UNIDAD 6: COLECCIONES

VERSION 1.2
(29/02/2016 23:13)

CONTENIDOS

6.1 INTRODUCCION

Una **colección** es un objeto que representa un grupo de objetos (por ejemplo la clásica clase Vector).

Las colecciones son estructuras de tamaño flexible ya que es posible añadir y eliminar elementos de una colección una vez creada. Esto es una ventaja frente a las arrays que eran estructuras de tamaño fijo.

Cuando escribimos programas necesitamos con frecuencia agrupar objetos en colecciones (un instituto que mantiene un registro de los alumnos matriculados, una agenda electrónica que permite guardar números de teléfono, una librería que almacena libros, ...). Es habitual, además, que el número de elementos en la colección varíe, que haya que añadir o borrar elementos. Históricamente existen estructuras dinámicas que se usan en muchos lenguajes de programación como son:

- Arboles
- Colas
- Conjuntos
- Listas
- Pilas
- Tablas de hash

En Java disponemos de clases que nos van a permitir crear y usar ese tipo de estructuras dinámicas de una forma sencilla y eficiente. Las colecciones en Java están organizadas en un marco de trabajo llamado: Java Collections Framework (ver apartado 6.9.1 del anexo en la página 19)

6.2 CLASES ENVOLVENTES (WRAPPER)

Las colecciones contienen objetos, pero si queremos tener una colección que almacene valores de tipo primitivo como números enteros (**int**) necesitamos “envolver” el dato de tipo primitivo dentro de un objeto.

Para eso existen las clases envolventes (**wrapper**). Para cada tipo primitivo disponemos de una clase envolvente que nos permitirá crear un objeto que contenga el dato de tipo primitivo y por tanto tener una colección de objetos por ejemplo de números enteros.

Tipo primitivo	Clase envolvente (wrapper)
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Todas las clases envolventes numéricas son subclases de la clase abstracta `Number` del paquete `java.lang`

Los objetos de clases envolventes son **inmutables** (una vez creados no es posible modificar el valor que contienen). No tienen métodos mutadores.

Debido a que es muy habitual la conversión entre tipos primitivos y clases envolventes, Java proporciona de mecanismos automáticos para dichas conversiones: **autoboxing** y **unboxing**.

6.2.1 AUTOBOXING

Autoboxing es la conversión automática que realiza el compilador de Java de un tipo primitivo a su correspondiente clase envolvente (**wrapper**). Por ejemplo, conversión de int a Integer, un double a Double, etc. Si la conversión se realiza en el sentido contrario se denomina **unboxing**.

Ejemplo 1

```
Character c = 'a';
Integer i   = 2;
Boolean b   = true;
Double d    = 2.5;
```

Ejemplo 2

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 10; i += 2)
    li.add(i);
```

El código del ejemplo anterior (Ejemplo 2) funciona ya que el compilador realiza un proceso de **autoboxing** creando automáticamente un objeto `Integer` para contener el valor de tipo primitivo i.

Otra forma de hacerlo sería mediante el código del ejemplo siguiente (Ejemplo 3), pero en este caso no hay **autoboxing**.

Ejemplo 3

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 10; i += 2)
    li.add(Integer.valueOf(i));
```

El compilador de Java aplica **autoboxing** cuando un valor de tipo primitivo:

- Se pasa como parámetro a un método que espera un objeto de la clase envolvente (**wrapper**) correspondiente
- Se asigna a una variable de la clase envolvente correspondiente

6.2.2 UNBOXING

Es la conversión de un objeto de clase envolvente (**wrapper**) su correspondiente valor de tipo primitivo.

El compilador de Java aplica **unboxing** cuando un objeto de clase envolvente:
- Se pasa como parámetro a un método que espera un valor del correspondiente tipo primitivo
- Se asigna a una variable del correspondiente tipo primitivo

Ejemplo 4

```
import java.util.ArrayList;
import java.util.List;

public class Unboxing {

    public static void main(String[] args) {
        Integer i = -8;                          // (1)

        // Unboxing en paso de parámetros
        int va = valorAbsoluto(i);
        System.out.println("valor absoluto de " + i + " = " + va);

        List<Double> ld = new ArrayList<>();
        ld.add(Math.PI);                         // (2)

        // Unboxing en asignación
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int valorAbsoluto(int i) {
        return (i < 0) ? -i : i;
    }
}
```

Actividad 6.1 Analizar y probar el código del ejemplo anterior (Ejemplo 4). ¿Qué tipo de conversión automática se da en los casos (1) y (2)?

Actividad 6.2 Analizar y probar el código siguiente para ver si se da **autoboxing** o **boxing**.

```
public static int sumarPares(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
            sum += i;
        return sum;
}
```

6.3 LISTAS

6.3.1 LISTAS EN GENERAL

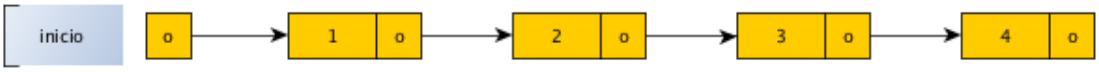
Las listas son secuencias de elementos donde cada elemento tiene un enlace al elemento siguiente. Puede contener elementos duplicados.

Son estructuras dinámicas ya que pueden crecer y decrecer en la memoria.

Cada elemento de la lista se suele representar por una clase nodo que contiene un dato (o varios) y una referencia (o dos en el caso de las listas doblemente enlazadas) al siguiente nodo de la lista.

Hay dos tipos principales de listas:

- **Simplemente enlazadas**: cada nodo tiene una referencia al siguiente. Sólo se pueden recorrer del primer al último elemento. Solo disponemos de una referencia al primer elemento de la lista para poder recorrerla.



- **Doblemente enlazadas**: cada nodo tiene dos referencias. Una al siguiente nodo de la lista y otra referencia al nodo anterior de la lista. Disponemos de dos referencias para recorrer la lista: primer nodo (para recorrer la lista en el sentido natural) y último nodo (para recorrer la lista en sentido inverso).

Sobre una lista se pueden realizar (al menos) las siguientes operaciones: agregar, insertar, borrar, buscar, filtrar.

Ventajas: es posible insertar y eliminar elementos y suelen ser operaciones rápidas. En los arrays no era posible.

Desventajas: las listas en general no tienen un índice como los arrays para acceder a cada elemento, por tanto, el acceso a cada elemento es lento como media al tener que recorrer todos los elementos anteriores hasta llegar al elemento buscado.

Actividad 6.3	Sin usar las clases proporcionadas por JCF, crea manualmente una lista simplemente enlazada (ListaSE) de números enteros que permita las siguientes operaciones:
void agregar(nuevo)	Añade un nuevo nodo al final de la lista. Este método estará sobrecargado para recibir un número entero o bien un nodo.
void insertar(nuevo)	Inserta un nodo al principio de lista
boolean insertarAntes(nuevo,nodo)	Inserta el nuevo nodo antes del nodo que recibe
boolean insertarDespues(nuevo,nodo)	Inserta el nuevo nodo después del nodo que recibe
boolean buscar(nodo)	Buscar un nodo en la lista (equals)
int contar(nodo)	Devuelve cuantos nodos hay en la lista iguales al buscado (equals)
boolean borrar(nodo)	Borra el nodo (equals)
void borrar()	Borra todos los nodos de la lista
ListaSE filtrar(nodo)	Devuelve una nueva lista que contiene todos los nodos de la lista original iguales al nodo recibido
int get(posicion)	Devuelve el elemento que está en la posición indicada en la lista (primera posición = 0)

6.3.2 INTERFACE LIST

La interface List es miembro de la JCF.

```
public interface List<E>
extends Collection<E>
```

Las listas en Java son colecciones donde los elementos están dispuestos en secuencia.

Esta interface List dispone de métodos para acceder a los elementos de la lista por su posición usando un índice entero (comenzando en 0).

Es posible también buscar elementos en la lista (equals).

A diferencia de los conjuntos, las listas permiten elementos duplicados (equals).

Pueden admitir elementos nulos.

Este interface proporciona un iterador especial llamado ListIterator que permite la inserción y modificación de elementos, además de acceso bidireccional aparte de las operaciones típicas que proporciona el interface Iterator. Hay un método para obtener un iterador que comience en una posición específica de la lista.

Hay métodos para insertar y eliminar varios elementos en un punto determinado de la lista.

Algunas listas tienen restricciones sobre los elementos que pueden contener. Por ejemplo, algunas pueden prohibir los elementos nulos, y otras tienen restricciones sobre el tipo de sus elementos. Si intentamos añadir elementos incorrectos se pueden producir excepciones como:

```
NullPointerException
ClassCastException
```

Algunas clases que implementan la interface List son:

```
ArrayList
LinkedList
*Stack
Vector
```

(*) Desaconsejada: para implementar estructuras de pila (LIFO), Oracle aconseja usar las clases que implementan la interface Deque como ArrayDeque o LinkedList en vez de la clase Stack

Actividad 6.4	Buscar en la API de Java si la clase LinkedList tiene los métodos de la clase Stack

6.3.3 ARRAYLIST

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

La clase ArrayList es una colección que implementa una lista de objetos.

- Permite almacenar un número arbitrario de elementos, cada uno de ellos es un objeto (o bien null). Las colecciones guardan objetos, no valores de tipo primitivo a menos que éstos se envuelvan en un objeto de clase envolvente (*wrapper*).
- Crece/decrece automáticamente a medida que los elementos se añaden/borran
- Mantiene un contador privado que indica cuántos elementos tiene la colección
- Está en el paquete java.util
- Cada elemento se añade al final
- Es posible acceder a los elementos de manera directa usando un índice que comienza en 0
- Los elementos se almacenan de manera contigua en la memoria
- La colección tiene una capacidad inicial que crecerá automáticamente cuando se añadan más elementos que la capacidad actual

Esta implementación es no sincronizada. Si varios hilos acceden a un objeto ArrayList concurrentemente, y al menos uno de los hilos modifica la lista estructuralmente, debe ser sincronizado externamente. Una modificación estructural es cualquier operación que añada o elimine uno o más elementos, e explícitamente redimensione el array de respaldo; la mera modificación del valor de un elemento no es una modificación estructural.

La sincronización se consigue típicamente sincronizando un objeto que encapsule (contenga) un objeto ArrayList. Si tal objeto no existe, entonces las lista debe ser envuelta (**wrapped**) usando el método:

```
Collections.synchronizedList
```

Es preferible hacer esto en el momento de la creación de la lista para evitar accesos no sincronizados accidentales a la lista. Por ejemplo:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

6.3.3.1 CONSTRUCTORES

`ArrayList()`

Crea un objeto de la clase ArrayList vacío (sin elementos) pero con una capacidad inicial por defecto de 10 elementos.

`ArrayList(int initialCapacity)`

Crea un objeto de la clase ArrayList vacío (sin elementos) pero con una capacidad inicial por defecto especificada por nosotros.

Excepciones

`IllegalArgumentException`	Si la capacidad inicial especificada es negativa

`ArrayList(Collection<? extends E> c)`

Crea un objeto de la clase ArrayList que contendrá inicialmente los objetos de la colección c.

Excepciones

`NullPointerException`	Si la colección especificada es **null**

6.3.3.2 METODOS BASICOS

`boolean add(E e)`

Añade el elemento especificado al final de la lista

Si la capacidad actual de la lista no admite un elemento más entonces la capacidad aumenta automáticamente un 50% de la capacidad actual (lo hace la clase ArrayList internamente). Por ejemplo, si la capacidad es 10 la nueva capacidad al añadir un elemento sería 15.

`void add(int index, E element)`

Inserta el elemento especificado en una posición específica de la lista.

El elemento actual de la posición (**index**) especificada y todos los elementos siguientes se desplazan una posición a la derecha (se suma uno a sus índices).

Excepciones

`IndexOutOfBoundsException`	Si el índice está fuera de rango (index<0 || index>size())

`void clear()`

Elimina todos los elementos de la lista

`boolean contains(Object o)`

Devuelve true si la lista contiene el elemento especificado, más formalmente, si contiene al menos un elemento e tal que (o==null ? e==null : o.equals(e)), es decir,

si el elemento buscado es null devuelve true si la lista al menos contiene un elemento null,

y si el elemento buscado no es null entonces devuelve true si existe un elemento e en la lista tal que o.equals(e).

`E get(int index)`

Devuelve el elemento que está en la posición especificada en la lista

Excepciones

`IndexOutOfBoundsException`	Si el índice está fuera de rango (index<0 || index>=size())

`int indexOf(Object o)`

Devuelve el índice de la primera ocurrencia del elemento especificado en la lista, o -1 si la lista no contiene el elemento.

Más formalmente, , devuelve el índice i más bajo tal que

(o==null ? get(i)==null : o.equals(get(i))),

o -1 si el índice no existe.

`boolean isEmpty()`

Devuelve true si la lista está vacía (no contiene elementos)

`Iterator<E> iterator()`

Devuelve un objeto iterador para recorrer los elementos de la lista en una secuencia adecuada.

Los métodos de un iterador son “fail-fast”: si la lista es modificada estructuralmente en cualquier momento después de que el iterador haya sido creado, de la manera que sea excepto mediante los propios métodos `add` y `remove`, el iterador lanzará la excepción `ConcurrentModificationException`.

Por tanto, ante una modificación concurrente, el iterador falla rápida y limpiamente, en vez de una forma peligrosamente arbitraria, sin provocar comportamientos no deterministas en momentos indeterminados de tiempo en el futuro.

`E remove(int index)`

Elimina el elemento que está en la posición especificada de la lista.

Excepciones

`IndexOutOfBoundsException`	Si el índice está fuera de rango (index<0 || index>=size())

`boolean remove(Object o)`

Elimina de la lista la primera ocurrencia del elemento especificado, si existe.

Si la lista no contiene el elemento, la lista queda sin modificar.

Más formalmente, elimina el elemento de índice i más bajo tal que

(o==null ? get(i)==null : o.equals(get(i))),

si tal elemento existe.

Devuelve true si la lista contenía el elemento especificado y la lista cambió como resultado de la llamada.

`E set(int index, E element)`

Reemplaza en la lista el elemento de la posición especificada con el elemento especificado.

Excepciones

`IndexOutOfBoundsException` Si el índice está fuera de rango (index<0 || index>=size())

`int size()`

Devuelve el número de elementos de la lista.

6.3.3.3 METODOS AVANZADOS

ensureCapacity, forEach, removeIf, sort

`void sort(Comparator<? super E> c)`

Ordena la lista según el criterio de comparación establecido por el comparador especificado.

Todos los elementos de la lista deben ser mutuamente comparables usando el comparador especificado.

Es decir, c.compare(e1, e2) no debe lanzar una excepción `ClassCastException` para ningún par de elementos (e1, e2) de la lista.

Si el comparador especificado es null entonces todos los elementos de la lista deben implementar la interface Comparable y se usará el orden natural (**natural ordering**) de los elementos para la ordenación de la lista.

Para más información sobre la interfaces Comparable y Comparator ver apartados 6.9.3.1 INTERFACE COMPARABLE y 6.9.3.2 INTERFACE COMPARATOR en el anexo.

6.3.3.4 OTROS METODOS

addAll, clone, lastIndexOf, listIterator, removeAll, removeRange, replaceAll, retainAll

6.3.4 VECTOR

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

La clase vector implementa un array flexible de objetos. Funcionalmente es muy parecido a ArrayList.

En general usaremos la clase ArrayList en vez de Vector excepto cuando estemos realizando un programa que necesite sincronización de hilos (*threads*[1]).

ArrayList es más eficiente que Vector ya que no tiene la sobrecarga de gestión de la sincronización de hilos de ejecución.

6.3.5 LINKEDLIST

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Implementación de una lista doblemente enlazada. Permite elementos null.

Dispone de todas las operaciones permitidas para una estructura de lista doblemente enlazada.

Las operaciones de acceso por índice a elementos de la lista como get(), recorrerán la lista desde un extremo de la lista al contrario comenzando por el extremo más cercano al índice especificado.

Para recorrer la lista proporciona los iteradores devueltos por los métodos: iterator() y listIterator() que son “fail-fast”.

Esta implementación es no sincronizada, con las implicaciones que ya vimos para el caso de ArrayList.

LinkedList es una clase implementada de forma no sincronizada (necesitaría de sincronización externa en el caso de acceso concurrente desde varios hilos de ejecución).

[1] Procesos e hilos en Java: https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

6.3.5.1 CONSTRUCTORES

`LinkedList()`

Crea una lista vacía.

`LinkedList(Collection<? extends E> c)`

Crear un objeto de la clase LinkedList que contendrá inicialmente los objetos de la colección c.

Excepciones

`NullPointerException` Si la colección especificada es **null**

6.3.5.2 METODOS BASICOS

`boolean add(E e)`

Añade el elemento especificado al final de la lista

Este método es equivalente a addLast(E).

`void add(int index, E element)`

Inserta el elemento especificado en una posición específica de la lista.

Excepciones

`IndexOutOfBoundsException` Si el índice está fuera de rango (index<0 || index>size())

`void addFirst(E e)`

Insertar el elemento especificado al principio de la lista

`void addLast(E e)`

Añade el elemento especificado al final de la lista

Este método es equivalente a add(E).

`void clear()`

Elimina todos los elementos de la lista

`boolean contains(Object o)`

Devuelve true si la lista contiene el elemento especificado, más formalmente, si contiene al menos un elemento e tal que (o==null ? e==null : o.equals(e)), es decir,

si el elemento buscado es null devuelve true si la lista al menos contiene un elemento null,

y si el elemento buscado no es null entonces devuelve true si existe un elemento e en la lista tal que o.equals(e).

`E get(int index)`

Devuelve el elemento que está en la posición especificada en la lista

Excepciones

`IndexOutOfBoundsException`	Si el índice está fuera de rango (index<0 || index>=size())

`E getFirst()`

Devuelve el primer elemento de la lista

Excepciones

`NoSuchElementException`	Si la lista está vacía

`E getLast()`

Devuelve el último elemento de la lista

Excepciones

`NoSuchElementException`	Si la lista está vacía

`int indexOf(Object o)`

Devuelve el índice de la primera ocurrencia del elemento especificado en la lista, o -1 si la lista no contiene el elemento.

Más formalmente, , devuelve el índice i más bajo tal que

(o==null ? get(i)==null : o.equals(get(i))),

o -1 si el índice no existe.

`boolean offer(E e)`

Añade el elemento especificado al final de la lista.

`boolean offerFirst(E e)`

Inserta el elemento especificado al principio de la lista

`boolean offerLast(E e)`

Equivalente a offer.

`E peek()`

Devuelve, sin eliminar, el primer elemento de la listac

Equivalente a peekFirst()

`E peekFirst()`

Equivalente a peek()

`E peekLast()`

Devuelve, sin eliminar, el último elemento de la lista, o null si la lista está vacía

`E poll()`

Recupera y elimina el primer elemento de la lista

`E pollFirst()`

Equivalente a poll.

`E pollLast()`

Recupera y elimina el último elemento de la lista

`E pop()`

Desapila un elemento de la pila representada por esta lista

Excepciones

`NoSuchElementException`	Si la lista está vacía

`void push(E e)`

Apila un elemento en la cima de la pila representada por esta lista

`E remove()`

Recupera y elimina el primer elemento de la lista

Excepciones

`NoSuchElementException`	Si la lista está vacía

`E remove(int index)`

Elimina el elemento que está en la posición especificada de la lista.

Excepciones

`IndexOutOfBoundsException`	Si el índice está fuera de rango (index<0 || index>=size())

`boolean remove(Object o)`

Elimina de la lista la primera ocurrencia del elemento especificado, si existe.

Si la lista no contiene el elemento, la lista queda sin modificar.

Más formalmente, elimina el elemento de índice i más bajo tal que

(o==null ? get(i)==null : o.equals(get(i))),

si tal elemento existe.

Devuelve true si la lista contenía el elemento especificado y la lista cambió como resultado de la llamada.

`E removeFirst()`

Recupera y elimina el primer elemento de la lista

Excepciones

`NoSuchElementException`	Si la lista está vacía

`E removeLast()`

Recupera y elimina el último elemento de la lista

Excepciones

`NoSuchElementException`	Si la lista está vacía

`E set(int index, E element)`

Reemplaza en la lista el elemento de la posición especificada con el elemento especificado.

Excepciones

`IndexOutOfBoundsException`	Si el índice está fuera de rango (index<0 || index>=size())

`int size()`

Devuelve el número de elementos de la lista.

6.3.5.3 METODOS AVANZADOS

listIterator, spliterator

`ListIterator<E> listIterator(int index)`

Devuelve un objeto iterador de lista para recorrer los elementos de la lista en una secuencia adecuada, comenzando en la posición especificada.

Sigue el contrato especificado en: `List.listIterator(int)`

Los métodos de un iterador de lista son “fail-fast”: si la lista es modificada estructuralmente en cualquier momento después de que el iterador haya sido creado, de la manera que sea excepto mediante los propios métodos `add` y `remove`, el iterador lanzará la excepción `ConcurrentModificationException`.

Por tanto, ante una modificación concurrente, el iterador falla rápida y limpiamente, en vez de una forma peligrosamente arbitraria, sin provocar comportamientos no deterministas en momentos indeterminados de tiempo en el futuro.

El iterador de listas permite al programador recorrer la lista en cualquier sentido, modificar la lista durante la iteración, y obtener la posición actual del iterador en la lista. Un objeto ListIterator no tiene elemento actual; su posición de cursor siempre está entre el elemento que sería devuelto por una llamada a `previous()` y el elemento que sería devuelto por una llamada a `next()`. Un iterador de lista para una lista de longitud n tiene n+1 posiciones del cursor posibles, como se ilustra mediante los símbolos de circunflejo a continuación:

```
                 Elemento(0)   Elemento(1)   Elemento(2)   … Elemento(n-1)
Cursor       ^             ^             ^             ^                 ^
```

Los métodos `remove()` y `set(Object)` no están definidos en términos de la posición del cursor sino que operan sobre el último elemento devuelto por una llamada a `next()` o `previous()`.

Este interface es miembro de la JCF.

Métodos: add, hasNext, hasPrevious, next, nextIndex, previous, previousIndex, remove, set

6.3.5.4 OTROS METODOS

addAll, clone, lastIndexOf, listIterator, removeFirstOcurrence, removeLastOcurrence, toArray

6.4 PILAS Y COLAS

6.4.1 PILAS EN GENERAL

Una pila es una estructura LIFO (Last In First Out). Los elementos salen de la pila en el orden inverso al que entraron.

Operaciones que se pueden realizar con una pila:
- apilar un elemento (push): se añade un nuevo elemento en la cima de la pila
- desapilar un elemento (pop): se extrae el elemento que esté en la cima de la pila
- saber si la pila está vacía, es decir, no contiene ningún elemento
- consultar el elemento que está en la cima de la pila sin extraerlo (peek)

Ejemplo 5

Si apilara carácter a carácter la palabra “EDUARDO”, al desapilar carácter a carácter obtendría la palabra “ODRAUDE”.

Ejemplo 6

Intercambio de 2 variables

PUSH A
PUSH B
POP A
POP B

Las pilas se suelen implementar como listas enlazada de nodos. Los elementos se insertan al principio de la lista (cima) y se extraen también del principio de la lista (cima).

Actividad 6.5	Usando una clase nodo de tipo genérico implementar “a mano” (sin usar las clases al efecto de la JCF) una pila que permita realizar las operaciones: apilar, desapilar, estaVacia, cima.

6.4.2 COLAS EN GENERAL

Una cola es una estructura FIFO (First In First Out). El primer elemento que entra es el primero que sale. Es decir, los elementos salen de la cola en el mismo orden en el que entraron.

Operaciones que se pueden realizar con una cola:
- añadir un elemento al final de la cola
- extraer un elemento del principio de la cola
- consultar si la cola está vacía
- recorrer la cola

Ejemplo 7

La cola de personas en una taquilla de un cine.

Actividad 6.6	Usando una clase nodo que contenga un objeto de tipo Persona implementar "a mano" (sin usar las clases al efecto de la JCF) una cola.

6.4.3 STACK

```
public class Stack<E>
extends Vector<E>
```

La clase Stack representa una pila de objetos. Extiende la clase Vector con 5 métodos que permiten que un objeto de la clase Vector sea tratado como una pila:

boolean	empty()
E	peek()
E	pop()
E	push(E item)
int	search(Object o)

Oracle aconseja usar las clases que implementan la interface Deque por disponer de un conjunto de métodos más completo y consistente que la clase Stack

6.4.4 VECTOR

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Esta clase representa un array redimensionable igual que la clase ArrayList.

La diferencia con la clase ArrayList es que la clase Vector es sincronizada (a diferencia de ArrayList que no lo es) y se debe usar cuando necesitemos usar arrays redimensionables en un entorno de concurrencia con varios hilos de ejecución.

6.4.5 INTERFACE DEQUE

No entraremos en detalles por ahora.

6.4.6 ARRAYDEQUE

No entraremos en detalles por ahora.

6.4.7 LINKEDLIST

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

La clase LinkedList implementa una lista doblemente enlazada y proporciona operaciones para usar dicha lista como una pila o como una cola.

Actividad 6.7	Consultar los métodos de la clase LinkedList en la API de Java que permiten usar la lista como una estructura LIFO (pila).
Actividad 6.8	Idem anterior: FIFO (cola)
Actividad 6.9	Realizar un programa donde se use una pila mediante la clase LinkedList
Actividad 6.10	Idem anterior: cola.

6.5 CONJUNTOS

6.5.1 INTERFACE SET
6.5.2 HASHSET
6.5.3 SORTEDSET

6.6 MAPAS

6.6.1 INTERFACE MAP
6.6.2 HASHMAP
6.6.3 SORTEDMAP

6.7 RECURSIVIDAD

6.7.1 FACTORIAL
6.7.2 FIBONACCI

6.8 ARBOLES

6.8.1 ARBOLES EN GENERAL
6.8.2 ARBOLES BINARIOS
6.8.3 IMPLEMENTACION DE UN ARBOL BINARIO
6.8.4 RECORRIDO DE UN ARBOL BINARIO
6.8.4.1 PREORDEN
6.8.4.2 INORDEN
6.8.4.3 POSTORDEN
6.8.5 TREESET
6.8.6 TREEMAP

6.9 ANEXO

6.9.1 JAVA COLLECTIONS FRAMEWORK

La plataforma Java incluye un marco de trabajo para colecciones (**Java Collections Framework**).

El marco de trabajo (**framework**) de colecciones en Java es una arquitectura unificada para representar y manipular colecciones permitiendo que las colecciones sean manipuladas de forma independiente a los detalles de implementación.

6.9.1.1 VENTAJAS

1. **Reducción del esfuerzo de programación:** gracias a que JCF proporciona estructuras de datos y algoritmos para que no tengamos que escribirlos nosotros mismos.
2. **Aumento del rendimiento:** gracias a que JCF proporciona implementaciones de alto rendimiento de estructuras de datos y algoritmos. Los programas pueden ser fácilmente refactorizados para mejorar el diseño, la legibilidad y reducir la complejidad, ya que las distintas implementaciones de cada interface son intercambiables. Por ejemplo: podemos cambiar la implementación de una clase que use listas (interface List) desde un ArrayList a una lista enlazada (LinkedList) ya que ambas son listas, sin afectar al comportamiento externo.
3. **Interoperabilidad entre APIs no relacionadas:** mediante la definición de un lenguaje común para pasar y recibir colecciones.
4. **Reducción del esfuerzo requerido para aprender APIs:** evitando que tengamos que aprender APIs especificas para colecciones.
5. **Reducción del esfuerzo requerido para diseñar e implementar APIs:** no obligándonos a crear APIs específicas para colecciones.
6. **Fomento de la reutilización de código:** gracias a que JCF proporciona un interface estándar para colecciones y algoritmos con los que manipularlas.

6.9.1.2 COMPONENTES

1. **Interfaces de colecciones:** representan diferentes tipos de colecciones, como conjuntos (Set) , listas (List) y mapas (Map). Estas interfaces son la base del JCF.
2. **Implementaciones de propósito general:** implementaciones básicas de interfaces de colecciones.
3. **Implementaciones heredadas:** clases colección de versiones anteriores, **Vector** y **Hashtable**, han sido reestructuradas para implementar interfaces de colecciones.
4. **Implementaciones de propósito concreto:** implementaciones diseñadas para usar en situaciones concretas. Estas implementaciones ofrecen características de rendimiento no estándar, restricciones de uso, o comportamiento.
5. **Implementaciones para concurrencia:** diseñadas para alto uso de concurrencia.
6. **Implementaciones de clases de envoltura (wrapper):** añaden funcionalidad, como sincronización, a otras implementaciones.
7. **Implementaciones a medida:** mini-implementaciones de alto rendimiento de interfaces de colecciones.

8. **Implementaciones abstractas:** implementaciones parciales de interfaces de colecciones para facilitar la personalización de las implementaciones.
9. **Algoritmos:** métodos estáticos que realizan funciones de utilidad sobre colecciones, como ordenar una lista.
10. **Infraestructura:** interfaces que proporcionan soporte esencial para las interfaces de las colecciones.
11. **Utilidades de arrays:** funciones de utilidad para trabajar con arrays de tipos primitivos y arrays de objetos.

6.9.1.3 INTERFACES DE COLECCIONES

Las interfaces de colecciones están divididas en dos grupos.

Grupo 1: interface **`java.util.Collection`** y sus descendientes

`java.util.List`
`java.util.Set`
`java.util.SortedSet`
`java.util.NavigableSet`
`java.util.Queue`
`java.util.concurrent.BlockingQueue`
`java.util.concurrent.TransferQueue`
`java.util.Deque`
`java.util.concurrent.BloquickDeque`

Grupo 2: interface **`java.util.Map`** `y sus descendientes`
`java.util.SortedMap`
`java.util.NavigableMap`
`java.util.concurrent.ConcurrentMap`
`java.util.concurrent.ConcurrentNavigableMap`

Los interfaces mapas del grupo 2 no son realmente colecciones pero se incluyen en JCF ya que proporcionan 3 operaciones que devuelven vistas de colecciones de elementos del mapa para manipular los mapas como si fueran colecciones.

Algunas implementaciones restringen qué elementos (o en el caso de los mapas, qué pares clave-valor) pueden ser almacenados. Las restricciones posibles suponen requerir a los elementos:

- Ser de un tipo determinado
- No ser nulos
- Cumplir determinado predicado

El resultado de intentar añadir un elemento que viole una restricción de implementación es una excepción en tiempo de ejecución (`RuntimeException`), típicamente alguna de estas:

```
ClassCastException
IllegalArgumentException
NullPointerException
```

6.9.1.4 IMPLEMENTACIONES DE COLECCIONES

Las clases que implementan interfaces de colecciones tienen nombres típicamente en la forma:

XXXInterface

donde XXX es el nombre de estilo de implementación (Hash, Array, Tree, …)
e Interface es el nombre del interface del colección (Set, List, Deque, Map)

Las implementaciones de propósito general se resumen en la siguiente tabla:

Interface	Tabla de hash	Array redimensionable	Arbol balanceado	Lista enlazada	Tabla de hash + Lista enlazada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Las clases abstractas siguientes

```
AbstractCollection
AbstractSet
AbstractList
AbstractSequentialList
AbstractMap
```

proporcionan implementaciones básicas de las principales interfaces de colecciones para minimizar el esfuerzo requerido para implementarlas.

La documentación de la API para estas clases describen con exactitud cómo está implementado cada método para que el programador sepa qué métodos deben ser sobreescritos.

6.9.1.5 COLLECCIONES CONCURRENTES

Las aplicaciones que usan colecciones desde más de un hilo (**thread**) deben ser programadas con cuidado. Esto se conoce como **programación concurrente**.

La plataforma Java incluye amplio soporte para la programación concurrente.
No veremos programación concurrente en este curso.

6.9.2 COMPARATIVA DE RENDIMIENTO

6.9.2.1 DEQUE

LinkedList vs ArrayDeque
https://docs.oracle.com/javase/tutorial/collections/implementations/deque.html

6.9.2.2 LIST

ArrayList vs LinkedList
https://docs.oracle.com/javase/tutorial/collections/implementations/list.html

6.9.2.3 MAP

HashMap vs TreeMap vs LinkedHashMap
https://docs.oracle.com/javase/tutorial/collections/implementations/map.html

6.9.2.4 QUEUE

LinkedList vs PriorityQueue
https://docs.oracle.com/javase/tutorial/collections/implementations/queue.html

6.9.2.5 SET

HashSet vs TreeSet vs LinkedHashSet
https://docs.oracle.com/javase/tutorial/collections/implementations/set.html

6.9.3 INTERFACES PARA ORDENACION

6.9.3.1 INTERFACE COMPARABLE

```
public interface Comparable<T>
```

Las listas (y arrays) de objetos que implementen esta interface Comparable pueden ser ordenadas automáticamente mediante el método estático Collections.sort (y Arrays.sort).

Los objetos que implementen este interface pueden ser usados como claves en un mapa ordenado (ver apartado 6.6.3 SORTEDMAP en la página 18) o como elementos en un conjunto ordenado (ver apartado 6.5.3 SORTEDSET en la página 18).

Orden natural (**natural ordering**): según la documentación de la API de Java para la interface Comparable, la interface impone un orden total sobre los objetos de cada clase que implementa la citada interface. A este orden se le llama “orden natural de la clase”, y se realiza mediante el llamado método de comparación natural: compareTo. Para más información sobre la interface Comparable y el método compareTo.

6.9.3.2 INTERFACE COMPARATOR

PTE

6.9.4 TIPOS GENERICOS

PTE

6.10 BIBLIOGRAFIA

6.10.1 TIPOS GENERICOS

https://docs.oracle.com/javase/tutorial/java/generics/index.html

6.10.2 AUTOBOXING Y UNBOXING

https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html