















CONSEJOS PARA LA PRUEBA ONLINE DE PROGRAMAME

- Buscar en internet fórmulas o algoritmos para aquellos problemas que penséis que puedan tener información en internet.
- Cada miembro del equipo leerá 3 o 4 ejercicios, clasificándolos por nivel de dificultad (fáciles, medios y difíciles), de tal manera que tendremos los 10 ejercicios divididos en 3 bloques. Una vez hecho esto cada miembro del equipo empezará a hacer uno de los ejercicios catalogados como fáciles, continuando con los medios.....
- **No empecinarse en un ejercicio.** Si en una hora no consigues que un ejercicio funcione pasa al siguiente. Retómalo al final si hay tiempo, u otro compañero puede ayudarte a rematarlo.
- Si en la columna de un problema predomina el color verde, hay que intentar resolverlo, ya que al resolverlo más equipos, en teoría es fácil (fijaros en la columna H de la imagen). De la misma manera, dejar para el final aquellos ejercicios de los que no hay envíos o hay muchos no aceptados (columna E, F o D).

RANK	TEAM		SCORE	A ●	B ●	C ○	D ●	E ○	F ●	G ●	H ●
1	IES Lluís Simarro 	Velivsa IES Lluís Simarro	6 823	1/112	4/95	2/82	6/67	0	0	2/67	5/120
2	IES Henri Matisse 	Diógenes digital IES Henri Matisse	4 364	0	2/124	2/67	0	0	0	1/82	1/51
3	IES San Vicente 	While(TRUE) IES San Vicente	4 425	1/133	4/54	0	0	0	0	2/83	2/55
4	IES Jaume II El Just 	InstanceOf group IES Jaume II El Just	3 278	1/127	3	0	0	0	0	3/77	1/34
5	IES San Vicente 	CopyPaste IES San Vicente	3 319	1/85	0	5	0	0	0	1/105	2/109
6	IES Jaume II El Just 	Los Solfamidas IES Jaume II El Just	3 322	0	0	1/63	0	0	0	1/132	1/127
7	EFA Moratalaz 	StackOverflow EFA Moratalaz	3 334	0	0	1/133	1/109	0	0	0	2/72
8	IES Henri Matisse 	Sin Excepción IES Henri Matisse	3 402	0	2/104	2/77	0	0	0	0	4/121
9	IES San Vicente 	Garra, orgullo y pasión IES San Vicente	3 457	0	5/116	0	0	0	0	1/106	5/75
10	IES Albarregas 	Lo que diga Ventura IES Albarregas	2 112	6	0	2/31	1	0	0	0	1/61
11	IES Henri Matisse 	matissa y exponentel IES Henri Matisse	2 132	0	0	0	0	0	0	1/51	1/81
12	IES Abastos 	Madavo IES Abastos	2 192	1/55	0	0	0	0	0	1	4/77
13	IES Albarregas 	Los hombres de Javi IES Albarregas	2 196	0	0	0	0	0	0	5/62	2/34
14	IES San Vicente 	bool win=false IES San Vicente	2 220	0	9	3/71	1	0	0	6	4/49
15	IES Abastos 	Ganesh IES Abastos	2 242	0	2/112	1	0	0	0	0	3/70
16	IES Lluís Simarro 	Delegatteam IES Lluís Simarro	2 291	0	5/113	1/98	0	0	0	0	0
17	IES San Vicente 	KillerC IES San Vicente	2 528	0	4	11/89	0	0	0	2	8/99



La mayor parte de los problemas responden a los patrones que se exponen a continuación. Se incluye un esqueleto de programa para procesarlos:

- 1) Leo primero cuántos casos de prueba se van a procesar

```
import java.util.Scanner;

public class Main{

    public static void main(String[] args) {
        int casos;
        Scanner leo = new Scanner(System.in);

        casos = leo.nextInt();
        for (int i = 0; i < casos; i++) {

        }

    }
}
```

- 2) Leemos valores hasta que leamos un valor que indique que el programa tiene que finalizar. Por ejemplo, leemos cadenas y finalizamos cuando leemos un 0.

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner leo = new Scanner(System.in);
        String linea;

        linea = leo.nextLine();
        while (!linea.equals("0")) {

            //código del problema

            linea = leo.nextLine();
        }

    }
}
```

- 3) Estamos leyendo permanentemente datos desde el teclado, en este caso strings.

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        String linea;

        Scanner leo = new Scanner(System.in);

        while (leo.hasNext()) {
            linea = leo.nextLine();
        }
    }
}
```



```
}  
}  
}
```

Cuando se ejecuta en un IDE el programa se va a estar ejecutando permanentemente. Cuando enviáis los programas al juez automático, en este hay un fichero que tiene muchos casos de prueba, y el programa lee ese fichero, y en este caso si acaba de ejecutarse, ya que lo está procesando `while leo.hasNext()`, es decir mientras en el fichero haya información. Cuando acabe de leerlo finaliza el programa.

IMPORTANTE:

Si después de leer un `int`, tengo que leer un `String`, tengo que leer el retorno de carro:

```
casos = leo.nextInt();  
leo.nextLine();  
linea = leo.nextLine();
```



ORDENACION DE INFORMACIÓN

Para ordenar un array de manera creciente:

```
int[] vectorInt = {3, 4, 6, 1, 2, 3, 4, 5, 7, 8, 9, 7, 7};

Arrays.sort(vectorInt);
for (int w : vectorInt) {
    System.out.print(w + " ");
}
```

Escribe: 1 2 3 3 4 4 5 6 7 7 8 9 BUILD SUCCESSFUL (total time: 0 seconds)

Para ordenar en orden inverso:

Tenemos diferencias entre enteros y cadenas

Con String e Integer funcionaría collections.reverseorder (ATENCIÓN definimos el array como Integer, no como int)

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.Scanner;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        Scanner leo = new Scanner(System.in);
        String linea;
        Integer[] vectorInt = {3, 4, 6, 1, 2, 3, 4, 5, 7, 8, 9, 7, 7};

        for (int w : vectorInt) {
            System.out.print(w + " ");
        }

        Arrays.sort(vectorInt, Collections.reverseOrder());
        System.out.println("");
        System.out.println(Arrays.asList(vectorInt));
    }
}
```

Para Enteros Integer (otra manera sencilla sería pasarlo a lista y desde ahí ordenarlos en orden inverso)

```
List<Integer> list = Arrays.asList(1, 4, 9, 16, 9, 7, 4, 9, 11);
System.out.println(list);
Collections.reverse(list);
System.out.println(list);
```

Para un ordenar un ArrayList: Collections.sort(arrayListInt);



Comparando y ordenando clases de objetos definidos por el usuario (por uno o mas criterios de ordenación)

http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=587:ejercicio-ejemplo-resuelto-interface-comparable-y-metodo-compareto-api-java-comparar-objetos-cu00911c&catid=58:curso-lenguaje-programacion-java-nivel-avanzado-i&Itemid=180

<https://jarroba.com/ordenar-un-arraylist-en-java/>

```
public class Persona implements Comparable<Persona>{
    public int dni, edad;
    public Persona( int d, int e){
        this.dni = d;
        this.edad = e;
    }

    public int compareTo(Persona o) {
        int resultado=0;
        if (this.edad<o.edad) {    resultado = -1;        }
        else if (this.edad>o.edad) {    resultado = 1;        }
        else {
            if (this.dni<o.dni) {    resultado = -1;        }
            else if (this.dni>o.dni) {    resultado = 1;        }
            else {    resultado = 0;    }
        }
        return resultado;
    }
}

public class Programa {

    public static void main(String arg[]) {
        Persona p1 = new Persona (749999999,35);
        Persona p2 = new Persona (72759474,30);
        if (p1.compareTo(p2) < 0 ) { System.out.println("La persona p1:
es menor."); }
        else if (p1.compareTo(p2) > 0 ) {System.out.println("La persona
p1: es mayor."); }
        else { System.out.println ("La persona p1 es igual a la persona
p2"); }
    }
}
```

En Programame sólo podemos enviar una clase, así que tendríamos que definir la clase persona dentro de la clase programa. Para enviarla NO podríamos crearla en un fichero aparte.

El siguiente código nos ordenaría un array de tres objetos de clase Persona, primero por edad, y después por dni.



```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.Scanner;
import java.util.List;

public class Main {

    static public class Persona implements Comparable<Persona> {

        public int dni, edad;

        public Persona(int d, int e) {
            this.dni = d;
            this.edad = e;
        }

        @Override
        public String toString() {
            return "Persona{" + "dni=" + dni + ", edad=" + edad + '}';
        }

        public int compareTo(Persona o) {
            int resultado = 0;
            if (this.edad < o.edad) {
                resultado = -1;
            } else if (this.edad > o.edad) {
                resultado = 1;
            } else {
                if (this.dni < o.dni) {
                    resultado = -1;
                } else if (this.dni > o.dni) {
                    resultado = 1;
                } else {
                    resultado = 0;
                }
            }
            return resultado;
        }
    }

    public static void main(String[] args) {

        Persona p1 = new Persona(749999999, 30);
        Persona p2 = new Persona(72759474, 30);

        if (p1.compareTo(p2) < 0) {
            System.out.println("La persona p1: es menor.");
        } else if (p1.compareTo(p2) > 0) {
            System.out.println("La persona p1: es mayor.");
        } else {
            System.out.println("La persona p1 es igual a la persona
p2");
        }
        Persona [] vector = new Persona[3];
    }
}
```



```
vector[0]=p1;  
vector[1]=p2;  
vector[2]= new Persona(72759474, 35);  
  
Arrays.sort(vector);  
System.out.println(Arrays.asList(vector));  
Arrays.sort(vector,Collections.reverseOrder());  
System.out.println(Arrays.asList(vector));  
    }  
}
```



RANGO DE LOS TIPOS BASICOS DE DATOS

<http://www.c4learn.com/java/java-primitive-data-types/>

Type	Contains	Default	Size	Range
boolean	true or false	false	1 bit	NA
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
byte	Signed integer	0	8 bits	-128 to 127
short	Signed integer	0	16 bits	-32768 to 32767
int	Signed integer	0	32 bits	-2147483648 to 2147483647
long	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	$\pm 1.4\text{E}-45$ to $\pm 3.4028235\text{E}+38$
double	IEEE 754 floating point	0.0	64 bits	$\pm 4.9\text{E}-324$ to $\pm 1.7976931348623157\text{E}+308$



COMO SEPARAR CADENA EN PARTES (Split ó stringtokenizer)

Mediante el método Split:

<http://codigomaldito.blogspot.com.es/2011/06/el-metodo-split-en-java.html>

En el siguiente ejemplo, se separa una cadena en subcadenas utilizando el carácter ',' dando lugar a un array.

```
public class EjemploSplit {  
    public static void main (String args[]) {  
        String numeros = "1,2,3,4,5,6";  
        String[] numerosComoArray = numeros.split(",");  
        for (int i = 0; i < numerosComoArray.length; i++) {  
            System.out.println(numerosComoArray[i]);  
        }  
    }  
}
```

Ejemplo de Split para separar las palabras que hay en una frase (aunque haya mas de un blanco entre palabras). OJO! Cuando hay mas de un espacio entre dos palabras, Split genera una cadena vacía por cada blanco que haya de mas.

```
public class EjemploSplit {  
    public static void main (String args[]) {  
        String numeros = "Dabale arroz a la zorra el abad";  
        String[] numerosComoArray = numeros.split(" ");  
        for (int i = 0; i < numerosComoArray.length; i++) {  
            System.out.println(numerosComoArray[i]);  
        }  
    }  
}
```

Salida generada:

Dabale
arroz
a
la

zorra
el

abad

Solución para quedarnos solo con las palabras de la cadena (desechamos las cadenas vacías):

```
public class EjemploSplit {  
  
    public static void main(String args[]) {  
        String numeros = "Dabale arroz a la zorra el abad";
```



```
String[] numerosComoArray = numeros.split(" ");
for (int i = 0; i < numerosComoArray.length; i++) {
    if (numerosComoArray[i].length() > 0) {
        System.out.println(numerosComoArray[i]);
    }
}
}
```

También podemos crear otro array o lista con los strings de longitud mayor que 0.

Mediante la clase StringTokenizer:

Por defecto separa una cadena por el espacio en blanco

<http://www.webtutoriales.com/articulos/java-stringtokenizer-y-split>

```
String linea = "elemento1 elemento2 elemento3";

StringTokenizer tokens = new StringTokenizer(linea);

while(tokens.hasMoreTokens()){

    System.out.println(tokens.nextToken());

}
```

El código anterior con split y una expression regular para separar por más de un blanco.

```
int j = 0;

String linea = "elemento1 elemento2 elemento3";

String [] campos = linea.split("\\s+");

while(j<campos.length){

    System.out.println(campos[j]);

    j++;

}
```



<http://www.adictosaltrabajo.com/tutoriales/introduccion-a-colecciones-en-java/>

Introducción a Colecciones en Java

En este tutorial vamos a ver qué son las colecciones de Java, los tipos más usados que hay y sus principales características.

Índice de contenidos

- [1. Introducción](#)
- [2. Entorno](#)
- [3. ¿Qué es un índice?](#)
- [4. Tipos de colecciones](#)
 - [4.1. Set](#)
 - [4.2. List](#)
 - [4.3. Map](#)
- [5. Stream API](#)
- [6. Conclusiones](#)
- [7. Referencias](#)

1. Introducción

En este tutorial vamos a profundizar en las colecciones en Java. Vamos a ver qué son y los distintos tipos de colecciones más usados que existen (Set, List y Map). También, vamos a ver que cada uno de los distintos tipos de colecciones puede tener, además, distintas implementaciones, lo que ofrece funcionalidad distinta. Por último, veremos por encima una de las novedades de Java 8 en las colecciones como son los *streams* lo que nos permite trabajar con las colecciones de una forma mucho más óptima y eficiente.

2. Entorno

El tutorial se ha realizado usando el siguiente entorno:

- Hardware: Portátil MacBook Pro Retina 15' (2.5 Ghz Intel Core I7, 16GB DDR3).
- Sistema Operativo: Mac OS Yosemite 10.10
- Entorno de desarrollo: IntelliJ IDEA
- Versión de Java: 1.8
- Versión de Maven: 3.1

3. ¿Qué son las colecciones?

Una colección representa un grupo de objetos. Estos objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica **Collection** para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección... Partiendo de la interfaz genérica **Collection** extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

4. Tipos de colecciones

En este apartado, vamos a analizar los principales tipos de colecciones que se encuentran, por defecto, en la plataforma de Java.

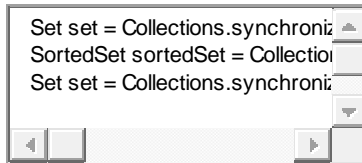
4.1. Set

La interfaz **Set** define una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de **Collection** añadiendo la restricción de que los elementos duplicados están prohibidos. Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos **equals** y **hashCode**. Para comprobar si dos **Set** son iguales, se comprobarán si todos los elementos que los componen son iguales sin importar en el orden que ocupen dichos elementos.

Dentro de la interfaz **Set** existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

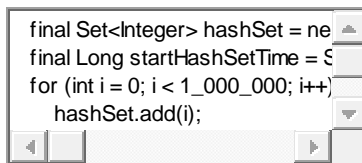
- **HashSet**: esta implementación almacena los elementos en una tabla *hash*. Es la implementación con **mejor rendimiento** de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Es la implementación más empleada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función *hash* disperse de forma correcta los elementos dentro de la tabla *hash*. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- **TreeSet**: esta implementación almacena los elementos ordenándolos en función de sus valores. Es bastante más lento que **HashSet**. Los elementos almacenados deben implementar la interfaz **Comparable**. Esta implementación garantiza, siempre, un rendimiento de $\log(N)$ en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- **LinkedHashSet**: esta implementación almacena los elementos en función del orden de inserción. Es, simplemente, un poco más costosa que **HashSet**.

Ninguna de estas implementaciones son sincronizadas; es decir, no se garantiza el estado del **Set** si dos o más hilos acceden de forma concurrente al mismo. Esto se puede solucionar empleando una serie de métodos que actúan de *wrapper* para dotar a estas colecciones de esta falta de sincronización:



- 1 Set set = Collections.synchronizedSet(new HashSet());
- 2 SortedSet sortedSet = Collections.synchronizedSortedSet(new TreeSet());
- 3 Set set = Collections.synchronizedSet(new LinkedHashSet());

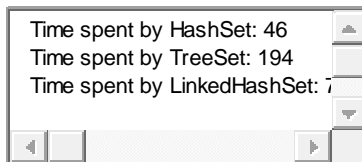
Una vez explicados los distintos tipos de **Set**, veremos cómo se crean y mostraremos sus diferencias en los tiempos de inserción. Como hemos visto anteriormente, el más rápido debería ser **HashSet** mientras que, por otro lado, el más lento debería ser **TreeSet**. Vamos a comprobarlo con el siguiente código:



- 1 final Set<Integer> hashSet = new HashSet<Integer>(1_000_000);
- 2 final Long startHashSetTime = System.currentTimeMillis();
- 3 for (int i = 0; i < 1_000_000; i++) {
- 4 hashSet.add(i);
- 5 }
- 6 final Long endHashSetTime = System.currentTimeMillis();
- 7 System.out.println("Time spent by HashSet: " + (endHashSetTime - startHashSetTime));
- 8
- 9 final Set<Integer> treeSet = new TreeSet<Integer>();
- 10 final Long startTreeSetTime = System.currentTimeMillis();
- 11 for (int i = 0; i < 1_000_000; i++) {
- 12 treeSet.add(i);
- 13 }
- 14 final Long endTreeSetTime = System.currentTimeMillis();

```
15 System.out.println("Time spent by TreeSet: " + (endTreeSetTime - startTreeSetTime));  
16  
17 final Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(1_000_000);  
18 final Long startLinkedHashSetTime = System.currentTimeMillis();  
19 for (int i = 0; i < 1_000_000; i++) {  
20     linkedHashSet.add(i);  
21 }  
22 final Long endLinkedHashSetTime = System.currentTimeMillis();  
23 System.out.println("Time spent by LinkedHashSet: " + (endLinkedHashSetTime -  
    startLinkedHashSetTime));
```

A continuación, el resultado de los tiempos obtenidos:



- 1 Time spent by HashSet: 46
- 2 Time spent by TreeSet: 194
- 3 Time spent by LinkedHashSet: 74

Los tiempos obtenidos demuestran que, efectivamente, el tiempo de inserción es menor en **HashSet** y mayor en **TreeSet**. Es importante destacar que la inicialización del tamaño inicial del **Set** a la hora de su creación es importante ya que, en caso de insertar un gran número de elementos, podrían aumentar el número de colisiones y; con ello, el tiempo de inserción.

4.2. List

La interfaz **List** define una sucesión de elementos. A diferencia de la interfaz **Set**, la interfaz **List** sí admite elementos duplicados. A parte de los métodos heredados de **Collection**, añade métodos que permiten mejorar los siguientes puntos:

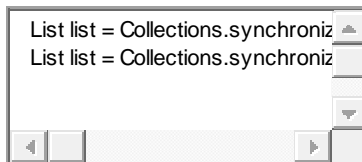
- Acceso posicional a elementos: manipula elementos en función de su posición en la lista.
- Búsqueda de elementos: busca un elemento concreto de la lista y devuelve su posición.
- Iteración sobre elementos: mejora el **Iterator** por defecto.

- **Rango de operación:** permite realizar ciertas operaciones sobre rangos de elementos dentro de la propia lista.

Dentro de la interfaz **List** existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

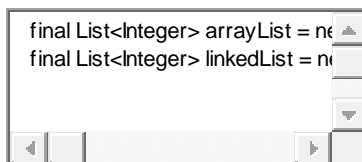
- **ArrayList:** esta es la implementación típica. Se basa en un *array* redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones.
- **LinkedList:** esta implementación permite que mejore el rendimiento en ciertas ocasiones. Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

Ninguna de estas implementaciones son sincronizadas; es decir, no se garantiza el estado del **List** si dos o más hilos acceden de forma concurrente al mismo. Esto se puede solucionar empleando una serie de métodos que actúan de *wrapper* para dotar a estas colecciones de esta falta de sincronización:



- 1 `List list = Collections.synchronizedList(new ArrayList());`
- 2 `List list = Collections.synchronizedList(new LinkedList());`

A continuación, vamos a ver cómo se crean los distintos tipos de interfaces:



- 1 `final List<Integer> arrayList = new ArrayList<Integer>();`
- 2 `final List<Integer> linkedList = new LinkedList<Integer>();`

El cuándo usar una implementación u otra de **List** variará en función de la situación en la que nos encontremos. Generalmente, **ArrayList** será la implementación que usemos en la mayoría de situaciones. Sobre todo, varían los tiempos de inserción, búsqueda y eliminación de elementos, siendo en unos casos una solución más óptima que la otra.

4.3. Map

La interfaz **Map** asocia claves a valores. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.

Dentro de la interfaz **Map** existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- **HashMap:** esta implementación almacena las claves en una tabla *hash*. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función *hash* disperse de forma correcta los elementos dentro de la tabla *hash*. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- **TreeMap:** esta implementación almacena las claves ordenándolas en función de sus valores. Es bastante más lento que *HashMap*. Las claves almacenadas deben implementar la interfaz **Comparable**. Esta implementación garantiza, siempre, un rendimiento de $\log(N)$ en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- **LinkedHashMap:** esta implementación almacena las claves en función del orden de inserción. Es, simplemente, un poco más costosa que **HashMap**.

Ninguna de estas implementaciones son sincronizadas; es decir, no se garantiza el estado del **Map** si dos o más hilos acceden de forma concurrente al mismo. Esto se puede solucionar empleando una serie de métodos que actúan de *wrapper* para dotar a estas colecciones de esta falta de sincronización:

```
Map map = Collections.synchronizedMap(new HashMap());  
SortedMap sortedMap = Collections.synchronizedSortedMap(new TreeMap());  
Map map = Collections.synchronizedMap(new LinkedHashMap());
```

- 1 `Map map = Collections.synchronizedMap(new HashMap());`
- 2 `SortedMap sortedMap = Collections.synchronizedSortedMap(new TreeMap());`
- 3 `Map map = Collections.synchronizedMap(new LinkedHashMap());`

A continuación, vamos a ver cómo se crean los distintos tipos de interfaces:

```
final Map<Integer, List<String>> hashMap = new HashMap<Integer, List<String>>();  
final Map<Integer, List<String>> treeMap = new TreeMap<Integer, List<String>>();  
final Map<Integer, List<String>> linkedHashMap = new LinkedHashMap<Integer, List<String>>();
```

- 1 `final Map<Integer, List<String>> hashMap = new HashMap<Integer, List<String>>();`
- 2 `final Map<Integer, List<String>> treeMap = new TreeMap<Integer, List<String>>();`
- 3 `final Map<Integer, List<String>> linkedHashMap = new LinkedHashMap<Integer, List<String>>();`

El cuándo usar una implementación u otra de **Map** variará en función de la situación en la que nos encontremos. Generalmente, **HashMap** será la implementación que usemos en la mayoría de situaciones. **HashMap** es la implementación con mejor rendimiento (como se ha podido comprobar en el análisis de **Set**), pero en algunas ocasiones podemos decidir renunciar a este rendimiento a favor de cierta funcionalidad como la ordenación de sus elementos.

ac

5. Stream API

Gracias a la llegada de Java 8, las colecciones han aumentado su funcionalidad con la llegada de los **streams**. Los **streams** permiten realizar operaciones funcionales sobre los elementos de las colecciones.

A continuación, mostramos un ejemplo de las bondades de los **streams** donde, a partir de una lista de personas (donde cada una de ellas tiene un nombre), obtenemos una lista con todos los nombres:

```
1 List<Person> people = new ArrayList<Person>();  
2 List<String> names = people.stream().map(Person::getName).collect(Collectors.toList());
```

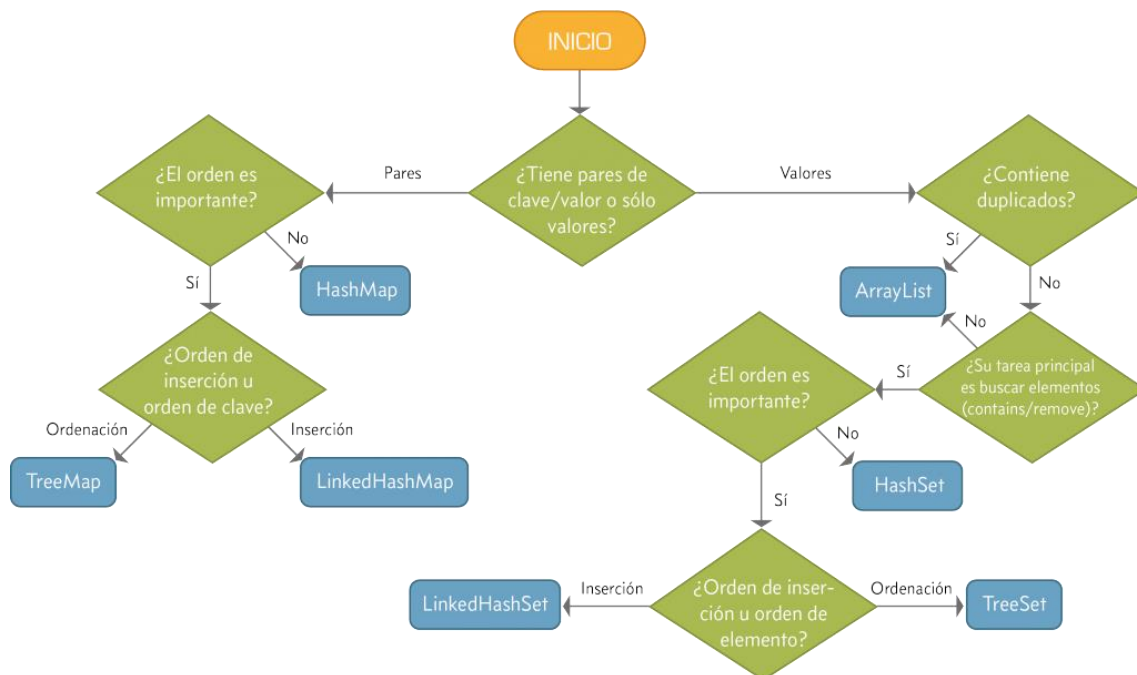
Como hemos visto en el ejemplo, de una forma muy fácil hemos obtenido todos los nombres. Para más detalles de este tipo de novedades, tenemos un [Curso de Java 8](#) con el que se podrán ahondar en este tipo de conocimientos.

6. Conclusiones

Como hemos visto a lo largo de este tutorial, Java proporciona una serie de estructuras muy variadas para almacenar datos. Estas estructuras, ofrecen diversas funcionalidades: ordenación de elementos, mejora de rendimiento, rango de operaciones... Es importante conocer cada una de ellas para saber cuál es la mejor situación para utilizarlas. Un buen uso de estas estructuras mejorará el rendimiento de nuestra aplicación.

Para conocer qué tipo de colección usar, podemos emplear el siguiente diagrama:

Diagrama de decisión para uso de colecciones Java



Además, gracias a Java 8 y sus *streams*, las operaciones con las colecciones pueden ser mucho más óptimas.



JAVA ES MUCHO MAS LENTO QUE C EN LA ENTRADA/SALIDA

Faster Input for Java

The ACM ICPC allows competitors to use Java, but is Java a feasible option? Compared to C, Java requires many more lines of code to perform the same task and (perhaps most important) Java is generally slower than C. Input/output is one of the slowest parts (compared to C). So, can we do anything to speed I/O in Java?

Problem: *Scanner is Sloooooow*

Using `Scanner` to parse input is convenient, but too slow. Using `BufferedReader` and `StringTokenizer` is much faster, but its a lot of typing during a competition. Can we make Java easier for the ACM ICPC?

Too much typing in Java:

C	<pre>double x; scanf("%lf", &x);</pre>
Java with Scanner	<pre>Scanner input = new Scanner(System.in); double x = input.nextDouble();</pre>
Java with BufferedReader	<pre>import java.io.* // must "throw IOException", too BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); // this works only if line contains just 1 value double x = Double.parseDouble(br.readLine());</pre>

How Slow?

I ran some tests to read 10,000,000 `int` or `double` from a file. Times are shown in the tables below. I ran the tests on an Intel Core2 Duo 2.4GHz cpu with Windows XP Pro SP3, Sun JDK 6.0r22, and GNU gcc 4.5.2.

Table 1. Time to read 10,000,000 `int` values from file

Input Method	Time (sec)
C <code>scanf("%d", &arg)</code>	3.78



Input Method	Time (sec)
Scanner.parseInt()	29.52
BufferedReader + inline Integer.parseInt	2.89
BufferedReader + Reader.nextInt method	3.01

Table 2. Time to read 10,000,000 double values from file

Input Method	Time (sec)
C scanf("%lf", &arg)	11.9
Scanner.parseDouble()	66.86
BufferedReader + inline Double.parseDouble	3.06
BufferedReader + Reader.nextDouble method	3.14

The times show that Scanner is much slower than C. On some example problems in Aj. Jittat's ACM training session, my Java solutions failed because they exceeded the run time limit for the task.

BufferedReader plus a parser method are almost as fast as C, but require a lot more typing. If you can use Eclipse, Eclipse will automatically add imports and try/catch or throws, which is a help but not enough help.

Listing 1. Sample methods to read using Scanner and BufferedReader. We split the input line into string tokens, since one line may contain multiple values. To split the input, `StringTokenizer` is 4X faster than `string.split()`.



<pre>/** Read count integers using Scanner */ static int scanInteger(int count) { Scanner scanner = new Scanner(input); int last = 0; while (count-- > 0) { last = scanner.nextInt(); } return last; }</pre>	<pre>/** Read count integers using BufferedReader */ static int readIntegers(int count) throws IOException { BufferedReader reader = new BufferedReader(new InputStreamReader(input)); StringTokenizer tokenizer = new StringTokenizer(""); int last = 0; while (count-- > 0) { if (! tokenizer.hasMoreTokens()) { tokenizer = new StringTokenizer(reader.readLine()); } last = Integer.parseInt(tokenizer.nextToken()); } return last; }</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Creating Reusable Code

How to reduce the burden of reading input in Java? Java has some advantages over C/C++: fewer syntax errors, maybe fewer logic errors, good code completion, automatic compilation, and debugger in Eclipse.

To make Java easier to use, let's put the reader code in a separate class. (You can put many classes in one source file, provided that only one class is "public".) We can copy our `Reader` class into *every* ACM task, so we only have to write it once during a competition. One person can write it while the others study the tasks and design a solution.

Listing 2 is an example. Let me know if you find any shorter, more efficient code. I use `StringTokenizer` instead of `string.split()` because `StringTokenizer` is much faster. I use static methods so we don't have to create or keep track of a "Reader" object, and use the default (package) scope so I don't have to type "public".

Putting the input methods in a separate class has almost no effect on the speed in my benchmark. But it *does* make our code more reusable and simplifies the rest of your task code.

Its still a lot of code, but after you type it once you can copy the class into every task.

So in your ACM tasks you can write:

```
Reader.init( System.in ); // connect Reader to an input stream
double x = Reader.nextDouble();
int n = Reader.nextInt();
```

Listing 2. Reusable class for reading int and double.



```
/** Class for buffered reading int and double values */
class Reader {
    static BufferedReader reader;
    static StringTokenizer tokenizer;

    /** call this method to initialize reader for InputStream */
    static void init(InputStream input) {
        reader = new BufferedReader(
            new InputStreamReader(input) );
        tokenizer = new StringTokenizer("");
    }

    /** get next word */
    static String next() throws IOException {
        while ( ! tokenizer.hasMoreTokens() ) {
            //TODO add check for eof if necessary
            tokenizer = new StringTokenizer(
                reader.readLine() );
        }
        return tokenizer.nextToken();
    }

    static int nextInt() throws IOException {
        return Integer.parseInt( next() );
    }

    static double nextDouble() throws IOException {
        return Double.parseDouble( next() );
    }
}
```



Cambiar el formato de la separación de los doublé:

```
Locale usa = new Locale("es", "ES");  
        NumberFormat formatter =  
NumberFormat.getCurrencyInstance(usa);  
  
System.out.println(formatter.format(datosParticipantes[i].mangal));
```