

Biblioteca de Clases



Contenido

- Organización en paquetes
- Clases básicas: `java.lang`
- El paquete `java.util`

Organización en paquetes (packages)

- Un paquete en Java es un mecanismo para agrupar clases e interfaces relacionados desde un punto de vista lógico, con una protección de acceso, que delimita un ámbito para el uso de nombres.
- La plataforma Java incorpora unos paquetes predefinidos para facilitar determinadas acciones.
- Se pueden definir paquetes nuevos que deberán incluirse en el CLASSPATH.

Creación de paquetes

- Para definir un paquete hay que encabezar cada fichero que componga el paquete con la declaración

package <nombre>;

- Cuando no aparece esta declaración, se considera que las clases e interfaces de los ficheros pertenecen a un paquete anónimo.

Uso de paquetes

- Desde fuera de un paquete sólo se puede acceder a clases e interfaces **public** (exceptuando el acceso a clases herederas en el caso de declaraciones **protected**).
- Para acceder desde otro paquete a una clase o interfaz se puede:
 - Utilizar el nombre calificado con el nombre del paquete
gráfico.Rectángulo r;
 - importarla al comienzo del fichero y usar su nombre simple
import gráfico.Rectángulo;
 - importar el paquete completo al comienzo del fichero y usar los nombres simples de todas las clases e interfaces del paquete
import gráfico.*;
- El sistema de ejecución de Java importa de forma automática el paquete anónimo, **java.lang** y el paquete actual.

API

(Application Programming Interface)

- API es una biblioteca de paquetes que se suministra con la plataforma de desarrollo de Java (J2SDK).
- Estos paquetes contienen interfaces y clases diseñados para facilitar la tarea de programación.
- Los paquetes más básicos son: **java.lang** y **java.util**.

El paquete java.lang

- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object**
 - **System**
 - **Class**
 - **Math**
 - **String, StringBuilder y StringBuffer**
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Cloneable**
 - **Comparable**
 - **Runnable**
- Contiene también excepciones y errores.

La clase Object

- Es la clase superior de toda la jerarquía de clases de Java.
 - Define el comportamiento mínimo común de todos los objetos.
 - Si una definición de clase no extiende a otra, entonces extiende a **Object**. Todas las clases heredan de ella directa o indirectamente.
 - No es una clase abstracta pero no tiene mucho sentido crear instancias suyas.

Métodos de instancia importantes:

- `boolean equals(Object)`
- `Object clone()`
- `String toString()`
- `void finalize()`
- `Class getClass()`
- `int hashCode()`
- ... consultar la documentación.

El método equals ()

- Compara dos objetos de la misma clase.
- Por defecto realiza una comparación por ==.
- Este método se puede redefinir en cualquier clase para comparar objetos de esa clase.
- Todas las clases del sistema tienen redefinido este método.

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad    = e;  
    }  
    public boolean equals(Object o) {  
        return  
            (o instanceof Persona) &&  
            (edad == ((Persona) o).edad) &&  
            (((Persona) o).nombre.equals(nombre));  
    }  
}
```

`equals ()` y `hashCode ()`

- El método `hashCode ()` devuelve un `int` para cada objeto de la clase.
- Hay una relación que debe mantenerse entre `equals()` y `hashCode()`;
`a.equals(b)` \Rightarrow `a.hashCode() == b.hashCode()`
- Todas las clases del API de Java verifican esa relación.

```
class Persona {
    private String nombre;
    private int edad;
    public Persona(String n, int e) {
        nombre = n;
        edad = e;
    }

    public boolean equals(Object o) {
        return (o instanceof Persona) &&
            (edad == ((Persona) o).edad) &&
            (((Persona) o).nombre.equals(nombre));
    }
    public int hashCode() {
        return nombre.hashCode() + edad;
    }
}
```

El método `clone()`

- Realiza la copia de un objeto.
- Para poder utilizarlo, la clase debe implementar la interfaz **Cloneable**. Si no la implementa el método lanza la excepción **CloneNotSupportedException**.
- **clone()** es un método **protected** en **Object**.
- Por defecto crea una instancia nueva y copia por asignación cada campo del objeto (copia superficial).
- Cada subclase puede redefinir el método para realizar una copia adecuada de los objetos de dicha clase.

```
class Persona implements Cloneable {
    private String nombre;
    private int edad;
    public Persona(String n, int e) {
        nombre = n;
        edad    = e;
    }
    public Object clone()
        throws CloneNotSupportedException {
        Persona e = (Persona) super.clone();
        e.nombre = new String(nombre);
        return e;
    }
}
```

La clase **Class**

- Dota al lenguaje de lo que se denomina “capacidad de *reflexión*”.
- Permite interrogar sobre características de una clase:

- Métodos, constructores, interfaces, superclase, etc.

- Conocer el nombre de la clase de un objeto:

- `String getName()`

- Crear una instancia de esa clase:

- `Object newInstance()`

- Saber si un objeto es de la familia de una clase (instancia de la clase o de una clase heredera) :

- `boolean isInstance(Object)`

- Similar al operador: `Object instanceof Class`

- Ejemplo:

- `System.out.println(o.getClass().getName());`

La clase **System**

- Maneja particularidades del sistema.
- Tres variables de clase (estáticas) públicas:
 - `PrintStream out, err`
 - `InputStream in`
- Métodos de clase (estáticos) públicos:
 - `void exit(int)`
 - `long currentTimeMillis()`
 - `void gc()`
 - `void runFinalization()`
 provoca la ejecución inmediata de los `finalize()` pendientes
 - ...
- Consultar documentación para más información.

La clase **Math**

- Incorpora como *métodos de clase* (estáticos), constantes y funciones matemáticas:
 - Constantes
 - `double E`, `double PI`
 - Métodos :
 - `double sin(double)`, `double cos(double)`, `double tan(double)`, `double asin(double)`, `double acos(double)`, `double atan(double)`,...
 - `xxx abs(xxx)`, `xxx max(xxx,xxx)`, `xxx min(xxx,xxx)`, `double exp(double)`, `double pow(double, double)`, `double sqrt(double)`, `int round(double)`,...
 - `double random()`,
 - ...
 - Consultar la documentación para información adicional.

Ej.: `System.out.println(Math.sqrt(34)) ;`

Cadenas de caracteres

- Las cadenas de caracteres se representan en Java como secuencias de caracteres Unicode encerradas entre comillas dobles.
- Para manipular cadenas de caracteres, por razones de eficiencia, se utilizan tres clases incluidas en **java.lang**:
 - **String** - para cadenas constantes
 - **StringBuilder** - para cadenas modificables
 - **StringBuffer** - para cadenas modificables (seguras ante tareas)

La clase **String**

- Cada objeto alberga una cadena de caracteres.
- Los objetos de esta clase se pueden inicializar...
 - de la forma normal:
`String str = new String("¡Hola!");`
 - de la forma simplificada:
`String str = "¡Hola!";`
- Las cadenas de los objetos **String** no pueden modificarse (crecer, cambiar un carácter, ...).
- Una variable **String** puede recibir valores distintos.

Métodos de la clase `String`

- Métodos de consulta:

`length()`

`charAt(int pos)`

`indexOf/lastIndexOf(char car)`

`indexOf/lastIndexOf(String str)`

- Métodos que producen nuevos objetos `String`:

`substring(int posini, int posfin+1)`

`substring(int posini)`

`toUpperCase()`

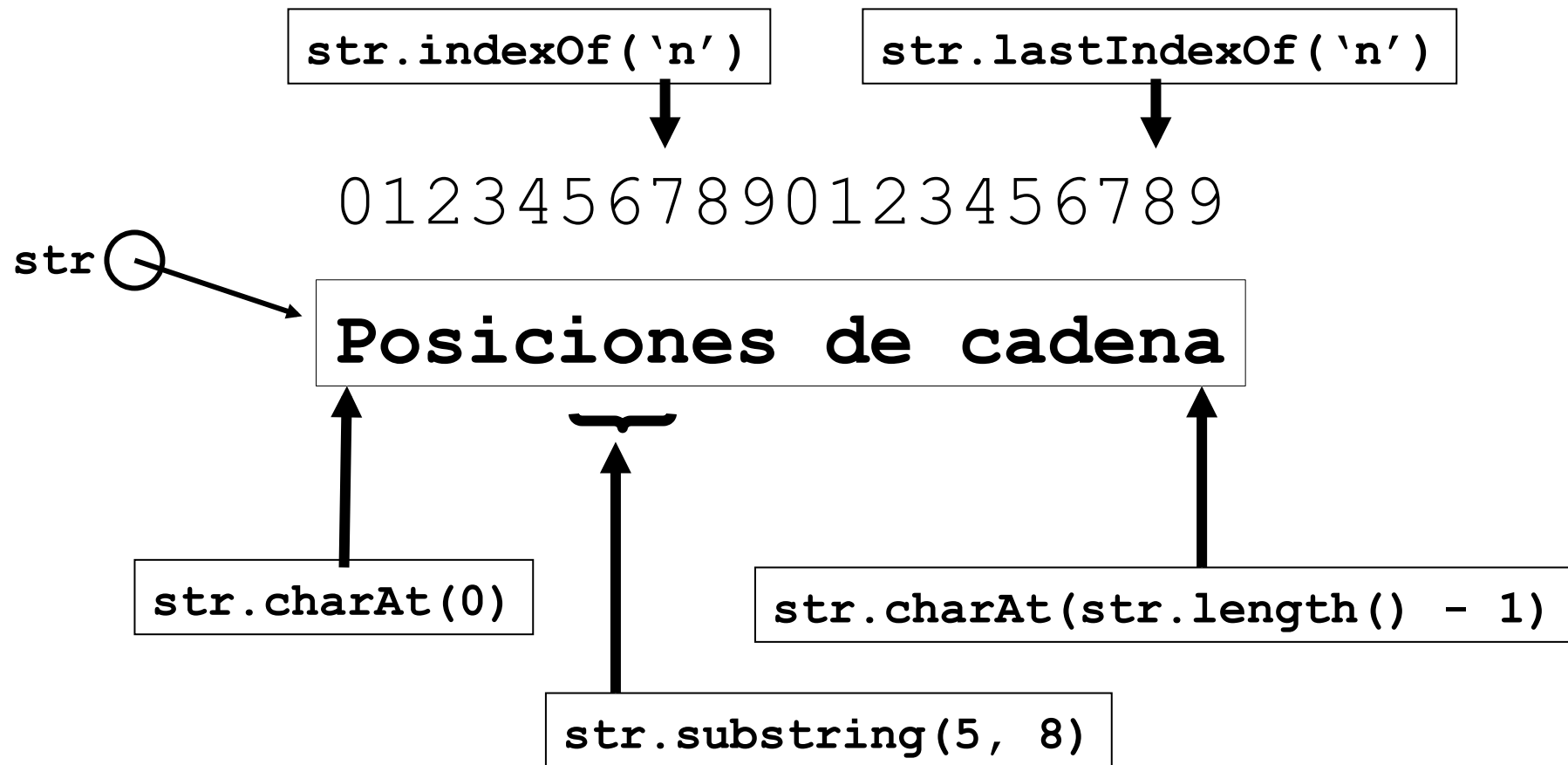
`toLowerCase()`

`static format(String formato,...)`

- Comparación:

`compareTo(String str) // -, 0 ó +`

Posiciones de una cadena



```

public class NombreFichero {
    private String camino;
    private char separadorCamino, separadorExtensión;

    public NombreFichero(String str, char sep, char ext) {
        camino = str;
        separadorCamino = sep;
        separadorExtensión = ext;
    }
    public String extensión() {
        int pto = camino.lastIndexOf(separadorExtensión);
        return camino.substring(pto + 1);
    }
    public String nombre() {
        int pto = camino.lastIndexOf(separadorExtensión);
        int sep = camino.lastIndexOf(separadorCamino);
        return camino.substring(sep + 1, pto);
    }
    public String directorio() {
        // completar (ejercicio)
    }
    ...
}

```

El método estático `format`

- A partir de JDK 1.5.
- Permite construir salidas con formato.

```
String ej = "Cadena de ejemplo";  
String s = String.format("La cadena %s mide %d", ej, ej.length());  
System.out.println(s);
```
- Formatos más comunes (se aplican con %):
 - s para cualquier objeto. Se aplica `toString()`. `"%20s"`
 - d para números sin decimales. `"%7d"`
 - f para números decimales. `"%9.2f"`
 - b para booleanos `"%b"`
 - c para caracteres. `"%c"`
- Se pueden producir las excepciones:
 - `MissingFormatArgumentException`
 - `IllegalFormatConversionException`
 - `UnknownFormatConversionException`
 - ...

El método estático `format`

- Las clases `PrintStream` y `PrintWriter` incluyen el método `printf(String formato,...)`

```
class EjPf {
    static public void main(String[] args) {
        String s = String.format("El objeto %20s con %d", new A(65), 78);
        System.out.println(s);
        System.out.printf(
            "Cadena %40s\nEntero %15d\nFlotante %8.2f\nLógico %b\n",
            "Esto es una cadena", 34, 457.2345678, 3 == 3);
    }
}
```

<pre>class A { int a; public A(int s){ a = s; } public String toString() { return "A[" + a + "];" } }</pre>	<pre>El objeto Cadena Entero Flotante Lógico true</pre>	<pre>A[65] con 78 Esto es una cadena 34 457,23 true</pre>
---	---	---

La clase **StringBuilder**

- Cada objeto alberga una cadena de caracteres.
- Los objetos de esta clase se inicializan de cualquiera de las formas siguientes:

```
StringBuilder strB = new StringBuilder(10);
```

```
StringBuilder strB2 = new StringBuilder("ala");
```

- Las cadenas de los objetos **StringBuilder** se pueden ampliar, reducir y modificar mediante mensajes.
- Cuando la capacidad establecida se excede, se aumenta automáticamente.
- En versiones anteriores del JDK se usaba **StringBuffer** en lugar de **StringBuilder**.
- **StringBuffer** se diferencia de **StringBuilder** en que la primera es segura frente a tareas.

Métodos de la clase **StringBuilder**

- Métodos de consulta:

- `length()`
 - `capacity()`
 - `charAt(int pos)`

- Métodos para construir objetos **String**:

- `substring(int posini, int posfin+1)`
 - `substring(int posini)`
 - `toString()`

- Métodos para modificar objetos **StringBuilder**:

- `append(String str)`
 - `insert(int pos, String str)`
 - `setCharAt(int pos, char car)`
 - `replace(int pos1, int pos2+1, String str)`
 - `reverse()`

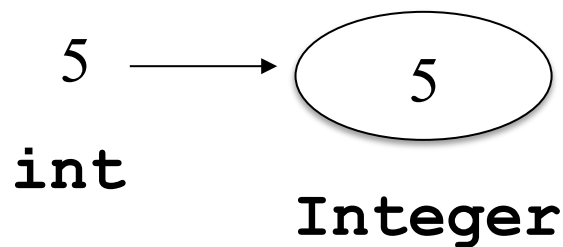
```

public class StringDemo {
    public static void main(String[] args) {
        String cadena = "Aarón es Nombre";
        int long = cadena.length();
        StringBuilder réplica = new StringBuilder(long);
        char c;
        for (int i = 0; i < long; i++) {
            c = cadena.charAt(i);
            if (c == 'A') {
                c = 'V';
            } else if (c == 'N') {
                c = 'H';
            }
            réplica.append(c)
        }
        System.out.println(réplica);
    }
}

```


Las clases envoltorios (*wrappers*)

- Supongamos que tenemos un array de tipo **Object**.
- ¿Qué podemos introducir en el array?
 - Sólo objetos. Los tipos básicos no son objetos, por lo que no pueden introducirse en ese array.
 - Para ello se utilizan los envoltorios.
 - A partir de JDK1.5 se envuelve y desenvuelve automáticamente.



Tipo básico	Envoltorio
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>

Los envoltorios numéricos

- Constructores: crean envoltorios a partir de los datos numéricos o cadenas de caracteres:

```
Integer oi = new Integer(34) ;  
Double od = new Double("34.56") ;
```

- Métodos de clase para crear números a partir de cadenas de caracteres:

```
xxxx parseInt(String)  
int i = Integer.parseInt("234") ;  
double d = Double.parseDouble("34.67") ;
```

- Métodos de clase para crear envoltorios de números a partir de cadenas de caracteres:

```
Xxxx valueOf(String)  
Integer oi = Integer.valueOf("234") ;  
Double od = Double.valueOf("34.67") ;
```

- Métodos de instancia para extraer el dato numérico del envoltorio:

```
xxxx intValue()  
int ni = oi.intValue() ;  
double nd = od.doubleValue() ;
```

Ejemplos de envoltorios numéricos

```
int a = Integer.parseInt("34");
```

```
Double d = new Double("-45.8989");    // envolver  
double dd = d.doubleValue();
```

```
double ddd = Double.parseDouble("32.56");
```

```
Long l = Long.valueOf("27.98");        // envolver
```

```
double dddd = dd + d;                  // desenvolver
```

Se produce la excepción `NumberFormatException` si algo va mal.

El envoltorio Boolean

- Los constructores crean envoltorios a partir de valores lógicos o cadenas de caracteres:

```
Boolean ob = new Boolean("false");
```

- Método de clase para crear un envoltorio lógico a partir de cadenas de caracteres:

```
Boolean valueOf(String)
```

```
Boolean ob = Boolean.valueOf("false");
```

- Métodos de instancia para extraer el valor lógico del envoltorio:

```
boolean booleanValue()
```

```
boolean b = (new Boolean("false")).booleanValue();
```

- Si el dato introducido no es lógico no produce error, sino que lo toma como **false**:

```
boolean b = (new Boolean("mal")).booleanValue();
```

El envoltorio Character

- Constructor único que crea un envoltorio a partir de un carácter:

```
Character oc = new Character('a');
```

- Métodos de instancia para extraer el dato carácter del envoltorio:

```
char charValue()
```

```
char c = oc.charValue();
```

- Métodos de clase para comprobar el tipo de los caracteres:

```
boolean isDigit(char)
```

```
boolean isLetter(char)
```

```
boolean isLowerCase(char)
```

```
boolean isUpperCase(char)
```

```
boolean isSpaceChar(char)
```

```
boolean b = Character.isLowerCase('g');
```

- Métodos de clase para convertir caracteres:

```
char toLowerCase(char)
```

```
char toUpperCase(char)
```

```
char c = Character.toUpperCase('g');
```

Envolver y desenvolver automáticamente: *boxing/unboxing* automático en JDK1.5

- El compilador realiza de forma automática la *conversión* de tipos básicos a objetos y viceversa.
- No es posible enviar un mensaje a valores de tipo básico.

```
public class Lista<E> {  
    ...  
    public void añadir(E c) {...}  
    public E elemento(int i) { ... }  
    public int tamaño() { ... }  
    ...  
}
```

ENVUELVE
DESENVUELVE

```
Lista<Double> d = new Lista<Double>();  
d.añadir(45.5);  
double r = 5.2 + d.elemento(0);  
d.añadir(d.elemento(3) + 1);
```

El paquete `java.util`

- Contiene clases de utilidad
 - Las colecciones.
 - La clase **StringTokenizer**.
 - La clase **Random**.
 - Interfaces y excepciones.
 - ... consultar la documentación.

La clase `StringTokenizer`

- Proporciona analizadores léxicos simples para cadenas de caracteres.
 - En el constructor se proporciona la cadena que queremos “tokenizar” y opcionalmente los delimitadores:

```
StringTokenizer st =  
    new StringTokenizer("La-cosa, ajena; es", " .,;:-");
```

- Por defecto, el delimitador es el espacio (`" \t\n\r\f"`).

- Su uso básico se hace con los métodos:

```
boolean hasMoreTokens()  
String nextToken()
```

- Si se intenta acceder a un token que no existe se produce una excepción `NoSuchElementException`

Ejemplo

```
import java.util.StringTokenizer;

class EjST {
    static public void main(String[] args) {
        StringTokenizer st =
            new StringTokenizer("El agua:es;buen", " :");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

El
agua
es;buen

La clase Random

- Los objetos representan variables aleatorias de distinta naturaleza:

```
Random r = new Random();
```

- Permite generar números aleatorios de diversas formas:

```
float nextFloat()
```

```
double nextDouble()
```

```
int nextInt(int n)    // 0 <= res < n
```

```
double nextGaussian()
```

```
...
```

- Consultar la documentación para información adicional.

Clases ordenables

- Una clase puede especificar una relación de orden por medio de:
 - la interfaz **Comparable**<T> (*orden natural*)
 - la interfaz **Comparator**<T> (*orden total*)
- Sólo es posible definir un orden natural, aunque pueden especificarse varios órdenes totales.
 - El orden natural se define en la propia clase.

```
public class Persona implements Comparable<Persona> {  
    ...  
}
```

- Cada uno de los órdenes totales puede implementarse en una clase diferente.

```
public class SatPersona implements Comparator<Persona> {  
    ...  
}  
public class OrdPersona implements Comparator<Persona> {  
    ...  
}
```

- Si se intentan comparar dos objetos no comparables se lanza una excepción **ClassCastException**.

La interfaz Comparable<T>

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

- *Orden natural* para una clase. $\left\{ \begin{array}{llll} \text{negativo} & \text{si receptor} & \text{menor} & \text{que o} \\ \text{cero} & \text{si receptor} & \text{igual} & \text{que o} \\ \text{positivo} & \text{si receptor} & \text{mayor} & \text{que o} \end{array} \right.$
- **compareTo()** *no debe* entrar en contradicción con **equals()**.
- Muchas de las clases estándares en la API de Java implementan esta interfaz:

Clase	Orden natural
Byte, Long, Integer, Short, Double y Float	numérico
Character	numérico (sin signo)
String	lexicográfico
Date	cronológico
...	

La interfaz **Comparator<T>**

- Las clases que necesiten una relación de orden distinta del orden natural han de utilizar clases “satélite” que implementen la interfaz **Comparator<T>**.

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

{	negativo	si o1 menor que o2
	cero	si o1 igual que o2
	positivo	si o1 mayor que o2

- compare()** *no debe* entrar en contradicción con **equals()**.

Ejemplo: clase Persona

```
import java.util.*;
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public String nombre() {
        return nombre;
    }
    public int edad() {
        return edad;
    }
    public boolean equals(Object obj) {
        return obj instanceof Persona &&
            nombre.equals(((Persona) obj).nombre) &&
            edad == ((Persona) obj).edad;
    }
    public int hashCode() {
        return
            (nombre.hashCode() + (new Integer(edad)).hashCode()) / 2;
    }
    ...
}
```

Persona implementa Comparable<Persona>

...

```
// Se comparan por edad, y a igualdad de edad, por nombres
```

```
public int compareTo(Persona p) {  
    int resultado = 0;  
    if (edad == p.edad) {  
        resultado = nombre.compareTo(p.nombre);  
    } else {  
        resultado = (new Integer(edad)).compareTo(p.edad);  
    }  
    return resultado;  
}  
}
```

OrdenPersona implementa Comparator<Persona>

```
import java.util.*;

public class OrdenPersona implements Comparator<Persona> {

    // Se comparan por nombres, y a igualdad de nombres, por edad

    public int compare(Persona p1, Persona p2) {
        int resultado = p1.nombre().compareTo(p2.nombre());
        if (resultado == 0) {

            resultado = (new Integer(p1.edad())).compareTo(p2.edad());
        }
        return resultado;
    }
}
```

La clase **Persona** debe disponer de los métodos **edad()** y **nombre()**.

La clase Arrays I

- Contiene métodos estáticos que implementan algoritmos sobre arrays de elementos de tipo básico u Object.

Tipo representa un tipo básico u Object




```
static int binarySearch(Tipo [] ar, Tipo key);
```

- Devuelve el índice de la posición del elemento key en ar.
- Devuelve -pi-1 si key no está, donde pi es la posición en la que se debería insertar para mantener ar ordenado.

- También existe una versión en la que se puede proporcionar un objeto Comparator:

```
static <T> int binarySearch(T[] ar, T key,  
                           Comparator<? super T> c);
```



Parámetro genérico (se verá más adelante)

La clase Arrays II

```
static void fill(Tipo [] ar, Tipo key);
```

- Asigna el valor key a cada elemento de ar.

```
static void sort(Tipo[] ar)
```

- Ordena el array ar según el orden natural de los elementos.

- También existe una versión en la que se puede proporcionar un objeto Comparator:

```
static <T> void sort(T[] ar, Comparator<? super T> c);
```

- El método que devuelve la representación textual de un array:

```
static String toString(Tipo[] ar);
```