Análisis y Diseño de Algoritmos

Semana 6 Adicional

Agenda

- ArrayLists
- La interface Comparable
- Lists
- Estudio de caso: Comparación de Vocabulario
- Sets Maps, Iterators
- Iteratos

Ejercicio

- Escriba un programa que lea un archivo y muestre las palabras de ese archivo como una lista.
  - Primero muestre todas las palabras.
  - Luego muestre todos los plurales (que terminan en "s") en mayúscula.
  - Luego muéstrelos en orden inverso.
  - Luego muéstrelos con todas las palabras plurales eliminadas.

- ¿Deberíamos resolver este problema usando un arreglo?
  - ¿Por qué sí o por qué no?

Solución ingenua

```
String[] allWords = new String[1000];
int wordCount = 0;

Scanner input = new Scanner(new File("data.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords[wordCount] = word;
    wordCount++;
}
```

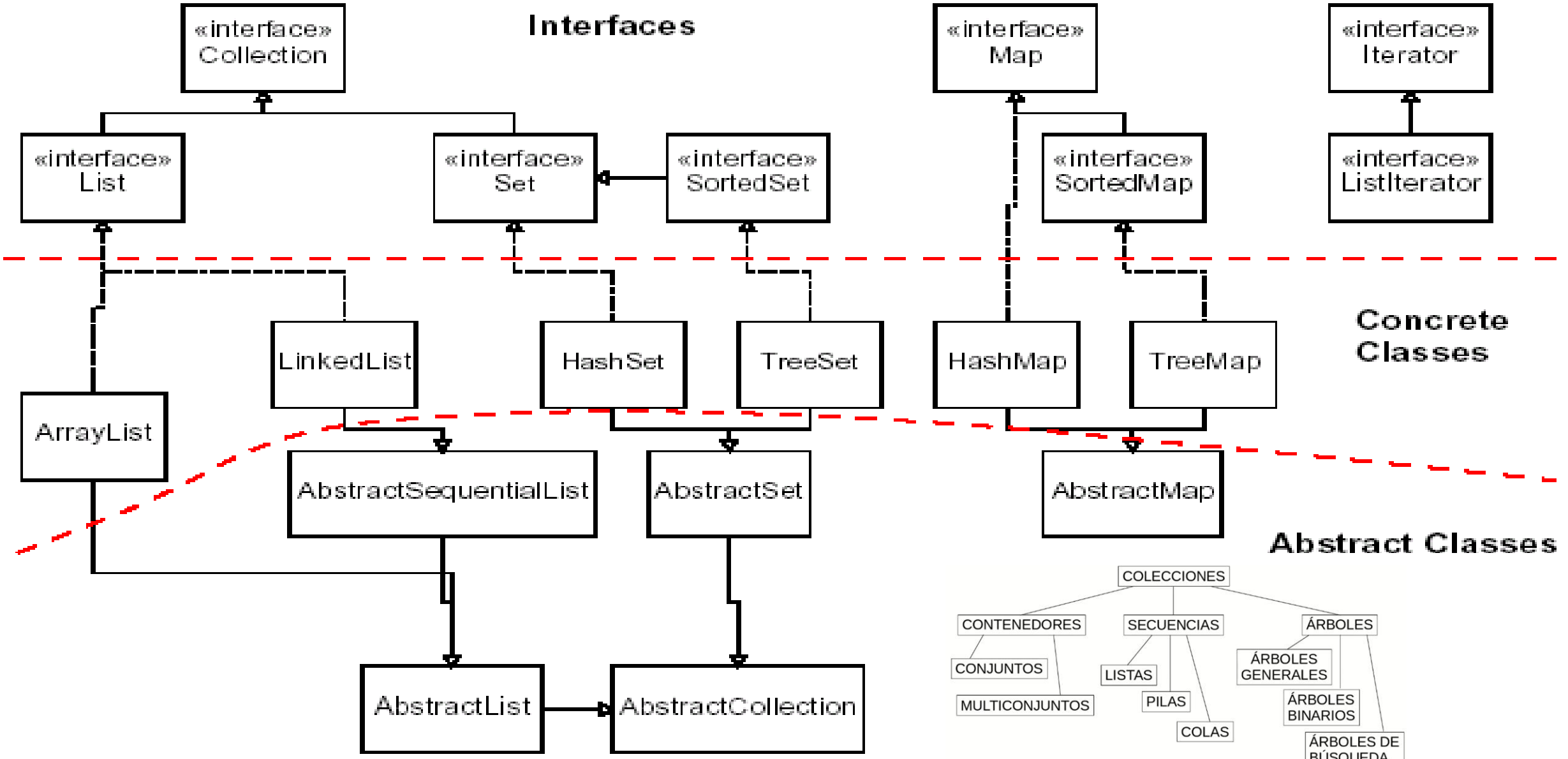
- Problema: No sabe cuántas palabras tendrá el archivo.
  - Difícil de crear un arreglo del tamaño apropiado.
  - Las partes posteriores del problema son más difíciles de resolver.
- Afortunadamente, hay otras formas de almacenar datos además de hacerlo en un arreglo.

Colecciones

- **Colecciones**: un objeto que almacena datos; también conocido como "estructura de datos"
  - Los objetos almacenados son llamados **elementos**
  - Algunas colecciones mantienen un orden; algunas permiten duplicados
  - Típicas operaciones: *add*, *remove*, *clear*, *contains* (search), *size*
  - Ejemplos encontrados en las bibliotecas de clases Java:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
  - Todas las colecciones están en el paquete java.util

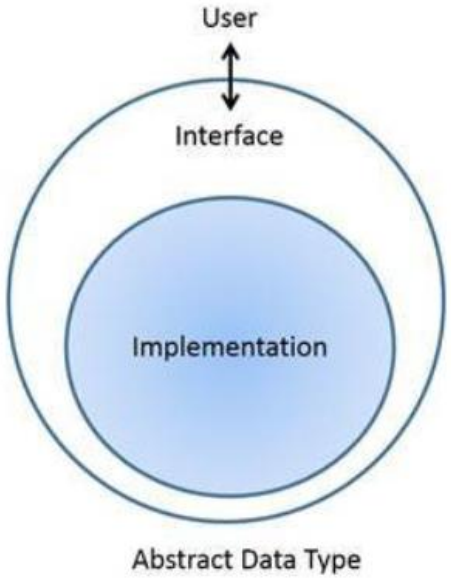
```
import java.util.*;
```

Marco de colecciones Java



Tipo de datos abstractos (ADT)

- **Tipo de datos abstractos (ADT)** : una especificación de una colección de datos y las operaciones que se pueden realizar en ellos.
  - Describe *lo que* hace una colección, no *cómo* lo hace.



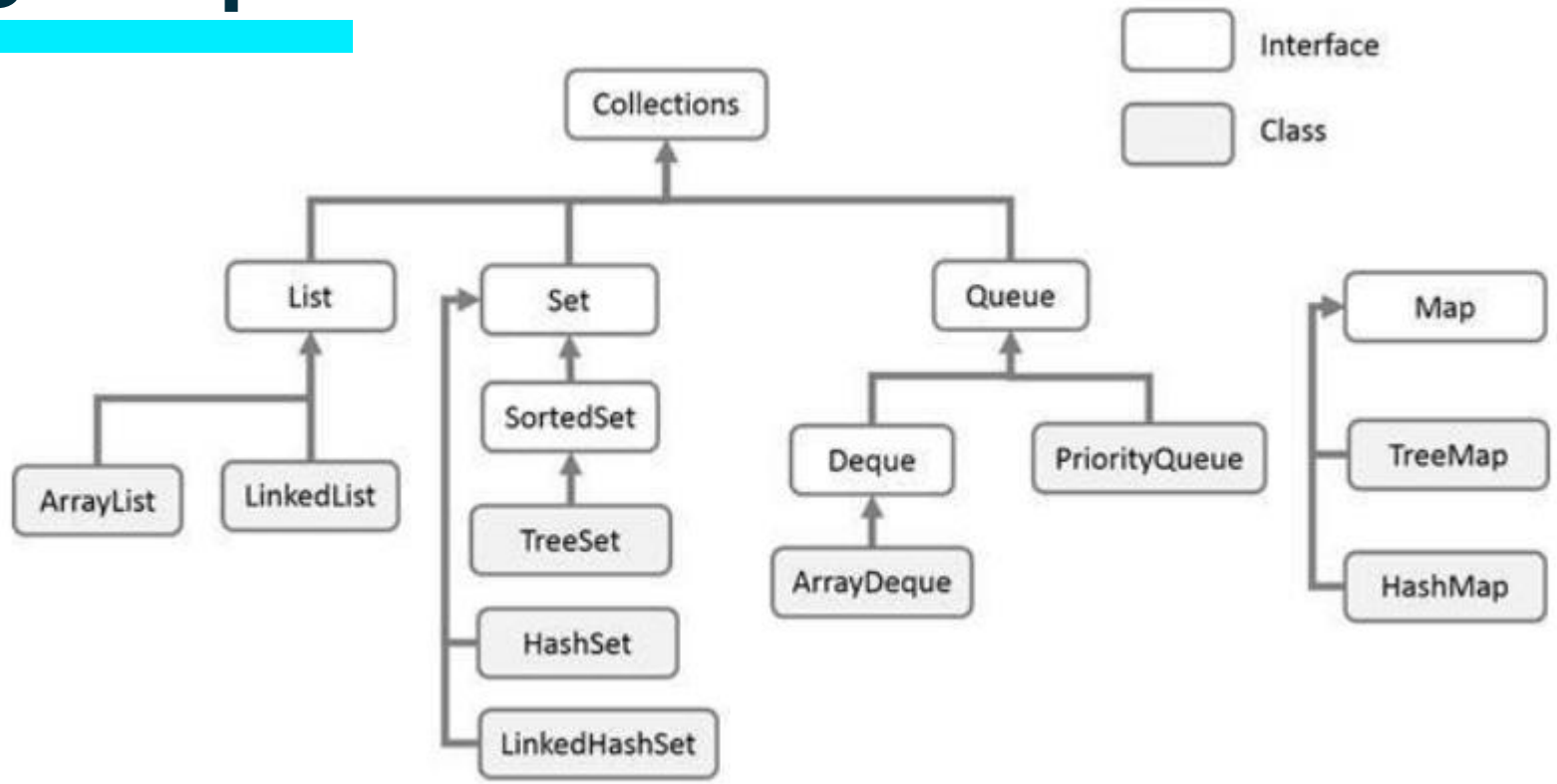
- El marco de colección de Java describe ADT con interfaces:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`, `SortedMap`

- Un ADT se puede implementar de múltiples maneras por clases:
  - `ArrayList` y `LinkedList` implementan `List`
  - `HashSet` y `TreeSet` implementan `Set`
  - `LinkedList` , `ArrayDeque` , etc. implementan `Queue`

ADT (Tipo de datos abstractos)

Un tipo de datos abstracto (ADT) es una descripción lógica de los datos y las operaciones que se permiten en ellos. ADT se define desde el punto de vista del usuario de los datos. ADT se preocupa por los posibles valores de los datos y la interfaz expuesta por ellos. ADT no se preocupa por la implementación real de la estructura de datos. Por ejemplo, un usuario quiere almacenar algunos números enteros y encontrar su valor medio. ADT para esta estructura de datos admitirá dos funciones, una para sumar números enteros y otra para obtener el valor medio. ADT para esta estructura de datos no habla sobre cómo se implementará exactamente.

¿Porqué usar el Marco de colecciones de Java?



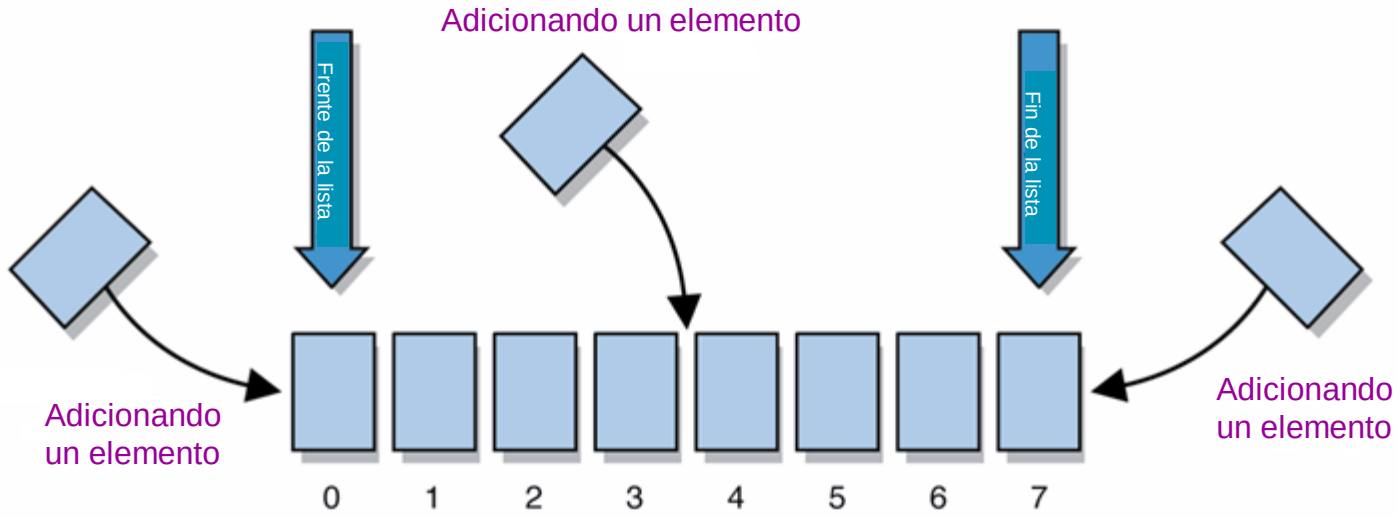
Estructura de datos

Las estructuras de datos son representaciones concretas de datos y se definen desde el punto de vista del programador de datos. La estructura de datos representa cómo se almacenarán los datos en la memoria. Todas las estructuras de datos tienen sus pros y sus contras. Dependiendo del tipo de problema, elegimos la estructura de datos que mejor se adapte a él. Por ejemplo, podemos almacenar datos en una matriz, una lista enlazada, una pila, una cola, un árbol, etc. Nota: - En estas primeras sesiones del curso, estudiaremos varias estructuras de datos y su API, para que el usuario pueda utilizarlos sin conocer su implementación interna.

1. Los programadores no tienen que implementar estructuras de datos y algoritmos básicos repetidamente. De ese modo evita la reinvención de la rueda. Así, el programador puede dedicar más esfuerzo a la lógica empresarial.
2. El marco de colecciones Java es un código bien probado, de alta calidad y alto rendimiento. Su uso aumenta la calidad de los programas.
3. El costo de desarrollo se reduce a medida que se reutilizan las estructuras de datos básicas y los algoritmos que se implementan en el marco de colecciones.
4. Fácil de revisar y comprender los programas escritos por otros desarrolladores, ya que la mayoría de los desarrolladores de Java utilizan el marco de colecciones. Además, está bien documentado.

Listas

- **Lista** : Una colección que almacena una secuencia ordenada de elementos
  - Cada elemento es accesible por un índice basado en 0
  - Una lista tiene un tamaño (número de elementos que se han adicionado)
  - Los elementos se pueden agregar al frente, por la parte posterior o en cualquier otro lado
  - En Java, una lista se puede representar como un objeto ArrayList



Idea de una lista

- En lugar de crear un arreglo, cree un objeto que represente una "lista" de elementos. (inicialmente una lista vacía)

  []

- Puede agregar elementos a la lista.
  - El comportamiento predeterminado es adicionar al final de la lista.

    [hello, ABC, goodbye, okay]

- El objeto lista mantiene seguimiento de los valores de los elementos que se adicionan, su orden, índices y su tamaño total.
  - Piense en un “ArrayList" como un arreglo de objetos de cambio de tamaño automático.
  - Internamente, la lista se implementa utilizando un arreglo y un atributo de tamaño.

Métodos ArrayList

`add(`**valor**`)`	Adiciona valor al final de la lista
`add(`**indice**`, `**valor**`)`	Inserta el valor dado justo antes del índice dado, desplazando los valores posteriores a la derecha
`clear()`	Elimina todos los elementos de la lista
`indexOf(`**valor**`)`	Devuelve el primer índice donde se encuentra el valor dado en la lista (-1 si no se encuentra)
`get(`**indice**`)`	Devuelve el valor en el índice dado
`remove(`**indice**`)`	Elimina/devuelve el valor en el índice dado, desplazando los valores posteriores a la izquierda
`set(`**indice**`, `**valor**`)`	Reemplaza el valor en el índice dado con el valor dado
`size()`	Devuelve el número de elementos en la lista
`toString()`	Devuelve una representación de cadena de la lista como "[3, 42, -7, 15]"

Métodos ArrayList 2

`addAll(`**list**`)` `addAll(`**indice**`, `**list**`)`	Adiciona todos los elementos de la lista dada a esta lista (al final de la lista, o los inserta en el índice dado)
`contains(`**valor**`)`	Devuelve true si el valor dado se encuentra en algún lugar de esta lista
`containsAll(`**list**`)`	Devuelve true si esta lista contiene todos los elementos de la lista dada
`equals(`**list**`)`	Devuelve true si otra lista contiene los mismos elementos
`iterator()` `listIterator()`	Devuelve un objeto utilizado para examinar el contenido de la lista (visto más adelante)
`lastIndexOf(`**valor**`)`	Devuelve el último índice donde se encuentra el valor en la lista (-1 si no se encuentra)
`remove(`**valor**`)`	Encuentra y elimina el valor dado de esta lista
`removeAll(`**list**`)`	Elimina todos los elementos encontrados en la lista dada de esta lista
`retainAll(`**list**`)`	Elimina cualquier elemento que no se encuentre en la lista dada de esta lista
`subList(`**desde**`, `**hasta**`)`	Devuelve la subparte de la lista entre índices desde (inclusivo) hasta (exclusivo)
`toArray()`	Devuelve los elementos en esta lista como un arreglo

Parámetros de tipo (genéricos)

```
ArrayList<Tipo> nombre = new ArrayList<Tipo>();
```

- Al construir un ArrayList, debe especificar el tipo de elementos que contendrá entre < >.
  - Esto se llama un parámetro de tipo o una clase genérica.
  - Permite que la misma clase ArrayList almacene listas de diferentes tipos.

```
ArrayList<String> nombres = new ArrayList<String>();
nombres.add("Javier Antonio");
nombres.add("Vanessa Karina");
```

ArrayList vs. array

- Construcción

```
String[] nombres = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- Almacenar un valor

```
nombres[0] = "Jessica";
list.add("Jessica");
```

- Recuperar un valor

```
String s = nombres[0];
String s = list.get(0);
```

ArrayList vs. Array 2

- Haciendo algo a cada valor que comienza con "B"

```
for (int i = 0; i < nombres.length; i++) {
    if (nombres[i].startsWith("B")) { ... }
}
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}
```

- Ver si se encuentra el valor "Benson"

```
for (int i = 0; i < nombres.length; i++) {
    if (nombres[i].equals("Benson")) { ... }
}
for (int i = 0; i < list.size(); i++) {
if (list.contains("Benson")) { ... }
}
```

Revisemos nuevamente el ejercicio inicial

- Escriba un programa que lea un archivo y muestre las palabras de ese archivo como una lista.
  - Primero muestre todas las palabras.
  - Luego muestre todos los plurales (que terminan en "s") en mayúscula.
  - Luego muéstrelos en orden inverso.
  - Luego muéstrelos con todas las palabras plurales eliminadas.

Solución de ejercicio (parcial)

```
ArrayList<String> allPalabras = new ArrayList<String>();
Scanner input = new Scanner(new File("palabras.txt"));
while (input.hasNext()) {
    String palabra = input.next();
    allPalabras.add(palabra);
}
System.out.println(allPalabras);
// eliminar todas las palabras plurales
for (int i = 0; i < allPalabras.size(); i++) {
    String palabra = allPalabras.get(i);
    if (palabra.endsWith("s")) {
        allPalabras.remove(i);
        i--;
    }
}
System.out.println(allPalabras);
```

ArrayList como parámetro

```
public static void nombre(ArrayList<Tipo> nombre) {
```

- Ejemplo:

```
// Elimina todas las palabras plurales de la lista dada.
public static void removePlural(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        String str = list.get(i);
        if (str.endsWith("s")) {
            list.remove(i);
            i--;
        }
    }
}
```

- También puede devolver una lista:

```
public static ArrayList<Tipo> nombreMetodo(params)
```

ArrayList de primitivas?

- El tipo que especifique al crear una ArrayList debe ser un tipo de objeto; No puede ser un tipo primitivo.

```
// ilegal: int no puede ser un parámetro de tipo
ArrayList<int> list = new ArrayList<int>();
```

- Pero aún podemos usar ArrayList con tipos primitivos mediante el uso de clases especiales llamadas clases “wrapper” (envoltura) en su lugar.

```
// crea una lista de ints
ArrayList<Integer> list = new ArrayList<Integer>();
```

Clases envoltura de tipos (“wrapper”)

Tipo Primitiva	Tipo Wrapper
int	Integer
double	Double
char	Character
boolean	Boolean

- Una envoltura es un objeto cuyo único propósito es mantener un valor primitivo.
- Una vez que construya la lista, úsela con primitivas como de costumbre:

```
ArrayList<Double> notas = new ArrayList<Double>();
notas.add(13.2);
notas.add(12.7);
...
double miNota = notas.get(0);
```

Ejercicio

- Escriba un programa que lea un archivo lleno de números y muestre todos los números como una lista, luego:
  - Imprime el promedio de los números.
  - Imprime el número más alto y más bajo.
  - Filtra todos los números pares.

Solución de ejercicio (parcial)

```
ArrayList<Integer> numeros = new ArrayList<Integer>();
Scanner input = new Scanner(new File("numeros.txt"));
while (input.hasNextInt()) {
    int n = input.nextInt();
    numeros.add(n);
}
System.out.println(numeros);
filtrarPares(numeros);
System.out.println(numeros);
...
// Elimina todos los elementos con valores pares de la lista dada.
public static void filtrarPares(ArrayList<Integer> list) {
    for (int i = list.size() - 1; i >= 0; i--) {
        int n = list.get(i);
        if (n % 2 == 0) {
            list.remove(i);
        }
    }
}
```

Out-of-bounds

- Los índices legales están entre 0 y el tamaño de la lista()- 1
  - Leer o escribir cualquier índice fuera de este rango provocará una excepción IndexOutOfBoundsException.

```
ArrayList<String> nombres = new ArrayList<String>();
nombres.add("Marty");
nombres.add("Kevin");
nombres.add("Vicki");
nombres.add("Larry");
System.out.println(nombres.get(0));        // okay
System.out.println(nombres.get(3));        // okay
System.out.println(nombres.get(-1));       // exception
nombres.add(9, "Aimee");                   // exception
```

ArrayList "misterio"

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i = 1; i <= 10; i++) {
    list.add(10 * i);   // [10, 20, 30, 40, ..., 100]
}
```

- ¿Cuál es la salida del siguiente código?

```
for (int i = 0; i < list.size(); i++) {
    list.remove(i);
}
System.out.println(list);
```

- Respuesta:

  [20, 40, 60, 80, 100]

ArrayList "misterio“ 2

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) {
    list.add(2 * i);   // [2, 4, 6, 8, 10]
}
```

- ¿Cuál es la salida del siguiente código?

```
int size = list.size();
for (int i = 0; i < size; i++) {
    list.add(i, 42);   // adicione 42 en el índice i
}
System.out.println(list);
```

- Respuesta:

```
[42, 42, 42, 42, 42, 2, 4, 6, 8, 10]
```

ArrayList como parámetro

```
public static void nombre(ArrayList<Tipo> nombre) {
```

- Ejemplo:

```
// Elimina todas las palabras plurales de la lista dada.
public static void removePlural(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        String str = list.get(i);
        if (str.endsWith("s")) {
            list.remove(i);
            i--;
        }
    }
}
```

- También puede devolver una lista:

```
public static ArrayList<Tipo> nombreMetodo (params)
```

Ejercicios

- Escriba un método addStars que acepte un arraylist de cadenas como parámetro y coloque un * después de cada elemento.
  - Ejemplo: si un arraylist llamado lista almacena inicialmente:

    [the, quick, brown, fox]
  - Luego de addStars (lista); lo hace almacenar:

    [the, *, quick, *, brown, *, fox, *]

- Escriba un método removeStars que acepte un arraylist de cadenas, y elimine las estrellas (deshaciendo lo que hizo addStars arriba).

Solución de los ejercicios

```
public static void addStars(ArrayList<String> list)
   {
      for (int i = 0; i < list.size(); i += 2) {
          list.add(i, "*");
      }
}
```

```
public static void removeStars(ArrayList<String>
   list) {
      for (int i = 0; i < list.size(); i++) {
          list.remove(i);
      }
}
```

Ejercicio

- Escriba un método “intersect” que acepte dos listas ordenadas de enteros como parámetros y devuelva una nueva lista que contenga solo los elementos que se encuentran en ambas listas.
  - Ejemplo: si las listas llamadas list1 y list2 inicialmente se almacenan:

    [1, **4**, 8, 9, **11**, 15, **17**, **28**, 41, **59**]

    [**4**, 7, **11**, **17**, 19, 20, 23, **28**, 37, **59**, 81]

  - Luego, la llamada intersect(list1, list2) devuelve la lista:

    [4, 11, 17, 28, 59]

Otros ejercicios

- Escriba un método que invierta el orden de los elementos en un ArrayList de cadenas.

- Escriba un método que acepte un ArrayList de cadenas y reemplace cada palabra que termina con una "s" con su versión en mayúsculas.

- Escriba un método que acepte un ArrayList de cadenas y elimine cada palabra de la lista que termine con una "s", sin distinción entre mayúsculas y minúsculas.

Objetos que almacenan colecciones

- Un objeto puede tener un arreglo, una lista u otra colección como atributo.

```
public class Curso {
    private double[] notas;
    private ArrayList<String> nombreAlumno;

    public Curso() {
        notas = new double[4];
        nombreAlumno = new ArrayList<String>();
        ...
    }
```

- Ahora cada objeto almacena una colección de datos dentro de él.

La Interfaz Comparable

- El método Collections.sort se puede usar para ordenar una ArrayList. Es parte del paquete java.util. El siguiente programa muestra cómo usar Collections.sort:

```
    // Construye una ArrayList de Strings y la ordena.
import java.util.*;
public class Main {
  public static void main(String[] args) {
    ArrayList<String> palabras = new ArrayList<String>();
    palabras.add("four");
    palabras.add("score");
    palabras.add("and");
    palabras.add("seven");
    palabras.add("years");
    palabras.add("ago");
    // muestra la lista antes y después de ordenar
    System.out.println("antes de ordenar, las palabras son = " + palabras);
    Collections.sort(palabras);
    System.out.println("despues de ordenar, las palabras son = " + palabras);
    }
  }
```

- Este programa produce el siguiente resultado:

```
antes de ordenar las palabras son = [four, score, and, seven, years, ago]
despues de ordenas las palabras son = [ago, and, four, score, seven, years]
```

La Interfaz Comparable

- Si intenta hacer una llamada similar con un objeto “Punto”, a un ArrayList <Punto>, encontrará que el programa no se compila.
- ¿Por qué es posible ordenar una lista de objetos String pero no una lista de objetos Punto?
- La respuesta es que la clase String implementa la interfaz Comparable, mientras que la clase Punto no.

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class String

java.lang.Object
  java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

compareTo

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns `0` exactly when the `equals(Object)` method would return `true`.

La Interfaz Comparable

```
public interface Comparable<E> {
    public int compareTo(E otro);
}
```

- Una clase puede implementar la interfaz Comparable para definir una función de orden natural para sus objetos (comparación).

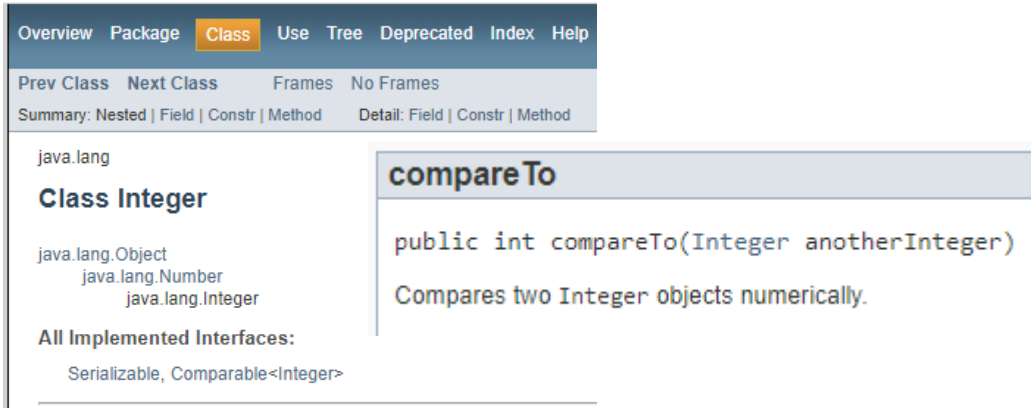
- Una llamada a su método compareTo debería devolver:
  un valor <0 si el otro objeto viene “después" de este,
  un valor> 0 si el otro objeto viene “antes" de este,
  o 0 si el otro objeto se considera "igual" a este.

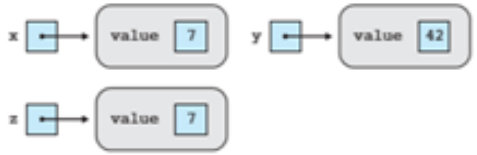
Plantilla Comparable

```
public class nombre implements Comparable<nombre> {

    ...

    public int compareTo(nombre otro) {
        ...
    }
}
```






```
Integer x = 7;
Integer y = 42;
Integer z = 7;
System.out.println(x.compareTo(y));
System.out.println(x.compareTo(z));
System.out.println(y.compareTo(x));
```

```
-1
0
1
```

El método compareTo

- La forma estándar para que una clase Java defina una función de comparación para sus objetos es definir un método compareTo.
  - Ejemplo: en la clase String, hay un método:

    ```
    public int compareTo(String other)
    ```

- Una llamada de **A**.compareTo(**B**) devolverá:

  un valor <0 si **A** viene "antes" de **B** en el orden,

  un valor> 0 si **A** viene "después" de **B** en el orden,

  o 0 si **A** y **B** se consideran "iguales" en el orden.

Usando compareTo

- compareTo puede usarse como prueba en una declaración if.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {  // true
    ...
}
```

Primitivas	Objetos
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

compareTo y Colecciones

- Puede usar un arreglo o una lista de cadenas con el método de búsqueda binaria incluido en Java porque llama a compareTo internamente.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- TreeSet/Map de Java utiliza compareTo internamente para realizar ordenamientos.

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Ordenando nuestros propios tipos

- No podemos realizar búsquedas binarias o hacer un `TreeSet/Map` de tipos arbitrarios de objetos, porque Java no sabe cómo ordenar los elementos.

  - El programa compila pero falla cuando lo ejecutamos.

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
…

Exception in thread "main"
  java.lang.ClassCastException
        at java.util.TreeSet.add(TreeSet.java:238)
```

Ejemplo Comparable

- Puede hacer que sus propias clases implementen la interfaz. Esto abrirá una gran cantidad de soluciones que se incluyen en las bibliotecas de clases Java.

- Por ejemplo, hay métodos integrados para ordenar listas y para acelerar las búsquedas.

```
public class Punto implements Comparable<Punto> {
    private int x;
    private int y;
    …

    // ordenar por x y romper lazos por y
    public int compareTo(Punto otro) {
        if (x < otro.x) {
            return -1;
        } else if (x > otro.x) {
            return 1;
        } else if (y < otro.y) {
            return -1;    // mismo x, y más pequeño
        } else if (y > otro.y) {
            return 1;     // mismo x, mayor y
        } else {
            return 0;     // mismo x, mismo y
        }
    }
}
```

Trucos CompareTo

- ***Truco de resta*** : restar valores numéricos relacionados produce el resultado correcto para lo que desea que compareTo devuelva:

```
// ordenar por x y romper lazos por y
public int compareTo(Punto otro) {
    if (x != otro.x) {
        return x - otro.x;   // diferente x
    } else {
        return y - otro.y;   // mismo x; compara y
    }
}
```

  - La idea:
    - Si `x > otro.x`, entonces `x - otro.x > 0`
    - Si `x < otro.x`, entonces `x - otro.x < 0`
    - Si `x == otro.x`, entonces `x - otro.x == 0`
      - NOTA: Este truco no funciona para double (pero vea Math.signum)

Trucos CompareTo 2

- ***Truco de delegación:*** si los atributos de su objeto son comparables (como cadenas), use sus resultados compareTo para ayudarlo a:

```
// ordenar por nombre de empleado, por ejemplo, "Jim" <"Susan"
public int compareTo(Empleado otro) {
    return nombre.compareTo(otro.getNombre());
}
```

- *Truco toString:* Si la representación toString de tu objeto está relacionada con el orden, úsala para ayudarte a:

```
// ordenar por fecha, por ejemplo, "19/09"> "01/04"
public int compareTo(Fecha otro) {
    return toString().compareTo(otro.toString());
}
```

Ejemplo Comparable

- Exploremos una clase que se puede usar para realizar un seguimiento de una fecha del calendario. La idea es realizar un seguimiento de un mes y día en particular, pero no del año.
- Por ejemplo, una organización podría querer una lista de los cumpleaños de sus empleados que no indique cuántos años tienen..

```
// La clase FechaCalendario almacena información sobre una sola fecha
// FechaCalendario (mes y día pero no año)

public class FechaCalendario implements Comparable<FechaCalendario> {
private int mes;
private int dia;
public FechaCalendario(int dia, int mes){
this.mes = mes;
this.dia = dia;
}
// Compara esta fecha del calendario con otra fecha.
// Las fechas se comparan por mes y luego por día.
public int compareTo(FechaCalendario otro) {
if (mes != otro.mes) {
return mes - otro.mes;
} else {
return dia - otro.dia;
}
}
public int getMes() {
return mes;
}
public int getDia() {
return dia;
}
public String toString() {
return dia + "/" + mes;
}

}
```

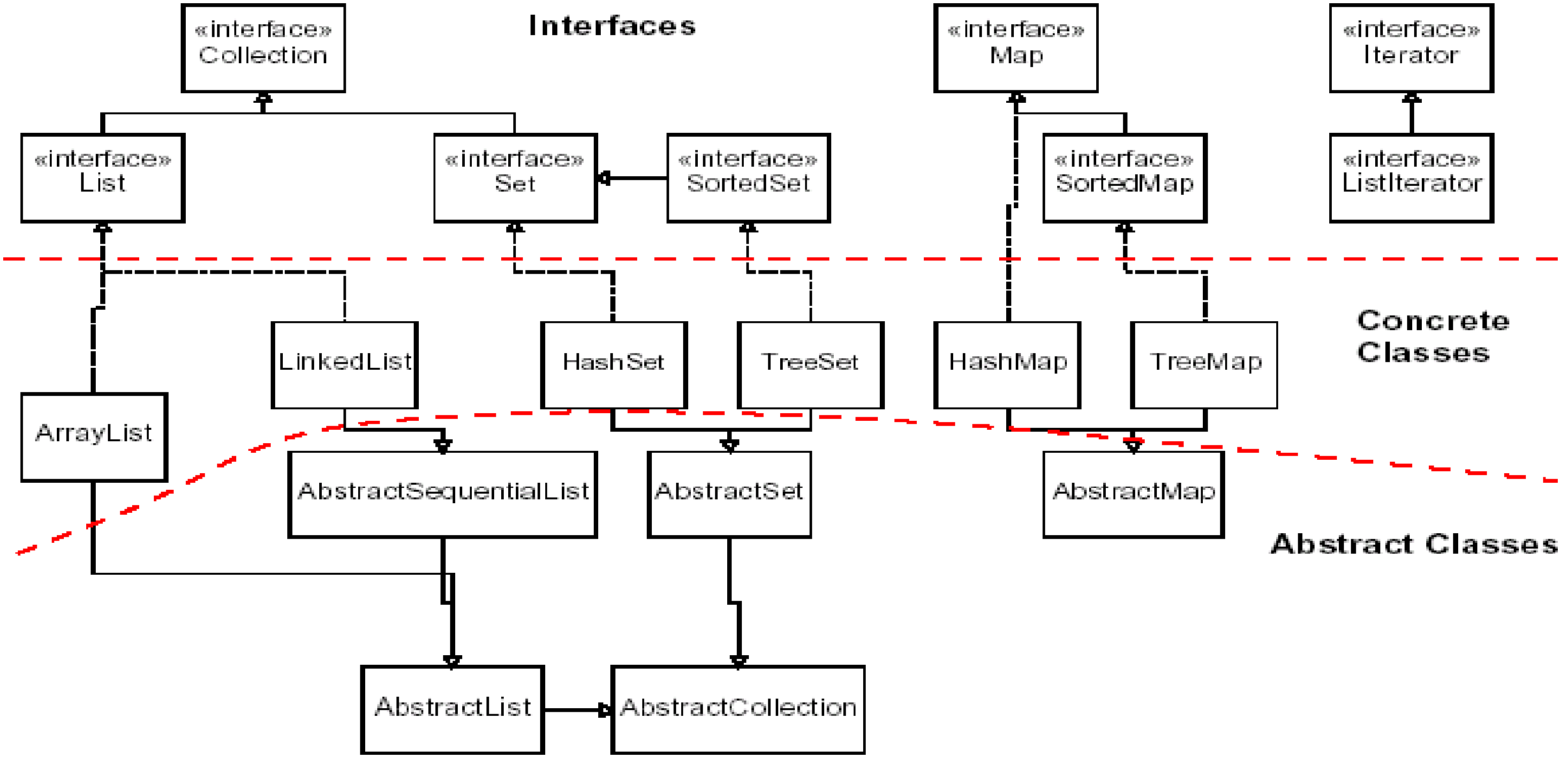
Ejemplo Comparable

- Una de las principales ventajas de implementar la interfaz Comparable es que le brinda acceso a utilidades integradas como Collections.sort.
- La clase FechaCalendario implementa la interfaz Comparable, por lo que, podemos usar Collections.sort para ordenar un ArrayList <FechaCalendario>:

```
// Programa que crea una lista de los cumpleaños y los ordena
import java.util.*;
public class Main {
public static void main(String[] args) {
ArrayList<FechaCalendario> fechas = new ArrayList<FechaCalendario>();
fechas.add(new FechaCalendario(22, 2));
fechas.add(new FechaCalendario(30, 10));
fechas.add(new FechaCalendario(13, 4));
fechas.add(new FechaCalendario(16, 3));
fechas.add(new FechaCalendario(28, 4));
System.out.println("Cumpleaños = " + fechas);
Collections.sort(fechas);
System.out.println("Cumpleaños = " + fechas);
}
}
```

Estudio de caso: Vocabulario

Marco de colecciones Java



Ejercicio

- Escriba un programa que cuente la cantidad de palabras únicas en un archivo de texto grande (por ejemplo, Moby Dick o la Ciudad y los Perros).
  - Almacene las palabras en una colección e informe el número de palabras únicas.
  - Una vez que haya creado esta colección, permita que el usuario busque una palabra para ver si aparece varias veces en el archivo de texto.

- ¿Qué colección es apropiada para este problema?

Análisis empírico

Ejecutar un programa y medir su rendimiento.

System.currentTimeMillis()

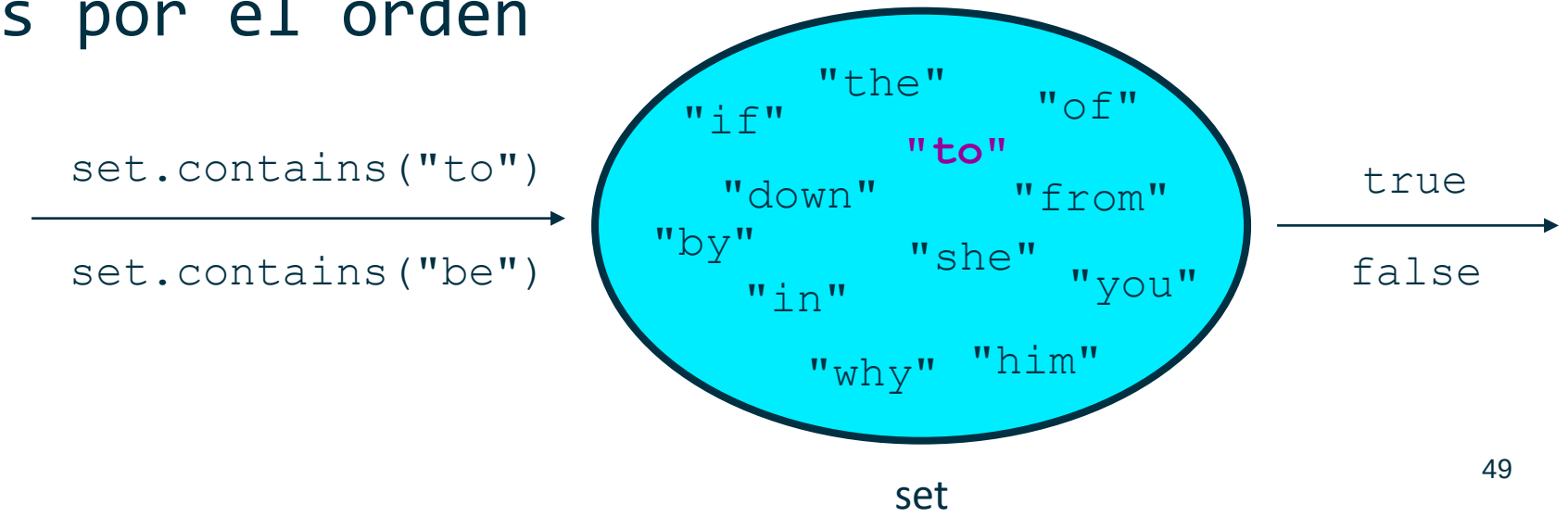
- Devuelve un número entero que representa el número de milisegundos que han pasado desde las 12:00 a.m., 1 de enero de 1970.
  - El resultado se devuelve como un valor de tipo long, que es como int pero con un rango numérico mayor (64 bits frente a 32).

- Se puede llamar dos veces para ver cuántos milisegundos han transcurrido entre dos puntos en un programa.

- ¿Cuánto tiempo lleva almacenar Moby Dick en una lista?

Sets

- **set**: Una colección de valores únicos (no se permiten duplicados) que pueden realizar las siguientes operaciones de manera eficiente:
  - add, remove, search (contains)
  - No pensamos que Set (un conjunto) tenga índices; simplemente agregamos cosas al conjunto en general y no nos preocupamos por el orden



Implementación Set

- En Java, sets son representados por la interface Set en java.util
- Set es implementado por las clases HashSet y TreeSet

  - HashSet: implementado utilizando un arreglo de "tabla hash"; muy rápido: **O(1),** para todas las operaciones los elementos se almacenan en un orden impredecible.
  - TreeSet: Implementado usando un "árbol de búsqueda binario"; bastante rápido: **O(log N),** para todas las operaciones los elementos se almacenan en forma ordenada
  - LinkedHashSet: **O(1)** y se almacena en orden de inserción

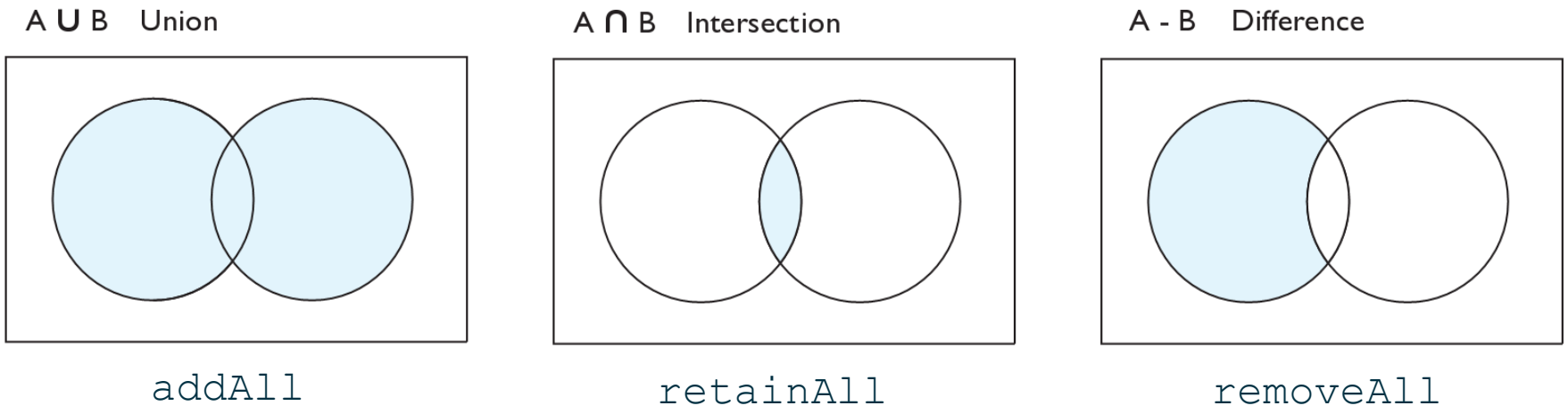
Métodos Set

```
List<String> list = new ArrayList<String>();
...
Set<Integer> set = new TreeSet<Integer>();     // vacío
Set<String> set2 = new HashSet<String>(list);
```

- Puede construir un conjunto vacío o uno basado en una colección dada

`add(`**valor**`)`	agrega el valor dado al conjunto
`contains(`**valor**`)`	devuelve true si el valor dado se encuentra en este conjunto
`remove(`**valor**`)`	elimina el valor dado del conjunto
`clear()`	elimina todos los elementos del conjunto
`size()`	devuelve el número de elementos en la lista
`isEmpty()`	devuelve verdadero si el tamaño del conjunto es 0
`toString()`	devuelve una cadena como `"[3, 42, -7, 15]"`

Operaciones Set



`addAll(`**coleccion**`)`	agrega todos los elementos de la colección dada a este conjunto
`containsAll(`**coleccion**`)`	devuelve true si este conjunto contiene todos los elementos del conjunto dado
`equals(`**set**`)`	devuelve true si otro conjunto contiene los mismos elementos
`iterator()`	devuelve un objeto utilizado para examinar el contenido del conjunto
`removeAll(`**coleccion**`)`	elimina todos los elementos de la colección dada de este conjunto
`retainAll(`**coleccion**`)`	elimina elementos de este conjunto que no se encuentran en una colección determinada
`toArray()`	devuelve un arreglo de los elementos en este conjunto

Uso Set

- HashSet: los elementos se almacenan en un orden impredecible

```
Set<String> nombres = new HashSet<String>();
nombres.add("Jake");
nombres.add("Robert");
nombres.add("Marisa");
nombres.add("Kasey");
System.out.println(nombres);
// [Kasey, Robert, Jake, Marisa]
```

- TreeSet: los elementos se almacenan en su orden "natural"

```
Set<String> nombres = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

- LinkedHashSet: elementos almacenados en orden de inserción

```
Set<String> nombres = new LinkedHashSet<String>();
...
// [Jake, Robert, Marisa, Kasey]
```

El Bucle For Each

```
for (tipo nombre : coleccion) {
    instrucciones;
}
```

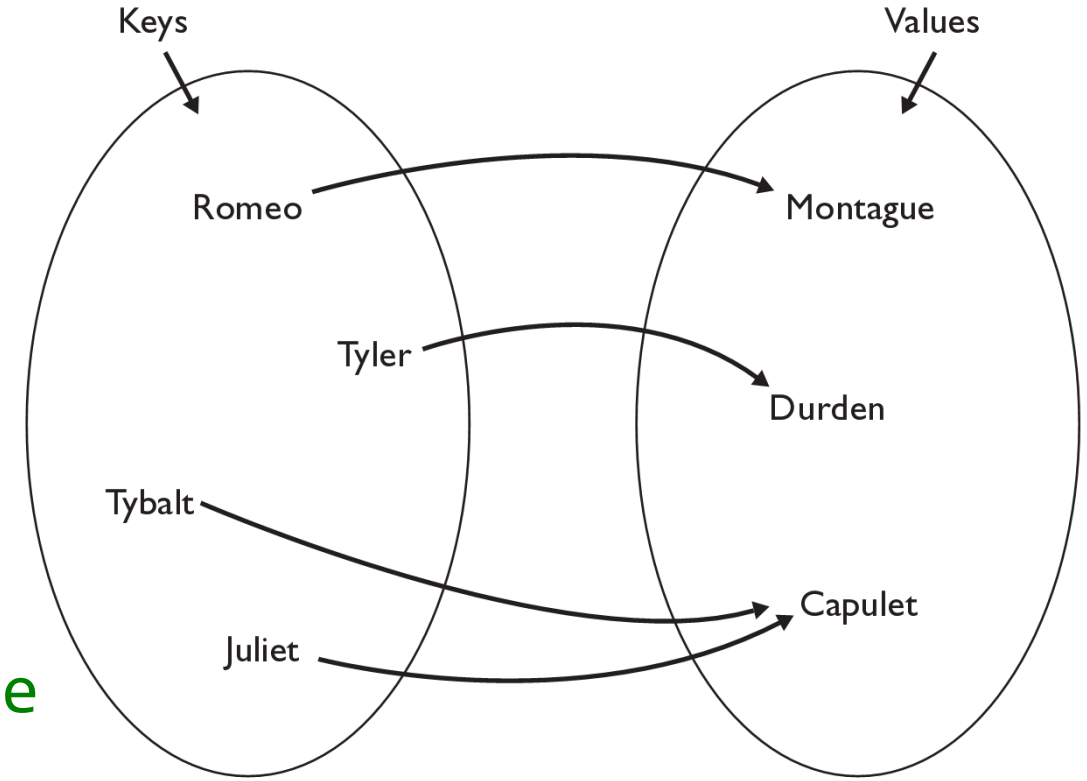
- Proporciona una sintaxis limpia para recorrer los elementos de un conjunto, lista, arreglo u otra colección

```
Set<Double> notas = new HashSet<Double>();
…
for (double nota : notas) {
    System.out.println("Nota de los estudiantes: " + nota);
}
```

  - Necesario porque los Sets no tienen índices; no se puede obtener el elemento i

El ADT Mapa

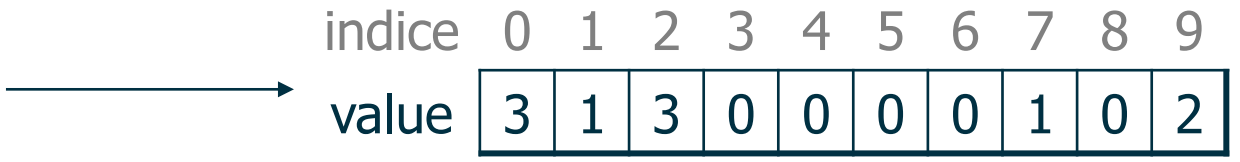
- **map**: contiene un conjunto de claves únicas y una colección de valores, donde cada clave está asociada a un valor.
  - también conocido como "diccionario", “arreglo asociativo", "hash"
- operaciones básicas del mapa:
  - **put**(clave, *valor*): agrega una asignación de una clave a un valor.
  - **get**(clave): recupera el valor asignado a la clave.
  - **remove**(clave): elimina la clave dada y su valor asignado.



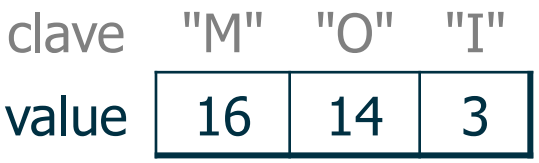
`myMap.get("Juliet")` devuelve `"Capulet"`

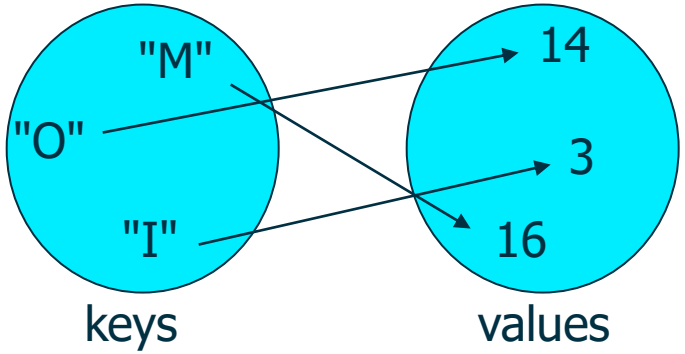
Mapas y recuento

- Un mapa puede considerarse como una generalización de un arreglo de conteo
  - el "índice" (clave o key) no tiene que ser un int
- Ejemplos:
  - Contar digitos: 22092310907



  - Contar votos: "MOOOOOOMMMMMOOOOOOMOMMIMOMMIMOMMIO"





Implementación Map

- En Java, los mapas están representados por la interfaz Map en java.util
- Map es implementado por las clases HashMap y TreeMap
  - HashMap: implementado usando un arreglo llamado "tabla hash"; extremadamente rápido: O (1) ; las claves se almacenan en un orden impredecible
  - TreeMap: implementado como una estructura de "árbol binario“ enlazado; muy rápido: O (log N) ; las claves se almacenan en forma ordenada
  - Un mapa requiere 2 parámetros de tipo: uno para claves, uno para valores.

```
// maps de claves String a valores Integer
Map<String, Integer> votos = new HashMap<String, Integer>();
```

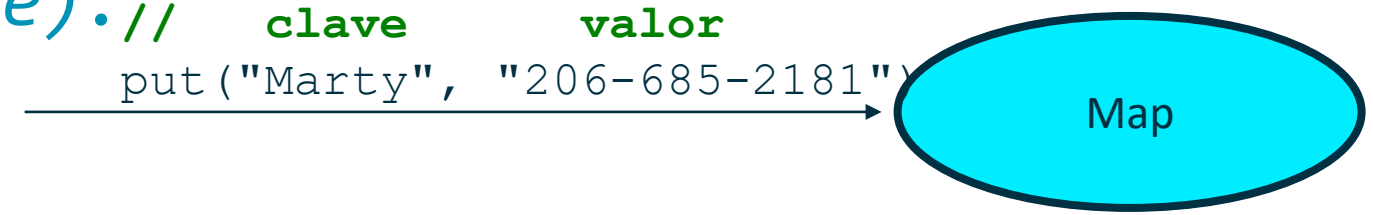
Métodos Map

`put(`**clave**`, `**valor**`)`	agrega una asignación de la clave dada al valor dado; si la clave ya existe, reemplaza su valor con el dado
`get(`**clave**`)`	devuelve el valor asignado a la clave dada (nulo si no se encuentra)
`containsKey(`**clave**`)`	devuelve true si el mapa contiene una asignación para la clave dada
`remove(`**clave**`)`	elimina cualquier mapeo existente para la clave dada
`clear()`	elimina todos los pares clave / valor del mapa
`size()`	devuelve el número de pares clave / valor en el mapa
`isEmpty()`	devuelve true si el tamaño del mapa es 0
`toString()`	devuelve una cadena como `"{a=90, d=60, c=70}"`

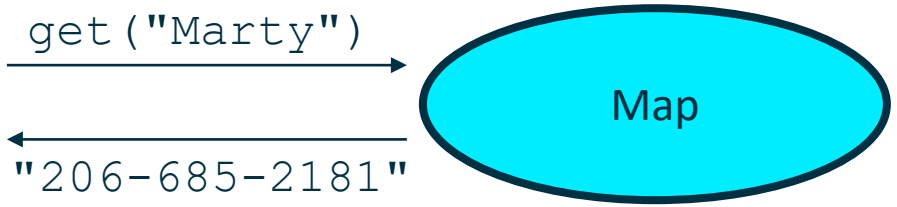
`keySet()`	devuelve un conjunto de todas las claves en el mapa
`values()`	devuelve una colección de todos los valores en el mapa
`putAll(`**map**`)`	agrega todos los pares clave / valor del mapa dado a este mapa
`equals(`**map**`)`	devuelve true si el mapa dado tiene las mismas asignaciones que este

Usando Maps

- Un mapa le permite obtener con una mitad de un par el otro valor del par.
  - *Recuerda una pieza de información sobre cada índice (clave).*



  - *Más tarde, podemos suministrar solo la clave y recuperar el valor relacionado:*
    - *Nos permite preguntar: ¿Cuál es el número de teléfono de Marty?*

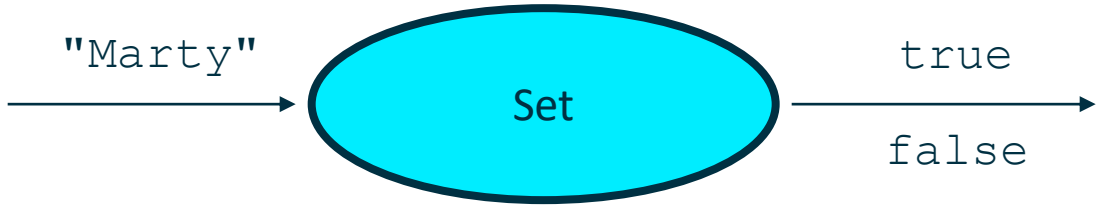


Solución al ejercicio

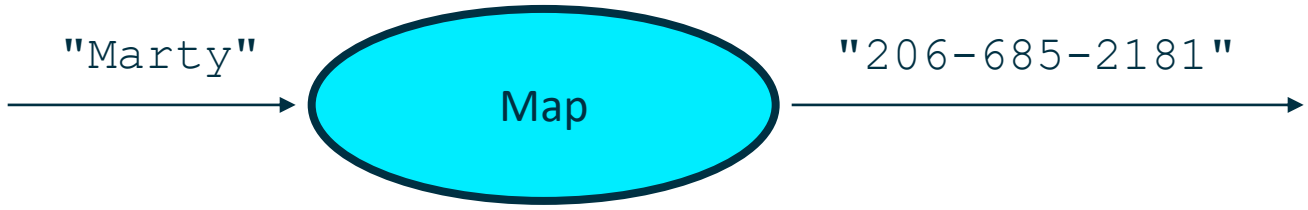
```
// lee el archivo en un mapa de [palabra -> número de ocurrencias]
Map<String, Integer> palabraCount = new HashMap<String, Integer>();
Scanner input = new Scanner(new File("mobydick.txt"));
while (input.hasNext()) {
    String palabra = input.next();
    if (palabraCount.containsKey(palabra)) {
        // visto esta palabra antes; aumentar el contador en 1
        int count = palabraCount.get(palabra);
        palabraCount.put(palabra, count + 1);
    } else {
        // nunca he visto esta palabra antes
        palabraCount.put(palabra, 1);
    }
}
System.out.println("Hay " + palabraCount.size() + " únicas.");
Scanner console = new Scanner(System.in);
System.out.print("Palabra a buscar? ");
String palabra = console.next();
System.out.println("aparece " + palabraCount.get(palabra) + " veces.");
```

Maps vs Sets

- Un set es como un mapa de elementos a valores booleanos.
  - *Set: ¿Se encuentra "Marty" en el conjunto? (true o false)*



  - *Map: ¿Cuál es el número de teléfono de "Marty"?*



keySet y values

- El método keySet devuelve un conjunto de todas las claves en el mapa
  - puede recorrer las claves en un bucle for each
  - puede obtener el valor asociado de cada clave llamando a get en el mapa

```
Map<String, Integer> edades = new TreeMap<String, Integer>();
edades.put("Marty", 19);
edades.put("Geneva", 2);
edades.put("Vicki", 57);
// edades.keySet() devuelve Set<String>
for (String nombre : edades.keySet()) {          // Geneva -> 2
    int edad = edades.get(nombre);               // Marty -> 19
    System.out.println(nombre + " -> " + edad); // Vicki -> 57
}
```

- El método values devuelve una colección de todos los valores en el mapa
  - puede recorrer los valores en un ciclo for each
  - no hay una manera fácil de pasar de un valor a su(s) clave(s) asociada(s)

Problema: mapeo opuesto

- Es legal tener un mapa de Sets, una lista de lists, etc.
- Supongamos que queremos hacer un seguimiento del promedio de calificaciones de cada postulante por nombre.

```
Map<String, Double> promFinal = new HashMap<String, Double>();
promFinal.put("Jared", 13.6);
promFinal.put("Alyssa", 14.0);
promFinal.put("Steve", 12.9);
promFinal.put("Stef", 13.6);
promFinal.put("Rob", 12.9);
...
System.out.println("El promedio final de Jared es " +
                   promFinal.get("Jared"));   // 13.6
```

- Esto no nos permite preguntar fácilmente qué postulante obtuvo un promedio final determinado.
  - ¿Cómo estructuraríamos un mapa para eso?

Invertir un mapa

- Podemos revertir el mapeo para que sea de Promedio Final a nombres.

```
Map<Double, String> promFinal = new HashMap<Double, String>();
promFinal.put(13.6, "Jared");
promFinal.put(14.0, "Alyssa");
promFinal.put(12.9, "Steve");
promFinal.put(13.6, "Stef");
promFinal.put(12.9, "Rob");
...
System.out.println("Quien tiene un 13.6? " +
                   promFinal.get(13.6));   // ???
```

- ¿Qué hay de malo con esta solución?
  - Más de un postulante puede tener el mismo promedio.
  - El mapa almacenará solo la última asignación que agreguemos.

Correcta inversión del mapa

- Realmente cada Promedio se asigna a una colección de personas.

```
Map<Double, Set<String>> promFinal = new HashMap<Double, Set<String>>();
promFinal.put(13.6, new TreeSet<String>());
promFinal.get(13.6).add("Jared");
promFinal.put(14.0, new TreeSet<String>());
promFinal.get(14.0).add("Alyssa");
promFinal.put(12.9, new TreeSet<String>());
promFinal.get(12.9).add("Steve");
promFinal.get(13.6).add("Stef");
promFinal.get(12.9).add("Rob");
...
System.out.println("Quien tiene un 13.6? " +
                   promFinal.get(13.6));   // [Jared, Stef]
```

  - debe tener cuidado al inicializar el conjunto para un promedio determinado antes de agregar

Ejercicios

- Modifique el programa de conteo de palabras para imprimir cada palabra que apareció en el libro al menos 1000 veces, se debe ordenar de menor a mayor la cantidad de veces.

- Escriba un programa que lea una lista de los nombres de alumnos y el promedio del semestre, luego imprima los promedios en orden creciente y los alumnos que tienen ese promedio, en forma ordenada.

```
Allison 5          1 : [Brian]
Alyssa 8           2 : ...
Brian 1            5 : [Allison, Kasey]
Kasey 5
...
```

Iterators

Examinando conjuntos y mapas

- No se puede acceder a los elementos de Sets y Maps por índice
  - debe usar un bucle "for each":

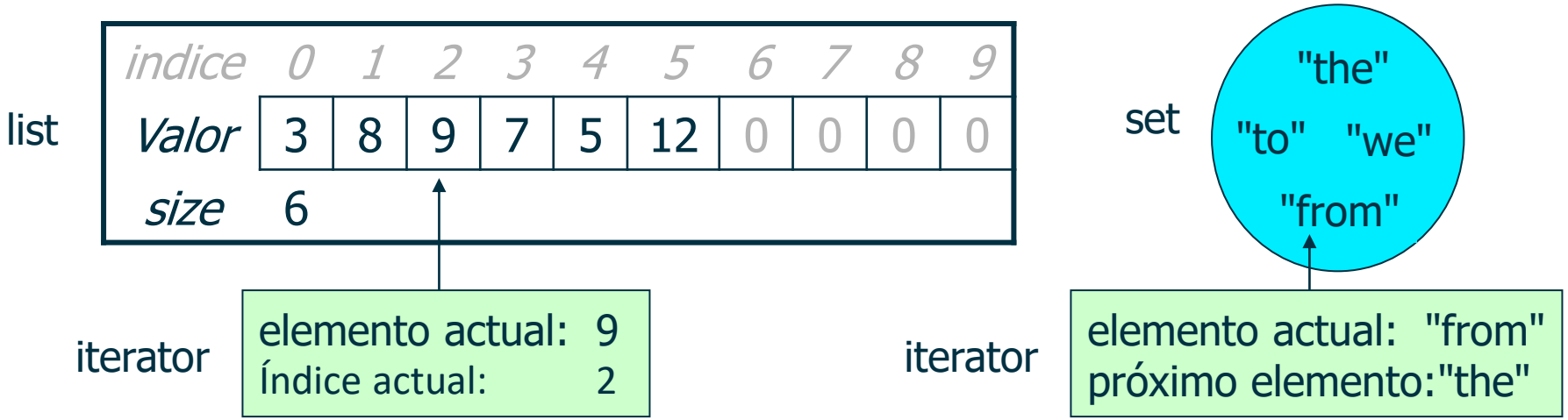
```
Set<Integer> puntajes = new HashSet<Integer>();
for (int puntaje : puntajes) {
    System.out.println("El puntaje es " + puntaje);
}
```

  - Problema: for each es de solo lectura; no se puede modificar el conjunto mientras se repite

```
for (int puntaje : puntajes) {
    if (puntaje < 60) {
    // lanza una ConcurrentModificationException
        puntajes.remove(puntaje);
    }
}
```

Iterators

- **iterator**: un objeto que permite a un cliente atravesar los elementos de cualquier colección.
  - Recuerda una posición y permite:
    - obtener el elemento en esa posición
    - avanzar a la siguiente posición
    - eliminar el elemento en esa posición



Métodos Iterator

`hasNext()`	devuelve true si hay más elementos para examinar
`next()`	devuelve el siguiente elemento de la colección (arroja una NoSuchElementException si no queda ninguna para examinar)
`remove()`	elimina el último valor devuelto por next () (arroja una IllegalStateException si aún no ha llamado a next ())

- Interfaz de iterator en java.util
  - cada colección tiene un método iterator() que devuelve un iterador sobre sus elementos

```
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();
...
```

Ejemplo Iterator

```
Set<Integer> puntajes = new TreeSet<Integer>();
puntajes.add(94);
puntajes.add(38);   // Kim
puntajes.add(87);
puntajes.add(43);   // Marty
puntajes.add(72);
…
Iterator<Integer> itr = puntajes.iterator();
while (itr.hasNext()) {
    int puntaje = itr.next();

    System.out.println("El puntaje es " + puntaje);

    // elimina cualquier calificación reprobatoria
    if (puntaje < 60) {
        itr.remove();
    }
}
System.out.println(puntajes);  // [72, 87, 94]
```

Ejemplo 2 - Iterator

```
Map<String, Integer> puntajes = new TreeMap<String, Integer>();
puntajes.put("Kim", 38);
puntajes.put("Lisa", 94);
puntajes.put("Roy", 87);
puntajes.put("Marty", 43);
puntajes.put("Marisa", 72);
…

Iterator<String> itr = puntajes.keySet().iterator();
while (itr.hasNext()) {
    String nombre = itr.next();
    int puntaje = puntajes.get(nombre);
    System.out.println(nombre + " obtuvo " + puntaje);

    // elimina a los estudiantes que fallan
    if (puntaje < 60) {
        itr.remove();     // elimina el nombre y la puntuación
    }
}
System.out.println(puntajes);  // {Lisa=94, Marisa=72, Roy=87}
```

Ejercicios

- Modifique el programa de Búsqueda de libros para eliminar cualquier palabra que esté en plural o en mayúscula de la colección.

- Escriba un programa para contar las apariciones de cada palabra en un archivo de texto grande (por ejemplo, Moby Dick o la Ciudad y los Perros).
  - Permita que el usuario escriba una palabra e informe cuántas veces apareció esa palabra en el libro.
  - Informe todas las palabras que aparecieron en el libro al menos 500 veces, en orden alfabético.
  - ¿Cómo almacenaremos los datos para resolver este problema?

Big O Colecciones Java

- Implementaciones de Listas

	get	add	contains	next	remove(0)	iterator.remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWrite-ArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

- Implementaciones de Conjuntos (Set)

	add	contains	next	notes
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

- Implementaciones Mapas o Diccionarios (Map)

	get	containsKey	next	Notes
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

- Implementaciones de pilas y colas (queue)

	offer	peek	poll	size
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDeque	O(1)	O(1)	O(1)	O(1)

https://www.bigocheatsheet.com/

¿Preguntas?

Resumiendo y Repasando…

- La clase ArrayList en el paquete java.util de Java representa una lista ampliable de objetos implementados usando un arreglo. Puede usar una ArrayList para almacenar objetos en orden secuencial. Cada elemento tiene un índice basado en cero.
- ArrayList es una clase genérica. Una clase genérica acepta un tipo de datos como parámetro cuando se crea, como ArrayList <String>.
- Una ArrayList mantiene su propio tamaño para usted; Se pueden agregar o quitar elementos de cualquier posición hasta el tamaño de la lista. Otras operaciones de ArrayList incluyen get, set, clear y toString.
- Se puede buscar en ArrayLists usando los métodos contains, indexOf y lastIndexOf.

Resumiendo y Repasando…

- El bucle Java for-each se puede usar para examinar cada elemento de un ArrayList. La lista no se puede modificar durante la ejecución del ciclo for-each.
- Cuando almacene valores primitivos como int o double en un ArrayList, debe declarar la lista con tipos especiales de contenedor como Integer y Double.
- La interfaz comparable define un orden natural para los objetos de una clase. Los objetos que implementan Comparable se pueden colocar en una ArrayList y ordenarlos. Muchos tipos comunes (como String e Integer) implementan Comparable. Puede implementar la interfaz Comparable en sus propias clases escribiendo un método compareTo.

Resumiendo y Repasando…

- Una colección es un objeto que almacena un grupo de otros objetos. Ejemplos de colecciones son ArrayList, HashSet y TreeMap. Las colecciones se utilizan para estructurar, organizar y buscar datos.
- Un conjunto es una colección que no permite duplicados. Los conjuntos generalmente se pueden buscar muy rápidamente para ver si contienen un valor de elemento particular. La interfaz Set representa conjuntos.
- Hay dos clases principales de conjuntos en Java: TreeSet y HashSet. Un TreeSet contiene datos comparables y está ordenado; un HashSet puede contener cualquier información y puede buscarse más rápido, pero sus elementos se almacenan en un orden impredecible.

Resumiendo y Repasando…

- Un mapa es una colección que asocia objetos clave con objetos de valor. Los mapas se utilizan para crear relaciones de asociación entre datos, como el nombre y el número de teléfono de una persona.
- Hay dos clases principales de mapas en Java: TreeMap y HashMap. Un TreeMap contiene claves comparables están en un orden ordenado; un HashMap puede contener cualquier dato como sus claves y realiza búsquedas de valores más rápido, pero sus claves se almacenan en un orden impredecible.
- Un iterador es un objeto que realiza un seguimiento de la posición actual en una lista y agiliza el examen de sus elementos en orden secuencial. Las listas enlazadas a menudo se usan con iteradores para aumentar la eficiencia.

FIN

Ejercicios

- Haga que la clase `HtmlTag` de un validador HTML sea comparable.
  - Compare las etiquetas por sus elementos, alfabéticamente por nombre.
  - Para el mismo elemento, las etiquetas de apertura vienen antes de cerrar las etiquetas.

```
// <body><b></b><i><b></b><br/></i></body>
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));   // <body>
tags.add(new HtmlTag("b", true));      // <b>
tags.add(new HtmlTag("b", false));     // </b>
tags.add(new HtmlTag("i", true));      // <i>
tags.add(new HtmlTag("b", true));      // <b>
tags.add(new HtmlTag("b", false));     // </b>
tags.add(new HtmlTag("br"));           // <br/>
tags.add(new HtmlTag("i", false));     // </i>
tags.add(new HtmlTag("body", false));  // </body>
System.out.println(tags);
// [<b>, </b>, <body>, </body>, <br/>, <i>, </i>]
```

Solución ejercicio

```
public class HtmlTag implements Comparable<HtmlTag> {
    ...
    // Compara tags (etiquetas) por su elemento ("body" antes que "head"),
    // rompiendo vínculos con las etiquetas de apertura antes q' las etiquetas de
    // cierre.
    // Devuelve < 0 para menor, 0 para igual, > 0 para mayor.
    public int compareTo(HtmlTag other) {
        int compare = element.compareTo(other.getElement());
        if (compare != 0) {
            // etiquetas diferentes; use el resultado compareTo de String
            return compare;
        } else {
            // misma etiqueta
            if ((isOpenTag == other.isOpenTag()) {
                return 0;   // exactamente el mismo tipo de etiqueta
            } else if (other.isOpenTag()) {
                return 1;   // he=open, I=close; I am after
            } else {
                return -1;  // I=open, he=close; I am before
            }
        }
    }
}
```