INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Curso online: **PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA (TELEFORMACIÓN - ON LINE)**

Módulo 4: UTILIDADES Y ESTRUCTURAS DE DATOS

Autores

Jorge Molinero Muñoz

Miguel Sallent Sánchez

ÍNDICE

1. Introducción

En este tema se describirán diversas utilidades que se encuentran en el paquete java.util, como analizadores léxicos y las estructuras de datos pilas, listas, colas, etc. Previamente se describirá el API del lenguaje.

2. La API de Java

2.1. ¿Qué es la API?

La **API de Java** (o librería de Java) está formada por una serie de paquetes que son distribuidos junto al JDK en forma de librerías de clases. Todas estas clases están disponibles para el programador, de forma que antes de desarrollar una nueva utilidad o estructura de datos debemos buscar si lo que queremos ya existe en la API de Java.

Las ventajas que proporciona tener una serie de paquetes integrados dentro de la API para ofrecérselos al programador son:

- **Se programa con una mayor facilidad**, ya que se cuenta con una gran cantidad de clases diseñadas, que no hay que escribir en cada desarrollo. Por este motivo es más difícil cometer errores programando en Java que en otros lenguajes.
- **Los programas son mucho más pequeños y legibles.**
- **Se fomenta la reutilización de las clases**, que es uno de los baluartes de la programación orientada a objetos.

Para saber qué podemos encontrar en la API conviene tener una idea general de la utilidad de cada uno de los paquetes. En este tema se introduce la API del JDK 8, que consta de 14 paquetes de desarrollo, 21 paquetes de extensión (comienzan por el prefijo javax) y 4 paquetes org, los cuales son especificados por el Java Community Process.

El **Java Community Process**, es un procedimiento, en el cual están involucrados empresas y fabricantes relacionados con la tecnología JAVA, orientado al desarrollo de especificaciones técnicas y estándares para la plataforma JAVA. A estas especificaciones se les denomina **Java Specification Request** (JSR).

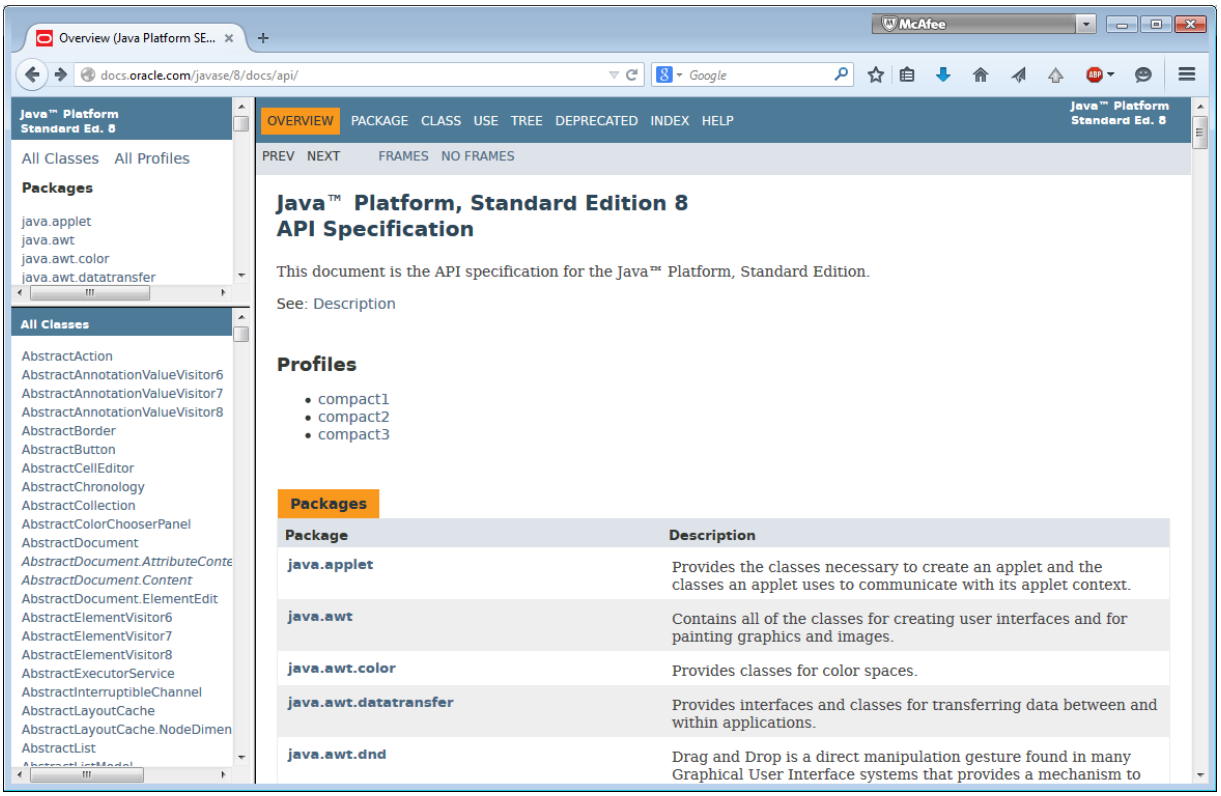
Los más utilizados son los paquetes de desarrollo:

PAQUETE	DESCRIPCIÓN
java.applet	Applets
java.awt	Interfaz de usuario y dibujo de gráficos e imágenes
java.beans	Java Beans
java.io	Entrada / Salida
java.lang	Clases básicas del lenguaje
java.math	Utilidades aritméticas
java.net	Conexiones a través de redes
java.nio	Utilidades para entrada / salida intensiva a bajo nivel
java.rmi	Remote Method Invocation (llamada remota a métodos)
java.security	Seguridad
java.sql	SQL
java.text	Utilidades para manejar texto
java.time	Utilidades para fechas y cantidades de tiempo.
java.util	Utilidades diversas: colecciones, eventos, fechas, internacionalización, etc.

Debemos tener en cuenta que esta API básica de Java o Core API forma parte del propio lenguaje, ya que no se pueden modificar o sustituir los elementos existentes en ella, y estarán siempre disponibles para cualquier desarrollador.

2.2. Ayuda de la API

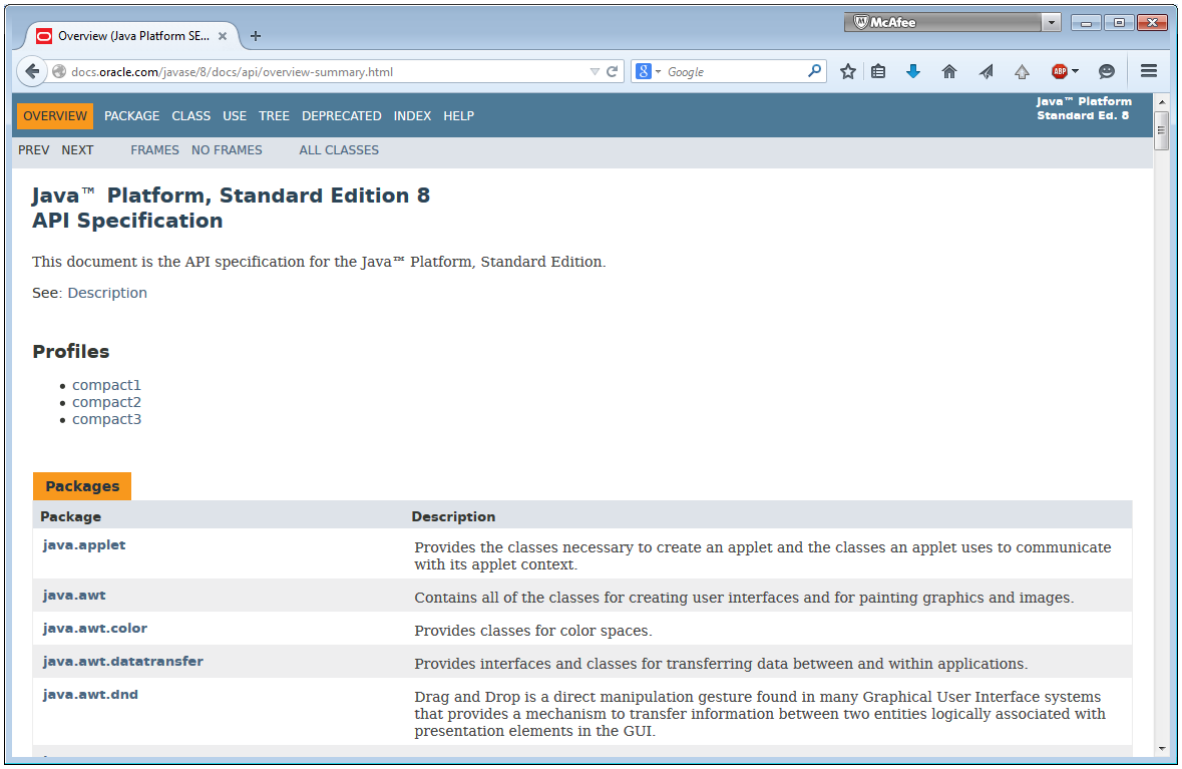
La documentación de la API se encuentra en la dirección http://docs.oracle.com/javase/8/docs/api/



La documentación de la API ofrece una página con *tres marcos*: el marco superior izquierdo muestra todos los paquetes de los que consta la *API de Java*, pulsando en cualquiera de ellos, se muestra en el marco inferior izquierdo las clases, interfaces y excepciones de las que consta y, finalmente, pulsando en cualquiera de ellos se muestra su descripción en el marco derecho.

Aquí encontraremos toda la información necesaria para utilizar las clases que Java nos proporciona. Para cada clase se explican todos sus métodos, indicando detalladamente su funcionalidad, qué representa cada uno de sus argumentos, el resultado devuelto y los posibles errores.

También se puede visualizar esta documentación sin frames:



Desde cualquiera de los modos de consulta de la documentación puede accederse a los demás mediante los enlaces existentes en la parte superior de los documentos.

Al acceder a cada paquete de la API, observaremos que están organizados en una serie de apartados relativos a los elementos que los componen, cuyo orden es:

- **Interfaces**
- **Clases**
- **Excepciones**

Cada uno de estos elementos muestra información sobre los siguientes puntos:

- **Clases de las que hereda e interfaces que implementa**
- **Atributos**
- **Constructores**
- **Métodos**

Búsqueda en la documentación de la API

Analicemos cómo *buscar y obtener información* de algún elemento de la API. La API puede ser utilizada para obtener información general buscando desde el índice que contiene todos los paquetes y profundizando en la jerarquía ya comentada, pero otra situación bastante usual es la búsqueda de información de elementos que encontramos en código ya programado.

Para explicar este tipo de búsqueda vamos a emplear un ejemplo. A partir de una sentencia muy común en las aplicaciones escritas en Java (quizás es la primera con la que se encuentran los desarrolladores que comienzan en este lenguaje), vamos a describir cómo obtener la mayor información posible.

System.out.println("...mensaje...");

Para iniciar la búsqueda, que deberá ir de izquierda a derecha (de lo general a lo particular) en la sentencia:

- En primer lugar, se debe acceder al listado de todas las clases que contiene todos los elementos de la API, y buscar el elemento *System*. En este caso, se localizaría la clase *System*, perteneciente al paquete *java.lang*.
- En el caso de no existir ninguna importación, como podría ser el caso del ejemplo, habrá que pensar que la clase pertenece al paquete por defecto, que recordemos es *java.lang*.
- Una vez localizada la clase o el paquete al que ésta pertenece (y accediendo desde el mismo a la clase), retomaremos la búsqueda del resto de elementos de la sentencia.
- Lo siguiente que aparece es el elemento *out*, mediante la notación *punto*. Debido a que tras este elemento no aparecen paréntesis, lo que indicaría que corresponde a un método, debemos interpretar que se trata de un atributo de la clase. Si buscamos en los atributos de *System*, veremos que contiene tres atributos, correspondientes a la *entrada (in), salida (out) y error estándar (err)*.
- Lo primero que observamos es que estos tres atributos son estáticos (*static*). Esto quiere decir que son atributos de la clase, no de sus instancias, y por tanto podemos acceder a ellos directamente sobre la clase (*System.out*) sin necesidad de crear una instancia de ella (no necesitamos hacer *new System()*).
- Si nos introducimos en la descripción del atributo *out*, observaremos que pertenece a la clase *PrintStream*, de modo que para continuar nuestra búsqueda de la sentencia deberemos examinar dicha clase, ya que a continuación se identifica un método de la misma (al ir seguido de paréntesis), que en este caso es *println*.

- Como dentro de esta clase existen varios métodos con ese nombre, nos fijaremos en aquél que recibe un único parámetro de tipo *String*. Al examinarlo, obtendremos toda la información sobre su uso.

2.3. JAVADOC

La Ayuda de la API de Java tiene la particularidad de que es un HTML que se genera automáticamente a partir del código fuente de las clases. Para ello en el código hay unos comentarios con cierto formato, y posteriormente mediante la utilidad Javadoc se genera toda la documentación.

Esto también lo podemos utilizar con las clases que desarrollamos nosotros, de forma que podemos generar automáticamente una documentación similar a la del API de Java pero para nuestras clases.

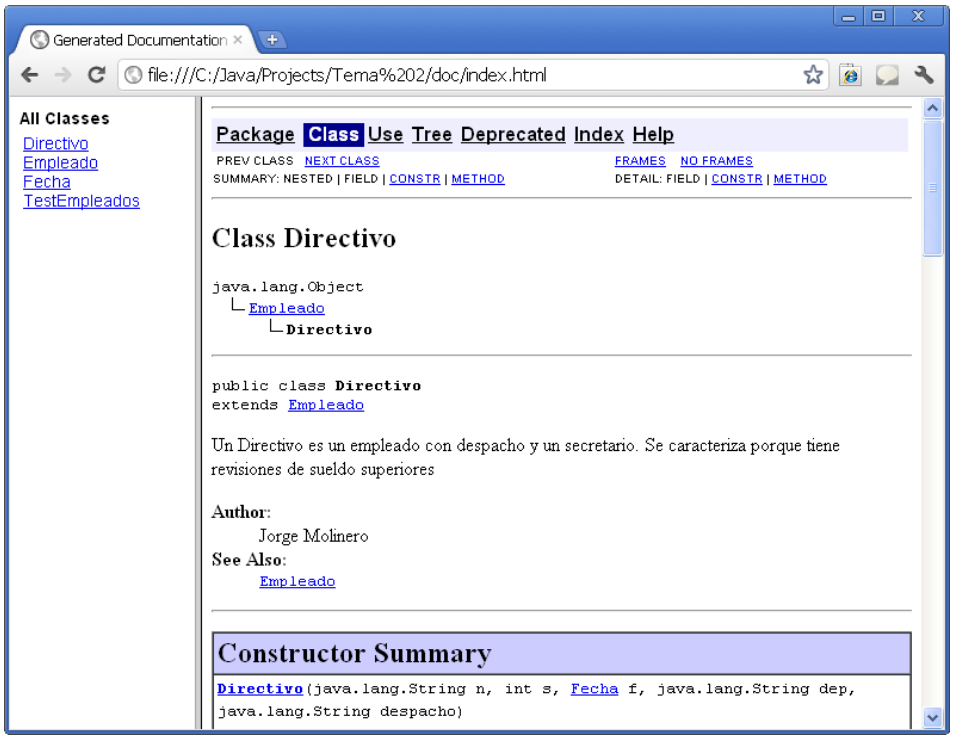
Por ejemplo:

```
/**
 * Un Directivo es un empleado con despacho y un secretario.
 * Se caracteriza porque tiene revisiones de sueldo superiores
 * @see Empleado
 * @author Jorge Molinero
 */
public class Directivo extends Empleado {

/**
 * Nombre del secretario con el que trabaja el directivo
 */
 private String secretario;
 …
/**
 * Método que permite subir el sueldo
 * @param porcentaje El tanto por ciento que se le subirá
 * @param prod La productividad que se le sumará
 */
 public void subirSueldo(double porcentaje, int prod){
…..
```

Una vez insertados lo comentarios generamos el html usando la herramienta Javadoc:

```
C:\Program Files\Java\jdk1.8.0\bin>javadoc -d C:\API C:\Tema2\*.java
```



Los comentarios javadoc comienzan por /**, en lugar de por /*

Muchas organizaciones obligan a sus programadores a incluir los comentarios javadoc en todas las clases y métodos, ya que es una manera de asegurar que el código esté documentado, lo que facilita su reutilización en diferentes proyectos.

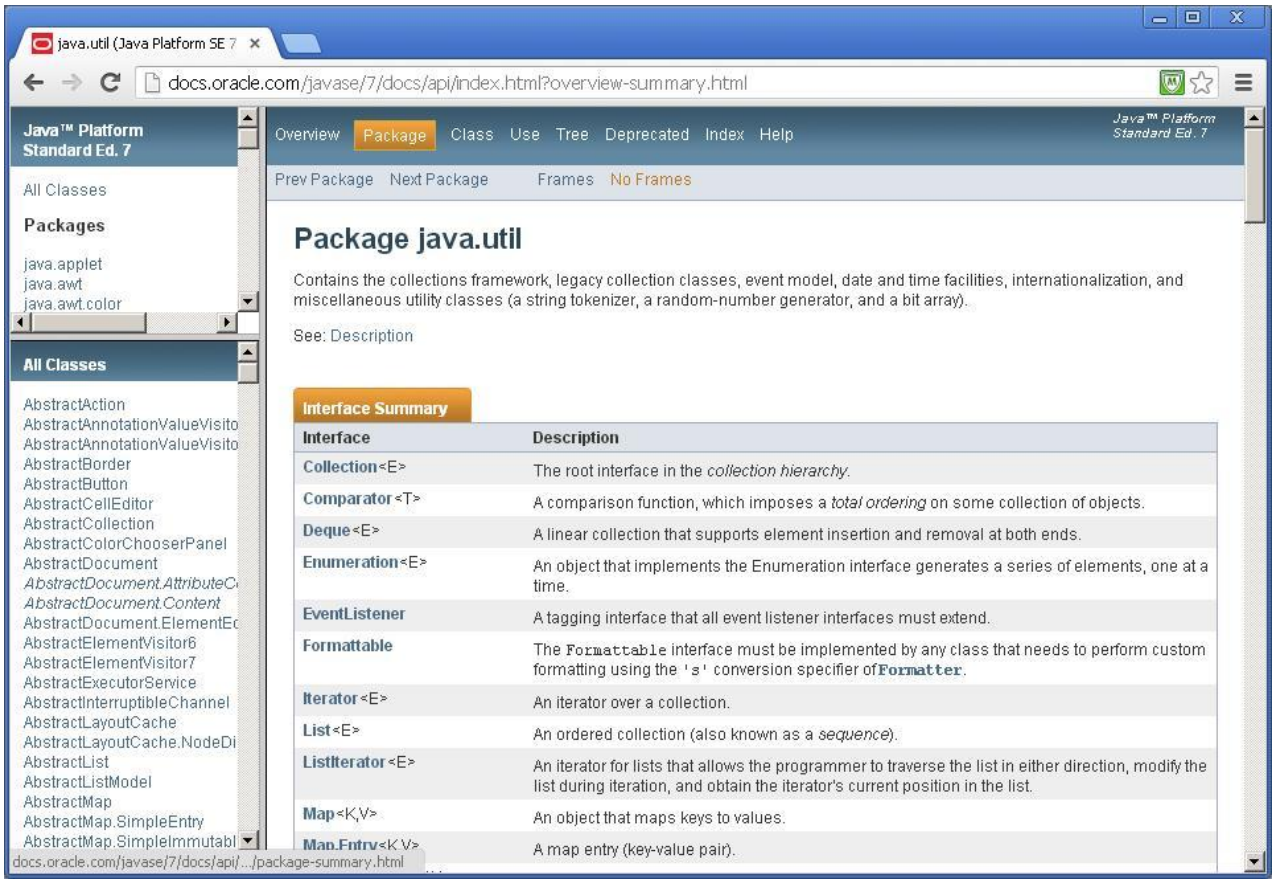
3. Utilidades

3.1. Introducción

Dos de las principales características del diseño orientado a objetos son su flexibilidad y las facilidades que ofrece de cara a la reutilización. Por ello, resulta especialmente importante aprovechar todas las facilidades que proporciona el interfaz de programación (API) de que dispone el lenguaje Java.

El paquete ***java.util***, incluye numerosas clases de utilidades y, más importante, clases que implementan las estructuras de datos más frecuentemente utilizadas. Entre estas utilidades se cuenta con clases para tratamiento de fechas (*Calendar*, *Date*, *GregorianCalendar*, *SimpleTimeZone y TimeZone*), clases para la realización de analizadores léxicos (*StringTokenizer*), clases para la generación básica de números aleatorios (*Random*), etc.

Entre las estructuras de datos se encuentran los vectores dinámicos (*Vector*), tablas hash (*Hashtable*), pilas (*Stack*), enumeraciones (*Enumeration*), etc.



A continuación veremos algunas de las utilidades de uso más frecuente.

3.2. Textos

La base del tratamiento de textos en Java es la clase String. Algunos de los métodos usados más frecuentemente son:

- `int length()`: devuelve el número de caracteres de la cadena.
- `char charAt(int index)`: devuelve el carácter en la posición indicada. El primer carácter tiene la posición 0 y el último la posición length()-1.
- `String substring (int beginIndex, int endIndex)`: devuelve una subcadena que comienza en beginIndex y termina en endIndex -1.

- `int indexOf(String s)`: busca una cadena de texto dentro de otra y devuelve su posición, o -1 si no la encuentra.
- `String toLowerCase() / toUpperCase()`: convierte a minúsculas / mayúsculas.
- `String trim()`: elimina espacios en blanco al principio y final de la cadena

La clase String proporciona muchos otros métodos, en función de nuestras necesidades consultaremos la API para seleccionar el más adecuado.

Es importante resaltar que la clase String es **inmutable**. Esto quiere decir que una vez que hemos creado un objeto String ya no se puede modificar. Por tanto todas las operaciones de manipulación como por ejemplo toLowerCase() o trim(), **devuelven una nueva cadena** que es la transformada, dejando la original intacta.

```
String a = "Hola";
String b = a.toUpperCase(); //a es "Hola", b es "HOLA".
```

También podríamos hacer lo siguiente:

```
String a = "Hola";
a = a.toUpperCase();
```

En este caso se crea una nueva cadena que es la transformada, y se asigna a la misma variable. La cadena original, puesto que ya no estaría referenciada por ninguna variable, pasaría a disposición del recolector de basura siendo liberada su memoria. Es decir, aunque aparentemente hemos modificado la cadena original, esto no es así, sino que hemos creado una nueva y la hemos asignado a la misma variable.

Otro ejemplo en el que afecta la inmutabilidad es la concatenación. Por ejemplo:

```
String a = "Hola";
a = a + " alumnos";
```

En este caso primero se crea una cadena "Hola", después otra cadena " alumnos" y finalmente una tercera cadena que es la concatenación de ambas y se asigna a la misma variable. Nuevamente parece que hemos modificado la primera cadena, pero no es así, sino que se crea una nueva cadena y se asigna a la variable original.

Este proceso como se puede observar no es muy eficiente, por ejemplo si estamos construyendo una cadena de texto mediante un número elevado de concatenaciones, cada vez creará una nueva cadena y copiará todos los caracteres. Cuando el rendimiento sea crítico podemos usar la clase StringBuilder, que es similar a la clase String pero sí es mutable.

3.3. Números

Cuando se trabaja con números normalmente se usan los tipos primitivos. Por ejemplo:

```
int i = 500;
float gpa = 3.65f;
byte mask = 0xff;
```

Sin embargo en algunas ocasiones nos interesará utilizar objetos en lugar de tipos primitivos. Para ello Java proporciona una serie de clases, llamadas Envoltorios (*Wrappers*): Integer, Float, Double, Byte, Short, Long, etc.

Hay varios casos en los que nos podría interesar utilizar objetos en lugar de tipos primitivos:

- Como argumentos de un método que espera recibir un objeto. Un ejemplo típico es una colección de números.
- Para usar constantes definidas en esas clases. Por ejemplo Integer.MAX_VALUE nos da el valor máximo que puede tomar un entero.
- Para convertir a cadenas de texto, a otros tipos primitivos o entre sistemas numéricos (hexadecimal, octal, decimal, etc).

A continuación se muestran algunos ejemplos típicos de operaciones con números:

- Convertir una cadena de texto que representa un número al tipo primitivo correspondiente

```
String txt = "27";
int val = Integer.parseInt(txt);
```

- La inversa sería convertir un número tipo primitivo a texto:

```
int val = 27;
String txt = String.valueOf(val);
```

- Podemos hacer lo mismo pero en vez de con tipos primitivos con el objeto numérico correspondiente (*wrapper*):

```
String txt = "27";
Integer val = new Integer(txt);
```

- Y su inversa:

```
Integer val = new Integer(27);
String txt = val.toString();
```

- Obtener la representación hexadecimal de un número entero

```
int val = 27;
String txt = Integer.toHexString(val); //txt es "1b"
```

- Y su inversa, obtener el número que corresponde a su representación hexadecimal:

```
String txt = "1b";
int val = Integer.parseInt(txt,16);
```

Formatear números

Para números con decimales (float, double), podemos usar la clase java.text.DecimalFormat para controlar el formato, por ejemplo para usar la coma en lugar del punto como separador decimal, indicar el número de decimales, etc.

El siguiente ejemplo formatearía redondeando a dos decimales

```
DecimalFormat myFormatter = new DecimalFormat("###,###,###.00");
String formatted = myFormatter.format(1265.348);
System.out.println(formatted);
```

El resultado es "1.265,35".

En la documentación de esta clase está explicado lo que representa cada símbolo.

Como separador decimal y separador de miles utilizará el definido para el idioma por defecto. En el ejemplo ha utilizado el del español, pero podemos cambiarlo. Por ejemplo para usar el formato inglés podríamos hacerlo de la siguiente manera:

```
DecimalFormatSymbols symbols =
      DecimalFormatSymbols.getInstance(Locale.forLanguageTag("en-US"));
DecimalFormat myFormatter = new DecimalFormat("###,###,###.00",symbols);
String formatted = myFormatter.format(1265.348);
System.out.println(formatted);
```

En este caso el resultado sería "1,265.35".

Esta misma clase nos sirve para la operación inversa, es decir, convertir de texto a número:

```
String txt = "1.265,35";
DecimalFormat myFormatter = new DecimalFormat("###,###,###.00");
try {
      Number val = myFormatter.parse(txt);
      System.out.println(val);
} catch (ParseException e) {
      System.out.println("El número no tiene un formato correcto");
}
```

La clase Math

Para operaciones aritméticas complejas disponemos de la clase java.lang.Math. La mayoría de sus métodos son estáticos, es decir, no hay que crear un objeto Math sino que se aplican sobre la propia clase. Ejemplos:

- redondeos, máximos, mínimos, etc:

```
System.out.println(Math.abs(a)); //valor absoluto
System.out.println(Math.floor(b)); //truncar los decimales
System.out.println(Math.round(b)); //redondear los decimales
System.out.println(Math.max(a,b)); //el máximo de dos números
```

- operaciones trigonométricas:

```
System.out.println(Math.sin(Math.PI/2)); //seno
System.out.println(Math.asin(1)); //arcoseno
System.out.println(Math.toRadians(180)); //radianes
System.out.println(Math.toDegrees(Math.PI)); //grados
```

- operaciones exponenciales y logarítmicas:

```
System.out.println(Math.pow(2,10)); //potencia
System.out.println(Math.sqrt(64)); //raíz cuadrada
System.out.println(Math.exp(2)); //potencia del número e
System.out.println(Math.log10(1000)); //logaritmo decimal
System.out.println(Math.log(548785)); //logaritmo natural
```

3.4. Fechas

A continuación veremos algunos ejemplos de operaciones con fechas.

Para convertir una fecha en texto podemos usar la clase java.text.SimpleDateFormat. El funcionamiento es similar al que hemos visto para formatear números: creamos un objeto que contiene la información sobre cómo formatear las fechas, y después lo utilizamos tanto para convertir de fecha a texto como su inversa. Para obtener la fecha actual nos ayudamos de la clase java.util.Calendar.

Por ejemplo, para pasar de fecha a texto:

```
Calendar hoy = Calendar.getInstance();

SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

System.out.println(formatter.format(hoy.getTime()));
```

Y su inversa, de texto a fecha:

```
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

try {
   Date fecha = formatter.parse("17/06/2014 17:03:05");
   System.out.println(fecha);
} catch (ParseException e) {
   System.out.println("Formato de fecha incorrecto");
}
```

Obsérvese que se utiliza MM para meses y mm para minutos (no confundirlos). También obsérvese que para los años se usa yyyy (de *year*), no aaaa (de *año*), y que las horas en formato 24 horas son HH en mayúsculas (hh en minúsculas son las horas en formato 12 horas).

Operaciones aritméticas con fechas

También es muy habitual realizar operaciones aritméticas con fechas, por ejemplo, añadir un día, o un mes, extraer el año, etc. Para todo ello utilizamos la clase java.util.Calendar. Ejemplos:

```
Calendar fecha = Calendar.getInstance();  //fecha actual

fecha.add(Calendar.MONTH, 2); //añado dos meses

fecha.set(Calendar.DAY_OF_MONTH,1); //fijo el día al primero de mes

int año = fecha.get(Calendar.YEAR); //obtengo el año

System.out.println(fecha.getTime());
```

A partir de Java 8 está disponible un nuevo paquete para el manejo de fechas: java.time. Sin embargo la clase java.util.Calendar sigue siendo de uso generalizado por lo que conviene conocerla.

4. Colecciones

4.1. Introducción

Otra de las utilidades que ofrece java son las colecciones. Una colección no es más que un objeto que agrupa múltiples elementos en una unidad, como podría ser por ejemplo una lista de registros de una base de datos.

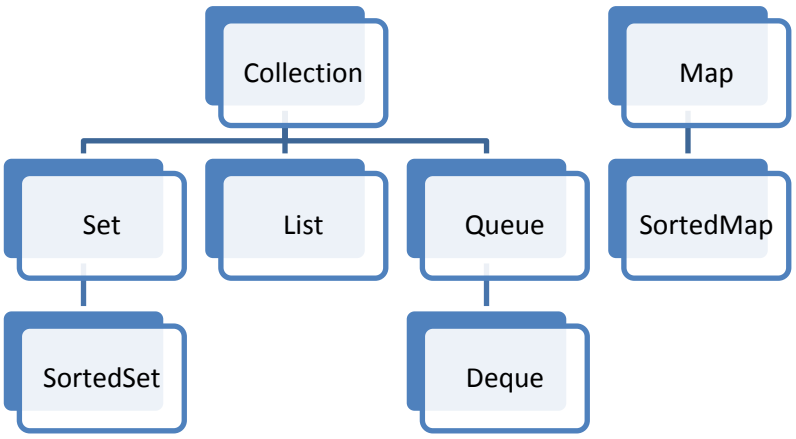
Las colecciones se usan de forma generalizada por lo que es imprescindible conocer bien su funcionamiento. Dentro del paquete java.util encontramos lo que se conoce como "*collections framework*", formado por

- **Interfaces**: son tipos abstractos que nos permiten operar con colecciones sin tener que conocer los detalles de su implementación.
- **Implementaciones**: son clases que implementan los interfaces anteriores
- **Algoritmos**: utilidades que nos permiten realizar operaciones habituales como búsquedas u ordenaciones.

A continuación veremos los más habituales

4.2. Interfaces

Los interfaces permiten manipular colecciones con independencia de los detalles de su implementación. Como se puede ver en la figura siguiente los interfaces forman una jerarquía:



- Collection: representa un grupo de objetos, llamados *elementos.* Es la base de la jerarquía, y especifica ciertas operaciones que toda colección debe tener, como añadir un elemento, borrar un elemento, comprobar si la colección contiene un elemento, etc.
- List: una lista es un conjunto ordenado de elementos. Puede contener elementos repetidos.
- Set: es un conjunto de elementos que no contiene duplicados. Puede estar ordenado o no.
- SortedSet: es un conjunto ordenado de elementos que no contiene duplicados.
- Queue: diseñada para almacenar elementos antes de procesarlos. El uso típico es usarlo como una cola FIFO (first in - first out): el primer elemento en entrar en la cola es el primer elemento en salir de ella.
- Deque: similar a Queue pero permite acceder a los elementos por ambos extremos. Permite implementar tanto colas FIFO (first in - first out) como pilas LIFO (last in - first out).

- Map: es un objeto que almacena parejas clave – valor. Para cada clave hay un único valor asociado, y no puede haber claves repetidas. Las operaciones básicas son insertar y recuperar un valor para una determinada clave.
- SortedMap: es un Map que mantiene las claves ordenadas.

4.3. Genéricos

Todos estos interfaces han sido diseñados como "genéricos". Para entender el concepto de objetos genéricos veamos primero cómo se recorrían colecciones en versiones previas de Java donde no había genéricos. El ejemplo más sencillo es crear una lista, añadir ciertos elementos, y después recorrerla:

```
ArrayList list = new ArrayList();
list.add(new Integer(2));
list.add(new Integer(4));

Iterator listIterator = list.iterator();
while (listIterator.hasNext())
{
     Integer obj = (Integer) listIterator.next();
     System.out.println("elemento: "+ obj);
}
```

Vemos que se creaba una lista de objetos que podían ser de cualquier clase (Object). Al extraer los objetos mediante el método .next(), lo que se devuelve es un Object, por lo que nos vemos obligados a realizar un cast a Integer. Es decir, nosotros sabemos que lo devolverá es un Integer pero el compilador no lo sabe, porque podríamos haber metido un objeto de cualquier tipo. Esto era fuente de errores, ya que si el tipo especificado en el cast era incorrecto, daba un error en tiempo de ejecución.

A partir de la introducción de los tipos genéricos, cuando creamos una lista podemos indicar que los objetos de esa lista van a ser todos de un determinado tipo. De esta manera se comprueba *en tiempo de compilación* que los tipos son correctos. Quedaría así:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(2));
list.add(new Integer(4));

Iterator<Integer> listIterator = list.iterator();
while (listIterator.hasNext())
{
    Integer obj = listIterator.next();
    System.out.println("elemento: "+ obj);
}
```

Cuando creamos la lista indicamos la clase de objetos que vamos a insertar. Observamos que ya no es necesario hacer el cast al extraer el elemento, y además si intentáramos insertar un objeto que no fuera un Integer daría un error de compilación.

El uso de los tipos genéricos no es exclusivo de las colecciones, podemos definir nuestras propias clases y configurarlas como genéricas cuando veamos que es conveniente.

4.4. Implementaciones

A continuación veremos algunas de las implementaciones de los interfaces anteriores.

Implementaciones de List

Como hemos indicado List es un interfaz que solamente especifica las operaciones que debe tener toda lista. Para crear listas necesitaremos utilizar alguna clase que implemente este interfaz. Las implementaciones más utilizadas son ArrayList y Vector. Su principal diferencia es que Vector permite la modificación concurrente, es decir, varios hilos de ejecución concurrente que accedan a una misma lista para modificarla. ArrayList por el contrario no lo permite. Por tanto en general utilizaremos ArrayList a no ser que preveamos que la lista va a ser accedida por hilos concurrentes, ya que el rendimiento será mejor.

Podemos ver las operaciones más habituales de una lista en el siguiente ejemplo, en el que rellenaremos una lista y la recorreremos de varias formas posibles:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(2));
list.add(new Integer(4));

//recorremos usando el índice de posición:
for (int i=0;i<list.size();i++)
{
	Integer obj = list.get(i);
	System.out.println("elemento: "+ obj);
}

//recorremos usando el iterator:
Iterator<Integer> listIterator = list.iterator();
while (listIterator.hasNext())
{
	Integer obj = listIterator.next();
	System.out.println("elemento: "+ obj);
}

//recorremos usando una forma mejorada del bucle for:
for (Integer obj: list)
{
	System.out.println("elemento: "+ obj);
```

```
}
```

En general suele ser recomendable recorrer las colecciones usando el Iterator o la forma mejorada del bucle *for*, ya que está diseñado para ser más eficiente, proporcionando tiempos de respuesta menores que el acceso a través del índice de posición.

Implementaciones de Set. El método *hashCode*. Los interfaces *Comparable* y *Comparator*.

Como hemos comentado anteriormente, el interfaz Set representa un conjunto de elementos sin duplicados. Es decir, si intentamos añadir un elemento a la colección que es igual a uno que ya está insertado, el elemento no se añadirá.

Las dos implementaciones más utilizadas son HashSet y TreeSet.

HashSet utiliza una tabla hash internamente para almacenar los elementos. Para ello calcula un código hash (un número entero) para cada elemento que se va a insertar, invocando el método *hashCode()* de cada elemento antes de hacer una inserción. Este número le indica la posición del elemento dentro de su estructura interna, por lo que la búsqueda es muy rápida, y el tiempo de búsqueda no depende del tamaño de la colección.

El método *hashCode()* es un método definido en la clase Object, por tanto todos los objetos disponen de este método. Aunque la clase Object proporciona una implementación por defecto de este método, las clases suelen sobrescribirlo para proporcionar una implementación correcta. Este método debe cumplir que si dos objetos son iguales de acuerdo a su método *equals*, entonces deben devolver el mismo código hash.

Por tanto, si tenemos una serie de objetos cuyo tipo es de una clase que hemos desarrollado nosotros, y vamos a utilizar un HashSet para almacenar dichos objetos, debemos asegurarnos de que el método hashCode() está sobrescrito en la definición de la clase para que el funcionamiento sea correcto.

La clase TreeSet se diferencia de HashSet en que mantiene la lista ordenada, para lo cual utiliza internamente una estructura en árbol. Como desventaja, la operación de inserción de elementos es más lenta, porque es más costoso buscar la posición en el árbol que calcular el hashCode. Por tanto cuando no necesitemos que la colección esté ordenada utilizaremos HashSet ya que es más rápido.

Hay que señalar que el orden dentro del TreeSet no se trata del orden en el que los elementos son insertados (como ocurre con las listas), sino del "orden natural" de los elementos. Una clase decimos que es "ordenable" cuando implementa el interfaz Comparable. Este interfaz define un único método, *compareTo*, que debe devolver un número negativo, cero o positivo en función de si el objeto es menor, igual o mayor respectivamente que el recibido como argumento, según su orden *natural*. Cada clase determina lo que según ella es el orden natural. Por ejemplo, para la clase Integer es el orden numérico, para la clase String es el orden alfabético, etc.

Para entender esto a continuación veremos un ejemplo en el que creamos una clase Persona, que tiene varios atributos, siendo uno de ellos la edad. Decidimos que el orden natural de las Personas es su edad, así que implementamos el método *compareTo* de acuerdo a ello. Después implementaremos el método *main* para ejecutar un ejemplo de prueba: crearemos un TreeSet, insertaremos una serie de elementos desordenados y lo recorreremos. Veremos que los elementos salen ordenados correctamente según la edad. También insertaremos varios elementos iguales, y veremos que al final no hay duplicados.

```
import java.util.Iterator;
import java.util.TreeSet;

public class Persona implements Comparable<Persona> {

	private String nombre;
	private int edad;

	public Persona(String nombre, int edad) {
		this.nombre = nombre;
		this.edad = edad;
	}

	@Override
	public int compareTo(Persona otraPersona) {
		return (this.edad - otraPersona.edad);
	}

	@Override
	public String toString() {
		return "{" + nombre + "," + edad + "}";
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Persona other = (Persona) obj;
		if (edad != other.edad)
			return false;
		if (nombre == null) {
			if (other.nombre != null)
				return false;
		} else if (!nombre.equals(other.nombre))
			return false;
		return true;
	}

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

```
	{

		TreeSet<Persona> set = new TreeSet<Persona>();

		set.add(new Persona("Jorge",30));
		set.add(new Persona("Carlos",48));
		set.add(new Persona("Miguel",29));
		set.add(new Persona("Carlos",48));

		Iterator<Persona> iterator = set.iterator();
		while (iterator.hasNext())
		{
			Persona persona = iterator.next();
			System.out.println(persona);
		}
	}
}
```

El resultado es:

```
	{Miguel,29}
	{Jorge,30}
	{Carlos,48}
```

Vemos que al haber utilizado un TreeSet nos salen ordenados por edad y además Carlos no aparece duplicado.

También podemos decidir que queremos utilizar otro orden que no sea el natural, por ejemplo decidimos ordenar alfabéticamente por el nombre, pero sin cambiar el orden natural que seguirá siendo la edad. En este caso al crear el TreeSet le pasamos como argumento un comparador que será el encargado de decidir el orden.

La sentencia en la que creamos el TreeSet quedaría así:

```
TreeSet<Persona> set = new TreeSet<Persona>(new Comparator<Persona>(){

		@Override
		public int compare(Persona arg0, Persona arg1) {
			return arg0.nombre.compareToIgnoreCase(arg1.nombre);
		}

	});
```

El resultado será:

```
	{Carlos,48}
	{Jorge,30}
	{Miguel,29}
```

Obsérvese que podríamos haber creado una clase aparte que implementara el interfaz Comparator y después crear un objeto que instanciara esa clase para pasárselo como argumento al constructor del TreeSet. En vez de ello hemos utilizado una sintaxis especial que nos permite definir una clase e instanciarla todo a la vez. Esta sintaxis se llama "clases anónimas" (no hemos necesitado darle un nombre a la clase). Para utilizarla, simplemente después del constructor incluimos entre llaves la definición de la clase.

Además hemos aprovechado que el orden natural de la clase String es el orden alfabético, por tanto podemos utilizar el método *compareTo* del atributo *nombre* para la comparación. Y si usamos *compareToIgnoreCase* ordenaremos alfabéticamente sin distinguir mayúsculas de minúsculas.

No hay que confundir los interfaces *Comparable* y *Comparator*. *Comparable* debe ser implementado por los propios objetos ordenables, y se comparan a sí mismos con otro objeto según su orden natural, como veíamos en el ejemplo:

```
@Override
public int compareTo(Persona otraPersona) {
	return (this.edad - otraPersona.edad);
}
```

Sin embargo *Comparator* compara otros dos objetos entre sí:

```
@Override
public int compare(Persona arg0, Persona arg1) {
	return arg0.nombre.compareToIgnoreCase(arg1.nombre);
}
```

Utilizaremos los objetos *Comparator* cuando queramos ordenar por un criterio que no sea el orden natural.

Implementaciones de Map

El interfaz Map nos permite almacenar un conjunto de parejas clave – valor, de forma similar a un diccionario. Los objetos Map se utilizan frecuentemente para almacenar parámetros de configuración o en operaciones de agrupamiento.

Las operaciones básicas son:

- para una clave, insertar un valor: *map.put(clave,valor)*
- para una clave, obtener el valor: *map.get(clave)*

Pensemos por ejemplo en una serie de propiedades de configuración de una aplicación. Cada propiedad tendrá un nombre y un valor. Por ejemplo podemos tener los parámetros de conexión a la base de datos:

```
db.url=ejemplo.com:1527
db.user=john.doe
db.password=1234
```

Ahora pensemos que queremos cargarlos en un objeto en nuestra aplicación para poder acceder a ellos fácilmente. Una opción es crear una clase que tenga un atributo por cada propiedad que deseemos almacenar. Sin embargo este mecanismo es poco flexible a la hora de añadir nuevas propiedades, ya que habría que modificar la clase. En vez de ello podemos recurrir a un Map para almacenar las propiedades, siendo la clave el nombre de la propiedad, y el elemento su valor:

```
HashMap<String,String> map = new HashMap<String,String>();
map.put("db.url","ejemplo.com:1527");
map.put("db.user","john.doe");
map.put("db.password","1234");
```

Cuando necesitemos acceder al valor de una propiedad, la buscaremos por su clave:

```
String dbUrl = map.get("db.url");
```

Hay tres implementaciones de propósito general del interfaz Map: HashMap, TreeMap y LinkedHashMap. Si necesitamos que los elementos estén ordenados, utilizaremos TreeMap. Si buscamos la máxima velocidad y no nos importa el orden, utilizaremos HashMap. LinkedHashMap es un término medio, que ofrece un buen rendimiento y nos permite iterar según el orden de inserción.

Ahora veamos un ejemplo de uso de un Map. Supongamos que tenemos una lista de objetos de tipo Persona, como en el ejemplo del apartado anterior. Queremos agruparlos por edades, de manera que para cada edad tendremos una lista de personas.

```
//creamos una lista de personas de prueba
ArrayList<Persona> list = new ArrayList<Persona>();

list.add(new Persona("Jorge",30));
list.add(new Persona("Carlos",48));
list.add(new Persona("Miguel",29));
list.add(new Persona("Ana",48));
list.add(new Persona("Marcos",29));

//creamos un mapa en el que almacenaremos para cada edad una
//lista de personas:
HashMap<Integer,List<Persona>> mapa = new HashMap<Integer,List<Persona>>();

//recorremos la lista original de personas y vamos
//añadiendo al mapa:
```

```
Iterator<Persona> iterator = list.iterator();
while (iterator.hasNext())
{
      Persona persona = iterator.next();
      Integer edad = new Integer(persona.edad);

      //Buscamos la lista de personas de esa edad en el mapa:
      List<Persona> lista = mapa.get(edad);
      if (lista == null)
      {
            //si es el primero de esa edad, la lista
            //no existe aún, hay que crearla y meterla
            //en el mapa
            lista = new ArrayList<Persona>();
            mapa.put(edad,lista);
      }
      //añadimos la persona a la lista de personas de esa edad:
      lista.add(persona);
}

//ahora recorremos el mapa y vemos el resultado:
Set<Integer> edades = mapa.keySet();
Iterator<Integer> iteraEdades = edades.iterator();
while (iteraEdades.hasNext())
{
      Integer edad = iteraEdades.next();
      System.out.println("Personas con edad " + edad + ":");
      Iterator<Persona> iteraPersonas = mapa.get(edad).iterator();
      while (iteraPersonas.hasNext())
      {
            Persona persona = iteraPersonas.next();
            System.out.println(persona);
      }
}
```

El resultado será el siguiente:

```
Personas con edad 48:
{Carlos,48}
{Ana,48}
Personas con edad 29:
{Miguel,29}
{Marcos,29}
Personas con edad 30:
{Jorge,30}
```

Obsérvese que al haber utilizado HashMap las edades no han salido ordenadas.

Otro detalle a tener en cuenta es que cuando utilicemos HashMap debemos asegurarnos de que los objetos utilizados como clave implementan el método *hashCode()* correctamente, ya que HashMap utiliza este método para crear la tabla hash interna.

4.5. Ordenación de colecciones. La clase Collections.

Hasta ahora hemos visto algunas colecciones que mantienen los elementos ordenados, bien por el orden de inserción o bien por su orden natural. Sin embargo es habitual tener colecciones desordenadas que posteriormente queremos ordenar. Para ello podemos utilizar la clase java.util.Collections (no confundir con el interfaz *Collection*, en singular). La clase *Collections* proporciona una serie de métodos estáticos, entre ellos el método sort() que nos permite ordenar una lista. Al igual que vimos con el TreeSet, podemos ordenar según el orden natural de los elementos (si implementan el interfaz *Comparable*) o bien según otro orden definido por un *Comparator* que pasaremos como argumento.

El algoritmo utilizado es una adaptación de *mergesort,* lo que nos asegura una ejecución rápida y estable.

Veamos un ejemplo, basándonos de nuevo en la clase Persona, de ordenación en base al orden natural (la edad):

```
//Creamos una lista de personas de prueba
ArrayList<Persona> list = new ArrayList<Persona>();

list.add(new Persona("Jorge",30));
list.add(new Persona("Carlos",48));
list.add(new Persona("Miguel",29));
list.add(new Persona("Ana",48));
list.add(new Persona("Marcos",29));

//Ordenamos según el orden natural
Collections.sort(list);

//Recorremos la lista de personas
Iterator<Persona> iterator = list.iterator();
while (iterator.hasNext())
{
	Persona persona = iterator.next();
	System.out.println(persona);
}
```

El resultado es:

```
{Miguel,29}
{Marcos,29}
{Jorge,30}
{Carlos,48}
{Ana,48}
```

Ahora veamos el mismo ejemplo pero ordenando según el nombre (orden alfabético):

```
//Creamos una lista de personas de prueba
ArrayList<Persona> list = new ArrayList<Persona>();

list.add(new Persona("Jorge",30));
list.add(new Persona("Carlos",48));
list.add(new Persona("Miguel",29));
list.add(new Persona("Ana",48));
list.add(new Persona("Marcos",29));

//Ordenamos según el orden alfabético
Collections.sort(list,new Comparator<Persona>(){
	@Override
	public int compare(Persona arg0, Persona arg1) {
	    return arg0.nombre.compareToIgnoreCase(arg1.nombre);
	}
});

//Recorremos la lista de personas
Iterator<Persona> iterator = list.iterator();
while (iterator.hasNext())
{
	Persona persona = iterator.next();
	System.out.println(persona);
}
```

Como se puede observar hemos vuelto a utilizar una clase anónima para el *Comparator*, al igual que en el ejemplo del TreeSet. El resultado es:

```
{Ana,48}
{Carlos,48}
{Jorge,30}
{Marcos,29}
{Miguel,29}
```

A partir de Java 8 además de la clase *Collections* podemos utilizar el método *sort* del interfaz *List*, siendo equivalentes. Este método toma como argumento un *Comparator*, si es null utilizará el orden natural. Por tanto para el orden natural pondríamos:

```
list.sort(null);
```

Y para el orden no natural haríamos esto:

```
list.sort(new Comparator<Persona>(){
	@Override
	public int compare(Persona arg0, Persona arg1) {
	    return arg0.nombre.compareToIgnoreCase(arg1.nombre);
	}
});
```

La clase Collections proporciona muchos otros métodos para operar con listas, algunos de ellos son:

- shuffle: desordena una lista aleatoriamente.
- reverse: invierte el orden de la lista.
- addAll: añade a una lista un conjunto de elementos en lugar de uno solo.
- binarySearch: busca un elemento utilizando el algoritmo de búsqueda binaria.
- min/max: busca el máximo o el mínimo en una lista.

La idea final con la que debemos quedarnos es que cuando queramos hacer operaciones complejas con colecciones, debemos primero mirar en la clase *Collections* si hay algún método que nos permita hacer lo que deseamos, ya que en este caso no sólo ahorraremos tener que desarrollar el algoritmo sino que además probablemente el de la clase *Collections* será más eficiente y robusto.

5. Ejercicios Prácticos

5.1. Ejercicio Resuelto: tratamiento de textos

En el siguiente ejercicio mostraremos cómo procesar una cadena de texto utilizando las utilidades *String.split()* y *TreeSet*. *String.split()* es un método que nos permite dividir una cadena de texto en trozos, delimitados por un determinado texto en la cadena original.

A partir de una lista de direcciones de correo electrónico que se encuentran en una cadena de texto, el programa extrae los dominios y los almacena en una lista para utilizarlos posteriormente. El dominio es la parte de la dirección de correo que queda a la derecha de la arroba.

Las direcciones de correo electrónico se encuentran separadas entre sí por el carácter punto y coma ";". Utilizaremos el método *String.split()* para separar tanto las direcciones de correo entre sí (delimitador ";"), como los nombres de dominio de cada dirección (delimitador "@").

Guardaremos los dominios en un objeto de tipo *TreeSet* para evitar duplicados y obtener el resultado ordenado alfabéticamente.

```
import java.util.Iterator;
import java.util.TreeSet;

public class EjemploStrings {

    /**
     * @param args
     */
    public static void main(String[] args) {
```

```
        String textoOriginal = "uno@hotmail.com;dos@gmail.com;tres@yahoo.es;" +
                    "cuatro@gmail.com;cinco@yahoo.com;seis@hotmail.com";

        TreeSet<String> domains = new TreeSet<String>();

        String[] emails = textoOriginal.split(";");

        for (int i=0;i<emails.length;i++){
            String email = emails[i];
            String[] emailParts = email.split("@");
            if (emailParts.length == 2){
                String userName = emailParts[0];
                String domainName = emailParts[1];
                domains.add(domainName);
            }
        }

        //recorremos y mostramos los dominios recogidos:
        Iterator<String> iteraDomains = domains.iterator();

        for(int i=1;iteraDomains.hasNext();i++){
            String domainName = iteraDomains.next();
            System.out.println(i + ": " + domainName);
        }

    }

}
```