


Programación Java



**Si tengo una duda y no pregunto por no pasar
vergüenza 5 minutos...
es muy probable que esa duda me haga pasar
vergüenza muchas veces en mi vida**

**POR FAVOR, preguntad todo lo que se os ocurra...
AQUÍ NOS HEMOS REUNIDO PARA APRENDER!!!**

Un poco de historia

1954-1957  FORTRAN


1959: Grace Hopper  COBOL

1972: Dennis Ritchie  C programming language

1986: Bjarne Stroustrup  C++


1995: Sun Microsystems  Java (primer versión oficial)


2002: Microsoft  C#

2010: Oracle Corporation  Compra Sun Microsystems

Orígenes de Java

1990-1992 “Green Project”  Oak ... Green

WWW de texto a gráfico  Distribución libre de Green

Primera aplicación  Browser para HTML

Green  Java

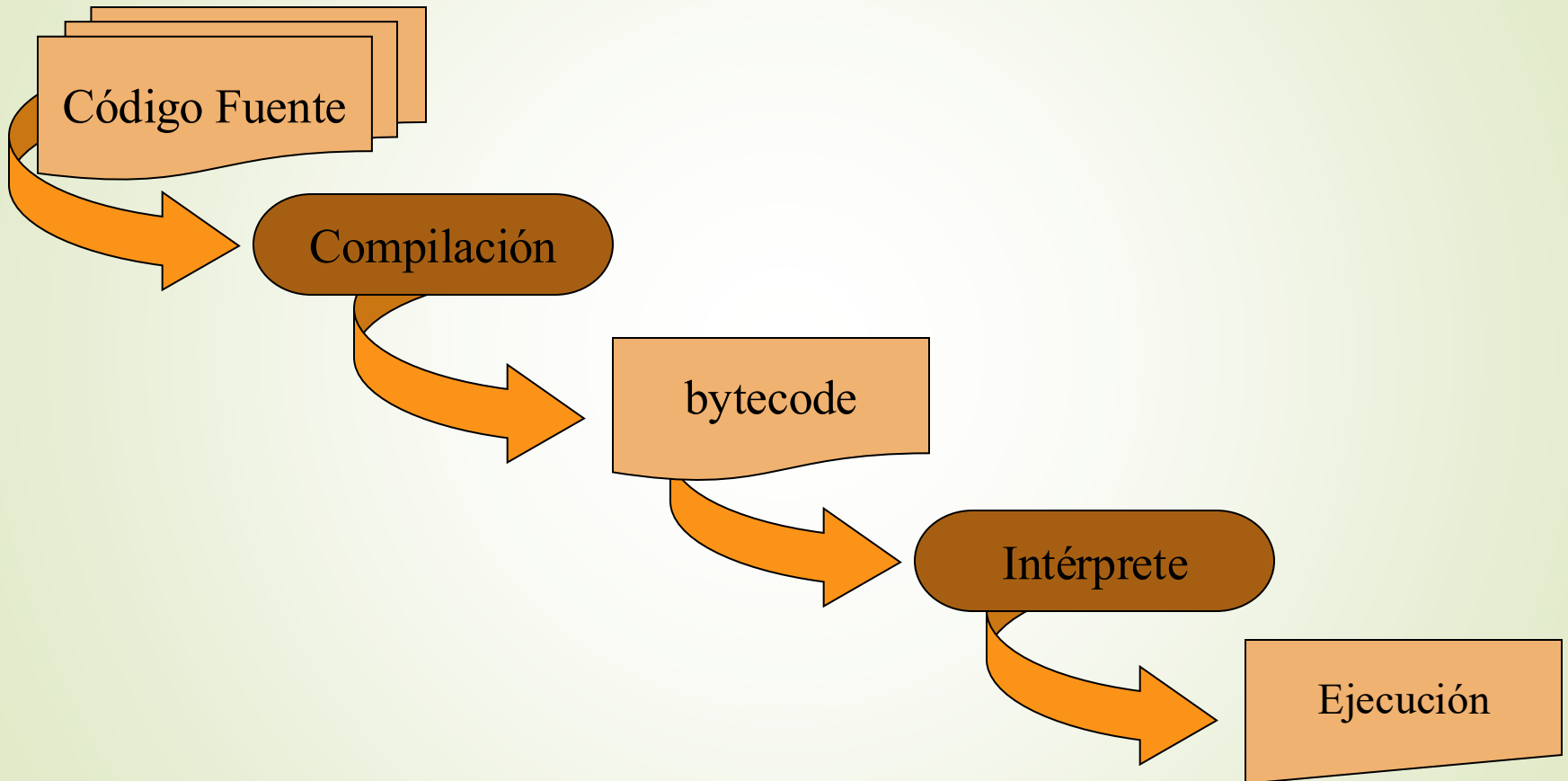
Versiones

año	Plataforma	Nombre	Características principales
1995	(Beta)		
1996	JDK 1.0		
1997	JDK 1.1		Inner classes, Java Beans, Reflection
1998	J2SE 1.2	Playground	Swing classes (GUI)
2000	J2SE 1.3	Kestrel	Java Naming and Directory Interface (JNDI)
2002	J2SE 1.4	Merlin	New I/O, regular expressions, XML parsing
2004	J2SE 5.0	Tiger	Generic types, annotations, enum types, ...
2006	Java SE 6	Mustang	Scripting language support
2011	Java SE 7	Dolphin	String in switch statement, exceptions, ...
2014	Java SE 8	(s/nombre)	API Fechas, métodos default en Interfaces, Lambda expressions
2018	Java SE 11	(s/nombre)	
2021	Java SE 17	(s/nombre)	
2023	Java SE 21	(s/nombre)	

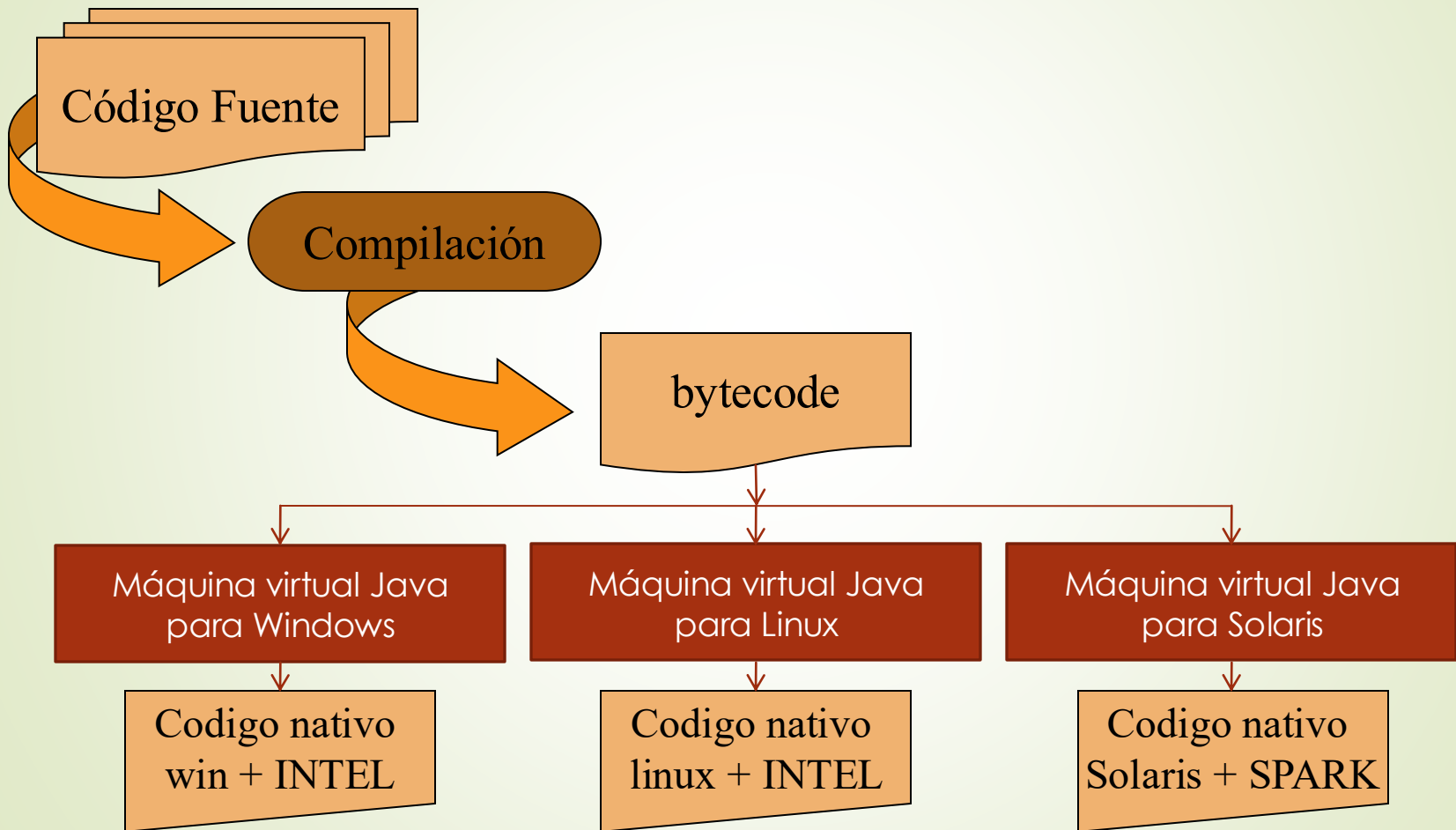
Características

- Lenguaje de propósito general.
- Sintaxis basada en C++.
- Orientado a objetos.
- Librerías: red, hilos, excepciones, gráficos.
- Código fuente y compilado, independiente de la plataforma.

Independencia de la Plataforma



Independencia de la Plataforma



JRE (Java Runtime Environment)

- Máquina virtual (java.exe)
- Conjunto de librerías estándar

JDK (Java Development Kit)

- Compilador (javac.exe)
- Intérprete (java.exe)
- Depurador (jdb.exe)
- Utilidades (jar, javadoc, etc.)

- Opcional: documentación y fuentes de librerías

Tecnología Java

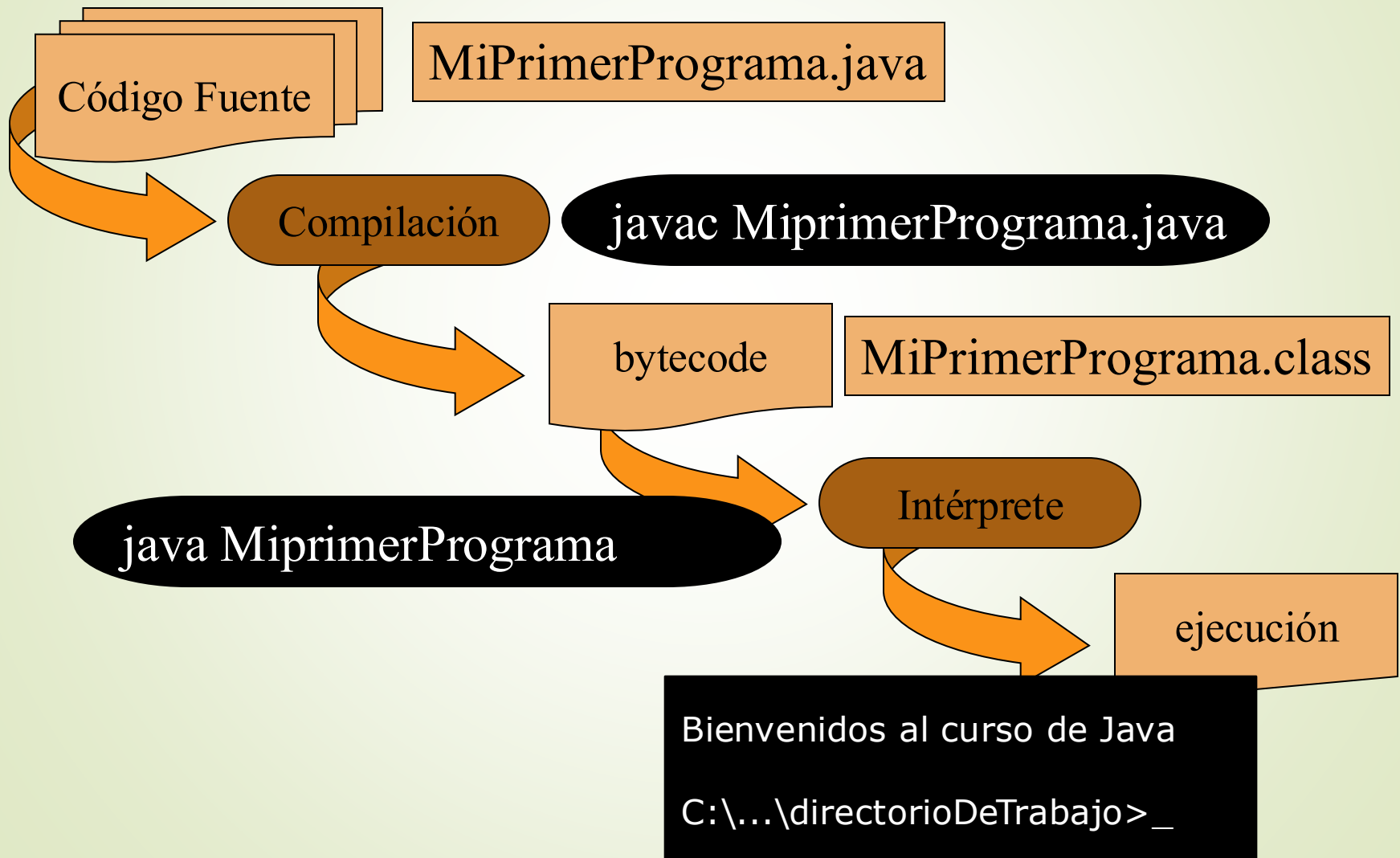
- Java SE (Java Platform Standard Edition)
- Java EE (Java Platform Enterprise Edition)
 - EJB (Enterprise JavaBeans)
 - Servlets y JSP
 - JPA
 - JSF

El Primer Programa

```
class MiPrimerPrograma {  
    public static void main(String[] arg) {  
        System.out.println("Bienvenidos al curso de Java");  
    }  
}
```

```
C:\...\directorioDeTrabajo>javac MiPrimerPrograma.java  
C:\...\directorioDeTrabajo>java MiPrimerPrograma  
Bienvenidos al curso de Java  
C:\...\directorioDeTrabajo>_
```

Compilación y Ejecución



Sintaxis

EXPRESIONES:

Conjunto de elementos o tokens que son evaluados para devolver un resultado o realizar una tarea.

TOKENS:

- **Identificadores:** nombre de variables, clases, paquetes, métodos y constantes.
- **Palabras claves:** Identificadores reservados del lenguaje
- **Literales:** valores que pueden tomar las variables, son valores constantes.
- **Operadores:** Indican la evaluación que se debe aplicar a un objeto o dato.
- **Separadores:** Indican al compilador donde se ubican los diferentes elementos del código: `{,;,;`
- **Los comentarios, retornos de carro, espacios vacíos y de tabulación, no se consideran tokens, ya que el compilador los elimina.**

Sintaxis

EXPRESIONES:

Conjunto de elementos o tokens que son evaluados para devolver un resultado o realizar una tarea.

TOKENS:

- **Identificadores:** nombre de variables, clases, paquetes, métodos y constantes.
- **Palabras clave:** `//...`
- **Literales:** valores y constantes.
- **Operadores:** Inicialización de variables o dato.
- **Separadores:** Separadores de los elementos del código.
- Los comentarios, retornos de carro, espacios vacíos y de tabulación, no se consideran tokens, ya que el compilador los elimina.

```
//...
while (i == 0) {
    resu = a + b;
    i = 1;
    //bucle que no sirve para nada
}
System.out.println(resu);
//...
```

Sintaxis

- ▶ En java se hace distinción entre mayúsculas y minúsculas.
- ▶ Las instrucciones de java terminan en punto y coma.
- ▶ Los bloques de código se delimitan mediante llaves }

Identificadores

p.ej. nombres de variables

- Secuencia de letras, dígitos y los caracteres _ y \$
- Debe comenzar con una letra, \$ ó _. Se recomienda que inicien con una letra.
- Letra: cualquier carácter Unicode que describa una letra: ñ, á, Π, ä, etc.
- No se puede utilizar palabras reservadas.

Ej.:

nombre, saldoActual, PORCENTAJE_IVA,
año, paísDeNacimiento, valor01

Convenciones y Recomendaciones

Clases: Empiezan en mayúscula y si hay cambio de palabra, la primera letra también en mayúscula.

```
class PrimerEjemplo { ..... }
```

Métodos y atributos: Empiezan en minúsculas pero si cambia la palabra, la primera letra de la palabra es en mayúsculas.

```
void mostrarMensaje(){  
    String mensajeAMostrar = “Hola, buenas tardes”;  
    System.out.println(mensajeAMostrar);  
}
```

Constantes: Todo en mayúsculas y cuando cambia de palabra se separan con guión bajo.

Paquetes: Todo en minúsculas. (java.util, java.lang)

(Las palabras reservadas del lenguaje son todas en minúsculas)

Palabras claves

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

Comentarios

Junto al código fuente de un programa, se pueden insertar comentarios. Pueden ser de una sola línea, en ese caso empiezan con dos barras de división (//) o pueden ser de varias líneas, en ese caso el comentario se encierra con barra y asterisco (/* comentario de varias líneas */).

```
//ESTE ES EL PRIMER PROGRAMA

public class MiPrimerPrograma {
    /*muestra un mensaje
       por la
       consola
    */
    public static void main(String[] args) {//otro comentario
        System.out.println("Bienvenidos al curso de Java");
    }
}
```

Tipos Primitivos de Datos

		TIPO	TAMAÑO	RANGO
N U M É R I C O S	Enteros	byte	8 bits (1byte)	-128 a 127
		short	16 bits (2 bytes)	-32768 a 32767
		int	32 bits (4 bytes)	$-2... \times 10^9$ a $2... \times 10^9$
		long	64 bits (8 bytes)	$-9.. \times 10^{18}$ a $9.. \times 10^{18}$
	Reales	float	32 bits (4 bytes)	IEEE 754
		double	64 bits (8 bytes)	IEEE 754
		char	16 bits (2 bytes)	'\u0000' a '\uFFFF'
		boolean		true o false

Operadores


Aritméticos	$+$, $-$, $*$, $/$, $\%$
	$++$, $--$
Relacionales	$>$, $>=$, $<$, $<=$, $==$, $!=$
Lógicos	$\&\&$, $\ $, $!$, $\&$, $ $
Desplazamiento y Lógicos de bits	$<<$, $>>$, $>>>$
	$\&$, $ $, $^$, \sim
Asignación	$=$, $+=$, $-=$, $*=$, $/=$, $\%=$
	$\&=$, $ =$, $\wedge=$, $<<=$, $>>=$, $>>>=$
Condicional	$?:$

Comprobación De Tipos Estricta

<tipoDeDato> <identificador>;


Ej. 1:

```
int edad; //declaración  
edad = 32; //inicialización  
float altura, peso;
```



Ej. 2:

```
int edad = 32;  
int cont = 0, acu = 0;
```



Ej. 3:

✗

```
int a = 6;  
b = a;
```

b no está
declarada

Ej. 4:

```
int cantidad = 45;  
int cantidad = cantidad + 1; ✗
```

cantidad ya
se declaró

Declaración de Constantes

```
final <tipoDeDato> <identificador>;
```

```
final int TIPO_IVA = 21;
```

```
final double COMISION = 1.5;
```


El Tipo char

“D” es una
cadena

```
char c1 = 'D';  
char c2 = 'a';
```

c2 es un
char

✗

```
char c3 = "D";
```


✗

```
String c4 = 'a';
```

‘a’ es un char, no
asimilable a
cadenas

```
String str = c2;
```

 ✗

```
String str = c2 + "";
```

 ✓

```
char c5 = 65;           // 'A'  
char c6 = 'A' + 1;     // 'B'  
int val1 = 'A';         // 65  
int val2 = 'A' + 'B';   // 131
```

 ✓

String: vista rápida

Los String en Java NO SON TIPOS PRIMITIVOS DE DATOS.

String es una clase de librería Java + operador de concatenación

No se deben comparar con `==`, debe utilizarse el método `.equals(..)`

Declaración:

```
String palabra;  
palabra = "hola";
```

```
String nombre = "Pepe";  
String apellido = "Jimenez";
```

Ejemplo uso:

```
String nombreCompleto;  
nombreCompleto = apellido + ", " + nombre;  
System.out.println(nombreCompleto);
```

Literales(numéricos)

ENTEROS

LITERAL	BASE	VALOR
101	Decimal (10)	101
0b101	Binario (2)	5
0101	Octal (8)	65
0x101	Hexadecimal (16)	257

Ej: 5 -578 +5 12_000_000 12_00

REALES

LITERAL	NOTACIÓN	VALOR
15300.0	Normal	15.300,0
1.53e+4	Científica	15.300,0

Ej: .5 -.5 +.5 .7E10 .3e-10

Literales

CARACTERES

LITERAL	VALOR
'A'	A
65	A
'\u0041'	A

BOOLEAN

LITERAL	VALOR
true	true
false	false

STRING

“No soy exactamente un literal, sino un objeto de la clase String”

Literales y tipos

- Además de su valor, tienen TIPO.
 - En java $5 \neq 5.0$
- Literales enteros: su tipo es int pero pueden asignarse valores a byte y short que serán convertidos automáticamente.
- Literales reales: su tipo de datos es double.

SUFIJOS

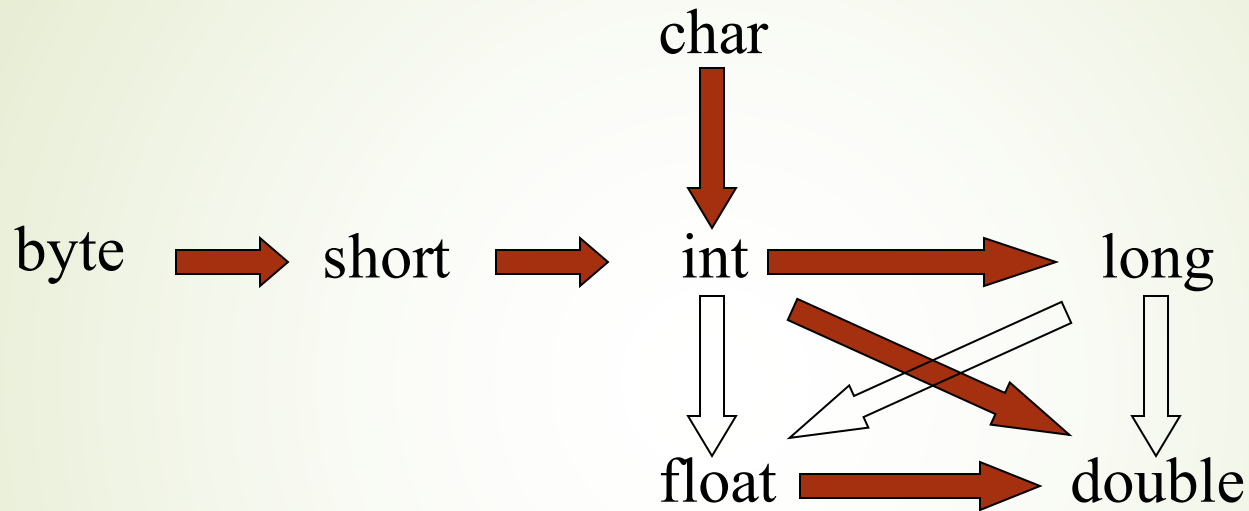
```
byte b = 98;  
short s = 32000;  
int i = 100_000;
```

```
float val = 2.5F;  
long l = 3_000_000_000L;
```

Secuencias De Escape

\"	"
\'	'
\b	Carácter backspace (BS)
\t	Tabulador (TAB)
\n	Nueva Línea (LF)
\r	Retorno de Carro (CR)
\f	Avance de Página (FF)
\uXXXX	Carácter UNICODE, XXXX hexadecimal
\\	\

Compatibilidad De Tipos De Datos



```
byte b = 65;  
int i = b;  
double d = i;
```



```
int a = 12.0; ❌
```

```
int peq = 8;  
byte b1 = peq; ❌
```

12.0 es
double

peq es int

Conversiones Explícitas (typecast)

(tipo_de_dato) expresión

```
float sueldoAnual;
```

```
...
```

```
 int sueldoMensual = sueldoAnual / 12;
```

sueldoAnual es
float, la expresión es
float

```
float sueldoAnual;
```

```
...
```

```
int sueldoMensual = (int) (sueldoAnual / 12);
```



Sentencias

- **Simples**
- **Compuestas**

- Toda sentencia simple finaliza con ;
- Sentencia compuesta o bloque: conjunto de sentencias, se delimita con {}

```
//Calculo raices
{
    discr = Math.sqrt(b * b - 4 * a * c);
    if(discr >= 0) {
        x1 = (- b + discr) / (2*a);
        x2 = (- b - discr) / (2*a);
    }
}
```

Estructuras De Control

Condicional Simple

```
if (expresión_lógica)  
    sentencia;
```

Condicional Doble

```
if (expresión_lógica)  
    sentencia_1;  
else  
    sentencia_2;
```

Estructuras De Control (cont.)

Condicional Múltiple

```
switch (pivote) {  
    case valor_1:  
        sentencia_1;  
    case valor_2:  
        sentencia_2;  
    ...  
    case valor_n:  
        sentencia_n;  
    default:  
        sentencia;  
}
```

El pivote puede ser:

- char
- Byte
- short
- int
- enum
- String (desde JDK7)

Estructuras De Control (cont.)

Condicional Múltiple

SW

```
int dia;  
// ...  
switch (dia) {  
case 1: System.out.println("Lunes");  
        break;  
case 2: System.out.println("Martes");  
        break;  
case 3: System.out.println("Miércoles");  
        break;  
        // ...  
case 7: System.out.println("Domingo");  
        break;  
default: System.out.println("Día incorrecto");  
}
```

}

Estructuras De Control (cont.)

Estructuras Iterativas

```
while (expresión_lógica)  
    sentencia;
```

```
do  
    sentencia;  
while (expresión_lógica);
```

```
int i = 5;  
while (i < 8) {  
    System.out.println(i++);  
}
```

```
int i = 5;  
do {  
    System.out.println(i++);  
} while (i < 8);
```

Estructuras De Control (cont.)

Estructuras Iterativas

```
for (sentencia_inicializ; expresión_lógica; sentencia_actualiz)  
    sentencia;
```

```
for (int i = 1; i < 8; i++) {  
    System.out.println(i);  
}
```

```
int i;  
for (i = 1; i < 8; i++) {  
    System.out.println(i);  
}
```

Estructuras De Control (cont.)

Estructuras de Ruptura

➤ **break**

Rompe la ejecución de la estructura de control.
La ejecución continúa en la siguiente instrucción de la estructura.

➤ **continue**

Rompe la ejecución de la iteración actual de la estructura de control.
La ejecución continúa en la siguiente iteración.

➤ **return**

Sale de la rutina actual y devuelve el control al llamador.
Devuelve el valor necesario en funciones.

Métodos

```
[especificadores] tipoDevuelto nombreMetodo([lista parámetros]) [...] {  
    // instrucciones  
    [return valor;]  
}
```

tipoDevuelto

- Tipos Primitivos
- Objetos
- void

Lista parámetros

- Tipo nombre
- Separados por ,

return valor;

- Obligatorio si TipoDevuelto no es void
- El tipo de valor debe ser compatible con tipoDevuelto

Métodos

```
public class ClasePrueba {  
    public static void mensaje(){  
        System.out.println("Ejecutado el procedimiento");  
    }  
  
    public static long suma(int a, int b){  
        long resu;  
        resu = a + b;  
        return resu;  
    }  
  
    public static long producto(int a, int b){  
        return a * b;  
    }  
  
    public static void main(String[] args) {  
        mensaje();  
        long resultado;  
        resultado = suma (17, 89);  
        System.out.println("17 + 89 = " + resultado);  
        int a = 20, b = 35;  
        System.out.println(a + " * " + b + " = "+ producto(a,b));  
    }  
}
```

Sobrecarga de métodos

Signature

```
public class ClasePrueba {  
    public static long producto(int a, int b){  
        return a * b;  
    }  
}
```

producto(int , int)

paquete.ClasePrueba.producto(int , int)

Sobrecarga de métodos

- No pueden existir dos métodos con la misma “signature”
- Pueden existir varios métodos con el mismo nombre, siempre y cuando difiera la lista de parámetros.

```
public class ClasePrueba {  
  
    public static long producto(int a, int b){  
        return a * b;  
    }  
  
    public static double producto(double a, double b){  
        return a * b;  
    }  
}
```

Ambito Variables (scope)

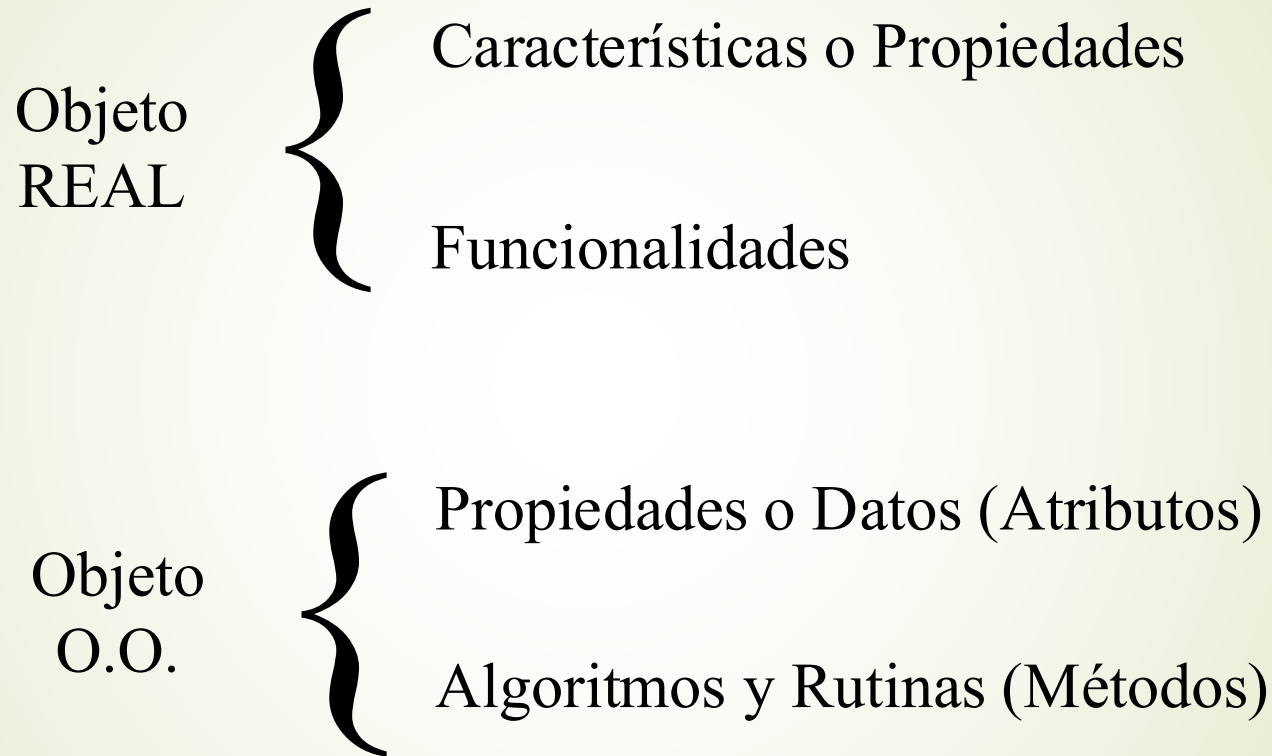
```
public class AmbitoVariables {
    static int atributoDeClase = 1;

    public static void main(String[] args) {
        int localMain = 2;
        while (localMain == 2) {
            int localWhile = 3;
            System.out.println(atributoDeClase);
            System.out.println(localMain);
            System.out.println(localWhile);
        }
        // System.out.println(localWhile);
        System.out.println(localMain);
        System.out.println(atributoDeClase);
        {
            int localBloque = 4;
            System.out.println(localBloque);
        }
        // System.out.println(localBloque);
    }

    public static void otroMetodo(){
        System.out.println(atributoDeClase);
        // System.out.println(localMain);
    }
}
```

Orientación a Objetos

Objeto



Orientación a Objetos

- **Objetivo principal:**

Reducir la complejidad del desarrollo y mantenimiento de sw.

- **Ventajas:**

Facilita el desarrollo de sistemas complejos.

Facilita la reutilización.

Paradigmas De La O.O.

- Abstracción
- Encapsulamiento
- Herencia
- Relaciones entre objetos
 - Asociación o uso (utiliza un...)
 - Agregación y composición (tiene un...)
 - Herencia (es un...)
 - Mensajes
- Polimorfismo

Abstracción



Abstracción



Es el proceso de simplificar un problema complejo enfocándose tan sólo en los aspectos relevantes para la solución. En el desarrollo de software esto significa centrarse en lo que es y hace un objeto antes de decidir cómo debería ser implementado.

TDA
Tipo de **D**ato **A**bstracto

Paradigmas De La O.O.

- Abstracción

- Encapsulamiento

- Herencia

- Relaciones entre objetos

 - Asociación o uso (utiliza un...)

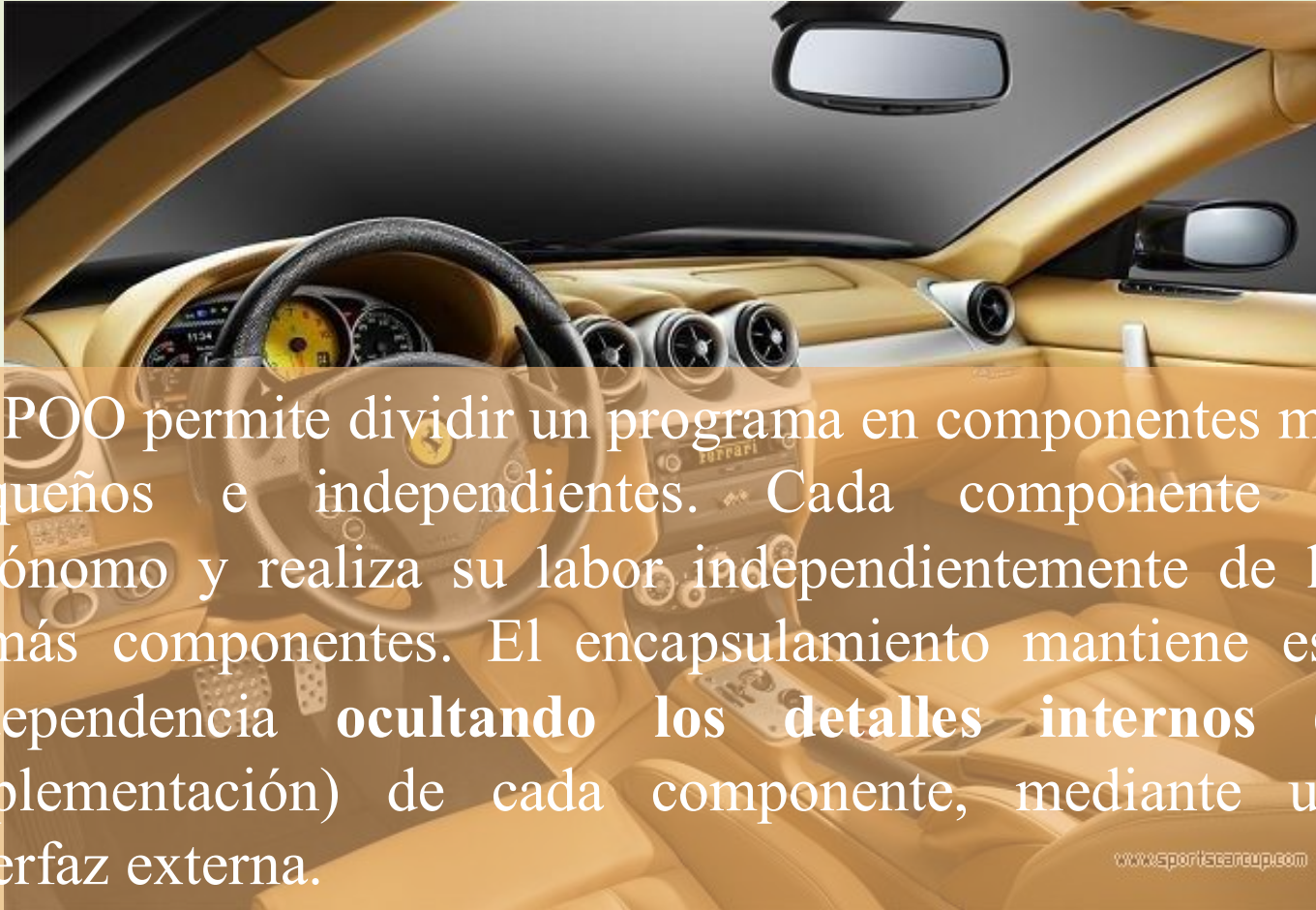
 - Agregación y composición (tiene un...)

 - Herencia (es un...)

 - Mensajes

- Polimorfismo

Encapsulamiento



La POO permite dividir un programa en componentes más pequeños e independientes. Cada componente es autónomo y realiza su labor independientemente de los demás componentes. El encapsulamiento mantiene esta independencia **ocultando los detalles internos** (la implementación) de cada componente, mediante una interfaz externa.

Paradigmas De La O.O.

- Abstracción

- Encapsulamiento

- Herencia

- Relaciones entre objetos

 - Asociación o uso (utiliza un...)

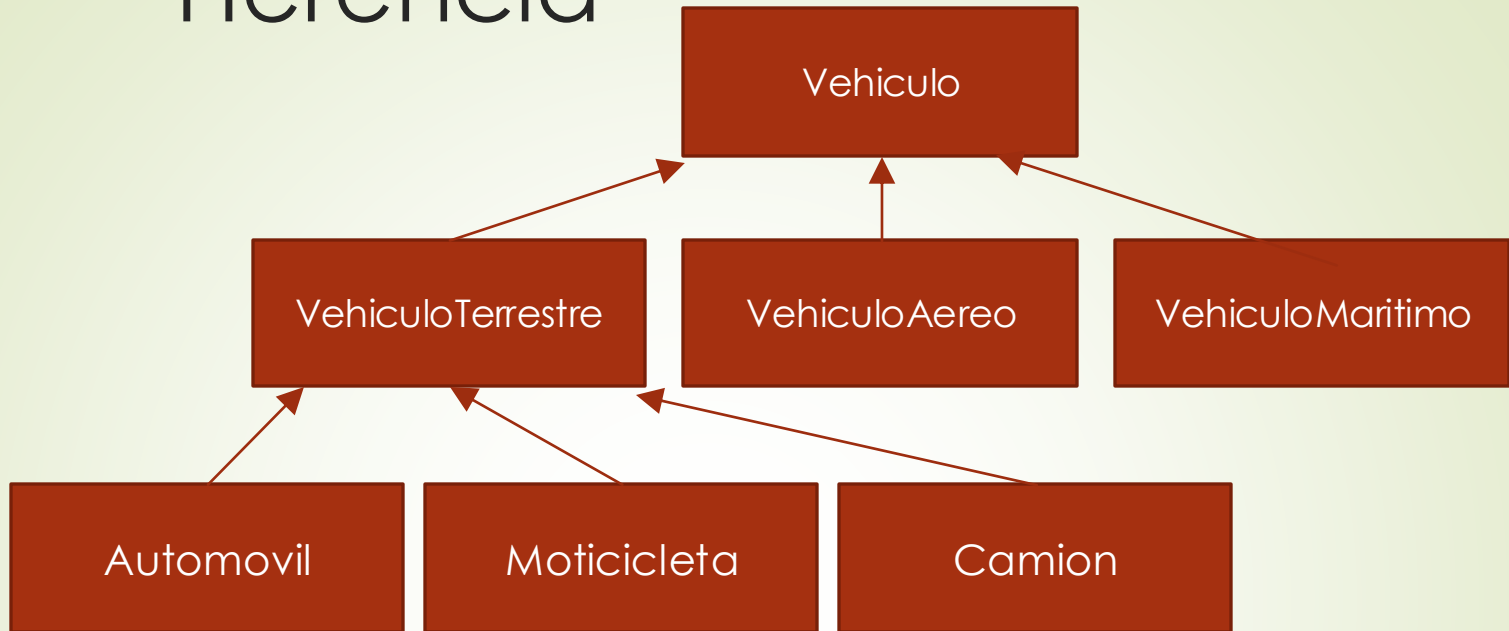
 - Agregación y composición (tiene un...)

 - Herencia (es un...)

 - Mensajes

- Polimorfismo

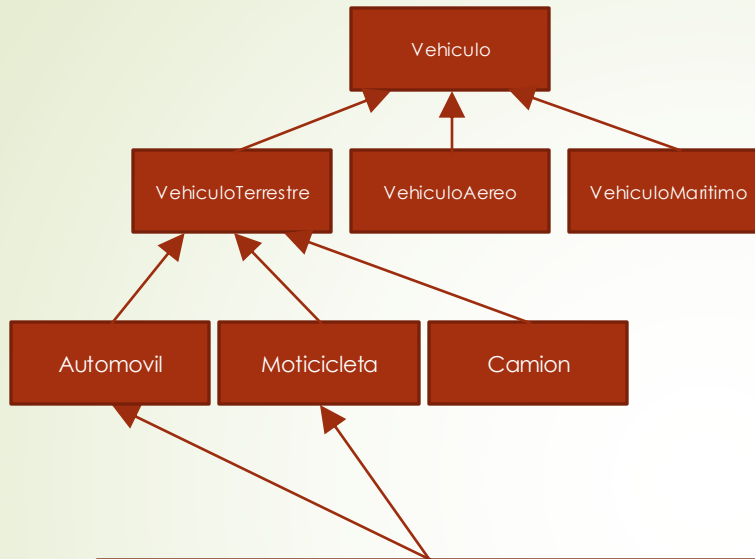
Herencia



La Herencia permite agrupar propiedades y funcionalidades en común de varias clases en una nueva, creando una estructura jerárquica de clases cada vez más especializada.

Se pueden adquirir bibliotecas de clases que ofrecen una base que puede especializarse a voluntad (la compañía que vende estas clases tiende a (y debe) proteger la implementación usando la encapsulación).

Herencia

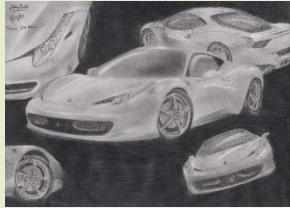


SIMPLE



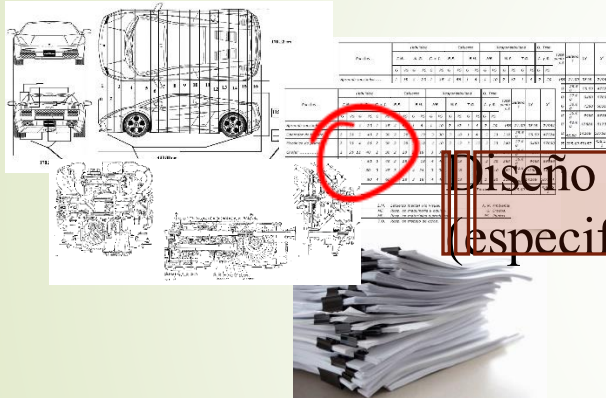
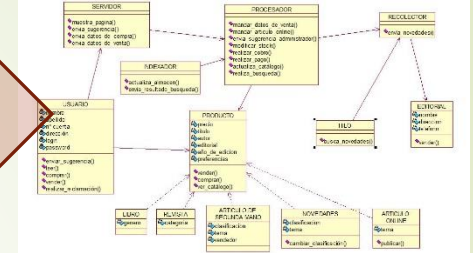
MÚLTIPLE

Clase y Objeto



Diseño (concepto)

Diseño UML



Diseño detallado
(especificaciones)

Clase Java

```
9 using System;
10 using NUnit.Framework;
11
12 namespace TestCodeCoverage
13 {
14     public class Test<T>
15     {
16         where T: struct
17         {
18             public void TestMethod(T x) {
19                 Console.WriteLine(x);
20             }
21         }
22     }
23
24     [TestFixture]
25     public class TestFixture
26     {
27         [Test]
28         public void DoTest()
29         {
30             var t = new Test<bool>();
31             t.TestMethod(true);
32         }
33     }
34 }
```



Fabricación

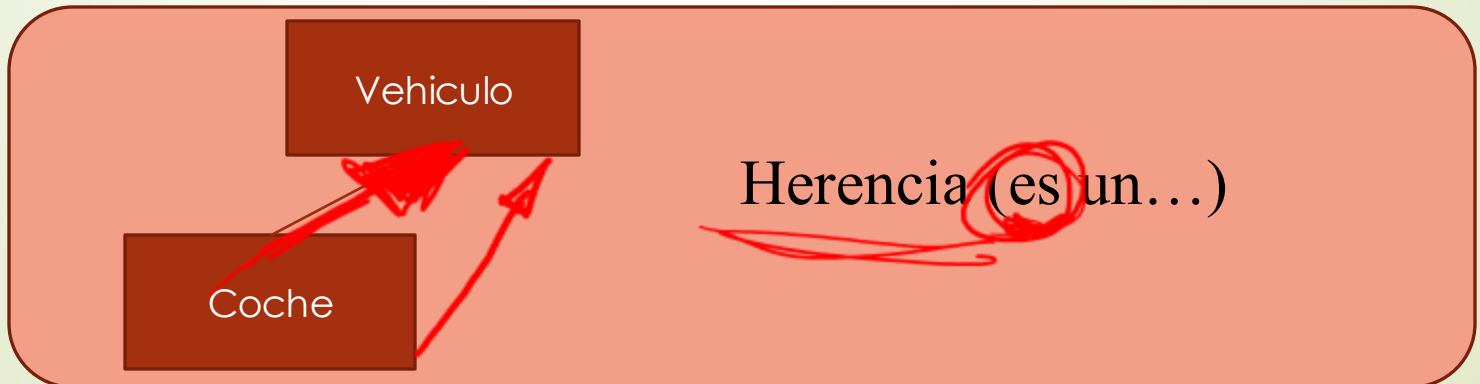
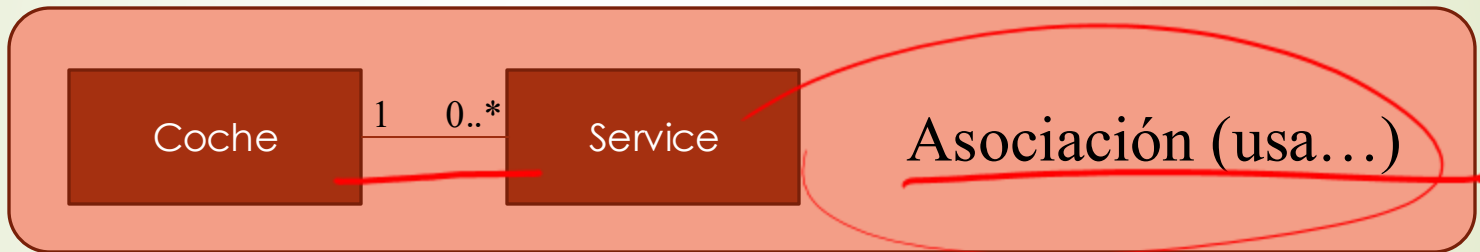
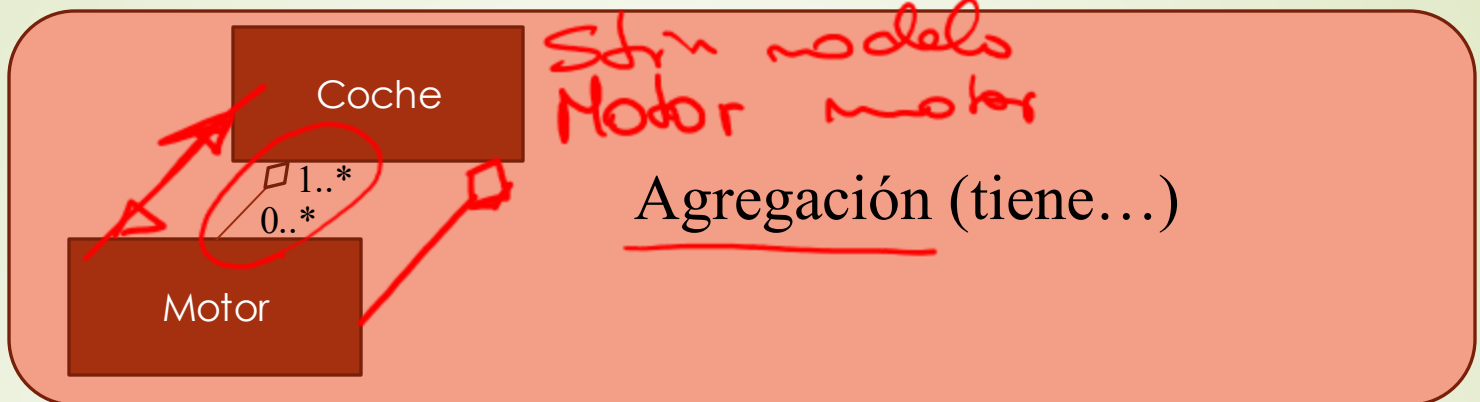
Objeto Java (instancia)



Paradigmas De La O.O.

- Abstracción
- Encapsulamiento
- Herencia
- Relaciones entre objetos
 - Asociación o uso (utiliza un...)
 - Agregación y composición (tiene un...)
 - Herencia (es un...)
 - Mensajes
- Polimorfismo

Relaciones entre objetos



Atributos y Métodos

➤ Estaticos

Conservan el mismo valor en todas los objetos.

Es como una constante de clase

➤ No Estáticos

Métodos static

Los métodos estáticos solamente pueden acceder a los atributos estáticos de la clase, no a los atributos dinámicos porque no sabrían de que objeto tomarlo.

Ejemplo de una clase con un método estático que facilita la tarea de leer del teclado.

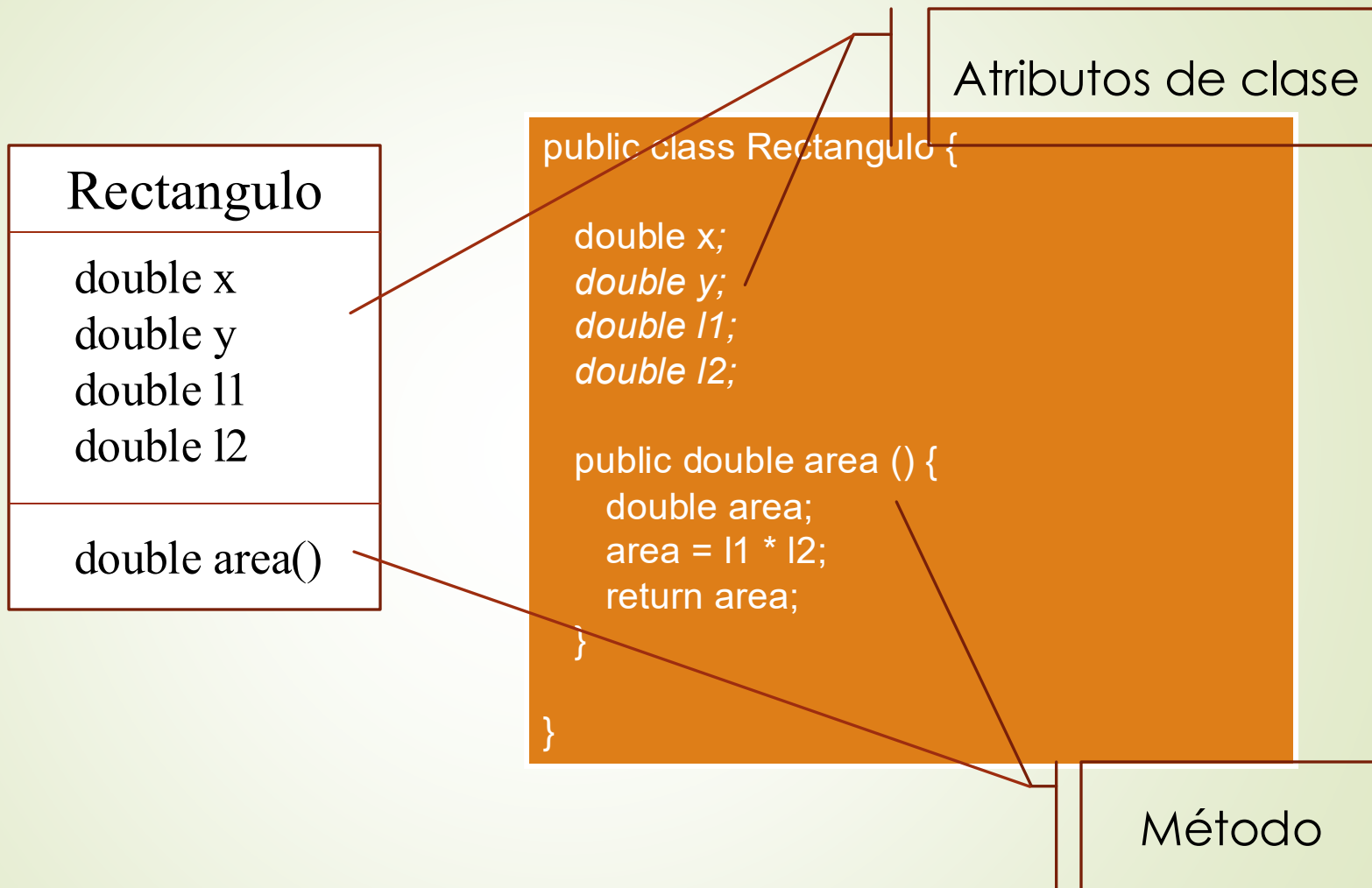
```
import java.util.Scanner;

public class Teclado{
    static Scanner teclado = new Scanner(System.in);

    public static String leer(){
        return teclado.nextLine();
    }
}
```

Uso del método estático ejecutándolo directamente desde la clase, sin necesidad de una instancia.

Miembros de Clase



Métodos

- Modificador (setter)
- Selector (getter)
- Iterador
- Constructor
- Destructor
- Propósito general

Clases Básicas Del API

Clase Math

La clase Math proporciona métodos static (que se ejecutan sobre la clase sin necesidad de crear instancias) para realizar las operaciones matemáticas más habituales.

abs()	valor absoluto	sqrt()	raiz cuadrada.
cos(), acos()	coseno y arco coseno	pow(,)	el primer argumento elevado al segundo
tan(), atan(), atan2(,)	tangente, arco tangente entre $-\pi/2$ y $\pi/2$, arco tangente entre $-\pi$ y π .	exp(), log() log10()	el número “e” elevado al argumento. Logaritmo neperiano Logaritmo decimal.
sin(), asin()	seno y arco seno	random()	número aleatorio entre [0 y 1[
ceil(), floor()	redondeo por exceso y por defecto.	toDegrees()	convierte radianes a grados
round()	entero más cercano	toRadians()	convierte grados a radianes
E	constante, número e	PI	constante, número π

Clase String

Un String es un Objeto. Se debe instanciar la clase String.

Para acceder a un objeto, también se emplean variables (hacer referencia o apuntar) a los objetos que haya creados en la memoria del sistema.

Declaración:

```
String miTexto; // igual que con tipos primitivos
```

Creación de un objeto y asignación a una variable:

```
miTexto = new String();           // Cadena vacía.
```

```
miTexto = new String("Hola");     // Con un texto.
```

```
miTexto = "Hola";                 // objeto String abreviado
```

Asignación de referencia a un objeto desde otra variable:

```
String otroObjeto = miTexto;
```

Asignación de "ninguna" referencia:

```
miTexto = null;
```

Clase String (II)

Ejemplo: String s = “Hola a todos”;

Métodos

int length()

ej. int tam = s.length(); // tam = 12

Devuelve el número de caracteres de la cadena

char charAt(int indice)

ej. char letra = s.charAt(3); // letra = ‘a’

Devuelve el carácter en la posición especificada por índice.

int indexOf(String str)

ej. int pos = s.indexOf(“tod”); //pos = 7

Devuelve la posición en la que aparece por primera vez un String (str) en el string sobre el que se aplica el método. Si no la encuentra retorna -1.

String substring(int beginIndex, int endIndex)

ej. String sub = s.substring(7, 11); //sub = “todo”

Devuelve un String extraído a partir de beginIndex y hasta endIndex.(sin incluirlo)

Clase String (III)

Ejemplo: `String s = “Hola a todos”;`

Métodos

`boolean startsWith(String prefix)` **ej. `s.startsWith(“Hola”); //true`**
Indica si un String comienza con otro String o no.

`String toLowerCase()` **ej. `String min = s.toLowerCase(); //min=“hola a todos”`**
Retorna una nueva cadena convertida a minúsculas.

`String toUpperCase()` **ej. `String may = s.toUpperCase(); //min=“HOLA A TODOS”`**
Retorna una nueva cadena convertida a mayúsculas.

`String replace(String oldStr, String newStr)`
 ej. `String s2=s.replace(“todos”,”todas”); // s2=“Hola a todas”`
Retorna un String en el que se ha sustituido una subcadena (oldStr) por otra (newStr).

StringBuffer y StringBuilder

Principales métodos

<code>append(...)</code>	agrega distintos tipos de datos convertidos a String al final de la cadena
<code>insert(...)</code>	agrega distintos tipos de datos convertidos a String en la posición indicada
<code>deleteCharAt(int index)</code>	Borra el carácter indicado
<code>delete(int start, int end)</code>	Borra los caracteres desde hasta(s/incl.)
<code>indexOf(...)</code>	
<code>charAt(int index)</code>	
<code>replace(...)</code>	
<code>Substring(...)</code>	

Classes Wrapper

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

```
int edad = Integer.parseInt("30");  
double importe = Double.parseDouble("15.45");  
  
int Integer.parseInt(String valor, int base)  
String Integer.toBinaryString(int valor)  
String Integer.toHexString(int valor)  
String Integer.toOctalString(int valor)  
String Integer.toString(int valor, int base)
```

Arrays

Arrays (I)

Son objetos con una característica especial, que pueden contener referencias, ordenadas bajo un índice, a otros objetos.

Definición de matrices de tipos primitivos de datos:

```
double[ ] x = new double[100]; // también double x[]
```

Definición de matrices de referencias a objetos:

```
String[ ] s = new String[3];
```

Acceso a elementos de una matriz:

```
x[2]=5.3;
```

```
s[0]= new String("Martin"); s[0] = "Maritn";
```

Inicialización:

```
int v[ ] = {4, 1, 6, 3, 42, 51, 61, 27, 98, 119};
```

```
String [ ] dias ={"lunes", "martes"};
```

```
System.out.println("Hoy es "+ dias[1]); // "Hoy es martes"
```


Arrays (II)

- Los índices de una matriz van desde cero hasta el tamaño menos uno.
- El tamaño se puede conocer con la propiedad length.

Ejemplo de bucle que recorre todos los elementos de una matriz:

```
int [] matriz ={53, 4, 43, 54};  
for( int i=0; i<matriz.length; i++) {  
    System.out.println(matriz[i]);  
}
```

- Se pueden crear matrices anónimas (por ejemplo: como parámetro de un metodo.

```
obj.metodo( new String[ ] {"uno", "dos", "tres"} );
```


Arrays (III)

Una matriz bidimensional es un vector de referencias a los vectores fila. Así pues, cada fila podría tener un número de elementos diferente.

Una matriz se puede crear directamente en la forma:

```
int [ ][ ] mat = new int[3][4];
```

O dando los siguientes pasos:

1. Crear la referencia (con un doble corchete): `int[][] mat;`
2. Crear el vector de referencias a las filas:

```
mat = new int[3][ ];
```

3. Crear los vectores correspondientes a las filas:

```
for (int i=0; i<3; i++) mat[i] = new int[4];
```

4. Asignar valores a cada elemento

```
for (int i=0; i<3; i++) for (int j=0; j<4; j++)  
    mat[i][j] = 2*i*j;
```

Implementación de la Programación orientada a objetos

POO – Clases (I)

Estructura de una clase

```
class NombreClase {  
    // Cuerpo de la clase.  
}
```

Las clases se componen de atributos y métodos

```
class TrianguloRectangulo {  
    double c1; double c2; // atributos  
  
    void asignarValores (double valor1, double valor2) {  
        c1 = valor1; c2 = valor2;  
    }  
  
    double hipotenusa() {  
        return Math.sqrt(c1*c1 + c2*c2);  
    }  
}
```

POO – Clases (II)

Los programas de Java están estructurados en clases. Las clases son la definición de las características de los diferentes tipos de objetos que vamos a crear (instanciar).

```
class TestTrianguloRectangulo{  
    public static void main(String [] args){  
        TrianguloRectangulo tr= new TrianguloRectangulo();  
        tr.asignarValores(2,6);  
        System.out.println("El triangulo de catetos " + tr.c1 + " y " + tr.c2 + " tiene una "  
            + "hipotenusa de valor "+ tr.hipotenusa());  
    }  
}
```

POO – Herencia (I)

Nos permite aprovechar los atributos y métodos de una clase, para crear otra que “extienda” su funcionalidad.

```
class Figura1 {  
    double x;  
    double y;  
  
    void inicializar(double x, double y) {  
        this.x=x;  
        this.y=y;  
    }  
  
    double area() {  
        return 0;  
    }  
}
```

POO – Herencia (II)

Todas las clases que no heredan de ninguna clase, heredan de la clase Object, de forma que todas las clases de forma directa o indirecta heredan de Object.

```
class Rectangulo1 extends Figura1 {  
    double l1, l2;  
    void inicializar(double x, double y, double l1, double l2){  
        super.inicializar(x, y); // super = la clase de la cual se hereda  
        this.l1 = l1; this.l2 = l2;. // this representa al objeto actual.  
    }  
    double area() { return l1 * l2; }  
    public String toString(){  
        return "Rect["+x+" "+y+"), "+l1+" "+l2+"]";  
    }  
}
```

```
class TestFiguras1 {  
    public static void main(String args[]) {  
        Rectangulo1 r1 = new Rectangulo1();  
        r1.inicializar(2,3,5,7);  
        System.out.println(r1 + " tiene un area = " + r1.area());  
    }  
}
```

POO – Herencia (III)

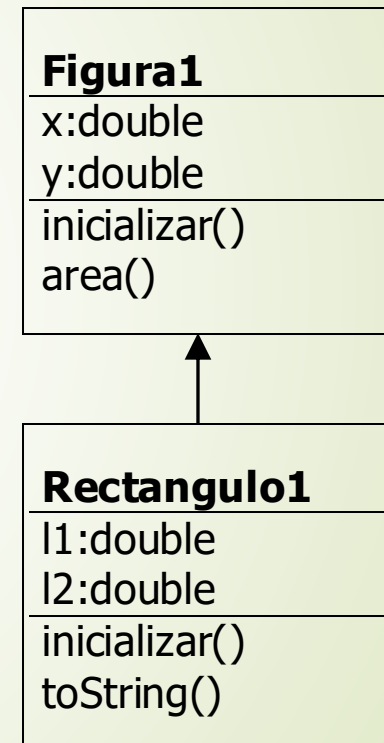
Diferentes formas de llamar a las clases denotando su relación de herencia.

Figura1	Rectangulo1
Superclase	Subclase
Clase Base	Clase Derivada
"Madre"	"Hija"

Rectangulo1 hereda de Figura1

Rectangulo1 deriva de Figura1

Notacion UML



POO – Constructores (I)

Son métodos que se ejecutan con el operador new al crear cada ejemplar de la clase. Tienen el nombre de la clase y no retornan ningún valor.

```
public class Figura2 {  
    double x;  
    double y;  
    public Figura2() {}  
    public Figura2(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    public double area() {  
        return 0;  
    }  
    public String toString() {  
        return "Fig(" + x + "," + y + ")";  
    }  
}
```


POO – Constructores (II)

En la primera línea de un constructor, debe ejecutarse uno de los constructores de la superclase, explícita o implícitamente u otro constructor de la misma clase.

```
public class Rectangulo2 extends Figura2 {
    double l1, l2;
    public Rectangulo2(double x, double y, double l1, double l2){
        super(x, y);
        this.l1 = l1;
        this.l2 = l2;
    }
    public double area() {
        return l1 * l2;
    }
    public String toString(){
        return "Rect[("+x + ","+y+"), "+l1 + ","+l2+"]";
    }
}
```

POO – Constructores (III)

```
public class Circulo2 extends Figura2{
    double r;

    public Circulo2(double x, double y, double r){
        super(x,y);
        this.r = r;
    }

    public double area() {
        return Math.PI * r * r;
    }

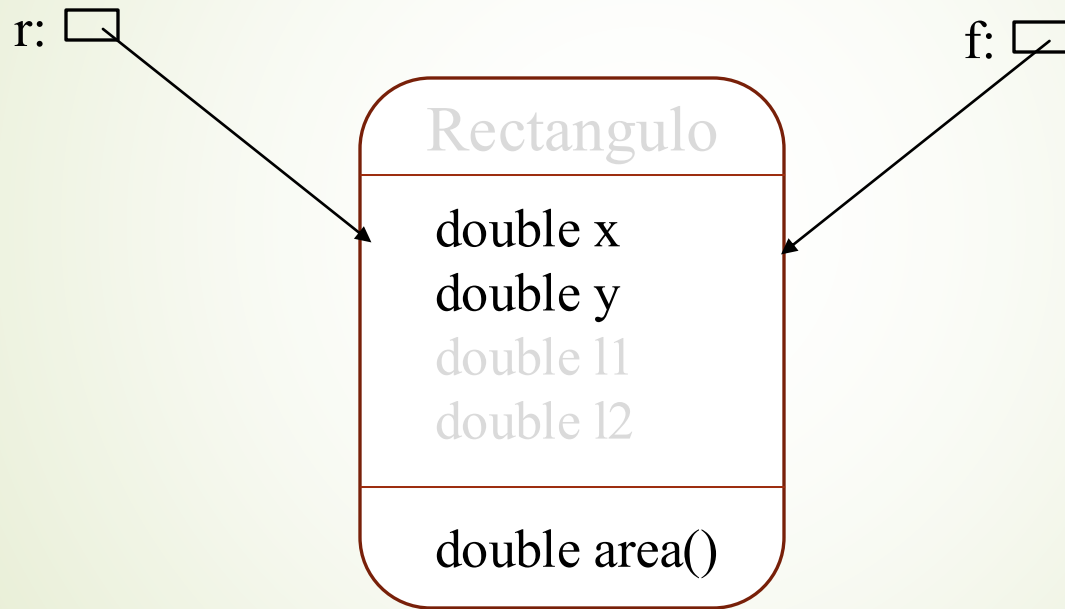
    public String toString() {
        return "Cir[(“ + x + “,” + y + “), “ + r + “]”;
    }
}
```

POO – Constructores (IV)

```
class TestFiguras2{  
    public static void main (String args[]){  
        Circulo2 c = new Circulo2(4, 3, 2);  
        Rectangulo2 r = new Rectangulo2(3, 3, 1, 2);  
        System.out.println(c + " tiene un area = " + c.area());  
        System.out.println(r + " tiene un area = " + r.area());  
    }  
}
```

Polimorfismo

```
Rectangulo r = new Rectangulo(...);  
Figura f = r;
```



Polimorfismo

El polimorfismo nos permite tratar de una forma común a diferentes clases que heredan de una misma superclase.

```
public double sumaAreas(Figura2[] figs) {  
    double areas = 0;  
    for (int i = 0; i < figs.length; i++) {  
        areas += figs[i].area();  
    }  
    return areas;  
}
```

Conversión de Referencias

Conversiones implícitas: Se puede asignar a una referencia de una determinada clase, otra referencia de una subclase.

```
Rectangulo2 r = new Rectangulo2(2,3,5,8);
```

```
Figura2 f = r; // Conversión implícita de la subclase
```

Conversiones explícitas (type-cast): Se puede asignar a una referencia de una determinada clase, otra referencia de una superclase realizando el type-cast:

```
Figura2 f = new Rectangulo2(1,9,4,5);
```

```
Rectangulo2 rr = (Rectangulo2) f; // Casting o conversión explícita
```

instanceof

`instanceof` es un operador que nos permite comprobar si una referencia apunta a un objeto de una determinada clase

instanceof

Referencia **instanceof** NombreClase;

Retorna true si la variable hace referencia a un objeto de la clase indicada.

Ejemplo:

```
Rectangulo2 r;
```

```
Figura2 f = new Rectangulo2(3,4,5,8);           //Con f no puedo acceder a  
                                                  //l1 y l2 de este rectangulo
```

```
if (f instanceof Rectangulo2) {  
    r=(Rectangulo2) f; // Tenemos la garantía que no producirá  
                      //error en tiempo de ejecución.  
    r.l1=22;  
}
```

Class Object

Class .getClass()

String .toString()

boolean .equals(Object o)

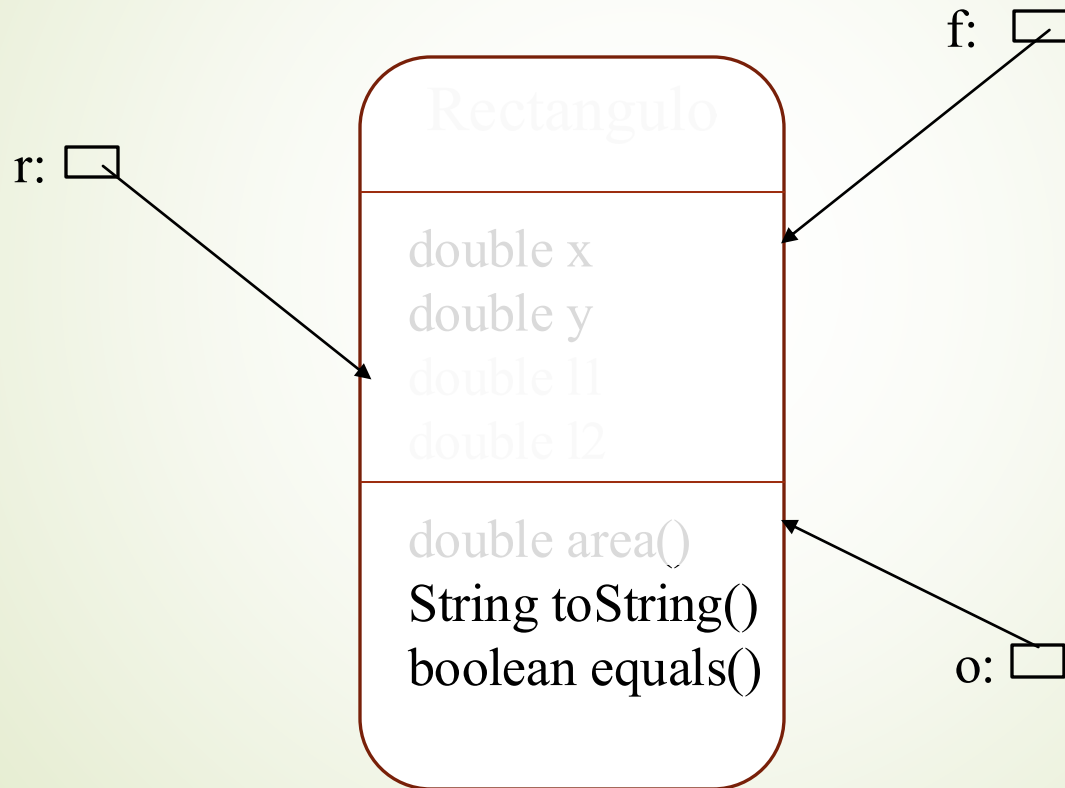
Object .clone()

Polimorfismo

Rectangulo r = new Rectangulo(...);

Figura f = r;

Object o = r;



Método toString()

```
public class MiClase{  
  
    String nombre;  
    int valor;  
  
    //...  
  
    public String toString() {  
        return "[" + nombre + ", " + valor + "];  
    }  
}
```

Método equals - propiedades

- REFLEXIVIDAD ($x=x$)
- SIMETRÍA (si $x=y$, $y=x$)
- TRANSITIVIDAD (si $x=y \wedge y=z$, $x=z$)
- COHERENCIA (lógica de negocio)
- `x.equals(null) => false`

Método equals

```
public class MiClase{

    String nombre;
    int valor;
    //...
    public boolean equals(Object otro) {
        if (this == otro) return true;
        if (otro == null) return false;
        if (getClass() != otro.getClass()) return false;

        MiClase nuevo = (MiClase)otro;
        return this.nombre.equals(nuevo.nombre) &&
               this.valor==nuevo.valor;
    }
}
```

Paquetes (package)

Las clases en java se agrupan en paquetes (package). Para hacer uso de una clase de otro paquete debemos hacer import de la misma.

Para indicar que una clase pertenece a un paquete, la primera sentencia del archivo debe ser: `package nombre_paquete;`

Se pueden crear jerarquias de paquetes separados por puntos
`package paquete1.paquete2.paquete3;`

Los archivos .class de un paquete deben estar en un directorio con el nombre del paquete.

Para ejecutar una clase que pertenece a un paquete, debemos anteponer el nombre del paquete: `java paquete.MiClase`

Modificadores de acceso a Clases

Las clases con el modificador `public` son accesibles desde clases de otros paquetes. Las clases sin modificador `public` solamente son accesibles desde clases del mismo paquete.

- Modificador **public**

La clase es accesible desde toda la aplicación.

- Sin modificador (package)

La clase es accesible sólo desde su paquete.

Modificadores de acceso a miembros

VISIBILIDAD	private	(sin modificador)	protected	public
Desde la misma clase	SI	SI	SI	SI
Desde otra clase o subclase del mismo paquete	NO	SI	SI	SI
Desde una subclase de otro paquete	NO	NO	SI	SI
Desde una clase de otro paquete	NO	NO	NO	SI

Getters y Setters

```
public class Contacto {  
    private int idContacto;  
    private String nombre;  
    private String apellidos;  
    private String apodo;  
  
    public int getIdContacto() {  
        return idContacto;  
    }  
    public void setIdContacto(int idContacto) {  
        this.idContacto = idContacto;  
    }  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    ...  
}
```


Clases Abstractas

- No se pueden instanciar objetos de clases abstractas. Pero si se pueden crear referencias de estas clases, así como heredar de ellas.
- Las clases abstractas pueden tener alguno de sus métodos abstractos. Estos métodos no tienen cuerpo (y terminan en punto y coma). La funcionalidad de estos métodos se indicará en las clases que hereden de la clase abstracta.
- Las clases que hereden de la clase abstracta, si no son abstractas, deberán obligatoriamente implementar (añadirles el código) los métodos abstractos de la superclase.
- Las clases abstractas nos permiten agrupar varias clases bajo sus características comunes de forma que podremos tratar de una forma común, mediante referencias de la clase abstracta, objetos de diferentes subclases. (Polimorfismo)

Atributos static

- De cada atributo estático se puede decir que hay una variable por clase en lugar de una por objeto.
- Los atributos y métodos static pueden ser accedidos sin necesidad de crear una instancia para ello, basta con poner el nombre de la clase:
- `double a = 2 * Math.sin(Math.PI / 6);`

Interfaces

Son estructuras similares a las clases abstractas con todos sus métodos implícitamente abstractos y todos sus atributos implícitamente finales y estáticos (Constantes).

```
interface DosDimensiones {  
    int ROJO = 3;           // equivale a un static final de una Clase  
    public void dibujar(); // equivale a un abstract de una Clase  
}
```

Las clases que implementen la interfaz, “heredan” los atributos y están obligadas a sobre-escribir los métodos de la I.

Interfaces

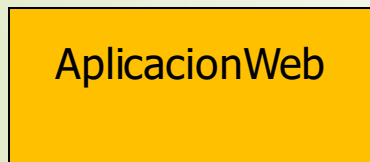
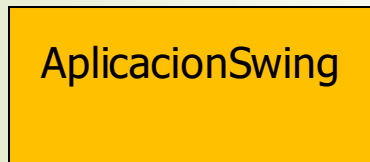
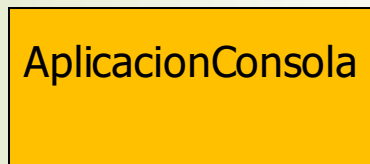
```
class Rectangulo extends Figura3 implements DosDimensiones {
    double l1, l2;
    Rectangulo(double x, double y, double l1, double l2) {
        super(x,y);
        this.l1=l1; this.l2=l2;
    }
    public void dibujar() {
        System.out.println("-----");
        System.out.println("|               |");
        System.out.println("|               |");
        System.out.println("-----");
    }
    double area() {
        return (l1*l2)/2;
    }
}
```

Interfaces

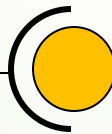
Mediante una interface podemos indicar los mecanismos de comunicación entre dos subsistemas, sin indicar su estructura interna. De esta forma podemos tener deferentes implementaciones de la misma.

Ejemplo: La interface AgendaInterface que modela la información de la aplicación (Capa de Datos). Esta puede implementarse para guardar los datos en memoria (AgendaVector), en un archivo (AgendaArchivo) o en una Bbdd (AgendaBbdd). Asi mismo podemos tener varias aplicaciones (Capa de presentación) que hagan uso de objetos que implementen la misma interface.

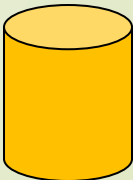
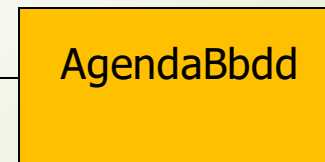
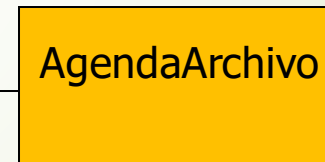
Capa de Presentación



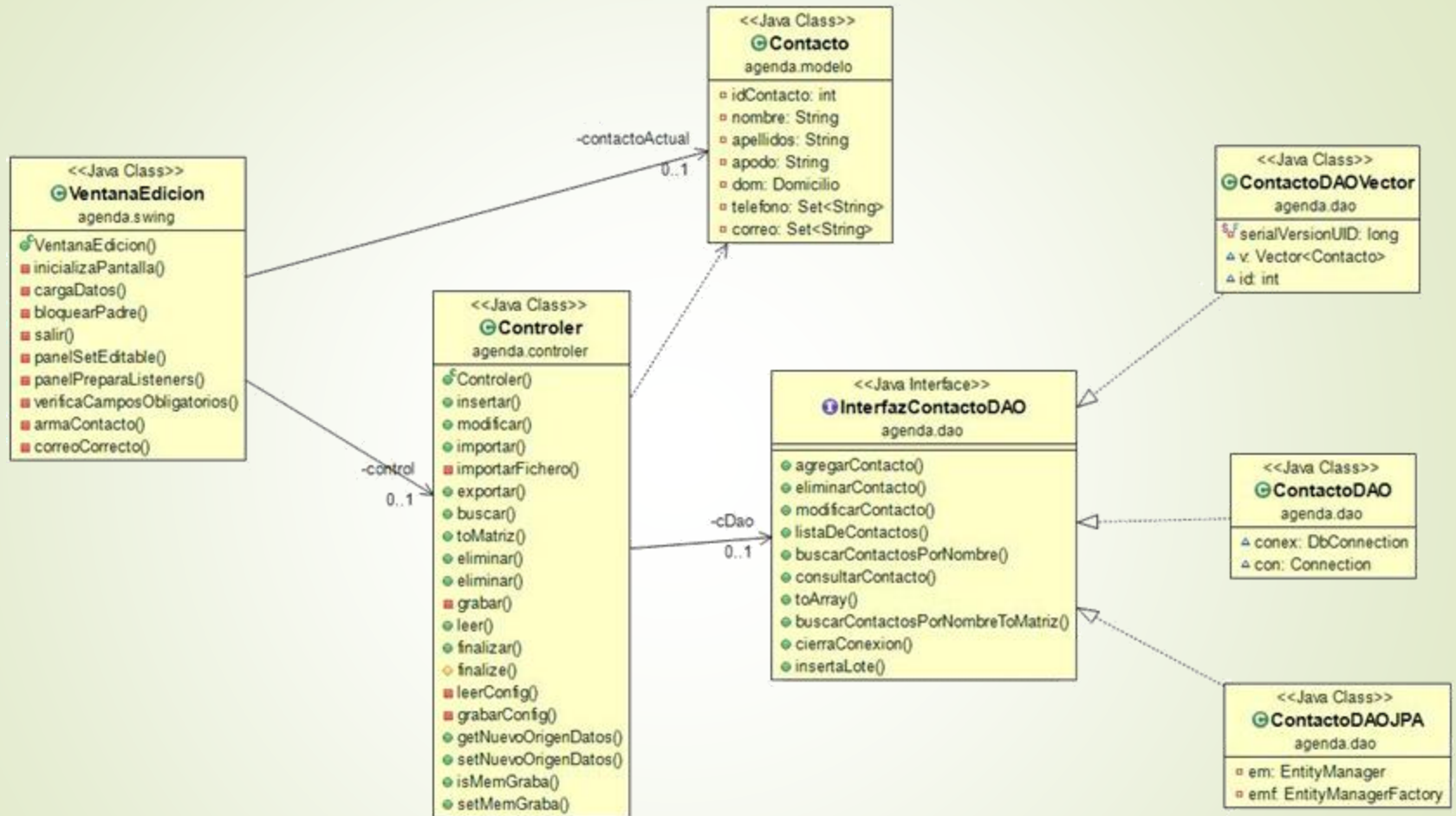
AgendaInterface



Capa de Datos



Interfaces



Interfaces

```
public interface InterfazContactoDAO {  
  
    public void agregarContacto(Contacto contacto);  
  
    public boolean eliminarContacto(int IdContacto);  
  
    public void modificarContacto(Contacto contacto);  
  
    public List<Contacto> listaDeContactos();  
  
    public List<Contacto> buscarContactosPorNombre(String palabra);  
  
    public Contacto consultarContacto(int idContacto);  
  
    //...  
}
```


Interfaces

Java 1.8

Métodos default

Son métodos con cuerpo.
Pueden sobre-escribirlo las
clases que implementan la
interfaz.

Métodos estáticos

No pueden sobre-escribirse.
Sólo se acceden desde la interfaz

Modificador final

Indica que ya no se puede modificar (que es definitivo). En un atributo, no se puede cambiar su valor (es una constante), en un método, que no se puede reescribir en las subclases.

Si una clase se declara con el modificador final, ya no se podrá heredar de ella. Esto ocurre con la clase String que está declarada como “final class String”

```
class HeredaString extends String{  
    // No se puede compilar porque la clase  
    // String está declarada como final.  
}
```

Cuando se quiere definir una constante, generalmente se define como **static final**. Ya que como es constante y con el mismo valor para todos los objetos, es suficiente con una sola copia de la variable. Las constantes en java se suelen nombrar con mayúsculas y si hay cambio de palabra se separa con guión bajo.

```
static final double EURO_PESETAS=166.386;  
EURO_PESETAS=200; // Error de compilación, no se puede cambiar.
```

Clases Anidadas

Las clases anidadas son clases definidas dentro de otras. Podemos distinguir tres tipos **internas**, **locales** y **anónimas**.

Clases **internas**

La clase interna se define como un miembro (no está dentro de ningún método) de la clase externa.

La clase interna tiene acceso a los miembros de la clase externa.

Para acceder desde una clase interna al objeto de la clase externa (this de la clase externa) se emplea:

NombreClaseExterna.this

Clases Anidadas (II)

Clases **locales**

Se definen dentro de un método y solamente se usan dentro de él.

Clases **anónimas**

Al instanciar un objeto de una determinada clase, se puede heredar de esa clase en ese mismo punto en el que se instancia el objeto. Esta será una clase anónima. Las clases anónimas no tienen constructor.

Tipo enum

- Tipo de dato especial. Incluidos a partir de Java 5
- Los tipos enumerados permiten restringir el contenido de una variable a una serie de valores predefinidos.
- Permiten definir tipos de datos personalizados a través de una serie de constantes.
- Se pueden definir fuera o dentro de una clase.
- Como se definen de forma similar a una clase, se les puede añadir métodos, atributos y constructores.

Tipo enum simples

```
public class EjemploEnum {  
  
    enum Dia {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO}  
  
    public static void main(String[] args) {  
  
        Dia d = Dia.MARTES;  
        System.out.println("Dia " + (d.ordinal()+1) + ": " + d);  
  
    }  
}
```

Dia 2: MARTES

enum

```
public enum Puesto{
    PILIER_IZQ(1, "Pilier", "Prop"), PILIER_DER(3, "Pilier", "Prop"),
    TALONADOR(2, "Talonador", "Hooker"), SEGUNDA_4(4, "Segunda", "Second Row");

    private final int num;
    private final String nombre;
    private final String ingles;

    private Puesto(int num, String nombre, String ingles){
        this.num = num;
        this.nombre = nombre;
        this.ingles = ingles;
    }

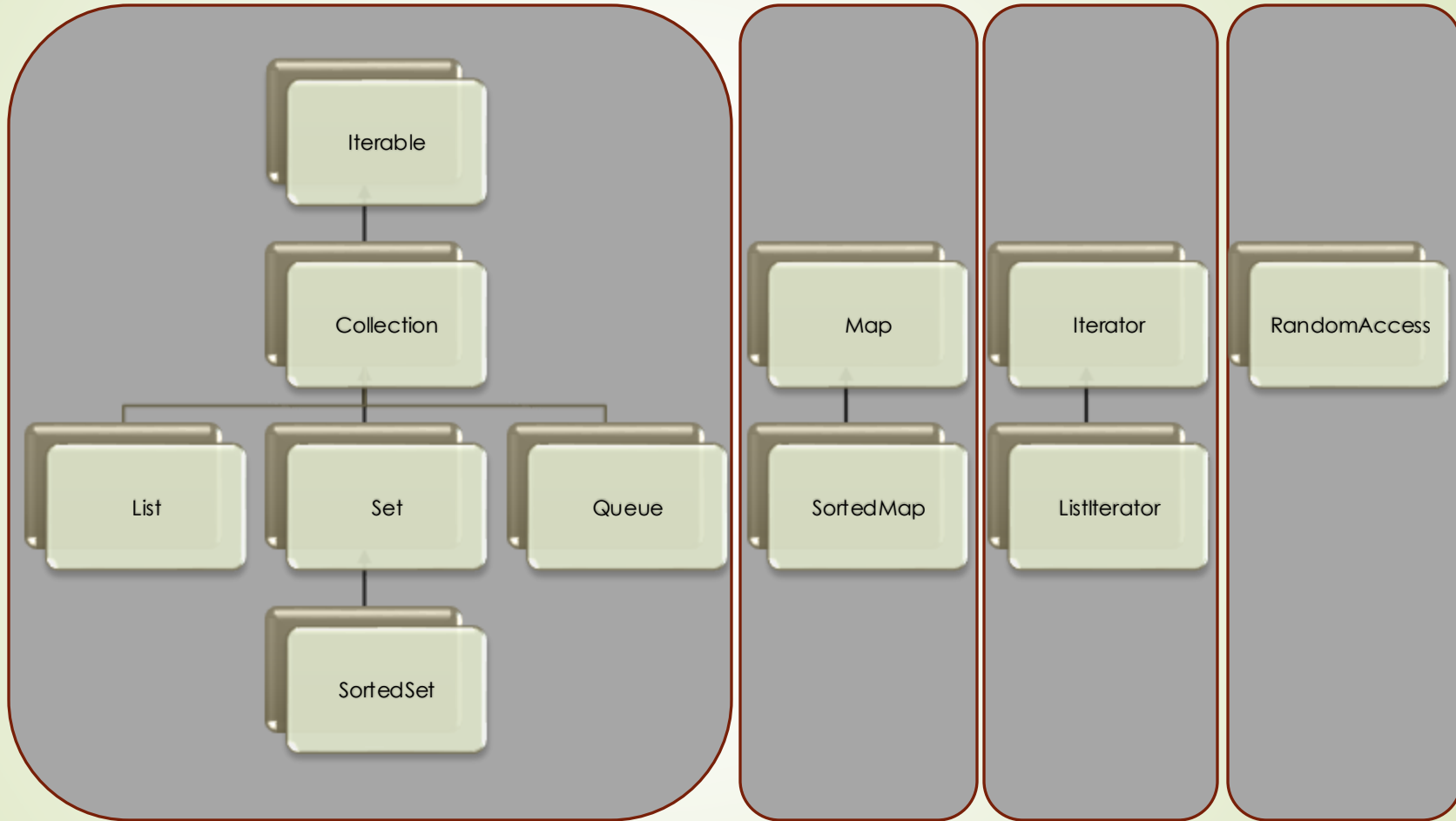
    public int getNum() {
        return num;
    }

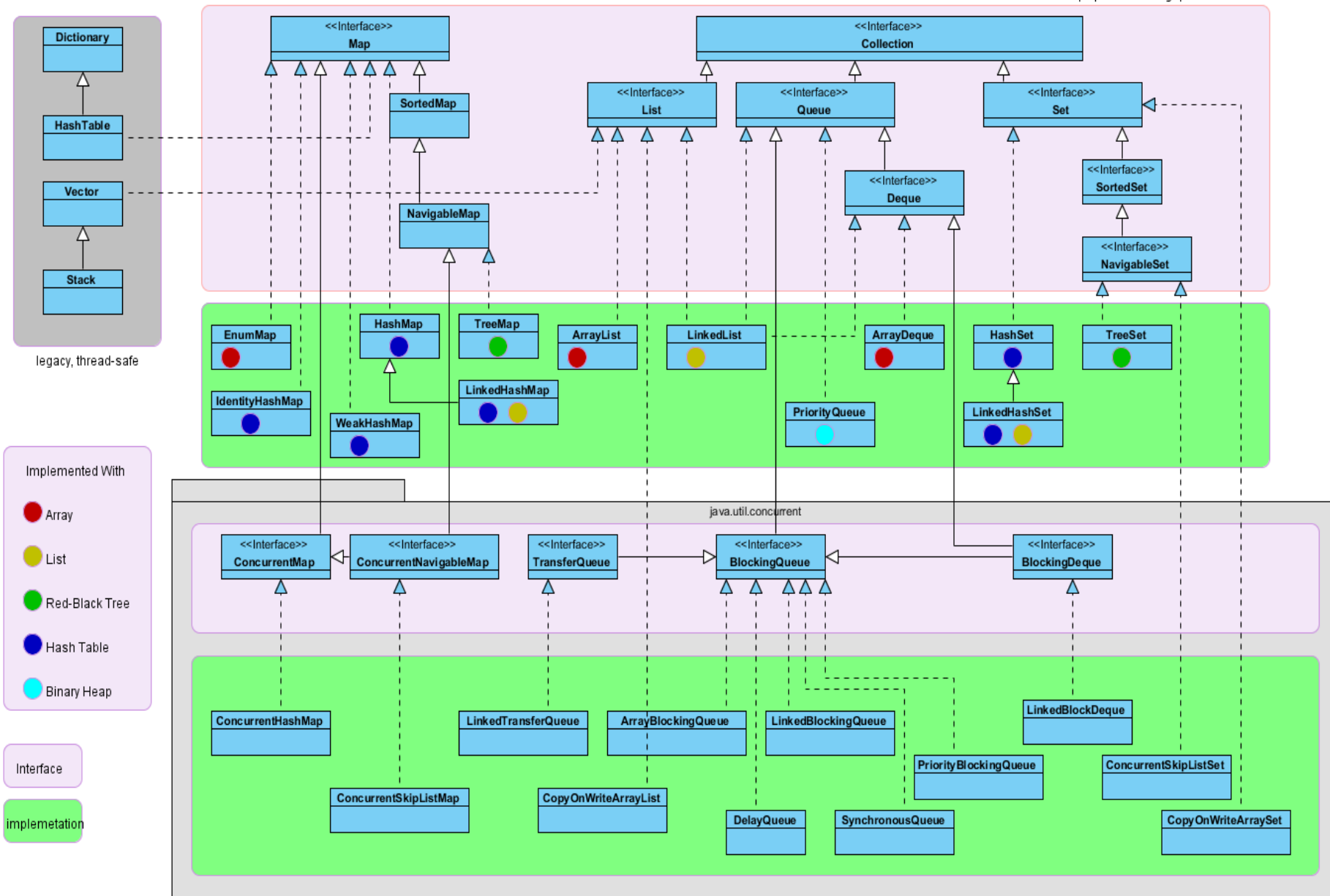
    public String getNombre() {
        return nombre;
    }

    public String getIngles() {
        return ingles;
    }
}
```

Las Colecciones

Las Interfaces

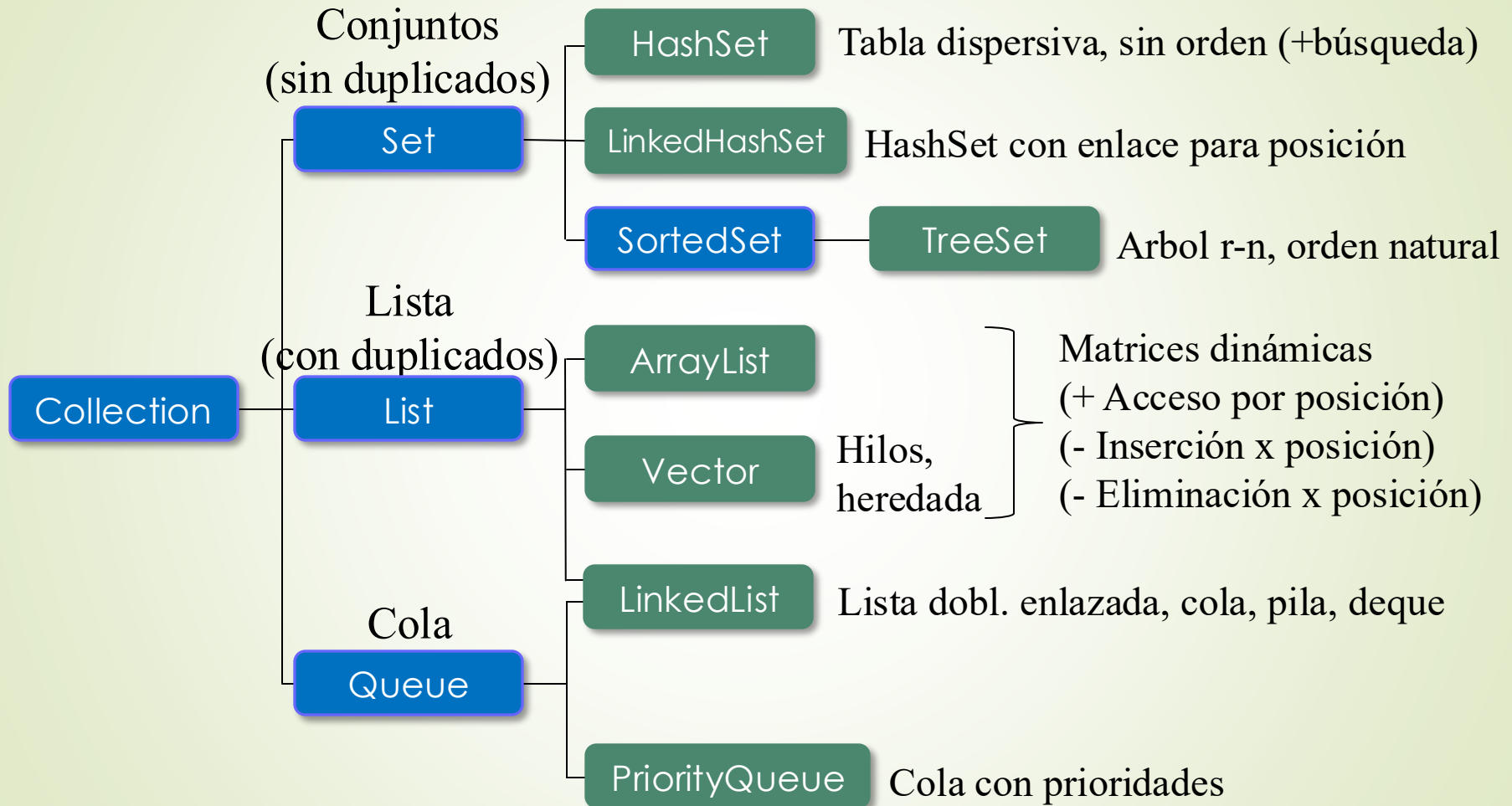




Collections

Interface

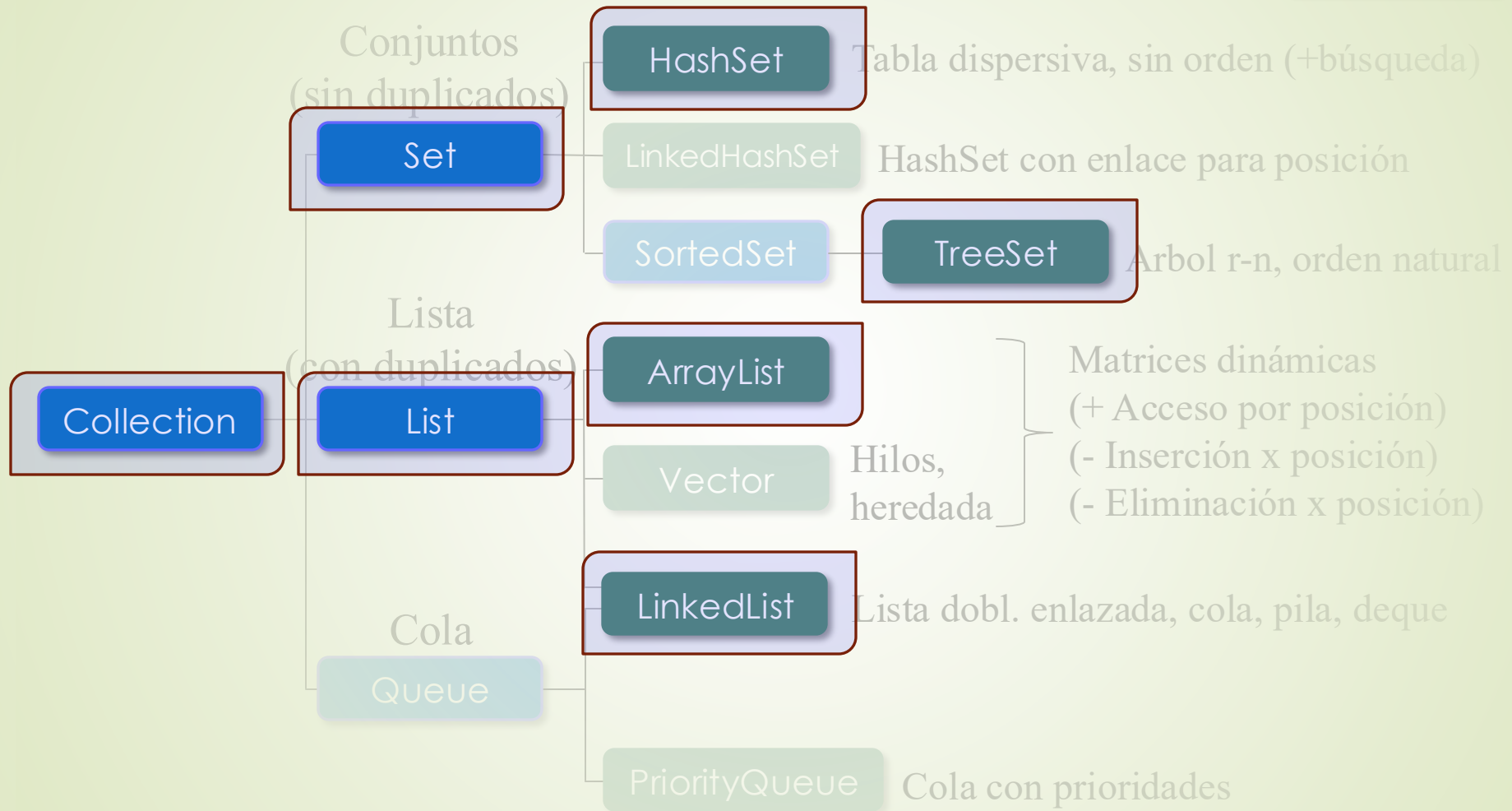
Clase



Collections

Interface

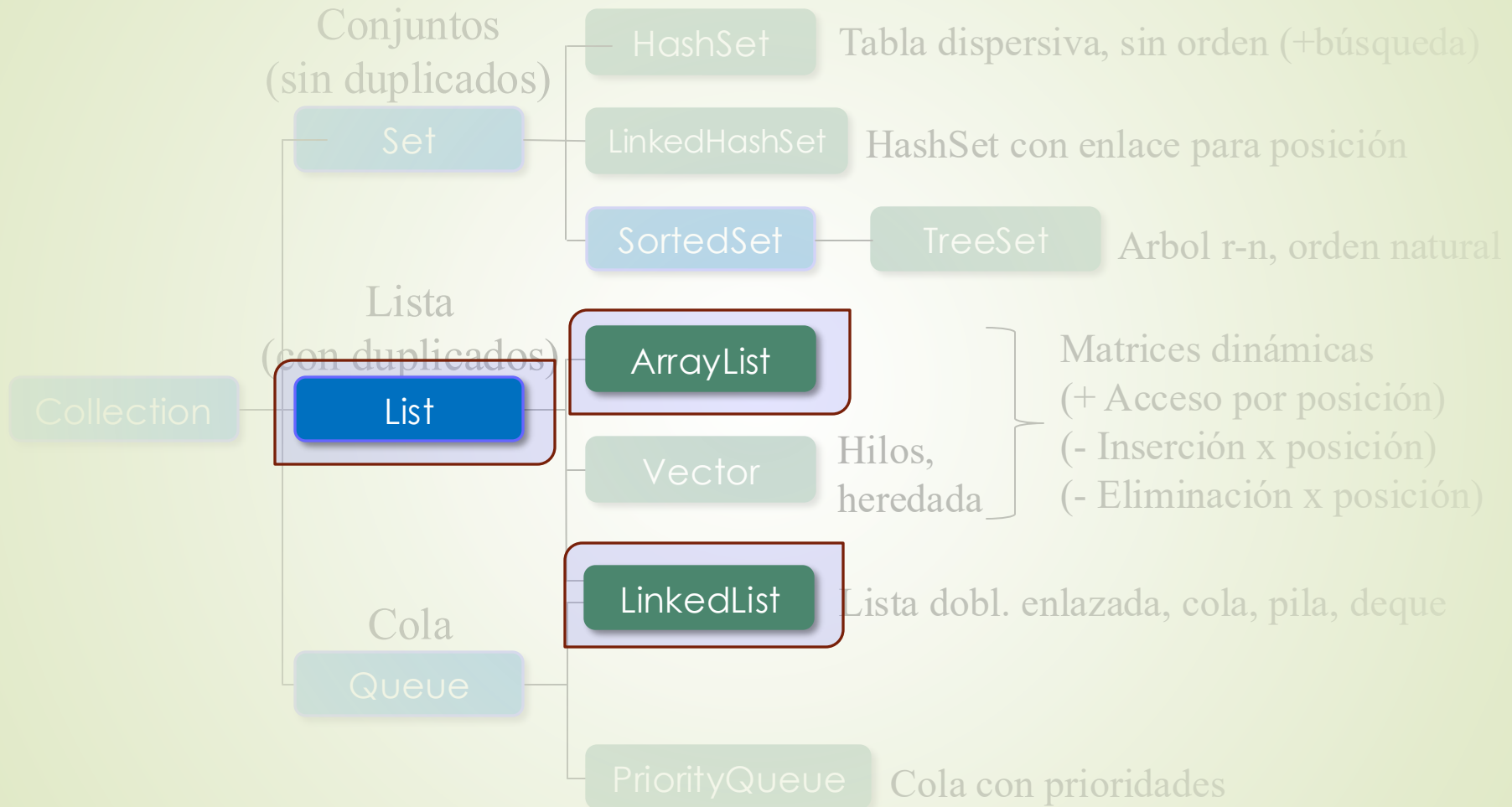
Clase



List

Interface

Clase



Interface List (I)

Permite crear matrices dinámicas (Que pueden cambiar su numero de elementos). Clases que implementan la interface List: ArrayList, LinkedList, Vector...

void add(Object obj)	Añade un objeto al final
Object get(int posicion)	Retorna el objeto que hay en la posicion indicada como parámetro.
int size()	Devuelve el número de elementos
boolean remove (Object obj) Object remove (int indice)	Elimina el primer objeto con la referencia suministrada o en la posición indicada por el indice. Si no encuentra el objeto retorna false. Si se indica el indice se retorna el objeto eliminado de la lista.
void clear ()	Quita todos los elementos de la colección.
boolean isEmpty()	Devuelve true si no tiene elementos
boolean contains(Object obj)	Retorna true si el objeto suministrado está en la lista.
int indexOf(Object obj)	Retorna la posición donde está el objeto. -1 si no encontrado.

Interface List (II)

<code>Iterator iterator()</code>	Retorna un Iterator con todos los elementos de la lista.
<code>Object set(int indice, Object obj)</code>	Remplaza el objeto que hay en la posición indicada por indice con el objeto suministrado. Retornando una referencia al objeto sustituido.
<code>T[] toArray(T[] a)</code>	Retorna una matriz con todos los objetos de la lista, pero la matriz es del mismo tipo de la matriz pasada como parámetro.
<code>Object [] toArray()</code>	Retorna una matriz de Object con todos los elementos del la lista.
<code>boolean addAll(Collection<E> c)</code>	Agrega todos los elementos de c, en el orden que determina el Iterator
<code>void add(int index, Object obj)</code>	Inserte un objeto en la posición indicada, incrementando la posición de los objetos que hay detrás.

Clase ArrayList

Implementa List. Permite crear matrices dinámicas (Que pueden cambiar su numero de elementos). Como implementa la interface List solamente se indican los métodos adicionales.

ArrayList()	Lista vacía con capacidad 10
ArrayList(Collection <E> c)	Lista con los elementos que contiene c (orden Iterator)
ArrayList(int capacidadInicial)	Lista vacía con capacidad capacidadInicial

Object trimToSize()	Ajusta la capacidad al tamaño actual (size)
----------------------------	---

Ejemplo uso ArrayList

```
ArrayList<String> al = new ArrayList<String>();  
al.add("Miercoles");  
al.add("Lunes");  
al.add("Martes");  
  
System.out.println(al.get(0));  
  
System.out.println("El elemento 2 es: " + al.get(2));  
  
for (int i = 0; i < al.size(); i++) {  
    System.out.println(al.get(i));  
}  
  
System.out.println(al.size());
```


Ejemplo uso List

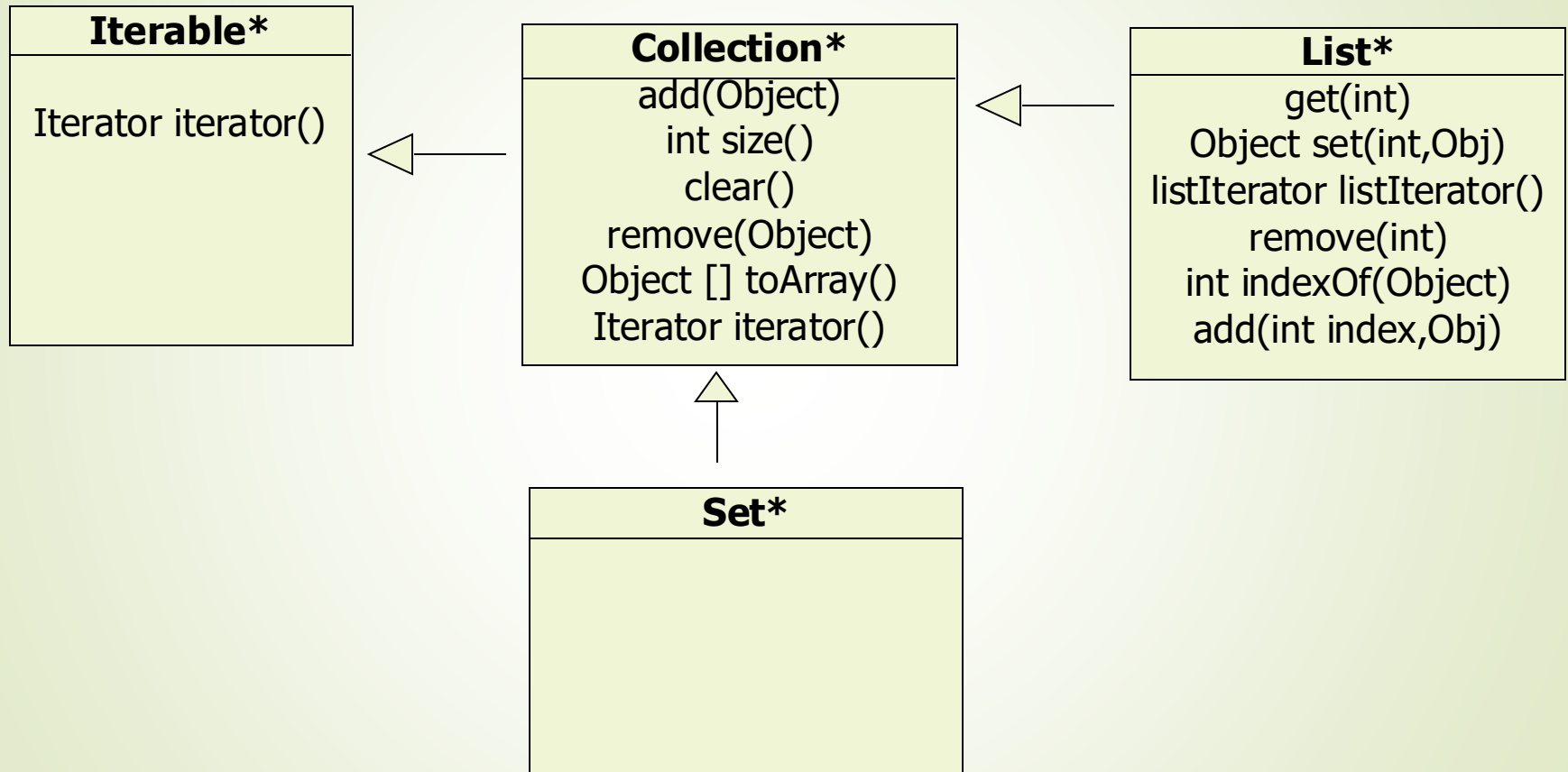
```
List<String> al = new ArrayList<String>();  
al.add("Miercoles");  
al.add("Lunes");  
al.add("Martes");  
  
System.out.println(al.get(0));  
  
System.out.println("El elemento 2 es: " + al.get(2));  
  
for (int i = 0; i < al.size(); i++) {  
    System.out.println(al.get(i));  
}  
  
System.out.println(al.size());
```

Iterator

Proporcionan métodos para recorrer secuencialmente los elementos de una colección de objetos.

Iterator	
boolean hasNext()	Retorna true si hay más elementos a continuación.
Object next()	Retorna una referencia al siguiente objeto de la colección a la que está conectado el Iterator
void remove()	Elimina, de la colección a la que está conectado, el elemento que se acaba de acceder.

Collection, Set e Iterable



Iteradores (Iterator)

boolean .hasNext()
Object .next()
Object .remove()

```
Iterator it = ....iterator();  
  
while (it.hasNext()) {  
    Object dato = it.next();  
    ...trabajar con dato...  
}
```

Ejemplo uso List (Iterator)

```
List<String> al = new ArrayList<String>();  
al.add("Miercoles");  
al.add("Lunes");  
al.add("Martes");  
  
System.out.println(al.get(0));  
  
System.out.println("El elemento 2 es: " + al.get(2));  
  
Iterator<String> it = al.iterator();  
while (it.hasNext()) {  
    System.out.println(it.next());  
}  
  
System.out.println(al.size());
```

for each (o for extendido)

```
for (tipo varTemp : colección){  
    varTem.XXX  
}
```

```
ArrayList<Figura> lista = new ...;  
Iterator it = lista.iterator();
```

```
while (it.hasNext()) {  
    Figura f = it.next();  
    f.area();  
    ...  
}
```

```
ArrayList<Figura> lista = new ...;  
  
for (Figura f : lista) {  
    f.area();  
    ...  
}
```

Ejemplo uso List (foreach)

```
List<String> al = new ArrayList<String>();  
al.add("Miercoles");  
al.add("Lunes");  
al.add("Martes");  
  
System.out.println(al.get(0));  
  
System.out.println("El elemento 2 es: " + al.get(2));  
  
for (String actual : al) {  
    System.out.println(actual);  
}  
  
System.out.println(al.size());
```

Clase LinkedList

Almacena objetos en una lista doblemente enlazada.

Implementa la interface List.

void add (int indice, Object elem)	Inserta un elemento en la posición indicada por el índice. Los elementos que había en esa posición y siguientes se desplazan una posición.
boolean add (Object elem) void addLast (Object elem)	Inserta un elemento al final de la lista.
void addFirst (Object elem)	Inserta un elemento al principio de la lista.
Object set (int indice, Object elem)	Sustituye el elemento ubicado en la posición indicada por el índice con el objeto suministrado. Retorna una referencia al objeto sustituido.
Object[] toArray ()	Retorna una matriz con los objetos de la lista.
Object remove (int index) boolean remove (Object elem)	Elimina un elemento de la lista, bien por su índice o por la referencia al objeto.
ListIterator listIterator (int index)	Retorna un ListIterator enlazado a los elementos almacenados a partir del elemento con índice indicado. ListIterator es un Iterator que además tiene métodos para retroceder (hasPrevious() y previous())

Clase Stack

Los objetos almacenados se rigen por la filosofía de una pila que sigue la regla: “El último en entrar es el primero en salir”.

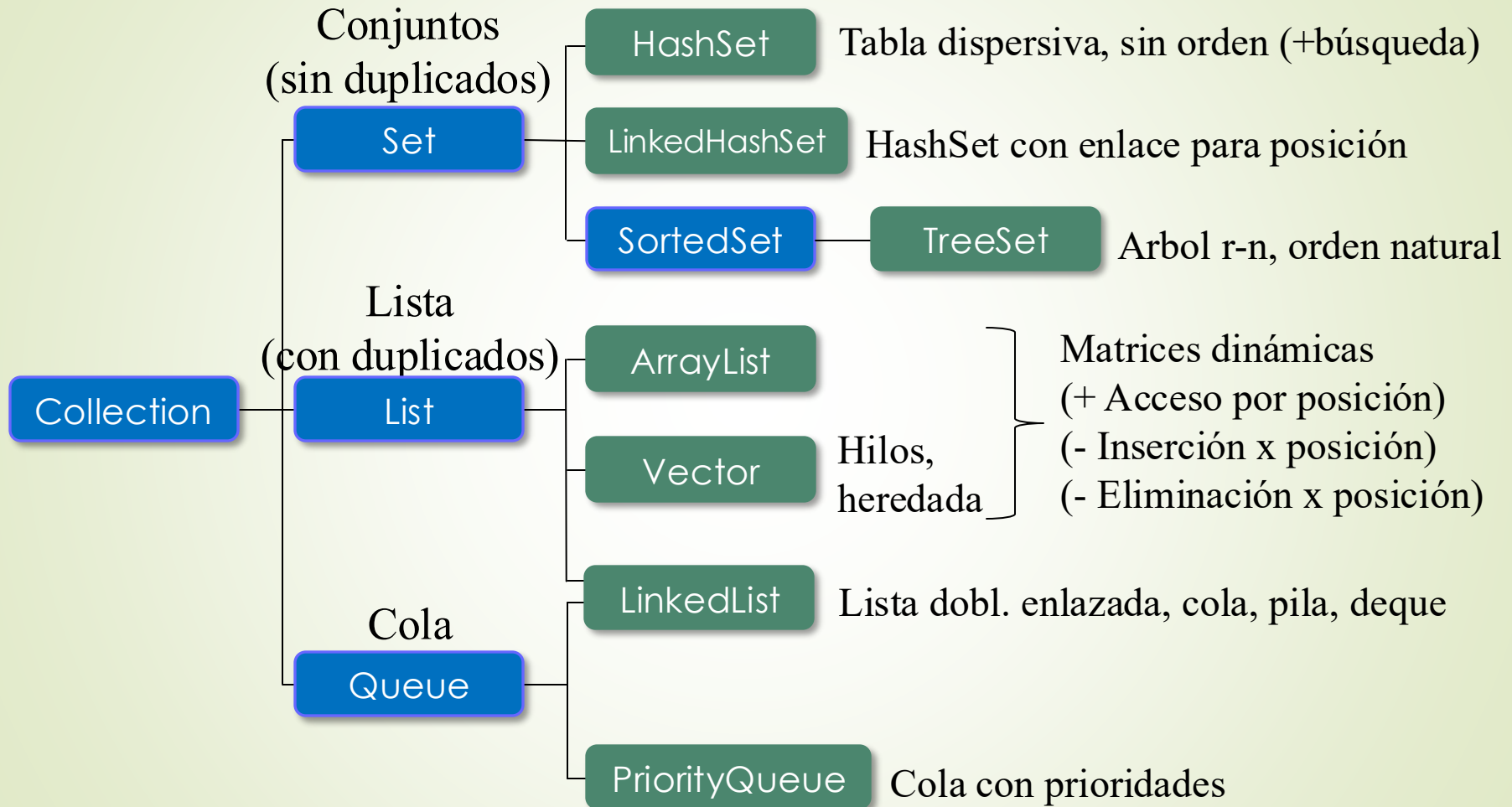
Object push (Object item)	Inserta un elemento en la pila.(Retorna la referencia insertada)
Object pop ()	Extrae el último elemento insertado en la pila.
Object peek ()	Retorna el último elemento de la pila, sin sacarlo de esta.
boolean empty ()	Indica si la pila está vacía.(Hace lo mismo que isEmpty, heredado de Vector)
int size ()	Retorna el número de elementos de la pila.

La clase Stack hereda de Vector, por lo tanto, tiene además de los métodos indicados, los métodos de Vector y de List.

Collections

Interface

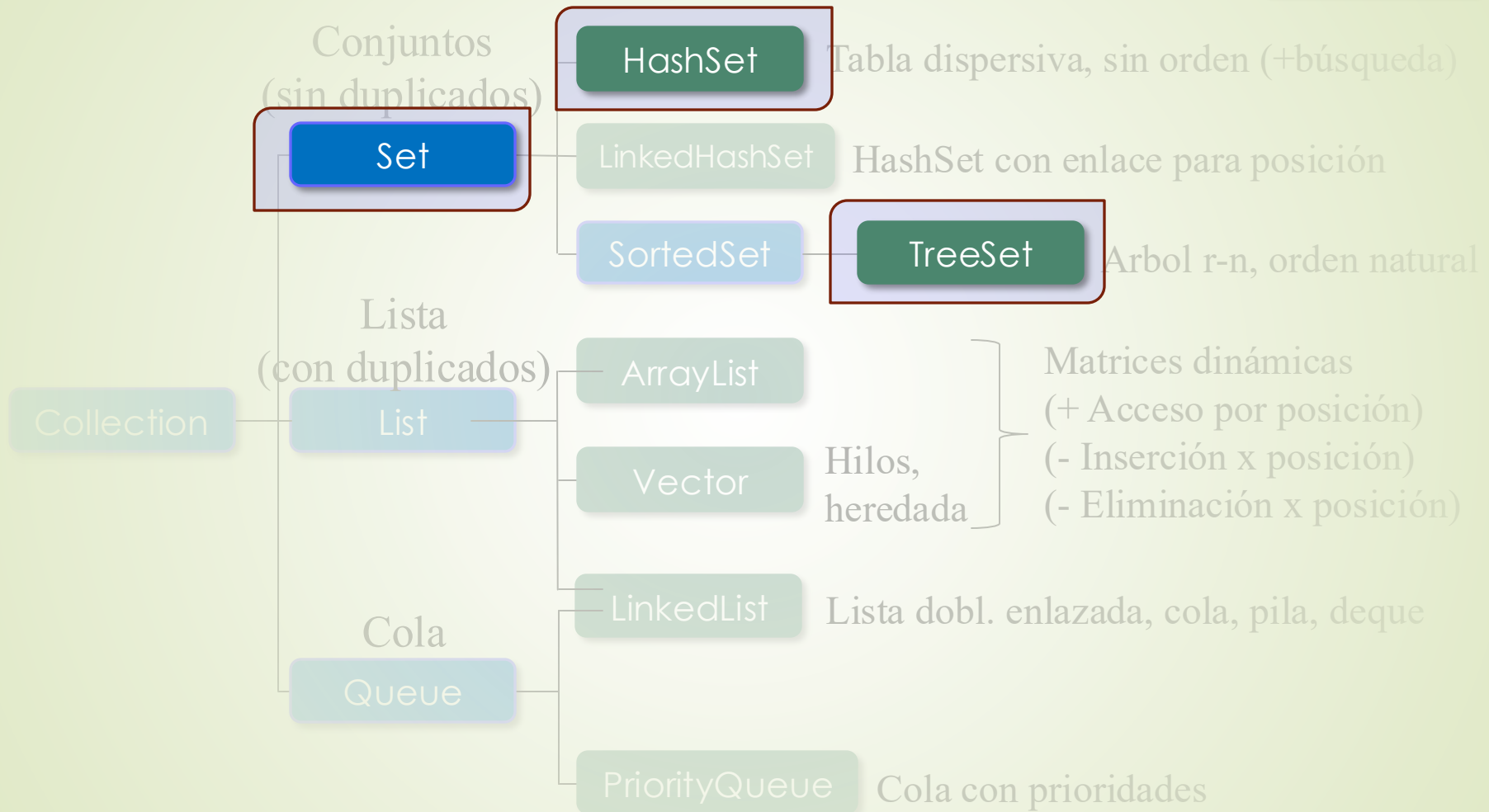
Clase



Set (Conjuntos)

Interface

Clase



Interfaces Set

Set es un conjunto. Set no tiene elementos duplicados. Los métodos son los de Collection

Object add (Object valor)	Añade un objeto al final
boolean contains (Object valor)	Indica si contiene o no un objeto
boolean isEmpty ()	Devuelve true si no tiene elementos
Iterator iterator ()	Retorna un iterator para recorrer los objetos que contiene.
Object [] toArray ()	Retorna una matriz con todos los objetos que contiene.

Clase HashSet

Almacena objetos, **sin importar el orden** y sin posibilidad de repetir elementos.

- Conjuntos Dispersivos
- Implementa conjuntos mediante Tablas Dispersivas
- Utiliza en método hashCode

HashSet()	Crea un conjunto
HashSet(Collection <E> c)	Conjunto con los elementos que contiene c
HashSet(int capacidadInicial)	Conjunto vacío con una tabla de tamaño capacidadInicial
HashSet(int capacidadInicial, float factorCarga)	Conjunto vacío con una tabla de tamaño capacidadInicial y factor de Carga*

* Factor de Carga: valor entre 0.0 y 1.0 que determina el % de carga para realizar la dispersión

Clase TreeSet

Almacena objetos, **ordenados (orden natural)**.

- Conjunto Ordenado
- Implementa conjuntos mediante árbol rojo-negro
- Extrae los elementos ordenados
- Menos eficiente en búsqueda que Hash pero mucho más que List
- Los objetos que almacena deben implementar la interfaz Comparable (método compareTo(T otro)**

TreeSet()	Crea un conjunto
TreeSet(Collection <E> c)	Conjunto con los elementos que contiene c
TreeSet(Comparator <E> comp)	Conjunto que se ordena de acuerdo a comp*

* Ver Interfaz Comparator

Interfaz Comparator

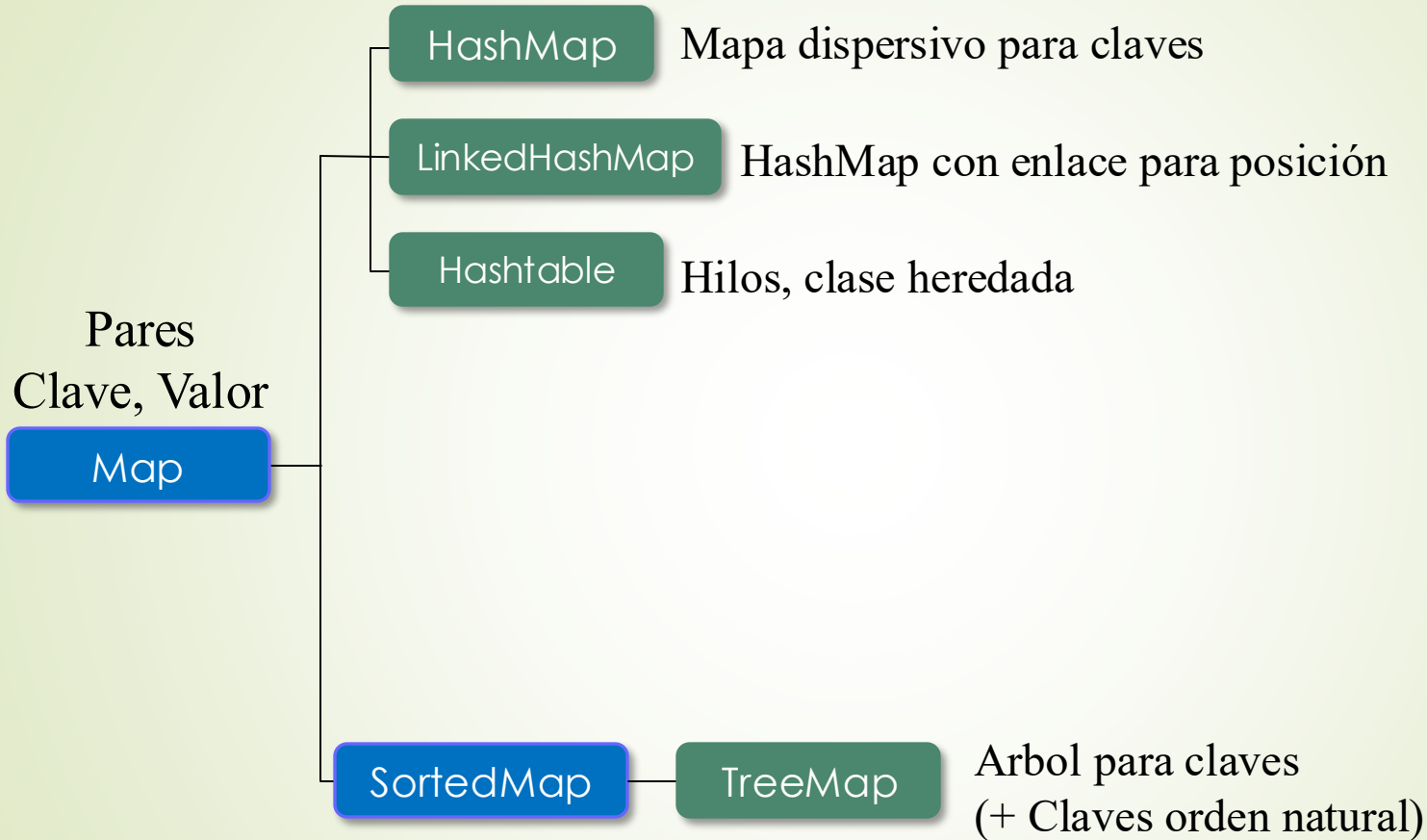
- Un solo método abstracto, el resto son static o default (1.8)
- Se suele crear una clase anónima

```
Set<Empleado> emp = new TreeSet<>(new Comparator<Empleado>() {  
    public int compare(Empleado e1, Empleado e2) {  
        return e1.nombre.compareTo(e2.nombre);  
    }  
});  
...  
  
class Empleado{  
    int id;  
    String nombre;  
    ...  
}
```

Maps

Interface

Clase



Maps

Interface

Clase

HashMap

Mapa dispersivo para claves

LinkedHashMap

HashMap con enlace para posición

Hashtable

Hilos, clase heredada

Pares

Clave, Valor

Map

SortedMap

TreeMap

Arbol para claves
(+ Claves orden natural)

Clase TreeMap

Almacena objetos, asociándole a cada uno una clave, los elementos quedarán ordenados según la clave. (No puede haber repeticiones, es decir dos elementos con la misma clave).

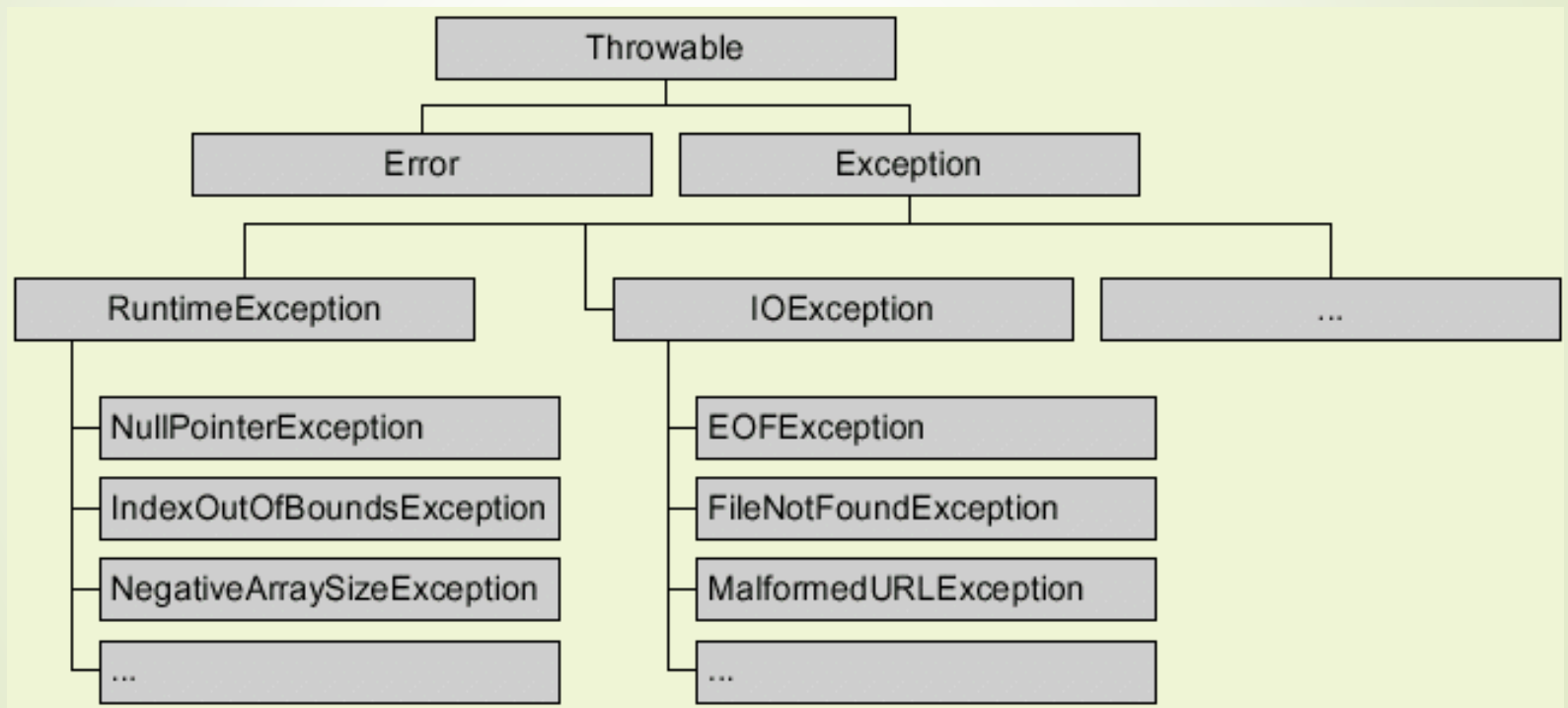
Object put (Object clave, Object elem)	Inserta un elemento (elem) asociándole una clave. Retorna el objeto previamente asociado a la clave (Si no existía retorna null)
Collection values ()	Retorna una Collection con todos los objetos insertados en el mapa.
Set keySet ()	Retorna un Set con las claves del TreeMap (Nota: Set hereda de Collection).
Object get (Object clave)	Retorna el objeto asociado a la clave indicada.

TreeMap()	Constructor sin Parámetros crea un TreeMap cuyos objetos quedarán ordenados por su orden natural, es decir, su método compareTo.
TreeMap(Comparator c)	Este constructor permite suministrarle un objeto que implemente la interface Comparator para indicar le criterio de ordenación. Habrá que sobrescribir el método int compare(Object clave1, Object clave2) de Comparator. De modo que retornará negativo si la primera clave es menor, positivo si es mayor y cero si son iguales.

Excepciones

Excepciones

Diagrama de Clases



Tipos de Excepciones

Error

Normalmente no son tratadas por el programa y es el interprete de java el que realiza esta tarea. (p.e.: Desbordamiento de pila)

RuntimeException (excepciones implícitas, en inglés unchecked)

Son excepciones relacionadas con errores de programación. No se suelen emplear gestores de excepciones para ellas. Las dos causas más frecuentes son intentar acceder a un método o propiedad mediante una referencia null (NullPointerException) e intentar acceder a un elemento fuera del rango de los índices de una matriz (IndexOutOfBoundsException).

Excepciones explícitas (en inglés checked)

Son aquellas de tipo Exception que no son RuntimeException. Este tipo de excepciones es obligatorio tratarlas, bien con un bloque try...catch o bien mediante delegación. Cuando un método de una determinada clase puede producir una excepción de este tipo, se advierte en su declaración mediante la instrucción throws seguido del tipo o tipos de excepción que puede producir.

Gestión de excepciones

Cuando en el código de un método hay una sentencia que puede producir una excepción. Se presentan estas dos alternativas:

Se puede tratar la excepción encerrando la sentencia en un bloque **try...catch** (si en el método sabe qué hacer con ella)

Delegarla en la declaración del método mediante la instrucción **throws**, de modo que cuando se realice una llamada al método se le delega la gestión de la excepción.

Delegación (throws)

En la declaración de un método, *throws* seguido de uno o varios tipos de excepción indica que el método contiene sentencias que pueden producir una excepción y no han sido incluidas en un bloque *try...catch* (no han sido tratadas)

```
public void miRutina() throws IOException {  
    ...  
}
```

De esta forma se delega a que la clase que haga uso del método, gestione las posibles excepciones que se puedan producir.

Generar una excepción throw

Nos permite generar una excepción. La palabra *throw* seguida de un objeto de la clase *Throwable* o de una subclase de ella, lanzará ese objeto como una nueva excepción.

```
throw new FileNotFoundException("El archivo no existe");
```

```
throw new MiException("El valor es incorrecto");
```


Bloque finally

Se pone a continuación de los bloques *try... catch* para insertar las instrucciones que se quieren ejecutar independientemente de que se produzcan o no excepciones e independientemente de que se traten.

Posibilidades de ejecución ante un bloque finally:

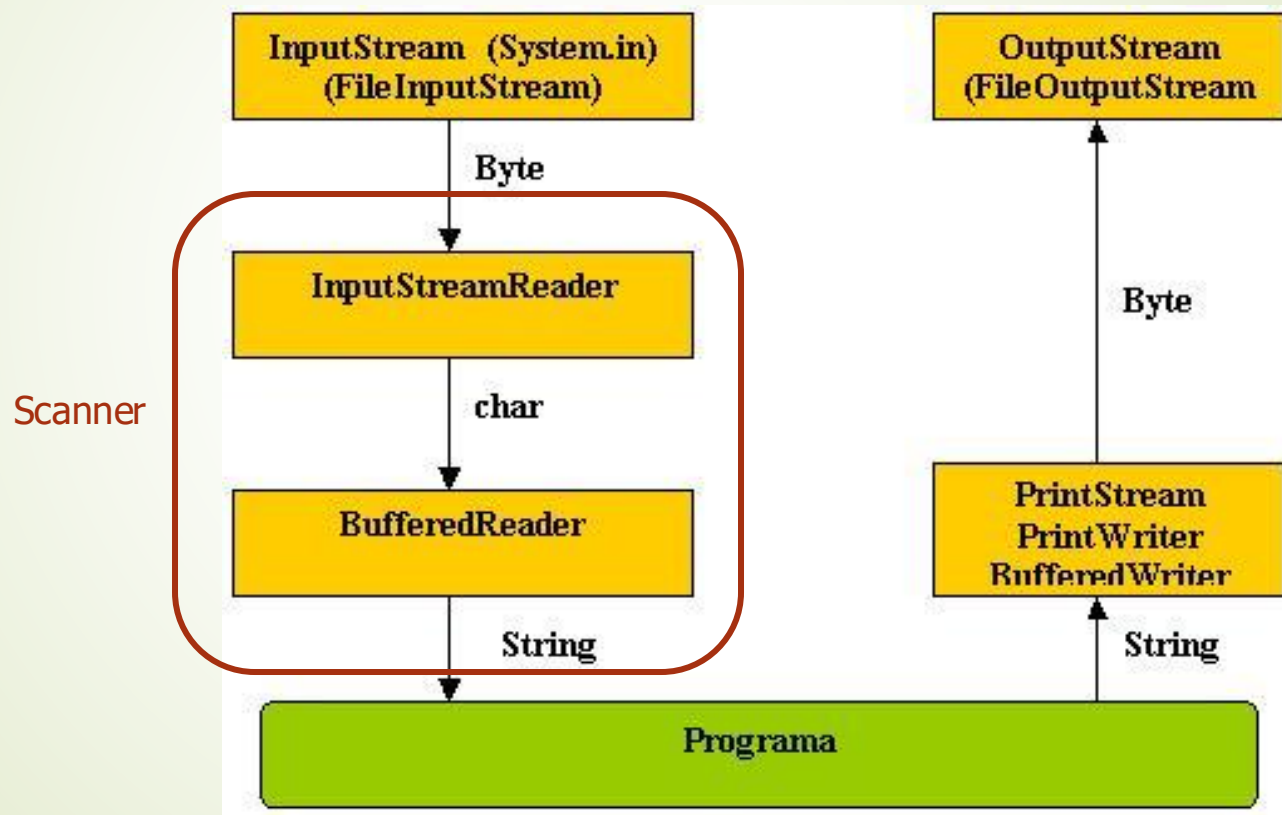
- 1.- Que no se produzca excepción
 - Se ejecuta el Finally
 - El programa continúa correctamente
- 2.- Que se produzca una excepción y se capture
 - Se ejecuta el catch
 - Se ejecuta el Finally
 - El programa continúa correctamente
- 3.- Que se produzca una excepción y no se capture
 - Se ejecuta el Finally

Conclusión: El finally siempre se ejecuta.

Flujos de Entrada - Salida

Java IO

El paquete java.io proporciona las clases necesarias para trabajar con flujos de entrada-salida de datos al programa, estos flujos serán comunes para trabajar con la pantalla, el teclado, archivos, comunicaciones y servidores web.



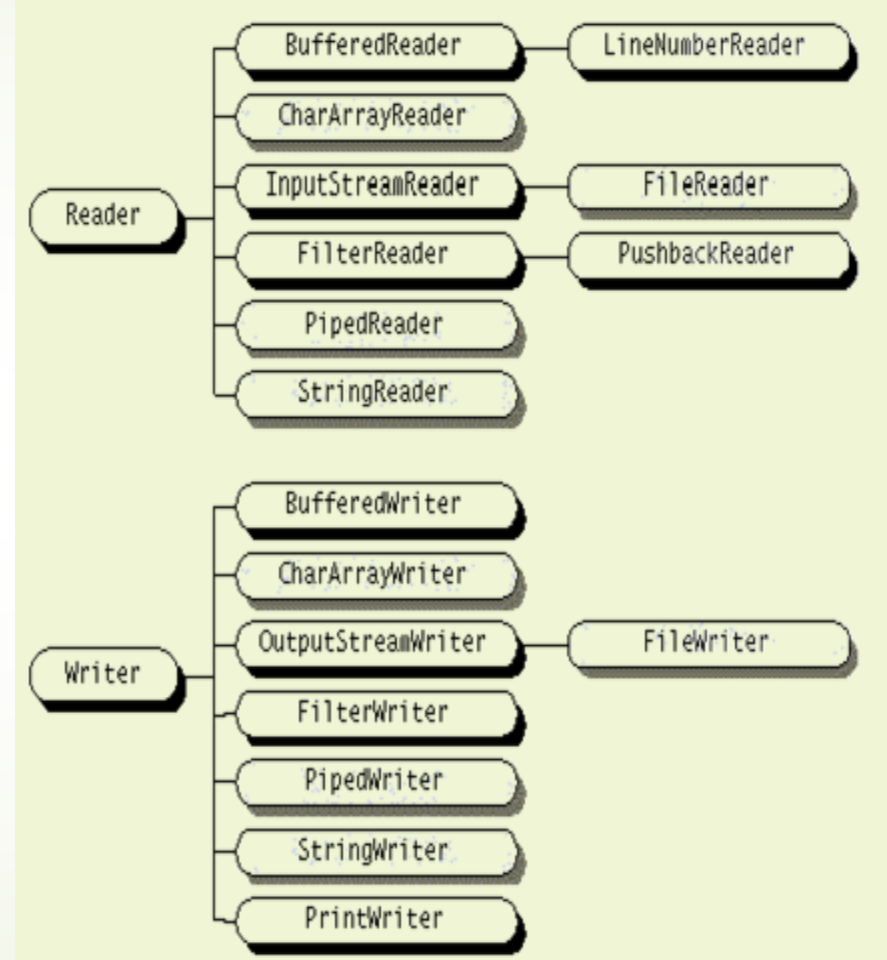
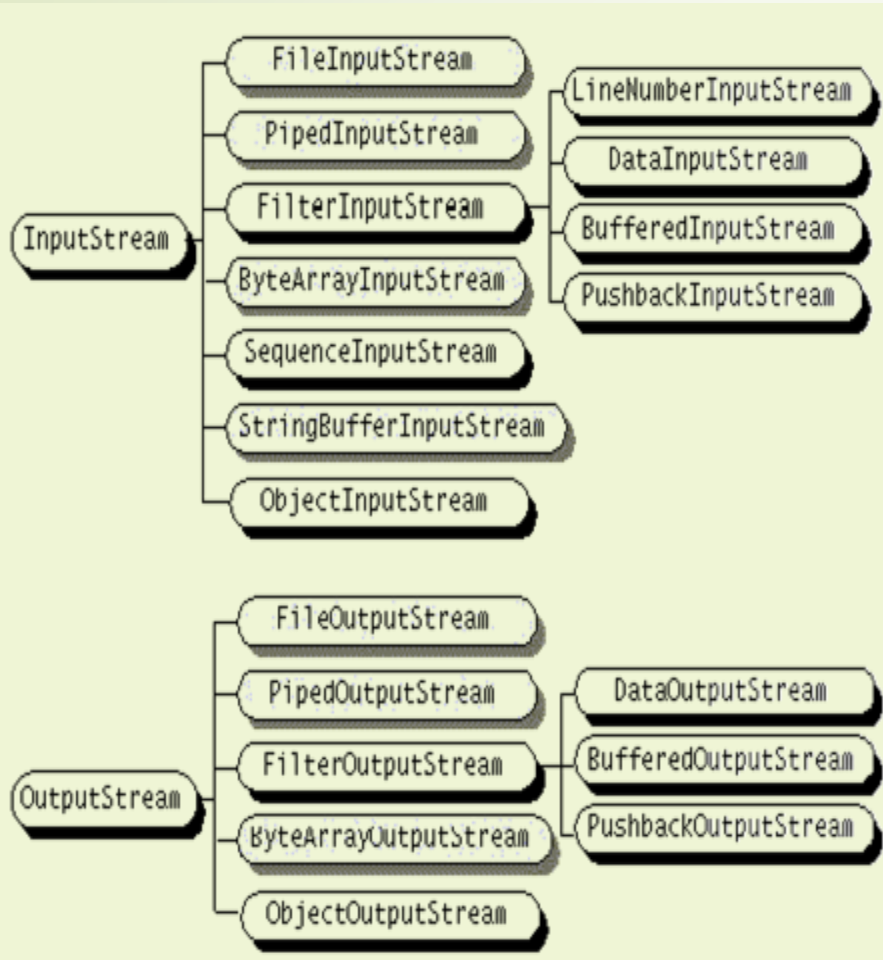
Las clases fundamentales

Los flujos de entrada y salida se representan al más bajo nivel con las clases **InputStream** y **OutputStream** respectivamente. El inconveniente de estas clases es que leen y envían la información sin formato alguno (bytes o secuencias de bytes).

Para poder leer la información en forma de caracteres, tenemos las clases **Reader** y **Writer**, que a partir de los flujos básicos (de bytes) ofrecen métodos para leer caracteres. Estas clases son **InputStreamReader** e **OutputStreamWriter**. El inconveniente de estas clases es que bloquean la ejecución hasta que hay un carácter disponible.

Para evitar el inconveniente anterior tenemos las clases **BufferedReader** y **PrintWriter** que nos permiten leer/escribir cadenas de caracteres completas.

Las clases del Sistema IO



Entrada y Salida de bytes

InputStream (System.in es un InputStream)(FileInputStream)	
int available ()throws IOException	Retorna el número de Bytes disponibles para ser leídos.
int read ()throws IOException	Lee un byte llegado por el flujo de entrada.
int read (byte[])throws IOException	Lee una matriz de bytes.
void close ()throws IOException	Cierra el flujo de entrada.
OutputStream (FileOutputStream)	
void write (int b) throws IOException	Escribe un byte en el flujo de salida
void flush ()throws IOException	Fuerza la salida por los flujos de los datos enviados a esta.
void close ()throws IOException	Cierra el flujo de salida.
void write (int[] b) throws IOException	Escribe una matriz de bytes en el flujo de salida

InputStreamReader

InputStreamReader	
InputStreamReader (InputStream in)	Constructor con el flujo al que se enlaza
int read ()throws IOException	Lee un carácter llegado por el flujo de entrada. Retorna -1 si se ha llegado al final.
boolean ready ()throws IOException	Retorna true si hay caracteres disponibles.
void close ()throws IOException	Cierra el flujo de entrada.

Reader's y Writer's

BufferedReader	
BufferedReader (Reader in)	Constructor, recibe un flujo de caracteres
String readLine () throws IOException	Lee una línea del flujo de entrada en forma de String.
void close ()throws IOException	Cierra el flujo de entrada.
PrintStream o PrintWriter (System.out es un PrintStream)	
PrintStream (OutputStream out)	Constructor, recibe un flujo de salida
void print (String s) void println (String s)	Escribe un String en el flujo de salida. println además termina la línea.
void close ()throws IOException	Cierra el flujo de salida.

Coverision String – char[]

byte[] -> String

Creamos un String pasándole en su constructor la matriz de bytes.

```
byte [] m = {'a','b','c','d','e'};  
String s = new String(m); // s="abcde";
```

String -> byte[]

Ejecutamos el método `getBytes()` de la clase `String` que retorna el String como matriz de bytes.

```
String s = "abcde";  
byte [] m = s.getBytes(); //byte [] m = {'a','b','c','d','e'};
```

Leer o grabar en ficheros

FileInputStream	
FileInputStream (File file) throws FileNotFoundException	Constructor, se suministra el archivo con el que se vincula mediante un File o mediante un String
int read ()throws IOException	Lee un byte llegado por el flujo de entrada.
int available ()throws IOException	Retorna el número de Bytes disponibles para ser leídos.
void close ()throws IOException	Cierra el flujo de entrada desde archivo.
FileOutputStream	
FileOutputStream(String name, boolean append) throws FileNotFoundException	Constructor, append permite indicar en el caso de que el archivo exista, si queremos añadir datos o sobrescribirlos.
write (int b); flush (); close ();	Igual que en la clase outputStream

Clase File

Permite manejar archivos y directorios. Crearlos, renombrarlos y borrarlos. También permite listar los archivos de un directorio. Así mismo permite conocer los atributos de los archivos.

<u>File(String pathname)</u>	<u>Constructor, recibe la ruta absoluta o relativa del archivo o directorio. El separador de directorios puede ser "/" o "\\"</u>
<u>exists(), isFile(), isDirectory(), canRead(), canWrite(), isHidden().</u>	<u>Retornan un booleano indicando si disponen de una determinada característica.</u>
<u>boolean createNewFile()</u>	<u>Crea el archivo, si el archivo ya existía retorna false y no se crea.</u>
<u>boolean mkdir();</u> <u>boolean mkdirs();</u>	<u>Crea el directorio (mkdir) y todos los directorios padre que no existieran (mkdirs).</u>
<u>boolean delete();</u>	<u>Borra el archivo o directorio (si está vacío)</u>
<u>File[] listFiles()</u>	<u>Retorna el contenido del directorio en forma de una matriz de objetos File.</u>
<u>boolean renameTo(File dest)</u>	<u>Renombra el archivo con el nombre indicado.</u>

Clase serializable

```
class Punto implements java.io.Serializable{

    // Las clases serializables solamente pueden tener
    // referencias a objetos de clases serializables, ya
    // que estos objetos también serán serializados.

    int x;
    int y;
    transient int z;//Transient indica que no se debe serializar

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

    public String toString(){
        return "[" + x + ", " + y + "]" (z=" + z + ");
    }
}
```

Interface java.io.Serializable

Java tiene una forma específica para convertir un objeto en binario. De esta forma se podrán enviar objetos por un flujo de entrada salida (archivos, red, etc). Para ello, las clases cuyos objetos vayan a ser enviados por un flujo, deben implementar la interface Serializable.

ObjectInputStream	
ObjectInputStream (InputStream)	Constructor en el que le asociamos el flujo de entrada del cual se leen los datos binarios.
int readInt() throws IOException	Lee un int llegado por el flujo de entrada. Hay métodos análogos para todos los tipos primitivos
Object readObject()	Lee un Objeto del flujo binario de entrada.
ObjectOutputStream	
ObjectOutputStream (OutputStream)	Constructor en el que le asociamos el flujo de salida en el cual se escriben los datos binarios.
writeInt (int b)	Escribe un entero en el flujo binario.
writeObject (Object obj)	Escribe un objeto en el flujo binario.

Properties

Clase Properties

Estructura de un archivo de propiedades(miaplicacion.properties):

```
#Esto es un comentario  
bbdd.user=alumno17  
bbdd.pass=java
```

Escribir propiedades:

```
Properties p= new Properties();  
p.setProperty("bbdd.user", "alumno17");  
p.setProperty("bbdd.pass", "java");  
p.store(new FileOutputStream("miaplicacion.properties"),"Esto es un comentario");
```

Leer Propiedades

```
Properties p= new Properties();  
p.load(new FileInputStream("miaplicacion.properties"));  
System.out.println("bbdd.user -> " + p.getProperty("bbdd.user"));  
System.out.println("bbdd.pass -> " + p.getProperty("bbdd.pass"));
```

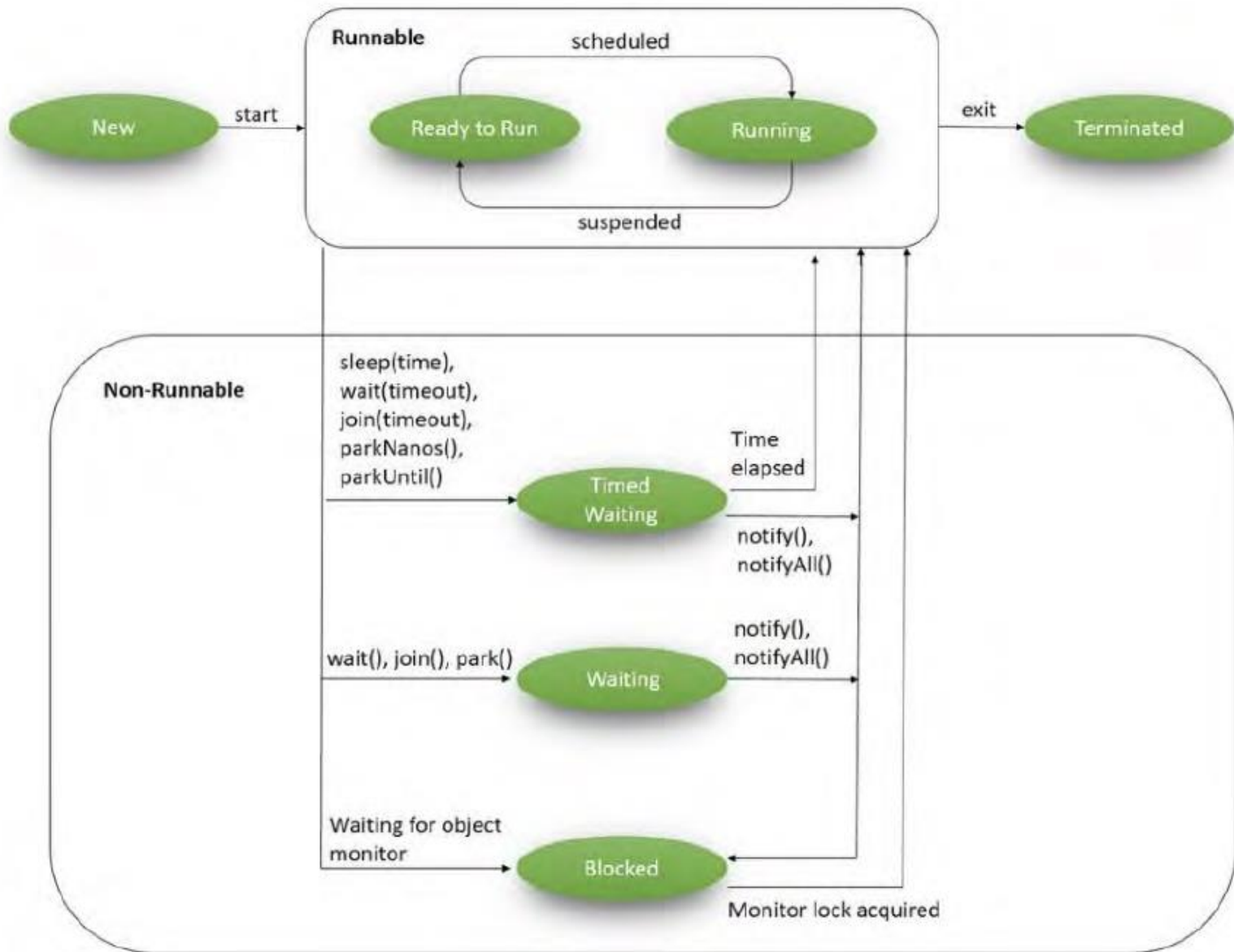
Threads

Procesamiento Paralelo

Threads

- En java se puede trabajar con varios hilos de ejecución en paralelo.
- Cuando se ejecuta el método main, se inicia el hilo principal. Pero, a partir de ahí, se pueden lanzar hilos que se ejecuten en paralelo.

Estados



Estados

NEW: un hilo recién creado que aún no ha comenzado la ejecución.

RUNNABLE: en ejecución o listo para ejecutarse, pero esperando la asignación de recursos.

BLOCKED: esperando adquirir un bloqueo de monitor para entrar o reingresar a un bloque/método sincronizado.

WAITING: esperando a que otro hilo realice una acción específica sin límite de tiempo.

TIMED_WAITING: esperando a que otro hilo realice una acción específica durante un período específico.

TERMINATED: ha completado su ejecución.

Clase Thread

Para crear un hilo de ejecución diferente al del hilo main, se debe crear un objeto Thread o heredero de Thread y posteriormente ejecutar su método *start()*.

Esto arranca el nuevo hilo y ejecuta las instrucciones que se hayan insertado en el método *run()* de Thread.

La clase Thread

void run()	Este método se debe reescribir para recoger las instrucciones que debe ejecutar el hilo.
void start()	Método que arranca el hilo y llama a run();
void yield()	Cede la ejecución del hilo actual al siguiente hilo.
final void wait()	Detiene la ejecución del hilo hasta que otro hilo le notifique (mediante en método <i>notify()</i>)que puede continuar.
final void notify()	Indica al hilo que ya puede continuar ejecutándose.
final void sleep(miliseg)	Detiene la ejecución de un hilo un número determinado de milisegundos.
void interrupt()	Aborta la ejecución de un hilo.(produce una InterruptedException)
void setPriority(prioridad)	Establece la prioridad de un hilo (1-> mínima prioridad; 10-> máxima prioridad;) Si no se indica se hereda la prioridad del hilo en el que ha sido creado el Thread. Main tiene prioridad 5.
void setName(String n)	Establece el nombre asignado a un hilo.

La Interface Runnable

```
class ClaseConHilos implements Runnable{
```

```
    ...
```

```
    public void run(){
```

```
        ...
```

```
    }
```

```
}
```

```
class Testr{
```

```
    public static void main(String [] args){
```

```
        Thread c1= new Thread(new ClaseConHilos());
```

```
        Thread c2 = new Thread(new ClaseConHilos());
```

```
        c1.start();
```

```
        c2.start();
```

```
        System.out.println("Fin de main");
```

```
    }
```

```
}
```

Invocar métodos de Thread trabajando con Runnable

Por ejemplo, para dormir el hilo desde una clase que implemente runnable, en lugar de ejecutar:

```
this.sleep(1000);
```

Ejecutaremos:

```
Thread.currentThread().sleep(1000);
```

o simplemente

```
Thread.sleep(1000);
```

Sincronización de Hilos

- Existen tareas que no deben ser ejecutadas simultáneamente por dos hilos
 - (p.e.: el acceso a determinados recursos, archivos ...). ACCESO CONCURRENTE.
- Para ello, utilizamos **bloqueos** o la palabra clave **synchronized**.

synchronized

- Una forma de sincronizar hilos es utilizando la palabra reservada `synchronized`
- En estos casos, es necesario un objeto que controle y cambie de estados los hilos (monitor). Cualquier objeto java puede hacerlo
- Se puede declarar un método como `synchronized` o bien un bloque de código
- Método `synchronized`
 - El objeto monitor es `this`
- Bloque `synchronized`
 - Le indicamos qué objeto será el monitor

Sincronización de Hilos

wait() – notify()

- Sólo se pueden utilizar dentro de un bloque/método synchronized
- Object.wait()
 - Fuerza a que el thread actual se detenga (pasa al estado **WAITING**) y libera el monitor
 - Queda a la espera a recibir aviso de otro thread con notify() o notifyAll() (para pasar al estado **RUNNABLE**)
- Object.notify()
 - Notifica a un hilo (al azar) del mismo monitor para que vuelva a estado **RUNNABLE**
- Object.notifyAll()
 - Notifica a todos los hilos del mismo monitor

Executor Service

- Implementa el patrón Thread-Pool (o worker)
- Gestiona un pool de threads
 - Llamadas a múltiples APIs
 - Procesar múltiples ficheros
 - Consultas a diferentes fuentes de datos o Bases de Datos diferentes
 - Armado previo de información para montar un informe
- `newFixedThreadPool(int cant)`
 - Si están ocupados los “cant” threads, las tareas se encolan
- `newCachedThreadPool()`
 - Crea nuevos hilos si los necesita

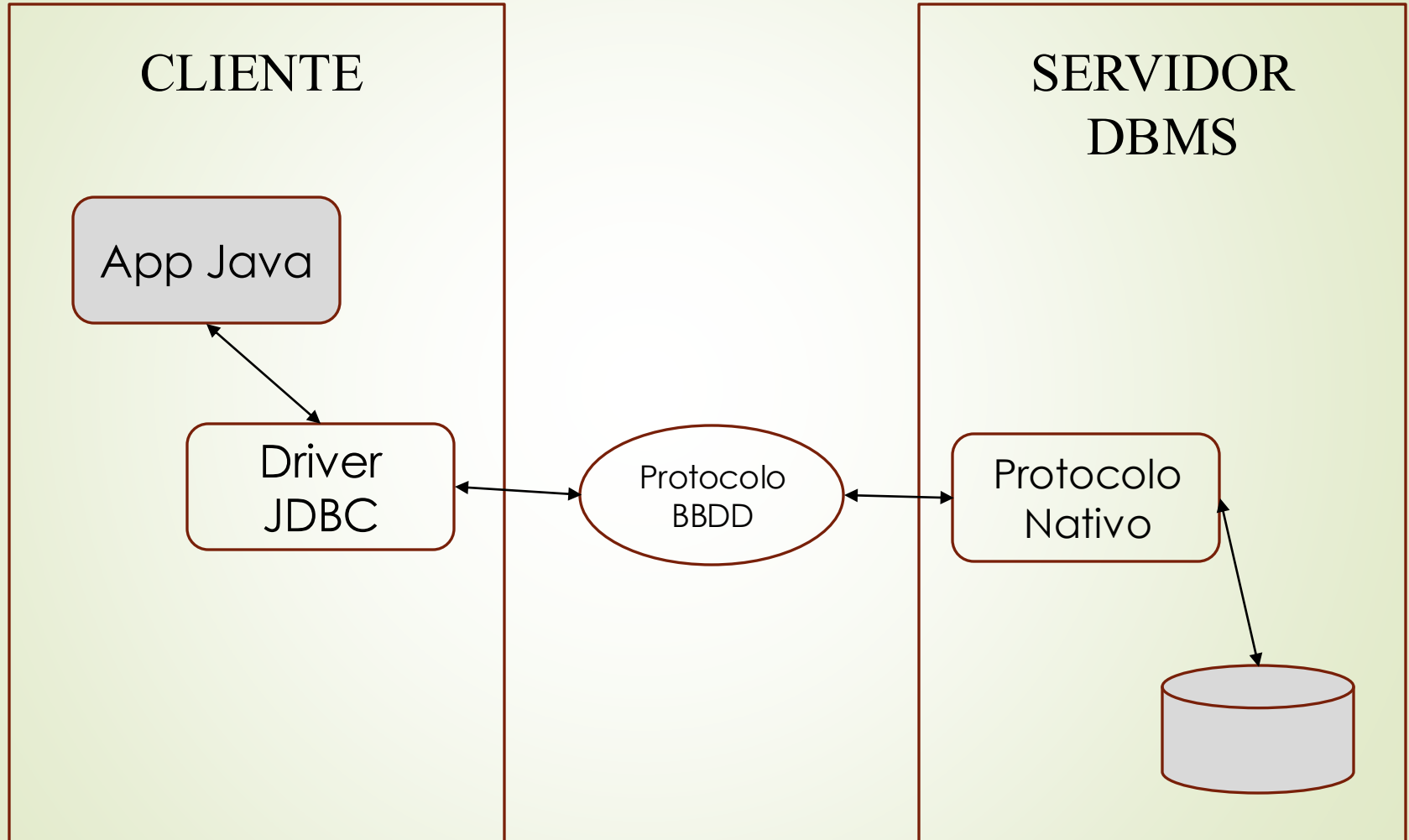
JDBC

Java DataBase Connectivity

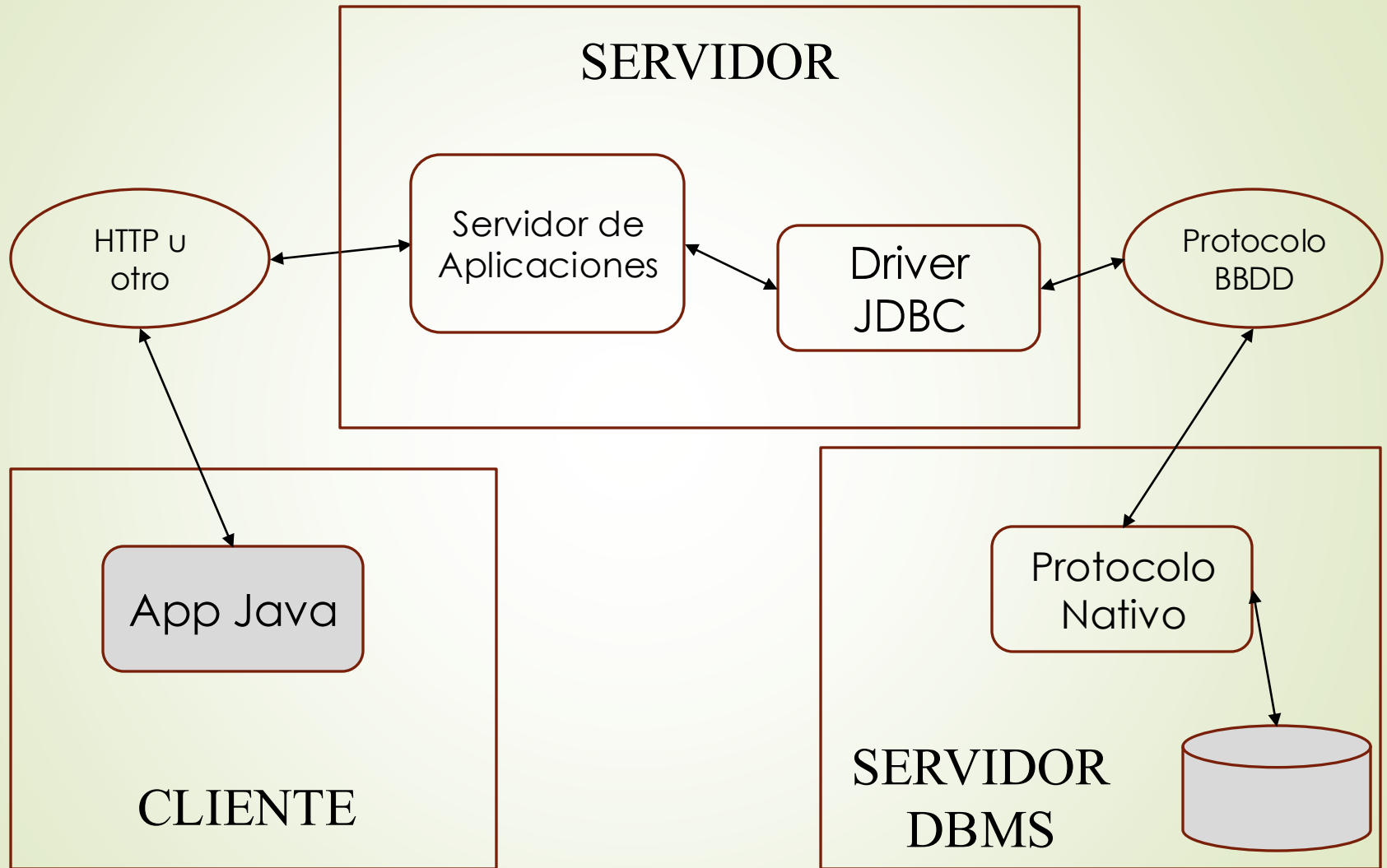
JDBC

- ESPECIFICACIÓN de clases y métodos que permiten acceder a Sistemas de BB.DD. en forma homogénea.
- Acceso a un driver JDBC apropiado.
- La conectividad se basa en sentencias SQL.
- Funcionalidad JDBC en el paquete `java.sql`

Modelo 2 capas



Modelo 3 capas



Acceder a la BBDD

1. CARGAR EL DRIVER /
OBTENER DATASOURCE
2. OBTENER UNA CONEXIÓN
3. CREAR UNA SENTENCIA DE EJECUCIÓN
4. EJECUTAR SQL

1-Cargar el Driver / Crear DataSource

Class.forName(String driver) throws ClassNotFoundException

```
...  
// MySQL  
Class.forName("com.mysql.jdbc.Driver");
```

Crear DataSource / Obtener DataSource de Servidor

javax.sql.DataSource

```
BasicDataSource bds = new BasicDataSource();  
bds.setDriverClassName(...);  
bds.setURL(...);  
bds.setUsername(...);  
bds.setPassword(...);  
DataSource ds = bds;
```

2-Obtener una conexión

**Connection DriverManager.getConnection(String URL, String us, String pwd)
(SQLException)**

URL:

jdbc:<subprotocolo>//<servidor>:<puerto>/baseDeDatos

```
... //MySQL
Class.forName("com.mysql.jdbc.Driver");
Connection con;
con= DriverManager.getConnection("jdbc:mysql://localhost/pruebas", "", "");
...
```

```
...
DataSource ds = ...;
Connection con;
con= ds.getConnection();
...
```

3-Sentencia de Ejecución

Statement Connection.createStatement() (SQLException)

```
... //MySQL
Class.forName("com.mysql.jdbc.Driver"); // v5...
Class.forName("com.mysql.cj.jdbc.Driver"); // v8...
Connection con;
con= DriverManager.getConnection("jdbc:mysql://localhost/pruebas", "", "");
Statement sql;
sql = con.createStatement();
...
```

```
...
DataSource ds = ...;
Connection con;
con= ds.getConnection();
Statement sql;
sql = con.createStatement();
...
```

4-Ejecutar SQL

Statement

Actualización
ddl, dml

{ int .executeUpdate(String cons)
boolean .execute(String cons)

Consultas
sql

{ **ResultSet** .executeQuery(String cons)

```
...  
DataSource ds = ...;  
Connection con;  
con = ds.getConnection();  
Statement sql;  
sql = con.createStatement();  
int filas = sql.executeUpdate("UPDATE Productos SET nombre='XX' WHERE codigo=1");  
...
```

ResultSet

`boolean .next()`

`XXX .getXXX(String columna)`

`XXX .getXXX(int columna)`

```
ResultSet rs = sql.executeQuery("SELECT ...");  
while (rs.next()){  
    //Obtener datos de cada fila  
}
```

Ejemplo Consulta

```
...  
List<String> nombreClientes = new ArrayList<>();  
  
DataSource ds = ...;  
Connection con;  
con = ds.getConnection();  
Statement sql;  
sql = con.createStatement();  
ResultSet rs = sql.executeQuery("SELECT * FROM Clientes");  
while (rs.next()) {  
    nombreClientes.add(rs.getString("nombre_cliente"));  
}  
...
```

PASO 1

PASO 2

PASO 3

PASO 4

Sentencias pre-compiladas

```
PreparedStatement Connection.prepareStatement(String sql)
```

```
void PreparedStatement.setXXX(int indice, XXX valor)
```

```
...  
String consulta;  
consulta = "INSERT INTO contactos VALUES (NULL,?,?,?,?,?,?,?,?,?)";  
PreparedStatement ps = con.prepareStatement(consulta);  
ps.setString(1, contacto.getNombre());  
ps.setString(2, contacto.getApellidos());  
...  
ps.setInt(6, contacto.getDom().getNumero());  
...  
ResultSet rs = ps.executeQuery();  
...
```

Transacciones

- Cuando se crea una conexión, está en modo auto-commit.
- Para ejecutar una transacción:
 1. Desabilitar el modo auto-commit.
 2. Lanzar las consultas.
 3. Finalizar con `commit()` o bien con `rollback()`.

Ejemplo transacción

```
int cuentaA, cuentaB, transferencia;
Connection con=null;
Statement sql=null;
...
con.setAutoCommit(false);
try {
    int resu1 = sql.executeUpdate("UPDATE Cuentas SET saldo=saldo-" + transferencia +
                                " WHERE codigoCuenta=" + cuentaA);
    int resu2 = sql.executeUpdate("UPDATE Cuentas SET saldo=saldo+" + transferencia +
                                " WHERE codigoCuenta=" + cuentaB);

    if (resu1==1 && resu2==1) {
        con.commit();
    }else{
        con.rollback();
    }
}catch (Exception e) {
    con.rollback();
    throw e;
}
con.setAutoCommit(true); ...
```

Tipos de ResultSet

Statement Connection.createStatement()

Statement Connection.createStatement(int tipoRS, int concurrencia)

Tipo de ResultSet	
TYPE_FORWARD_ONLY	default
TYPE_SCROLL_INSENSITIVE	acceso aleatorio
TYPE_SCROLL_SENSITIVE	acceso aleatorio

Tipo de concurrencia	
CONCUR_READ_ONLY	default
CONCUR_UPDATEABLE	actualizable

ResultSet

SCROLL_INSENSITIVE

- `boolean .first()`
- `void .beforeFirst()`
- `boolean .absolute(int fila)`
- `boolean .previous()`
- `int .getRow()`
- `boolean .isBeforeFirst ()`
- `boolean .isFirst()`
- `boolean .last()`
- `void .afterLast()`
- `boolean .relative(int fila)`
- `boolean .isAfterLast()`
- `boolean .isLast()`

Procedimientos almacenados

A partir de la conexión (conexion) a la base de datos, ejecutaremos:

```
CallableStatement sentenciaAlmacenada =  
    conexion.prepareCall("call NOMBRE_PROCEDIMIENTO");  
ResultSet rs= sentenciaAlmacenada.executeQuery();
```

Para poder ejecutar sentencias almacenadas en mysql, hay que dar permiso al usuario para leer la tabla donde están guardados los procedimientos (mysql.proc) y así leer los parametros de E/S.

```
GRANT SELECT ON mysql.proc TO user;
```

En Access en lugar de *call* es *exec*

Metadatos del ResultSet

```
ResultSetMetaData ResultSet.getMetaData()
```

ResultSetMetaData

```
int .getColumnCount()  
String .getColumnName(int columna)  
int .getColumnType(int columna)  
etc.
```

```
...  
if (rsmd.getColumnType(1) == Types.VARCHAR) {  
    ...  
}  
...
```

Metadatos de la Conexión

```
DatabaseMetaData Connection.getMetaData()
```

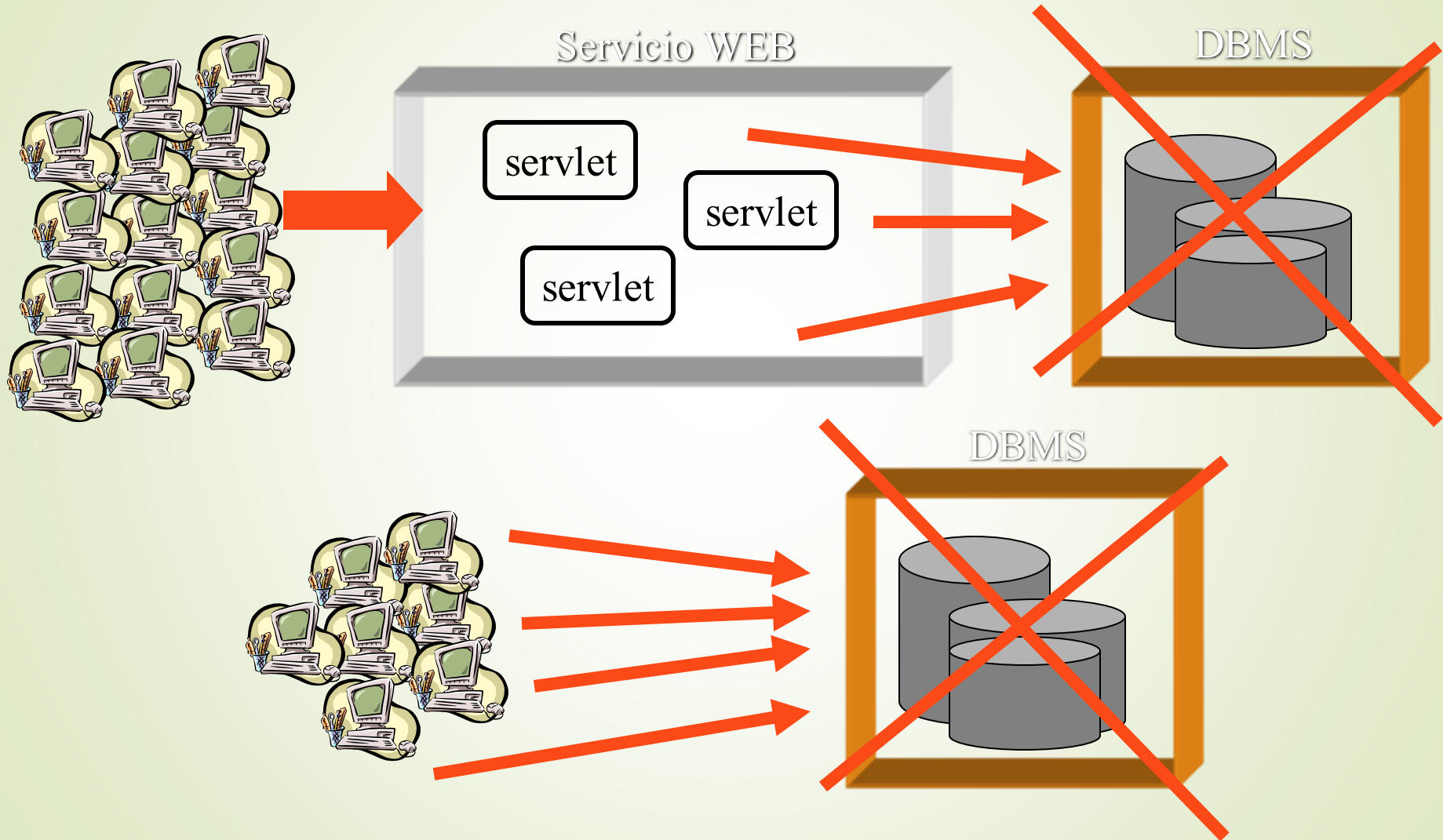
Capacidades límites del DBMS

Métodos de
DatabaseMetaData

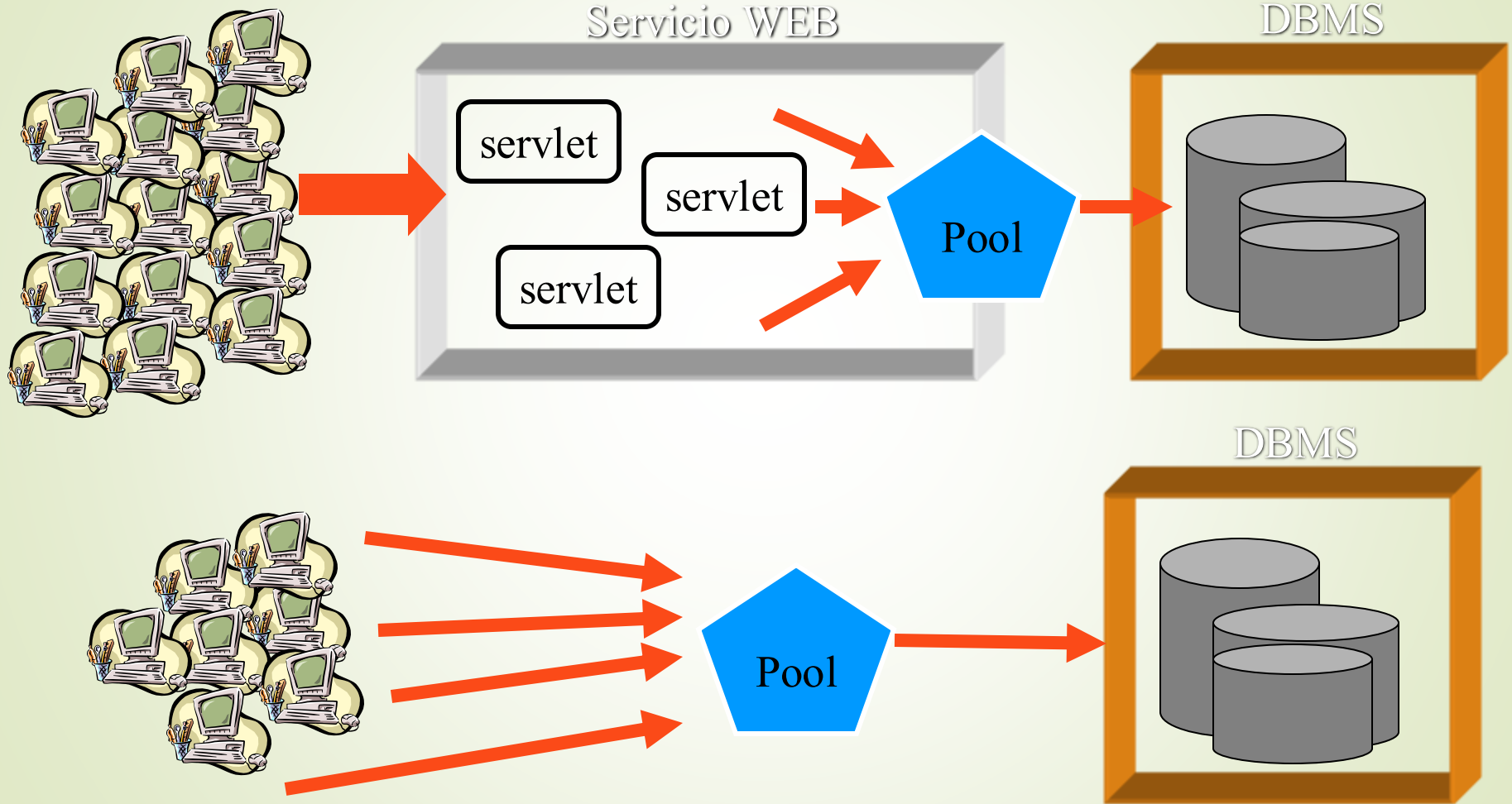
Funcionalidad del DBMS

MetaDatos {
 Inf. de Catálogos
 Inf. de Esquemas
 Inf. de Tablas

Accesos Simultáneos



Pool de Conexiones



Patrón Singleton

```
public class Singleton {  
    private static Singleton instancia;  
  
    private Singleton() {}  
  
    public static Singleton obtenerInstancia() {  
        if (instancia == null)  
            instancia = new Singleton();  
        return instancia;  
    }  
}
```