de parámetros del método y la implementación:

```
parametros->implementación
```

- Los parámetros pueden indicar o no el tipo
- La lista de parámetros se puede indicar o no entre paréntesis (obligatorio si hay dos o más) y también si se indica el tipo
- En caso de devolver un resultado, la implementación puede omitir la palabra return si consta de una sola instrucción

Streams

- Creación de un stream

- A partir de una colección:

```
ArrayList<Integer> nums=new ArrayList<>();
nums.add(20);nums.add(100);nums.add(8);
Stream<Integer> st=nums.stream();
```

- A partir de un array:

```
String[] cads={“a”,”xy”,”jk”,”mv”};
Stream<String>st= Arrays.stream(cads);
```

- A partir de una serie discreta de datos:

```
Stream<Double> st=Stream.of(2.4, 7.4, 9.1);
```

- A partir de un rango de datos:

```
IntStream stint=IntStream.range(1,10);
IntStream stint2=IntStream.rangeClosed(1,10);
```

Stream de tipos primitivos

Streams

- Tipos de métodos de Stream

- Métodos intermedios. El resultado de su ejecución es un nuevo Stream. Ejemplos: filtrado y transformación de datos, ordenación, etc.
- Métodos finales. Generan un resultado. Pueden ser void o devolver un valor resultado de alguna operación. Ejemplos: calculo (suma, mayor, menor, …), búsquedas, reducción, etc.

Acceso a Bases de Datos

C/ Manuel Tovar, 31 28034-Madrid

Introducción

- JDBC (Java Database Connectivity) está compuesto por un número determinado de clases que permiten a cualquier programa escrito en Java acceder a una base de datos.
- Este conjunto de clases reside en los paquetes: java.sql y javax.sql
- javax.sql añade funciones de servidor como los RowSet, los pools de conexiones o las transacciones.

Conectar la BBDD

- Para poder acceder a una base de datos es necesario establecer una conexión.
- La interfaz Connection es la que se encarga de la conexión con la base de datos. Mediante el método getConnection(), obtenemos un objeto de la clase Connection. Esta clase contiene todos los métodos que nos permiten manipular la base de datos.
- Para cerrar la conexión utilizamos el método close() de la interfaz Connection.

Conectar la BBDD

- Antes de conectarnos con la base de datos hay que registrar el controlador apropiados.

- Java utiliza la clase DriveManager para cargar inicialmente todos los controladores JDBC disponibles y que deseemos utilizar.

- Para cargar un controlador se emplea del metodo forName() de la clase Class que devuelve un objeto Class asociado con la clase que se le pasa como parámetro.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection cnn = DriverManager.getConnection("jdbc:odbc:Libros");
```

Ejemplo

```
import java.sql.*;
public class Main {
    public Main() {
    }
    public static void main(String[] args) {
        String url = "jdbc:odbc:Libros";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection cnn = DriverManager.getConnection(url,"","");
            cnn.close();
            System.out.println("OK: conexión cerrada");
        } catch (Exception e) {
            System.out.println("KO: no se pudo conectar " + e);
        }
    }
}
```

Ejemplo

```
import java.sql.*;
public class Ejemplo_Consulta {
    Connection cnn;
    public Ejemplo_Consulta() {
        String url ="jdbc:odbc:Licencias";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cnn = DriverManager.getConnection(url ,"","");
            acceso();
            cnn.close();
            System.out.println("OK: conexión cerrada");
        } catch (Exception e) {
            System.out.println("KO: no se pudo conectar " + e);
        }
    }
    public void acceso(){
        try{
            String sql= "SELECT * FROM licencias";
            System.out.println("Formato nativo " + cnn.nativeSQL(sql));
            Statement stm = cnn.createStatement();
            ResultSet rs = stm.executeQuery(sql);
            System.out.println("OK acceso sql ");
        } catch (Exception e) {
            System.out.println("error en acceso sql " + e);
        }
    }
    public static void main(String args[]){
        new Ejemplo_Consulta();
    }
}
```

La interfaz Statement

- Al crear una instancia del objeto Statement, podremos realizar sentencias SQL sobre la base de datos.

- Existen dos tipos de sentencias a realizar:
  - Sentencias de modificación (update); Engloban a todos los comandos SQL que no devuelven ningún tipo de resultado como pueden ser los comandos INSERT, UPDATE, DELETE o CREATE.
  - Sentencias de consulta (query); Son sentencias del tipo SELECT (sentencias de consulta) que retornan algún tipo de resultado.

La interfaz Statement

- Para las sentencias “update” la clase Statement nos proporciona el método siguiente que devolverá el número de filas afectadas por la sentencia SQL:

```
public abstract int executeUpdate(String sentenciaSQL)
```

- Para las sentencias “query” la clase Staement utiliza el método siguiente:

```
public abstract ResultSet executeQuery(String sentenciaSQL)
```

PreparedStatement

- Define métodos para trabajar con instrucciones SQL precompiladas, que son más eficientes.
- También se usan para poder utilizar instrucciones SQL parametrizadas. Para parametrizar una instrucción SQL basta con sustituir el parámetro por un signo de interrogación. (Select * from Tabla where Nombre=?). El parámetro se sustituye por el valor mediante el método setString(int lugar,String Valor).

- PreparedStatement no utiliza los métodos de Statement para ejecutar las query's, en su lugar, se pasa la query en el propio constructor.

CallableStatement

- Los objetos de este tipo se crean a través del método de Connection, prepareCall(). Se pasa como argumento de este método la llamada al método almacenado. Los callableStatement pueden ser parametrizados.
- Un método almacenado es una macro registrada en el sistema gestor y que realiza operaciones de cualquier tipo.

Obtener los resultados de la consulta

- La interfaz ResultSet contendrá las filas o tuplas obtenidas mediante la ejecución de la sentencia de tipo “query”. Cada una de esas filas obtenidas se divide en columnas.

- La interfaz ResultSet contiene un puntero que está apuntando a la fila actual.

- Si deseamos utilizar los métodos de posicionamiento debemos abrir la consulta con un tipo distinto a TYPE_FORWARD_ONLY.

Métodos de posicionamiento por filas

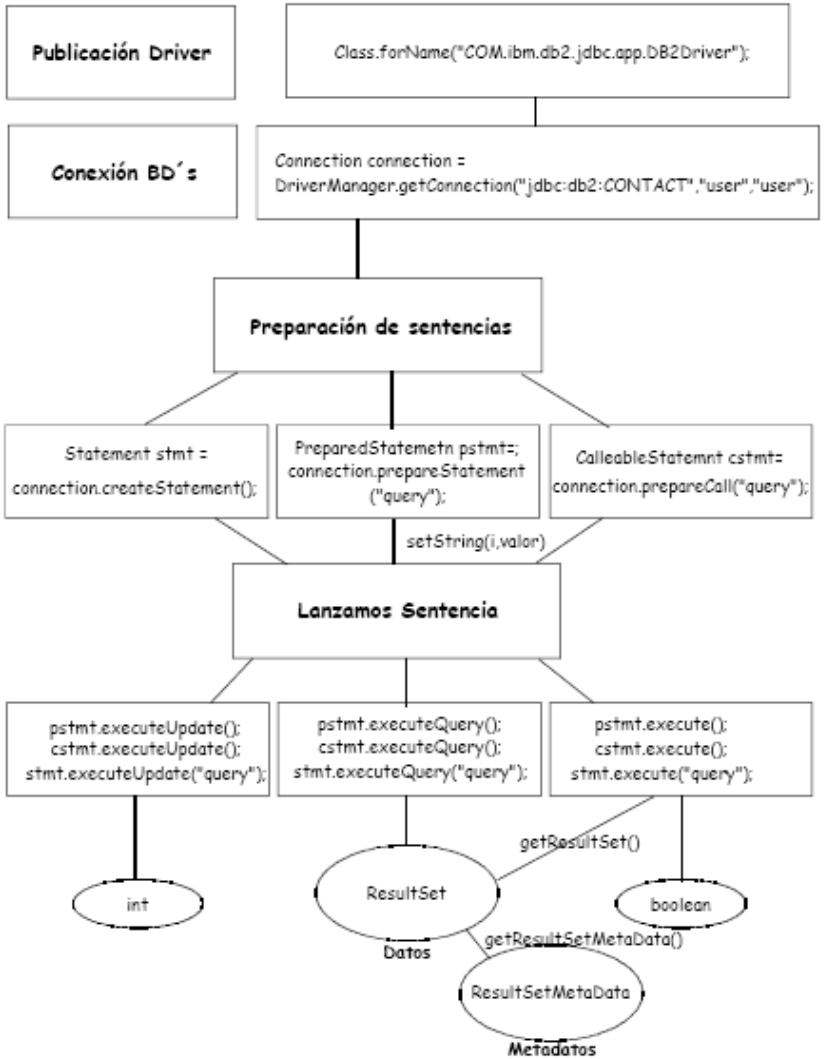
- **absolute(int fila);** Se sitúa en la fila pasada como parámetro.
- **afterLast();** Sitúa el cursor justo detrás de la última fila.
- **beforeFirst();** Sitúa el cursor justo antes de la primera fila.
- **isLast();** Comprueba si estamos situados en la última fila.
- **isFirst();** Comprueba si estamos situados en la primera fila.
- **last();** Sitúa el cursor en la última fila del ResultSet
- **first();** Sitúa el cursor en la primera fila del ResultSet
- **next();** Mueve el cursor a la fila siguiente.
- **previous();** Mueve el cursor a la fila anterior.
- **getRow();** Nos devuelve la posición actual del cursor.
- **relative(int filas);** Mueve el cursor n filas a partir de la posición en la que esté situado el cursor en ese momento.

Métodos de posicionamiento por columnas

- Una vez posicionados en una fila concreta, podemos obtener los datos de una columna específica utilizando los métodos getxxx() que proporciona la interfaz ResultSet, la “xxx” especifica el tipo de dato presente en la columna.

- Para cada tipo de dato existen dos métodos getxxx():

  - getxxx(String nombreColumna); Donde especificamos el nombre de la columna donde se encuentra el dato.
  - getxxx(int numeroColumna); Donde se especifica la posición de la columna dentro de la consulta. Siempre empezando desde 1.


Publicación Driver
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
Conexión BD´s
Connection connection =
DriverManager.getConnection("jdbc:db2:CONTACT","user","user");
Preparación de sentencias
Statement stmt =
connection.createStatement();
PreparedStatemetn pstmt=;
connection.prepareStatement
("query");
CalleableStatemnt cstmt=
connection.prepareCall("query");
setString(i,valor)
Lanzamos Sentencia
pstmt.executeUpdate();
cstmt.executeUpdate();
stmt.executeUpdate("query");
pstmt.executeQuery();
cstmt.executeQuery();
stmt.executeQuery("query");
pstmt.execute();
cstmt.execute();
stmt.execute("query");
getResultSet()
int
ResultSet
boolean
Datos
getResultSetMetaData()
ResultSetMetaData
Metadatos

Que es JPA?

- JPA es el acrónimo de Java Persistence API y se podría considerar como el estándar de los frameworks de persistencia.

- En JPA utilizamos anotaciones como medio de configuración.

Que es una entidad?

- Consideramos una entidad al objeto que vamos a persistir o recuperar de una base de datos. Se puede ver una entidad como la representación de un registro de la tabla.
- Toda entidad ha de cumplir con los siguientes requisitos:
  - Debe implementar la interface Serializable
  - Ha de tener un constructor sin argumentos y este ha de ser público.
  - Todas las propiedades deben tener sus métodos de acceso get() y set().
- Para crear una entidad utilizamos la anotación @Entity, con ella marcamos un POJO como entidad

Crear una entidad

- Para crear una entidad utilizamos la anotación @Entity, con ella marcamos un POJO como entidad

```
@Entity
public class Persona implements Serializable
```

Anotaciones en JPA

- En este apartado vamos a ver las anotaciones más utilizadas para mapear cada una de las propiedades de la entidad contra la BBDD.

@Id

@IdClass

@Embeddable

@Embedded

@EmbeddedId

@Table

@Column

@SecondaryTable

@Enumerated

@Lob

@Basic

@Temporal

@Id

- Se utiliza para marcar una propiedad como clave primaria tanto si es una PK (Primary Key) simple, de un solo cambio o si forma parte de una PK compuesta, de varios campos.

```
@Entity
public class Persona implements Serializable {

	@Id
	private String nif;
```

@IdClass

- Esta anotación nos sirve para poder anotar una clave primaria compuesta.
- En una clase aparte definimos la clave primaria compuesta cumpliendo los siguientes requisitos:
  - La clase ha de implementar la interface Serializable.
  - En la clase creamos una propiedad por cada campo que forma parte de la PK.
  - Estas propiedades deben tener un método get() y set() cada una de ellas.
  - También debemos facilitar un constructor sin argumentos.
  - Debemos implementar los métodos equals() y hashCode().

@IdClass

```
public class PersonaPK implements Serializable{

    private Long telefono;
    private String nif;

    public PersonaPK() {
    }

    @Override
    public boolean equals(Object obj) {...}

    @Override
    public int hashCode() {...}
```

@IdClass

- En la entidad utilizamos la anotación @IdClass para especificar la clase que utilizamos como clave primaria compuesta.

```
@Entity
@IdClass(PersonaPK.class)
public class Persona implements Serializable {

    @Id
    @Column(name = "id_telefono", nullable = false)
    private Long telefono;


    @Id
    @Column(name = "id_nif", nullable = false)
    private String nif;
```

@Embeddable

- Cada una de las propiedades definida en la entidad, pasará a ser un campo en la tabla.
- Esta anotación nos permite embeber una clase.

```
@Embeddable
public class Direccion implements Serializable{

    private String calle;
    private String localidad;

    @Column(name="codigo_postal")
    private int cp;

    public Direccion() {
    }
```

@Embedded

- Con esta anotación hacemos que la propiedad de tipo Dirección se persista según la clase Embeddable creada anteriormente.
- Si no utilizásemos la anotación se generaría un campo direccion con el identificador del objeto.

```
@Embedded
private Direccion direccion;
```

@EmbeddedId

- Es otra alternativa a la creación de claves primarias compuestas.
- Utilizamos esta anotación para especificar la propiedad que actúa como PK.

```
@Entity
public class Persona implements Serializable {

    @EmbeddedId
    PersonaPK personapk;
```

- La clase PersonaPK estará marcada como @Embeddable.

@Table

- Mediante esta anotación indicamos el nombre de la tabla donde se persistirán o recuperaran los datos de la entidad.
- Si no la utilizásemos coge como nombre de la tabla el nombre de la clase. Según el ejemplo sería Persona.

```
@Entity
@IdClass(PersonaPK.class)
@Table(name = "Ejemplo1_PERSONAS")
public class Persona implements Serializable {
```

@Column

- Lo mismo ocurre con las columnas, podemos especificar su nombre, tamaño, ...etc. Cada propiedad de la entidad puede utilizar una anotación @Column para establecer las propiedades de la columna en la tabla.
- Si no se utiliza, coge como nombre de columna el nombre de la propiedad.

```
@Id
@Column(name = "id_nif", nullable = false)
private String nif;
```

```
@Column(length = 1)
private char sexo;
```

@SecondaryTable

- Una entidad puede utilizar tablas secundarias para almacenar datos
- La anotación @SecondaryTable permite crear tablas secundarias. En esta anotación utilizamos dos atributos:
  - El atributo name; establecemos el nombre de la tabla.
  - El atributo pkJoinColumns; unimos la tabla primaria y la tabla secundaria.

- Para esta última operación utilizamos otra anotación @PrimaryKeyJoinColumn cuyos atributos son:
  - name; es el nombre del campo generado en la tabla secundaria
  - referencedColumnName; especificamos el nombre del campo de la tabla primaria por el cual se unen.

@SecondaryTable

```
@Entity
@IdClass(PersonaPK.class)
@Table(name = "Ejemplo1_PERSONAS")
@SecondaryTable(name = "Ejemplo1_CV",
                pkJoinColumns = {
                    @PrimaryKeyJoinColumn(name = "id_telefono",
                                          referencedColumnName = "id_telefono"),
                    @PrimaryKeyJoinColumn(name = "id_nif",
                                          referencedColumnName = "id_nif")
})
public class Persona implements Serializable {
```

```
    @Lob
    @Basic(fetch = FetchType.LAZY)
    @Column(table = "Ejemplo1_CV")
    private String cv;
```

@Enumerated

- En Java utilizamos los tipos enumerados. Este tipo de datos no lo reconoce la base de datos, por lo cual, debemos utilizar esta anotación para indicar como persistir una propiedad de tipo enumerado.
- Lo podemos hacer de dos formas:
  - Almacenando su índice; EnumType.ORDINAL
  - Almacenando su valor; EnumType.STRING

```
@Enumerated(EnumType.STRING)
private EstadoCivil estado;
```

@Lob

- Designa la propiedad de un campo como:
  - CLOB (Character Large Object); se utiliza para introducir un campo de texto muy grande. En nuestro ejemplo lo utilizamos para el curriculum de la persona.
  - BLOB (Binary Large Object); permite almacenar archivos binarios como por ejemplo una imagen.

```
@Lob
@Basic(fetch = FetchType.LAZY)
@Column(table = "Ejemplo1_CV")
private String cv;
```

@Basic

- Cuando utilizamos la anotación anterior es recomendable especificar el tipo de recuperación. Esto se denomina el tipo fetch.
- Para este fin utilizamos la anotación @Basic, mediante la cual especificamos si recuperamos automáticamente los datos o no.
    - FetchType.LAZY; Con este tipo fetch estamos indicando que al recuperar los datos de la entidad, el campo marcado de esta forma no se recupera hasta que no se solicite específicamente.
    - FetchType.EAGER; Este otro tipo indica que los datos referentes a este campo se recuperan de forma implícita.

@Temporal

- Utilizamos esta anotación para indicar como queremos persistir los objetos de tipo fecha. Puede adoptar las siguientes constantes:
  - TemporalType.DATE; almacena la fecha como día, mes y año.
  - TemporalType.TIME; se almacena la hora como horas, minutos y segundos.
  - TemporalType.TIMESTAMP; almacena fecha y hora juntas.

```
@Temporal(TemporalType.DATE)
private Date fechaNacimiento;
```

Persistence.xml

- En este archivo vamos a definir las unidades de persistencia. Se necesita una unidad de persistencia por cada base de datos.

- También se necesitan distintas unidades de persistencia si elegimos diferentes estrategias de generación de tablas. Por ejemplo: si elegimos eliminar y crear la tabla para persistir datos y elegimos ninguna para efectuar lecturas.

Persistence.xml

- Cada unidad de persistencia ha de contener los siguientes datos:
  - **name**; nombre de la unidad de persistencia. Necesitamos un nombre para luego poder generar el objeto EntityManagerFactory del cual hablaremos más adelante.
  - **transaction-type**; especificamos el tipo de transaccionalidad elegida.
  - **provider**; necesitamos de un proveedor de persistencia. JPA actúa igual que JDBC, esto es, necesitamos una implementación del API para poder trabajar con la BBDD. En JDBC necesitábamos el driver de la BBDD y en JPA necesitamos de un proveedor de persistencia. En este ejemplo hemos optado por elegir TopLink.
  - **class**; aquí deberán especificarse todas las entidades que vamos a manejar. En nuestro ejemplo detallamos la entidad Persona.
  - **properties**; Como propiedades introducimos los datos necesarios para poder establecer una conexión a la base de datos. También como propiedad elegimos la estrategia de generación de tablas, pudiendo tomar estos valores:
    - **create-tables**; Intentará crear las tablas para cada operación.
    - **drop-and-create-tables**; Eliminará y creará las tablas de nuevo.
    - **Ninguno**; Ni elimina, ni crea tablas. Si optamos por este valor no aparece la propiedad en la unidad de persistencia.

EntityManager

- Podriamos decir que EntityManager es el objeto clave en JPA.
- El EntityManager realiza las operaciones CRUD (crear, leer, actualizar y eliminar) sobre las entidades.
- Además, el EntityManager también trata de mantener las entidades sincronizadas con la base de datos automáticamente.

EntityManagerFactory

- Para obtener el EntityManager necesitamos previamente obtener un objeto de tipo EntityManagerFactory.

- Se creará a partir del nombre de la unidad de persistencia facilitado como argumento.

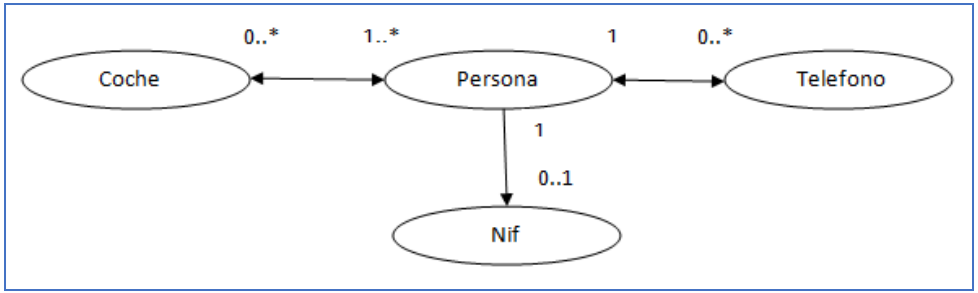
EntityTransaction

- Dependiendo de la operación que vayamos a realizar necesitamos una transacción o no. En el caso de las operaciones de persistencia es obligatorio obtener una, sin embargo en las operaciones de lectura no es necesario.

- Mediante el objeto EntityTransaction podemos obtener una transacción a partir del EntityManager.

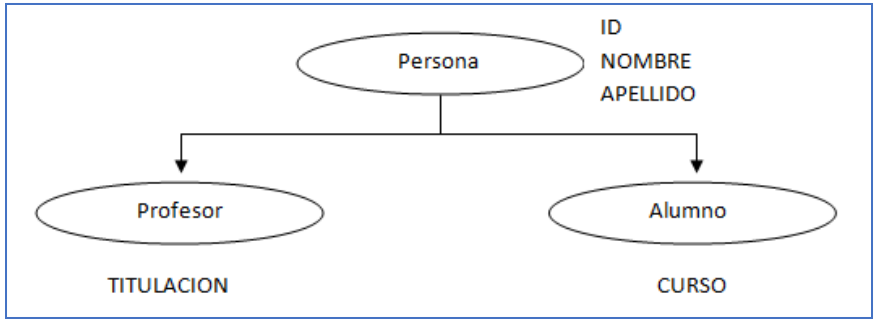
Relacción entre entidades

- Para establecer las relaciones entre entidades utilizamos anotaciones como:
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany



Estrategias del mapeo de herencia

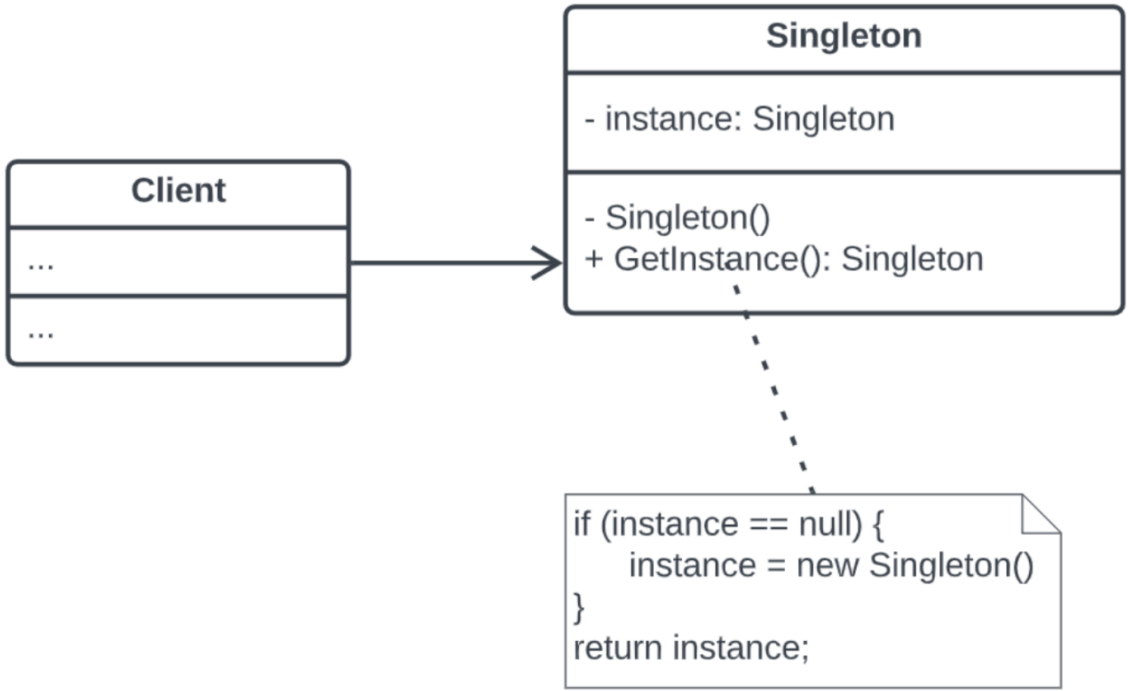
- Para mapear la herencia utilizamos tres estrategias:
  - SINGLE_TABLE
  - JOINED
  - TABLE_PER_CLASS



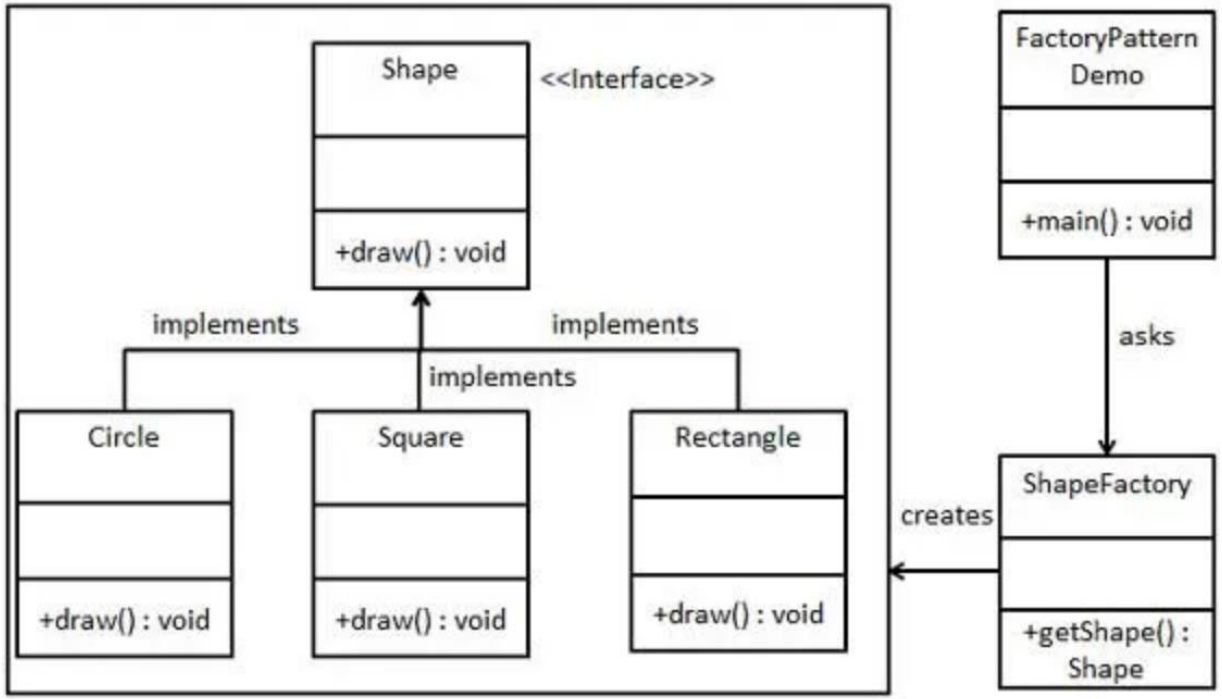
Patrones de diseño: Singleton y Factory

C/ Manuel Tovar, 31 28034-Madrid

Singleton


Singleton
- instance: Singleton
- Singleton()
+ GetInstance(): Singleton
Client
...
...
if (instance == null) {
instance = new Singleton()
}
return instance;

Factory


Shape
<<Interface>>
+draw() : void
implements
implements
implements
Circle
+draw() : void
Square
+draw() : void
Rectangle
+draw() : void
FactoryPattern Demo
+main() : void
asks
creates
ShapeFactory
+getShape() : Shape

Tecnofor by SNGULAR

Spring Boot

C/ Manuel Tovar, 31 28034-Madrid

Que es Spring Boot

- Spring Boot es una parte de Spring que nos permite crear diferentes tipos de aplicaciones de una manera rápida y sencilla.

- Sus características principales son que provee out-of-the-box una serie de elementos que nos permiten desarrollar diferentes tipos de aplicaciones de forma casi inmediata. Algunas de estas características son:

  - Servidores de aplicaciones embebidos (Tomcat, Jetty, Undertow)
  - POMs con dependencias y plug-ins para Maven
  - Uso extensivo de anotaciones que realizan funciones de configuración, inyección, etc.

Configuración del pom

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
</parent>

<properties>
    <java.version>1.8</java.version>
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.1.11</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-logging</artifactId>
        <version>2.1.4.RELEASE</version>
    </dependency>
</dependencies>
```

Principales Anotaciones

- La etiqueta **@Configuration**, indica que la clase en la que se encuentra contiene la configuración principal del proyecto.

- La anotación **@EnableAutoConfiguration** indica que se aplicará la configuración automática del starter que hemos utilizado. Solo debe añadirse en un sitio, y es muy frecuente situarla en la clase main.

- En tercer lugar, la etiqueta **@ComponentScan**, ayuda a localizar elementos etiquetados con otras anotaciones cuando sean necesarios.

- Para no llenar nuestra clase de anotaciones, podemos sustituir las etiquetas @Configuration, @EnableAutoConfiguration y @ComponentScan por **@SpringBootApplication**, que engloba al resto.

Clase principal

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloWorldApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloWorldApplication.class, args);
    }
}
```

Tecnofor by SNGULAR

JSON y XML

C/ Manuel Tovar, 31 28034-Madrid

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<listadeclientes>
    <cliente>
        <numerodecliente>12345</numerodecliente>
        <Nombre>Luis García</Nombre>
        <Direccion>
            <Calle>Calle de la Princesa</Calle>
            <Ciudad>Madrid</Ciudad>
            <Codigo Postal>28020</Codigo Postal>
            <Pais>España</Pais>
        </direccion>
    </cliente>
</listadeclientes>
```

JSON

```
{
  “firstName”: “Jonathan”,
  “lastName”: “Freeman”,
  “loginCount”: 4,
  “isWriter”: true,
  “worksWith”: [“Spantree Technology Group”, “InfoWorld”],
  “pets”: [
    {
      “name”: “Lilly”,
      “type”: “Raccoon”
    }
  ]
}
```

XML y JSON

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<endereco>
    <cep>31270901</cep>
    <city>Belo Horizonte</city>
    <neighborhood>Pampulha</neighborhood>
    <service>correios</service>
    <state>MG</state>
    <street>Av. Presidente Antônio Carlos, 6627</street>
</endereco>
```

vs.

JSON

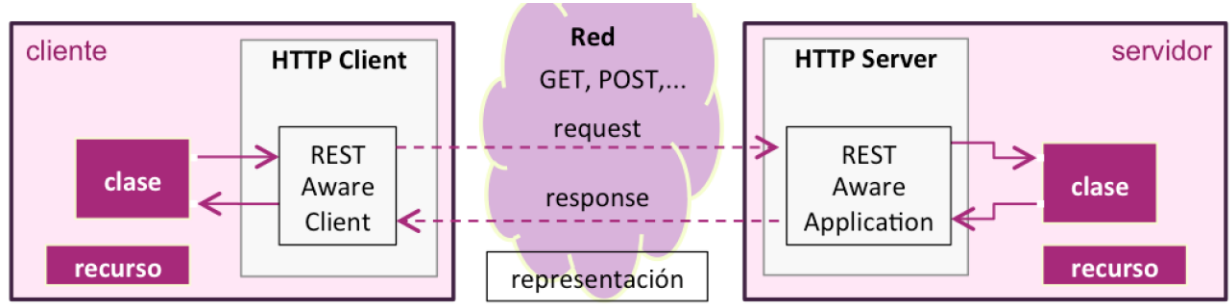
```json
{
    "endereco": {
        "cep": "31270901",
        "city": "Belo Horizonte",
        "neighborhood": "Pampulha",
        "service": "correios",
        "state": "MG",
        "street": "Av. Presidente Antônio Carlos, 6627"
    }
}
```

Servicios REST

C/ Manuel Tovar, 31 28034-Madrid

Principios de la arquitectura REST

- **REST (Representational State Transfer)** es un estilo de arquitectura para sistemas distribuidos, desarrollada por la W3C, junto con el protocolo HTTP.
- Las arquitecturas REST tienen clientes y servidores.
- El cliente realiza un envío (request) al servidor, el cual lo procesa y retorna una respuesta al cliente.
- Las peticiones y respuestas son construidas alrededor de representaciones de recursos. Recurso es una entidad, y representación es cómo se formatea.



Principios de la arquitectura REST

- Una API del tipo RESTful, o RESTful Web Service, es una API web implementada con HTTP y los principios REST, con los siguientes aspectos:
  - Una URI base del servicio.
  - Un formato de mensajes, por ejemplo JSON o XML.
  - Un conjunto de operaciones, que utilizan los métodos HTTP (GET, PUT, POST o DELETE).
- La API debe manejar hipertextos.
- A diferencia de los Web Services basados en SOAP, no hay un estándar comúnmente aceptado para los RESTful. Esto es porque REST es una arquitectura, mientras que SOAP es un protocolo.
- Esta desventaja se compensa con la simplicidad de su utilización y el bajo consumo de recursos durante el binding. Esto es especialmente útil en aplicaciones para dispositivos móviles

Principios de la arquitectura REST

- Con REST, los **métodos HTTP** se asocian a tipos de operaciones sobre recursos. El uso comúnmente aceptado es el siguiente:
  - **GET**: Para recuperar la representación de un recurso. Es idempotente, es decir, si se invoca múltiples veces, retorna el mismo resultado.
  - **POST**: Para crear un recurso, o para actualizarlo. También, por las características del método, se utiliza para envíos grandes, o para evitar limitaciones de los otros métodos.
  - **PUT**: Para actualizar un recurso, ya que POST no es idempotente.
  - **DELETE**: Para eliminar un recurso.
  - **OPTIONS**: Se puede utilizar para hacer un "ping" del servicio, es decir, verificar su disponibilidad.
  - **HEAD**: Para buscar un recurso o consultar estado. Similar a GET, pero no contiene un body.

Web Services REST con Spring Boot

```java
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SaludoRest {

    // http://localhost:8080/hola
    @RequestMapping("/hola")
    public String hola() {
        return "Bienvenidos al curso";
    }

    // http://localhost:8080/adios?usuario="Anabel"
    @RequestMapping("/adios")
    public String adios(@RequestParam(value="usuario", defaultValue="Admin") String user) {
        return "Nos vamos a desayunar " + user;
    }

}
```

Web Services REST con Spring Boot

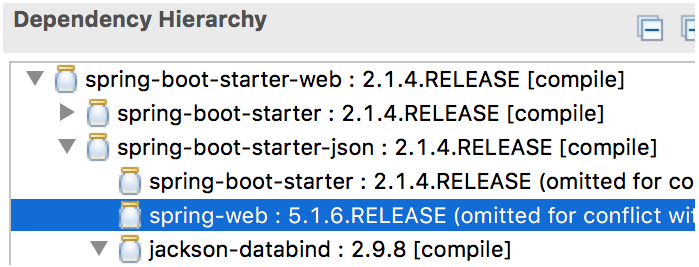
- Al agregar esta dependencia al pom.xml:

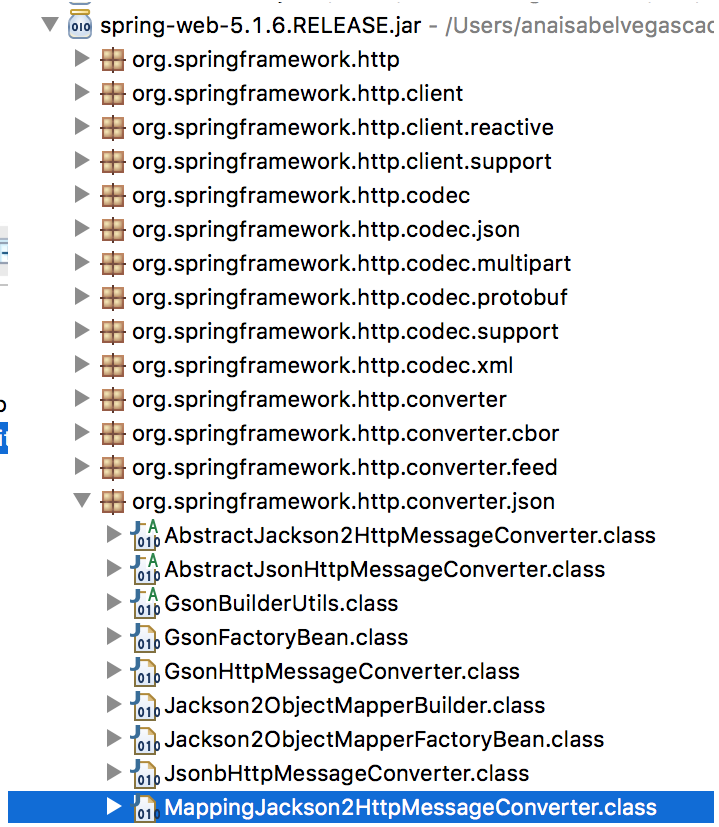
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- La clase MappingJackson2HttpMessageConverter se encarga de convertir automáticamente la instancia a devolver en un formato JSON.

Web Services REST con Spring Boot

- Formateando la respuesta a formato JSON:





Consumiendo un servicio REST

- Para poder consumir un servicio Rest la dependencia Spring –Web nos proporciona un objeto que nos facilitara mucho la conectividad con el servicio. **RestTemplate**.

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
```

Consumiendo un servicio REST

- Una vez obtenido el objeto **RestTemplate** podemos lanzar la petición al servicio:

```
Producto producto = restTemplate.getForObject(
        "http://localhost:8080/productos?codigo=2", Producto.class);
```

- Para mostrarlo en formato json debemos agregar la siguiente dependencia al pom.xml:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

Tecnofor by SNGULAR

Gracias por vuestra asistencia

C/ Manuel Tovar, 31 28034-Madrid