Ciclo de Desarrollo de Aplicaciones Multiplataforma

Grado Superior

U.T.5: ALMACENAMIENTO DE LA INFORMACIÓN EN ESTRUCTURAS DE DATOS

Módulo de Programación

Departamento de Informática

I.E.S. Monte Naranco

5.1 ESTRUCTURAS

Una estructura es un tipo de declaración que puede contener más de un dato.

5.2 ARRAYS

Cuando en una aplicación el número de elementos que debemos utilizar para resolver un problema es grande, resulta pesado tener que declarar, definir y utilizar una variable por cada elemento (por ejemplo, no es muy práctico si queremos gestionar las notas de programación de 30 alumnos tener que declarar 30 variables). Para solucionar estas situaciones los lenguajes de programación poseen estructuras de datos que permiten declarar y utilizar con un solo nombre un conjunto de variables del mismo tipo. Estas estructuras de datos son los arrays.

Las propiedades de los arrays son:

1. Se utilizan para poder guardar un conjunto de datos relacionados.
2. Todos los datos incluidos en el array son del mismo tipo. Los arrays pueden contener tanto tipos primitivos de datos, como tipos complejos (clases).
3. Antes de utilizar un array se ha de indicar su tamaño. A partir de ese momento no se puede cambiar. Son estructuras de datos estáticas.
4. A un elemento se accede a través de la posición que ocupa.

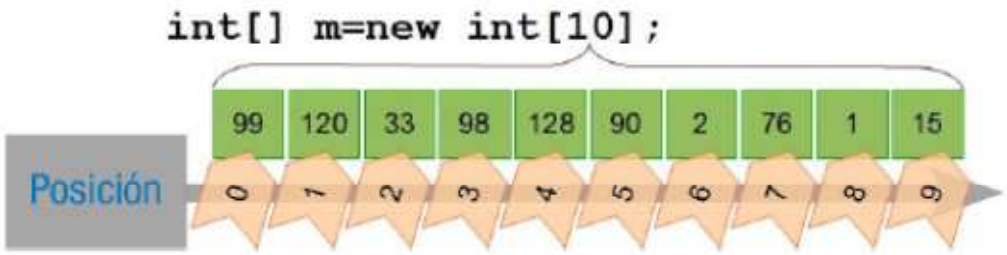
Existen tres tipos de arrays dependiendo de las posiciones que sean necesarias indicar para acceder a un determinado dato:

Arrays unidimensionales: para acceder a un dato del array se ha de indicar una única posición. También son conocidos con el nombre de vectores.

Arrays bidimensionales: para poder acceder a un dato del array se han de indicar dos posiciones. También se denominan matrices.

Arrays multidimensionales: son aquellos donde para poder acceder a un determinado dato hacen falta tres o más posiciones.

5.3 ARRAYS UNIDIMENSIONALES O VECTORES



Declaración

Para declarar un array se utilizan corchetes, así se especifica que es un array y no una simple variable del tipo indicado.

tipo [ ] identificador;

o bien

tipo identificador [ ];

Donde:

tipo es el tipo de dato de los elementos del vector.

identificador es el nombre del vector.

Ejm:

float [ ] notas; /*declara un vector llamado notas de tipo float, donde notas es un puntero; su inicialización por defecto es null*/

Creación

Los arrays se crean con el operador **new**.

identificador=new tipo [número de elementos];

Dónde:

tipo es el tipo de dato de los elementos del vector.

identificador es el nombre del array.

número de elementos es el tamaño del vector.

Ejm:

notas = new float [12];

Se puede declarar y crear todo junto:

tipo [ ] identificador = new tipo [nº elementos]; /*declarar y definir un vector de número de elementos de un tipo concreto*/

Ejm: float [ ] notas = new float [100]; /* Una vez creado el array todas sus posiciones son inicializadas al valor por defecto (0, 0.0, false o null si es de tipo complejo (clases)*/

Inicialización

Los vectores tienen posiciones o índices, siendo la primera posición la 0.

Podemos hablar de dos tipos de inicializaciones:

1. **Inicialización explícita:** primero se hace la declaración y luego se dan valores a las distintas posiciones del vector.

   Ejm:

   int [ ] mi_vector=new int [2]; /*Cuando se instancia un array sus componentes se inicializan a sus valores por defecto, en este caso 0*/

   mi_vector[0]=15;

   mi_vector[1]=92;

2. **Inicialización implícita:** se puede dar valores a las posiciones del vector en el mismo momento de la declaración.

   Ejm:

   Int [ ] vector= {1,2,3,4,5}; /*No hace falta instanciar, ya se hace directamente la reserva de memoria y se le asignan valores*/

   String [ ] lista_Nombres={“Maria”,”Pepe”,Gonzalo”};

   Integer [ ] enteros={new Integer(12), new Integer(28)};

   Java determina el tamaño del array en función de los valores asignados, haciendo la reserva de memoria sin tener que hacer new.

   No es mejor ni una forma ni otra. Hay veces que es más cómodo usar un método y otras otro.

Acceso a los elementos

Para acceder a los diferentes elementos de un array se han de utilizar los índices, con el fin de indicar la posición del elemento al que queremos acceder.

nombre_vector [pos] /*nombre del vector y entre corchetes la posición a la que queremos acceder*/

- En Java el índice del primer componente de un vector es siempre 0.
- El tamaño del vector se puede conocer utilizando la propiedad length, por lo tanto el índice del último componente es: nombre_vector.length.

Ejm:

```
numeros[2]=3;
cadenas[0]="hola";
cadenas[1]=new String ("adiós");
enteros[1]=new Integer(28);
```

Aunque añadamos más elementos al vector no hay problema, con el length llegamos hasta el final.

Recorrido

Para recorrer un vector hace falta una estructura repetitiva que permita pasar por todas las posiciones del array:

- **Bucle for normal**

```
int vEnteros[ ]=new int[10];
for(int indice = 0; indice < vEnteros.length; indice++) {
 System.out.print("\nTeclee un numero : ");
 vEnteros[indice] = teclado.nextInt();
}
```

- **Bucle for each o for extendido**

Facilita el recorrido sin necesidad de definir el número de elementos a recorrer. Es útil cuando tenemos que realizar recorridos completos (empezando desde la primera posición hasta la última). Se incorporó con la versión 5.0 de la máquina virtual de java, para que resultase más sencillo la creación de bucles for a la hora de recorrer los arrays.

Sintaxis:

```
for (tipo variable: nombre_array) {
    instrucciones
}
```

Por cada elemento que se encuentra dentro del vector empezando por el primero, debemos ejecutar las instrucciones que se indican. La variable almacena en cada paso el valor que se visita, solo existiendo durante la ejecución del bucle, desapareciendo después.

Ejm:

```
int [ ] vec=new int[5];
for(int recc:vec)
    System.out.println (recc);
```

Operaciones con Arrays

1. Ordenación

Cuando manejamos listas de información con muchos elementos como es el caso de los arrays, el hecho de que éstos estén ordenados facilita bastante las búsquedas de un elemento determinado para poder realizar operaciones posteriores, como pueden ser: borrar, consultar y modificar datos, ya que en muchos casos no habrá que recorrer todo el array para saber si el elemento que buscamos está o no en él.

La ordenación de un array consiste en organizar sus elementos con respecto a un criterio. Por ejemplo, si los elementos son enteros podemos referimos a una ordenación que puede ser creciente o decreciente.

Vamos a ver algunos métodos aplicados a los vectores o arrays unidimensionales, puesto que estos métodos se pueden generalizar a tablas de dos o más dimensiones.

- **Método sort dentro de la clase Arrays, facilitado por la API de Java**

```
package ordenacion;
import java.util.*;
public class Ordenar {
    public static void main(String[] args) {
    //Array de String
            String[ ] nombres = {"Pepe", "Juan", "Alex","Julian", "Francisco", "Luis"};
     //Ordena el array en orden creciente.
            Arrays.sort(nombres);
        /*Para ordenarlo de forma decreciente debemos indicarlo utilizando el método
    reverseOrder() de la clase Collections, en el método sort de la clase Arrays
    (Arrays.sort(nombres,Collections.reverseOrder()))*/
```

```
//Mostramos el array ya ordenado
        for (String i : nombres)  {
            System.out.print(i + ", ");
        }
    }
}
```

Tanto para la clase Arrays como Collections, debemos importar el paquete java.util.

Así como sort() funciona para arrays de cualquier tipo de datos, Collections.reverseOrder(), solo funciona para arrays de objetos. Por eso si queremos ordenar de forma descendente arrays de tipos de datos simples, podemos utilizar la clase envolvente o wrapper equivalente al tipo de datos básico.

Un wrapper permite convertir un dato de tipo primitivo en un objeto. Existe un wraper (Boolean, Char, Byte, Short, Long, Integer, Float, Double) para cada tipo primitivo de datos (boolean, char, byte, short, long, int, float, double) en el paquete java.lang. Por ejemplo, declarar un vector de tipo Integer en lugar de int sería: Integer[ ] numeros={1,3,6}

➢ Ordenación por inserción directa o método de la baraja

Supongamos que tenemos un vector V de "n" posiciones que deseamos ordenar de manera ascendente (de menor a mayor). El método de la baraja consiste en repetir el siguiente proceso desde la segunda componente del vector hasta la última. Se toma la componente que toca y se inserta en el lugar que le corresponde entre las componentes situadas a su izquierda (que ya estarán ordenadas). Las componentes superiores a la tratada se desplazan un lugar a la derecha.

Ejemplo	3	2	4	1	2
Paso 1	2	3	4	1	2
Paso 2	2	3	4	1	2
Paso 3	1	2	3	4	2
Paso 4	1	2	2	3	4

La función Java correspondiente sería:

```
package ordenar;
import java.util.Scanner;
public class Baraja {
      private Scanner tecla = new Scanner (System.in);
      private final int TAMAÑO2=3;
```

```
public Baraja(){
      int[] vec2=new int[TAMAÑO2];
      System.out.println("Vamos a cargar el vector");
      cargar(vec2);
      ordenarBaraja(vec2);
      System.out.println("El vector es: ");
      visualizar(vec2);
}
public void cargar (int [] vec){
  for ( int cont=0;cont<vec.length;cont++){
      System.out.println("Meter el número de la posición "+cont);
      vec[cont]=tecla.nextInt();
  }
}
public void ordenarBaraja (int[] vec){
      int aux;
      for (int recorr1=0;recorr1<vec.length;recorr1++){
            for (int recorr2=0;recorr2<recorr1;recorr2++)
      /*Ordena de forma creciente. Para ordenar en orden decreciente
       * debo cambiar if(vec[recorr1]<vec[recorr2]), por
      if(vec[recorr1]>vec[recorr2])*/
            if(vec[recorr1]<vec[recorr2]){
                  aux=vec[recorr2];
                  vec[recorr2]=vec[recorr1];
                  vec[recorr1]=aux;
            }
      }
}

public void visualizar (int [] vec){
      for (int variable:vec)
      System.out.print(variable+" ");
      System.out.println();
}

public static void main(String[] args) {
      Baraja interfaz=new Baraja();
}
```

➢ Ordenación por selección directa

El método consiste en repetir el siguiente proceso desde la primera componente hasta la penúltima. Se selecciona la componente de menor valor de todas las situadas a la derecha de la tratada, y se intercambia con ésta.

Ejemplo	3	2	4	1	2
Paso 1	1	2	4	3	2
Paso 2	1	2	4	3	2
Paso 3	1	2	2	3	4
Paso 4	1	2	2	3	4

La función Java correspondiente sería:

```
package ordenar;
import java.util.Scanner;
public class SeleccionDirecta {
	private Scanner tecla = new Scanner (System.in);
	private final int TAMAÑO2=3;
	public SeleccionDirecta(){
		int[] vec2=new int[TAMAÑO2];
		System.out.println("Vamos a cargar el vector");
		cargar(vec2);
		ordenarSeleccion(vec2);
		System.out.println("El vector es: ");
		visualizar(vec2);
	}

	public void cargar (int [] vec){
		for ( int cont=0;cont<vec.length;cont++){
			System.out.println("Meter el número de la posición "+cont);
			vec[cont]=tecla.nextInt();
		}
	}
```

```
    public void ordenarSeleccion (int[] vec){
          int recorr1,recorr2,aux,minimo=0,posicion=0;
          for (recorr1=0;recorr1<vec.length-1;recorr1++){
               minimo=vec[recorr1];
               for (recorr2=recorr1+1;recorr2<vec.length;recorr2++)
                    if(minimo>vec[recorr2]){/*Para ordenar en orden
                    decreciente debo poner if(minimo<vec[recorr2])*/
                         minimo=vec[recorr2];
                         posicion=recorr2;
                    }

                    if(vec[recorr1]>minimo){/*Para ordenar en orden
                    decreciente debo poner if(vec[recorr2]<minimo)*/
                         aux=vec[recorr1];
                         vec[recorr1]=vec[posicion];
                         vec[posicion]=aux;
                    }
          }
     }

     public void visualizar (int [] vec){
          for (int variable:vec)
          System.out.print(variable+" ");
          System.out.println();

     }

     public static void main(String[] args) {
          SeleccionDirecta interfaz=new SeleccionDirecta();
     }
}
```

➢ **Método de la burbuja**

Podemos ordenar de izquierda a derecha o viceversa. Si se ordena de izquierda a derecha, en la primera pasada el elemento mayor se deja en la última posición, en la segunda pasada el segundo elemento mayor se deja en la penúltima posición, y así sucesivamente hasta llegar a "n-1" pasadas, siendo "n" la dimensión del vector.

Si se ordena de derecha a izquierda, en la primera pasada el elemento menor de todos se coloca en la primera posición, en la segunda pasada el segundo elemento menor se coloca en la segunda posición, y así sucesivamente hasta llegar a "n-1" pasadas.

Recorrido de izquierda a derecha

Ejemplo	3	2	4	1	2
Paso 1	2	3	1	2	4
Paso 2	2	1	2	3	4
Paso 3	1	2	2	3	4
Paso 4	1	2	2	3	4

Recorrido de derecha a izquierda

Ejemplo	3	2	4	1	2
Paso 1	1	3	2	4	2
Paso 2	1	2	3	2	4
Paso 3	1	2	2	3	4
Paso 4	1	2	2	3	4

La función Java correspondiente sería:

```
package ordenar;
import java.util.Scanner;
public class Burbuja {
	private Scanner tecla = new Scanner (System.in);
	private final int TAMAÑO2=3;
	public Burbuja(){
		int[] vec2=new int[TAMAÑO2];
		System.out.println("Vamos a cargar el vector");
		cargar(vec2);
		ordenarBurbuja(vec2);
		System.out.println("El vector es: ");
		visualizar(vec2);
	}
	public void cargar (int [] vec){
		for ( int cont=0;cont<vec.length;cont++){
			System.out.println("Meter el número de la posición "+cont);
			vec[cont]=tecla.nextInt();
		}
	}
```

```
	public void ordenarBurbuja (int[]vec){

		int recorr1,recorr2,aux;
		for (recorr1=0;recorr1<vec.length-1;recorr1++){
			for (recorr2=0;recorr2<vec.length-1;recorr2++)
		/*Ordena de forma creciente. Para ordenar en orden decreciente
		 * debo cambiar if(vec[recorr2]>vec[recorr2+1]), por
		if(vec[recorr2]<vec[recorr2+1])*/
			if(vec[recorr2]>vec[recorr2+1]) {
				aux=vec[recorr2];
				vec[recorr2]=vec[recorr2+1];
				vec[recorr2+1]=aux;
			}
		}
	}

	public void visualizar (int [] vec){
		for (int variable:vec)
		System.out.print(variable+" ");
		System.out.println();
	}

	public static void main(String[] args) {

		Burbuja interfaz=new Burbuja();
	}
}
```

2. Búsquedas

La búsqueda en un array consiste en comprobar si un elemento determinado se encuentra en el mismo. Dependiendo si el array está ordenado o no, la búsqueda se podrá realizar mediante un método u otro.

2.1. Búsqueda en arrays desordenados

✓ **Búsqueda secuencial**

Consiste en recorrer los elementos del array, para cada uno de ellos debemos comprobar si dicho elemento coincide con el valor buscado. Al finalizar la búsqueda, nos podemos encontrar con las siguientes situaciones:

- Se ha localizado el elemento que se buscaba.
- El elemento no existe.

En las búsquedas lo más importante es controlar las posiciones a las que se accede, es decir, no acceder a posiciones inexistentes, ya que entonces nos daría el error de ejecución**: "ArrayIndexoutBoundsException".**

La función en java correspondiente a la búsqueda secuencial en arrays desordenados utilizando un interruptor sería:

Un interruptor, también denominado conmutador o switch, es una variable que solo puede tomar dos valores diferentes durante el desarrollo del programa (true-false, 0-1). Se utiliza para:

- Saber si algo ha sucedido.
- Hacer que dos bloques de instrucciones se ejecuten alternativamente.

```
package buscarSecuencial;
import java.util.*;
public class BuscarSecuencial{
      private Scanner tecla = new Scanner (System.in);
      private final int TAMAÑO2=3;
      public BuscarSecuencial(){
            int[] vec2=new int[TAMAÑO2];
            System.out.println("Vamos a cargar el vector ");
            cargar(vec2);
            System.out.println("El vector es: ");
            visualizar(vec2);
            buscar(vec2);
      }

      public void cargar (int [] vec){
            for (int cont=0;cont<vec.length;cont++){
              System.out.println("Meter el número de la posición"
              +cont);
              vec[cont]=tecla.nextInt();
            }
      }

      public void buscar (int []vec){
            //Toma de consola el valor a buscar
             System.out.print("\n\nTeclee el valor a buscar : ");
             int valor = tecla.nextInt();
```

```
            //Realiza la búsqueda lineal en un vector ordenado
            int indice=0, sw=0;
            while (indice < vec.length && sw==0){
                  if (vec[indice]==valor)
                        sw=1;
                  else
                        indice++;
             }

            //Analiza el resultado de la búsqueda
            if (sw==1)
                  System.out.println("\nEl valor: "+valor+" está en la
                  posición:"+ indice);
             else
                  System.out.println("\nEl valor: "+valor+"no está en el
                  vector");
            }
      public void visualizar (int [] vec){
            for (int cont=0;cont<vec.length;cont++)
                  System.out.print(vec[cont]+" ");
                  System.out.println();
            }
      public static void main(String[] args){
            BuscarSecuencial interfaz=new BuscarSecuencial();
      }
}
```

2.2. Búsqueda en arrays ordenados

✓ Búsqueda secuencial

Es el mismo método visto para arrays desordenados, pero teniendo en cuenta que si el array está ordenado, por ejemplo de forma ascendente, la búsqueda se realizará hasta encontrar el elemento en cuestión, o hasta alcanzar un elemento con un valor superior al buscado. La búsqueda en vectores ordenados es más rápida, que en desordenados

La función en java correspondiente a la búsqueda secuencial en arrays ordenados utilizando un interruptor sería:

```
package ordenacion;
import java.util.*;
public class OrdenadoSecuencial {
   Scanner teclado=new Scanner(System.in);
   public OrdenadoSecuencial(){
      int [] vEnteros={2,3,4,5};
      int indice=0, sw=0;
      System.out.print("\n\nTeclee el valor a buscar : ");
      int valor = teclado.nextInt();
      while(indice<vEnteros.length && sw==0) {
            if (vEnteros[indice]>=valor)
               sw=1;
            else
                 indice++;
      }

      //Analiza el resultado de la búsqueda
      if ((sw==1) && (vEnteros[indice]==valor))
         System.out.println(valor+" está en la posición "+ indice);
      else
          System.out.println (valor+" no existe en el vector");
   }

      public static void main(String[] args) {
           OrdenadoSecuencial interfaz=new OrdenadoSecuencial();
      }
 }
```

✓ **Búsqueda binaria o dicotómica**

Este método consiste en calcular cuál es la posición del medio de la tabla y comprobar si el valor que buscamos está en esa posición. Si no lo está, buscamos en el lado correspondiente desechando con ello la mitad de la tabla. Después en ese intervalo de búsqueda volvemos a realizar nuevamente todo el proceso; calculamos la posición del medio, comprobamos si lo que buscamos está en esa posición, si no es así volvemos a realizar la búsqueda en el lado donde sabemos que puede estar, así hasta que nos quedamos sin intervalo de búsqueda o que la posición del medio contenga el valor buscado.

```
package ordenacion;
import java.util.Scanner;
public class Dicotomica {
 Scanner teclado=new Scanner(System.in);
 public Dicotomica(){
      int [] vEnteros={2,3,4,5};
      //Toma de consola el valor a buscar
        System.out.println("Teclee el valor a buscar : ");
        int valor = teclado.nextInt();
      //Realiza la busqueda binaria en un vector ordenado
        int indiceIzquierdo = 0;
        int indiceDerecho = vEnteros.length - 1;
        int indiceCentral = (indiceIzquierdo + indiceDerecho) / 2;
        while(vEnteros[indiceCentral] != valor && indiceIzquierdo <
       indiceDerecho) {
         if (vEnteros[indiceCentral] > valor)
            indiceDerecho = indiceCentral - 1;
          else
            indiceIzquierdo = indiceCentral + 1;
            indiceCentral = (indiceIzquierdo + indiceDerecho) / 2;
        }
      //Analiza el resultado de la búsqueda
        if (vEnteros[indiceCentral] == valor)
            System.out.println("\nEl valor: "+vEnteros[indiceCentral]+ "
            está en la posición:  "+indiceCentral);
         else
           System.out.println(valor+" no está en el vector");
 }
 public static void main(String[] args) {
      Dicotomica intefaz=new Dicotomica();
 }
}
```

ARRAYS UNIDIMENSIONALES DE OBJETOS

El tipo de los componentes de los arrays no tienen porqué ser solo tipos primitivos de datos; los componentes de los arrays pueden ser objetos.

Declaración

Lo primero que tenemos que hacer es definir la clase de la cual se generarán los objetos o se crearán las instancias; a continuación se declarará el array de la misma forma que se declara cuando sus componentes son un tipo primitivo de datos, cambiando el tipo primitivo por el nombre de la clase.

```
public class nombre_clase{

-------
}

nombre_clase[ ] nombre_array;
```

Ejm:

```
public class Empleado{
-----
}

Empleado [ ] array_objetos;
```

Creación

Los arrays se crean con el operador **new.** Cuando se crea, el valor inicial de los elementos del array es null; esto no significa que se han creado los objetos o instancias por cada posición.

```
nombre_array =new nombre_clase [nº elementos];
```

Ejm:

```
final int TAM=3;
array_objetos = new Empleado[TAM];
```

Se puede declarar y crear todo junto

```
nombre_clase [ ] nombre_array = new nombre_clase [nº elementos];
```

Ejm:

```
Empleado [ ] array_objetos= new Empleado [TAM];
```

Inicialización

1. Inicialización explícita (se da valor a las distintas posiciones del vector**)**

```
public class Persona{
        private String nombre;
        private int edad;

        public Persona (String nomb, int ed){
                nombre=nomb;
                edad=ed;
        }
        public String getNombre(){
                return nombre;
        }

        public int getEdad(){
                return edad;
        }
}
```

```
Persona[ ] vec =new Persona[2];
vec[0]= new Persona ("Jose", 20);
vec[1]= new Persona ("Ana", 40);
```

2. Inicialización implícita (se da valor a las posiciones del vector en el mismo momento de la declaración**)**

```
Persona[ ] vec={ new Persona ("Jose", 20), new Persona ("Ana", 40)};
```

Acceso a los elementos

Se han de utilizar los índices con el fin de indicar la posición del elemento al que queremos acceder + el operador punto + el miembro de la clase al que queremos hacer referencia.

nombre_array [posición] . miembro;

Recorrido

Para recorrer el vector haría falta una estructura repetitiva que permitiera pasar por todas las posiciones del array.

- **Bucle for normal**

```
Persona[ ] vec={ new Persona ("Jose", 20), new Persona ("Ana", 40)};
 for (int indice = 0; indice < vec.length; indice++)
    System.out.println (vec[indice].getNombre()+"  "+ vec[indice].getEdad());
```

- **Bucle for each o for extendido**

Facilita el recorrido sin necesidad de definir el número de elementos a recorrer. Es útil cuando tenemos que realizar recorridos completos (empezar desde la primera posición e ir hasta la última).

Sintaxis:

```
 for (tipo variable: nombre_array){
        instrucciones
}
```

Por cada elemento que se encuentre dentro de cada una de las posiciones del vector empezando por la primera, debemos ejecutar las instrucciones que se indican. La variable almacena en cada paso el valor que se visita, y solo existe durante la ejecución del bucle para desaparecer después.

Ejm:

```
Persona[ ] vec={ new Persona ("Jose", 20), new Persona ("Ana", 40)};
 for (Persona indice: vec)
    System.out.println (indice.getNombre()+"  "+ indice.getEdad());
```

Ordenación de arrays unidimensionales de objetos (vectores de objetos)

Cuando los elementos que almacena un array no son tipos primitivos de datos sino objetos pertenecientes a una clase no predefinida, para ordenarlos no es suficiente con especificar el método sort() de la clase Arrays.

En el caso de que los elementos que almacena el array son objetos pertenecientes a una clase definida por nosotros, es necesario indicar un comparador que determinará en función de que atributo o atributos se deben ordenar los objetos.

Definición de un comparador

Un comparador puede ser definido de dos formas:

- **En la misma clase que se desea comparar**

  En este caso se debe implementar la interfaz **Comparable** perteneciente al paquete java.lang, y redefinir el método **compareTo** (único método de la interfaz) dentro de la clase definida por nosotros, en el cual se compara el objeto actual con otro que se envía como argumento.

- **En otra clase separada de la clase que se desea comparar**

  En este caso se debe crear una clase y hacer que la misma implemente la interfaz **Comparator** del paquete java.util, y definir el método **compare** en el cual se comparan dos objetos enviados como argumento.

Si A y B son los objetos a comparar, tanto el método compareTo como compare, nos va a devolver un entero que será :

- Mayor a 1 si A>B
- Menor a 1 si A<B
- 0 si A=B

La clase String al ser una clase predefinida de java no necesita que se especifique nada, puesto que implementa por defecto la interface Comparable (véase la API de java). Tampoco necesita redefinir el método compareTo, puesto que este método tal y como viene definido en la API nos permite ordenar las clases predefinidas según el orden natural de los elementos, que en el caso de los String es el alfabético.

Una vez definido un comparador, el método sort() ordena el array según el criterio de comparación especificado en compareTo o compare.

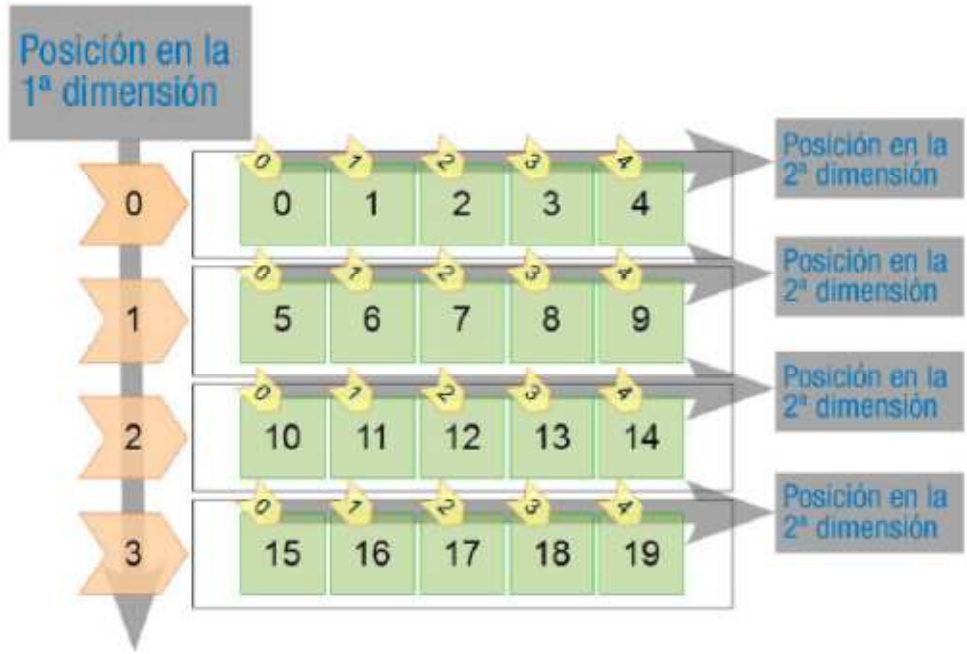
Ejm:

```
package juntasClases;
import java.util.*;
public  class Persona implements Comparable <Persona> {
      private String nombre;
      private String apellidos;
      private int edad;
      public Persona (String nomb, String apell, int edad){
            nombre=nomb;
            apellidos=apell;
            this.edad=edad;
      }
```

```
	public int compareTo (Persona p){
		int r;
		r=edad-p.edad;
		if(r==0)
			r=nombre.compareTo(p.apellidos);
			if(r==0)
				r=apellidos.compareTo(p.nombre);
		return r;
	}
	public String getNombre(){
		return nombre;
	}

	public String getApellidos(){
		return apellidos;
	}

	public int getEdad(){
		return edad;
	}
}
class Principal{
	private Scanner tecla= new Scanner(System.in);
	private final int TAM=3;
	private Persona[] vec;

	public Principal(){
		String nombre,apellido;
		int edad;
		vec=new Persona[TAM];
		for(int indice=0; indice<TAM; indice++) {
			System.out.println("Meter nombre");
			nombre=tecla.next();
			System.out.println("Meter apellido");
			apellido=tecla.next();
			System.out.println("Meter edad");
			edad=tecla.nextInt();
			vec[indice]=new Persona (nombre,apellido,edad);
		}
		Arrays.sort(vec,Collections.reverseOrder());
		visualizar(vec);
	}
```

```
    public static void visualizar(Persona [] vec){
        for(Persona elemento:vec)
            System.out.println(elemento.getNombre()+"
            "+elemento.getApellidos()+" "+elemento.getEdad());
    }

    public static void main(String[] args){
            Principal interfaz=new Principal();
    }

}
```

5.4 ARRAYS BIDIMENSIONALES O MATRICES



Declaración

Los elementos de un array bidimensional al igual que ocurre con los unidimensionales son del mismo tipo.

tipo identificador [ ] [ ];

O bien

tipo [ ] [ ] identificador;

Donde:

Tipo: es el tipo de datos de los elementos del array.

Identificador: es el nombre del array.

Se ponen tantos corchetes como dimensiones tenga el array, en este caso dos.

Ejm:

float [ ] [ ] sueldos; //declara una matriz llamada sueldos de tipo float

Creación

Se crea con el operador **new,** seguido del tipo y los corchetes, en los cuales se especifica el tamaño de cada dimensión.

Identificador=new tipo [número de filas] [número de columnas];

número de filas X número de columnas, indica el tamaño de la matriz.

Ejm:

int [ ] [ ] matriz;

matriz=new int [3] [4];

Java permite que cada fila tenga un nº diferente de columnas.

int [ ] [ ] matriz;
matriz=new matriz [3] [ ];
matriz[0]=new int[3];
matriz[1]=new int[4];
matriz[2]=new int[2];

Se puede declarar y crear todo junto

tipo [ ] [ ] identificador = new tipo [nº filas] [nº columnas];

Ejm: float [ ] [ ] sueldos = new float [100] [20];

Inicialización

Las matrices al igual que los vectores tienen posiciones o índices, siendo la primera posición la (0,0) (fila 0, columna 0).

Podemos hablar de dos tipos de inicializaciones:

1. **Inicialización explícita:** primero se hace la declaración y luego se da valor a las distintas posiciones de la matriz, indicando la posición de cada una de las dimensiones entre corchetes.

   Ejm:

   ```
   int [ ] [ ] matrix= new int [3] [4];
   matrix [0] [0] =15;
   matrix [0] [1] =21;
   ```

   Se puede dar valor a las distintas posiciones mediante una estructura repetitiva.

2. **Inicialización implícita:** se puede dar valor a las posiciones de la matriz en el mismo momento de la declaración.

   Ejm:

   ```
   int matriz[ ][ ]= {{1,2,3},{4,5,6}};
   ```

   Java determina el tamaño de la matriz en función de los valores asignados, haciendo la reserva de memoria sin tener que hacer new. En este caso se trata de una matriz de 2 filas y 3 columnas.

   No es mejor ni una forma ni otra, hay veces que es más cómodo usar un método y otras otro.

Acceso a los elementos

Para acceder a los diferentes elementos de una matriz se han de utilizar dos índices con el fin de indicar la posición del elemento al que queremos acceder (el primero indica la fila y el segundo la columna).

nombre_matriz [posFila] [posColumna];

- Java empieza a numerar las filas por el cero al igual que las columnas, por ello el primer elemento es el (0,0) y la última posición es el tamaño de la dimensión en cuestión -1.

- Para saber cuál es el tamaño de un array bidimensional se usa la propiedad length.

  - **nombre_matriz.length**: nos da el número de filas, es decir, length aplicado directamente sobre el nombre del array nos permite saber el tamaño de la primera dimensión (filas).
  - **nombre_matriz [posF].length**: nos da el número de columnas que tiene la fila posF.

Recorrido

Para recorrer una matriz hacen falta dos estructuras repetitivas, una que recorra cada una de las filas del array y la otra que recorra cada una de las columnas.

```
Scanner teclado=new Scanner(System.in);
final int FILAS    = 2;
final int COLUMNAS = 5;
int [ ] [ ] mEnteros = new int [FILAS] [COLUMNAS];
//Inicializa una matriz de enteros
for (indiceF = 0; indiceF < mEnteros.length; indiceF++)
 for (indiceC = 0; indiceC < mEnteros[indiceF].length; indiceC++)  {
            System.out.print("\nTeclee un número : ");
             mEnteros [indiceF] [indiceC] = teclado.nextInt();
 }
```

5.5 COLECCIONES

Las colecciones son almacenes de objetos dinámicos.
Las diferencias entre arrays y colecciones son:

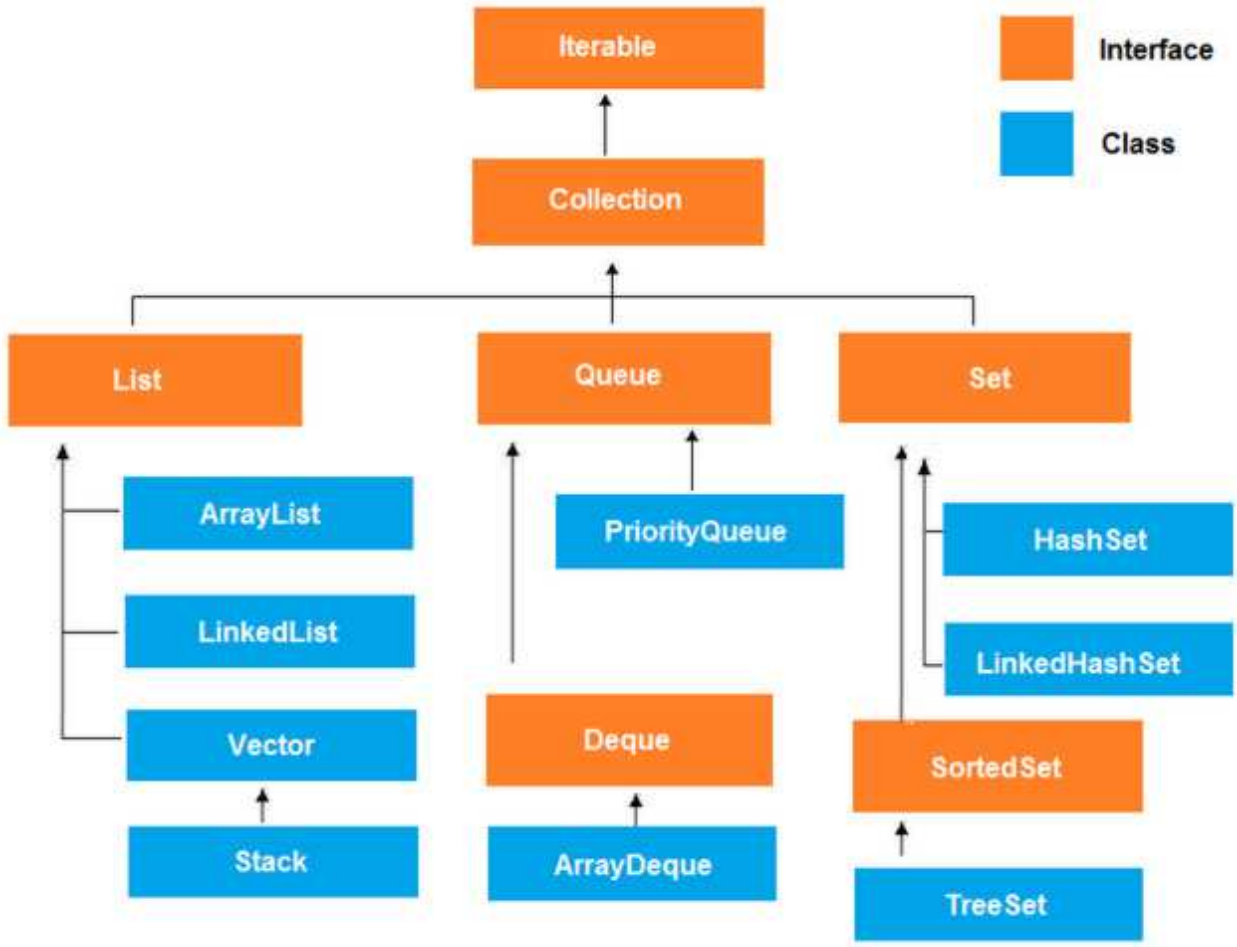
1. Las colecciones durante la ejecución del programa pueden aumentar o disminuir, es decir, se pueden insertar y eliminar elementos de forma dinámica, al contrario de los arrays que son estructuras estáticas y por tanto se debe reservar memoria al crearlos.

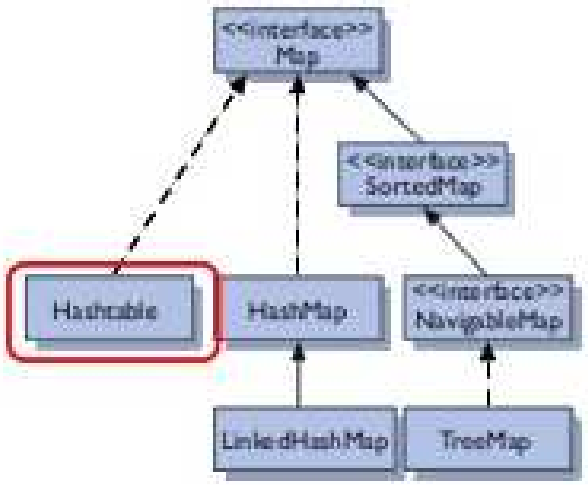
2. Sus elementos siempre son objetos por lo que no permiten almacenar tipos primitivos, en su lugar tendremos que utilizar las clases envolventes (wrapper): Integer, Double, Float.

3. En la librería de java (API) dentro del paquete java.util existen una serie de interfaces (Set, List, Queue y Map), con clases asociadas que nos permiten implementar o crear este tipo de estructuras llamadas colecciones. Esto es así porque una interface no se puede instanciar. No se puede poner:

List referencia= new List(), Set referencia= new Set()....

Todas las interfaces excepto Map heredan de Collection. Es la raíz de las interfaces, conteniendo la definición de métodos genéricos que pueden ser utilizados por las clases predefinidas.

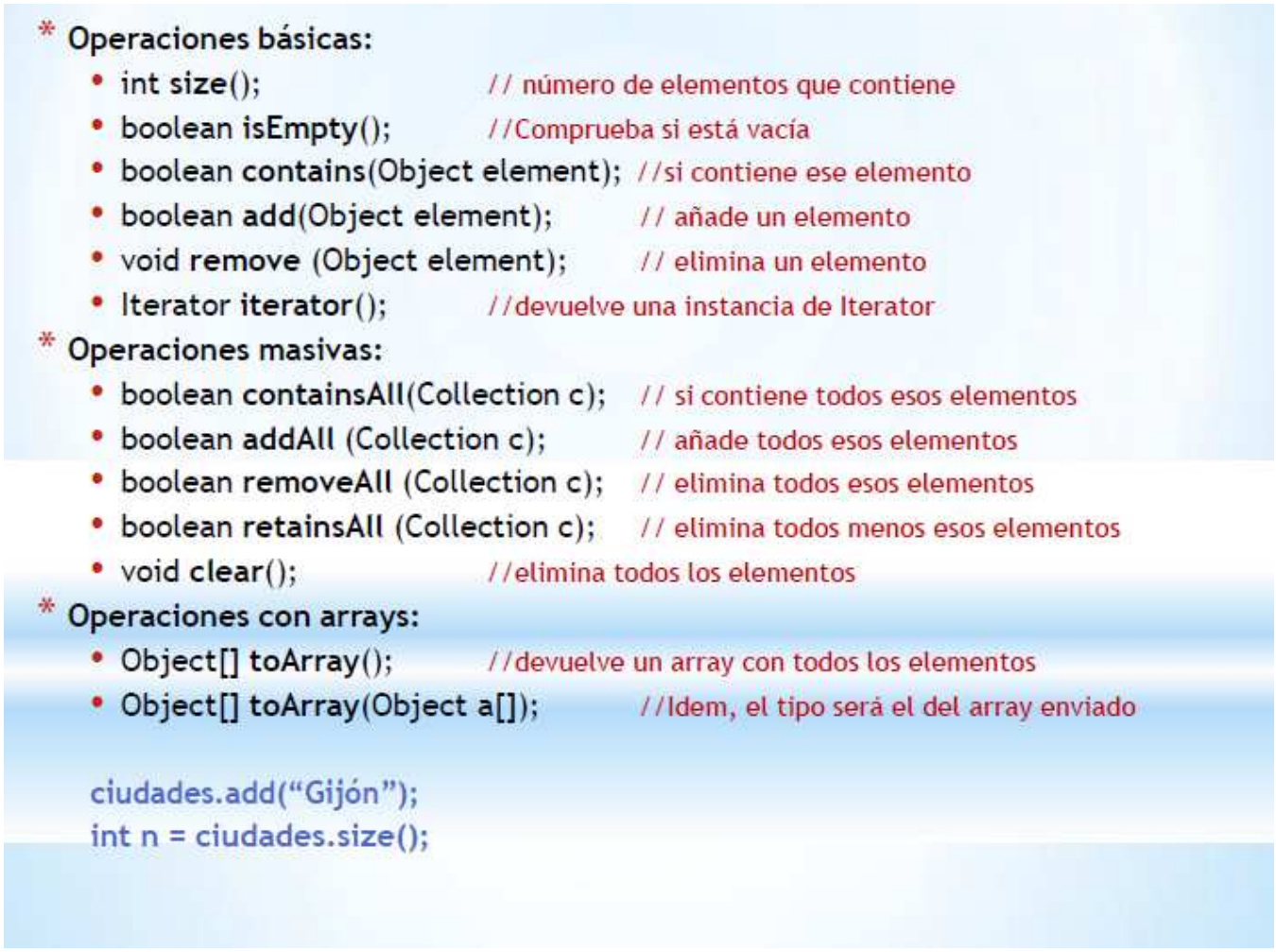




La elección de una colección u otra está en sus características. Así por ejemplo la interfaz List nos permite crear colecciones con elementos duplicados, pudiendo ser ordenados.

La interfaz Set nos permite crear colecciones sin elementos duplicados, no siempre pudiendo ordenar los mismos.

Los métodos más importantes de la interfaz Collection son los siguientes:



Declaración de una colección

TipoColeccion <tipoDato> nombre_colección;

nombre_colección =new TipoColeccion <tipoDato>();

Se puede unir las dos sentencias en una sola:

TipoColeccion <tipoDato> nombre_colección= new TipoColeccion <tipoDato>();

Ejm:

ArrayList <String> ciudades =new ArrayList <String>();

ArrayList <Integer> valor =new ArrayList <Integer>();

Cuando se declara una referencia a una colección es más conveniente definirla del tipo de su interface para hacer su uso más genérico. De esta forma se puede reutilizar la misma referencia con diferentes implementaciones de la interface (new ArrayList(), new LinkedList()).

Ejm:

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

Genéricos en las colecciones

El uso de genéricos es muy importante en las colecciones. Las colecciones admiten cualquier tipo de objetos. Cuando usamos genéricos en la declaración de una colección estaremos indicando el tipo de objetos que se puede almacenar en la misma, de tal forma que si intentamos insertar un elemento que no corresponde al tipo especificado, nos dará un error de compilación.

Ejm:

List <Alumno> clase =new ArrayList <Alumno>();

Con esta declaración estamos indicando que crearemos un arrayList cuyos elementos van a ser objetos de la clase Alumno. No habría ningún problema si añadimos esta información a dicha colección:

clase.add (new Alumno ("David",11));

Si añadimos: clase.add ("hola"), nos daría un error de compilación.

Si no hubiéramos usado genéricos, es decir, si el ArrayList lo hubiéramos declarado como:

List clase=new ArrayList();

Nos dejaría añadir cualquier tipo de objetos a la colección. Sin embargo esto conllevaría el uso frecuente de casting.

INTERFAZ LIST (colecciones con duplicados) (java.util.List)

- Permite crear colecciones con objetos repetidos.
- Se mantiene el orden en que metemos los elementos.
- Usando el método sort podemos ordenar los elementos que contiene.
- Las listas permiten acceder a un determinado elemento por su posición (indexación), para lo cual se incorporan nuevos métodos de tipo posicional.
- Podemos añadir/eliminar elementos sin restricciones.

A parte de los métodos generales heredados de Collection añade otros métodos que se adaptan a las características propias de las colecciones de tipo List.

Para crear colecciones de tipo List las clases más frecuentemente usadas son:

- **Vector:** se utiliza en concurrencia, aunque ha quedado obsoleto y hoy en día se usa la clase CopyOnWriteArrayList.

- **ArrayList:** es la implementación usada en la mayoría de situaciones. No tiene un tamaño fijo, es decir, puede variar la cantidad de elementos que la forman según las necesidades que vayan surgiendo a lo largo de la ejecución del programa, pero el espacio en memoria donde se almacenan los elementos debe ser adyacente.

  El acceso a los elementos de un ArrayList es rápido, ya que indicando su posición podemos acceder directamente a ese elemento (acceso aleatorio). La primera posición es la cero.

  Los inconvenientes de este tipo de colecciones se dan para borrar e insertar elementos por el tiempo que requieren ambas operaciones. El borrado de un elemento se realiza desplazando todos los elementos que van a continuación del que queremos borrar un lugar hacia la izquierda. La inserción de un elemento en una posición determinada se realiza moviendo todos los elementos desde esa posición un lugar hacia la derecha para dejar hueco para el nuevo objeto.

- **LinkedList:** se utiliza para listas doblemente enlazadas. Cada uno de los elementos tiene dos campos de enlace: uno apunta al objeto que va por detrás del elemento y el otro al que va por delante, por tanto no se necesita memoria contigua para almacenar los elementos.

La principal ventaja es que permite borrados e inserciones de una manera fácil, ya que lo único que se necesita es cambiar las referencias de los elementos que van delante y detrás del elemento que queremos borrar o del que queremos insertar. No es tan eficiente cuando queremos leer los datos.

Si vamos a eliminar o agregar muchos objetos a lo largo del programa debemos utilizar LinkedList, pues la diferencia de rendimiento respecto a ArrayList es enorme. Si vamos a agregar o eliminar elementos de forma poco frecuente, pero queremos acceder muchas veces a la colección, mejor usar ArrayList.

Para recorrer los elementos de la lista se puede usar el iterator o bien la interfaz ListIterator (hereda de Iterator) que permite mayores posibilidades, ya que podemos añadir y modificar objetos de la lista además de poder recorrerla en cualquier sentido.

INTERFAZ SET (colecciones sin duplicados)

- Se usa cuando queremos que no haya objetos repetidos. Para conseguirlo es absolutamente necesario sobrescribir los métodos equals y hashcode dentro de la clase a la que pertenecen los objetos que hay en dicha colección.

Con el método equals indicamos cuándo consideramos que dos objetos son iguales.

El método hashCode determina cuál es el código de identificación de cada objeto.

- No siempre se pueden ordenar los elementos. En caso de que se pueda es necesario implementar la interface Comparable y redefinir el método compareTo.
- No tiene acceso aleatorio.
- Esta interfaz contiene únicamente los métodos heredados de Collection, no define ningún método nuevo.

Ejm: si queremos almacenar los titulares de una cuenta corriente utilizaríamos este tipo de colecciones, pues los titulares de una cuenta corriente no se repiten y no tienen por qué estar ordenados.

Para crear colecciones de tipo Set las clases más frecuentemente usadas son:

- **HashSet:** su acceso a los elementos es rápido. No admite ordenación. No tiene acceso aleatorio aunque su almacenamiento lo es.

Es la implementación más equilibrada de la interfaz Set y por tanto la más empleada debido a que su rendimiento es el mejor de todos. Debe usarse si se necesita un control de duplicados pero ningún tipo de ordenación o acceso aleatorio.

Almacena los elementos en una tabla hash. Es importante definir el tamaño inicial de la tabla ya que marcará el rendimiento de esta implementación.

- **LinkedHashSet:** ordenada por orden de entrada. Eficiente al acceder a un elemento. No eficiente al agregar elementos. Debe usarse si se necesita un Set ordenado por orden de inserción o si se van a realizar operaciones solo de acceso.

- **TreeSet:** esta clase implementa la interfaz SortedSet, consiguiendo con ello que los elementos que la forman estén ordenados en función de sus valores. Los elementos almacenados deben implementar la interfaz Comparable y el método compareTo para indicar como queremos que estén ordenados los elementos de la colección, o bien hacer la ordenación por defecto según el orden natural.

Para modificar y buscar elementos es más lenta que usar un HashSet. Con inserciones se comporta muy mal. Su rendimiento es mucho peor en cualquier operación. La debemos usar solo si se necesita un Set con un criterio de ordenación específico.

INTERFAZ QUEUE

- Permite crear colas de elementos muy eficientemente.
- Solo permite acceder a elementos que están al principio o al final.
- No admite elementos aleatorios.

Para crear colecciones de tipo Queue las clases más frecuentemente usadas son:

- **ArrayDeque**
- **PriorityQueue**

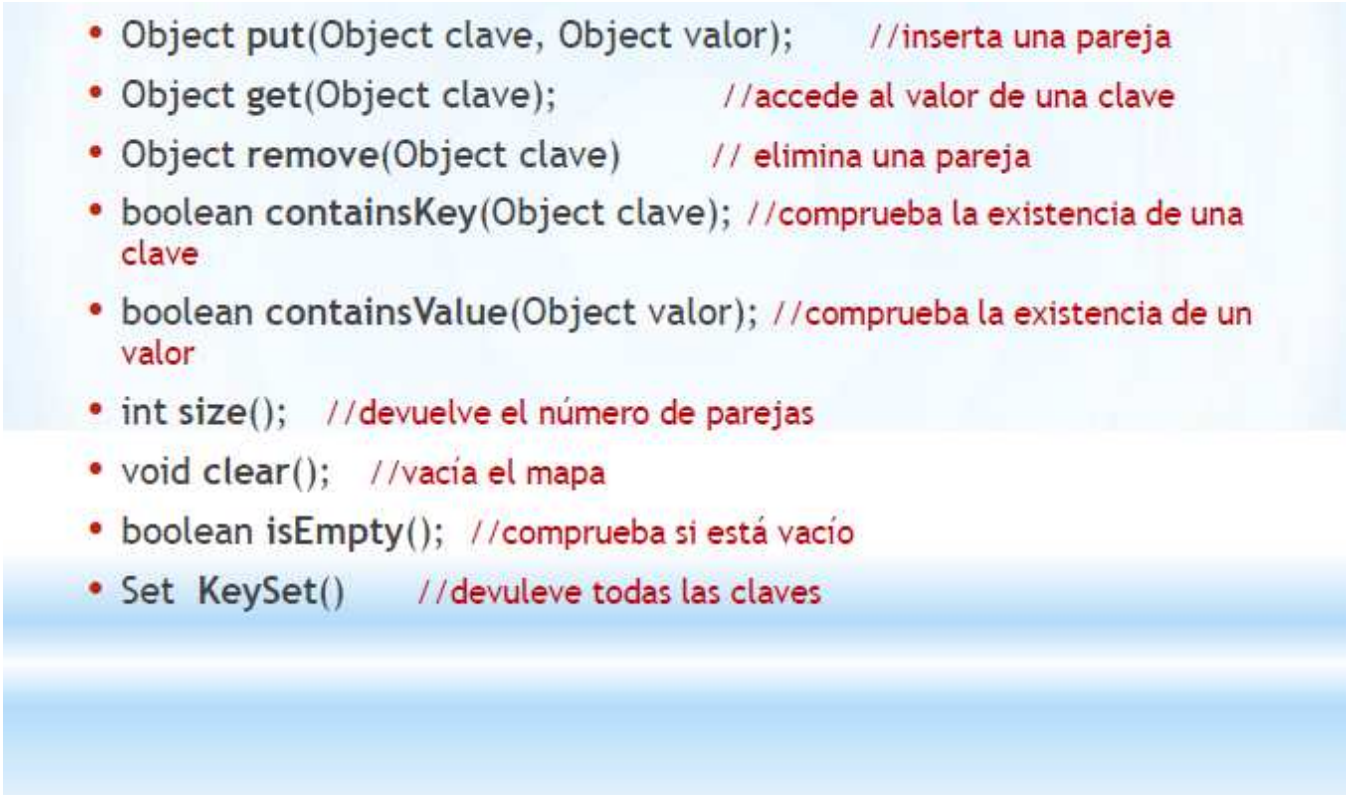
INTERFAZ MAP (java.util.Map)

- No hereda de Collection, sin embargo se utiliza para tratar colecciones de objetos.
- Permite crear una colección de elementos repetidos e indexados por clave única arbitraria, que no tiene que ser un número.
- La interfaz Map representa colecciones de valores con parejas de objetos formadas por una clave y un valor. Es decir, asocia clave a valor. Ambos pueden ser de tipo primitivo, objetos, colecciones, etc, no admitiendo claves iguales pero sí valores duplicados.
- Ofrece un tiempo de acceso óptimo a los elementos cuando dicho acceso es aleatorio.
- Es poco eficiente comparada con las demás colecciones.
- Para calcular la colocación de un elemento se utiliza el método: public int hashCode();
- Su implementación es muy parecida a los Set, debido a que internamente usan un Set para garantizar que no hay claves duplicadas.

Para crear colecciones de tipo Map las clases más frecuentemente usadas son:

- **HashMap:** no está ordenada.
- **LinkedHashMap:** ordenada por inserción. Permite ordenación por uso. Eficiente lectura pero poco eficiente la escritura.
- **TreeMap:** ordenada por clave. Poco eficiente en todas sus operaciones.
- **HashTable:** utilizado en operaciones de concurrencia. Se quedó un poco obsoleta, hoy en día se utiliza más la clase ConcurrentHashMap.

Los métodos que añade son:



Iterador (java.util.Iterator)

Para poder recorrer los elementos de una colección podemos utilizar un objeto de tipo iterator. Este objeto sirve como un cursor que se va moviendo secuencialmente por los elementos de la colección.

Todas las colecciones permiten crear el cursor por medio del método iterator() que se encuentra en la interfaz iterable de la cual heredan todas las interfaces relacionadas con colecciones:

public Iterator iterator();

Declaración y creación de un iterador

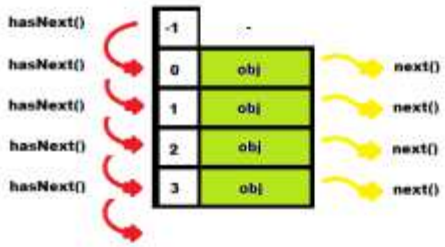
Iterator <tipoDato> nombre_iterador= nombre_colección.iterator();

Métodos de la clase Iterator

MÉTODO	DESCRIPCIÓN
boolean hasNext()	Devuelve true o false dependiendo si hay algún elemento después del objeto donde apunta el iterador. Es importante tener en cuenta que con este método el iterador no se mueve.
Object next()	Devuelve el siguiente objeto que hay después de donde apunta el iterador y avanza a dicho elemento.
void remove()	Elimina el objeto donde apunta el Iterator.

Cómo funciona el iterator

Observemos la siguiente imagen: lo que está en verde es nuestra lista.



Lo que hace iterator() es:

1. Poner el iterador al principio de la lista (Imaginariamente en la posición -1, fuera de la lista).
2. Con el método hasNext() pregunta si hay elemento siguiente.
3. Si hay un elemento debe sacarse con el método next() y volver a preguntar con el hashNext() si hay siguiente elemento. Así hasta que no exista ningún elemento siguiente.

Con el método next() obtenemos el elemento al que vamos a ir y nos situamos sobre ese elemento.

Lo anterior llevado a código quedaría de la siguiente forma:

```
Iterator <tipoDatos> it=nombre_coleccion.iterator();
while (it.hasNext())
    System.out.println (it.next());
```

- Object put(Object clave, Object valor); //inserta una pareja
- Object get(Object clave); //accede al valor de una clave
- Object remove(Object clave) // elimina una pareja
- boolean containsKey(Object clave); //comprueba la existencia de una clave
- boolean containsValue(Object valor); //comprueba la existencia de un valor
- int size(); //devuelve el número de parejas
- void clear(); //vacía el mapa
- boolean isEmpty(); //comprueba si está vacío
- Set KeySet() //devuleve todas las claves