

PROGRAMACIÓN EN JAVA

(I)



Diciembre 2010

Mónica Trella López

Correo electrónico: trella@lcc.uma.es

Despacho: 3.2.34

David Bueno Vallejo

Correo electrónico: bueno@lcc.uma.es

Despacho: 3.2.27

PARTE 1.

1. INTRODUCCIÓN

2. ELEMENTOS DE PROGRAMACIÓN

3. INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

3.1 CLASES Y HERENCIA

3.2 DOCUMENTACIÓN DE CLASES: JAVADOC

3.3. CLASES GENÉRICAS

3.4 INTERFACES Y PAQUETES

4. OBJETOS BÁSICOS: NÚMEROS Y CADENAS

5. EXCEPCIONES

6. ENTRADA / SALIDA

6.1 PAQUETE JAVA.IO

6.2 ARCHIVOS

6.3 ACCESO A RECURSOS DEL SISTEMA: PROPERTIES

6.4 INTERNACIONALIZACIÓN

6.5 SERIALIZACIÓN

PARTE 2.

7. COMUNICACIONES

7.1.TCP/IP

7.2.UDP

8. COLECCIONES

8.1 CONJUNTOS

8.2 LISTAS

8.3 MAPAS

9. CONCURRENCIA

9.1 INTRODUCCIÓN

9.2 CREACIÓN DE HEBRAS

9.3 CICLO DE VIDA DE UNA HEBRA

9.4 MÉTODOS E INSTRUCCIONES SINCRONIZADAS

9.5 WAIT Y NOTIFY

9.6 VARIABLES CONDICIÓN Y LOCKS

10. ACCESO A BASES DE DATOS CON JDBC

10.1. INTRODUCCIÓN

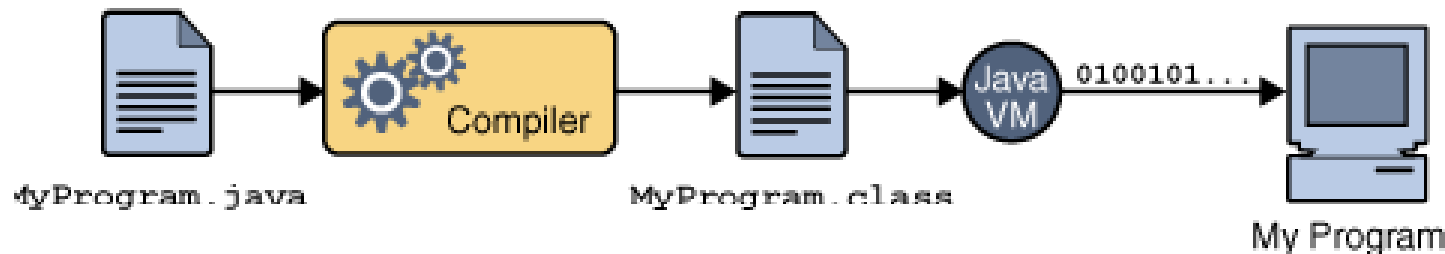
10.2. CONEXIÓN CON UNA BASE DE DATOS

10.3. CONSULTA A UNA BASE DE DATOS

10.4. SENTENCIAS PRECOMPILADAS Y LLAMADAS A PROCEDIMIENTOS

11. INTRODUCCIÓN AL DESARROLLO DE GUI CON NETBEANS

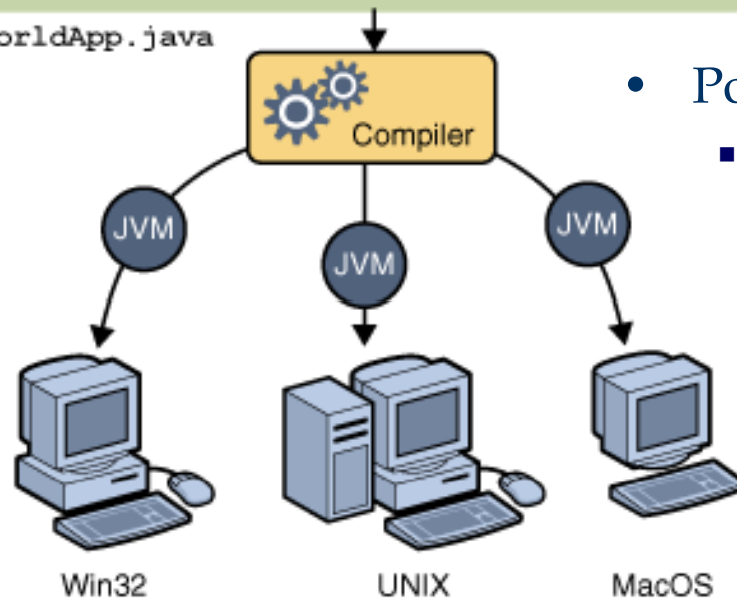
- Lenguaje de programación de alto nivel de sintaxis similar a C/C++ creado por SUN Microsystems en 1995.
- Entre otras destacamos las siguientes características del lenguaje:
 - Orientado a objetos
 - Compilado e interpretado
 - ✓ El compilador produce un código intermedio independiente del sistema denominado *bytecode*.
 - ✓ Este código es interpretado a código máquina por la *Java Virtual Machine* (JVM), intérprete dependiente de la plataforma.



Java Program

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

HelloWorldApp.java



- Portátil
 - Los programas Java se ejecutan en cualquier plataforma sin necesidad de volver a compilar.

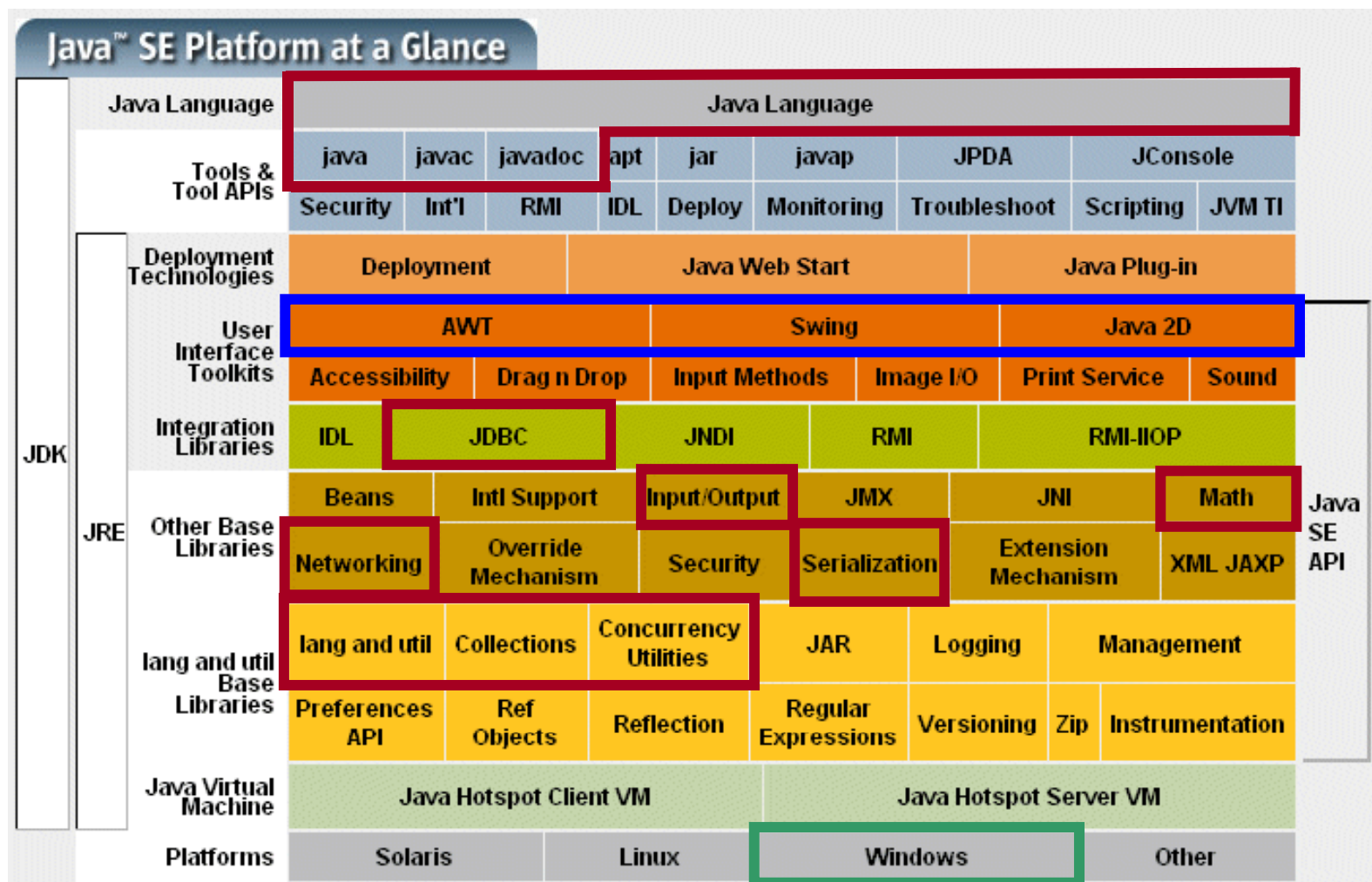
- Java se distribuye gratuitamente^(*) en tres plataformas
 - Java Platform, Standard Edition (Java SE)
 - Java Platform, Enterprise Edition (Java EE)
 - Java Platform, Micro Edition (Java ME)

(*) Descarga de JSE 6.0: <http://java.sun.com/javase/downloads/index.jsp>
Descarga de JEE 5: <http://java.sun.com/javaee/downloads/index.jsp>
Descarga de JME 3.0: <http://java.sun.com/javame/downloads/index.jsp>

- Todas las plataformas incluyen:
 - el **compilador** (*javac*),
 - la **máquina virtual** de Java (JVM, *Java Virtual Machine*)
 - ✓ un programa, para una plataforma hardware y software particular, que ejecuta aplicaciones Java.
 - la **documentación**,
 - el **código fuente**,
 - y una interfaz de programación de aplicaciones (**API**, *Java Application Programming Interface*):
 - ✓ El API es una colección de componentes software ya implementados que proporcionan numerosas funcionalidades y que se pueden usar para crear otros componentes y aplicaciones.
 - ✓ Se organiza en librerías de clases e interfaces relacionadas conocidas como **paquetes**.

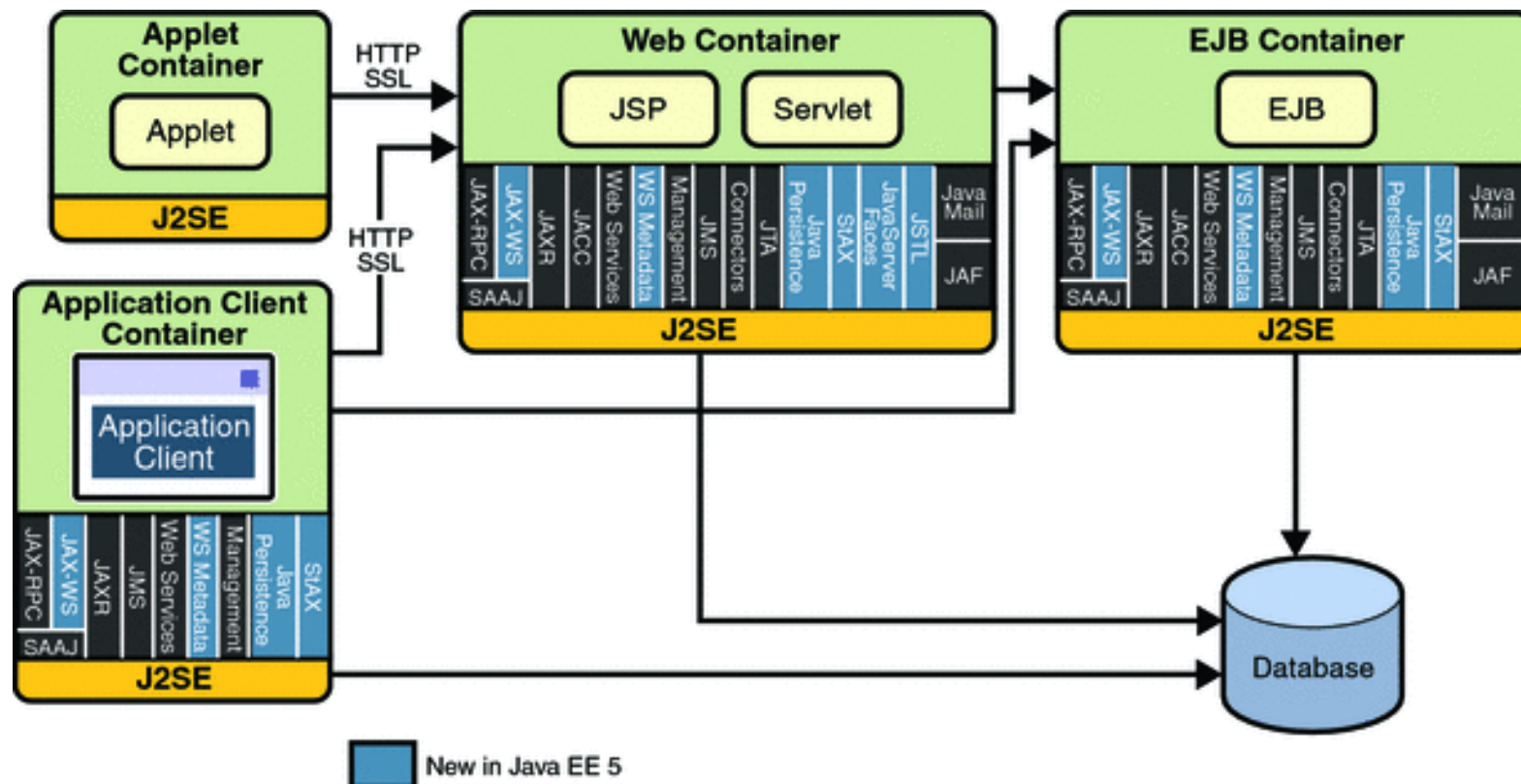
- **Java SE**

- Proporciona el núcleo de la funcionalidad del lenguaje Java
- Define desde los tipos básicos a clases de alto nivel usadas para comunicaciones en red, accesos a bases de datos, desarrollo de interfaces gráficas de usuario (GUI) y análisis de XML.



- **Java EE**

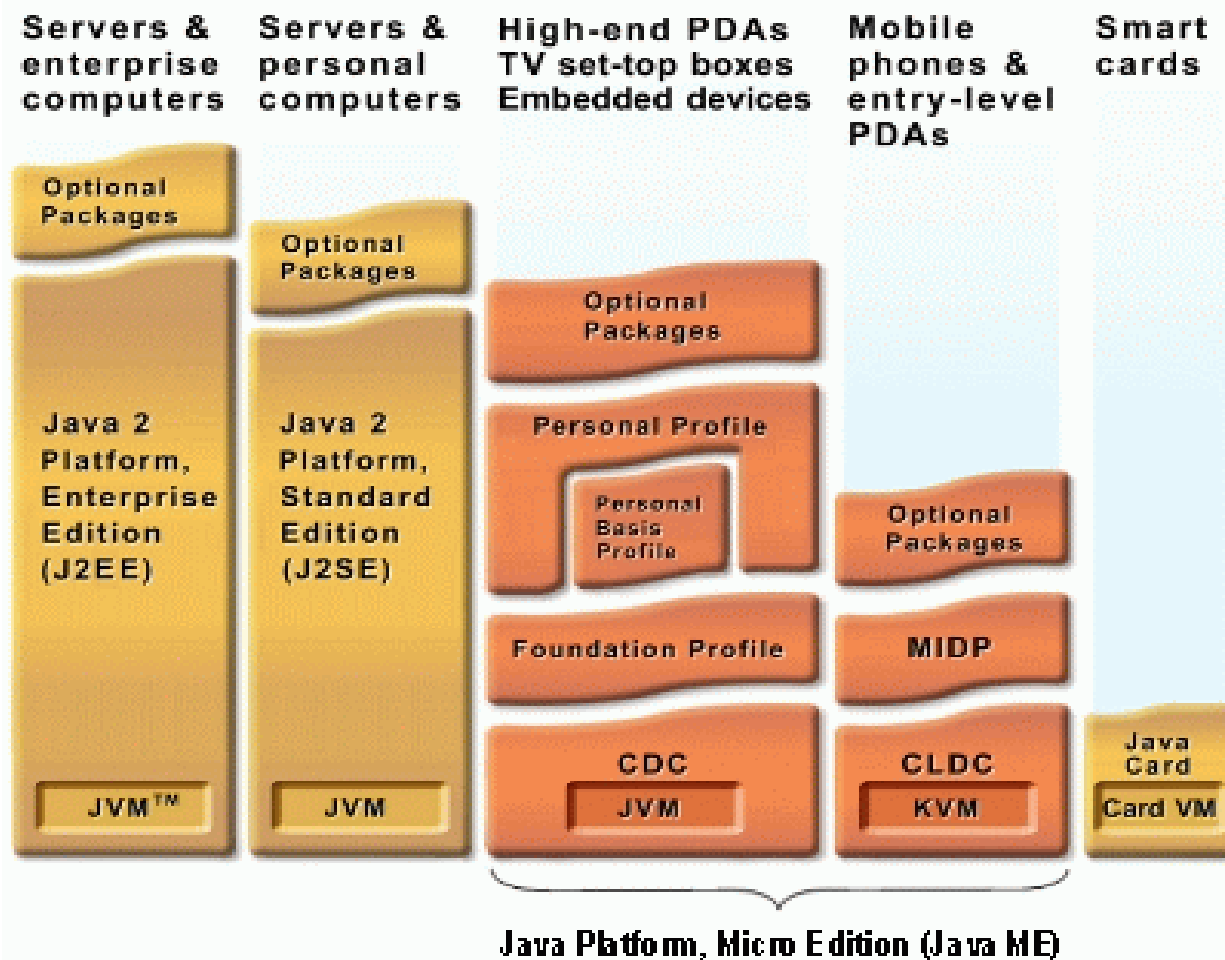
- Está construida sobre JSE.
- Proporciona un API y un entorno de ejecución para el desarrollo de aplicaciones en red a gran escala, multi-niveles, redimensionables (*scalable*) y seguras.



<http://java.sun.com/javaee/5/docs/tutorial/doc/bnaci.html>

• Java ME

- Es un subconjunto de JSE.
- Proporciona un API y una máquina virtual para la ejecución de aplicaciones en móviles y otros dispositivos.
- Las aplicaciones JME son a menudo clientes de aplicaciones JEE.



- Descargas imprescindibles:
 - <http://java.sun.com/javase/downloads/index.jsp>
 - J2SE 5.0. Entorno de desarrollo
 - J2SE 5.0 Documentación
- Instalarlo por ejemplo en c:\jdk1.6.0_17
 - y la documentación en c:\jdk1.6.0_17\docs
- Opcional entorno de desarrollo:
 - NetBeans 6.7 (gratuito de Sun disponible en la misma página)

- Para compilar en línea es necesario indicar el path de los ejecutables
- Están en la carpeta bin debajo de la instalación

`C:\jdk1.6.0_17\bin`

- En Windows XP se modifica en Mi PC → Propiedades → Opciones Avanzadas → Variables de Entorno → Variables del Sistema
- Hay que escribir el programa con extensión .java en un editor de texto

```
class HolaMundo {
    public static void main(String argv[]) {
        System.out.println("Hola mundo");
    }
}
```

Ficheros de demo
incluidos con las
trasparencias

HolaMundo.java

- Desde la línea de comandos llamar al compilador:

`javac HolaMundo.java`

- Eso genera un fichero de código intermedio HolaMundo.class que puede ejecutarse con el interprete java (no se pone el .class):

`java HolaMundo`

- Se pueden utilizar entornos de desarrollo como NetBeans/JBuilder que al principio complican al usuario pero que para programas grandes ayudan con:
 - Creación de proyectos que agrupan varios ficheros
 - Facilitan la organización en paquetes
 - Permiten depurar los programas
 - Permiten crear interfaces de usuario de una forma rápida
- El programa HolaMundo paso a paso en Netbeans está disponible en:

<http://java.sun.com/docs/books/tutorial/getStarted/cupojava/netbeans.html>

- Todo buen programador debe aprender un lenguaje con un libro/curso/tutorial al inicio
- Durante el resto del tiempo su mayor aliado será un manual de referencia
- Para cualquier lenguaje con el que trabajemos necesitamos un manual de referencia con los detalles de cada método, las librerías y las clases que se pueden utilizar
- En Java este manual de referencia es la API de su documentación. Que si se instaló como se indicó al principio estará en:

`C:\jdk1.6.0_17\docs\api\index.html`

- También es muy importante documentar nuestros programa utilizando la herramienta javadoc que se verá más adelante

Conceptos básicos del lenguaje

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

* No utilizada

** Añadida en la versión 1.2

*** Añadida en la versión 1.4

**** Añadida en la versión 5.0

- Un identificador es el nombre de un ‘item’ (Nombres de las clases, los métodos, las variables, etc...) en un programa escrito en Java
- Empiezan por una letra seguida de uno o varios caracteres alfanuméricos (letras o dígitos).
- Se consideran letras todos los caracteres de la A a la Z, (mayúsculas y minúsculas), el carácter _ y el carácter \$ (estos últimos no deben utilizarse en los identificadores)
- Las letras mayúsculas y minúsculas se consideran diferentes
- También se consideran letras las “letras especiales” de los lenguajes, tales como la letra Ñ, o las letras acentuadas (no se recomienda su uso para identificadores)

- En Java sólo existen dos tipos de datos: Primitivos y referencias
- Esta tabla contiene todos los tipos de datos primitivos de Java 5

	Descripción	Tamaño/Formato
Enteros		
byte	Byte-length integer	8-bit en Complemento a 2
short	Short integer	16-bit en Complemento a 2
int	Integer	32-bit en Complemento a 2
long	Long integer	64-bit en Complemento a 2
Reales		
float	Real de precisión sencilla	32-bit IEEE 754
double	Real de precisión doble	64-bit IEEE 754
Otros		
char	Un caracter	16-bit caracter Unicode
boolean	Un valor booleano (true or false)	true o false

- Cualquier clase, array o interfaz son referencias. Una referencia es en otros lenguajes conocido como un puntero/una dirección de memoria

- Ejemplos de literales en Java

Valor literal	Tipo de dato
178	int
8864L	long
37.266	double
37.266D	double
87.363F	float
26.77e3	double
'c'	char
true	boolean
false	boolean

- Una variable almacena el estado de un objeto
- Por convención suelen escribirse en minúsculas y si es una palabra compuesta cada inicial de palabra en mayúscula.
Ejemplos:

```
contador, miJuego, miVariableJava
```

- Se inicializan mediante el operador de asignación que es '='

```
int contador=65;  
char car='C';
```

- Si una variable se declara como final, equivale a una constante de otros lenguajes

```
static final int MAXNOTA=10;
```

- Las variables primitivas tienen un valor
- Las variables referencias tienen una referencia a un valor

Aritméticos

Operador	Uso
+	op1 + op2
-	op1 - op2
*	op1 * op2
/	op1 / op2
%	op1 % op2

ArithmeticDemo.java

Relacionales

Operador	Uso
>	op1 > op2
>=	op1 >= op2
<	op1 < op2
<=	op1 <= op2
==	op1 == op2
!=	op1 != op2

RelationalDemo.java

Unarios

Operador	Uso
+	+op
-	-op

Aritméticos breves

Operador	Uso
++	op++
++	++op
--	op--
--	--op

Desplazamiento nivel bits

Operador	Uso
<<	op1 << op2
>>	op1 >> op2
>>>	op1 >>> op2

Lógicos (bits)

BitwiseDemo.java

Operador	Uso
&	op1 & op2
	op1 op2
^	op1 ^ op2
~	~op

Asignación

Operador	Uso
+=	op1 += op2
-=	op1 -= op2
*=	op1 *= op2
/=	op1 /= op2
%=	op1 %= op2
&=	op1 &= op2
=	op1 = op2
^=	op1 ^= op2
<<=	op1 <<= op2
>>=	op1 >>= op2
>>>=	op1 >>>= op2

- Java ofrece diversas formas de alterar el flujo del programa y se resumen en la siguiente tabla.

Tipo de sentencia	Nombre
Bucles	while, do-while, for
Selecciones	if-else, switch-case
Manejo de excepciones	try-catch-finally, throw
Saltos	break, continue, label:, return

```
while (expresión) {  
    sentencias  
}
```

```
do {  
    sentencias  
} while (expresión);
```

```
int i=0;  
while(i<10) {  
    System.out.println("i="+i);  
    i++;  
}  
  
do {  
    System.out.println("i="+i);  
    i--;  
}  
while (i>0);
```

DBVWhile.java

- Clásico bucle para de java

```
for (inicialización; terminación; incremento) {
    sentencias
}
```

```
for (i=0;i<10;i++) {
    System.out.println("i="+i);
}
```

- A partir de la versión 5.0 se puede recorrer estructuras de forma equivalente al foreach de otros lenguajes (tcl o php): Para cada 'valor' de la estructura realiza las sentencias

```
for (tipo nombre: variableestructura) {
    sentencias
}
```

// Metodo tradicional

```
int[] diasMeses= {31,28,31,30,31,30,31,31,30,31,30,31};

for(i=0;i<diasMeses.length;i++) {
    System.out.println(" Valor2:"+diasMeses[i]);
}
```

DBVForEach.java

// Método foreach

```
int[] diasMeses= {31,28,31,30,31,30,31,31,30,31,30,31};
for (int valor : diasMeses) {
    System.out.println(" Valor:"+valor);
}
```

- Sentencia 'si' en sus tres formas

```
if (expresión) {  
    sentencias  
}
```

```
if (expresión) {  
    sentencias  
} else {  
    sentencias  
}
```

```
if (expresión) {  
    sentencias;  
} else if (expresión) {  
    sentencias;  
} else if (expresión) {  
    ...  
} else {  
    sentencias;  
}
```

ifElseDemo.java

- La condición del switch en java permite expresiones enteras o tipos enumerados (desde la version 5)

// Switch tradicional

```
int edad=1;
switch(edad) {
    case 1;;
    case 2;;
    case 5: System.out.print("Eres pequeñin");break;
    case 6: System.out.print("Eres Mayor");break;
    default: System.out.print("Todavía juegas a esto?");break;
}
```

// Switch con enumerados

```
// en la definición de la clase
public enum Colores {Rojo, Verde, Azul, Amarillo};
...
// en un subprograma
Colores color=Colores.Rojo;
switch (color) {
    case Rojo: System.out.print("Color Rojo");break;
    case Verde: System.out.print("Color Verde");break;
}
```

- Declaración

- Se pueden hacer arrays de tipos predefinidos o de objetos

```
int[] miArrayNum;  
StringBuffer[] miArrayCad;
```

- Creación

- Se reserva memoria para el array con el operador new

```
miArrayNum=new int[5];  
miArrayCad=new StringBuffer[5];
```

- Creación e Inicialización

- Se pueden realizar ambas operaciones a la vez asignando los valores iniciales

```
String[] miArrayCad2 ={"Cadena1","Cadena2"};
```

- Tamaño

- El tamaño es un atributo y no un método: length

```
miArrayCad.length;
```

- Ejemplo Resumen

```
public static void main(String[] args) {
    int i,j;
    int[] miArrayNum;
    int[][] miArray2D={{2,3},{5,4}};
    StringBuffer[] miArrayCad;
    String[] miArrayCad2 ={"Cadena1","Cadena2"};
    System.out.println(miArrayCad2[0]);
    miArrayNum=new int[5];
    // Array de tipos primitivos
    for(i=0;i<5;i++){miArrayNum[i]=i;}
    // mostrar con foreach
    for(int valor:miArrayNum) { System.out.print(valor+" "); }
    System.out.println();
    // Arrays multidimensionales
    for(i=0;i<2;i++) {
        for(j=0;j<2;j++){
            System.out.print(miArray2D[i][j]+" ");
        }
        System.out.println();
    } // Inicializacion correcta para un array de objetos
    miArrayCad=new StringBuffer[5]; // Crea el array
    for(i=0;i<5;i++){
        miArrayCad[i]=new StringBuffer(); // Crea cada objeto
    }
    System.out.println("Longitud="+miArrayCad.length);
}
```

Resultado:

Cadena1

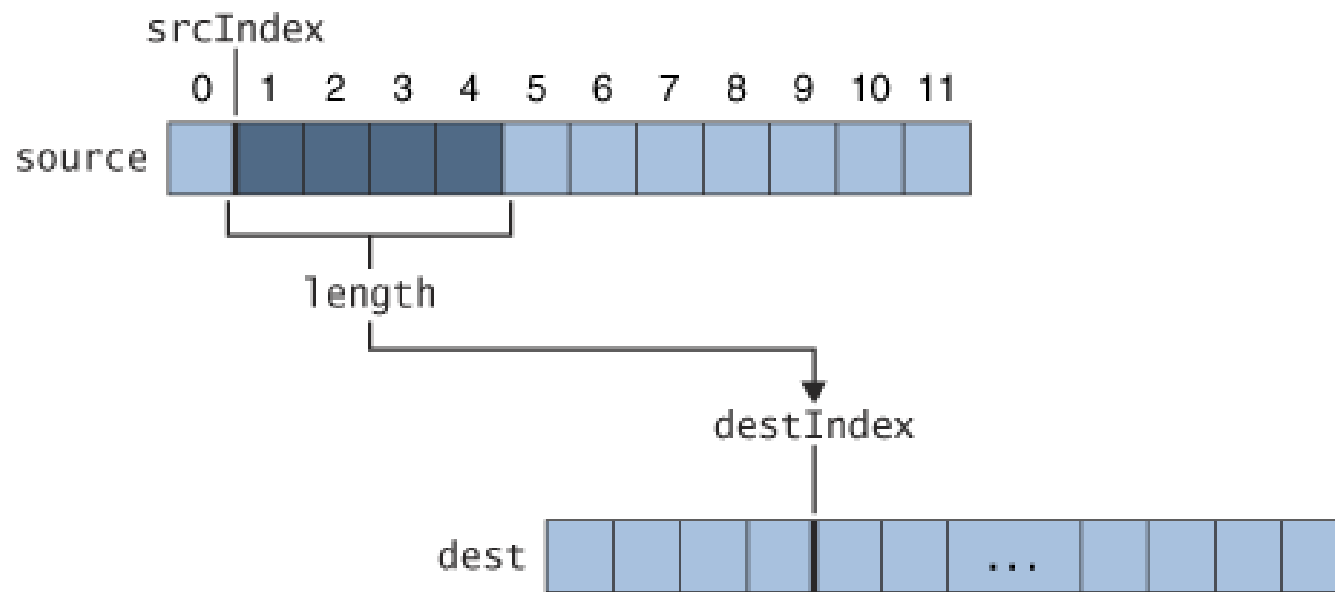
0 1 2 3 4

2 3
5 4

Longitud=5

DBVArrays

- `System.arraycopy(source,srcIndex,dest,destIndex,length)` permite copiar arrays o porciones de forma muy eficiente



Introducción a Clases y Objetos en Java

- Se van a explicar la declaración de clases, objetos mediante un ejemplo
- Este ejemplo se utilizará también para explicar cómo debe documentarse un programa para generar una ayuda completa mediante javadoc
- Se va a crear una clase **Alumno** que tendrá como atributos dos cadenas: el **nombre** y el **apellido**
- El nombre se declarará como **publico** con lo que desde fuera de la clase se podrá consultar su valor directamente
- El apellido se declarará como **privado** y sólo los objetos de la clase Alumno podrán acceder a él
- Se recomienda que todos los atributos sean privados y se acceda a ellos mediante métodos **get** (para darle un valor al atributo) y **set** (para consultar un valor del atributo)
- Se declara los métodos getter y setter para consultar y modificar los dos atributos
- Se crean dos constructores: Uno sin argumentos y otro que recibe los valores iniciales de nombre y apellido

```

/*
 * Alumno.java
 */
package introclases; // permite organizar las clases relacionadas guardandolas en la misma carpeta
/**
 * Esta es la clase Alumno que se utilizara para explicar clases, objetos
 * y como usar la documentacion
 * @author David Bueno Vallejo
 * @version 1.0
 */
public class Alumno {
    /** nombre del alumno */
    public String nombre;
    /** Apellido del alumno */
    private String apellido;
    /** Crea una nueva instancia de Alumno */
    public Alumno() {
        nombre=""; apellido="";
    }
    /**
     * Crea una nueva instancia recibiendo nombre y apellidos
     * @param nombre valor que se guardara en el atributo nombre
     * @param apellido valor que se guardara en el atributo apellido
     */
    public Alumno(String nombre,String apellido) {
        this.nombre=nombre;this.apellido=apellido;
    }
}

```

```
/**
 * Devuelve el apellido
 * @return devuelve una cadena con el apellido del alumno
 */
public String getApellido() {
    return apellido;
}
/**
 * Devuelve el nombre
 * @return devuelve una cadena con el nombre del alumno
 */
public String getNombre() { return nombre; }
/**
 * Modifica el apellido
 * @param apellido Contiene una cadena con el apellido a cambiar
 */
public void setApellido(String apellido) {
    this.apellido = apellido;
}
/**
 * Modifica el nombre
 * @param nombre Contiene una cadena con el nombre a cambiar
 */
public void setNombre(String nombre) {
    this.nombre = nombre;
}
}
```

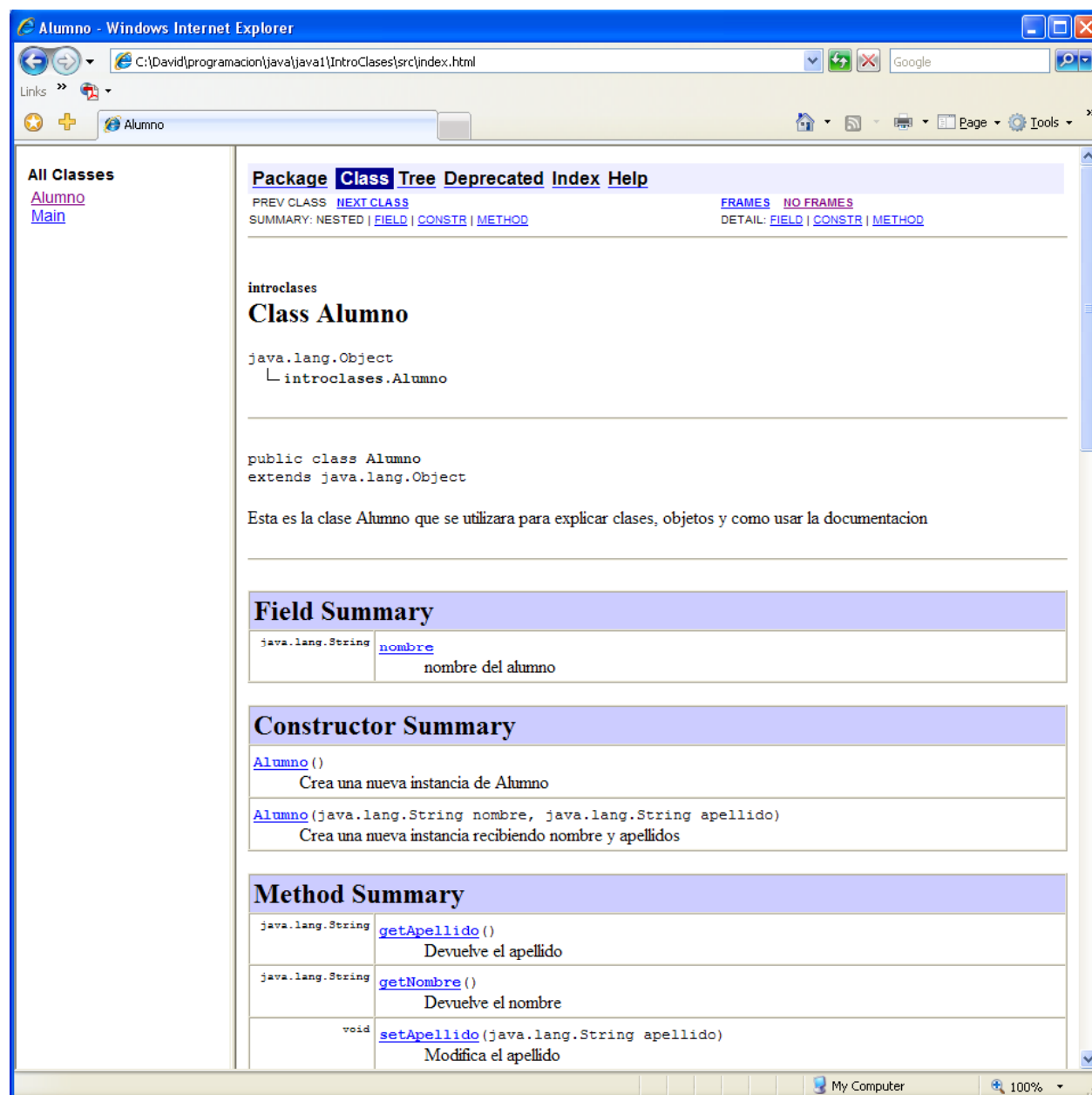
- Los comentarios que se incluyen en la documentación empiezan por `/**`
- Se deben incluir comentarios al principio de cada clase, método, atributo...

```
/**
 * Esta es la clase Alumno que se utilizara para explicar clases, objetos
 * y como usar la documentacion
 * @author David Bueno Vallejo
 * @version 1.0
 */
```

- Hay varios parámetros con información especial que empiezan con la `@` como:
 - `@author nombre.-` Nombre del autor de la página
 - `@version 1.0.-` Número de versión de la clase
 - `@param nombre descripción.-` Se utiliza para describir los parámetros de entrada de los métodos
 - `@return descripcion.-` Explica el formato que devuelve un método función
- Para generar la documentación se utiliza la herramienta javadoc incluida en la carpeta bin de instalación `C:\jdk1.6.0_17\bin\javadoc.exe` seguido de los ficheros a indexar o del paquete

javadoc introclases

- Resultado



The screenshot shows a web browser window titled "Alumno - Windows Internet Explorer". The address bar displays the file path: `C:\David\programacion\java\java1\IntroClases\src\index.html`. The browser shows a Javadoc-generated page for the `Class Alumno`.

Navigation Links: Package, **Class**, Tree, Deprecated, Index, Help.
 Summary: NESTED | FIELD | CONSTR | METHOD.
 Detail: FIELD | CONSTR | METHOD.

Class Hierarchy: `java.lang.Object`
 ↳ `introclases.Alumno`

Class Definition:

```
public class Alumno
extends java.lang.Object
```

Esta es la clase Alumno que se utilizara para explicar clases, objetos y como usar la documentacion

Field Summary

<code>java.lang.String</code>	nombre	nombre del alumno
-------------------------------	------------------------	-------------------

Constructor Summary

Alumno()	Crea una nueva instancia de Alumno
Alumno(java.lang.String nombre, java.lang.String apellido)	Crea una nueva instancia recibiendo nombre y apellidos

Method Summary

<code>java.lang.String</code>	getApellido()	Devuelve el apellido
<code>java.lang.String</code>	getNombre()	Devuelve el nombre
<code>void</code>	setApellido(java.lang.String apellido)	Modifica el apellido

Alumno - Windows Internet Explorer

C:\David\programacion\java\java1\IntroClases\src\index.html

Links

Alumno

All Classes

[Alumno](#)

[Main](#)

Field Detail

nombre

```
public java.lang.String nombre
```

nombre del alumno

Constructor Detail

Alumno

```
public Alumno()
```

Crea una nueva instancia de Alumno

Alumno

```
public Alumno(java.lang.String nombre,  
              java.lang.String apellido)
```

Crea una nueva instancia recibiendo nombre y apellidos

Parameters:

nombre - valor que se guardara en el atributo nombre

apellido - valor que se guardara en el atributo apellido

Method Detail

getApellido

```
public java.lang.String getApellido()
```

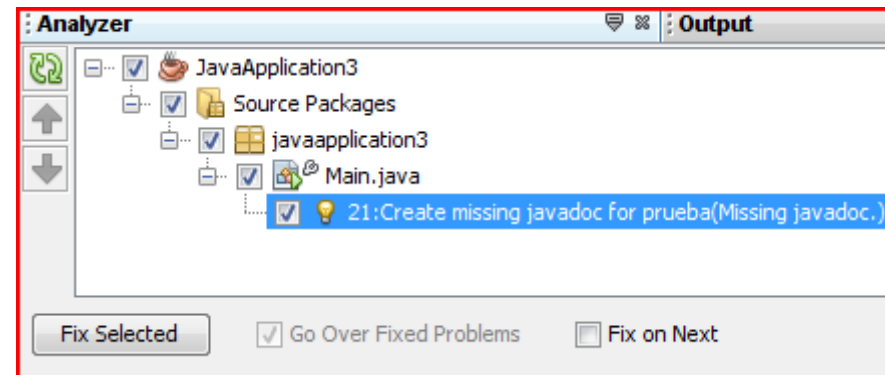
Devuelve el apellido

Returns:

devuelve una cadena con el apellido del alumno

My Computer 100%

- Netbeans simplifica la tarea de documentación de dos formas
 - Permite verificar si la clase está correctamente documentada y si no es así indica que campos hay que poner para corregirla → no tener que aprender los diferentes parámetros de javadoc
 - Esto se hace con el botón derecho sobre la clase (en la pestaña proyectos de la izquierda) → tools → Analyze Javadoc...
 - Aparecen los errores y con el botón fix selected se añaden los comentarios que faltan.



- En segundo lugar permite generar la documentación directamente mediante Run → Generate Javadoc

- Se ha visto como declarar una clase, sus atributos, sus métodos y cómo comentarlos correctamente
- El siguiente paso es instanciar objetos de esa clase. Eso se realiza en dos pasos: Declaración y Creación

```
Alumno al,al2;      // Declaración de dos objetos de la clase Alumno  
al=new Alumno(); // Creación del objeto al
```

- Posteriormente se podrá llamar a sus métodos o atributos públicos por ejemplo:

```
al.setNombre("David");  
al.setApellido("Bueno");  
al2=new Alumno("Monica","Trella");  
System.out.print("Nombre del alumno 1" +al.getNombre());  
System.out.print("Nombre del alumno 2" +al.getNombre());  
al2.nombre="Moni";  
/* al2.apellido="Trell"; Da error por que apellido es privado*/
```

- Ejemplo de instanciación de objetos de la clase alumno

```
package introclases;

/**
 * Clase principal de la aplicacion
 * @author David Bueno Vallejo
 */
public class Main {
    /**
     * Metodo en el que comienza la ejecucion
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Alumno al,al2;
        al=new Alumno();
        al.setNombre("David");
        al.setApellido("Bueno");
        al2=new Alumno("Monica","Trella");
        System.out.print("Nombre del alumno 1" +al.getNombre());
        System.out.print("Nombre del alumno 2" +al.getNombre());
        al2.nombre="Moni";
        /* al2.apellido="Trell"; Da error por que apellido es privado*/

    }
}
```

Proyecto Netbeans

IntroClases

- En java no hay que preocuparse por liberar la memoria reservada
- Cada cierto tiempo un proceso Garbage Collector (recolector de basura) recorre el programa buscando objetos que no estén referenciados y los libera
- Si un objeto se pone a null, será candidato para liberar su memoria si ningún otro objeto hacía referencia al mismo contenido

```
Alumno al1,al2  
al1=new Alumno();  
al2=al1;  
al1=null;  
// El objeto creado por al1 sigue estando  
referenciado por al2→No se liberará su memoria
```

```
Alumno al1,al2  
al1=new Alumno();  
al1=null;  
// la memoria reservada quedará disponible para  
el recolector de basura
```

- La clase Object define el método finalize que es llamado por el recolector de basura cuando el objeto no va a utilizarse más
- Este método sólo es necesario redefinirlo en nuestras clases cuando se quiere hacer algún tipo de 'limpieza especial' como cerrar ficheros
- Puede ser necesario que después de hacer las operaciones de terminación de nuestra clase se desee que se realicen las operaciones normales de terminación. Para eso se llama al método finalize de Object o de la clase de la que se herede mediante: `super.finalize();`
- Debe declararse como `protected void finalize () throws throwable`

```
protected void finalize () throws throwable {  
    if (mifich != null) {  
        mifich.close();  
        mifich = null;  
    }  
    super.finalize();  
}
```

Clases y Herencia

- Para acceder a ellos no es necesario instanciar un objeto de esa clase. Se muestran poniendo delante de la declaración la palabra static. El primer atributo de este tipo que se utiliza es 'out' de la clase System.

```
static PrintStream out
```

The "standard" output stream.

- Un ejemplo de método estático puede ser valueOf de la clase Integer que convierte un int en Integer

```
static Integer valueOf(int i)
```

Returns a Integer instance representing the specified int value.

- Para llamar a un método o atributo static hay que poner delante el nombre de la clase

```
Integer onum3;  
onum3= Integer.valueOf(5);
```

- A continuación se muestran algunos elementos que pueden utilizarse en la declaración de una clase para modificar su significado/utilidad
- La mayoría de ellos se mostrarán con ejemplos más adelante

Modificador	Uso
abstract	Una clase abstracta no puede instanciarse por un objeto ya que tendrá algunos métodos abstractos que sólo están definidos y no implementados. La utilidad está en definir comportamientos genéricos. Estos quedarán implementados en una clase que herede de la abstracta
final	Una clase es final cuando no se pueden generar subclases de esta
public	Una clase declarada publica puede ser utilizada por otras clases. Si no tiene este modificador, sólo podrán utilizarla las clases del mismo paquete
implements int1,int2...,intn	La clase implementa diferentes interfaces
extends <clase>	Es la forma de expresar la herencia

- Continuando con el ejemplo del alumno vamos a hacer la clase alumno abstracta al incluirle un método abstracto Calificación
- Se va a hacer una clase AlumnoMaster que heredará de la anterior e implementará el método calificación

```
abstract public class Alumno {
    /** nombre del alumno */
    public String nombre;
    ...
    abstract public int Calificacion();
    public String getApellido() {
    ...
}
```

Proyecto Netbeans

DBVClases

```
public class AlumnoMaster extends Alumno{
    private int nota;
    /** Crea una nueva instancia de AlumnoMaster */
    public AlumnoMaster() {
        nota=8;
    }
    public int Calificacion() {
        return nota;
    }
}
```

Hereda de alumno

- Es importante tener claro que otras clases podrán ver los atributos, métodos o a la clase dependiendo del modificador que se utilice delante de la definición de cada elemento
- En la siguiente tabla se resume que visibilidad tienen elementos de la misma clase, del mismo paquete, de una subclase o cualquiera en el proyecto según el modificador utilizado

Modificador	Clase	Paquete	Subclase	Todos
private	Sí	N	N	N
(por defecto)	Sí	Sí	N	N
protected	Sí	Sí	Sí	N
public	Sí	Sí	Sí	Sí

- Los argumentos primitivos se pasan todos por valor es decir, cualquier modificación que se haga no afectará fuera del método
- El resto de argumentos Objetos o arrays se pasan por referencia. Se pasa 'un puntero/referencia' al objeto y cualquier modificación se reflejará inmediatamente fuera del método

- Desde la versión 5 se pueden definir métodos con un número variable de argumentos. Esto es válido para tipos primitivos como objetos.
- Se indica que el número de elementos es variable mediante:
 - tipo ... nombre
- A cada uno de los argumentos se accede como si fuera un índice de un array
- A continuación se muestra un ejemplo con tipos primitivos y otro con objetos

```
static public void listaNotas(int ... notas){
    int i;
    for (i=0;i<notas.length;i++) {
        System.out.println("nota"+i+": "+notas[i]);
    }
}
static public void listaAlumnos(AlumnoMaster ... alumnos){
    int i;
    for (i=0;i<alumnos.length;i++) {
        System.out.println("nombre"+i+": "+
            alumnos[i].getNombre());
    }
}
```

```
public static void main(String[] args) {
    AlumnoMaster al1,al2;
    al1=new AlumnoMaster();
    al2=new AlumnoMaster("Maria","Lopez");
    al1.setNombre("Juan");
    System.out.println(al1.Calificacion());
    listaNotas(4);
    listaNotas(4,5,7,10);
    listaAlumnos(al1,al2);
}
```

```
Resultado:
8
nota0: =4
nota0: =4
nota1: =5
nota2: =7
nota3: =10
nombre0: =Juan
nombre1: =Maria
```

- Cuando una clase hereda de otra super se utiliza para ejecutar algún método de la clase 'padre' así se distinguiría del método con el mismo nombre si se hubiera redefinido
- También se utiliza para llamar al constructor del padre
- Por ejemplo en el constructor de la clase AlumnoMaster

```
public AlumnoMaster(String nombre,String apellido) {  
    super(nombre,apellido);  
}
```

DBVClases

Clases Genéricas

- Hay situaciones en las que se necesita reutilizar una clase cambiando sólo algún tipo base, como por ejemplo en los Tipos Abstractos de Datos (TADs)
- Lo tradicional es copiar la clase y cambiar el tipo a mano o utilizar la herencia
- Desde la versión 5 de Java este problema puede solucionarse de una forma más elegante mediante los atributos y métodos genéricos
- Para ilustrar las posibilidades de va a implementar una Pila Genérica que será válida para cualquier tipo base
- Este ejemplo será de mucha utilidad para explicar varios conceptos como:
 - Creación de TADs
 - Uso de estructuras enlazadas en Java (interesante para los conocedores de punteros y listas enlazadas en C)
 - Explicación de Interfaces
 - Excepciones

- Se va a definir un tipo nodo genérico con una referencia al siguiente de la pila y un dato de tipo <E> (que se concretará por un tipo/objeto cuando se instancie un objeto de la clase CNode
 - Se recuerda como se hacía en C/C++

```
// Definición de tipo nodo en C/C++
// Usando punteros
struct TNode {
    E elem;
    TNode *sig;
}
```

```
// Definición de clase nodo en Java
// Usando referencias
public class CNode<E> {
    public E elem;
    public CNode<E> sig;
    public CNode() { }
}
```

- La pila se definirá también genérica y su atributo principal será una referencia a un objeto de la clase CNode

```
// Definición de atributos en C++
class Cpila {
    private:
        TNode *p
}
```

```
// Definición del atributo en Java
public class CPila<E> {
    CNode<E> p;
    ...}
}
```

- El código completo de la clase pila
- En este ejemplo se ven algunas de las diferentes operaciones que se pueden realizar con los objetos genéricos
 - Declaración
 - Inicialización
 - Instanciación
 - Parámetro de método
 - Valor de vuelta de función

```
public class CPila<E> {
    private CNode<E> p;
    public CPila() { p=null; }
    public void apilar(E elem) {
        CNode<E> nuevo=new CNode<E>();
        nuevo.elem=elem;
        nuevo.sig=p;
        p=nuevo;
    }
    public E cabeza(){
        E elem=null;
        if(!pilaVacia()){
            elem=p.elem;
        }
        return elem;
    }
    public void desapilar() {
        if(!pilaVacia()){
            p=p.sig;
        }
    }
    public boolean pilaVacia() {
        return (p==null);
    }
}
```

- La aplicación de los genéricos se ve en el siguiente ejemplo en el que se instancian dos pilas una de enteros y otra de cadenas de forma muy sencilla

```
public static void main(String[] args) {
    CPila<Integer> p1;
    CPila<String> p2;
    String elem2;
    boolean ok;
    int i;
    Integer elem=null;
    p1=new CPila<Integer>();
    p1.apilar(5); // Autoboxing 5 int-->5
    Integer
    for(i=1;i<10;i++){
        p1.apilar(i);
    }
    while (!p1.pilaVacía()){
        elem=p1.cabeza();
        p1.desapilar();
        System.out.print(elem+" ");
    }
}
```

DBVPilaGenerica

```
p2=new CPila<String>();
p2.apilar("genericos");
p2.apilar("de los");
p2.apilar("esto");
p2.apilar("curioso");
p2.apilar("esta");
System.out.println();
while (!p2.pilaVacía()){
    elem2=p2.cabeza();
    p2.desapilar();
    System.out.print(elem2+" ");
}
}
```

Resultado:
9 8 7 6 5 4 3 2 1 5
esta curioso esto de los genéricos

- Los genéricos sólo funcionan con objetos. Cuando se declara una instancia de tipo genérico el argumento de tipo que se pasa ha de ser un tipo clase, no pueden utilizarse tipos simples.
 - Siguiendo con el ejemplo GPila.java, la siguiente línea daría error:

```
GPila<int> p = new GPila<int>(); //ERROR
```

- Una referencia a una versión específica de un tipo genérico no es de tipo compatible con otra versión del mismo tipo genérico.
 - Ejemplo: no se puede instanciar GPila a dos tipos diferentes e intentar asignar ambas referencias:

```
GPila<Integer> pi = new GPila<Integer>();  
GPila<String> ps = new GPila<String>();  
pi = ps; //ERROR
```

- En un tipo genérico se pueden declarar más de un parámetro de tipo

```
public class GBox2<T, V> {
    T obj1;
    V obj2;
    GBox(T obj1, V obj2) {
        this.obj1 = obj1;
        this.obj2= obj2;
    }
    public void muestraTipos {
        System.out.println("El tipo de T es " + obj1.getClass().getName());
        System.out.println("El tipo de V es " + obj2.getClass().getName());
    }
}
```

```
public class GBox2Demo {
    public static void main(String[] args) {
        GBox2<Integer, String> b = new Gbox2<Integer, String>(5, "Platero");
        b.muestraTipos();
    }
}
```

- La salida de GBox2Demo.java sería:

```
El tipo de T es java.lang.Integer
El tipo de V es java.lang.String
```

- Los parámetros de tipo también pueden ser declarados en un método y en un constructor para crear **métodos y constructores genéricos**
- El ámbito del parámetro se limita al método o constructor en el que se ha declarado.
- Se puede declarar un método genérico que se incluya en una clase no genérica

```
public class GBox<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        GBox<Integer> integerBox = new GBox<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
        integerBox.inspect(5);
    }
}
```

- La salida de **GBox.java** sería:

```
T: java.lang.Integer
U: java.lang.String
T: java.lang.Integer
U: java.lang.Integer
```

- Se usan cuando se quiere restringir los posibles tipos que pueden ser pasados a un parámetro de tipo. Por ejemplo, un método que trabaja con números podría querer aceptar sólo instancias de **Number** o de sus subclases.
- Para declarar un tipo vinculado se utiliza la palabra clave **extends** (en este contexto se usa tanto para clases como para interfaces)

```
public class GBoxV<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text"); // ERROR!
    }
}
```

- La salida de **GBoxV.java** sería:

GBoxV.java:21: <U>inspect(U) in GBoxV<java.lang.Integer> cannot be applied to (java.lang.String)

integerBox.inspect("10");

^

1 error

- Una clase genérica puede formar parte de una jerarquía de clases.
- Aun cuando una subclase no utilice el parámetro de tipo de la superclase (genérica) debe especificar los parámetros requeridos por la misma.

```
class Gen<T> {
    T ob;
    Gen (T o) {
        ob = o;
    }
    ...
}
class Gen2<T> extends Gen<T> {
    Gen2 (T o) {
        super(o);
    }
}
```

- Una subclase puede añadir sus propios parámetros de tipo

```
class Gen2<T, V> extends Gen<T> {
    V ob2;
    Gen2 (T o, V o2) {
        super(o);
        ob2 = o2;
    }
}
```

- Una subclase genérica puede ser subclase de una no genérica.

```
class NoGen {
    int num;
    NoGen (int i) {
        num = i;
    }
    ...
}
class Gen<T> extends NoGen {
    T ob;
    Gen (T o, int i) {
        super(i);
        ob = o;
    }
    ...
}
```

- En Java es posible asignar un objeto de un tipo a un objeto de otro tipo si son compatibles. Por ejemplo yo puedo declarar una referencia de la clase Object y asignarle un Integer a un Object ya que Object es superclase de Integer.

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger; // OK
```

- Con genéricos Java permite realizar una invocación de tipo genérica pasándole un elemento de un tipo compatible con la definición:

```
// La definición de Box es: public class Box<T>
Box<Number> box = new Box<Number>();

// La definición de add es: public void add (T t)

box.add(new Integer(10)); // OK
box.add(new Double(10.1)); // OK
```

- Sin embargo no se puede pasar indistintamente un objeto de una clase genérica ya instanciada aunque el parámetro de tipo sea compatible:

```
public void boxTest(Box<Number> n){...}

//ERROR: Box<Integer> y Box<Double> no son subclasses de Box<Number>.
boxTest(Box<Integer>);
boxTest(Box<Double>);
```

Interfaces y Paquetes

- Una interfaz es como una clase en la que todos sus métodos son abstractos, es decir, se definen pero no se implementan
- Habrá clases que implementen una interfaz y deberán incluir e implementar todos los métodos de la interfaz
- Si la interfaz tiene atributos todos son estáticos
- Para seguir con el ejemplo de la pila, una pila puede implementarse como una estructura dinámica, como arrays, etc.
- Si aprendemos una forma mejor de implementar la pila, lo ideal es que las aplicaciones que la utilizaran no se vieran afectadas (no tener que modificarlas) y sólo vieran la mejora en la eficiencia.
- Para eso se define una interfaz que además es genérica de pila con los métodos que cualquier implementación debe realizar:

```
public interface IPila<E> {  
    public void apilar(E elem);  
    public E cabeza();  
    public void desapilar();  
    public boolean pilaVacía();  
}
```

- Para implementar una interfaz la definición de la clase debe incluir la palabra implements seguido del nombre de la interfaz

```
public class CPila<E> implements IPila<E> {...}
```

- Con eso la pila anterior funcionaria igual y además implementaría la interfaz
- Si posteriormente descubrimos que Java ya tenía un tipo Stack que puede ser más eficiente se podrá definir otra clase que implemente la interfaz IPila y utilice la clase Stack

```
public class CPilaStack<E> implements IPila<E> {
    private Stack<E> pila;

    public CPilaStack() {
        pila=new Stack<E>();
    }
    public void apilar(E elem) {
        pila.push(elem);
    }
    public E cabeza(){
        return pila.peek();
    }
}
```

```
public void desapilar() {
    pila.pop();
}
public boolean pilaVacía() {
    return pila.empty();
}
}
```

DBVInterfazPila

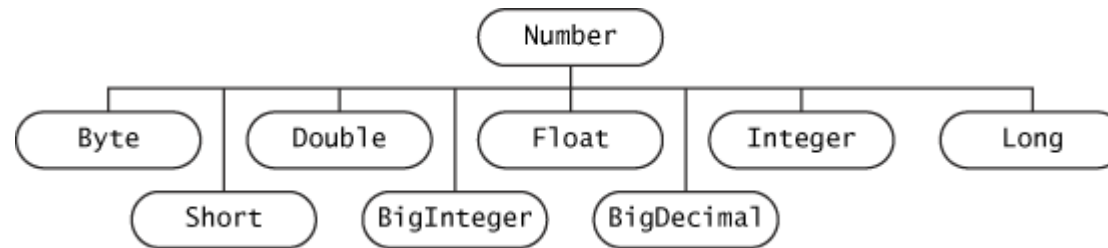
- Un paquete es una colección de elementos (clases, interfaces, enumerados y anotaciones) relacionados que provee protección de acceso y un espacio de nombre
- Un nombre de paquete coincide con los directorios en los que se almacenan las clase separados por puntos
- La primera línea del código de cada clase o interfaz debe indicar el paquete al que pertenece utilizando la palabra 'package'. Ese nombre debe coincidir con la subcarpeta en la que se encuentre Ejemplo: Si la clase 'alumno' esta dentro del paquete 'master' del paquete 'informatica' la primera línea del código será

```
package informatica.master;  
class Alumno {...}
```

- Hay una convenio para los nombre de los paquetes que es el nombre invertido del dominio en Internet. (com.compañia.paquete)

Números y cadenas

- Jerarquía de clases que tratan números



- ¿Para que necesito clases si tengo los tipos primitivos int, float, double o byte?
 - En algunas ocasiones se requiere un objeto, por ejemplo como parámetro a un método o a una definición genérica como `ArrayList<clase>` que puede ser instanciada como `ArrayList<Integer>` pero no como `ArrayList<int>`
 - En las clases asociadas a números se definen métodos para conversión de tipos además de valores de rango como `MIN_VALUE`
- Las clases `DecimalFormat` o `NumberFormat` permiten dar formato a los números teniendo en cuenta información de localización del sistema (moneda, idioma, puntos o comas para decimales, etc.)
- Desde la versión 5 se puede usar un formato de salida similar al clásico de C: `printf` y `format`. (Ver la clase `Formatter`)

- Ejemplos de creación, conversión a/de texto y formateo de números

```

Alumno al1,al2
int num1=5;
    String cad;
    Integer onum1,onum2;
    Float f1;
    cad=new String("25");
    onum1=new Integer(num1); // Constructor basado en int
    onum2=new Integer(cad); // Constructor basado en una cadena
    // toString se llama sólo si es necesario convertir un Integer a cadena
    System.out.print("Numeros="+ onum1.toString()+" "+ onum2);
    f1=new Float(45.54634);
    System.out.printf("%n%f",f1);// numero normal
    System.out.printf("%n%.3f",f1);// solo 3 decimales
    System.out.printf("%n%E",f1);// notacion científica
    System.out.printf("%n%15.3f",f1);// solo 3 decimales, 15 espacios para el número
    
```

DBVNumeros

- La clase Math permite operaciones matemáticas como seno, coseno, raíces cuadradas...
- Todos los métodos y atributos son estáticos

```

System.out.println("%n"+Math.abs(-20)+" "+ Math.cos(Math.PI));
    
```

- El tipo primitivo para caracteres es char. Su clase asociada es Character
- Desde la versión 5 existe la característica de autoboxing que convierte automáticamente un char en Character cuando es necesario. Para versiones anteriores la conversión debe hacerse manualmente mediante el constructor: Character(char value)
- Existen 3 clases para manejar cadenas de caracteres:

Clase	Uso
String	Cadenas que una vez creadas no cambian
StringBuffer	Cadenas que pueden modificarse en un entorno multihebra
StringBuilder	Cadenas muy eficientes para modificaciones pero que no pueden utilizarse en un entorno multihebra (Nuevo en Java 5)

- La clase Character permite además algunas operaciones interesantes como: `charValue`, `isDigit`, `isLetter`, `isLetterOrDigit`, `isLowerCase`, `isUpperCase`, `isWhiteSpace`, `toChars`, `toString`, `toLowerCase`, `toUpperCase`, `valueOf` cuyos detalles pueden verse en la API

```
public static void main(String[] args) {
    char car;
    Character ch1;
    StringBuilder sb1;
    car='d';
    ch1=Character.valueOf('d'); //desde 1.5 y preferible al constructor
    sb1=new StringBuilder();
    sb1.append("Caracter car=").append(car);
    sb1.append(" " + ch1);
    if(Character.isUpperCase(ch1.charValue())) {
        sb1.append(" Es mayusculas");
    } else {
        sb1.append(" Es minusculas");
    }
    System.out.println(sb1);
}
```

Resultado:

Caracter car=d Es minusculas

- El uso de StringBuffer y StringBuilder es similar. Algunos métodos relevantes: append, charAt, delete, insert, length, indexOf, substring

```
StringBuilder sb2,sal;
sb2=new StringBuilder("Cadena de prueba");
sal=new StringBuilder();
sal.append("Original="+sb2);
sb2.delete(3,5);
sal.append("\n"+sb2);
if (sb2.indexOf("Patata")!= -1) {
    sal.append("Es subcadena");
} else {
    sal.append("\nNo es subcadena");
}
System.out.println(sal);
```

Resultado:

Original=Cadena de prueba
Cada de prueba
No es subcadena

Dos formas de crear objetos String

```
String mistr="no cambiare";
String mistr2=new String("yo tampoco ");
```

DBVCharString

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
N	i	a	g	a	r	a	.		O		r	o	a	r		a	g	a	i	n	!

↘ charAt(0) charAt(9) ↗ charAt(length() - 1) ↗

- La clase **String** admite varios constructores
String() String(char[], int , int)
String(String s) String(byte[], int)
String(char[]) ...
- También pueden crearse implícitamente objetos de la clase **String** asignándoles un valor al crear la referencia. Ejemplos:

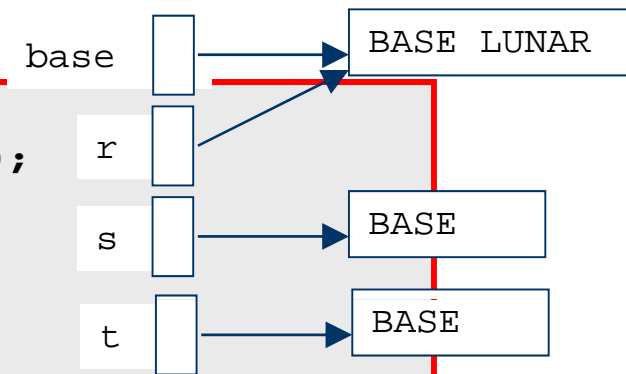
```
String s = new String();  
String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ;  
String s1 = new String(alfabeto);  
  
char[] vocales = { 'A' , 'E' , 'I' , 'O' , 'U' } ;  
String s3 = new String(vocales,2,3); // s3 = "IOU"  
  
byte[] letras = {65,66,67};  
String s2 = new String(letras,0); // s2 = "ABC"
```

- Si se crea un objeto StringBuffer a partir de una matriz el contenido se copia. Si se modifica la matriz después de crear la cadena el objeto StringBuffer no se altera.
- El operador de asignación = copia solo las referencias a objetos de la clase String, sin necesidad de duplicar su contenido.
- Para duplicar realmente el objeto hay que usar bien un constructor de copia, o bien el método toString()

```
public static void main(String[] args) {
    StringBuffer base = new StringBuffer("BASE");
    StringBuffer r = base;
    String s = new String (base);
    String t = base.toString();

    base.append(" LUNAR");

    System.out.println("r="+r);
    System.out.println("s="+s);
    System.out.println("t="+t);
    System.out.println("base="+base);
}
```



RESULTADO

```
r=BASE LUNAR
s=BASE
t= BASE
base=BASE LUNAR
```

- El operador + aplicado a referencias de la clase String provoca la creación de un nuevo objeto cuyo contenido concatena las cadenas de caracteres referenciadas. También puede usarse el operador += para incrementar el contenido de una misma cadena.

```
String titulo = "Ingeniero ";  
titulo = titulo + "de Caminos ";  
titulo += "Canales y Puertos";
```

- Este operador está sobrecargado para actuar razonablemente con objetos de cualquiera de las clases y tipos básicos predefinidos.

```
int dia = 24 ;  
String mes = "octubre" ;  
String fecha = dia + " de " + mes + " del " + '9' + '6';
```

- Cuidado con la asociatividad del operador +

```
String suma = "2+2="+2+2; // "2+2=22"
```

- Se puede sobrecargar implícitamente el operador de concatenación para que sea capaz de operar con objetos de cualquier clase. Para ello, la nueva clase debe definir el método `toString()` capaz de representar sus objetos mediante una cadena de caracteres.

```
class Punto {
    private double x,y ;
    ...
    public String toString () {
        return "Punto(" + x + "," + y + ")" ;
    }
    ...
}
class Programa {
    public static void main(String argv[]) {
        Punto p = new Punto(3,4);
        System.out.println("p="+p) ;
    }
}
```

RESULTADO

p=Punto(3.0,4.0)

```
public class Filename {
    private String fullPath;
    private char pathSeparator, extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String extension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    public String filename() { // gets filename without extension
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }

    public String path() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

```
public class FilenameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/mem/index.html";
        Filename myHomePage = new Filename(FPATH, '/', '.');

        System.out.println("Extension = " + myHomePage.extension());
        System.out.println("Filename = " + myHomePage.filename());
        System.out.println("Path = " + myHomePage.path());
    }
}
```

Diagram illustrating the string `/home/mem/index.html` and its components:

- `substring (dot + 1)`: Points to the extension `.html`.
- `substring (sep, dot)`: Points to the filename `index`.
- `sep = lastIndexOf ('/')`: Points to the last slash `/` before `mem`.
- `dot = lastIndexOf ('.')`: Points to the dot `.` before `html`.

Configuración de programas y acceso a recursos del sistema

- Los ficheros de propiedades son muy comunes en java para tareas de configuración. Tiene un formato propiedad=valor como en el ejemplo siguiente:

```
urlBase=http://altair.lcc.uma.es  
puerto=3000  
path=c:/programas/prueba
```

- La clase Properties de java permite leer este fichero de una forma muy sencilla, acceder a todas sus propiedades y modificarlas sin necesidad de recompilar
- En las aplicaciones normalmente se crea un fichero de configuración como el anterior y hay una clase con métodos y atributos estáticos para acceder a estas propiedades desde cualquier clase de la aplicación
- Veremos un ejemplo con un fichero llamado config.properties y dos clases una Configuracion.java que lee el fichero de configuración utilizando la clase Properties y otro fichero Main.java que contiene el main y un ejemplo de acceso a esos parámetros

DBVProperties

```
// Configuracion.java
package dbvproperties;
import java.util.Properties;
import java.io.*;

public class Configuracion {
    public static String urlBase;
    public static int puerto;
    public static String path;

    public static void Configurar() throws java.io.FileNotFoundException, java.io.IOException {
        Properties pr1=new Properties();
        FileInputStream config = new FileInputStream("./config.properties");
        pr1.load(config);
        config.close();
        // Lectura de propiedades el objeto Properties
        urlBase=pr1.getProperty("urlBase");
        puerto=(new Integer(pr1.getProperty("puerto"))).intValue();
        path=pr1.getProperty("path");
    }
}
```

```
// Main.java
package dbvproperties;

public class Main {
    public static void main(String[] args) throws java.io.FileNotFoundException, java.io.IOException {
        Configuracion.Configurar(); //Carga la configuración
        System.out.println("urlbase="+Configuracion.urlBase); // Accede a las propiedades
        System.out.println("puerto="+Configuracion.puerto);
        System.out.println("path="+Configuracion.path);
    }
}
```

- Con java a veces se pueden encontrar dificultades sobre la forma de acceder a los ficheros. Si el acceso se hace como a continuación, el fichero debe encontrarse donde se ejecute la aplicación:

```
FileInputStream config = new FileInputStream("./config.properties");
```

- Eso puede dar problemas cuando la aplicación se entregue en un solo fichero (.jar) por ejemplo. Otra posible solución es que se guarden los ficheros en un paquete y que sea Java la encargada de encontrarlo. Por ejemplo, si guardo mi fichero en el paquete mbgproperties y el fichero se llama main.properties quiero que java me abra el fichero mbgproperties/main.properties. Eso puede hacerse en dos partes
- Por un lado quiero que la aplicación sepa en que paquete se encuentra, así si este cambia no hay que modificar el código. Eso puede obtenerse haciendo:

```
Main.class.getPackage().getName()
```

- Con el nombre del paquete y el nombre del fichero quiero que Java lo abra y me devuelva en un InputStream. Eso nos lo ofrece la clase ClassLoader
- De esta forma el fichero se encontrará ya este ejecutando en línea, con Netbeans o con la aplicación y el fichero de configuración dentro de un .jar

```
InputStream f = ClassLoader.getResourceAsStream(Main.class.getPackage().getName() + "/" + FICHERO_CONFIGURACION);
```

- Una aplicación en su ciclo de vida:
 - Carga la configuración por defecto de un fichero “properties” desde una localización genérica.
 - Actualiza la configuración con las personalizaciones realizadas para el usuario desde una localización específica por usuario.
 - Al finalizar la ejecución almacena los cambios de la configuración en el fichero personal.

```
// Carga el fichero de propiedades generico
Properties defProps = new Properties();
// Intentando abrir el fichero mbgproperties/main.properties
InputStream f = ClassLoader.getResourceAsStream(Main.class.getPackage().getName() + "/" +
FICHERO_CONFIGURACION);
defProps.load(f);
f.close();
// Se sobrescriben las propiedades definidas en el fichero del usuario
Properties appProps = new Properties();
appProps.putAll(defProps); // Se copian las propiedades predefinidas a appProps
String fileName = System.getProperty("user.home") + "/" + FICHERO_CONFIG;
f = new FileInputStream(fileName);
appProps.load(f); // Se cargan manteniendo las que no estén en el fichero de usuario
f.close();

System.out.println(appProps);
```

MBGProperties

- Los argumentos del programa se reciben en args que es una cadena de cadenas

```
public static void main (String[] args) {
    // Lectura de argumentos
    System.out.println("Parametros leidos: ") ;
    for (String varent: args) {
        System.out.print(varent+" ");
    }
}
```

Librería para tratamiento de línea de comandos:
<http://jakarta.apache.org/commons/cli/>

- La clase System ofrece acceso al sistema como variables de entorno

```
// Mostrar una variable de entorno específica
System.out.println(System.getenv("path"));
// Mostrar todas la variables de entorno
Map<String, String> env = System.getenv();
for (String nombrevar : env.keySet()) {
    System.out.println(nombrevar + "=" + env.get(nombrevar));
}
// Ver las propiedades del sistema
Properties propsist;
propsist=System.getProperties();
propsist.list(System.out);
// Mostrar alguna propiedad
System.out.println("Bienvenido usuario: "+propsist.getProperty("user.name"));
System.out.println("Su sistema operativo es:"+propsist.getProperty("os.name"));
```

Propiedades del sistema

User.name
 Java.home
 Os.version
 Os.name
 file.separator
 path.separator
 line.separator...

Internacionalización

- Es el proceso de adaptación de una aplicación a diferentes lenguajes y regiones. El proceso de adaptación incluye traducción de texto, representaciones numéricas y fechas.
- Cada vez es más necesario que las aplicaciones estén preparadas para funcionar en distintos idiomas.
- Las posibilidades de internacionalización de Java permiten que con sólo añadir un fichero de texto se puedan tener nuestra aplicación en un nuevo idioma.
- Hay un fichero que se toma como referencia que es: `messages.properties`
- Cuando para un idioma un término no se haya definido se cogerá del fichero de referencia
- Para cada idioma se crea un fichero con el nombre:
`messages_es.properties` (español), `messages_es_ES.properties` (español de España), `messages_en_US.properties` (Inglés de EEUU),...
- En el siguiente ejemplo se muestra una clase `Messages` encargada de cargar la localización y una clase principal que la utiliza

`messages_es.properties`

```
hello=Hola\!
money=Dinero
number=Numero
date=Fecha
```

`messages_en_US.properties`

```
hello=Hello
money=Money
number=Number
date=Date
```

MBGI18n

```
package mbgi18n;

import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;

public class Messages {
    // La siguiente sentencia obtiene el Nombre del fichero Properties = mbgi18n.messages
    private static final String BUNDLE_NAME = Messages.class.getPackage().getName()+".messages";
    // Crea una tabla 'hash' con las entradas en el idioma elegido
    private static ResourceBundle RESOURCE_BUNDLE = ResourceBundle.getBundle(BUNDLE_NAME);
    // Metodo para Cambiar la localizacion
    public static void setLocale(Locale locale) {
        // Recarga las entradas de la tabla con la nueva localizacion
        RESOURCE_BUNDLE = ResourceBundle.getBundle(BUNDLE_NAME, locale);
    }
    // Metodo para consultar un termino
    public static String getString(String key) {
        try {
            return RESOURCE_BUNDLE.getString(key); // Devuelve una entrada de la tabla
        } catch (MissingResourceException e) {
            return '!' + key + '!';
        }
    }
}
```

- Al proceso de adaptación de un software internacionalizado a una lengua y región concreta se le denomina “Localización”

```
package mbgi18n;
import java.text.DateFormat;
import java.text.NumberFormat;
import java.util.Date;
import java.util.Locale;

public class Main {
    // Texto sin internacionalizar
    public static void nol18N () {
        String str = "Hello World!";
        Double number = 132532.34;
        Date date = new Date();
        System.out.println(str);
        System.out.println(number);
        System.out.println(date);
    }
}
```

MBGI18n

// Salida internacionalizada

```
public static void I18N (Locale locale) {
    // Objetos de internacionalizacion dateFormat, numberFormat y la clase Messages
    Messages.setLocale(locale);
    DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.DEFAULT, locale);
    NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
    NumberFormat currencyFormat=NumberFormat.getCurrencyInstance(locale);
    // Creación de variables para mostrar
    String str = Messages.getString("hello");
    Double number = 132532.34;
    Date date = new Date();
    Double money= 150.23;
    // Mostrar cadenas, numeros, fechas y moneda de forma localizada
    System.out.println(locale); System.out.println(str);
    System.out.println(Messages.getString("number") + ":" + numberFormat.format(number));
    System.out.println(Messages.getString("money") + ":" + currencyFormat.format(money));
    System.out.println(Messages.getString("date") + ":" + dateFormat.format(date)); }
public static void main(String[] args) {
    System.out.println("nol18N:\n");
    nol18N();// Para mostrar datos sin localizar
    System.out.println("\n\nI18N defaultLocale:\n");
    Locale locale = Locale.getDefault(); // Localización por defecto (español)
    I18N(locale);
    System.out.println("\n\nI18N en_US Locale:\n");
    locale = new Locale("en", "US"); // Localización de Ingles de EEUU
    I18N(locale);
}
```

```
es_ES
Hola!
Numero:132.532,34
Dinero:150,23 €
Fecha:17-nov-2006
```

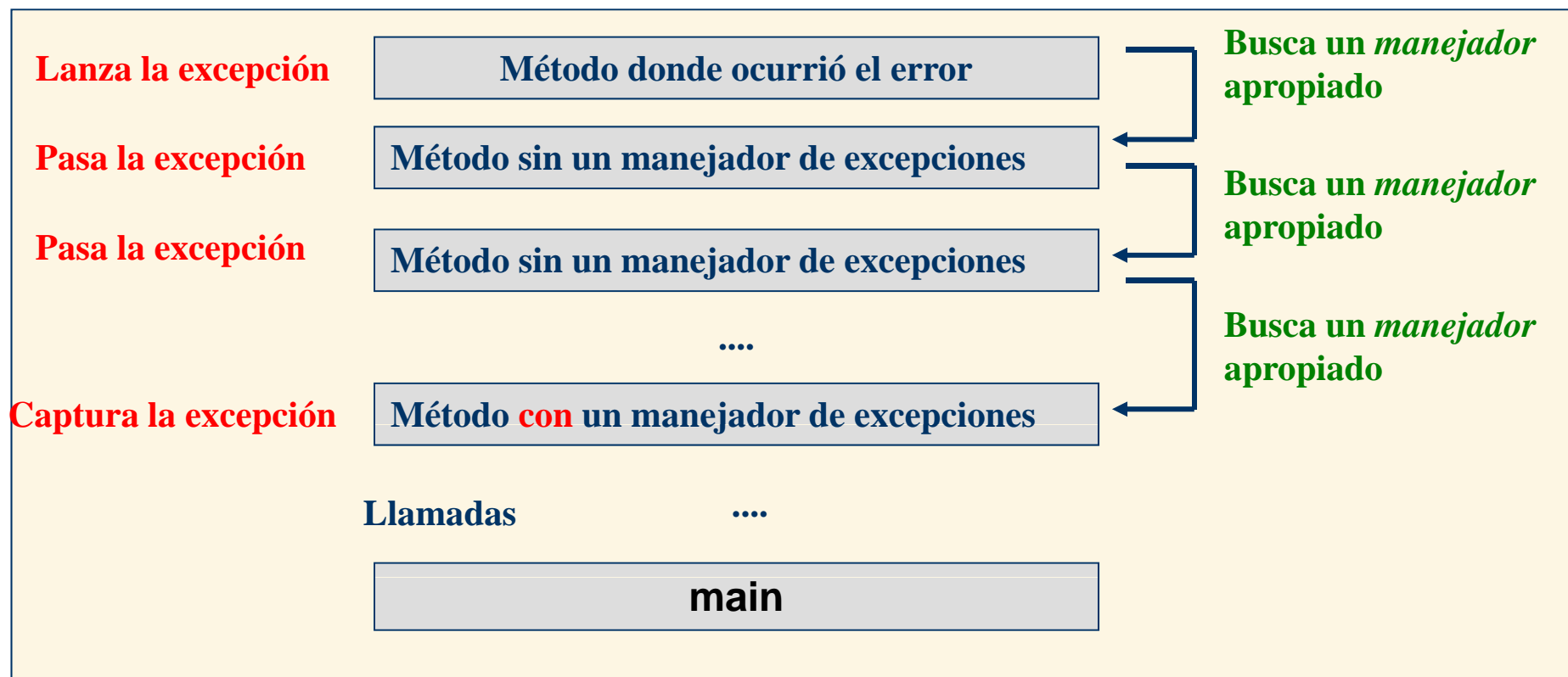
```
I18N en_US Locale:

en_US
Hello World!
Number:132,532.34
Money:$150.23
Date:Nov 17, 2006
```

Excepciones

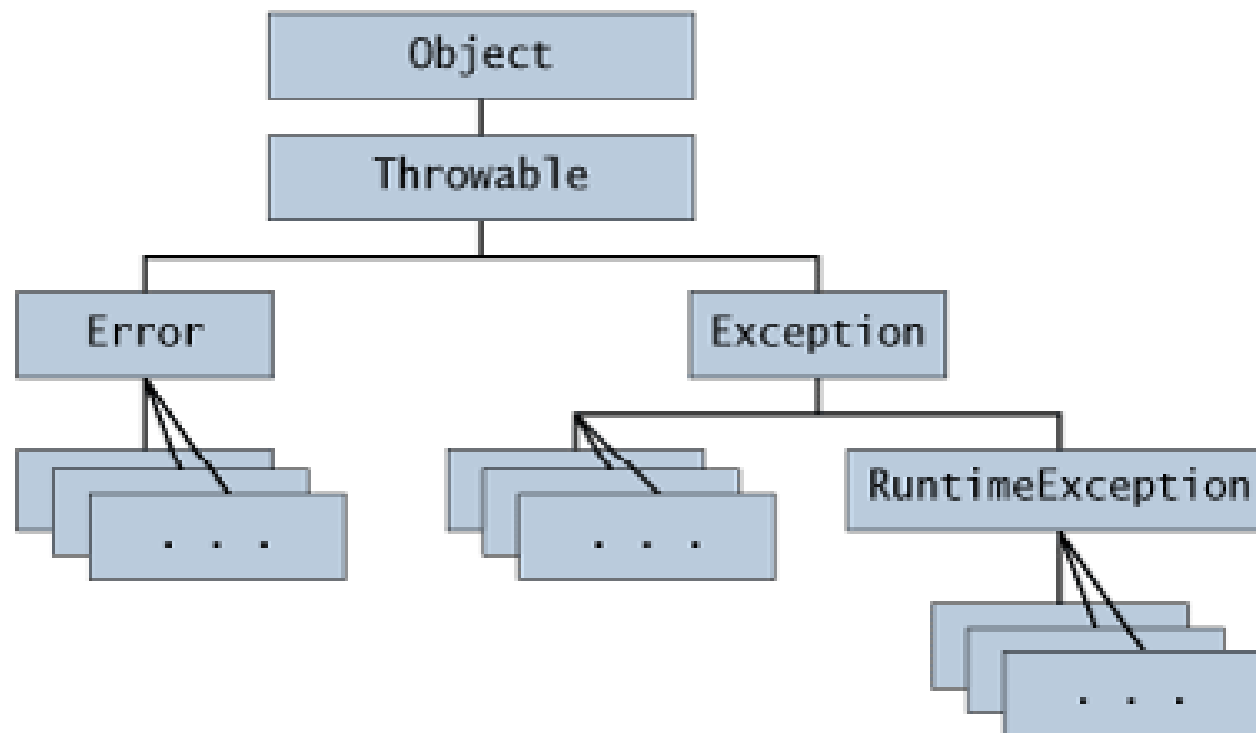
- Una **excepción** es un evento que ocurre durante la ejecución de un programa e interrumpe el flujo normal de instrucciones del programa (p.ej. fallo en apertura de ficheros, desbordamiento numérico, etc.)
- **JAVA** permite **capturar** estas **excepciones**, efectuar las acciones necesarias para **recuperar el error** y **proseguir la ejecución**.
- Cuando ocurre un error dentro de un método éste lanza una excepción. **Lanzar una excepción** consiste en:
 - ✓ crear un objeto que contiene información sobre el error incluyendo su tipo y el estado del programa cuando éste ocurrió
 - ✓ enviarlo al entorno de ejecución de java

- Cuando el entorno de ejecución recibe una excepción intenta encontrar a alguien que gestione el error. Los candidatos para **gestionar el error** son los métodos que han sido llamados hasta llegar al que generó el error: *pila de llamadas*
- La ejecución secuencial del programa se interrumpe y se continua donde se captura la excepción



- **Comprobadas:** representan condiciones excepcionales a las que un programa bien escrito debería **anticiparse** y de las que se debería **recuperar**. Por ejemplo “intentar abrir un fichero que no existe”, el programa podría notificar el error al usuario y pedirle que introduzca el nombre correcto del fichero.
- **No comprobadas:** reflejan errores de los que **no es posible recuperarse** de forma razonable durante la ejecución. Este tipo de excepciones pueden ser de dos tipos:
 - **Errores:** son condiciones excepcionales **externas a la aplicación** y por lo tanto ni se pueden prever ni es posible recuperarse de ellas. Por ejemplo, “el programa intenta abrir un fichero pero no puede leerlo porque el disco duro está dañado”. El programa puede capturar esta excepción pero lo único que puede hacer es notificarle al usuario el error y salir.
 - **Excepciones en tiempo de ejecución:** son condiciones excepcionales **internas a la aplicación** que surgen debido a algún **error en la lógica del programa**. Éste no es capaz de preverlas y por lo tanto de recuperarse de ellos. Por ejemplo, “intentar acceder fuera de los límites de un array”. La aplicación puede capturar la excepción y notificar el error para que el programador pueda localizar dentro del código el fallo que provocó que esa excepción ocurriera y arreglar el programa.

- La biblioteca estándar de JAVA contiene una jerarquía de clases de excepciones predefinidas
(<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/package-tree.html>)
- Todas las clases de excepciones son subclases de la clase **Throwable**
- El programador puede definir sus propias excepciones como clases que heredan de **Exception**



java.lang.Throwable (implements java.io.Serializable)

└─> java.lang.Exception

└─> java.lang.Error

□ java.lang.AssertionError

□ java.lang.LinkageError

■ java.lang.ClassCircularityError

■ java.lang.ClassFormatError

□ java.lang.UnsupportedClassVersionError

■ java.lang.ExceptionInInitializerError

■ java.lang.IncompatibleClassChangeError

□ java.lang.AbstractMethodError

□ java.lang.IllegalAccessError

□ java.lang.InstantiationException

□ java.lang.NoSuchFieldError

□ java.lang.NoSuchMethodError

■ java.lang.NoClassDefFoundError

■ java.lang.UnsatisfiedLinkError

■ java.lang.VerifyError

□ java.lang.ThreadDeath

□ java.lang.VirtualMachineError

■ java.lang.InternalError

■ java.lang.OutOfMemoryError

■ java.lang.StackOverflowError

■ java.lang.UnknownError

Errores

Excepciones
en tiempo de
ejecución

□ java.lang.ClassNotFoundException

□ java.lang.CloneNotSupportedException

□ java.lang.IllegalAccessException

□ java.lang.InstantiationException

□ java.lang.InterruptedException

□ java.lang.NoSuchFieldException

□ java.lang.NoSuchMethodException

□ java.lang.RuntimeException

■ java.lang.ArithmeticException

■ java.lang.ArrayStoreException

■ java.lang.ClassCastException

■ java.lang.EnumConstantNotPresentException

■ java.lang.IllegalArgumentException

□ java.lang.IllegalThreadStateException

□ java.lang.NumberFormatException

■ java.lang.IllegalMonitorStateException

■ java.lang.IllegalStateException

■ java.lang.IndexOutOfBoundsException

□ java.lang.ArrayIndexOutOfBoundsException

□ java.lang.StringIndexOutOfBoundsException

■ java.lang.NegativeArraySizeException

■ java.lang.NullPointerException

■ java.lang.SecurityException

■ java.lang.TypeNotPresentException

■ java.lang.UnsupportedOperationException

Excepciones
comprobadas

- El tratamiento de excepciones se realiza mediante las sentencias:
 - *try-catch-finally* que captura y maneja una excepción
 - *throws* que indica los tipos de excepciones que un método puede lanzar
 - *throw* que lanza un objeto Exception de forma explícita

```
try {  
    // bloque de código que controla los errores  
}  
catch (ExceptionType1 exOb1) {  
    // gestor de excepciones para la Excepción de tipo 1  
}  
catch (ExceptionType2 exOb2) {  
    // gestor de excepciones para la Excepción de tipo 2  
}  
// ...  
finally {  
    // bloque de código que ha de ejecutarse antes de  
    // que termine el bloque try  
}
```

- La parte **try** de la sentencia contiene un segmento de código en el que se prevé que se puede producir alguna excepción. Este código se ejecuta normalmente.
- Si en tiempo de ejecución se produce alguna excepción de tipo **ExceptionType** el control de flujo salta a la sentencia **catch** y se ejecuta el bloque de sentencias asociado.

- La sentencia **finally** no captura la excepción. Se ejecuta siempre, se haya capturado o no la excepción. Generalmente se usa para liberar recursos como ficheros abiertos, conexiones a bases de datos, etc.
- Una vez que se ha recogido y tratado la excepción, el programa continúa por la línea que sigue a la sentencia try-catch-finally
- Las excepciones que no se recogen con una sentencia catch, las captura el intérprete de JAVA, que manda un mensaje a la consola y detiene la ejecución del programa.

- Una **sentencia catch** captura excepciones del tipo especificado más todas las **subclases** de esa excepción.
- Cuando existen **cláusulas catch múltiples** se ejecuta la primera cuyo tipo coincida con la excepción que se ha producido.
 - Una vez que se ha ejecutado una, las demás se evitan y la ejecución continúa detrás del bloque try-catch-finally
 - Cuando se utilizan varias sentencias catch las **subclases de excepción** han de ir **delante de sus superclases**.

```

/* Este programa contiene un error. Si se genera una excepción de tipo
ArithmeticException, al ser subclase de Exception, sería capturada por
la primera sentencia catch, luego la segunda contendría un código
inalcanzable
*/
class SuperSubCatch {
    public static void main(String argv[]) {
        try {
            int a = 0;
            int b = 5/a;
        } catch (Exception e) {
            System.out.println("Excepción genérica");
        } catch (ArithmeticException e) {
            System.out.println("Division por cero,este código
                                nunca se alcanza");
        }
        System.out.println("Fin de programa");
    }
}

```

SuperSubCatch.java

```
class ProbarExcepciones {
    public static void main(String argv[]) {
        int a = (int) (Math.random() * 20) - 10 ; // numero entre -9 y 9
        int b = (int) (Math.random() * 20) - 10 ; // numero entre -9 y 9
        int[] m ;
        try {
            m = (a>0) ? new int[a] : null;
            m[0] = 1000 / b;
            m[b] = b;
        } catch (ArithmeticException e) {
            System.out.println("Division por cero");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("El rango de la matriz no es correcto");
        } finally {
            System.out.println("a="+a+ " b="+b);
        }
        System.out.println("Fin de programa");
    }
}
```

ProbarExcepciones.java

Probar el ejemplo:

para a = -1, b = 1;
 para a = 8, b = 0;
 para a = 8, b = 5;
 para a = 2, b = 5;

```
class ProbarExcepciones {
    public static void main(String argv[]) {
        ...
        try {
            ...
        } catch (ArithmeticException e) {
            System.out.println("Se ha producido la excepción: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("El rango de la matriz no es correcto");
        } finally {
            System.out.println("a="+a+ " b="+b);
        }
        System.out.println("Fin de programa");
    }
}
```

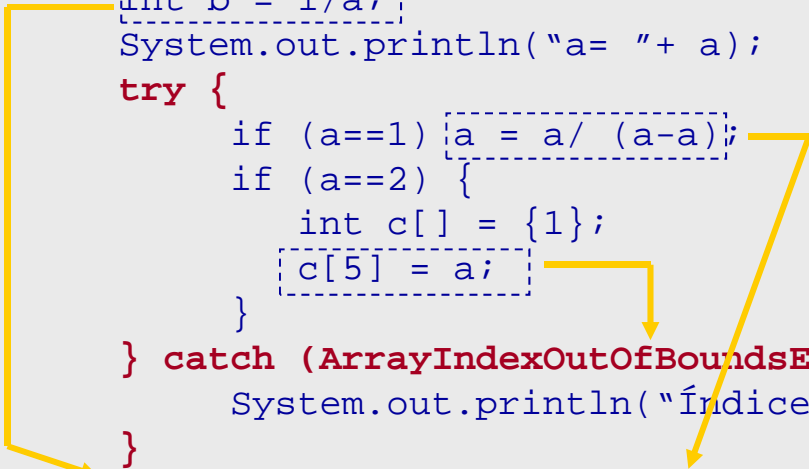
NOTA: La clase **Throwable** sobrescribe el método **toString()** definido por **Object** y devuelve una cadena que contiene la descripción de la excepción. Modificando el ejemplo anterior de la forma que se muestra, cuando se produzca una división por cero aparecería en la consola el mensaje:

Se ha producido la excepción: java.lang.ArithmeticException: / by zero

- Las sentencias **try** pueden **anidarse**, bien sea directamente en bloques de un **mismo método** o bien por causa de la **llamada de un método a otro** que provoca un anidamiento en el flujo de programa.
 - Cuando se produce una excepción dentro de un bloque **try**, **primero** se **busca** alguna sentencia **catch asociada** que sea capaz de capturar la excepción.
 - Si la excepción no es capturada, la sentencia **try-catch** es considerada como una sentencia que ha producido una excepción, por lo que se **busca entre las sentencias catch del bloque exterior**, y así sucesivamente hasta el bloque más externo.
 - Si la excepción no es capturada (se llega al bloque mas externo), **el intérprete de JAVA se hace cargo de ella** deteniendo la ejecución y se mostrándola por la consola.

- Ejemplo de anidamiento dentro del mismo método.

```
class ProbarExcepcionesAnidadas {
    public static void main(String argv[]) {
        try {
            int a = args.length;
            int b = 1/a;
            System.out.println("a= " + a);
            try {
                if (a==1) a = a/ (a-a);
                if (a==2) {
                    int c[] = {1};
                    c[5] = a;
                }
            } catch (ArrayIndexOutOfBoundsException e) { /* 1 */
                System.out.println("Índice del array fuera de límites");
            }
        } catch (ArithmeticException e) { /* 2 */
            System.out.println("Division por cero");
        }
    }
}
```



ProbarExcepcionesAnidadas.java

- Ejemplo de anidamiento en llamadas a métodos.

```
class ProbarExcepciones2 {
    public static void metodo (int a, int b) {
        int[] m ;
        try {
            m = (a>0) ? new int[a] : null;
            m[0] = 1000 / b;
            m[b] = b;
        } catch (ArithmeticException e) {
            System.out.println("Division por cero");
        } finally {
            System.out.println("a="+ a + " b="+ b);
        }
    }

    public static void main(String argv[]) {
        int a = (int) (Math.random() * 20) - 10 ; // numero entre -9 y 9
        int b = (int) (Math.random() * 20) - 10 ; // numero entre -9 y 9
        try {
            metodo(a,b);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("El rango de la matriz no es correcto");
        }
        System.out.println("Fin de programa");
    }
}
```

Diagram illustrating the flow of execution and exception handling:

- The `main` method calls `metodo(a,b)`.
- Inside `metodo`, a `try` block attempts to create an array `m` and perform calculations.
- If an `ArithmeticException` occurs (e.g., division by zero), the `catch` block handles it, printing "Division por cero".
- The `finally` block always executes, printing "a="+ a + " b="+ b.
- If an `ArrayIndexOutOfBoundsException` occurs (e.g., accessing `m[b]` where `b` is out of bounds), the `catch` block in `main` handles it, printing "El rango de la matriz no es correcto".
- After the `try-catch` block in `main`, it prints "Fin de programa".

ProbarExcepciones2.java

- Si un método es capaz de causar una excepción que él mismo no es capaz de gestionar debe especificar este comportamiento
- La cláusula **throws** se usa para **indicar qué excepciones comprobadas lanza un método y no captura** de modo que los métodos que lo llamen sean conscientes y actúen en consecuencia (capturándola o volviéndola a lanzar).

```
tipo nombre_método (lista_parámetros) throws lista_de_excepciones {  
    // cuerpo del método  
}
```

- Un método puede lanzar varias clases diferentes de excepciones, todas ellas extensiones de una clase particular de excepción y declarar sólo su superclase. Sin embargo, haciendo esto se oculta información interesante para el programador.

- Si un método no tiene cláusula **throws** no significa que no pueda generar un error o una excepción en tiempo de ejecución.
- Cuando se redefine un método heredado o se implementa uno abstracto:
 - no se permite que se declaren más excepciones en la cláusula **throws** que las que declaran los métodos originales.
 - se pueden lanzar subtipos de las excepciones declaradas

```

/* Este programa contiene un error. Si la apertura del fichero falla se
   genera una excepción del tipo IOException y el método writeList ni lo
   captura (no hay cláusula catch) ni indica que este error se puede
   producir.
*/
public void writeList() {
    PrintWriter out = new PrintWriter (new FileWriter("OutFile.txt"));
    ...
    out.close();
}

```

¿Cómo lo sabemos? Porque la definición del constructor `FileWriter` así lo indica:

```
public FileWriter (String fileName) throws IOException;
```

- ¿Cómo lo solucionamos? Hay dos opciones

```
/* 1 - Se captura la excepción */
public void writeList() {
    try {
        PrintWriter out = new PrintWriter (new FileWriter("OutFile.txt"));
        ...
    } catch (IOException e) {
        System.err.println("Caught IOException:" + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

```
/* 2 - Se lanza la excepción */
public void writeList() throws IOException {
    PrintWriter out = new PrintWriter (new FileWriter("OutFile.txt"));
    ...
    out.close();
}
```

- Las excepciones son objetos, y por tanto, se pueden crear excepciones en tiempo de ejecución, mediante el operador **new**
- Si el programador detecta alguna situación anómala de funcionamiento, puede lanzar explícitamente una excepción. utilizando para ello una sentencia **throw** .

```
throw instancia_que_se_puede_lanzar;
```

```
class ProbarExcepciones {  
    public static void demo(int[] m) {  
        if (m==null) {  
            throw new NullPointerException("demo") ;  
        }  
        ...  
    }  
}
```

- Se pueden definir nuevas clases de excepciones siempre que sean herederas de la clase `Exception`, o alguna de sus descendientes, lo que permite manejar situaciones anómalas específicas del programa aumentando así su robustez.
- No es necesario que estas subclases implementen nada, simplemente su presencia en el sistema permitirá que se usen como excepciones
- La clase `Exception` no define ningún método por sí sola pero hereda todos los de `Throwable`. Estos métodos pueden sobrescribirse en las clases de excepciones propias.

```
class AccesoIllegal extends Exception {  
    private int posicion ;  
    AccesoIllegal(int i) {  
        posicion = i ;  
    }  
    public String toString() {  
        return "Acceso Ilegal desde " + posicion ;  
    }  
}
```

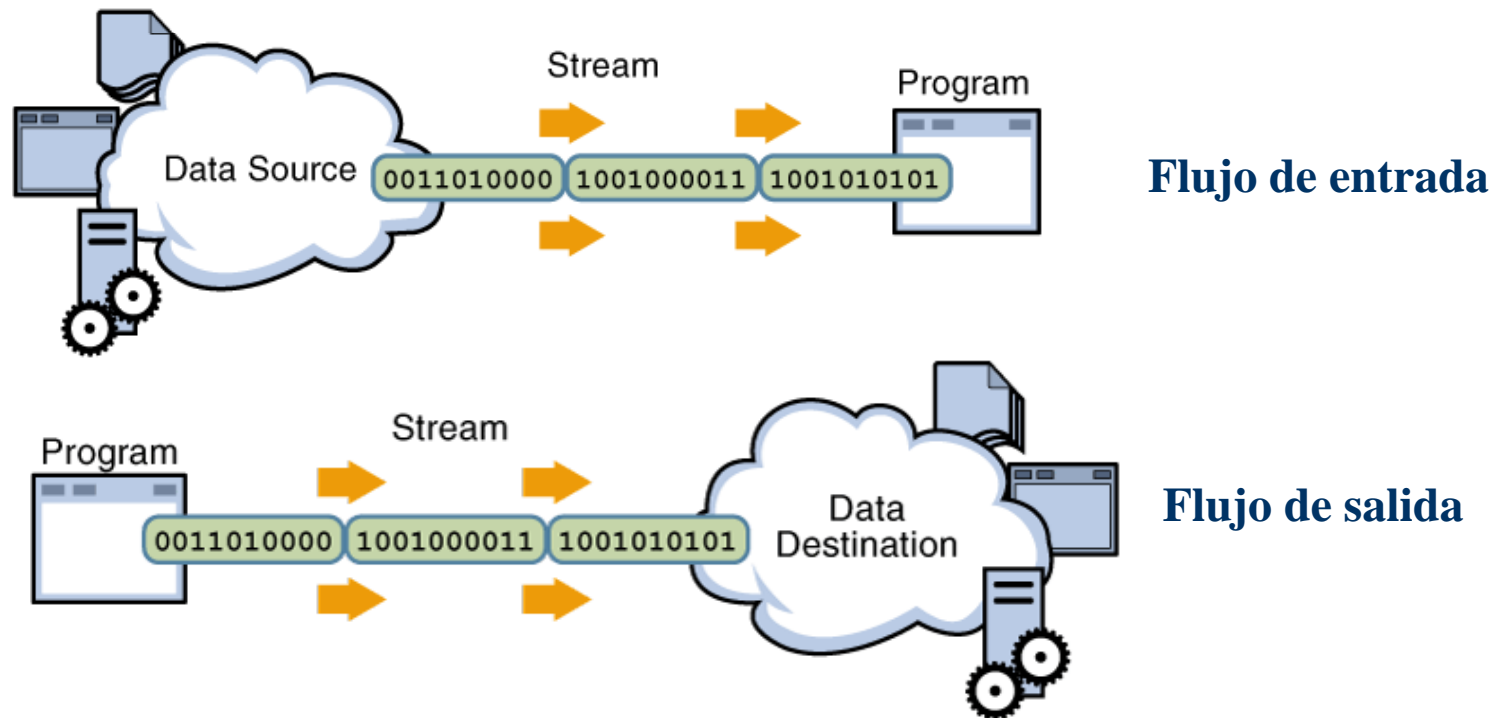
Accesollegal.java

```
class ExcepcionesPropias {
    public static void demo(int i) throws AccesoIllegal {
        if ( (i % 2) == 0 ) {
            throw new AccesoIllegal(i);
        }
        System.out.println("Fin del metodo");
    }
    public static void main(String argv[]) {
        int i = (int) (Math.random() * 20); // numero entre 0 y 9
        try {
            demo(i) ;
        } catch (AccesoIllegal e)
            System.out.println(e) ;
        }
        System.out.println("Fin de programa");
    }
}
```

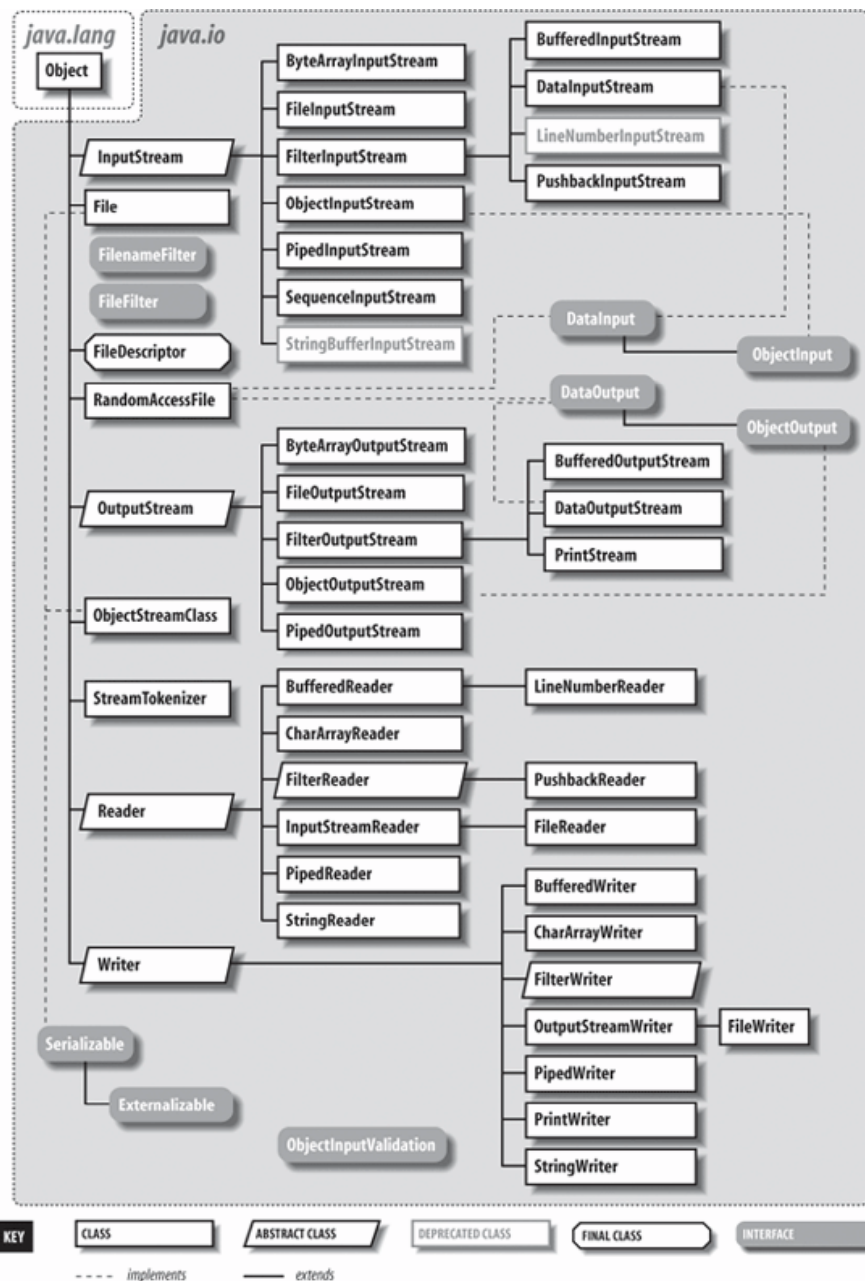
ExcepcionesPropias.java

Entrada/Salida

- Java define la entrada/salida en términos de flujos (streams). Los streams son secuencias ordenadas de datos que tienen una fuente (si son de entrada) o un destino (si son de salida)
- Por el sistema de E/S de Java, un stream está ligado a un dispositivo físico de modo que resulte igualmente fácil leer/escribir un fichero en el disco, desde el teclado, en la pantalla o en un archivo remoto a través de la red. Cualquiera de estos canales puede ser un flujo ("stream") de entrada/salida.



- Java define dos tipos de flujos:
 - de **bytes**: E/S basada en datos (binaria).
 - de **caracteres**: E/S basada en texto. A bajo nivel toda la E/S está orientada a bytes. Los flujos de caracteres simplemente proporcionan un medio para el manejo de caracteres de forma mas cómoda. Los caracteres en Java ocupan 16 bits. Esto permite utilizar el alfabeto UNICODE de 65535 caracteres. Cuando se utilizan caracteres del alfabeto ASCII (hasta 256) no hay ninguna diferencia en el comportamiento de las clases de lectura/escritura de bytes con las de lectura/escritura de caracteres
- Las clases para el manejo de la E/S pertenecen al paquete **java.io**
 - por cada clase que representa un flujo de entrada existe una correspondiente que representa un flujo de salida.
 - los flujos de entrada de caracteres se llaman **lectores** y los de salida **escritores**.
- Flujos de bytes
 - Se definen a través de dos jerarquías de clases encabezadas por las clases abstractas **InputStream** y **OutputStream**
 - Definen dos métodos abstractos **read** y **write** que leen y escriben bytes de datos y son sobrescritos por sus descendientes
- Flujos de caracteres
 - Se definen a través de dos jerarquías de clases encabezadas por las clases abstractas **Reader** y **Writer**
 - Definen dos métodos abstractos **read** y **write** que leen y escriben caracteres de los datos y son sobrescritos por sus descendientes



•InputStream / OutputStream

- Todos los flujos de bytes extienden estas clases. Definen la funcionalidad básica para leer o escribir una secuencia de bytes sin estructura.

•Reader/Writer

- Todos los flujos de caracteres implementan estas clases. Definen la funcionalidad básica para leer o escribir una secuencia de caracteres (Unicode).

•InputStreamReader/OutputStreamWriter

- Clases “puente” que convierten bytes a caracteres y viceversa.

•DataInputStream/DataOutputStream

- Flujos especializados en leer y escribir tipos de datos simples como numéricos primitivos y Strings, en un formato universal.

•ObjectInputStream/ObjectOutputStream

- Flujos especializados en escribir y reconstruir objetos serializados.

•BufferedInputStream/BufferedOutputStream/BufferedReader/BufferedWriter

- Para ganar eficiencia almacenan los datos en buffers para evitar sucesivas lecturas y escrituras sobre el stream.

•PrintWriter

- Flujo de caracteres que facilita la tarea de imprimir texto

•PipedInputStream/PipedOutputStream/PipedReader/PipedWriter

- Se utilizan como parejas de entrada/salida de modo que los datos que se escriben en el stream de salida de una pareja son los que se leen en el de entrada.

•FileInputStream/FileOutputStream/FileReader/FileWriter

- Se usan para lecturas y escrituras de ficheros del sistema de archivos local.

- La clase **System** del paquete **java.lang** contiene tres variables estáticas de flujo de bytes predefinidas: **in**, **out** y **err**, que hacen referencia a la entrada (teclado), la salida (monitor) y al flujo de error (monitor) estándares.
 - out** y **err** son de la clase **PrintStream**
 - in** es un **InputStream**
- A continuación se presentan algunos ejemplos que muestran como leer de la entrada estándar

```
class EntradaConsola {
    public static void main (String args[]) throws IOException {
        char c;
        BufferedReader entrada = new BufferedReader (new InputStreamReader (System.in));
        System.out.println("Introduzca caracteres. 'q' para salir");
        do {
            c = (char) entrada.read();
            System.out.println(c);
        }while (c!='q');
    }
}
```

EntradaConsola.java

```
class EntradaConsola2 {
    public static void main (String args[]) throws IOException {
        String str;
        BufferedReader entrada = new BufferedReader (new InputStreamReader(System.in));

        System.out.println("Introduzca líneas de texto.");
        System.out.println("Introduzca 'stop' para salir.");
        do {
            str = entrada.readLine();
            System.out.println(str);
        }while (!str.equals("stop"));
    }
}
```

EntradaConsola2.java

```
class EntradaConsola3 {
    public static void main (String args[]) {
        byte nombre[] = new byte[100];
        int num = 0;
        System.out.println("Nombre : ");
        try {
            num = System.in.read(nombre);
            System.out.print("Hola ");
            System.out.write(nombre,0,num);
        }
        catch (IOException e) {
            System.err.print("Error de E/S");
        }
    }
}
```

EntradaConsola3.java

- Los siguientes ejemplos explican como leer de la entrada estándar un entero (o cualquier otro dato numérico).
 - El **primero** de ellos (ReadStdInt.java) muestra como debe hacerse usando una clase puente para convertir el flujo de bytes de la entrada estándar en un flujo de caracteres que facilite la lectura de los datos.
 - El segundo (ReadStdInt15.java) es un ejemplo de uso de la clase **java.util.Scanner** para la lectura de datos de entrada si se dispone de la versión 5.0 de Java.

```
public class ReadStdInt {
    public static void main(String[] ap) {
        String line = null;
        int val = 0;
        try {
            BufferedReader is = new BufferedReader(new InputStreamReader(System.in));
            line = is.readLine();
            val = Integer.parseInt(line);
        } catch (NumberFormatException ex) {
            System.err.println("No es un número válido: " + line);
        } catch (IOException e) {
            System.err.println("IO ERROR: " + e);
        }
        System.out.println("He leído este número: " + val);
    }
}
```

ReadStdInt.java

- Los siguientes ejemplos explican como leer de la entrada estándar un entero (o cualquier otro dato numérico).
 - El **primero** de ellos (ReadStdInt.java) muestra como debe hacerse usando una clase puente para convertir el flujo de bytes de la entrada estándar en un flujo de caracteres que facilite la lectura de los datos.
 - El segundo (ReadStdInt15.java) es un ejemplo de uso de la clase **java.util.Scanner** para la lectura de datos de entrada si se dispone de la versión 5.0 de Java.

```
public class ReadStdInt15 {
    public static void main(String[] ap) {
        int val;
        try {
            Scanner sc = new Scanner(System.in); // Necesita J2SE 1.5
            val = sc.nextInt();
        } catch (InputMismatchException ex) {
            System.err.println("No es un número válido: " + ex);
            return;
        }
        System.out.println("He leído este número: " + val);
    }
}
```

ReadStdInt15.java

- En Java los archivos se consideran flujos de bytes que se pueden envolver en un objeto basado en caracteres
- Las clases que se usan para manejar archivos como un flujo de entrada o de salida son: **FileInputStream**, **FileOutputStream**, **FileReader** y **FileWriter**
- Estas clases contienen constructores que pueden crear un flujo a partir del nombre del archivo, de un objeto **File** o de un objeto **FileDescriptor**:
 - La clase **File** no pertenece a la jerarquía de clases de entrada y salida. Sirve para manejar ficheros del sistema operativo y conocer sus propiedades: nombre, fecha, tamaño, subdirectorios, etc. pero no accede a su contenido.
 - La clase **FileDescriptor** representa un valor dependiente del sistema que describe a un archivo abierto. Representa la conexión al archivo en el sistema de ficheros que es usada por un flujo de bytes. Se obtiene a través del método **getFD** del flujo.

```
import java.io.*;
/* Para utilizar este programa especifique el archivo que
   quiere ver en la línea de comandos*/
public class ShowFile {
    public static void main(String args[]) {
        int i;
        FileInputStream fin;
        try {
            fin = new FileInputStream(args[0]);
            do {
                i = fin.read();
                if (i!=-1) System.out.print((char)i);
            }while (i!= -1);
        } catch (FileNotFoundException e){
            System.out.println("Archivo no encontrado");
        } catch (IOException e){
            System.out.println("Error en la lectura del fichero");
        } catch (ArrayIndexOutOfBoundsException e){
            System.out.println("Escriba: ShowFile nombre_del_fichero");
        }
    }
}
```

ShowFile.java

```
import java.io.*;
class CopyFile1 {
    public static void main(String argv[]) throws IOException {
        InputStream in;
        OutputStream out;
        if (argv.length<2) {
            System.out.println("java CopiaFichero <fichEntrada> <fichSalida>");
            System.exit(0);
        }
        try {
            in= new FileInputStream(argv[0]);
        } catch(FileNotFoundException e) {
            System.out.println("Fichero de entrada no encontrado");
            return;
        }
        try {
            out= new FileOutputStream(argv[1]);
        } catch(FileNotFoundException e) {
            System.out.println("Fichero de salida no encontrado");
            return;
        }
        byte[] buffer= new byte[256];
        int n = 0;
        while ( (n = in.read(buffer)) > 0 ) {
            out.write(buffer, 0, n);
        }
        in.close();
        out.close();
    }
}
```

CopyFile1.java

```
import java.io.*;
class CopyFile2 {
    public static void main(String argv[]) {
        FileInputStream fin;
        FileOutputStream fout;
        if (argv.length<2) {
            System.out.println("java CopiaFichero <fichEntrada> <fichSalida>");
            System.exit(0);
        }
        try {
            fin= new FileInputStream(argv[0]);
        } catch(FileNotFoundException e) {
            System.out.println("Fichero de entrada no encontrado");
            return;
        }
        try {
            fout= new FileOutputStream(argv[1]);
        } catch(FileNotFoundException e) {
            System.out.println("Fichero de salida no encontrado");
            return;
        }
        try {
            do {
                i = fin.read();
                if (i!= -1) fout.write(i);
            } while (i!= -1);
        } catch (IOException e) {
            System.out.println("Error de E/S" + e);
            return;
        }
        fin.close();
        fout.close();
    }
}
```

CopyFile2.java

```
import java.io.*;
class File2String {
    public static void main(String argv[]) {
        String str;
        Reader r = new FileReader("ejemplo.txt");
        StringBuffer sb = new StringBuffer();
        char [] b =new char[1024];

        while((n=r.read(b))>0) {
            sb.append(b,0,n);
        }
        str = sb.toString();
    }
}
```

File2String.java

- La clase **File** no pertenece a la jerarquía de clases de entrada y salida. Sirve para manejar ficheros del sistema operativo y conocer sus propiedades: nombre, fecha, tamaño, subdirectorios, etc. pero no accede a su contenido.

```
import java.io.*;
import java.util.*;
/* Este ejemplo informa sobre el estado de un fichero */

public class FileStatus {
    public static void main(String[] argv) throws IOException {
        if (argv.length == 0) {
            System.err.println("Usage: FileStatus filename");
            System.exit(1);
        }
        for (int i = 0; i < argv.length; i++) {
            status(argv[i]);
        }
    }

    public static void status(String fileName) throws IOException {
        System.out.println("---" + fileName + "---");

        // Construye un objeto File para el archivo dado.
        File f = new File(fileName);

        // Comprueba si existe realmente
        if (!f.exists()) {
            System.out.println("file not found");
            System.out.println();
            return;
        }
        // Imprime el nombre completo
        System.out.println("Canonical name " + f.getCanonicalPath());
        .....
    }
}
```

FileStatus.java

.....

```
// Imprime el directorio padre si es posible
String p = f.getParent();
if (p != null) {
    System.out.println("Parent directory: " + p);
}
// Comprueba si el archivo es legible
if (f.canRead()) {
    System.out.println("File is readable.");
}
// Comprueba si el archivo se puede escribir
if (f.canWrite()) {
    System.out.println("File is writable.");
}
// Informa sobre la hora de la última modificación
Date d = new Date();
d.setTime(f.lastModified());
System.out.println("Last modified " + d);

// Comprueba si es un archivo, directorio u otro elemento. Si es un
// fichero imprime su tamaño.
if (f.isFile()) {
    // Report on the file's size
    System.out.println("File size is " + f.length() + " bytes.");
} else if (f.isDirectory()) {
    System.out.println("It's a directory");
} else {
    System.out.println("I dunno! Neither a file nor a directory!");
}

System.out.println();    // blank line between entries
}
}
```

FileStatus.java

```
import java.io.*;

class ListaSubDirectorio {
    public static void ls(File dir) throws IOException {
        if (dir.isDirectory()) {
            File s[] = dir.listFiles();
            for (int i=0; i < s.length; i++) {
                System.out.println(s[i].getCanonicalPath());
                ls(s[i]);
            }
        } else {
            System.out.println("No es un directorio");
        }
    }

    public static void main (String argv[]) throws IOException {
        if (argv.length<1) {
            ls( new File(".") );
        } else {
            ls( new File(argv[0]) );
        }
    }
}
```

ListaSubDirectorio.java

```
import java.io.*;
public class Create {
    public static void main(String[] argv) throws IOException {
        if (argv.length == 0) {
            System.err.println("Usage: Creat filename");
            System.exit(1);
        }
        // Construir un objeto File no afecta al disco, el método createNewFile() sí
        for (int i = 0; i < argv.length; i++) {
            new File(argv[i]).createNewFile();
        }
    }
}
```

Create.java

```
import java.io.*;
public class Delete {
    public static void main(String[] argv) {
        for (int i=0; i<argv.length; i++)
            delete(argv[i]);
    }
    public static void delete(String fileName) {
        try {
            File target = new File(fileName);
            if (!target.exists()) {
                System.err.println("File " + fileName + " not present to begin with!");
                return;
            }
            if (target.delete())
                System.err.println("** Deleted " + fileName + " **");
            else
                System.err.println("Failed to delete " + fileName);
        } catch (SecurityException e) {
            System.err.println("Unable to delete " + fileName + e.getMessage());
        }
    }
}
```

Delete.java

- Esta clase implementa las interfaces **`DataInput`** y **`DataOutput`** y puede ser usada tanto para leer como para escribir
- Al abrir un fichero es necesario indicar si será sólo para leerlo o también para escribirlo

```
new RandomAccessFile("ejemplo.txt", "r");    //Lectura  
new RandomAccessFile("ejemplo.txt", "rw");   //Lectura y escritura
```

- Después de que el fichero ha sido abierto se pueden usar los métodos **`read`** o **`write`** para leer o escribir.
- **`RandomAccessFile`** mantiene un puntero que indica la posición actual dentro del fichero. Cuando se abre el fichero el puntero se coloca al principio inicializándolo a 0. Las llamadas a **`read`** o **`write`** van modificando el puntero según el número de bytes leídos o escritos.
- Además de los métodos habituales en la E/S con ficheros que implícitamente mueven el puntero, **`RandomAccessFile`** posee tres métodos para manipular explícitamente el fichero:
 - **`int skipBytes(int)`**, mueve el puntero hacia adelante el número especificado de bytes
 - **`void seek(long)`**, posiciona el puntero en la posición inmediatamente anterior al byte especificado
 - **`long getFilePointer()`**, devuelve el byte en el que el puntero está posicionado

```
import java.io.*;
/* El programa crea un fichero con un entero (offset) en la primera posición y
una cadena de texto, después lee el offset y devuelve la cadena a partir de esa
posicion */

public class RandomRead {
    final static String FILENAME = "random.dat";
    protected RandomAccessFile fichero;

    public static void main(String[] argv) throws IOException {
        RandomRead r = new RandomRead(FILENAME);
        System.out.println("Offset: " + r.leerOffset());
        System.out.println("Mensaje: " + r.leerCadena() + ".");
        r.cerrar();
    }

    /** Constructor: escribe el fichero, crea el objeto RandomAccessFile */
    public RandomRead(String fname) throws IOException {
        fichero = new RandomAccessFile(fname, "rw");
        int offset = (int) (Math.random()* 40);
        fichero.writeByte(offset);
        fichero.writeBytes("Este es un fichero de prueba para la clase RandomRead");
    }

    /** Lee el campo offset del fichero, situado en la posición 0. */
    public int leerOffset() throws IOException {
        fichero.seek(0); // situa el cursor al principio del fichero
        return fichero.readByte(); // lee el offset
    }

    /** Lee el mensaje en el offset dado */
    public String leerCadena() throws IOException {
        fichero.seek(leerOffset()); // se posiciona en el byte offset
        return fichero.readLine(); // lee la cadena
    }

    public void cerrar() throws IOException{
        fichero.close();
    }
}
```

RandomRead.java

- Esta clase divide un flujo de entrada en partes delimitadas por conjuntos de caracteres. El flujo de entrada debe ser algún tipo de **Reader**
- Esta clase está diseñada principalmente para analizar entradas del tipo de los lenguajes de programación. No es un divisor de *tokens* de tipo general aunque puede usarse a tal propósito.
- La clase **StringTokenizer** es similar, pertenece al paquete **java.util** y permite dividir en *tokens* un **String**.
- El **StreamTokenizer** puede reconocer identificadores, números, cadenas de texto y comentarios.
- Cada byte leído de la entrada es considerado como un carácter entre '\u0000' y '\u00FF'. Su valor es usado para determinar cinco posibles atributos del carácter: espacio en blanco, carácter alfabético (**TT_WORD**), carácter numérico (**TT_NUMBER**), separador de cadena y carácter de comentario. Cada carácter posee 0 o más de estos atributos.
- Una instancia de **StreamTokenizer** define cuatro propiedades:
 - Si los terminadores de línea deben ser devueltos como *tokens* o simplemente como espacios en blanco que separan *tokens*.
 - Si los comentarios de estilo C deben ser reconocidos e ignorados.
 - Si los comentarios de estilo C++ deben ser reconocidos e ignorados.
 - Si los caracteres de identificadores se convierten a minúsculas.

- Una aplicación típica que use StreamTokenizer:
 - Crea un objeto **StreamTokenizer**
 - Configura sus propiedades
 - Ejecuta un bucle que vaya llamando al método **nextToken** en cada iteración hasta que su valor sea **TT_EOF**.
 - ✓ Si el *token* es del tipo **TT_WORD** su valor viene dado por el miembro **sval** del objeto **StreamTokenizer**
 - ✓ Si el *token* es del tipo **TT_NUMBER** su valor viene dado por el miembro **nval** del objeto **StreamTokenizer**

```
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StreamTokenizer;
```

AnalizaStreamToken.java

```
/** Muestra como analizar un fichero usando StringTokenizer */

public class AnalizaStreamToken {
    protected StreamTokenizer tf;

    public static void main(String[] av) throws IOException {
        if (av.length == 0)
            new AnalizaStreamToken (new InputStreamReader(System.in)).procesar();
        else
            for (int i=0; i<av.length; i++)
                new AnalizaStreamToken(av[i]).procesar();
    }

    /** Construye un StreamTokenizer a partir del nombre del fichero */
    public AnalizaStreamToken(String fileName) throws IOException {
        tf = new StreamTokenizer(new FileReader(fileName));
    }

    /** Construye un StreamTokenizer a partir del nombre de un Reader existente */
    public AnalizaStreamToken(Reader rdr) throws IOException {
        tf = new StreamTokenizer(rdr);
    }

    ....
}
```

```

....

protected void procesar() throws IOException {
    int i;
    tf.eolIsSignificant(true);
    while ((i = tf.nextToken()) != StreamTokenizer.TT_EOF) {
        switch(i) {
            case StreamTokenizer.TT_EOF:
                System.out.println("End of file");
                break;
            case StreamTokenizer.TT_EOL:
                System.out.println("End of line");
                break;
            case StreamTokenizer.TT_NUMBER:
                System.out.println("Number " + tf.nval);
                break;
            case StreamTokenizer.TT_WORD:
                System.out.println("Word, length " + tf.sval.length() +
                                   "->" + tf.sval);
                break;
            default:
                System.out.println("¿Qué es i = " + i + "?");
        }
    }
}

```

AnalizaStreamToken.java

- La *serialización* consiste en transformar un objeto en una secuencia de bytes de manera que pueda ser enviado por un canal de entrada o de salida. El proceso contrario se llama *deserialización*.
- El proceso de *serialización* de un objeto consiste en escribir recursivamente todos los valores de los atributos de ese objeto que no hayan sido definidos como **static** o **transient** en la clase.
 - NOTA: cuando un miembro de una clase se declara como **transient** su valor no necesita persistir cuando el objeto se ha almacenado.
- Esto es útil cuando se quiere guardar el estado del programa en una zona de almacenamiento persistente como un fichero.
- La serialización también es necesaria para implementar la invocación remota de métodos (RMI) que permite que un objeto Java que está en un equipo llame a un método de un objeto que está en otro equipo. Si por ejemplo se debe pasar un objeto como argumento, el equipo emisor lo serializa y el receptor lo deserializa.

- Los tipos simples y casi todos los objetos básicos proporcionados por la biblioteca estandar de Java son *serializables*.
- Para *serializar* un objeto la clase tiene que implementar el interfaz **Serializable** (aunque no hay que escribir ningún método especial)
- Si una clase es serializable sus subclases también los son.
- Cuando se intenta *serializar* un objeto que no es *serializable* se produce una excepción **NotSerializableException**
- Los métodos de serialización y deserialización por defecto vienen dados respectivamente por **writeObject** (de la clase **ObjectOutputStream**) y **readObject** (de la clase **ObjectInputStream**).
- El siguiente programa (**SerializationDemo.java**) muestra el uso de la serialización y deserialización de un objeto de la clase **MiClase**

```
import java.io.*;
public class MiClase implements Serializable{
    String s;
    int i;
    double d;
    public MiClase (String s, int i, double d){
        this.s = s;
        this.i = i;
        this.d = d;
    }
    public String toString() {
        return "s=" + s + "; i=" + i + "; d=" + d;
    }
}
```

```
import java.io.*;
public class SerializationDemo {
    public static void main(String args[]){
        // Serialización del objeto
        try {
            MiClase objeto1 = new MiClase("Hola", -3, 7.2e10);
            System.out.println("objeto1 = " + objeto1);
            FileOutputStream fos = new FileOutputStream("serial.bin");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(objeto1);
            oos.flush();
            oos.close();
        } catch (IOException e){
            System.out.println("Error durante la serialización: " + e);
            System.exit(0);
        }
        // Deserialización del objeto
        try {
            MiClase objeto2;
            FileInputStream fis = new FileInputStream("serial.bin");
            ObjectInputStream ois = new ObjectInputStream(fis);
            objeto2 = (MiClase) ois.readObject();
            ois.close();
            System.out.println("objeto2 = " + objeto2);
        } catch (Exception e){
            System.out.println("Error durante la deserialización: " + e);
            System.exit(0);
        }
    }
}
```

SerializationDemo.java

- Solo en caso de que se quiera modificar el comportamiento de la serialización hay que proporcionar una implementación de los métodos **writeObject** y **readObject** en la clase que se quiera modificar.
- Si un atributo concreto no debe ser escrito al serializar, (por ejemplo una contraseña), debe marcarse con la palabra reservada **transient**. Al recuperarlo su valor será 0 o, si se trata de un objeto, null.
- Ejemplo: (para escribir todas un objeto **MiClase** siempre con la cadena "Lab.Ing.SW") habría que añadir los siguientes métodos a **MiClase.java**

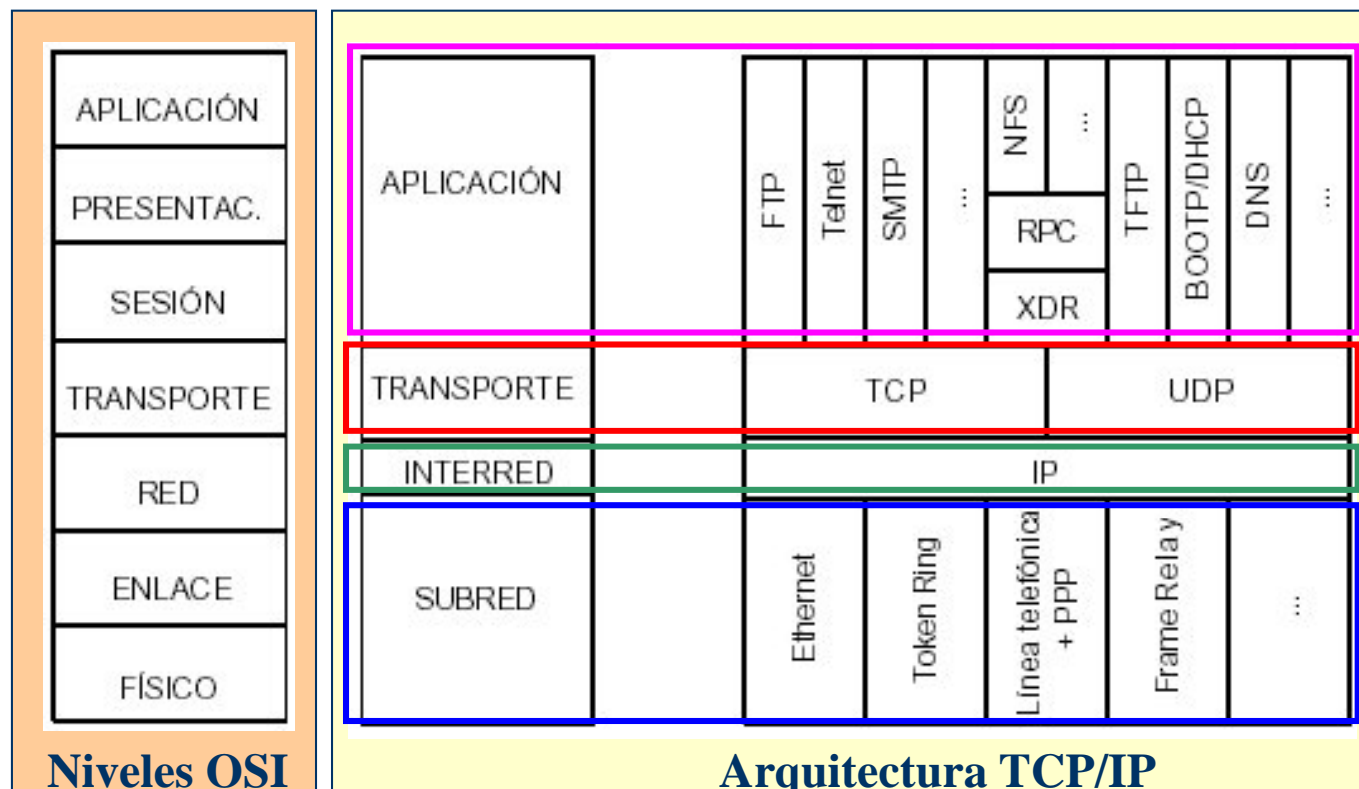
```
import java.io.*;
public class MiClase implements Serializable{
    String s;
    transient int i;
    double d;
    public MiClase (String s, int i, double d){
        this.s = s;
        this.i = i;
        this.d = d;
    }
    public String toString() {
        return "s=" + s + "; i=" + i + "; d=" + d;
    }
    private void writeObject (ObjectOutputStream out) throws IOException {
        out.writeObject(new String("Lab.Ing.SW"));
        out.writeDouble(d);
    }
    private void readObject (ObjectInputStream in) throws IOException,
                                                                    ClassNotFoundException {
        s = (String) in.readObject();
        d = in.readDouble();
    }
}
```

MiClase.java

- El entorno de ejecución asocia un número de versión a cada clase serializable llamado **serialVersionUID**
- Este número es usado durante la deserialización para verificar que tanto el emisor como el receptor de un objeto serializado han cargado las clases del objeto que son compatibles.
- Si el receptor carga una clase para el objeto con diferente **serialVersionUID** que la que cargó el emisor, al deserializar se lanzará la excepción **InvalidClassException**.
- Una clase serializable puede declarar explícitamente su propio número de serie estableciendo un atributo **static final long serialVersionUID**. Si no se declara, se le asignará uno por defecto.
- Se recomienda siempre asignar uno debido a que el cálculo del **serialVersionUID** por defecto es muy sensible a los detalles de la clase que puede variar dependiendo de las implementaciones del compilador. Por lo tanto para garantizar un número de serie consistente es mejor declararlo explícitamente.

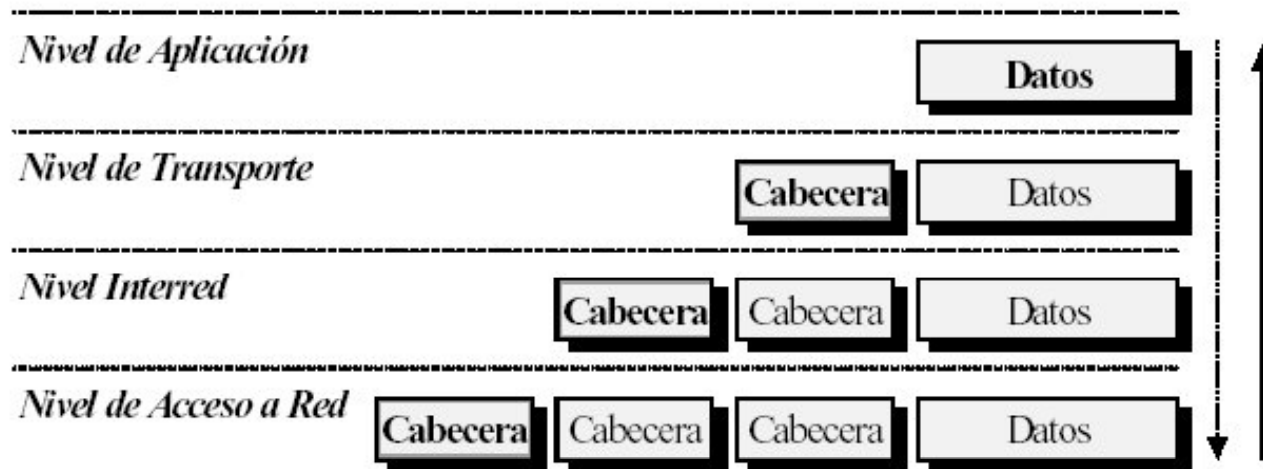
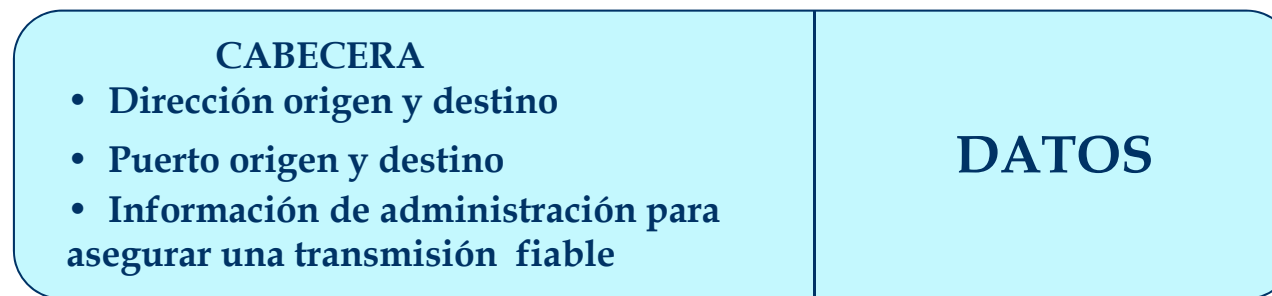
Comunicaciones

- INTERNET
 - Conjunto de computadoras que pueden comunicarse con las computadoras y redes registradas en InterNic (Centro de información de redes de Internet)
 - Correspondencia de los protocolos de Internet con los niveles OSI



- Los ordenadores conectados a Internet se comunican con los demás utilizando Transmission Control Protocol (**TCP**) o User Datagram Protocol (**UDP**).
- Los programas Java que se comunican sobre la red son programados en el nivel de aplicación de modo que el programador se desentiende de los niveles TCP/UDP usando las clases del paquete **java.net**.
- Estas clases proporcionan comunicación en red independientemente del sistema.

- ¿Cómo se transmiten los datos a través de Internet?
 - En paquetes de tamaño finito: datagramas
 - El datagrama es la unidad básica de transmisión en Internet



• UDP vs. TCP

TCP: Transport Control Protocol

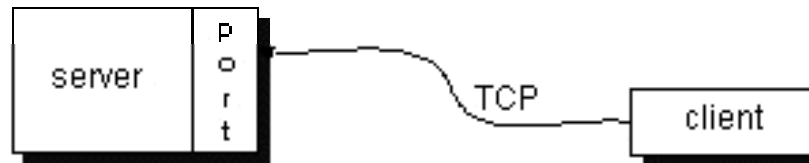
- **Definición:** es un protocolo basado en la **conexión** que proporciona un **flujo de datos fiable** entre dos computadoras.
- Envía datos a través de un **canal dedicado** de transmisión punto-a-punto
- **Símil:** es como una **comunicación telefónica**
- Establece una conexión entre dos máquinas que permanece abierta mientras dura la comunicación
- Es una comunicación fiable:
 - reenvía los datos si se pierde o estropea un paquete
 - ordena los paquetes que llegan desordenados
- **Aplicaciones:** FTP, HTTP,...

UDP: User Datagram Protocol

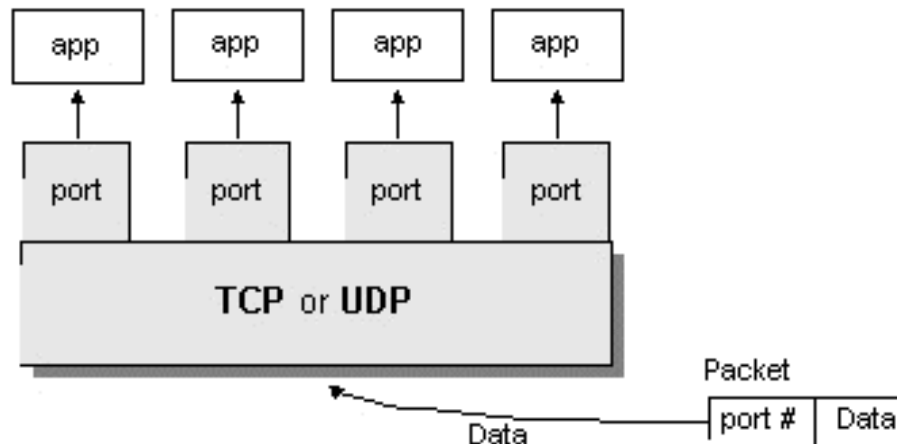
- **Definición:** es un protocolo que envía **paquetes de datos independientes** llamados **datagramas** de un ordenador a otro **sin garantizar su llegada**. No está basado en la **conexión** como TCP.
- **Símil:** es como una **comunicación postal**
- NO es una comunicación fiable:
 - no hay seguridad de entrega
 - los paquetes llegan desordenados
- No hay seguridad de entrega
- **Aplicaciones:** transmisiones de audio y vídeo en tiempo real, ...

- Cuando una máquina recibe un paquete determina si es un paquete TCP o un datagrama UDP inspeccionando la cabecera IP

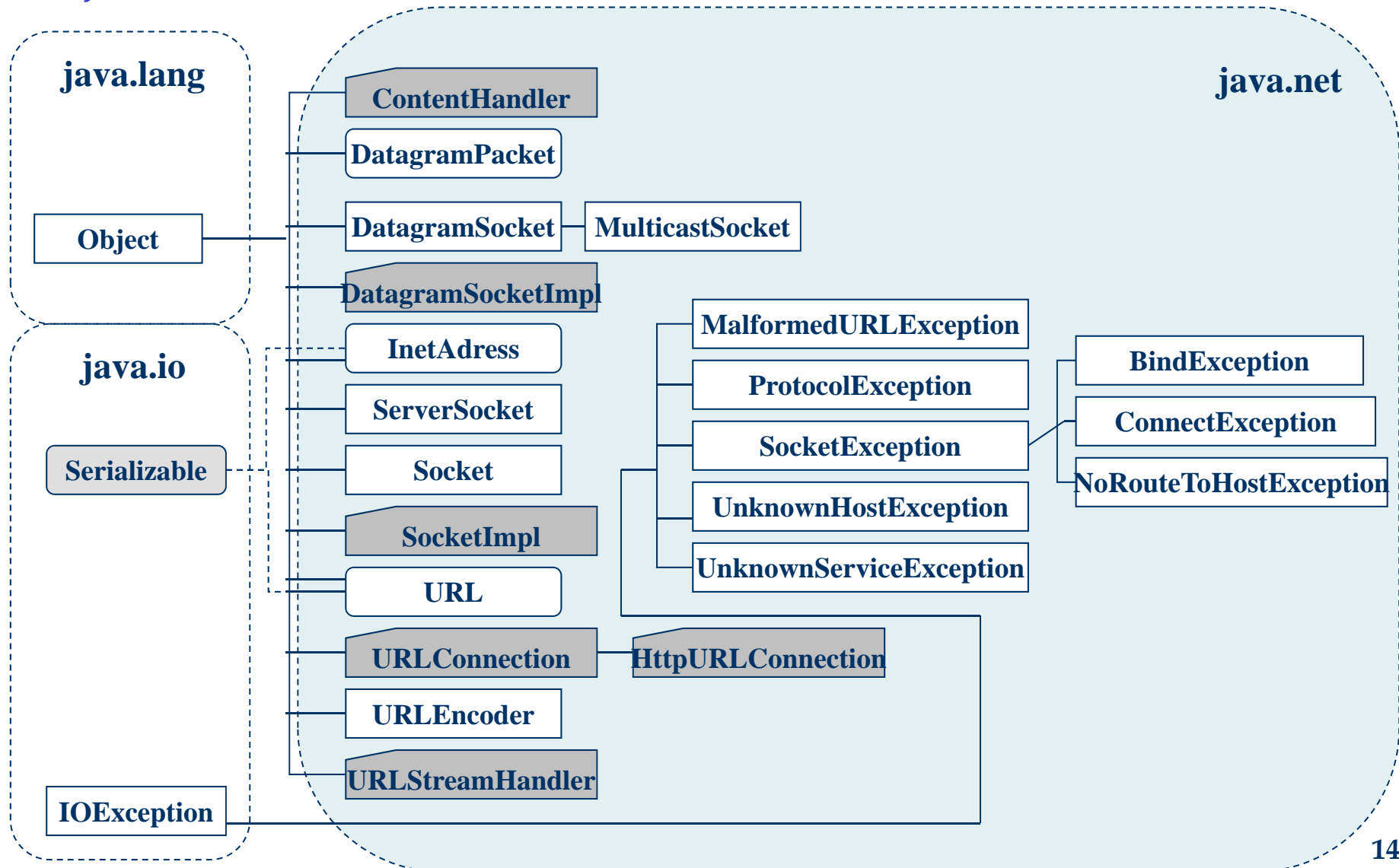
- Los protocolos TCP y UDP usan **puertos** para trazar el camino de unos datos de entrada hasta un proceso particular que se esté ejecutando en un ordenador.
- Los datos transmitidos por Internet van acompañados de información de direccionamiento que identifica el ordenador (la dirección IP localiza la máquina de la red a la que se han enviado los datos) y el puerto al que son destinados (identifica la aplicación dentro de ese ordenador que está esperando recibir los datos).
- TCP y UDP pueden compartir puertos
 - TCP: una aplicación servidora liga un socket a un número de puerto concreto de modo que todo lo que llegue a ese puerto le será destinado.



- UDP: los datagramas contienen el número de puerto de su destino y UDP lo encamina a la aplicación adecuada.



- Paquetes de Java relacionados con las comunicaciones: **java.net**, **java.rmi**



- Paquetes de Java relacionados con las comunicaciones:

- **java.net**

- ✓ Para aplicaciones sobre TCP usaremos URL, URLConnection, Socket y ServerSocket
 - ✓ Para aplicaciones sobre UDP usaremos DatagramPacket, DatagramSocket y MulticastSocket

- **java.rmi**

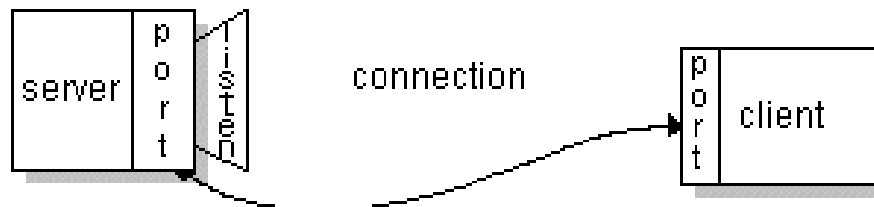
- ✓ Invocación remota de métodos, permite al usuario trabajar de forma transparente con objetos en máquinas remotas como si estuviesen en local

- En una **aplicación cliente-servidor** el servidor proporciona algún servicio que el cliente utiliza. La comunicación entre ambos debe ser fiable: ningún dato debe perderse y todos deben llegar en el mismo orden en el que se enviaron.
- **¿Qué es un Socket?**
 - TCP proporciona un canal de comunicación punto a punto fiable.
 - Para comunicarse sobre TCP tanto el cliente como el servidor establecen una conexión con el otro ligando un socket al final de esta conexión.
 - Por tanto, **un socket se define como un punto final del enlace de comunicación entre dos procesos que se están ejecutando en la red.**
 - Para comunicarse el cliente y el servidor leen y escriben en el socket asociado a la conexión.
 - Un socket se identifica por un número de puerto de modo que la capa de transporte TCP pueda localizar la aplicación destino de los datos
- **¿Cómo se comunican dos aplicaciones a través de un socket?**
 - Se realiza una **petición de comunicación** entre dos máquinas
 - Se **acepta la petición**
 - El cliente y el servidor se comunican **escribiendo y leyendo desde sus sockets**
 - Se **cierra la conexión.**

- ¿Cómo se comunican dos aplicaciones a través de un socket?
 - Se realiza una petición de comunicación entre dos máquinas
 - ✓ El cliente conoce el puerto de la máquina sobre la que se está ejecutando el servidor y el número de puerto asociado. El cliente solicita una “cita” con el servidor y le manda un “remite”, esto es, un número de puerto local donde puede ser localizado (normalmente asignado de forma transparente por el sistema).



- Se acepta la petición
 - ✓ Si todo va bien el servidor acepta la conexión. El servidor necesita un nuevo socket para poder continuar esperando peticiones en el puerto original mientras atiende al cliente conectado.



- Una vez establecida la conexión el cliente y el servidor se comunican escribiendo y leyendo de sus sockets. Para ello se utilizan clases del paquete java.io (streams) y se utilizan los mismos métodos que en la lectura/escritura de ficheros o de la entrada estándar.

- Clases de Java para trabajar con sockets

`java.net.ServerSocket`

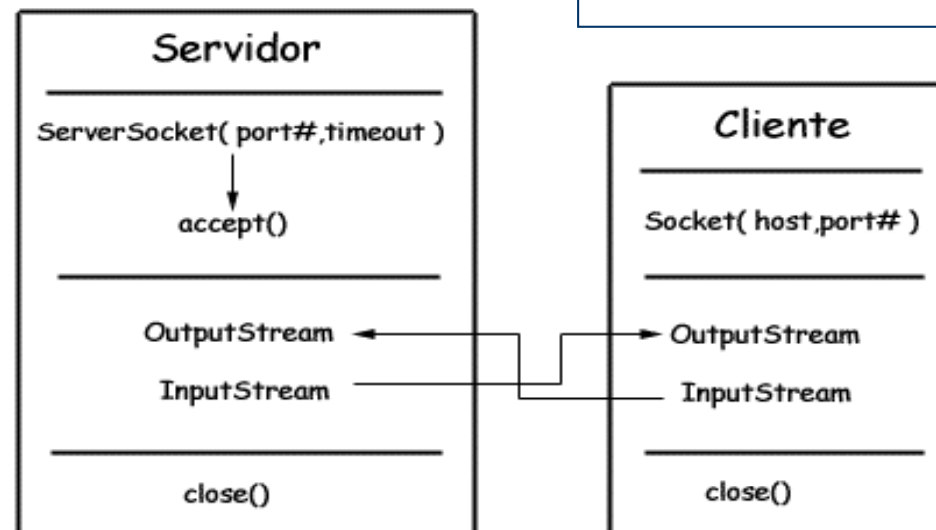
(sólo servidor)

- Reservar un puerto para un socket
- Esperar y aceptar peticiones de clientes en un puerto

`java.net.Socket`

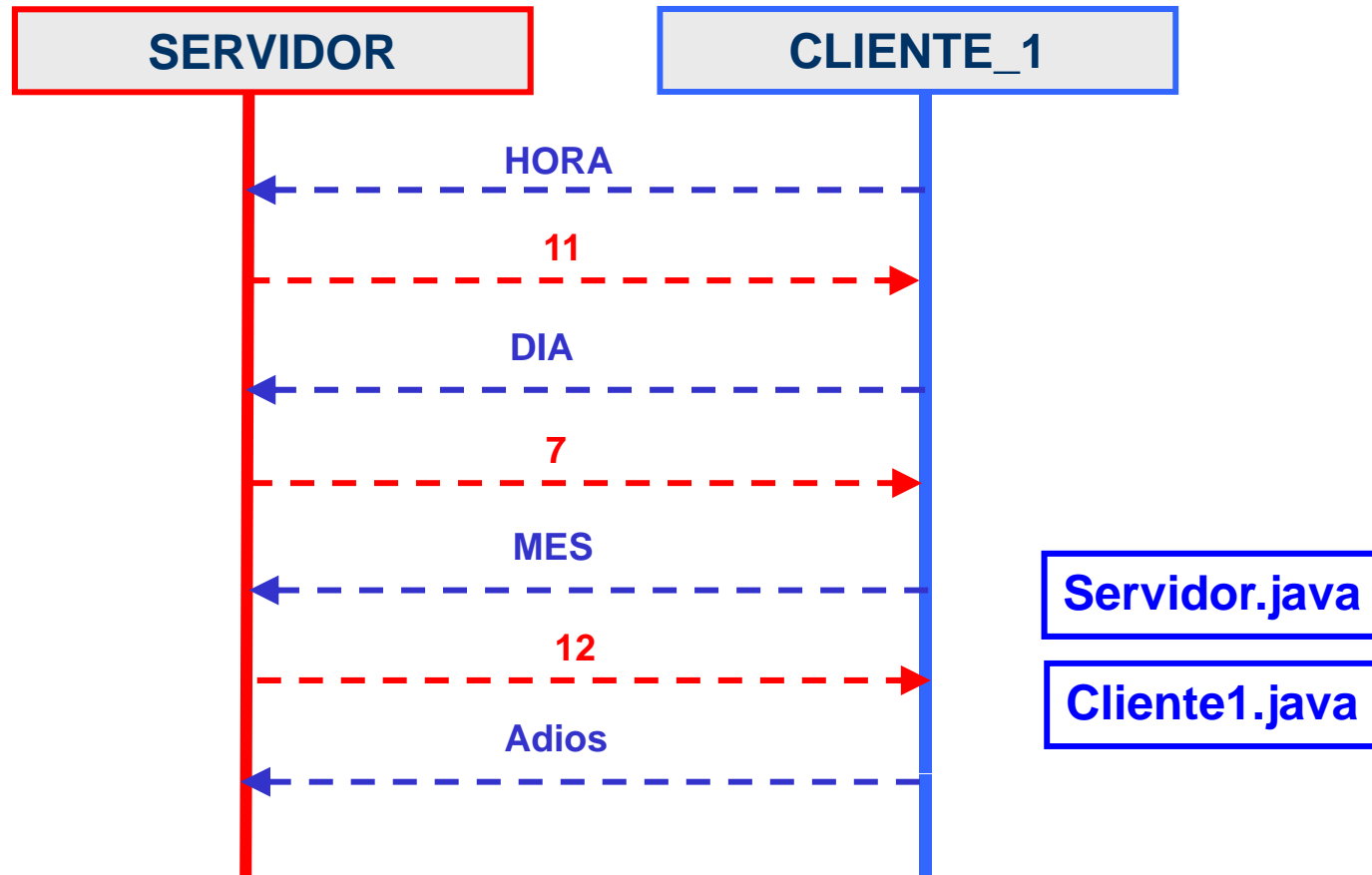
(servidor y cliente)

- Conectarse a una máquina remota
- Enviar datos
- Recibir datos
- Cerrar una conexión

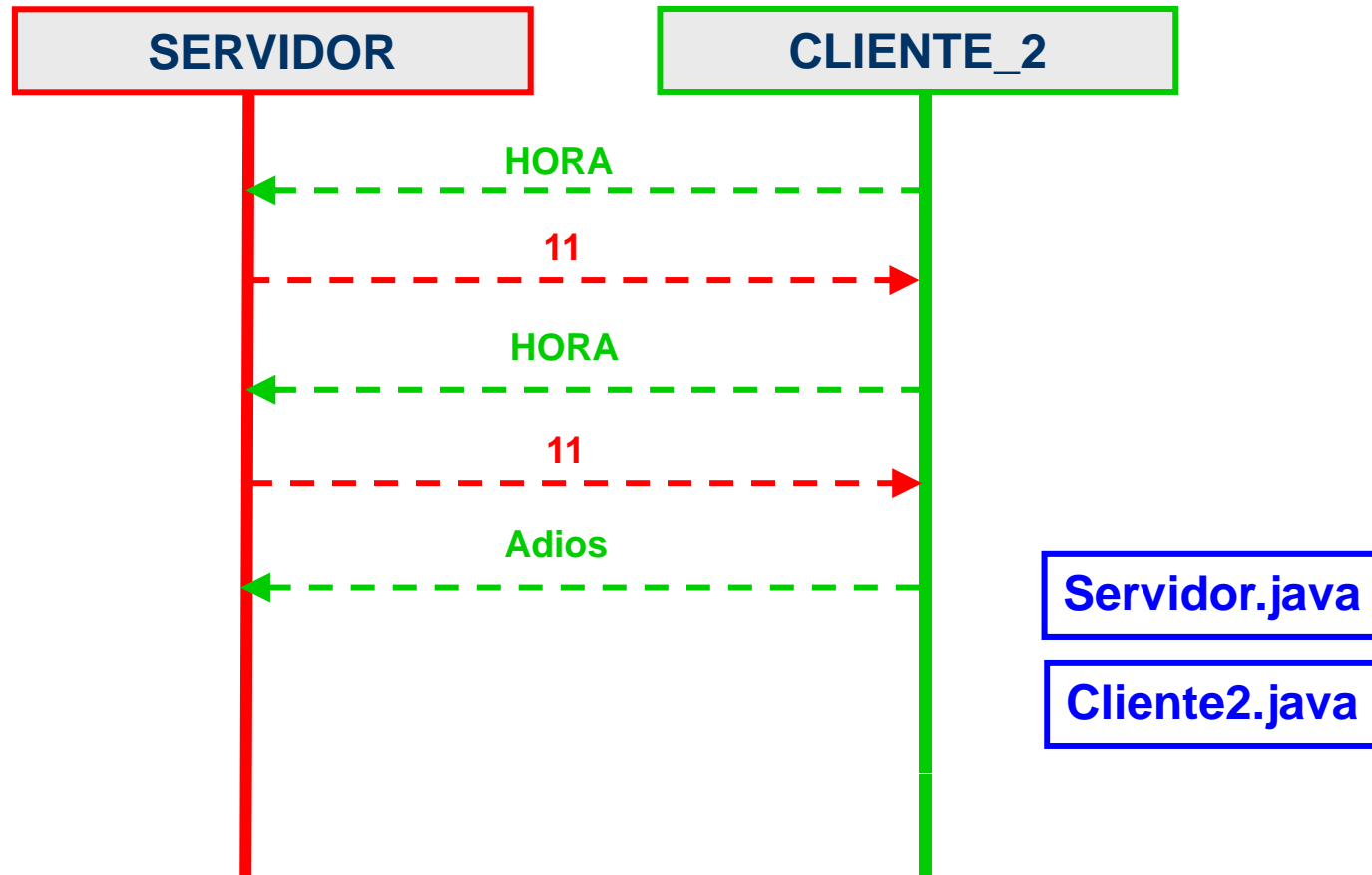


- ¿Cómo se escribe un SERVIDOR TCP en Java?
 - El servidor se está ejecutando y escuchando en un puerto
 - Espera la petición de conexión de un cliente.
 - Se abren canales (streams) para la entrada y salida de datos
 - Se leen y se escriben datos con el formato del protocolo del servidor
 - Se cierran los canales
 - Se cierra el socket
- ¿Cómo se escribe un CLIENTE TCP en Java?
 - Se realiza una petición de conexión y para ello se abre un socket a la máquina y puerto donde esté escuchando el servidor
 - Se abren canales (streams) para la entrada y salida de datos.
 - Se leen y se escriben datos con el formato del protocolo del servidor
 - Se cierran los canales
 - Se cierra el socket

- Ejemplo:
 - Una aplicación servidora proporciona el día, la hora o el mes a sus clientes.
 - Tenemos dos clientes que le envían distintas peticiones.
- Algunos ejemplos de “conversaciones” con distintos clientes:



- Ejemplo:
 - Una aplicación servidora proporciona el día, la hora o el mes a sus clientes.
 - Tenemos dos clientes que le envían distintas peticiones.
 - Algunos ejemplos de “conversaciones” con distintos clientes:



```
try {
    // Reservar un puerto de la máquina...
    ServerSocket servidor = new ServerSocket(4444);
    //...y detener la aplicación a la espera de alguna petición de algún
    cliente
    Socket cliente = servidor.accept();

    // Si esta ha sido aceptada (no ha saltado ninguna excepción) se
    // abren los canales para enviar y recibir datos
    InputStream in = cliente.getInputStream();
    OutputStream out = cliente.getOutputStream();

    // Recibir la petición del cliente
    Tipo var = in.read();
    // Componer la respuesta para el cliente y enviarla
    ...
    out.write(...);

    // Cuando se dé por finalizada la comunicación con el cliente cerrar
    // los canales y el socket
    out.close();
    in.close();
    cliente.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Servidor.java

```
try {
    // Establecer la comunicación con el servidor (IP y puerto)
    Socket servidor = new Socket("localhost",4444);

    // Si esta ha sido aceptada (no ha saltado ninguna excepción) se
    // abren los canales para enviar y recibir datos
    InputStream in = servidor.getInputStream();
    OutputStream out = servidor.getOutputStream();

    //Componer la petición del servicio y enviarla
    ....
    out.write(...);
    //Recibir la respuesta del servidor
    Tipo var = in.read();
    //Cuando ya no se necesite nada del servidor cerrar los canales y el
    //socket
    out.close();
    in.close();
    servidor.close();
} catch (UnknownHostException e) {
    System.out.println("No puedo encontrar el host");
} catch (IOException e) {
    System.out.println("Error en la conexión con el host");
}
```

Cliente1.java

- Protocolo de comunicación entre el cliente y el servidor del ejemplo:

```
....
do {
    // Recibir la petición del cliente
    petition = in.readLine();
    System.out.println("Petición recibida: " + petition);
    if (petition.equals("DIA")) {
        respuesta = "DIA: " + gc.get(Calendar.DAY_OF_WEEK);
    } else if (petition.equals("MES")) {
        respuesta = "MES: " + gc.get(Calendar.MONTH);
    } else if (petition.equals("HORA")) {
        respuesta = "HORA: " + gc.get(Calendar.HOUR);
    } else {
        respuesta = "No entiendo";
    } out.println(respuesta);
} while (!petition.equals("Adios"));
....
```

Servidor.java

- ¿Cómo se escribe un SERVIDOR TCP en Java que atienda simultáneamente a varios clientes?
 - Derivando la gestión de cada petición (cliente) a una hebra

```
try {
    ServerSocket servidor = new ServerSocket(4444);
    do {
        // Se acepta una petición de conexión
        Socket cliente = servidor .accept();
        // Se crea una hebra para manejar la petición
        (new HebraServidor(cliente)).start();
    } while (true);
} catch (IOException ex){
    System.exit(0);
}
```

ServidorConcurrente.java

```
class HebraServidor extends Thread {
    Socket cliente;
    public HebraServidor(Socket client) {
        this. cliente = client;
    }
    public void run() {
        try {
            //Abrir canales para comunicarse con el cliente
            //Leer y escribir
            //Cerrar canales
        } catch (IOException ex) {...}
        cliente.close();
    }
}
```

HebraServidor.java

- El **protocolo UDP** proporciona un modo de comunicación donde una aplicación envía paquetes de datos a otra llamados datagramas.
- **¿Qué es un datagrama?**
 - Es un mensaje independiente enviado a través de la red cuya:
 - ✓ llegada
 - ✓ hora de llegada
 - ✓ contenidono están garantizados
- Las clases **DatagramPacket** y **DatagramSocket** del paquete **java.net** implementan la comunicación usando UDP.

- ¿Cómo se comunican dos aplicaciones usando datagramas?
 - Los clientes y servidores no tienen un canal dedicado punto a punto.
 - Las aplicaciones que se comunican vía datagramas envían y reciben paquetes de información completamente independientes.
 - ✓ A cada paquete enviado o recibido se le pone la dirección y se encamina de forma individual. Paquetes enviados de una máquina a otra pueden ser encaminados de forma diferente.
 - ✓ Servidor
 - Espera en un puerto la recepción de un paquete con la petición de servicio
 - Lee de la cabecera del datagrama la dirección y puerto del cliente
 - Le envía una respuesta
 - ✓ Cliente
 - Construye un datagrama con un mensaje a la dirección y puerto del servidor
 - Asigna un puerto para enviar el datagrama e incluye esta información en la cabecera del datagrama
 - Envía el datagrama
 - Espera la respuesta del servidor
 - La llegada de datagramas así como su orden no están garantizados.

- Clases de Java para trabajar con datagramas

`java.net.DatagramPacket`

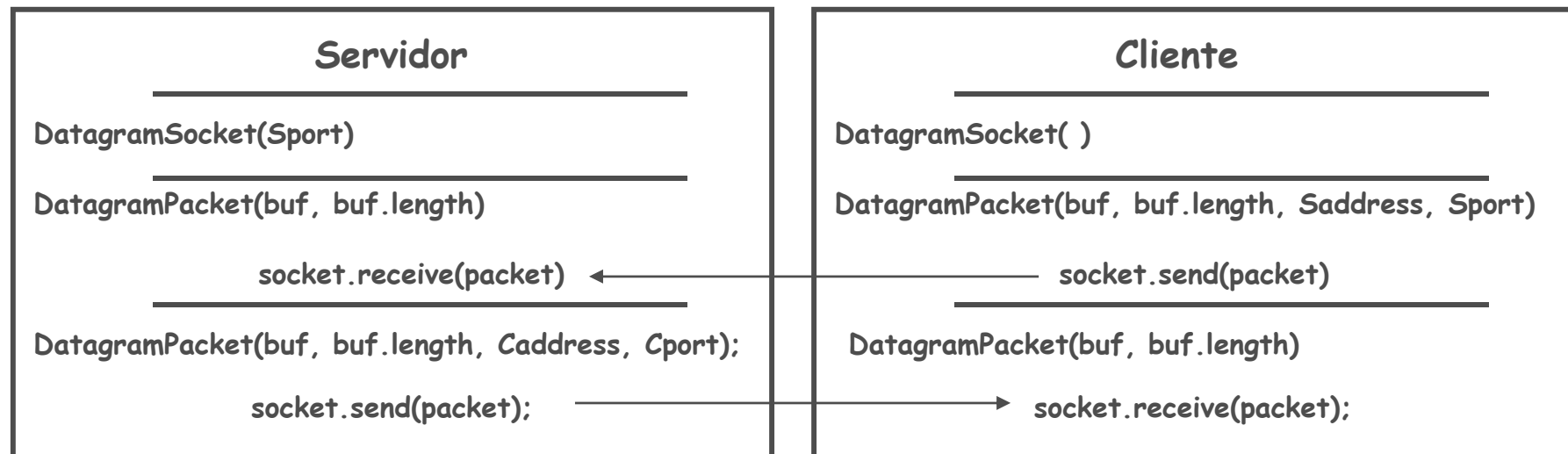
- Representa un datagrama

`java.net.DatagramSocket`

- Representa el punto de envío o recepción de paquetes

`java.net.MulticastSocket`

- Se usa para enviar y recibir paquetes IP de difusión múltiple



- ¿Cómo se escribe un SERVIDOR UDP en Java ?
 - Se crea un socket en un puerto específico que sirva como punto de envío y recepción de paquetes.
 - Se construye un datagrama para recibir la petición del cliente y se espera hasta que llegue.
 - Se construye la respuesta.
 - Se extraen la dirección y el puerto del cliente de la cabecera del datagrama recibido.
 - Se construye el datagrama con la respuesta y se envía
 - Se cierra el socket.
- ¿Cómo se escribe un CLIENTE UDP en Java ?
 - Se crea un socket que sirva como punto de envío y recepción de paquetes. El sistema asigna un puerto anónimo.
 - Se construye el datagrama con el mensaje y se envía.
 - Se construye un datagrama para recibir la respuesta.
 - Se espera la respuesta.
 - Se procesa la respuesta.
 - Se cierra el socket.

- Servidor UDP: escribir un programa ServidorHora.java que escuche en un socket UDP en el puerto 2345. ServidorHora puede recibir datagramas con dos tipos de peticiones. Si recibe "time" enviará como respuesta la fecha y hora actuales, si recibe "quit" el programa finaliza.
- Cliente UDP: escribir un programa PedirHora.java que envíe cinco datagramas seguidos con el texto time al puerto 2345. Una vez enviados todos los datagramas espera la respuesta de ServidorHora.java. Debe mostrar en consola los datagramas enviados y recibidos. Al final enviara un datagrama con el texto "quit".

```
// Se establece un punto de comunicación en un puerto determinado al que
// mandarán sus peticiones los clientes
DatagramSocket socket = new DatagramSocket(4444);

// Se crea un paquete o datagrama vacío para recibir las peticiones de los
// clientes
byte[] bufPetición = new byte[256];
DatagramPacket datagramaPetición = new DatagramPacket(bufPetición,
                                                         bufPetición.length);

// Se espera la petición.
socket.receive(datagramaPetición);
// Se lee y se analiza la petición
String petición = datagramaPetición.getData();
// Se construye la respuesta
String respuesta = "Esta es la respuesta";
byte[] bufRespuesta = new byte[256];
bufRespuesta = respuesta.getBytes();
// Se extraen la dirección y el puerto del cliente de la cabecera del datagrama
// que llegó con su petición
InetAddress address = datagramaPetición.getAddress();
int port = datagramaPetición.getPort();
// Se construye un datagrama con la respuesta y se envía
DatagramPacket datagramaRespuesta = new DatagramPacket(bufRespuesta,
                                                         bufRespuesta.length, address, port);

socket.send(datagramaRespuesta);
// Se cierra el socket
socket.close();
```

ServidorHora.java

```
// Se establece un punto de comunicación con el servidor para el
// envío y la recepción de paquetes. El sistema asigna un puerto anónimo
DatagramSocket socket = new DatagramSocket();

// Se crea un paquete o datagrama en el que se incluye, junto con la
// dirección IP y el puerto de la aplicación servidor, un buffer con la
// petición
InetAddress direccion = InetAddress.getByName( "apolo.lcc.uma.es");
byte[] bufPeticion = new byte[256];
String peticion = ...;
bufPeticion = peticion.getBytes();
DatagramPacket datagramaPeticion = new DatagramPacket(bufPeticion,
                                                    bufPeticion.length, direccion, 4444);

// Se envía la petición. El sistema incluye en la cabecera la
// dirección IP del cliente y el puerto anónimo para que el servidor
// sepa a donde tiene que enviar la respuesta
socket.send(datagramaPeticion);

// Se construye un datagrama vacío para recibir la respuesta
byte[] bufRespuesta = new byte[256];
DatagramPacket datagramaRespuesta = new DatagramPacket(bufRespuesta,
                                                    bufRespuesta.length);

// Se procesa la respuesta
String respuesta = new String(datagramaRespuesta.getData());

// Si no se necesita nada más del servidor se cierra el socket
socket.close();
```

PedirHora.java

- La clase **MulticastSocket** incluida en el paquete **java.net** se usa en el lado del cliente para recibir paquetes que el servidor transmite a múltiples clientes simultáneamente.
- Ejemplo: una aplicación en la que el cliente envía un datagrama pidiendo una cita y el servidor le devuelve la frase célebre del día.

```
import java.io.*; public class MulticastServer {  
    public static void main(String[] args) throws IOException {  
        new MulticastServerThread().start();  
    }  
}
```

MulticastServer.java

```

public void run() {
    while (moreQuotes)
        try {
            byte[] buf new
            // don't wait for
            String dString
            if (in == null) d
            else dString = g
            buf = dString.getBytes();
            InetAddress group = InetAddress.getByName( "230.0.0.1");
            DatagramPacket packet;
            packet = new DatagramPacket(buf, buf.length, group, 4446);
            socket.send(packet);
            try {
                SECO
            }
        }
        socket.close(),
    }
}

```

En el ejemplo anterior el servidor sacaba la dirección y el puerto donde debía enviar la respuesta del cliente, ahora, tanto la dirección como el puerto están incluidos en el código

"230.0.0.1" es un identificador de grupo (en vez de la dirección de Internet de una máquina) y ha sido arbitrariamente seleccionada de las reservadas para ese propósito(*).

Este paquete es para todos los clientes que están escuchando en el puerto 4446 y que son miembros del grupo 230.0.0.1

El cliente debe tener un MulticastSocket ligado a este puerto.

MulticastServerThread.java

(*) Un grupo multicast se especifica por una dirección IP de la clase D (224.0.0.0 – 239.255.255.255]

El cliente debe tener un MulticastSocket ligado al este puerto.

```
MulticastSocket socket = new MulticastSocket(4446);
InetAddress group = InetAddress.getByName("230.0.0.1");
socket.joinGroup(group);
DatagramPacket packet;
for (int i = 0; i < 5; i++) {
    byte[] buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length, group, 4446);
    socket.receive(packet);
    String received = new String(packet.getData());
    System.out.println("Quote of the Moment: " + received);
}
socket.leaveGroup(group);
socket.close();
```

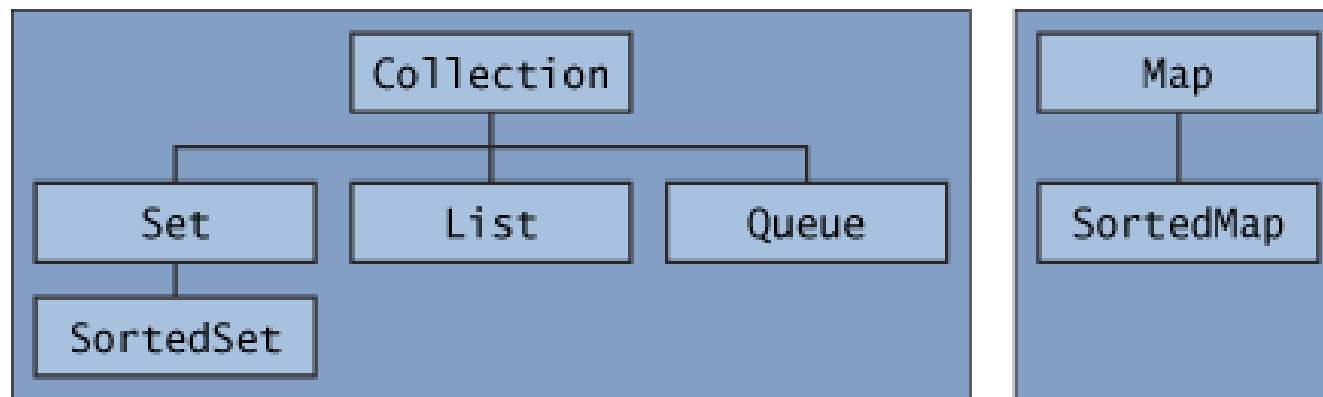
Para ser un miembro del grupo 230.0.0.1 el cliente debe unirse utilizando el método . A partir de este momento el cliente está listo para recibir los paquetes enviados al grupo al puerto 4446.

MulticastClient.java

Colecciones

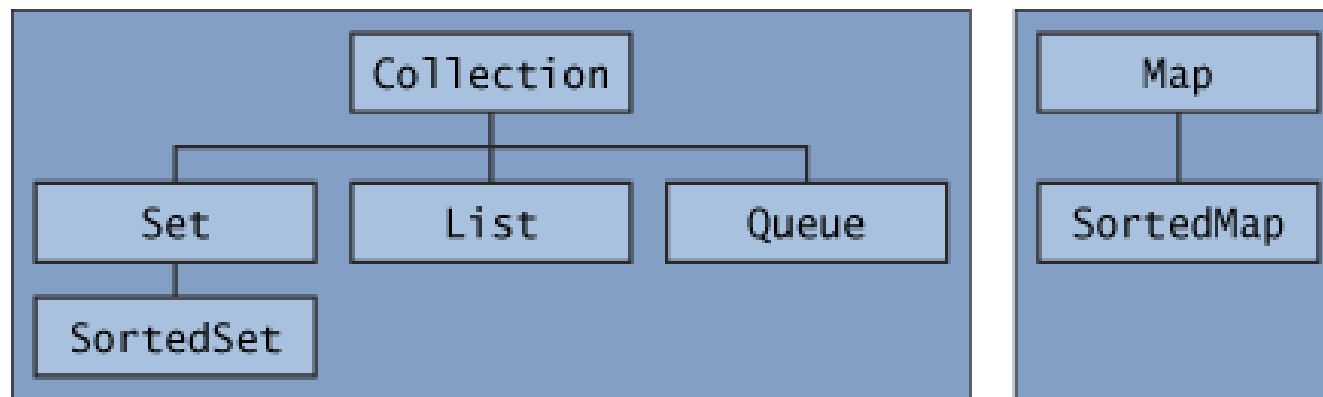
- Una colección es un objeto que agrupa múltiples elementos en uno sólo. Las colecciones se usan para almacenar y manipular datos de forma conjunta.
- El **marco de colecciones de Java** incluye:
 - **Interfaces**: son tipos abstractos de datos que representan colecciones. Java define una jerarquía de interfaces que permiten manejar las colecciones independientemente de los detalles de implementación.
 - **Implementaciones**: son implementaciones concretas de interfaces de colecciones.
 - **Algoritmos**: son métodos que desarrollan problemas de computación útiles, tales como búsqueda y ordenación y que se aplican sobre objetos que implementan interfaces de colecciones. Los algoritmos son polimórficos, es decir, el mismo método puede ser usado sobre distintas implementaciones de una interfaz.
- Tanto las interfaces como sus implementaciones son genéricas

- Java define dos jerarquías de interfaces de colecciones en el paquete `java.util`:



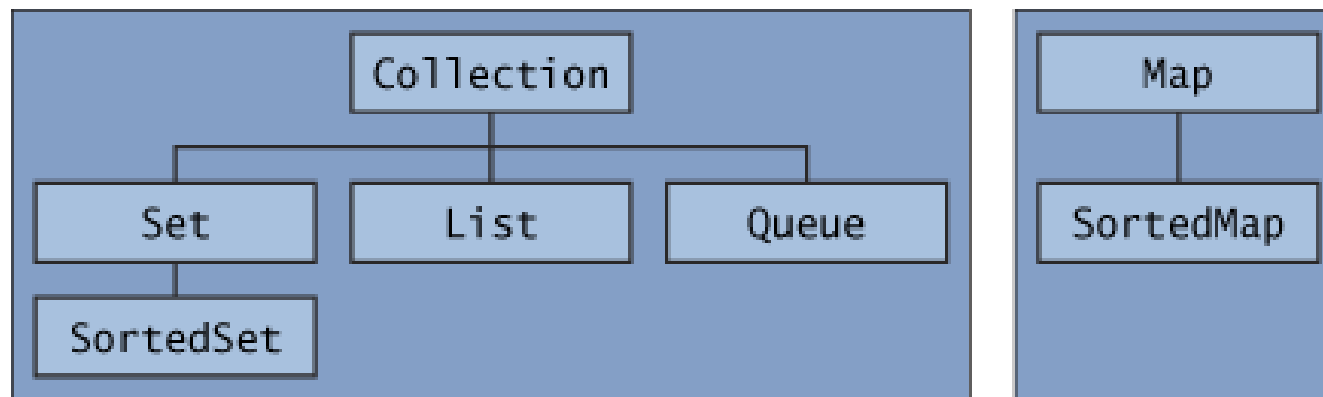
- **Collection**: estructuras formadas por objetos simples.
- **Map**: estructuras en las que se almacenan objetos por parejas. El primero es una clave para encontrar el segundo (valor).

- Java define dos jerarquías de interfaces de colecciones en el paquete `java.util`:



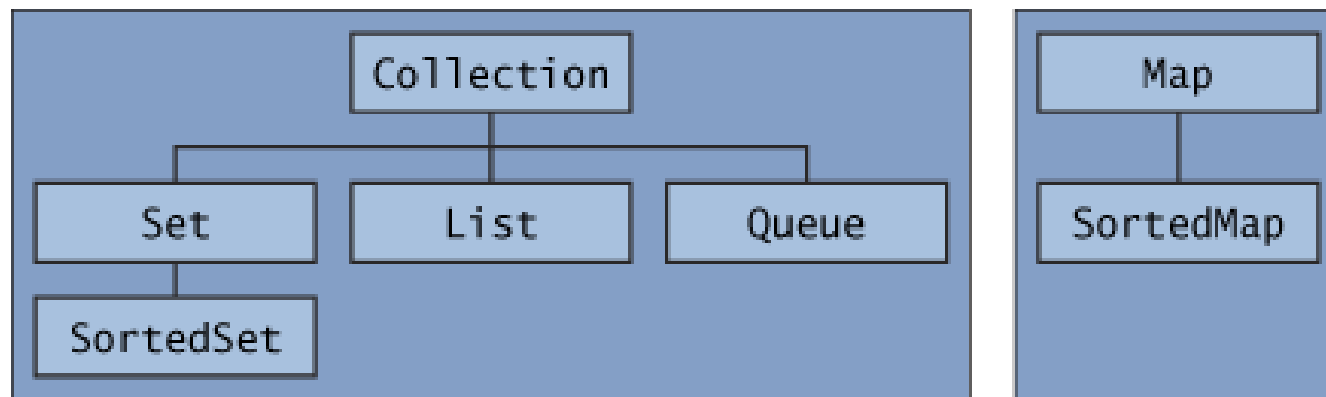
- Set**: es una colección que no puede contener elementos duplicados. Esta interfaz modela la abstracción matemática de un conjunto.
- SortedSet**: es un conjunto que mantiene sus elementos ordenados en orden ascendente. Proporciona operaciones adicionales para gestionar el orden del conjunto.

- Java define dos jerarquías de interfaces de colecciones en el paquete `java.util`:



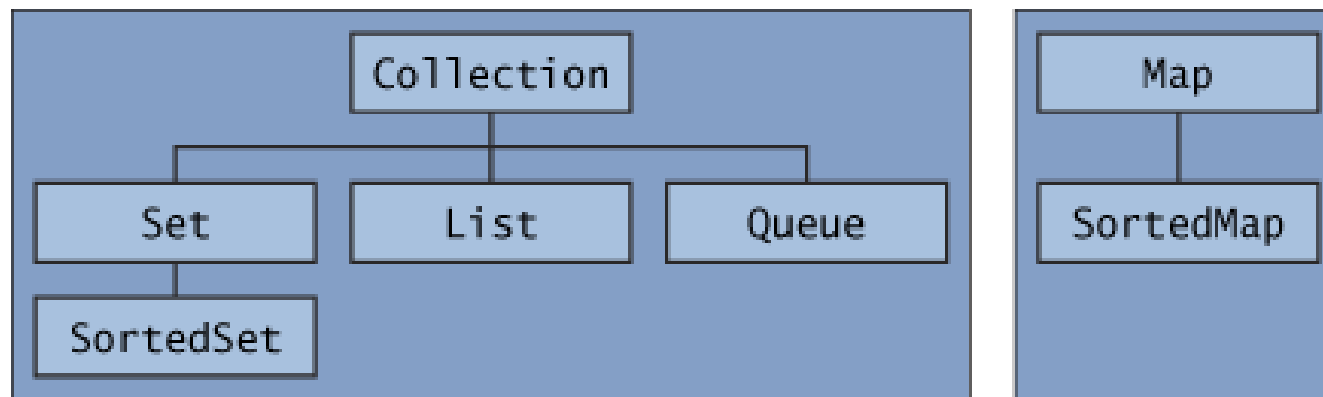
- List**: es una colección ordenada (secuencia). Las listas pueden contener elementos duplicados. el usuario de una lista normalmente tiene control sobre el lugar donde insertar cada elemento y puede acceder a ellos por su posición.

- Java define dos jerarquías de interfaces de colecciones en el paquete `java.util`:



- Queue:** sirve para manejar tipos especiales de listas en las que se eliminan elementos sólo de la parte superior. Las colas más comunes, aunque no todas, siguen un orden FIFO. En una cola FIFO los elementos se insertan al final. En otros tipos de colas existen otras formas de insertar elementos. Entre las excepciones están las colas de prioridad que ordenan los elementos según un comparador que se le proporciona al objeto durante su creación.

- Java define dos jerarquías de interfaces de colecciones en el paquete `java.util`:



- Map**: estructuras en las que se almacenan objetos por parejas. El primero es una clave para encontrar el segundo (valor).
- SortedMap**: un `Map` que mantiene sus correspondencias ordenadas en orden ascendente.

- El paquete **java.util** también define una serie de **implementaciones**, en la siguiente tabla se muestran algunas de las más habituales:

Interfaces	Implementaciones					
	Tabla hash	Heap (montón)	Array de tamaño variable	Árbol	Lista enlazada	Tabla hash + lista enlazada
Set	HashSet			TreeSet		LinkedHashSet
List			ArrayList		LinkedList	
Queue		PriorityQueue				
Map	HashMap			TreeMap		LinkedHashMap

- La interfaz Collection representa un grupo de objetos a los que llamamos elementos de la colección.
- Se usa para manejar colecciones de objetos con la máxima generalidad posible.

```
public interface Collection<E> extends Iterable<E> {  
    // Operaciones básicas  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator<E> iterator();  
    boolean equals(Object o)  
    int hashCode()  
    // Operaciones sobre todos los elementos de la colección  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
    // Operaciones de arrays  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

- El método **add**:
 - Garantiza que la colección contendrá el elemento especificado después de que se haya completado la llamada.
 - Devuelve true si la colección ha cambiado después de su ejecución.
 - Se utiliza tanto si la colección admite elementos duplicados como si no.
- El método **remove**:
 - Se usa para eliminar de la colección una única instancia del elemento especificado
 - Devuelve true si la colección ha cambiado después de su ejecución.

```
boolean add(E element);  
boolean remove(Object element);
```

- **Constructor de conversión**: todas las implementaciones de **Collection** tienen un constructor que toma un objeto **Collection** como argumento, que inicializa la nueva colección con los elementos de la segunda (independientemente de su tipo). Es decir me permite **convertir una colección de un tipo a otro**.

- Existen dos formas de **recorrer una colección**:
 - Con un bucle **for-each**
 - ✓ El siguiente código muestra por consola cada elemento de una colección.

```
for (Object o : collection)
    System.out.println(o);
```

- Con un **Iterator**: es un objeto que permite recorrer una colección y eliminar elementos selectivamente si se desea. Para **obtener un objeto Iterator a partir de una colección es necesario llamar al método iterator()**. Es mejor usar un Iterator que un for-each cuando:
 - ✓ es necesario eliminar el elemento actual
 - ✓ cuando se van a recorrer múltiples colecciones en paralelo
 - ✓ El siguiente método muestra como filtrar una colección eliminando determinados elementos. Este método es polimórfico, esto es, sirve sea cual sea la colección que estemos tratando.

```
static void filtrar(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

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

    List <String> lista = new ArrayList<String>();

    lista.add("D");
    lista.add("A");
    lista.add("A");
    lista.add("B");
    System.out.print("(for-each) Lista: ");
    for (String letra: lista){
        System.out.print(letra);
    }
    System.out.println();
    System.out.print("(iterador) Lista: ");
    for (Iterator<String> it = lista.iterator(); it.hasNext();){
        System.out.print(it.next());
    }
}
```

RecorridoColeccion.java

Resultado:
(for-each) Lista: DAAB
(iterador) Lista: DAAB

- Un **Set** es una **Collection** que no puede contener elementos duplicados.
- Modela la abstracción matemática de conjunto.
- La interfaz **Set** contiene sólo los métodos heredados de **Collection** y añade la restricción de que los elementos duplicados están prohibidos.
- Las instancias de **Set** pueden ser comparadas usando los métodos **equals** y **hashCode** incluso si sus implementaciones de tipo difieren.
 - **equals** devuelve true si el objeto especificado es también un conjunto, los dos conjuntos tienen el mismo tamaño y cada elemento de un conjunto está incluido en el otro.
 - **s1.equals(s2)** implica que **s1.hashCode()==s2.hashCode()** para cualquiera dos conjuntos **s1** y **s2**.

ACLARACIÓN: **hashCode()** es un método de **Object** que devuelve un código hash para un objeto, teniendo en cuenta las siguientes restricciones:

- Si se invoca sobre el mismo objeto más de una vez durante una ejecución de una aplicación el método **hashCode** devuelve siempre el mismo entero.
- Si dos objetos son iguales según el método **equal(Object)**, la llamada a **hashCode** sobre cada uno de los objetos produce el mismo resultado.

- Suma todos los elementos de un conjunto:

```
public class SumaConjunto {
    public static void main (String[] args) {
        int suma = 0;
        boolean hecho;
        TreeSet<Integer> conjunto = new TreeSet<Integer> ();
        hecho = conjunto.add(0); System.out.print("0 "+hecho);
        hecho = conjunto.add(2); System.out.print(", 2 "+hecho);
        hecho = conjunto.add(3); System.out.print(", 3 "+hecho);
        hecho = conjunto.add(3); System.out.print(", 3 "+hecho);
        hecho = conjunto.add(1); System.out.println(", 1 "+hecho);
        System.out.println("Conjunto=" + conjunto);
        for(Integer numero: conjunto){
            suma +=numero;
        }
        System.out.println("Suma = " + suma);
    }
}
```

SumaConjunto.java

add devuelve
false si el
conjunto no se
modifica

El método
toString está
sobrescrito y
muestra todos
los valores de
una colección

Resultado:
0 true, 2 true, 3 true, 3 false, 1 true
Conjunto=[0, 1, 2, 3]
Suma = 6

- Un objeto de la clase **SortedSet** es un **Set** que mantiene ordenados sus elementos en orden ascendente
- El orden se mantiene:
 - siguiendo el orden natural o,
 - según el orden especificado por un objeto de la clase **Comparator** que se le pasa al **SortedSet** cuando se crea
- **SortedSet** define, además de las operaciones heredadas en la interfaz **Set**, los siguientes métodos:

```
public interface SortedSet<E> extends Set<E> {
    // Operaciones que devuelven un subconjunto
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Extremos
    E first();
    E last();

    // Acceso al comparador
    Comparator<? super E> comparator();
}
```

- Suma todos los elementos de un conjunto:

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

        TreeSet<Integer> conjunto = new TreeSet<Integer> ();

        for (int i=0;i<30;i++)
            conjunto.add(i);

        System.out.println("Cola=" + conjunto.tailSet(20));
        System.out.println("Cabeza=" + conjunto.headSet(20));
        System.out.println("Subconjunto=" + conjunto.subSet(10,20));
    }
}
```

Subconjuntos.java

Resultado:

Cola=[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]

Cabeza=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Subconjunto=[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

- La librería **java.util** contiene varias **implementaciones de Set** y **SortedSet**:
 - Implementaciones de propósito general
 - ✓ **HashSet**: implementa un conjunto mediante una tabla *hash* (dispersión).
 - ✓ **LinkedHashSet**: implementa un conjunto mediante tablas hash y listas doblemente enlazadas.
 - ✓ **TreeSet**: implementa las interfaces Set y SortedSet.
 - Implementaciones de propósito específico
 - ✓ **EnumSet**: es una implementación para usar con tipos enumerados
 - ✓ **CopyOnWriteArraySet**: se implementa mediante una array. Las operaciones de modificación del array (añadir, eliminar, etc.) utilizan una nueva copia del array.

- **HashSet**

- Implementa un conjunto mediante una **tabla hash** (dispersión).
- No garantiza el orden de iteración que puede no permanecer constante a lo largo del tiempo.
- Esta clase permite el elemento *null*.
- Esta implementación **no es sincronizada**. Si múltiples hebras acceden a un conjunto concurrentemente y al menos una de las hebras lo modifica este debe ser sincronizado externamente. Un HashSet puede ser sincronizado durante su creación.

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

- El iterador devuelto por el método **iterator** de esta clase lanza una excepción (**ConcurrentModificationException**) si alguien modifica el conjunto después de creado el iterador (excepto a través del método `remove`).

- El siguiente programa lee de la entrada estándar un conjunto de palabras y construye un **HashSet** con las palabras que se repiten y otro con las que no.

```
public static void main(String args[]){
    Set<String> unicas = new HashSet<String>();
    Set<String> repetidas = new HashSet<String>();

    String palabras[] = {"yo", "ví", "yo", "vine", "yo", "vencí"};

    for (String p: palabras){
        if (!unicas.add(p))
            repetidas.add(p);
    }
    unicas.removeAll(repetidas);

    System.out.println("Únicas: " + unicas);
    System.out.println("Repetidas: " + repetidas);
}
```

Repetidas.java

Resultado:

Únicas: [vencí, vine, ví]

Repetidas: [yo]

- El siguiente programa construye dos conjuntos y calcula un tercero y un cuarto que se corresponden respectivamente con la unión y la intersección de los dos primeros

```
HashSet<String> coloresFrancia = new HashSet<String>();
coloresFrancia.add("Rojo");
coloresFrancia.add("Blanco");
coloresFrancia.add("Azul");
System.out.println("[FRANCIA]+"coloresFrancia);

HashSet<String> coloresItalia = new HashSet<String>();
coloresItalia.add("Rojo");
coloresItalia.add("Blanco");
coloresItalia.add("Verde");
System.out.println("[ITALIA]+"coloresItalia);

HashSet<String> unionColores = new HashSet<String>();
unionColores.addAll(coloresFrancia);
unionColores.addAll(coloresItalia);
System.out.println("[UNION]+"unionColores);

HashSet<String> interseccionColores = new HashSet<String>();
interseccionColores.addAll(coloresFrancia);
interseccionColores.retainAll(coloresItalia);
System.out.println("[INTERSECCION]+"interseccionColores);
```

UnionInterseccion.java

- Un **HashSet** NO garantiza el orden de iteración a lo largo del tiempo

Resultado (puede variar):

[FRANCIA]	[Azul, Blanco, Rojo]
[ITALIA]	[Blanco, Rojo, Verde]
[UNION]	[Azul, Blanco, Rojo, Verde]
[INTERSECCION]	[Blanco, Rojo]

- **LinkedHashSet**

- Implementa un conjunto mediante tablas hash y listas doblemente enlazadas.
- Esta implementación se diferencia de **HashSet** en que mantiene una lista doblemente enlazada que define el orden de iteración (orden de inserción).
- Al igual que **HashSet** no es sincronizada.
- Su iterador se comporta del mismo modo que el de **HashSet**, aunque **LinkedHashSet** **SÍ** garantiza el orden de iteración.

Si en el ejemplo UnionInteseccion.java sustituimos HashSet por LinkedHashSet el resultado siempre sería:

[FRANCIA]	[Azul, Blanco, Rojo]
[ITALIA]	[Rojo, Blanco, Verde]
[UNION]	[Azul, Blanco, Rojo, Verde]
[INTERSECCION]	[Blanco, Rojo]

- **TreeSet**

- Es una implementación de la interfaz **SortedSet** mediante árboles binarios.
- Los objetos se almacenan ordenados en orden ascendente de sus elementos o según el orden establecido por un comparador que se le proporcionará durante la creación del objeto, dependiendo del constructor que se use.
- Al igual que **HashSet** no es sincronizada.
- Su iterador se comporta del mismo modo que el de **HashSet**

```
class TreeSetDemo {
    public static void main(String argv[]) {
        TreeSet<String> letras = new TreeSet<String>();
        letras.add("A");
        letras.add("D");
        letras.add("B");
        letras.add("C");
        System.out.println(letras);
    }
}
```

TreeSetDemo.java

Resultado:

[A,B,C,D]

- **Comparable<T>**

- Interfaz genérica que pertenece a **java.lang**
- Las clases que implementan **Comparable** pueden ordenarse, es decir pueden compararse de forma coherente
- **T** representa el tipo de objetos que se compara
- Esta interfaz define el método **int compareTo(T obj)** que se usa para determinar lo que Java llama el *orden natural de las instancias de una clase*
 - ✓ compara el objeto que realiza la llamada con **obj** y devuelve 0 si son iguales y un valor < 0 si el primer objeto es menor que el segundo.
- Las clases **Byte**, **Character**, **Double**, **Float**, **Long**, **Short**, **String** e **Integer** implementan **Comparable** y definen un método **compareTo**

- **Comparator<T>**

- Interfaz genérica que pertenece a **java.util**
- Si una determinada clase no implementa **Comparable** o su ordenación natural no es la adecuada para algún propósito se puede proporcionar en ocasiones un objeto sustituto **Comparator**
- Esta interfaz define el método **int compare (T obj1, T obj2)** que proporciona una ordenación para **obj1** y **obj2**

```
class Carta implements Comparable<Carta> {
    int numero;
    int palo;
    public final static short AS = 1;
    ...
    public final static short PICAS = 1;
    ...
    public int compareTo(Carta c) {
        int mayor = 0;
        if (numero > c.numero) {
            mayor = 1;
        } else if (numero == c.numero) {
            if (palo > c.palo) {
                mayor = 1;
            } else if (palo == c.palo) {
                mayor = 0; // cartas iguales
            } else {
                mayor = -1;
            }
        } else {
            mayor = -1;
        }
        return mayor;
    }
}
```

Carta.java

El método para
comparar dos cartas
forma parte del objeto
Carta.java

El criterio de
ordenación es
1º por el
número y 2º
por el palo

...

- El método `add` utiliza el método `compareTo` para ordenar el conjunto, comparando el elemento que se va a añadir con cada uno de los que ya hay insertados y localizando su posición correcta

```
public static void main(String argv[]) {
    TreeSet<Carta> cartas = new TreeSet<Carta>();

    cartas.add(new Carta(Carta.TRES, Carta.TREBOLES));
    cartas.add(new Carta(Carta.Q, Carta.CORAZONES));
    cartas.add(new Carta(Carta.J, Carta.TREBOLES));
    cartas.add(new Carta(Carta.Q, Carta.PICAS));
    cartas.add(new Carta(Carta.Q, Carta.PICAS));
    cartas.add(new Carta(Carta.AS, Carta.CORAZONES));
    cartas.add(new Carta(Carta.J, Carta.TREBOLES));
    for(Iterator i=cartas.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}
```

CartasDemo.java

El resultado sería:

```
[AS,CORAZONES]
[TRES,TREBOLES]
[J,TREBOLES]
[Q,PICAS]
[Q,CORAZONES]
```

```
class ComparadorCartas implements Comparator<Carta> {
    public static int compare(Carta c1, Carta c2) {
        int mayor = 0;
        if (c1.palo > c2.palo) {
            mayor = 1;
        } else if (c1.palo == c2.palo) {
            if (c1.numero > c2.numero) {
                mayor = 1;
            }
            else if (c1.numero == c2.numero) {
                mayor = 0; // cartas iguales
            }
            else {
                mayor = -1;
            }
        }
        else {
            mayor = -1;
        }
        return mayor;
    }
}
```

El criterio de ordenación es al contrario que en el ejemplo anterior: 1º por el palo y 2º por el número

ComparadorCarta.java

```
class CartasDemo2 {
    public static void main(String argv[]) {
        TreeSet<Carta> cartas =
            new TreeSet<Carta>(new ComparadorCartas());

        cartas.add(new Carta("Q", "PICAS"));
        cartas.add(new Carta("A", "CORAZONES"));
        cartas.add(new Carta("J", "TREBOLES"));
        cartas.add(new Carta("10", "DIABLOS"));
        cartas.add(new Carta("K", "PICAS"));
        cartas.add(new Carta("A", "CORAZONES"));

        for(Iterator i = cartas.iterator(); i.hasNext(); ) {
            Carta c = (Carta) i.next();
            System.out.println(c);
        }
    }
}
```

El método para comparar dos cartas es externo a la clase Carta (viene definido en la clase ComparadorCartas) de esta forma podemos crear distintos conjuntos de Cartas cada uno con un criterio de ordenación diferente

CartasDemo2.java

El resultado sería:

```
[Q,PICAS]
[A,CORAZONES]
[J,TREBOLES]
[10,DIABLOS]
[K,PICAS]
```

- La interfaz **List** define el comportamiento de una secuencia ordenada de elementos.
- Las listas pueden contener elementos duplicados.
- Además de las operaciones heredadas de **Collection**, la interfaz **List** incluye las siguientes operaciones:

```
public interface List<E> extends Collection<E> {
    // Operaciones para acceder a un elemento por su posición
    E get(int index);
    E set(int index, E element);
    boolean add(E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index, Collection<? extends E> c); //opcional

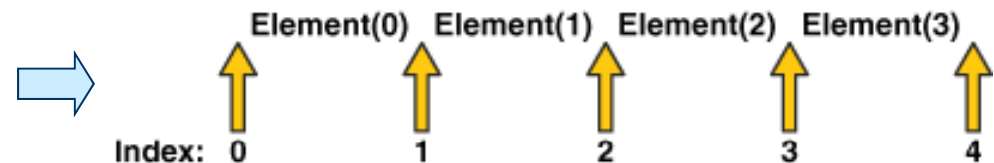
    // Operaciones de búsqueda
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteración, extiende Iterator para aprovechar la ventaja de
    // la secuencia natural de la lista
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Rango
    List<E> subList(int from, int to);
}
```

- **List** proporciona un iterador llamado **ListIterator** que permite:
 - recorrer la lista en cualquier dirección
 - modificar la lista durante la iteración
 - obtener la posición actual del iterador
- **List** puede proporcionar un **ListIterator** que inicialmente tenga el cursor bien en la primera posición, bien en la posición que se le indique.
- El cursor estará siempre entre dos elementos de modo que en una lista de longitud n hay $n+1$ valores válidos para el cursor (de 0 a n inclusive)

```
public interface ListIterator<E> extends
    Iterator<E>
{
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(E e);
    void add(E e);
}
```



- Las llamadas a **next** y **previous** se pueden intercalar, pero hay que tener cuidado porque la primera llama a **previous** devuelve el mismo elemento que la última llamada a **next**.
- Para recorrer una lista de atrás hacia adelante:

```
for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {
    Type t = it.previous();
    ...
}
```

- La interfaz `Iterator` proporciona la operación `remove()` para eliminar el último elemento de la colección devuelto por el método `next()`.
- Para `ListIterator`, esta operación elimina el último elemento devuelto por `next()` o `previous()`.
- El método `add` inserta un nuevo elemento en la lista inmediatamente antes de la posición del cursor.

*/*Este método genérico sustituye todas las ocurrencias de un valor de la lista por una secuencia de valores contenidos en otra lista que se le pasa como argumento*/*

```
public static <E> void reemplazar(List<E> list, E val,
                                List<? extends E> newVals) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); ){
        if (val.equals(it.next())) {
            it.remove();
            for (E e : newVals)
                it.add(e);
        }
    }
}
```

ReemplazarDemo.java

- El método **set** sobrescribe el último elemento devuelto por next o previous.

```
//Creamos una lista con los números pares entre 0 y 30
List<Integer> lista = new ArrayList<Integer>();
for (int i = 0; i < 30; i += 2) {
    lista.add(i);
}
System.out.println("PARES: " + lista);
//Sumamos uno a cada elemento de la lista y los convertimos en impares
for (ListIterator<Integer> it = lista.listIterator(); it.hasNext();) {
    it.set(it.next()+1);
}
System.out.println("IMPARES: " + lista);
```

ParesToImpares.java

El resultado sería:

PARES: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

IMPARES: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

LinkedList	ArrayList
Está implementada usando secuencias de referencias doblemente enlazadas	Está implementada usando matrices
El tiempo de acceso depende de la posición del elemento.	El tiempo de acceso es constante (independiente de la posición).
Es muy eficiente para insertar o borrar elementos en cualquier posición pero menos eficiente para buscar elementos intermedios.	Es menos eficiente para insertar y eliminar elementos de la lista
Tiene métodos para acceder, insertar y eliminar el primer y último elemento de la lista. También implementa la interfaz Queue	Se puede configurar su capacidad inicial que es el número de elementos que puede soportar antes de tener que crecer

- Sirve para manejar tipos especiales de listas en las que se **eliminan elementos** sólo de la **parte superior**.
- Las colas más comunes, aunque no todas, siguen un orden FIFO (por orden de llegada). En una cola FIFO los elementos se insertan al final.
- Existen tipos de colas en las que el orden se basa en otros criterios. Entre las excepciones están las colas de prioridad que ordenan los elementos según un comparador que se le proporciona al objeto durante su creación.
- Cada método de **Queue** tiene dos versiones según la forma en responden a un fallo de la operación: una que lanza una excepción y otra que devuelve un valor especial (**null** o **false**).

	Lanza una excepción	Devuelve un valor especial
Inserta	add(e)	offer(e)
Elimina	remove()	poll()
Examina	element()	peek()

- El siguiente ejemplo usa una cola para hacer una cuenta atrás a partir de un número que se da como argumento

```
public class CuentaAtras {

    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);

        // LinkedList implementa Queue
        Queue<Integer> queue = new LinkedList<Integer>();

        //Los elementos se añaden a la cola por el final (desde time hasta 0)
        for (int i = time; i >= 0; i--) {
            queue.add(i);
        }
        //Los elementos se sacan por el principio
        while (!queue.isEmpty()) {
            System.out.print(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

CuentaAtras.java

- **LinkedList**

- Implementa una cola FIFO

- **PriorityQueue**

- Es una cola basada en prioridades implementada por una estructura *heap* (montón)
- Ordena los elementos según el orden natural de los elementos o por el indicado por un objeto **Comparator** que se le proporciona durante su creación.
- Las operaciones de recuperación (**poll**, **remove**, **peek** y **element**) acceden al elemento de la cima de la cola que es el último elemento según el orden especificado
- El orden del iterador proporcionado es indefinido.

- La interfaz **Map** almacena objetos asociados a una clave.
- Las claves han de ser únicas pero los valores pueden estar duplicados.

```
public interface Map<K,V> {

    // Operaciones básicas
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Operaciones que afectan a todos los elementos
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Vistas de colecciones
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interfaz para los elementos de entrySet
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

- Es un **Map** que mantiene sus entradas ordenadas en orden ascendente según las claves.
- El orden puede ser el natural del tipo de elemento o el implementado en un objeto **Comparator** que se le proporciona durante su creación.
- Permiten una manipulación muy eficiente de los supmapas.
- Además de las operaciones propias de **Map**, **SortedMap** define las siguientes:

```
public interface SortedMap<K, V> extends Map<K, V>{

    //Acceso al comparador
    Comparator<? super K> comparator();

    // Proporcionan secciones del mapa
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);

    //Acceso a los extremos
    K firstKey();
    K lastKey();
}
```

- La librería **java.util** contiene varias implementaciones de **Map**:
 - Implementaciones de propósito general
 - ✓ **HashMap**: implementa un conjunto mediante una tabla *hash* (dispersión).
 - ✓ **LinkedHashMap**: implementa un map mediante tablas hash y listas doblemente enlazadas. Permite las iteraciones en orden de inserción.
 - ✓ **TreeMap**: implementa las interfaces **Map** y **SortedMap**.
 - Implementaciones de propósito específico
 - ✓ **EnumMap**: es una implementación para usar mapas con claves enumeradas
 - ✓ **WeakHashMap**: usa una tabla *hash* con claves débiles. Permite que la memoria de un elemento de un mapa se recicle cuando su clave no se use.

```
HashMap <String, Double> cuentas = new HashMap<String, Double>();

cuentas.put("Elena García", new Double(3434.34));
cuentas.put("Pepe Reyes", new Double(900.00));
cuentas.put("Ana Cruz", new Double(2557.23));
cuentas.put("Bernardo López", new Double(1356.78));

//Se obtiene un conjunto con las entradas para mostrarlas por //
//pantalla
Set<Map.Entry<String, Double> > set = cuentas.entrySet();
for (Map.Entry<String, Double> me : set){
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}

// Se depositan 1000 euros en la cuenta de Elena García
double saldo = cuentas.get("Elena García");
cuentas.put("Elena García", saldo + 1000);
System.out.println("Nuevo saldo de Elena García: " +
    cuentas.get("Elena García"));
```

HashMapDemo.java

El resultado sería

Ana Cruz: 2557.23
Elena García: 3434.34
Bernardo López: 1356.78
Pepe Reyes: 900.0
Nuevo saldo de Elena García:4434.34

- Al usar las clases de tipos abstractos de datos, hay que tener muy en cuenta la diferencia entre la estructura y los objetos. Cuando se crean copias de las estructuras, no se duplican los objetos. Si se quieren copiar los objetos hay que copiarlos explícitamente.
- Elegir el tipo de datos más eficiente según las operaciones más frecuentes que se vayan a emplear. Las clases más potentes pueden tener un coste en eficiencia. A veces este coste es despreciable.

- En la clase Collections existe un conjunto de métodos estáticos y genéricos que realizan operaciones de uso común con las colecciones de datos:
 - Ordenación: **sort**
 - Barajar: **shuffle**
 - Rutinas para la manipulación de datos:
 - ✓ **reverse**: invierte el orden de los elementos de una lista.
 - ✓ **fill**: sobrescribe cada elemento de la lista por uno dado.
 - ✓ **copy** : copia los elementos de una lista destino en una origen sobrescribiendo sus elementos.
 - ✓ **swap**: intercambia dos elementos de una lista.
 - ✓ **addAll**: añade los elementos especificados (uno a uno o como un array) a una colección.
 - Búsqueda
 - Composición
 - ✓ **frequency**: cuenta el número de veces que un elemento aparece en una lista
 - ✓ **disjoint**: determina si dos colecciones son disjuntas (no tienen ningún elemento en común)
 - Localización de valores extremos
 - ✓ **mínimo**
 - ✓ **máximo**

Algoritmos.java

Concurrencia

- Programación secuencial

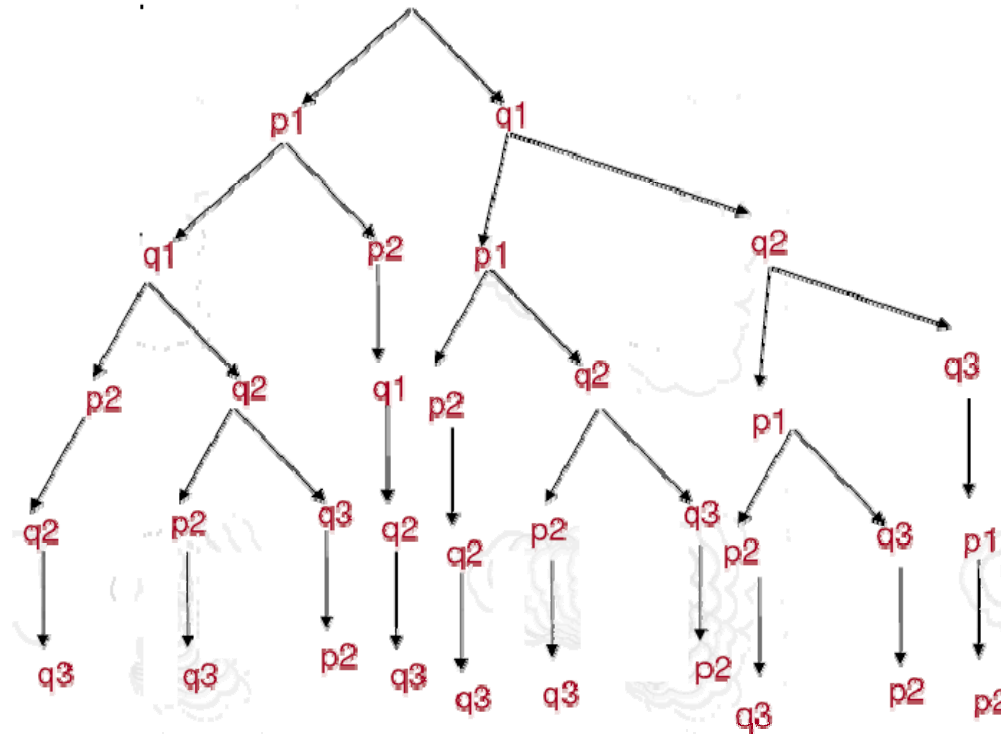
- Supongamos $P = p_1; p_2; \dots; p_n$ un programa secuencial donde p_i son instrucciones de modo que siempre se cumple que p_1 precede a p_2 , p_2 a p_3 , hasta concluir la ejecución de P .
- El programa es determinista. Las instrucciones están totalmente ordenadas y la siguiente instrucción a ejecutar siempre es conocida.
- Este comportamiento se mantiene incluso si el código de P contiene instrucciones de selección o bucles.
- Sin embargo no todas las instrucciones de un programa tienen por qué ejecutarse de forma secuencial, las tareas son independientes y podrían ejecutarse en otro orden. Por ejemplo:

```
int x, y, z;  
x=0;  
y=0;  
z=0;
```

- Podría ejecutarse $x=0; y=0; z=0$ o también $x=0; z=0; y=0$; etc.

- **Programación concurrente**

- ¿Y si tenemos dos programas $P=p1;p2$ y $Q=q1;q2;q3$?
- Una ejecución correcta se obtiene intercalando instrucciones de P y Q:
 - ✓ El orden relativo de las instrucciones de cada programa debe mantenerse, es decir p1 siempre irá delante de p2, q1 delante de q2 y q2 delante de q3.
 - ✓ No hay orden establecido entre las instrucciones de P y las de Q.



- Cada camino del árbol representa una posible ejecución de P y Q.

- **Multiprogramación:** técnica para la ejecución simultánea de dos o más programas en una misma computadora.
 - **Proceso**
 - ✓ instancia de un programa en ejecución
 - ✓ los procesos son tareas “pesadas” que poseen su propio espacio de direccionamiento
 - ✓ el cambio de contexto es una operación costosa
 - **Hebra** (o hilo)
 - ✓ se definen como “procesos ligeros”
 - ✓ una hebra existe como parte de un proceso
 - ✓ una hebra comparte con su proceso algunos recursos como el espacio de direccionamiento y los ficheros abiertos

- Un **programa secuencial** tiene una única hebra de control.
- Un **programa concurrente** tiene múltiples hebras de control que permiten ejecutar múltiples tareas en paralelo.
- En los sistemas monoprocesadores la **conurrencia** siempre se implementa **entrelazando las instrucciones de los procesos o hebras**.
 - Se aprovechan los ciclos del procesador mientras que está realizando operaciones de entrada/salida.
 - Los procesos a veces necesitan comunicarse y sincronizar sus acciones para ello pueden compartir memoria o utilizar algún mecanismo de paso de mensajes.
- Ejecutar un programa concurrente es más complejo que uno secuencial:
 - Hay que representar internamente cada **objeto hebra**
 - Hay que decidir en cada momento a qué hebra le toca ejecutarse: **planificación**

- El **modelo de ejecución multihilo** (multithreaded) es una característica fundamental de Java. Cada aplicación tiene al menos una hebra llamada hebra principal (main):
 - es la hebra desde la que se crea el resto de hebras del programa
 - cuando la hebra principal finaliza, el programa termina.
- La especificación de la máquina virtual de Java (JVM) no define una forma particular de planificar las hebras. Sólo indica que las hebras deberían planificarse utilizando una política basada en prioridades.
- Sin embargo, algunas implementaciones de la máquina virtual (java runtime) podrían no seguir esta recomendación por lo que el orden de ejecución de las hebras no está garantizado y depende de la plataforma de ejecución.

- Sun proporciona **dos implementaciones de las hebras**:
 - **Green threads**: la máquina virtual gestiona la planificación y otras tareas relacionadas con las hebras por sí misma sin necesidad del sistema operativo subyacente. Estas implementaciones son ineficientes, entre otras razones porque no son capaces de distribuir las hebras entre varias CPU por lo que no se pueden aprovechar de las ventajas que ofrece un sistema multiprocesador. Por otro lado las hebras que crea no son hebras al nivel del SO por lo que la JVM reparte su tiempo entre las distintas hebras que ha generado internamente.
 - **Native threads**: son creadas y planificadas por el sistema operativo subyacente. En vez de crear y planificar las hebras ella misma, la máquina virtual las crea llamando a los servicios del SO. Estas hebras pueden beneficiarse de múltiples procesadores y son más eficientes.

- En Java cada hebra está asociada con una instancia de la clase **Thread**.
- Una aplicación que crea una instancia de **Thread** debe proporcionar el código que se ejecutará en esa hebra y que confirman el método `run()`.
- El objeto Thread no es la hebra: encapsula la información sobre la hebra.
- Podemos definir una hebra creando una subclase de Thread



- Otra forma de crear una hebra es crear un objeto que implemente la interfaz Runnable, implementar su método run() y, una vez creado, pasárselo como parámetro al constructor de la hebra.

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread hebra = new Thread(new HelloRunnable());
        hebra.start();
    }
}
```

Código que ejecuta
la hebra

Comienzo de ejecución
de la hebra

Creación del objeto hebra
pasándole al constructor un
objeto de una clase que
implemente Runnable

HelloRunnable.java

- El uso de **Runnable** para la creación de hebras es más general porque no limita la herencia de la clase creada a **Thread**. En el ejemplo anterior, **HelloRunnable** puede ser subclase de cualquier clase, mientras que **HelloThread** no, porque ya lo es de **Thread**.
- En cualquier caso...
 - para arrancar la hebra se ejecuta el método **start** de la clase **Thread**. Esta llamada provoca que la máquina virtual de Java llame al método **run** de la hebra y ésta empiece a ejecutarse.
 - dentro de **run** se define el código que constituye la nueva hebra
 - el método **run** puede llamar a otros métodos, usar otras clases y declarar variables igual que se hace en el hilo principal.
 - una hebra terminará cuando **run** devuelva el control

```
public class MiHebra implements Runnable {
    public void run() {
        System.out.println("Comienza: " + Thread.currentThread());
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(Thread.currentThread().getName() + ": " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + "
                                interrumpida: " + e);
        }
        System.out.println("Saliendo de:" + Thread.currentThread().getName());
    }
}
```

La hebra escribe i y espera
500 ms. Esto lo hace cinco
veces

MiHebra.java

```
public class MultiplesHilosDemo {

    public static void main(String args[]) {
        Thread cero = new Thread(new MiHebra());
        Thread uno = new Thread(new MiHebra());
        Thread dos = new Thread(new MiHebra());
        cero.start(); uno.start(); dos.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Hebra " +
                Thread.currentThread().getName() + " interrumpida: " + e);
        }

        System.out.println("Saliendo de la hebra principal " +
            Thread.currentThread().getName());
    }
}
```

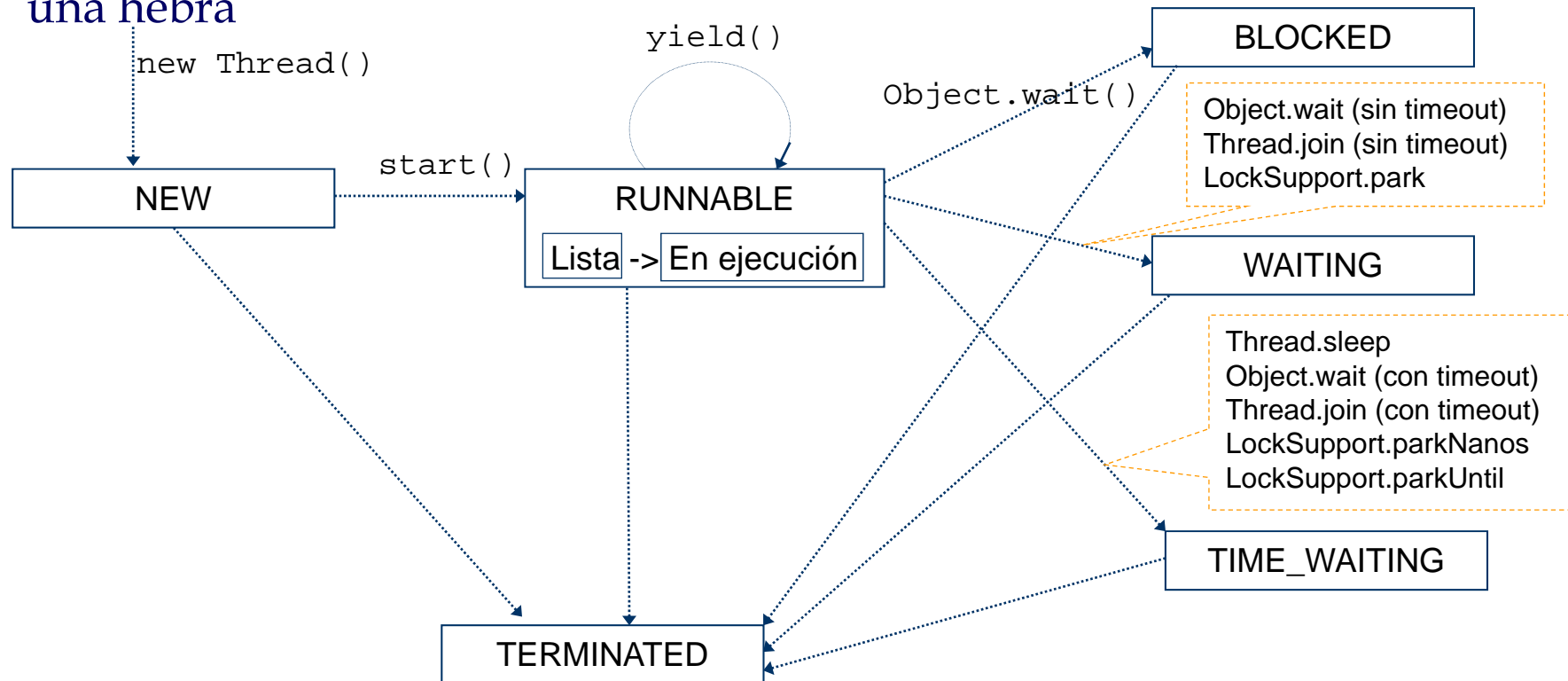
Esta clase lanza tres hebras y hace una pausa de 10 segundos.

MultiplesHilosDemo.java

```
Comienza: Thread[Uno,5,main]
Comienza: Thread[Dos,5,main]
Comienza: Thread[Tres,5,main]
Uno: 5
Dos: 5
Tres: 5
Uno: 4
Dos: 4
Tres: 4
Saliendo de la hebra principal
Uno: 3
```

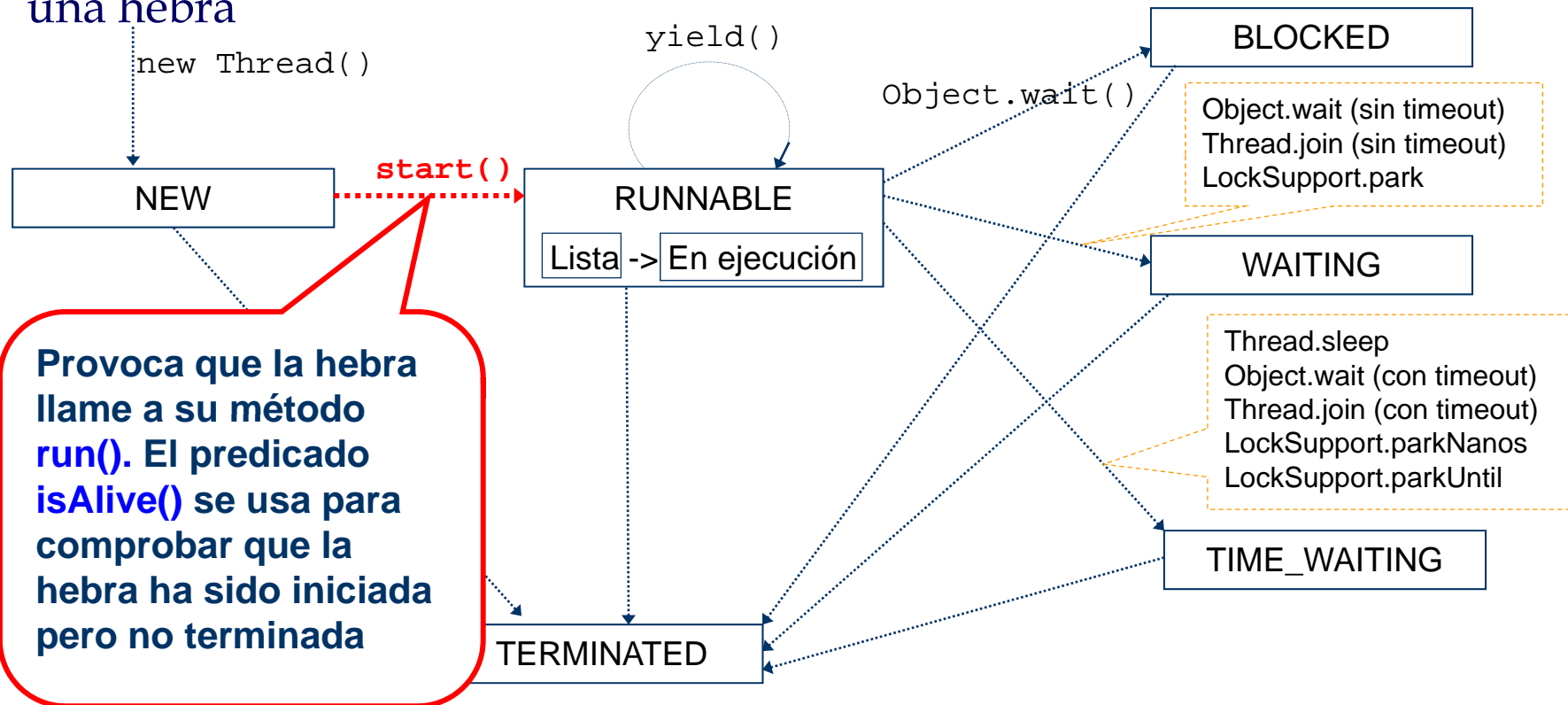
```
Dos: 3
Tres: 3
Uno: 2
Dos: 2
Tres: 2
Uno: 1
Dos: 1
Tres: 1
Saliendo de: Uno
Saliendo de: Dos
Saliendo de: Tres
```

- **java.lang.Thread.State:** define los distintos estados en los que puede estar una hebra



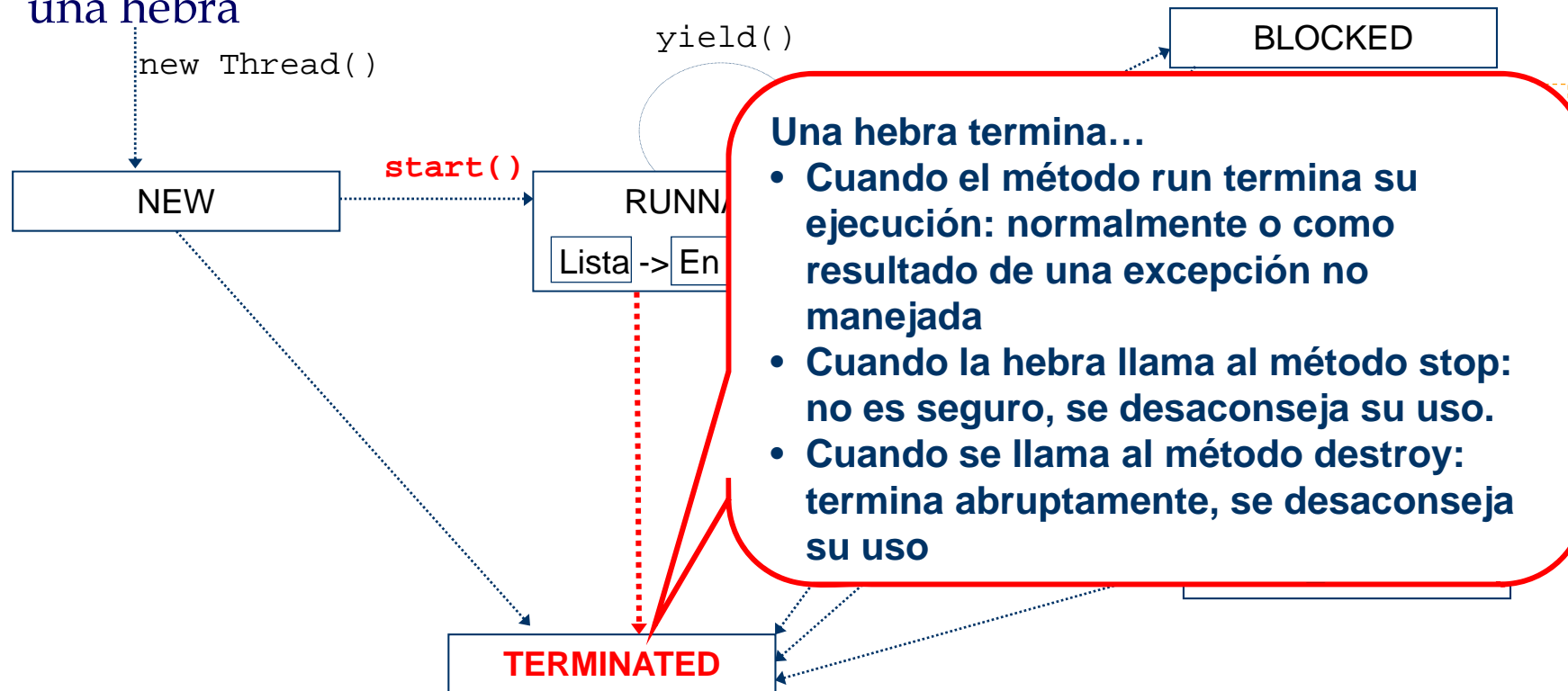
- **NEW:** la hebra todavía no ha comenzado su ejecución
- **RUNNABLE:** la hebra se está ejecutando en la JVM
- **BLOCKED:** la hebra está bloqueada esperando un *lock* de un monitor
- **WAITING:** la hebra está esperando indefinidamente a que otra hebra desarrolle su tarea.
- **TIMED_WAITING:** una hebra está esperando un tiempo determinado a que otra hebra acabe.
- **TERMINATED:** la hebra ha finalizado su ejecución.

- **java.lang.Thread.State**: define los distintos estados en los que puede estar una hebra



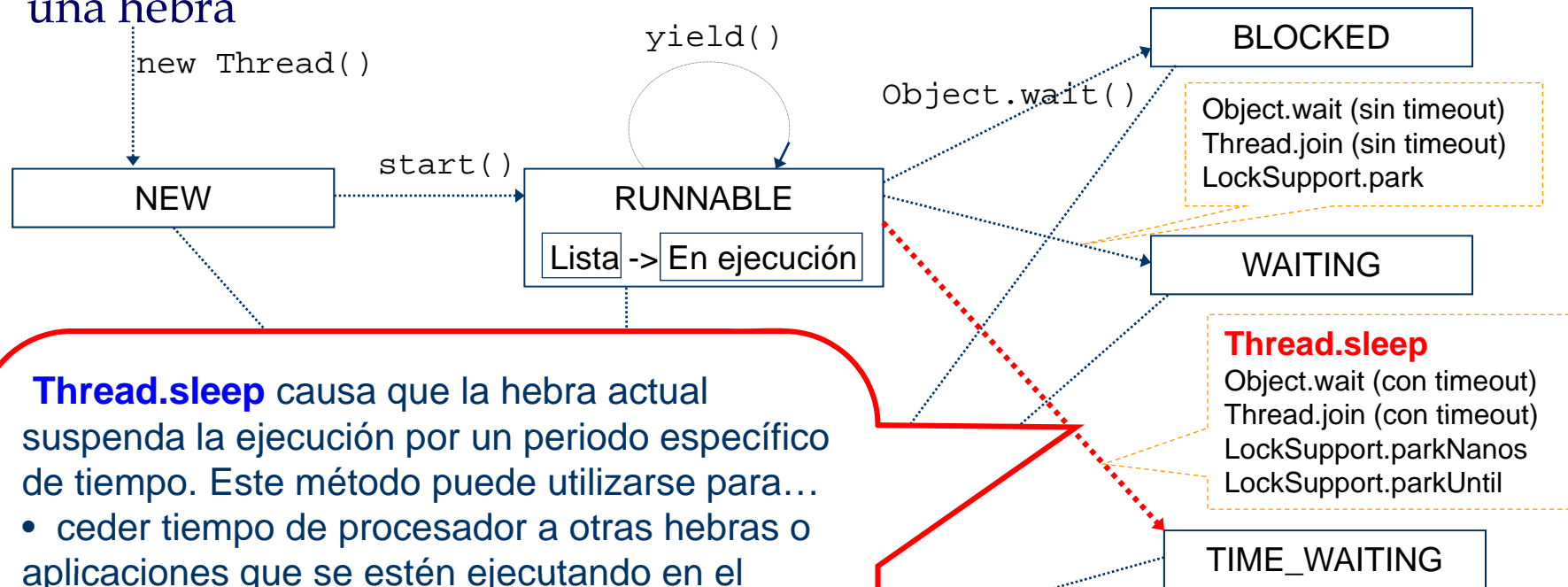
- **NEW**: la hebra todavía no ha comenzado su ejecución
- **RUNNABLE**: la hebra se está ejecutando en la JVM
- **BLOCKED**: la hebra está bloqueada esperando un *lock* de un monitor
- **WAITING**: la hebra está esperando indefinidamente a que otra hebra desarrolle su tarea.
- **TIMED_WAITING**: una hebra está esperando un tiempo determinado a que otra hebra acabe.
- **TERMINATED**: la hebra ha finalizado su ejecución.

- **java.lang.Thread.State**: define los distintos estados en los que puede estar una hebra



- **NEW**: la hebra todavía no ha comenzado su ejecución
- **RUNNABLE**: la hebra se está ejecutando en la JVM
- **BLOCKED**: la hebra está bloqueada esperando un *lock* de un monitor
- **WAITING**: la hebra está esperando indefinidamente a que otra hebra desarrolle su tarea.
- **TIMED_WAITING**: una hebra está esperando un tiempo determinado a que otra hebra acabe.
- **TERMINATED**: la hebra ha finalizado su ejecución.

- **java.lang.Thread.State:** define los distintos estados en los que puede estar una hebra



Thread.sleep causa que la hebra actual suspenda la ejecución por un periodo específico de tiempo. Este método puede utilizarse para...

- ceder tiempo de procesador a otras hebras o aplicaciones que se estén ejecutando en el sistema
- establecer un ritmo de ejecución del programa para
- permitir que otras hebras con tareas que requieren un tiempo determinado se ejecuten

- **NEW**: la hebra está esperando a que se cree
- **RUNNABLE**: la hebra está lista para ser ejecutada

- **BLOCKED**: la hebra está bloqueada esperando un *lock* de un monitor

WAITING: la hebra está esperando indefinidamente a que otra hebra desarrolle su trabajo.

TIME_WAITING: una hebra está esperando un tiempo determinado a que otra hebra acabe.

- **TERMINATED**: la hebra ha finalizado su ejecución.

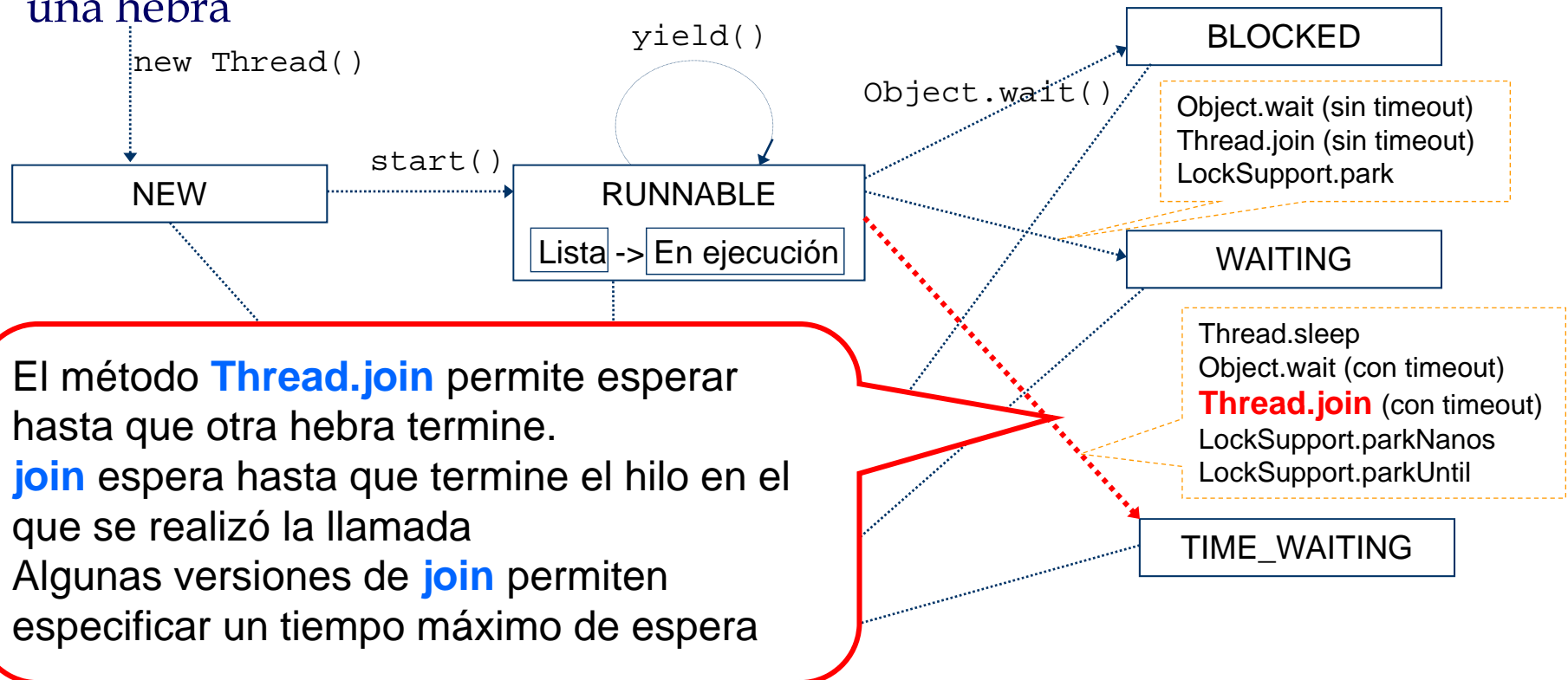
```
public class ContadorSleep implements Runnable {
    int n=10;
    Thread t;

    public ContadorSleep() {
        t = new Thread(this);
        System.out.println("COMIENZA " + t.getName());
        t.start();
    }
    public void run() {
        while (n>0) {
            System.out.println(t.getName() + ": " + n);
            n--;
            try {
                Thread.sleep(500);
            } catch (InterruptedException ex) {
                System.out.println(ex.getMessage());
            }
        }
        System.out.println("FINALIZA " + t.getName());
    }
}
```

ContadorSleep.java

ContadorSleepRun.java

- **java.lang.Thread.State**: define los distintos estados en los que puede estar una hebra



- **NEW**: la hebra todavía no ha comenzado su ejecución
- **RUNNABLE**: la hebra se está ejecutando en la JVM
- **BLOCKED**: la hebra está bloqueada esperando un *lock* de un monitor
- **WAITING**: la hebra está esperando indefinidamente a que otra hebra desarrolle su tarea.
- **TIMED_WAITING**: una hebra está esperando un tiempo determinado a que otra hebra acabe.
- **TERMINATED**: la hebra ha finalizado su ejecución.

```
/**
 * Esta clase se asegura que el hilo principal es el último en terminar
 */
public class JoinDemo {
    public static void main(String args[]){
        MiHebra2 h1 = new MiHebra2("Uno");
        MiHebra2 h2 = new MiHebra2("Dos");
        MiHebra2 h3 = new MiHebra2("Tres");
        System.out.println("La hebra Uno ¿está viva?:" + h1.t.isAlive());
        System.out.println("La hebra Dos ¿está viva?:" + h2.t.isAlive());
        System.out.println("La hebra Tres ¿está viva?:" + h3.t.isAlive());
        // Espera a que terminen todas las hebras
        try {
            System.out.println("Esperando que terminen...");
            h1.t.join();
            h2.t.join();
            h3.t.join();
        } catch (InterruptedException e){
            System.out.println("Hebra principal interrumpida");
        }
        System.out.println("La hebra Uno ¿está viva?:" + h1.t.isAlive());
        System.out.println("La hebra Dos ¿está viva?:" + h2.t.isAlive());
        System.out.println("La hebra Tres ¿está viva?:" + h3.t.isAlive());
    }
}
```

MiHebra2.java

JoinDemo.java

- Cuando se interrumpe una hebra se le indica que pare lo que está haciendo y haga alguna otra tarea.
- Corresponde al programador decidir exactamente como responderá una hebra a una interrupción. La opción más común es que la hebra termine.
- Una hebra lanza una interrupción invocando el método `interrupt` del objeto **Thread** de la hebra que va a ser interrumpida.
- El mecanismo de interrupción se implementa usando un *flag* interno conocido como *interrupt status*. La llamada a **interrupt()** habilita este *flag*.
- Para que el mecanismo de interrupción funcione correctamente la hebra interrumpida debe gestionar su propia interrupción, por ejemplo:

```
if (Thread.interrupted()) {  
    // La hebra ha sido interrumpida por algún proceso u otra hebra.  
    // Se termina su ejecución  
    return;  
}
```

- Cuando una hebra comprueba si ha habido una interrupción usando el método estático **interrupted()** que restaura el *interrupt status*.
- El método no estático **isInterrupted()** no cambia el *flag*.
- Si una hebra es interrumpida durante la invocación de los métodos **wait** de **Object** o **join** o **sleep** de **Thread** recibirá una excepción **InterruptedException**
- Una hebra puede interrumpirse a sí misma.
- Una hebra necesita tener permiso para poder interrumpir a otra. Si lo intenta se ejecuta el método **checkAccess** que puede lanzar la excepción **SecurityException**

ContadorInterrupt.java

ContadorInterruptRun.java

- El tiempo de CPU del que dispondrá una hebra depende de la política de planificación del Sistema Operativo. El comportamiento de un programa con múltiples hebras puede variar dependiendo del SO.
- Existen sistemas que se basan en las prioridades de las hebras.
- Para establecer la prioridad de una hebra en Java se utiliza el método **setPriority**. El valor de la prioridad asignada debe estar entre MIN_PRIORITY y MAX_PRIORITY. Actualmente estos valores son 1 y 10.
- Cuando se crea una hebra se le asigna por defecto la prioridad NORM_PRIORITY que es 5.

- Las hebras se **comunican** principalmente **compartiendo** el acceso a determinados **campos** (incluyendo referencias a otros objetos).
- Cuando dos o más hebras necesitan **acceder a un recurso compartido** es necesario asegurar que accederán a él de uno en uno, de lo contrario pueden producirse
 - **Interferencias**: ocurren cuando dos operaciones (compuestas por varios pasos) se están ejecutando simultáneamente en distintas hebras pero sobre el mismo dato, solapándose entre sí.
 - **Errores de consistencia de la memoria**: ocurren cuando diferentes hebras tienen diferentes vistas de lo que debería ser el mismo dato. Por ejemplo, supongamos que queremos mostrar por pantalla un valor después de ser incrementado. Si la sentencia que lo incrementa y la que lo muestra están en diferentes hebras nada nos asegura que se ejecuten en el orden correcto y podemos estar mostrando por pantalla un valor que no es el que se debería mostrar. Existen ocasiones en las que es necesario asegurarse de que determinadas acciones (de distintas hebras) se ejecutarán siempre en el mismo orden..

- Veamos un ejemplo para ilustrar un caso de interferencia entre hebras. Supongamos que las instrucciones `c++` y `c--` de la clase `Counter` se divide en tres pasos:
 - Leer de memoria el valor de `c`
 - Incrementar o decrementar el valor de `c`
 - Volcar en memoria el resultado
- Supongamos que la hebra A invoca el método `increment` al mismo tiempo que B invoca `decrement` y que, debido a la planificación del SO, se produce la siguiente secuencia de acciones (el valor inicial de `c` es 0):
 - Hebra A: Lee `c`
 - Hebra B: Lee `c`.
 - Hebra A: Incrementa el valor leído; el resultado es 1.
 - Hebra B: Decrementa el valor leído; el resultado es -1.
 - Hebra A: Almacena el resultado en `c`; `c` vale 1.
 - Hebra B: Almacena el resultado en `c`; `c` vale -1.
- La ejecución de la hebra B ha sobrescrito la de la hebra A, pero podía haber sido al revés. El resultado no es predecible.

```
class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

- Mientras una hebra esté dentro de un método sincronizado todas las demás hebras que intenten llamar a esos métodos, o a otro método sincronizado, sobre la misma instancia tendrán que esperar.
- Para hacer un método sincronizado se le añade la clave **synchronized**
- No se pueden declarar constructores sincronizados

```
class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() {  
        c++;  
    }  
    public synchronized void decrement() {  
        c--;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}
```

- ✖ El siguiente ejemplo muestra como una clase que implementa Runnable (Caller) contiene un objeto de otra clase (Callme) con un método (call) que imprime un mensaje por pantalla. Desde la función principal se crean distintas hebras que llamarán a ese método. El ejemplo ilustra la diferencia entre usar un método sincronizado o no.

```
public class Callme {
    void call(String mensaje){
        System.out.println("[ "+ mensaje);
        // Detiene el hilo durante un segundo antes de imprimir el corchete
        // cierre
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e){
            System.out.println("Hebra interrumpida");
        }
        System.out.println("]");
    }
}
```

Callme.java

```
/**
 * Esta clase contiene un objeto Callme y crea una hebra que llama al método call
 */
public class Caller implements Runnable{
    String msj;
    Callme obj;
    Thread t;

    public Caller(Callme o, String s) {
        obj = o;
        msj = s;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        obj.call(msj);
    }
}
```

Caller.java

```
public class CallDemo {
    public static void main(String[] args) {
        Callme c = new Callme();
        Caller ob1 = new Caller(c, "Hola");
        Caller ob2 = new Caller(c, "método");
        Caller ob3 = new Caller(c, "sincronizado");
        //Espera a que terminen las hebras
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Hebra principal interrumpida" + e);
        }
    }
}
```

CallDemo.java

La salida sería:

```
[Hola[método[sincronizado]
]
]
```

- ✖ Si se declara el método call como sincronizado el resultado cambia...

```
synchronized void call(String mensaje){
    System.out.println("[ "+ mensaje);
    // Detiene el hilo durante un segundo antes de
    // imprimir el corchete
    // cierre
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e){
        System.out.println("Hebra interrumpida");
    }
    System.out.println("]");
}
```

CallmeSincronizado.java

La salida sería:

```
[Hola]
[método]
[sincronizado]
```

- ✖ Otra forma de crear código sincronizado es con sentencias sincronizadas

```
synchronized (object) {
    //sentencias que deben sincronizarse
}
```

- ✖ Las sentencias sincronizadas deben especificar el objeto sobre el que se realizará el bloqueo. En el siguiente ejemplo un objeto sincroniza algunas instrucciones de uno de sus métodos.

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- ✖ Las sentencias sincronizadas permiten mejorar la concurrencia evitando bloqueos innecesarios.
- ✖ Una alternativa al ejemplo anterior es no declarar el método **call** como sincronizado y hacer sincronizada la llamada a dicho método dentro del método **run** de **Caller**.

```
public void run() {
    synchronized (obj){
        obj.call(msj);
    }
}
```

- El sondeo es un mecanismo por el cual se comprueba repetitivamente si se cumple una condición. Para evitarlo Java incluye un mecanismo de comunicación entre procesos mediante los métodos **wait**, **notify** y **notifyAll**.
 - Estos métodos se han implementado como **final** en **Object** de modo que están en todas las clases.
 - Sólo pueden llamarse desde un contexto sincronizado
- **wait**
 - Indica a la hebra que realiza la llamada que ha de abandonar el monitor y quedar suspendida hasta que alguna otra hebra entre en el mismo monitor y llame a notify
- **notify**
 - Despierta a la primera hebra que llamó a la función wait de ese mismo objeto
- **notifyAll**
 - Despierta a todas las hebras que llamaron a wait en el mismo objeto. Una de ellas tendrá el acceso garantizado.

Productor.java

Consumidor.java

ProducirConsumir.java

Sincronización: suspensión, reanudación y finalización de hebras

- ✗ Una hebra debe diseñarse de manera que el método run compruebe periódicamente si la hebra tiene que suspenderse, reanudar o detener su propia ejecución.
- ✗ Estas operaciones se realizan estableciendo una variable de marca que indique el estado de ejecución de la hebra.

NuevaHebra.java

```
public class NuevaHebra implements Runnable {
    String nombre;
    Thread t;
    boolean suspendFlag;

    NuevaHebra(String nombrehebra) {
        nombre = nombrehebra;
        t = new Thread(this, name);
        System.out.println("Nueva hebra" + t);
        suspendFlag = false;
        t.start();
    }

    public void run() {
        try {
            for (int i = 15; i>0; i--) {
                System.out.println(nombre + ":" + i);
                Thread.sleep(200);
                synchronized(this){
                    while(supendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(nombre + " interrumpida");
        }
        void suspender() {
            suspendFlag = true;
        }
        synchronized void reanudar() {
            suspendFlag = false;
            notify();
        }
    }
}
```

SuspendDemo.java

```

Class SuspendDemo {
    public static void main (String args[]) {
        NuevaHebra h1 = new NuevaHebra("Uno");
        NuevaHebra h2 = new NuevaHebra("Dos");

        try {
            Thread.sleep(1000);
            h1.suspend();
            System.out.println("Suspend la hebra Uno");
            Thread.sleep(1000);
            h1.reanudar();
            System.out.println("Reanudando la hebra Uno");
            h2.suspend();
            System.out.println("Suspend la hebra Dos");
            Thread.sleep(1000);
            h2.reanudar();
            System.out.println("Reanudando la hebra Dos");
        } catch (InterruptedException e){
            System.out.println("Hebra principal interrumpida");
        }
        //Espera a que las otras hebras terminen
        try {
            System.out.println("Esperando que terminen las otras hebras...");
            h1.t.join();
            h2.t.join();
        } catch (InterruptedException e){
            System.out.println("Hebra principal interrumpida");
        }
    }
}
    
```

Programación para bases de datos

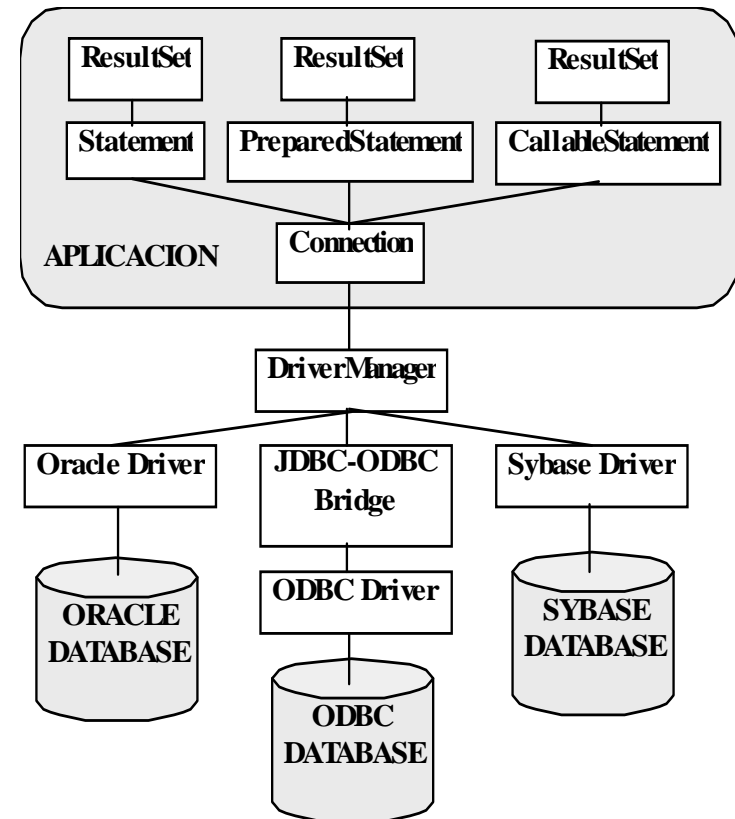
- ¿Qué es JDBC?
 - Es un API de Java que
 - ✓ sirve para comunicar una aplicación con un servidor de bases de datos
 - ✓ permite al programador ejecutar instrucciones en SQL
- ¿Cuáles son sus funciones principales?
 - establecer la conexión a una base de datos (remota o no)
 - enviar sentencias SQL a esa base de datos
 - procesar los resultados obtenidos de la base de datos
- ¿Cómo se comunica JDBC con una base de datos?
 - JDBC es una ESPECIFICACIÓN de un conjunto de clases, interfaces y operaciones que describen la funcionalidad necesaria para conectarse a un SGBD, enviarle una consulta y recibir el resultado de la misma.
 - Una aplicación Java que quiera operar sobre una base de datos necesitará un DRIVER que IMPLEMENTE la funcionalidad de JDBC.
 - Los fabricantes de los distintos Sistemas Gestores de Bases de Datos (Oracle, SysBase, Postgres,...) deben proporcionar un driver JDBC .
 - JDBC-ODBC puente para que las aplicaciones se comuniquen con la base de datos a través de los controladores ODBC de Microsoft.

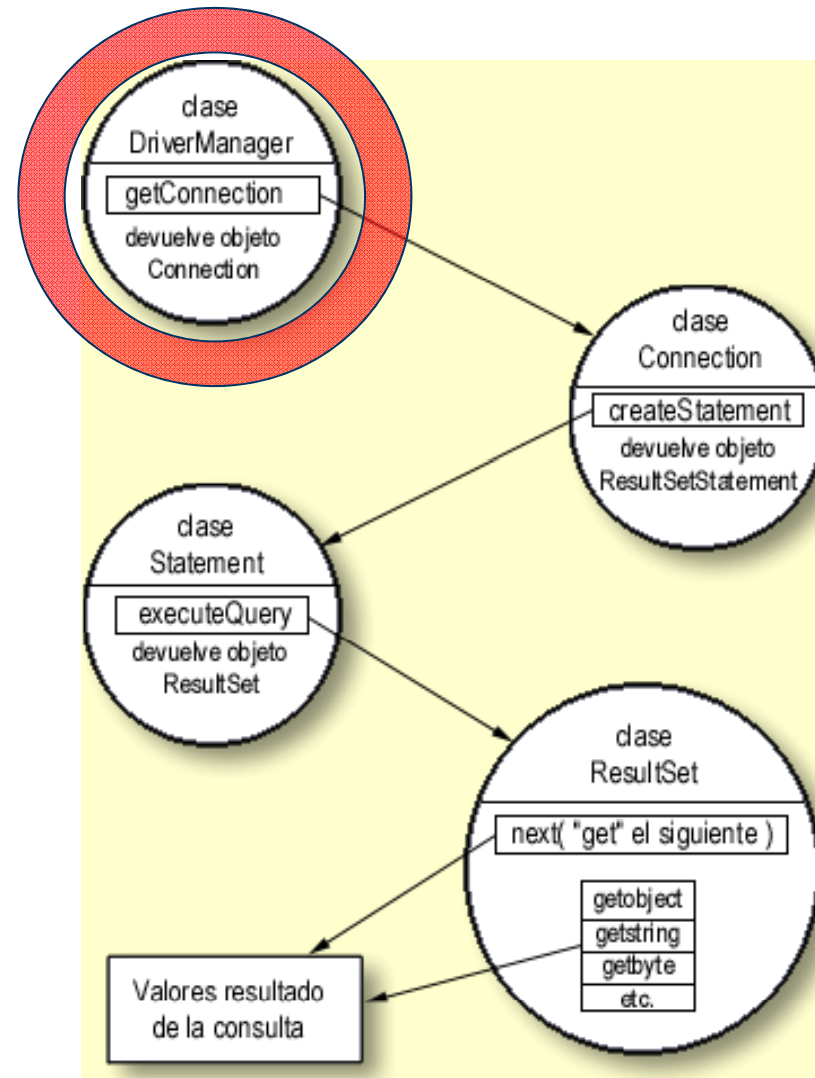
CLASES	DriverManager	Gestiona los drivers instalados en el sistema
	DriverPropertyInfo	Proporciona información acerca de un driver

I N T E R F A C E S	Driver	Es el interfaz que cada gestor de base de datos debe implementar. Permite conectarse a una base de datos
	Connection	Representa una conexión (sesión) con una base de datos. Una aplicación puede tener varias conexiones a distintas bases de datos
	DatabaseMetadata	Proporciona información acerca de una base de datos (tablas que contiene, vistas, funciones,...)
	Stament	Objeto usado para ejecutar una sentencia SQL y obtener los resultados
	PreparedStatement	Representa una sentencia SQL precompilada a la que se le pueden pasar parámetros de entrada
	CallableStatement	Se usa para llamar a procedimientos almacenados en la base de datos
	ResultSet	Una estructura de datos que contiene los registros obtenidos de una consulta SQL
	ResultSetMetadata	Permite obtener información sobre el tipo y las propiedades de las columnas en un objeto ResultSet

Secuencia de pasos para acceder a una base de datos a través de un controlador JDBC que implemente **java.sql.Driver**

- Establecimiento de una conexión con la base de datos:
 - Cargar la clase del driver específico de la base de datos
 - Abrir una conexión con la base de datos cuya localización viene dado por la URL `jdbc:driver:nombre`
- Consulta a la base de datos
 - Crear un objeto de la clase `java.sql.Statement`
 - Utilizar el método `executeQuery` para enviar sentencias SQL
- Recepción del resultado de la consulta
 - Se obtiene un objeto `ResultSet` con el resultado de la consulta
- Cierre de la conexión





- Establecer una conexión con la base de datos conlleva dos pasos:

1. Cargar el driver

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

2. Realizar la conexión: una aplicación JDBC se conecta al origen de los datos utilizando alguno de los siguientes mecanismos:

1. **DriverManager**: esta clase necesita una aplicación para cargar un driver concreto a partir de una URL.

Connection con =

```
DriverManager.getConnection("jdbc:oracle:thin:scott/tiger@10.137.2.21:1521:HPUXMA")
```

Driver
(subprotocolo)

Usuario

Password

IP del servidor
de bases de
datos

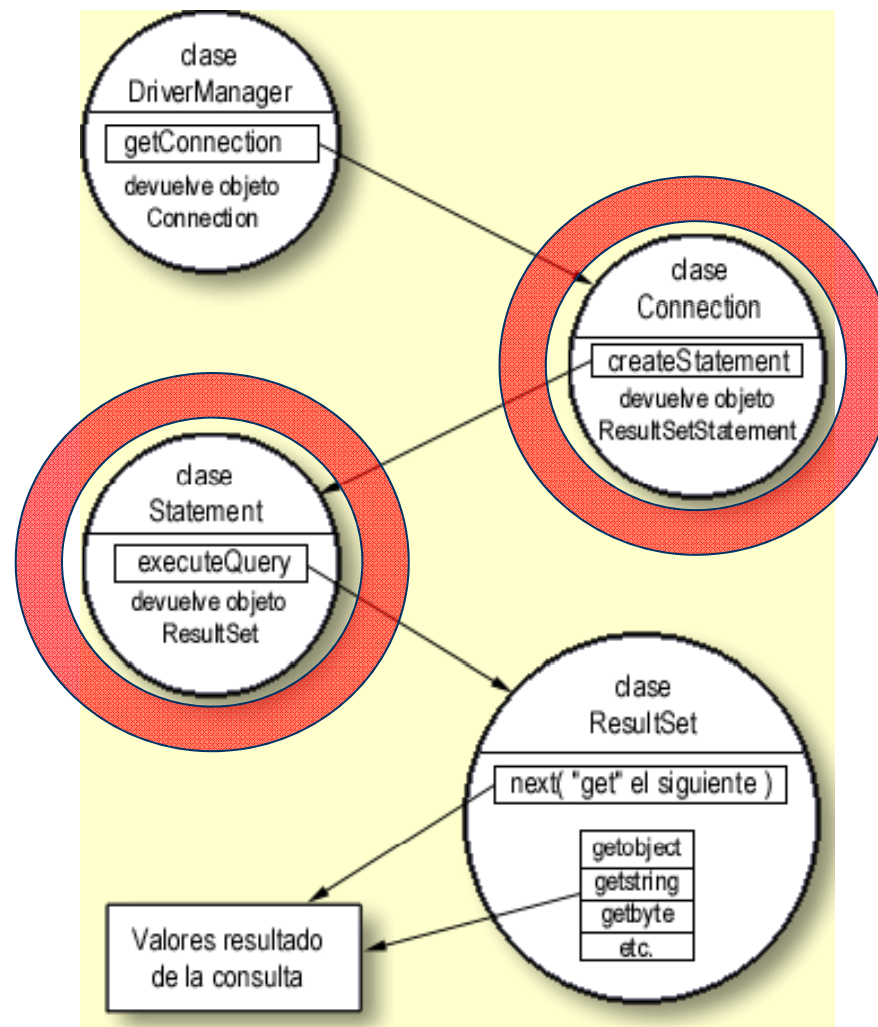
Nombre
de base
de datos

2. **DataSource**: este objeto es la representación de una fuente de datos en Java. Puede representar desde una sofisticada base de datos hasta un simple fichero con filas y columnas. Puede residir en un servidor remoto o en una máquina local. Las aplicaciones acceden a la fuente de los datos usando una conexión.

- Establecer una conexión con la base de datos conlleva dos pasos:
 1. Cargar el driver
 2. Realizar la conexión
 1. DriverManager
 2. **DataSource:**
 - 2.1. Crear y registrar un objeto DataSource
 - » Normalmente se crea, se despliega y se maneja separadamente de las aplicaciones Java que la usan.
 - » Un objeto DataSource normalmente es creado y desplegado por los desarrolladores o los administradores del sistema, no por los usuarios y probablemente se hará usando alguna herramienta GUI.
 - » Un DataSource se registra con un nombre de servicio JNDI. JNDI proporciona a una aplicación una manera uniforme de encontrar y acceder a servicios remotos sobre la red. Estos servicios pueden ser una aplicación específica o una base de datos. La raíz de la jerarquía JNDI es el contexto inicial (initial context) bajo el que existen subcontextos entre los que se encuentra el reservado para los JDBC data sources. El último elemento en la jerarquía es el objeto que se está registrando

- Establecer una conexión con la base de datos conlleva dos pasos:
 1. Cargar el driver
 2. Realizar la conexión
 1. DriverManager
 2. DataSource
 - 2.2. Conectarse a un DataSource
 - » Supongamos que tenemos registrado un DataSource con el nombre AcmeDB:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/AcmeDB");
Connection con = ds.getConnection("genius", "abracadabra");
.....
con.close();
```



- Para **realizar una consulta** se utiliza un objeto de la clase Statement que me representa la sentencia SQL que quiero enviar al SGBD.
- Para ello se utiliza alguno de los métodos execute que tiene Statement pasándole como argumento una cadena con la sentencia SQL.
 - **executeQuery** ejecuta una sentencia SELECT y devuelve un objeto **ResultSet** con los datos
 - **executeUpdate** envía una sentencia INSERT, UPDATE o DELETE y devuelve el número de tuplas afectadas por dicha sentencia

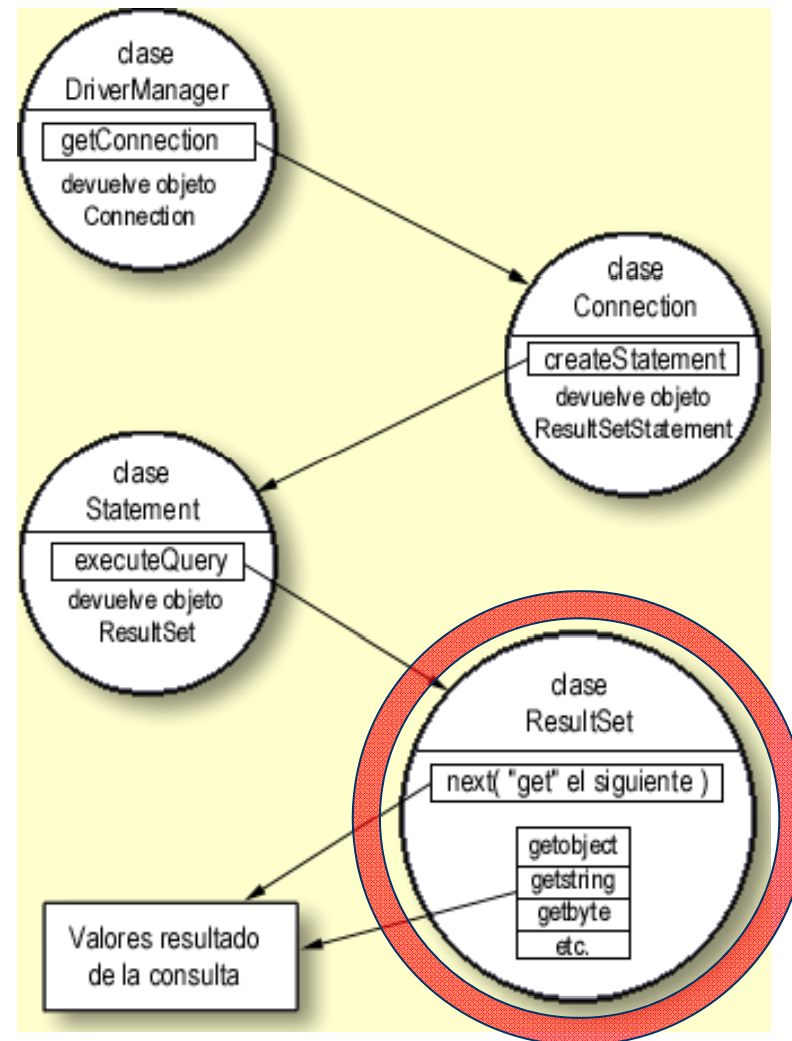
```
Statement stmt = con.createStatement();
```

```
ResultSet resultado = stmt.executeQuery("SELECT * FROM TABLA");
```

```
int numTuplas1 = stmt.executeUpdate("INSERT...");
```

```
int numTuplas2 = stmt.executeUpdate("UPDATE...");
```

```
int numTuplas3 = stmt.executeUpdate("DELETE...");
```



- Una vez realizada la consulta el SGBD nos envía el **resultado** que se **recoge** en un objeto de la clase ResultSet.
- Un ResultSet contiene el conjunto de filas resultado de una consulta SELECT y mantiene un cursor apuntando a la fila actual. Cuando se crea el cursor se posiciona antes de la primera fila.
- Para mover el cursor podemos usar (entre otros) el método next y para acceder a las distintas columnas los distintos métodos get

```
while (resultado.next()) {  
    String campo = resultado.getString("campo");  
    int campo2    = resultado.getInt("campo2");  
    ...  
}
```

- ¿Cómo consultar una tabla cuando no sé su estructura?
 - Utilizaremos la clase ResultSetMetaData que almacena información acerca de la estructura de las tablas consultadas. Supongamos que queremos mostrar por pantalla el contenido de una tabla de la que no sabemos nada acerca de sus campos, tipos, etc. Para ello seguiremos los siguientes pasos:

1. Consultar toda la tabla

```
Statement stmt = con.createStatement();  
ResultSet rs   = stmt.executeQuery("SELECT * FROM TABLA");
```

2. Obtener un objeto ResultSetMetaData a partir del resultado de la consulta

```
ResultSetMetaData rsmd = rs.getMetaData();
```

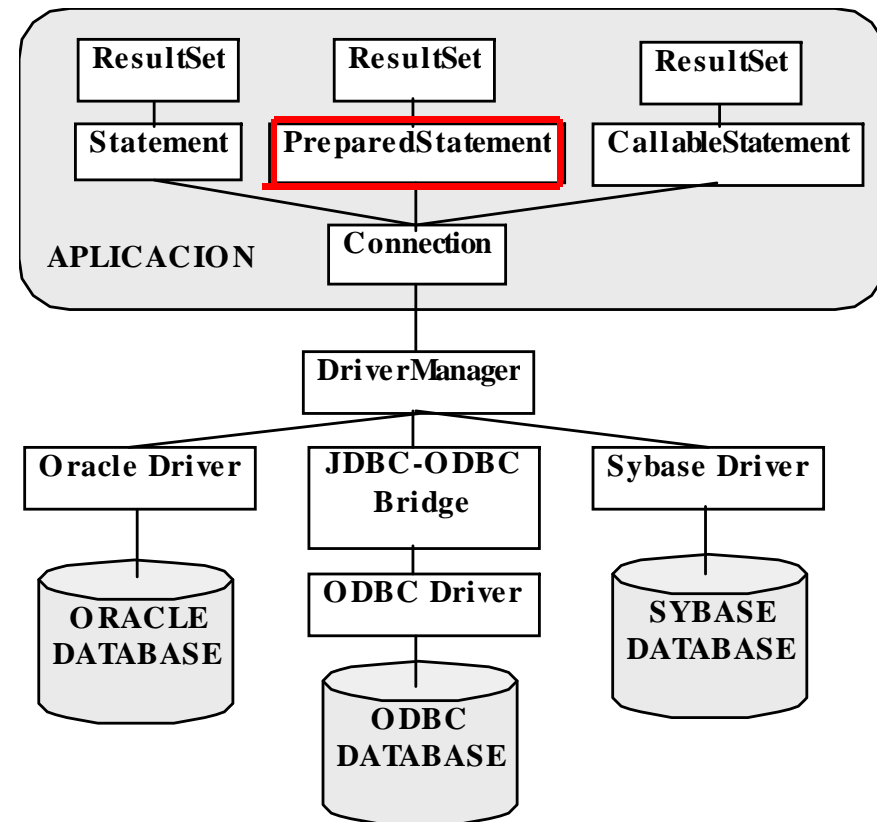
3. Averiguar cuántas columnas tiene la tabla

```
int numberOfColumns = rsmd.getColumnCount();
```

4. Ir recorriendo el ResultSet para obtener los resultados

```
while (rs.next()) {  
    for (int i=1;i<=numberOfcolumns;i++) {  
        String name = rsmd.getColumnName(i);  
        int tipo = rsmd.getColumnType(i);  
        swich (tipo) {  
            case java.sql.Types.VARCHAR:  
                String valor = getString(name);  
                break;  
            case java.sql.Types.DOUBLE:  
                double valor = getDouble(name);  
                break;  
            ....  
        }  
    }  
}
```

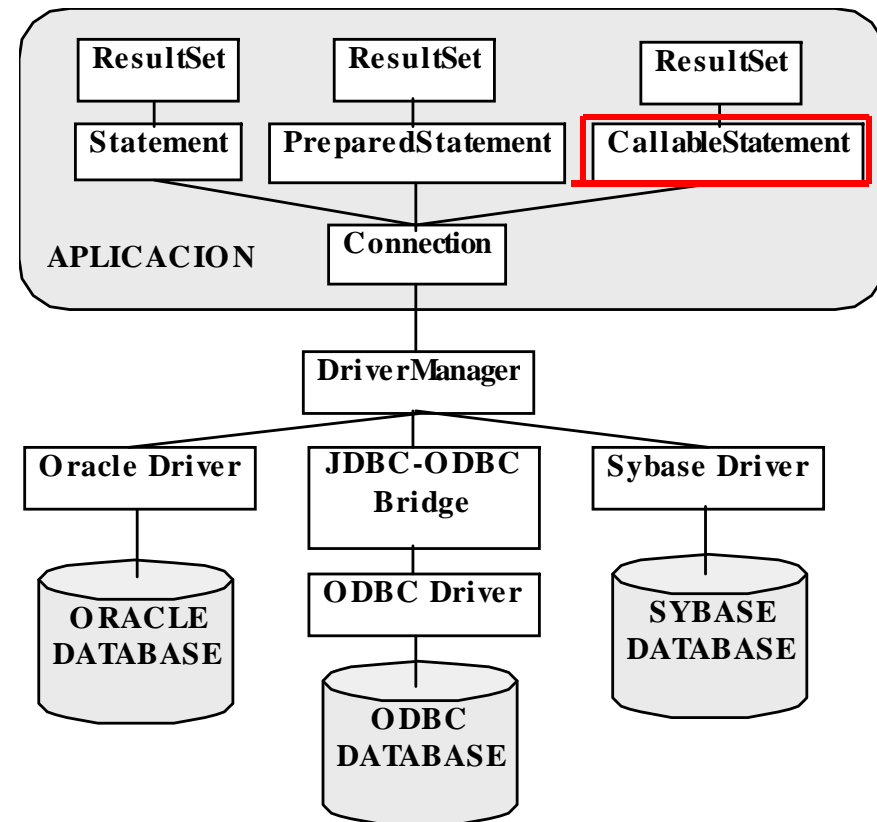
Interfaz PreparedStatement



- Son sentencias SQL compiladas en las que se pueden introducir uno o más parámetros de entrada
- Algunos drivers no soportan precompilación así que la sentencia es enviada a la base de datos en el momento de su ejecución.
- Los ResultSet creados usando PreparedStatement serán por defecto TYPE_FORWARD y CONCUR_READ_ONLY.

```
PreparedStatement pstmt = con.prepareStatement
    ("INSERT INTO tabla (c1,c2,c3) VALUES(?,?,?)");
pstmt.clearParameters();
pstmt.setInt(1,54);
pstmt.setDouble(2, 8.3);
pstmt.setString(3,"Pepe García");
pstmt.executeUpdate();
```

Interfaz CallableStatement



- Para llamar a procedimientos almacenados en el gestor de la base de datos se utiliza la interfaz .
- Esquema de llamada
 - Con valor de retorno
 - ✓ { ?= call nombre_procedimiento ([?,?,...,?]) }
 - Sin valor de retorno
 - ✓ { call nombre_procedimiento ([?,?,...,?]) }
- Los parámetros de entrada se establecen usando los métodos heredados de PreparedStatement.
- Los parámetros de salida deben ser registrados antes de ejecutar el procedimiento almacenado.
- Una CallableStatement puede devolver uno o más ResultSet. Si hay varios objetos ResultSet se manejarán usando las operaciones heredadas de Statement.

- Procedimiento que devuelve un solo valor por un parámetro OUT

```
CallableStatement cstmt = con.prepareCall("{call ejemplo(?,?)}");
cstmt.registerOutParameter(1, java.sql.Types.CHAR);
cstmt.setBoolean(2,true);
cstmt.execute();
byte x = cstmt.getBytes(1);
....
```

- Función que devuelve un valor

```
CallableStatement upperProc = con.prepareCall("{ ? = call upper( ? ) }");
upperProc.registerOutParameter(1, Types.VARCHAR);
upperProc.setString(2, "lowercase to uppercase");
upperProc.execute();
String upperCased = upperProc.getString(1);
upperProc.close();
```

- Función que devuelve un ResultSet

```
CallableStatement proc = con.prepareCall("{ ? = call doquery ( ? ) }");
proc.registerOutParameter(1, Types.Other);
proc.setInt(2, -1);
ResultSet results = proc.executeQuery();
while (results.next()) {
    // hacer algo con los resultados
}
results.close();
proc.close();
```

- Un objeto ResultSet mantiene un cursor apuntando a la fila de datos actual
- Inicialmente el cursor se sitúa antes de la primera fila
- El método next lo hace avanzar hacia la siguiente fila y devuelve false cuando no hay más filas
- Por defecto un objeto ResultSet:
 - ✓ No se puede actualizar
 - ✓ El cursor se mueve sólo hacia delante
- A partir del API JDBC 2.0 esto se puede modificar, para ello se debe crear del siguiente modo:

```
Statement stmt = con.createStatement(  
                                ResultSet.TYPE_SCROLL_SENSITIVE,  
                                ResultSet.CONCUR_UPDATABLE);
```

- Para desplazar el cursor se pueden usar los métodos:

```
public void afterLast()  
public boolean first()  
public boolean last()  
public void beforeFirst()  
public void moveToCurrentRow()  
...
```

- Por defecto una conexión funciona en modo autocommit
 - ✓ cada vez que se ejecuta una sentencia SQL se abre y se cierra automáticamente una transacción que sólo afecta a dicha sentencia
- NO modo autocommit
 - ✓ `setAutoCommit(false)`
 - ✓ Es necesario cerrar las transacciones usando los métodos
 - `commit()` si queremos que los cambios hechos desde la última transacción sean efectivos
 - `rollback()` si queremos anular todas las operaciones hechas desde la última transacción
- Nivel de aislamiento de una transacción
 - ✓ `setTransactionIsolation(int level)`
 - ✓ level:
 - `TRANSACTION_NONE`: no se pueden usar transacciones
 - `TRANSACTION_READ_UNCOMMITTED`: sólo ver registros modificados por otra transacción, no guardarlos
 - `TRANSACTION_READ_COMMITTED`: se ven sólo las modificaciones ya guardadas
 - `TRANSACTION_REPEATABLE_READ`: si se leyó un registro y otra transacción lo modifica y lo guarda, en la siguiente lectura seguiremos viendo el primer valor
 - `TRANSACTION_SERIALIZABLE`: se ven todos los registros como estaban antes de comenzar la transacción

- Algunos tipos de datos estándar de SQL no tienen su equivalente nativo en Java
- Es necesario mapear los tipos de datos SQL en Java utilizando las clases JDBC para acceder a los tipos de datos SQL
- Existe una constante para cada tipo de dato SQL en `java.sql.Types`
- JDBC proporciona clases de Java nuevas para representar varios tipos de datos SQL:
 - ✓ `java.sql.Date`
 - ✓ `java.sql.Time`
 - ✓ `java.sql.Timestamp`

JAVA	SQL
String	CHAR
String	VARCHAR
String	LONGVARCHAR
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	FLOAT
double	DOUBLE

Java	SQL
byte[]	BINARY
byte[]: imágenes, sonidos	VARBINARY (BLOBs)
byte[]	LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.math.BigDecimal	NUMERIC
java.math.BigDecimal	DECIMAL

Netbeans

- Generación automática de comentarios (Tools→Auto Comment...)
- Generación de documentación del proyecto (Build→Generate Javadoc)
- Reformatear código (Boton derecho sobre el código→Reformat Code)
- Internacionalización: (Tools→Internationalization→Internationalize...)
- En una clase se pueden generar los métodos de acceso a los atributos getter y setter de forma automática. Hay dos métodos
 1. Después de escribir un atributo de la clase pulsar ctrl+espacio eso mostrará un menú contextual como el siguiente que muestra las opciones de crear getter y setter para cada atributo
 2. La segunda opción consiste en:

Sobre alguno de los atributos acceder al menú contextual opciones refactor→Encapsulate fields
Aparecerá una lista con todos los atributos. Seleccionarlos para que se generen los métodos de acceso y en la ventana de salida “Output” elegir “Do refactoring” y todos los métodos se habrán creado

