# Estructuras de datos y algoritmos

UNIVERSIDAD DE CANTABRIA

1. Introducción
2. *Estructuras de datos lineales*
3. Estructuras de datos jerárquicas
4. Grafos y caminos
5. Implementación de listas, colas, y pilas
6. Implementación de mapas, árboles, y grafos

UNIVERSIDAD DE CANTABRIA

# 2. Estructuras de datos lineales

- 2.1 Colecciones o bolsas
- 2.2 Conjuntos
- 2.3 Listas y Vectores
- 2.4 Pilas
- 2.5 Colas
- 2.6 Colas de prioridad
- 2.7 Mapas

# 2.1 Colecciones o bolsas

UNIVERSIDAD DE CANTABRIA

**La *colección* es un ADT que permite almacenar grupos de objetos llamados *elementos***

- **pueden estar repetidos**
- **no es preciso almacenar ninguna relación de orden o secuencia**

**También se llaman *bolsas***

# Operaciones básicas de las colecciones

| operación | argumentos | retorna | errores |
|---|---|---|---|
| constructor | | Colección | |
| añade | elElemento | | elemento incompatible |
| borra | elElemento | booleano | elemento incompatible |
| hazNula | | | |
| pertenece | elElemento | booleano | elemento incompatible |
| estaVacia | | booleano | |
| tamaño | | entero | |
| iterador | | Iterador | |

# Notas:

UNIVERSIDAD DE CANTABRIA

- *constructor*: Crea la colección con cero elementos
- *añade*: Añade el parámetro **`elElemento`** a la colección. Si **`elElemento`** es incompatible con los elementos que se almacenan en esta colección lanza una excepción
- *borra*: Si existe en la colección al menos una instancia de **`elElemento`**, la borra de la colección y retorna **`true`**. En otro caso, retorna **`false`**
- *hazNula*: Elimina todos los elementos de la colección, dejándola vacía
- *pertenece*: Si existe en la colección al menos una instancia de **`elElemento`**, retorna **`true`**. En otro caso, retorna **`false`**
- *estaVacia*: Si la colección está vacía retorna **`true`**. En otro caso, retorna **`false`**
- *tamaño*: Retorna un entero que dice cuántos elementos hay en la colección
- *iterador*: Retorna un iterador asociado a la colección, para poder recorrer sus elementos

# Las colecciones en Java

UNIVERSIDAD DE CANTABRIA

**Las colecciones se representan en java por la interfaz `Collection`**

- **No existe ninguna clase que la implemente directamente**
- **Puede usarse un conjunto o lista, ya que son extensiones de ella**
  - **si queremos guardar elementos repetidos usar una lista**
- **O definir tu propia implementación**

**Los métodos que modifican la colección suelen retornar un booleano:**

- **`true`, si se ha modificado**
- **`false`, si no se ha modificado**

# La interfaz Collection

```
public interface Collection<E>
    extends Iterable<E>
{
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();
```

# La interfaz Collection (cont.)

UNIVERSIDAD DE CANTABRIA

```
    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
        //optional
    boolean removeAll(Collection<?> c);
        //optional
    boolean retainAll(Collection<?> c);
        //optional
    void clear();
       //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

# Parámetros genéricos comodín

UNIVERSIDAD DE CANTABRIA

**El carácter `?` en los parámetros genéricos se usa como comodín:**

- **`?`: para indicar cualquier clase**
- **`? extends E`: para indicar a `E`, o cualquier subclase de `E`**

# Notas:

La relación entre los métodos de la interfaz Collection y las operaciones del ADT Colección son:

| Interfaz Collection | ADT Colección |
|---|---|
| `size` | `tamaño` |
| `isEmpty` | `estaVacia` |
| `contains` | `pertenece` |
| `add` | `añade` |
| `remove` | `borra` |
| `iterator` | `iterador` |
| `clear` | `hazNula` |

El método **`add`** debe retornar **`true`**, o lanzar una excepción si no se inserta el elemento

# Constructores de las colecciones

UNIVERSIDAD DE CANTABRIA

**Aunque no aparece en la interfaz (las interfaces no tienen constructores), las clases que implementan la colección disponen de dos constructores**

- ***constructor sin parámetros*: crea la colección vacía**
- ***constructor con un parámetro* que es una colección: crea la nueva colección con una copia de los valores del argumento**

# Operaciones con colecciones

UNIVERSIDAD DE CANTABRIA

La clase `Collections` tiene muchas operaciones para manejar colecciones

- `singleton`: retorna un conjunto no modificable de un solo elemento
- `singletonList`: ídem, retornando una lista
- `singletonMap`: ídem, retornando un mapa

Dispone también de operaciones para obtener colecciones no modificables vacías:

- `emptySet`
- `emptyList`
- `emptyMap`

# Operaciones con múltiples datos

| Operación | Descripción |
|---|---|
| `containsAll()` | Devuelve `true` si todos los elementos de `c` pertenecen a la colección |
| `addAll()` | Añade todos los elementos de `c` a la colección. Retorna `true` si la colección ha cambiado |
| `removeAll()` | Elimina todos los elementos de `c` que existan en la colección, incluyendo las repeticiones. Retorna `true` si la colección ha cambiado |
| `retainAll()` | Elimina todos los elementos de la colección excepto los que estén en `c`. Retorna `true` si la colección ha cambiado |
| `clear()` | Equivale al `hazNula` |

# Operaciones con arrays

| Operación | Descripción |
|---|---|
| `Object[] toArray()` | Retorna un nuevo array de objetos de la clase `Object` que contiene todos los elementos de la colección |
| `<T> T[] toArray(T[] a)` | Igual que el anterior, pero el array es de objetos de la clase genérica `<T>`. Si el resultado cabe en `a` se devuelve ahí; si no, se devuelve un nuevo array |

**Se prefiere la segunda versión, por ser más segura**

**Ejemplos**

```
String[] a= v.toArray(new String[v.size()]);
String[] a= v.toArray(new String[0]);
```

# Ejemplos de uso

UNIVERSIDAD DE CANTABRIA

**Una aplicación de una colección es almacenar una lista de visitas a un lugar**

- una misma persona puede visitar el lugar varias veces
- no interesa el orden en el que se hacen
- interesa si alguien ha visitado el lugar o no, y cuántas veces

**Ejercicio:**

- Crear una clase capaz de almacenar en un atributo una colección de nombres de personas (Strings)
- Escribir un método para añadir una visita
- Escribir un método para saber cuántas visitas ha hecho una persona (que deje la colección igual que estaba)

# Ejemplo: libro de visitas

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;
public class LibroVisitas
{
    // lista de visitas
    private Collection<String> bolsa;

    /**
     * Constructor que deja el libro vacío
     */
    public LibroVisitas()
    {
        bolsa=new LinkedList<String>();
    }
```

# Ejemplo: libro de visitas

UNIVERSIDAD DE CANTABRIA

```
    public void anadeVisita(String nombre)
    {
        bolsa.add(nombre);
    }

    public int numeroVisitas(String nombre) {
        int contador=0;
        for(String n: bolsa) {
            if (n.equals(nombre)) {
                contador++;
            }
        }
        return contador;
    }
}
```

# 2.2 Conjuntos

UNIVERSIDAD DE CANTABRIA

**Un *conjunto* es un ADT que permite almacenar grupos de objetos llamados *elementos* de modo que**

- **no pueden estar repetidos**
- **no es preciso almacenar ninguna relación de orden o secuencia**

**Además suelen tener operaciones para operar con ellos**

- **intersección**
- **unión**
- **diferencia**

# Operaciones básicas de los conjuntos

| operación | argumentos | retorna | errores |
|---|---|---|---|
| constructor | | Conjunto | |
| añade | elElemento | booleano | elemento incompatible |
| borra | elElemento | booleano | elemento incompatible |
| hazNulo | | | |
| pertenece | elElemento | booleano | elemento incompatible |
| estaVacio | | booleano | |
| tamaño | | entero | |
| iterador | | Iterador | |

# Notas:

Las operaciones básicas son idénticas a las de las colecciones, con la excepción de *añade*:

- *constructor*: Crea el conjunto con cero elementos
- *añade*: Si **`elElemento`** ya pertenece al conjunto retorna **`false`**. En caso contrario, añade el parámetro **`elElemento`** al conjunto y retorna **`true`**. Si **`elElemento`** es incompatible con los elementos que se almacenan en este conjunto lanza una excepción
- *borra*: Si existe en el conjunto al menos una instancia de **`elElemento`**, la borra del conjunto y retorna **`true`**. En otro caso, retorna **`false`**
- *hazNulo*: Elimina todos los elementos del conjunto, dejándolo vacío
- *pertenece*: Si existe en el conjunto al menos una instancia de **`elElemento`**, retorna **`true`**. En otro caso, retorna **`false`**
- *estaVacio*: Si el conjunto está vacío retorna **`true`**. En otro caso, retorna **`false`**
- *tamaño*: Retorna un entero que dice cuántos elementos hay en el conjunto
- *iterador*: Retorna un iterador asociado al conjunto, para poder recorrer sus elementos

# Operaciones básicas de los conjuntos (cont.)

| operación | argumentos | retorna | errores |
|---|---|---|---|
| intersección | otroConjunto | Conjunto | |
| unión | otroConjunto | Conjunto | |
| diferencia | otroConjunto | Conjunto | |
| incluidoEn | otroConjunto | booleano | |

# Notas:

UNIVERSIDAD DE CANTABRIA

- *intersección*: Retorna un conjunto que contiene los elementos comunes al conjunto actual y a **`otroConjunto`**.
- *unión*: Retorna un conjunto que contiene los elementos del conjunto actual y los de **`otroConjunto`**.
- *diferencia*: Retorna un conjunto que contiene los elementos del conjunto actual que no pertenecen a **`otroConjunto`**.
- *incluidoEn*: Retorna **`true`** si todos los elementos del conjunto original pertenecen a **`otroConjunto`**, y **`false`** en caso contrario.

# Conjuntos en Java: La interfaz Set

UNIVERSIDAD DE CANTABRIA

```
public interface Set<E>
    extends Collection<E>
{
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);           //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();
```

# La interfaz Set (cont.)

UNIVERSIDAD DE CANTABRIA

```
    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
        //optional
    boolean removeAll(Collection<?> c);
        //optional
    boolean retainAll(Collection<?> c);
        //optional
    void clear();
       //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

# La interfaz Set

UNIVERSIDAD DE CANTABRIA

**Puede verse que la interfaz es idéntica a la de las colecciones**

**Las operaciones intersección, unión y diferencia se pueden realizar así:**

| Operación | Implementación |
|---|---|
| `x=a.interseccion(b)` | `x=new Set(a);`<br>`x.retainAll(b);` |
| `x=a.union(b)` | `x=new Set(a);`<br>`x.addAll(b);` |
| `x=a.diferencia(b)` | `x=new Set(a);`<br>`x.removeAll(b);` |
| `cond=a.incluidoEn(b)` | `cond=b.containsAll(a)` |

# Implementación de los conjuntos

UNIVERSIDAD DE CANTABRIA

**La implementación más usual y eficiente es `HashSet`**

- **utiliza una tabla de troceado (*hash*) para guardar los elementos**
- **es muy eficiente**
  - **las operaciones de pertenencia, inserción y borrado suelen ser O(1)**
- **no ordena los elementos**

# Las tablas de troceado

UNIVERSIDAD DE CANTABRIA

**Permiten implementar mapas y conjuntos en los que asignar, eliminar y buscar son operaciones muy eficientes**

- **el rendimiento puede degenerar, pero es muy improbable**
- **funciona incluso con elementos no comparables**

**Se basan en una conversión rápida del elemento a un valor numérico discreto, llamado *clave***

- **que sirva como índice de una tabla**
- **habitualmente se obtiene partiendo el dato original en "trozos" y haciendo operaciones con ellos**

**Debe ser compatible con `equals()`**

- **dos objetos *iguales* deben tener la misma *clave***

# Las tablas de troceado

UNIVERSIDAD DE CANTABRIA

La clase del elemento debe disponer de un método `hashCode()` para convertir un objeto de esa clase en un dato numérico llamado *clave*

```
public int hashCode()
```

El principal problema es la resolución de colisiones

- Cuando dos datos tienen la misma clave la eficiencia se reduce
- El método `hashCode()` debe distribuir las claves de modo muy homogéneo

Veremos las implementaciones más adelante

- de momento, usaremos el `hashCode()`de la clase `String`

# Tamaño de la tabla

UNIVERSIDAD DE CANTABRIA

Las tablas de troceado deben tener espacio suficiente para guardar todos los elementos con holgura

- se intenta que esté ocupada al 75% como máximo

Cuando se añaden elementos, si se supera el límite de ocupación, se *recrea* la tabla usando una más grande

- es una operación compleja (O(n))
- hay que copiar todos los elementos a la nueva tabla

La clase `HashSet` tiene un constructor al que se le pasa el tamaño inicial de la tabla (`int`)

- Si conocemos el *tamaño máximo*, podemos evitar la operación de recrear la tabla

# Ejemplo de función de troceado

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;
public class Alumno
        implements Comparable <Alumno>
{
    private String nombre;
    private String dni;

    /**
     * Troceado
     */
    public int hashCode() {
        return this.dni.hashCode();
    }
    ...
}
```

# Ejemplo de uso de los conjuntos

UNIVERSIDAD DE CANTABRIA

**En el libro de visitas del apartado anterior, añadir un método para mostrar todos los elementos duplicados (aquellos con más de una visita)**

**Hacer un método para obtener un conjunto con todas las visitas, pero sin duplicidades**

# Ejemplo (cont.)

```
/**
 * Mostrar una lista de las personas
 * duplicadas, una sola vez cada persona
 */
public void muestraDuplicadosUnaVez()
{
    // conjunto de visitas sin repeticiones
    // capacidad igual al tamaño de la bolsa/0.75
    HashSet<String> noRepetidos=
        new HashSet<String>(bolsa.size()*100/75);

    // conjunto de visitas repetidas
    // capacidad inicial sin especificar
    HashSet<String> duplicados=
        new HashSet<String>();
```

# Ejemplo (cont.)

```
        // construimos noRepetidos y duplicados
        for(String nombre: bolsa) {
            boolean esNuevo=noRepetidos.add(nombre);
            if (! esNuevo) {
                duplicados.add(nombre);
            }
        }

        // mostramos la lista de duplicados
        for (String nombre:duplicados) {
            System.out.println(nombre);
        }
}
```

# Ejemplo (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna un conjunto sin duplicados
 */
public Set<String> sinDuplicados()
{
    return new HashSet<String>(bolsa);
}
```

# 2.3 Listas y Vectores

UNIVERSIDAD DE CANTABRIA

**Una lista es una secuencia de objetos ordenados, en la que se dispone de un iterador especial, con el que se puede:**

- **insertar o eliminar elementos en cualquier posición**
- **recorrer los elementos de la lista hacia adelante y opcionalmente, hacia atrás**
- **etc.**

**Algunas listas (denominadas *Vectores*) disponen de acceso posicional eficiente, con un índice, como en las tablas**

**El orden es el que define la secuencia de elementos; no necesariamente es su orden natural**

# Operaciones básicas de las listas

| operación | argumentos | retorna | errores |
|---|---|---|---|
| constructor | | Lista | |
| hazNula | | | |
| estaVacia | | booleano | |
| tamaño | | entero | |
| iteradorDeLista | | Iterador | |

**El *iterador de lista* es un objeto que permite recorrer los elementos de la lista y operar con ella**

**Dispone de un *cursor* que se sitúa entre dos elementos: *previo* y *próximo***

# Notas:

Las operaciones básicas de las listas son:

- *constructor*: Crea la lista vacía
- *hazNula*: Elimina todos los elementos de la lista, dejándolo vacía
- *estaVacia*: Si la lista está vacía retorna `true`. En otro caso, retorna `false`
- *tamaño*: Retorna un entero que dice cuántos elementos hay en la lista
- *iteradorDeLista*: Retorna un iterador de lista asociado a la lista, para poder recorrer sus elementos

# Operaciones básicas de los iteradores de lista

| operación | argumentos | retorna | errores |
|---|---|---|---|
| añade | elElemento | | elementoIncompatible |
| borra | | | estadoIncorrecto |
| cambiaElemento | nuevoElemento | | elementoIncompatible<br>estadoIncorrecto |
| hayPrevio | | booleano | |
| hayProximo | | booleano | |
| previo | | elElemento | noExiste |
| proximo | | elElemento | noExiste |

# Notas:

Inicialmente el iterador de lista tiene el cursor situado antes del primer elemento

Las operaciones básicas que se pueden hacer con un iterador de lista son:

- *añade*: Añade **`elElemento`** a la lista entre los elementos previo y próximo, si existen; si no existen, lo añade como el único elemento de la lista. El cursor queda situado justo después del nuevo elemento.
- *borra*: borra el último elemento obtenido de la lista con **`previo`**() o **`proximo`**(). Si no hay tal elemento, lanza **`estadoIncorrecto`**.
- *cambiaElemento*: modifica el último elemento obtenido de la lista con **`previo`**() o **`proximo`**() sustituyéndolo por el **`nuevoElemento`**. Si no hay tal elemento, o si se ha llamado a **`borra`**() o **`añade`**() después de **`previo`**() o **`proximo`**(), lanza **`estadoIncorrecto`**.
- *hayPrevio*: Retorna **`true`** si existe un elemento previo al cursor y **`false`** en caso contrario.
- *hayProximo*: Retorna **`true`** si existe un elemento proximo al cursor y **`false`** en caso contrario.
- *previo*: Retorna el elemento previo al cursor, y hace que éste retroceda un elemento en la lista. Si no existe elemento previo, lanza **`noExiste`**.
- *proximo*: Retorna el elemento próximo al cursor, y hace que éste avance un elemento en la lista. Si no existe elemento próximo, lanza **`noExiste`**. Observar que si se llama a **`previo`**() después de llamar a **`proximo`**() se obtiene el mismo elemento.

# Operaciones de acceso posicional

UNIVERSIDAD DE CANTABRIA

**Hay listas llamadas *vectores*, que se usan como una tabla**

**Dependiendo de la implementación de la lista, el acceso puede ser O(1) (vectores) u O(n) (listas secuenciales)**

- **luego hay que sumarle el tiempo de la operación solicitada**

| operación | argumentos | retorna | errores |
|---|---|---|---|
| obtenElemento | índice | elElemento | índiceIncorrecto |
| cambiaElemento | índice<br>nuevoElemento | viejoElemento | índiceIncorrecto<br>elementoIncompatible |
| borra | índice | elElemento | índiceIncorrecto |
| anade | índice<br>elElemento | | índiceIncorrecto<br>elementoIncompatible |
| busca | elElemento | índice | elementoIncompatible |

# Notas:

UNIVERSIDAD DE CANTABRIA

La lista o vector se manipula como si fuese una tabla, con todos los elementos numerados con un índice que comienza en cero

Las operaciones básicas que se pueden hacer con un iterador de lista son:

- *obtenElemento*: Retorna el elemento que ocupa el **`índice`** indicado. Lanza **`indiceIncorrecto`** si el índice se refiere a una casilla inexistente. Es O(1) en un vector, y O(n) en una lista secuencial.
- *cambiaElemento*: Modifica el elemento que ocupa el **`índice`** indicado sustituyéndolo por **`nuevoElemento`**. Retorna el elemento que estaba anteriormente en esa casilla de la lista. Lanza **`indiceIncorrecto`** si el índice se refiere a una casilla inexistente. Es O(1) en un vector, y O(n) en una lista secuencial.
- *añade*: Añade **`elElemento`** a la lista en la casilla **`índice`**, moviendo ese elemento y todos los siguientes una casilla hacia adelante. Lanza **`indiceIncorrecto`** si el índice se refiere a una casilla inexistente. Es O(n) tanto en vectores como el listas secuenciales
- *borra*: Borra el elemento que ocupa la casilla índice, desplazando todos los siguientes una casilla hacia atrás. Lanza **`indiceIncorrecto`** si el índice se refiere a una casilla inexistente. Es O(n) tanto en vectores como el listas secuenciales.
- *busca*: Retorna el índice de la casilla donde se encuentra **`elElemento`**, o -1 si no se encuentra. Es O(n) tanto en vectores como el listas secuenciales.

# La interfaz List de Java

UNIVERSIDAD DE CANTABRIA

```
public interface List<E> extends Collection<E>
{
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);
```

# La interfaz List de Java (cont.)

UNIVERSIDAD DE CANTABRIA

```
    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

# Notas:

La relación entre los métodos de la interfaz **`List`** (incluyendo los heredados de **`Collection`**) y las operaciones del ADT *Lista* son:

| Interfaz List | ADT Lista |
|---|---|
| **`get`** | **`obtenElemento`** |
| **`set`** | **`cambiaElemento`** |
| **`add(i,E)`** | **`añade`** |
| **`remove`** | **`borra`** |
| **`indexOf`** | **`busca`** |
| **`listIterator()`** | **`iteradorDeLista`** |
| `clear` | `hazNula` |
| `isEmpty` | `estaVacia` |
| `size` | `tamaño` |

# Notas:

UNIVERSIDAD DE CANTABRIA

La interfaz **`List`** contiene otros métodos adicionales a los del ADT *listas*:

- **`add(E)`**: Añade el elemento indicado al final de la lista.
- **`addAll`**: Añade todos los elementos de la colección que se pasa como parámetro a partir de la casilla index, moviendo ese elemento y los posteriores hacia adelante un número de casillas igual al de elementos insertados
- **`lastIndexOf`**: Igual que **`indexOf`**(), pero buscando desde el final de la lista.
- **`listIterator(index)`**: retorna un iterador de lista, pero con el cursor situado justo antes de la casilla **`index`**.
- **`subList`**: retorna una vista parcial de la lista, entre los índices indicados, desde **`fromIndex`** inclusive hasta **`toIndex`** excluido. No es una nueva lista. La lista original no debe cambiar mientras se usa esta sublista (a no ser que el cambio se haga a través de la sublista.

# La interfaz ListIterator de Java

UNIVERSIDAD DE CANTABRIA

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

# Notas:

La relación entre los métodos de la interfaz ListIterator y las operaciones del ADT IteradorDeLista son:

| Interfaz ListIteratos | ADT IteradorDeLista |
|---|---|
| `hasNext` | `hayProximo` |
| `next` | `proximo` |
| `hasPrevious` | `hayPrevio` |
| `previous` | `previo` |
| `remove` | `borra` |
| `set` | `cambiaelemento` |
| `add` | `añade` |

Además, hay algunas operaciones en la interfaz Java que no estaban en el ADT:

- `nextIndex`: retorna el índice de la casilla próxima al cursor, o -1 si no existe
- `previousIndex`: retorna el índice de la casilla previa al cursor, o -1 si no existe

# Implementaciones de las listas

UNIVERSIDAD DE CANTABRIA

**Hay dos implementaciones de listas en las colecciones Java:**

- **`ArrayList`: Es una lista implementada en un array**
  - el acceso posicional es muy eficiente: O(1)
  - las operaciones de inserción y extracción de cualquier posición (excepto al final) son O(n)
  - necesita recrear el array, cuando se queda pequeño
  - se le puede dar el tamaño inicial
- **`LinkedList`: Es una lista implementada mediante una estructura doblemente enlazada**
  - cada elemento tiene una referencia al previo y siguiente
  - el acceso posicional es O(n)
  - la inserción y extracción con el iterador de listas es O(1)

# Ejemplos con listas

UNIVERSIDAD DE CANTABRIA

**Escribir un algoritmo que reemplace todos los elementos de una lista que coincidan con un elemento, por otro elemento**

**Escribir una clase que represente una baraja de cartas españolas, con las siguientes operaciones**

- ***constructor***: **inicialmente la baraja contiene las 40 cartas ordenadas**
- **`barajar`: para cada carta `i` desde la última hasta la segunda se intercambia esa carta con la de la casilla de índice aleatorio entre `0` e `i`**
- **`repartir`: retorna una lista que contiene las `num` últimas cartas de la baraja, y las elimina de la baraja**

**Hacer un programa de prueba para la baraja**

# Ejemplo 1

UNIVERSIDAD DE CANTABRIA

```
public static <E> void reemplazaGenerico
    (List<E> list, E valor, E nuevoValor)
{
   ListIterator<E> it = list.listIterator();
   while (it.hasNext()) {
      E elem=it.next();
      if (valor == null ? elem==null :
          valor.equals(elem))
      {
         it.set(nuevoValor);
      }
   }
}
```

# Ejemplo 2

```
import java.util.*;
public class Baraja
{
    // constantes estáticas
    private static String[] palo=
       {"Bastos", "Copas", "Oros", "Espadas"};

    private static String[] carta=
       {"As", "Dos", "Tres", "Cuatro", "Cinco",
        "Seis", "Siete", "Sota", "Caballo",
        "Rey"};

    // la baraja es una lista de cartas
    private ArrayList<String> mazo;
```

# Ejemplo 2 (cont.)

```
/**
 * Constructor de la baraja
 */
public Baraja()
{
    mazo= new ArrayList<String>(40);
    for (String p: palo) {
        for (String c: carta) {
            mazo.add(c+" de "+p);
        }
    }
}
```

# Ejemplo 2 (cont.)

```
private void intercambia(int i,int j) {
    String car=mazo.get(i);
    mazo.set(i,mazo.get(j));
    mazo.set(j,car);
}

public void barajar()
{
    Random rnd=new Random();

    for (int i=mazo.size()-1; i>=1; i--) {
       intercambia(i, rnd.nextInt(i+1));
    }
}
```

# Ejemplo 2 (cont.)

```
    /**
     * Repartir num cartas
     */
    public List<String> repartir(int num) {
        int numCartas=mazo.size();
        List<String> vistaDeMano =
            mazo.subList(numCartas - num,
                         numCartas);
        List<String> mano =
            new ArrayList<String>(vistaDeMano);
        vistaDeMano.clear();
        return mano;
    }
}
```

# 2.4 Pilas

UNIVERSIDAD DE CANTABRIA

**Una pila (o *stack*) es una lista especial en la que todos los elementos se insertan o extraen por un extremo de la lista (*LIFO*)**

**La implementación**

- **implementa estas operaciones eficientemente**
- **aprovecha las limitaciones para simplificar**

# Operaciones básicas de las pilas

| operación | argumentos | retorna | errores |
|---|---|---|---|
| constructor | | Pila | |
| apila | elElemento | | elementoIncompatible |
| desapila | | elElemento | noExiste |
| hazNula | | | |
| iterador | | Iterador | |
| cima | | elElemento | noExiste |
| estaVacia | | booleano | |
| tamaño | | entero | |

# Notas:

Las operaciones básicas que se realizan sobre una pila suelen ser las siguientes:

- *constructor*: Crea la pila con cero elementos
- *apila*: Añade el parámetro **`elElemento`** a la cima de la pila. Si **`elElemento`** es incompatible con los elementos que se almacenan en esta colección lanza una excepción
- *desapila*: Elimina de la pila el elemento que está en la cima y lo retorna. Si no hay ningún elemento, lanza una excepción
- *hazNula*: Elimina todos los elementos de la colección, dejándola vacía
- *iterador*: retorna un iterador para recorrer los elementos de la pila. No siempre se pone un iterador, sólo si es necesario
- *cima*: Retorna el elemento que está en la cima de la pila, sin eliminarlo. Si no hay ningún elemento, lanza una excepción
- *estaVacia*: Si la pila está vacía retorna **`true`**. En otro caso, retorna **`false`**
- *tamaño*: Retorna un entero que dice cuántos elementos hay en la pila

# Interfaz Java para la Pila

Las colecciones Java no disponen de ninguna interfaz específica para las pilas

Como implementación puede usarse una `LinkedList`, pues dispone de operaciones para insertar y extraer elementos por un extremo

```
public E getFirst();
public E removeFirst();
public void addFirst(E o);
```

Los dos primeros lanzan `NoSuchElementException` si la pila está vacía

# Notas:

La relación entre los métodos de la clase LinkedList y las operaciones del ADT Pila son:

| Clase LinkedList | ADT Pila |
|---|---|
| `size` | `tamaño` |
| `isEmpty` | `estaVacia` |
| `getFirst` | `cima` |
| `addFirst` | `apila` |
| `removeFirst` | `desapila` |
| `iterator` | `iterador` |
| `clear` | `hazNula` |

# Interfaz Pila propia

UNIVERSIDAD DE CANTABRIA

## La hacemos sin iterador, para simplificar

```
import java.util.*;

public interface Pila<E>
{
    void apila(E e);
    E desapila() throws NoSuchElementException;
    void hazNula();
    E cima() throws NoSuchElementException;
    boolean estaVacia();
    int tamano();
}
```

# Implementación de la Pila con LinkedList

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;
public class PilaEnlazada<E> implements Pila<E>
{
    // la pila
    private LinkedList<E> p;

    /**
     * Constructor de la pila; la crea vacía
     */
    public PilaEnlazada() {
        p=new LinkedList<E>();
    }
```

# Implementación de la Pila con LinkedList (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Añade un elemento a la pila
 */
public void apila(E e) {
    p.addFirst(e);
}

/**
 * Elimina y retorna un elemento de la pila
 */
public E desapila()
   throws NoSuchElementException
{
    return p.removeFirst();
}
```

# Implementación de la Pila con LinkedList (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Deja la pila vacía
 */
public void hazNula(){
    p.clear();
}

/**
 * Retorna el primer elemento
 */
public E cima()
   throws NoSuchElementException
{
    return p.getFirst();
}
```

# Implementación de la Pila con LinkedList (cont.)

```
    /**
     * Indica si la pila está vacía o no
     */
    public boolean estaVacia() {
        return p.isEmpty();
    }

    /**
     * Retorna el número de elementos
     */
    public int tamano(){
        return p.size();
    }
}
```

# Ejemplo del uso de las pilas

UNIVERSIDAD DE CANTABRIA

## Usar una pila para invertir el orden de una lista

# Ejemplo (cont.)

```
public static <E> void invierte(List<E> lista) {

    PilaEnlazada<E> pila=new PilaEnlazada<E>();

    // pasar los elementos de la lista a la pila
    for (E e:lista) {
       pila.apila(e);
    }
    lista.clear();

    // pasar los elementos de la pila a la lista
    while (!pila.estaVacia()) {
       lista.add(pila.desapila());
    }
}
```

# 2.5 Colas

UNIVERSIDAD DE CANTABRIA

**Una *cola* es una lista especial en la que todos los elementos se insertan por un extremo de la lista y se sacan por el otro extremo (*FIFO*)**

- **No confundir con la *cola de prioridad***

**Al igual que en las pilas, la implementación**

- **implementa estas operaciones eficientemente**
- **aprovecha las limitaciones para simplificar**

# Operaciones básicas de las colas

| operación | argumentos | retorna | errores |
|---|---|---|---|
| constructor | | Cola | |
| encola | elElemento | | elementoIncompatible |
| desencola | | elElemento | noExiste |
| hazNula | | | |
| estaVacia | | booleano | |
| tamaño | | entero | |

# Notas:

Las operaciones básicas que se realizan sobre una cola suelen ser las siguientes:

- *constructor*: Crea la pila con cero elementos
- *encola*: Añade el parámetro **`elElemento`** al extremo de inserción de la cola. Si **`elElemento`** es incompatible con los elementos que se almacenan en esta colección lanza una excepción
- *desencola*: Elimina de la cola el elemento que está en el extremo de extracción y lo retorna. Si no hay ningún elemento, lanza una excepción
- *hazNula*: Elimina todos los elementos de la cola, dejándola vacía
- *estaVacia*: Si la cola está vacía retorna **`true`**. En otro caso, retorna **`false`**
- *tamaño*: Retorna un entero que dice cuántos elementos hay en la cola

# La interfaz Queue

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

| | Operaciones que no fallan | Operaciones que pueden fallar |
|---|---|---|
| Encolar | `offer(e)` | `add(e)` |
| Desencolar | `poll()` | `remove()` |
| Consultar | `peek()` | `element()` |

# Notas:

UNIVERSIDAD DE CANTABRIA

La interfaz **`Queue`**, junto a la operación **`add`**() heredada de la colección, dispone de operaciones para encolar, desencolar, o consultar el primer elemento de la cola (el próximo que se va a desencolar). De cada una de ellas dispone de dos versiones, una que lanza una excepción si hay un fallo, y otra que retorna un valor especial si la acción solicitada no puede realizarse.

- **`offer`**(): retorna **`true`** si ha conseguido encolar el elemento y **`false`** si no (por ejemplo, si la cola es limitada, y el elemento no cabe)
- **`poll`**(): desencola y retorna un elemento si existe; si no existe, retorna **`null`**
- **`peek`**(): retorna sin desencolarlo el primer elemento si existe (el que se desencolaría con peek); si no existe, retorna **`null`**
- **`add`**(): intenta encolar el elemento **`e`**, y en caso de que no pueda (por ejemplo, si la cola es limitada, y el elemento no cabe) lanza **`IllegalStateException`**
- **`remove`**(): desencola y retorna un elemento si existe; si no existe, lanza **`NoSuchElementException`**
- **`element`**(): retorna sin desencolarlo el primer elemento si existe (el que se desencolaría con peek); si no existe, lanza **`NoSuchElementException`**

Las colas Java no deben usarse para almacenar elementos que sean **`null`**, ya que entonces los métodos **`poll`**() y **`peek`**() no funcionarían bien.

# Implementaciones de las colas

UNIVERSIDAD DE CANTABRIA

La implementación de las colas en Java es la `LinkedList`, que implementa la interfaz `Queue`

- las operaciones de inserción y extracción son O(1)

Puede ser conveniente hacer nuestra propia implementación para conseguir más sencillez y eficiencia

# Ejemplos con colas

UNIVERSIDAD DE CANTABRIA

**Escribir una clase para controlar el acceso de clientes a un servicio**

- **Se guardará una cola de espera de clientes y otra cola de clientes ya atendidos**
- **Cada cliente tiene un nombre, un número de móvil**
- **Junto al cliente se guarda su fecha y hora de llegada, y su fecha y hora de atención**

# Ejemplo con colas

UNIVERSIDAD DE CANTABRIA

## Operaciones

- añadir un cliente
- atender a un cliente
- obtener el tiempo medio de espera de los clientes que aún no han sido atendidos
- obtener el tiempo medio de espera de los clientes ya atendidos
- mostrar el estado de las colas

## Escribir también un programa de prueba

## Para la fecha y hora usar la clase predefinida `Calendar`

# Ejemplo: cola de espera

UNIVERSIDAD DE CANTABRIA

```
import java.util.Calendar;

/**
 * Clase que permite obtener la fecha y hora actual,
 * en milisegundos desde la época
 */
public class Reloj
{
   public static long ahora()
   {
      return Calendar.getInstance().getTimeInMillis();
   }
}
```

# Ejemplo: cola de espera

```
import java.util.*;
public class ColaEspera {
  /** Clase interna para almacenar todos los
   *  datos de un cliente
   */
  private static class DatosCliente {
     Cliente c;
     long entrada, salida; // milisegundos

     /** Constructor; pone la hora de entrada*/
     DatosCliente (Cliente c) {
        this.c=c;
        entrada=Reloj.ahora();
     }
```

# Ejemplo: cola de espera (cont.)

```
    void atiende() {
        salida=Reloj.ahora();
    }
}

// colas del servicio
private Queue<DatosCliente> colaEspera;
private Queue<DatosCliente> colaAtendidos;

/**Constructor de ColaEspera */
public ColaEspera() {
  colaEspera=new LinkedList<DatosCliente>();
  colaAtendidos=new
     LinkedList<DatosCliente>();
}
```

# Ejemplo: cola de espera (cont.)

```
/**
 * Nuevo cliente; se mete en la cola de espera
 */
public void nuevoCliente(Cliente c)
{
        DatosCliente datos=new DatosCliente(c);
        colaEspera.add(datos);
}
```

# Ejemplo: cola de espera (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Atender cliente: se saca de la cola de
 * espera y se mete en la de atendidos;
 * retorna el cliente atendido
 */
public Cliente atenderCliente()
  throws NoSuchElementException
{
      DatosCliente datos=colaEspera.remove();
      datos.atiende();
      colaAtendidos.add(datos);
      return datos.c;
}
```

# Ejemplo: cola de espera (cont.)

UNIVERSIDAD DE CANTABRIA

```
public double tiempoEsperaNoAtendidos()
{
    long tiempo=0;
    int num=0;
    long ahora=Reloj.ahora();
    for (DatosCliente datos: colaEspera) {
       tiempo=tiempo + ahora-datos.entrada;
       num++;
    }
    if (num==0) {
       return 0.0;
    } else {
       return (((double) tiempo)/num)/1000.0;
    }
}
```

# Ejemplo: cola de espera (cont.)

UNIVERSIDAD DE CANTABRIA

```
   public double tiempoEsperaAtendidos()
   {
       long tiempo=0;
       int num=0;
       for (DatosCliente datos: colaAtendidos) {
         tiempo=tiempo+datos.salida-datos.entrada;
         num++;
       }
       if (num==0) {
         return 0.0;
       } else {
         return (((double) tiempo)/num)/1000.0;
       }
   }
}
```

# 2.6 Colas de prioridad

UNIVERSIDAD DE CANTABRIA

**Las colas de prioridad permiten alterar el orden de salida de los elementos de una cola**

- **no es necesario seguir un orden FIFO**
- **el orden se puede basar en una función de comparación**

**Las operaciones de las colas de prioridad son las mismas que las de las colas**

- **con un comportamiento diferente**

**La interfaz `Queue` de Java se puede usar también para colas de prioridad**

# Implementación de colas de prioridad

UNIVERSIDAD DE CANTABRIA

**En Java existe la clase `PriorityQueue` que**

- **implementa la interfaz `Queue`**
- **se comporta como una cola de prioridad**
- **usa `compareTo`, o un *comparador*, para ordenar los elementos por su prioridad**
  - **los ordena de menor a mayor (sale primero el menor elemento)**
  - **para los que son iguales, el orden es arbitrario**
- **usa para su implementación un *montículo binario*, consiguiendo así operaciones con eficiencia O(log n) o mejor**

# Constructores de PriorityQueue

UNIVERSIDAD DE CANTABRIA

**`PriorityQueue()`**

- Crea la cola para ordenar sus elementos con **`compareTo()`**

**`PriorityQueue(Collection<? extends E> c)`**

- Crea la cola para ordenar sus elementos con **`compareTo()`**, y conteniendo inicialmente los elementos de la colección **`c`**

**`PriorityQueue(int initialCapacity)`**

- Crea la cola para ordenar sus elementos con **`compareTo()`**, y con la capacidad inicial indicada

# Constructores de PriorityQueue

```
PriorityQueue(int initialCapacity,
              Comparator<? super E> comparator)
```

- Crea la cola con la capacidad inicial indicada, y para ordenar sus elementos de acuerdo con el comparador indicado

# Ejemplo con cola de prioridad

UNIVERSIDAD DE CANTABRIA

**Escribir una clase para controlar el acceso de clientes a un servicio con varios grados de urgencia**

- Se guardará una cola de espera de clientes, ordenados por prioridad
- Cada cliente tiene un nombre, un número de móvil
- Junto al cliente se guarda su urgencia

**Operaciones**

- añadir un cliente
- atender a un cliente
- mostrar el estado de las colas

**Escribir también un programa de prueba**

# Ejemplo con cola de prioridad (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
   *  Clase enumerada que representa la
   *  urgencia de un cliente
   */
public enum Urgencia
{
    baja, media, alta
}
```

# Ejemplo con cola de prioridad (cont.)

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;
/public class ColaEsperaConUrgencia
{
  /**
   *  Clase interna para almacenar los datos
   *  de un cliente con urgencia
   */
  private static class DatosCliente
     implements Comparable <DatosCliente>
  {
     Cliente c;
     Urgencia urg;
```

# Ejemplo con cola de prioridad (cont.)

UNIVERSIDAD DE CANTABRIA

```
    /**
     * Constructor de DatosCliente
     */
    DatosCliente (Cliente c, Urgencia urg) {
        this.c=c;
        this.urg=urg;
    }

    /*
     * Comparación de clientes por su urgencia
     */
    public int compareTo(DatosCliente otro) {
        return this.urg.compareTo(otro.urg);
    }
} // final de la clase DatosCliente
```

# Ejemplo con cola de prioridad (cont.)

UNIVERSIDAD DE CANTABRIA

```
// cola del servicio
private Queue<DatosCliente> colaEspera;


/**
 * Constructor de ColaEspera
 */
public ColaEsperaConUrgencia()
{
    colaEspera=new
       PriorityQueue<DatosCliente>();
}
```

# Ejemplo con cola de prioridad (cont.)

```
/**
 * Nuevo cliente; se mete en la cola de espera
 */
public void nuevoCliente
    (Cliente c, Urgencia urg)
{
    DatosCliente datos=new DatosCliente(c,urg);
    colaEspera.add(datos);
}

/**
 * Atender cliente: se saca de la cola de
 * espera; retorna el cliente atendido
 */
```

# Ejemplo con cola de prioridad (cont.)

UNIVERSIDAD DE CANTABRIA

```
public Cliente atenderCliente()
    throws NoSuchElementException
{
    DatosCliente datos=colaEspera.remove();
    return datos.c;
}

/**
 * Mostrar el estado de la cola de espera
 */
public void muestraEstado() {
  System.out.println();
  System.out.println("--Estado de la cola--");
  System.out.println("No atendidos "+
                     colaEspera.size());
```

# Ejemplo con cola de prioridad (cont.)

```
      for (DatosCliente datos:colaEspera) {
        System.out.println(datos.c+" urgencia:
                                "+datos.urg);
      }
   }

   /**
    * Num clientes en espera
    */
   public int numClientesEnEspera() {
      return colaEspera.size();
   }
}
```

# 2.7 Mapas

UNIVERSIDAD DE CANTABRIA

Un *mapa* es una función de unos elementos de un tipo *clave*, en otros elementos de un tipo *destino*

Es una generalización del *array*

- el array es una tabla de valores en la que el índice es un entero
- el mapa es una tabla de valores en la que el índice es de cualquier tipo (la *clave*)

El mapa no puede tener claves repetidas

- pero dos claves pueden referirse al mismo valor

Las claves deben ser objetos inmutables (no cambiar)

# Operaciones básicas de los mapas

| operación | argumentos | retorna | errores |
|---|---|---|---|
| constructor | | Mapa | |
| asignaValor | clave, valor | | |
| obtenValor | clave | valor | NoExiste |
| borra | clave | valor | NoExiste |
| hazNulo | | | |
| contieneClave | clave | booleano | |
| tamaño | | entero | |
| estaVacio | | booleano | |

# Notas:

UNIVERSIDAD DE CANTABRIA

Un 'mapa' M es una función de unos elementos de un tipo clave, en otros elementos de otro (o el mismo) tipo denominado destino. Es una generalización del concepto de array para manejar índices (claves) no necesariamente enteras. Se representa como M(clave)=valor.

En ocasiones el mapa se puede indicar a través de una expresión matemática. Sin embargo en otras ocasiones es preciso almacenar para cada posible clave el valor de M(clave). En este caso se puede utilizar el tipo abstracto de datos denominado 'mapa', para el que se definen las siguientes operaciones:

- *constructor*: Crea el mapa con cero relaciones.
- *asignaValor*: Asigna a la clave indicada, el valor dado.
- *obtenValor*: Retorna el valor asociado a la clave indicada, o lanza un error si para esa clave no existe valor asociado.
- *borra*: Borra la clave indicada y su valor asociado del mapa, o lanza un error si esa clave no está en el mapa
- *hazNulo*: Borra todas las relaciones del mapa, dejándolo vacío
- *contieneClave*: Retorna *true* si el mapa contiene la clave indicada, y *false* en caso contrario
- *tamaño*: Retorna el número de relaciones clave-valor válidas que existen en el mapa
- *estaVacio*: Retorna *true* si el mapa no tiene relaciones, y *false* en caso contrario.

# La interfaz Map

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

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

    // Bulk operations
    void putAll(Map<? extends K,
                    ? extends V> m);
    void clear();
```

# La interfaz Map (cont.)

```
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

**Hay dos constructores, uno que crea un mapa vacío y otro que crea una copia de otro mapa**

# Notas:

La interfaz **`Map`** define las operaciones que pueden realizarse con los mapas en Java. La relación entre las operaciones abstractas de los mapas y los métodos de la interfaz **`Map`** se indica en la siguiente tabla

| Interfaz Map | ADT Mapa | Comentario |
|---|---|---|
| **`put`** | *asignaValor* | retorna el valor anterior asignado a la clave, si lo hay, o **`null`** si no lo hay |
| **`get`** | *obtenValor* | retorna el valor asignado a la clave, si lo hay, o **`null`** si no lo hay |
| **`remove`** | *borra* | retorna el valor asignado a la clave, si lo hay, o **`null`** si no lo hay |
| **`clear`** | *hazNulo* | |
| **`containsKey`** | *contieneClave* | funciona incluso aunque el valor asociado a la clave sea **`null`** |
| **`size`** | *tamaño* | |
| **`isEmpty`** | *estaVacio* | |

# "Vistas" del mapa

UNIVERSIDAD DE CANTABRIA

Es posible ver el mapa desde diversos puntos de vista, para iterar sobre él

- `keySet`(): retorna una vista del mapa como un conjunto de claves
- `values`(): retorna una vista del mapa como una colección de valores del tipo destino (posiblemente repetidos)
- `entrySet`(): retorna la vista del mapa como un conjunto de parejas *clave-valor*; son objetos de la clase interna `Entry`

Las vistas admiten las operaciones del conjunto o colección correspondiente, excepto `add`()

- se puede *iterar*
- se puede *borrar*: la relación se borra del mapa

# Notas:

UNIVERSIDAD DE CANTABRIA

Además, el mapa tiene las siguientes operaciones adicionales:

- **`containsValue`**(): Retorna *true* si el mapa contiene el valor indicado, al menos una vez, y *false* en caso contrario
- **`putAll`**(): Añade al mapa todas las relaciones del mapa **`m`**. el efecto es como invocar **`put`**() para cada pareja clave valor del mapa **`m`**.
- **`ketSet`**(): retorna una vista del mapa como un conjunto de claves
- **`values`**(): retorna una vista del mapa como una colección de valores, posiblemente repetidos
- **`entrySet`**(): retorna una vista del mapa como un conjunto de parejas clave-valor, de la clase interna **`Entry`**.

El método **`entrySet`**() retorna un objeto de la interfaz interna **`Entry`**, que representa una pareja clave-valor. Dispone de los siguientes métodos

- **`getKey`**(): retorna la clave
- **`getValue`**(): retorna el valor
- **`setValue`**(): cambia el valor haciéndolo igual a value, y retorna el valor anterior. El cambio se refleja en el mapa original.

Mientras se usa una vista el mapa no debe cambiar, excepto a través de esa vista.

# Implementaciones de los mapas

UNIVERSIDAD DE CANTABRIA

**Las implementaciones generales de los mapas en Java son:**

- **`HashMap`: mapa implementado mediante una tabla de troceado**
  - **operaciones comunes son habitualmente O(1)**
- **`TreeMap`: mapa implementado mediante un árbol binario**
  - **operaciones comunes son O(log n)**
  - **los datos se ordenan por su clave**
- **`LinkedHashMap`: mapa implementado mediante una tabla de troceado (tabla *hash*) en la que el orden de inserción se almacena además mediante una lista *doblemente enlazada***
  - **operaciones comunes son habitualmente O(1)**
  - **ordenación por el orden de inserción de las claves**

# Ejemplos con mapas

UNIVERSIDAD DE CANTABRIA

**1. Crear una agenda que relacione nombres de personas con su número de teléfono**

- **probar las diferentes vistas del mapa para iterar sobre él de varias formas**

**2. Implementar un diccionario**

# Ejemplo: agenda telefónica

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;

public class AgendaTelefonica
{
    // la agenda se guarda en un mapa
    private Map<String,Integer> agenda;

    /**
     * Constructor que deja la agenda vacía
     */
    public AgendaTelefonica()
    {
        agenda = new HashMap<String,Integer>();
    }
```

# Ejemplo: agenda telefónica (cont.)

```
/**
 * Añadir un nombre con su teléfono
 */
public void anadeTelefono
  (String nombre, int telefono)
{
    agenda.put(nombre,
               new Integer(telefono));
}

/**
 * Consultar un nombre; retorna el telefono,
 * o 0 si el nombre no existe
 */
```

# Ejemplo: agenda telefónica (cont.)

```
public int consulta (String nombre) {
    Integer tel= agenda.get(nombre);
    if (tel==null) {
        return 0;
    } else {
        return tel.intValue();
    }
}

/**
 * Saber si un nombre esta en el diccionario
 */
public boolean estaIncluido(String nombre) {
    return agenda.containsKey(nombre);
}
```

# Ejemplo: agenda telefónica (cont.)

```
    /**
     * Mostrar la lista de toda la agenda
     */
    public void mostrarNumeros() {
       Set<Map.Entry<String,Integer>>
           lista=agenda.entrySet();
       System.out.println();
       System.out.println("Nombre - Telefono:");
       for (Map.Entry<String,Integer> e:lista) {
         System.out.println(e.getKey()+" - "
                            +e.getValue());
       }
    }
}
```

# Mapa para tipos enumerados

UNIVERSIDAD DE CANTABRIA

**La clase `EnumMap` es una implementación especial de los mapas adaptada a claves de un tipo enumerado**

**Tiene la misma eficiencia que un array**

- **internamente es un array**
- **permite manejar los enumerados con fiabilidad**
- **mantienen la ordenación natural de las claves**

# Ejemplo con EnumMap

UNIVERSIDAD DE CANTABRIA

**Escribir una clase que relacione cada dia de la semana con un objeto de la clase `Menu`, que representa el menú de la comida**

- **El dia de la semana es un objeto de una clase enumerada**

# Ejemplo con EnumMap (cont.)

UNIVERSIDAD DE CANTABRIA

```
public enum DiaSemana
{
    lunes, martes, miercoles, jueves, viernes,
    sabado, domingo
}

public class Menu
{
    // atributos
    private String primerPlato;
    private String segundoPlato;
    private String postre;
```

# Ejemplo con EnumMap (cont.)

```
    public Menu(String primerPlato,
        String segundoPlato, String postre)
    {
        this.primerPlato=primerPlato;
        this.segundoPlato=segundoPlato;
        this.postre=postre;
    }

    public String toString()
    {
        return "1º: "+primerPlato+", 2º: "+
               segundoPlato+", postre: "+postre;
    }

}
```

# Ejemplo con EnumMap (cont.)

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;

public class MenuSemanal
{
    // mapa de menús
    private EnumMap<DiaSemana,Menu> menu;

    /**
     * Constructor
     */
    public MenuSemanal()
    {
        menu=new EnumMap<DiaSemana,Menu>
            (DiaSemana.class);
    }
```

# Ejemplo con EnumMap (cont.)

```
    /** poner el menú de un día concreto
     */
    public void ponMenu(DiaSemana dia,
            Menu comida)
    {
        menu.put(dia,comida);
    }

    /**
     * Consultar el menú de un día
     */
    public Menu consultaMenu(DiaSemana dia) {
        return menu.get(dia);
    }
}
```