Trabajo de Investigación - Colecciones en Java

Equipo 11

12 de Octubre de 2022

Integrantes:

- Alfaro Domínguez Arturo.
- Herrera Lezama Fabricio Daniel.
- Lopez Gonzalez Erick.

¿Qué son las colecciones?

Las colecciones representan un grupo de objetos, estos objetos son conocidos como elementos, cuando se necesita trabajar con un conjunto de elementos, se necesita de un contenedor o almacén para guardarlos. En Java se proporciona la interfaz **Collection**, dicha interfaz permite almacenar cualquier tipo de objeto, como también, es posible usar ciertos métodos para su manipulación, tales como: añadir, eliminar, conocer el tamaño de la colección, etc.

¿Qué es un marco de recopilación de Java?

Un marco de recopilación de Java proporciona una arquitectura para almacenar y manipular un grupo de objetos. Un marco de recopilación de Java incluye lo siguiente:

- **Interfaces:** Una interfaz en Java se refiere como tal a los tipos de datos abstractos, permitiendo manipular las colecciones de Java independientemente de los detalles de su representacion. Como tambien, forman una jerarquía en los lenguajes de programación con el paradigma orientado a objetos.
- **Clases:** Las clases para Java son la implementación de la interfaz de colección, es decir, las estructuras de datos que se utilizan.
- **Algoritmo:** Se refiere a los métodos que utilizan para realizar operaciones como búsqueda y clasificación en objetos que se implementan en las interfaces de colección.

El marco de recopilación de Java proporciona a los desarrolladores el acceso a estructuras de datos pre empaquetadas, así como a algoritmos para manipular datos.

Modelo de la jerarquía de colecciones en Java

En el paquete de utilidades de Java (java.util) contiene todas las clases e interfaces que requiere el marco de la colección. El marco de la colección contiene una interfaz denominada interfaz iterable que proporciona el iterador para iterar a través de todas las colecciones. Esta interfaz se amplía con la interfaz de recopilación principal, que actúa como raíz del marco de recopilación. Todas las colecciones amplían esta interfaz de coleccion ampliando así las propiedades del iterador y los métodos de esta interfaz.

A continuación se presenta el diagrama que ilustra la jerarquía de colecciones en Java:

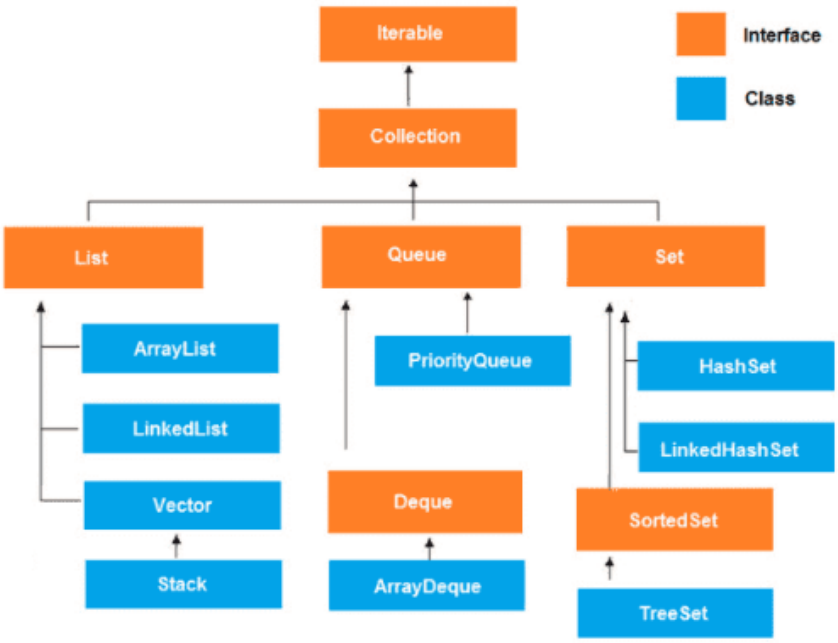

Figure 1: Jerarquía de Colecciones en Java.

Con el diagrama presentado, podemos observar cada una de las interfaces y clases que se conforman en dicha jerarquía. Como siguiente se procederá a explicar en detalle las clases presentadas, sus métodos principales, sus diferencias entre ellas y los aspectos más relevantes al momento de elegir alguno de ellos.

1 Colecciones de Java: Lista (List)

Para el apartado de la lista, tenemos que una lista es una colección ordenada de elementos que pueden contener en ellos elementos duplicados. Es una interfaz que amplía la interfaz de colección, dichas listas se clasifican además en lo siguiente:

1. **Lista de arreglo (Array List)**
2. **Lista enlazada (Linked List)**
3. **Vectores (Vector)**

- Lista de arreglo (Array List)

Lista de arreglo o Array List es la implementación de List Interface donde los elementos se pueden agregar o eliminar de forma dinámica en la lista. A su vez, el tamaño de la lista aumenta dinámicamente si los elementos se agregan más que el tamaño que posee en un inicio.

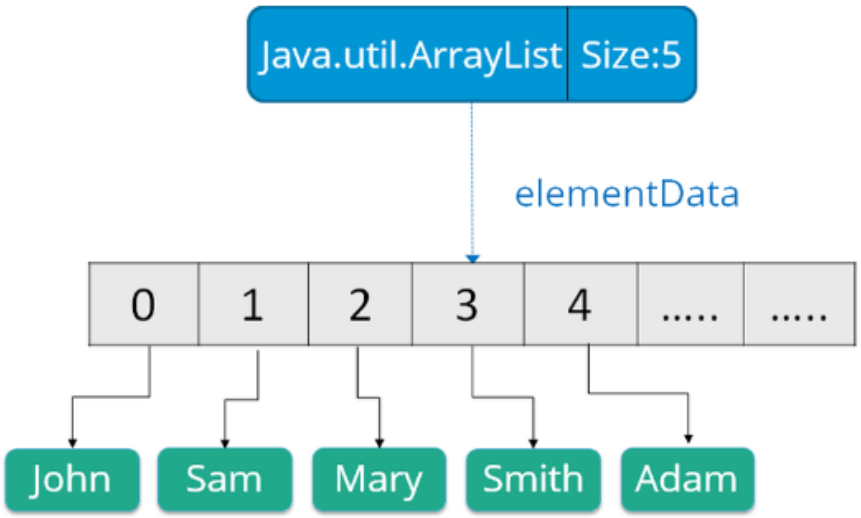

Figure 2: Forma gráfica de la utilería de Java ArrayList.

Forma para crear un Array List

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.ArrayList;

A continuación se presenta la forma para declarar un ArrayList:

$$ArrayList < [Tipodedato] > [Identificador] = newArrayList < [Tipodedato] > ([Tamano]);$$

Donde se tiene que:

- **Tipo de dato:** se definirá el tipo de dato contendra cada uno de los elementos del ArrayList.
- **Identificador:** se definirá su nombre.
- **Tamano:** se le designará el tamaño que tendrá.

Métodos principales en Array List

- **add():** Añade un elemento a el ArrayList, de forma que este se añade hasta el final. Como parámetro se le da el elemento a insertar.

  Ejemplo:

```
ArrayList<String> al = new ArrayList<String>();

al.add("Luis");
\textcolor{green}{al.add("Victor");
al.add("Elena");
```

- **remove():** Borra un elemento del ArrayList. Como parámetro se envía el índice del elemento a borrar.

  Ejemplo:

```
ArrayList<String> al = new ArrayList<String>();

al.add("Victor");
al.add("Luis");
al.add("Elena");

al.remove(1);
```

- **clear():** Limpia el ArrayList de elementos.

  Ejemplo:

```
ArrayList<String> al = new ArrayList<String>();

al.add("Victor");
al.add("Luis");
al.add("Elena");
al.remove(1);

al.clear();
```

- **size():** Nos devuelve el tamaño o número de elementos del ArrayList.

  Ejemplo:

```
ArrayList<Integer>arrlist = new ArrayList<Integer>();

arrlist.add(1);
arrlist.add(2);
arrlist.add(3);
arrlist.add(4);
arrlist.add(5);

System.out.println("Tamano de la lista = " + arrlist.size());
```

- **isEmpty():** Verifica si una lista está vacía o no. Devuelve verdadero si la lista no contiene elementos; de lo contrario, devuelve falso si la lista contiene algún elemento.

Ejemplo:

```
List<Integer> arr = new ArrayList<Integer>(10);

boolean ans = arr.isEmpty();

if(ans == true){
    System.out.println("La lista esta vacia :( ");
}
else{
    System.out.println("La lista no esta vacia :) ");
}
```

- **indexOf():** Nos devuelve el índice de la primera aparición del elemento especificado en el ArrayList, o -1 si no se contiene el elemento. Como parámetro se envía el elemento a buscar dentro del ArrayList.

  Ejemplo:

```
ArrayList<Integer> arr = new ArrayList<Integer>(5);

arr.add(1);
arr.add(2);
arr.add(3);
arr.add(4);

int posicion = arr.indexOf(3);

System.out.println("El elemento 3 se encuentra en el indice : " +
    posicion);
```

- **get():** Nos devuelve el elemento en el índice indicado. Como parámetro se le da el índice del elemento que queremos.

  Ejemplo:

```
ArrayList<Integer> lista = new ArrayList<>();

lista.add(1);
lista.add(2);
lista.add(3);

System.out.println("El primer elemento es: "+ lista.get(0));
```

- Lista enlazada (Linked List)

La lista vinculada o Linked List, es una secuencia de vínculos que contiene elementos. Cada enlace contiene una conexión a otro enlace.

La clase LinkedList en Java utiliza 2 tipos de listas vinculadas para almacenar los elementos:

1. **Lista individualmente vinculada:** En una lista enlazada individual, cada nodo de esta lista almacena los datos del nodo y un puntero o referencia al siguiente nodo de la lista.

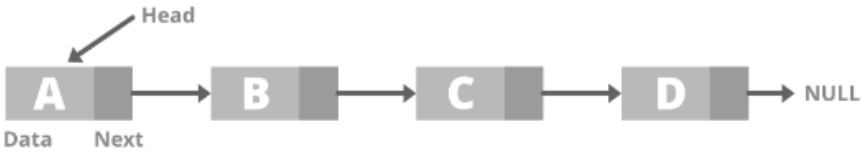

Figure 3: Ejemplo gráficdo de una lista individualmente vinculada.

2. **Lista doblemente enlazada:** En una lista doblemente enlazada, tiene dos referencias, una al nodo siguiente y otra al nodo anterior.

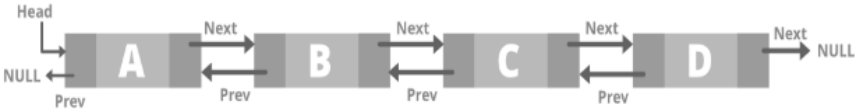

Figure 4: Ejemplo gráficdo de una lista doblemente enlazada

Forma para crear un LinkedList

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.LinkedList;

A continuación se presenta la forma para declarar un LinkedList:

$$LinkedList < [Tipodedato] > [Identificador] = newLinkedList < [Tipodedato] > ([Tamano]);$$

Donde se tiene que:

- **Tipo de dato:** Se definirá el tipo de dato contendrá cada uno de los elementos de la LinkedList.
- **Identificador:** Se definirá su nombre.
- **Tamano:** Se le designará el tamaño que tendrá.

Métodos principales en Linked List

- **add(Objeto):** Nos permite agregar un elemento al final de LinkedList.
- **add(int index, Object):** Permite agregar un elemento en un índice en específico en LinkedList.

  Ejemplo:

```
LinkedList<String> LL = new LinkedList<>();

LL.add("Hola");
LL.add(1, "Mundo");
```

- **get():** Nos permite buscar o recuperar un elemento de un índice en específico de LinkedList. Como parámetros se le da el índice del elemento que deseamos obtener. Este retorna el elemento encontrado en esa posición.

  Ejemplo:

```
LinkedList<String> list = new LinkedList<String>();

list.add("Esto");
list.add("␣es");
list.add("␣una␣prueba");
list.add("␣10");

System.out.println("El␣elemento␣en␣la␣posicion␣2␣es:␣"+ list.get
    (2));
```

- **set():** Dado que una LinkedList está indexada, el elemento que deseamos cambiar está referenciado por el índice del elemento. Por lo tanto, este método toma un índice y el elemento actualizado que debe insertarse en ese índice.

  Ejemplo:

```
LinkedList<String> LL = new LinkedList<>();

LL.add("Hola");
LL.add(1, "Mundo");

ll.set(1, "a␣todos");
```

- **remove(Object):** Este método se utiliza para eliminar un objeto de LinkedList. De forma que si hay varios objetos de este tipo, se elimina la primera aparición del objeto mandado como parámetro.
- **remove(int index):** Dado que una LinkedList está indexada, este método toma un valor entero que simplemente elimina el elemento presente en ese índice específico en LinkedList. Después de eliminar el elemento, todos los elementos se mueven hacia la izquierda para llenar el espacio y se actualizan los índices de los objetos.

Ejemplo:

```
LinkedList<String> LL = new LinkedList<>();

LL.add("Hola");
LL.add(" a");
LL.add(" todos :)");

LL.remove(1);
LL.remove(" todos");
```

- **contains():** Se utiliza para verificar si un elemento se encuentra dentro de una LinkedList creada. De forma que toma el elemento como parámetro, devuelve verdadero si el elemento se encuentra, en caso contrario, devuelve falso.

  Ejemplo:

```
LinkedList<String> LL = new LinkedList<>();

LL.add("Hola");
LL.add( " a");
LL.add( " todos :)");

if(LL.contains("Hola") == true){
    System.out.println("El elemento 'Hola' se encuentra en la
        lista :) ");
}
else{
    System.out.println("El elemento no esta en la lista :( ");
}
```

- **peek():** Examina el elemento que se encuentra en el encabezado de la lista, se forma que se recupera mas no se elimina.

- **peekFirst():** Similar al método anterior, recupera el primer elemento de esta lista, o devuelve nulo si la lista se encuentra vacía.

- **peekLast():** Nos permite recuperar el último elemento que se encuentre en la lista, o devuelve nulo si la lista se encuentra vacía.

  Ejemplo:

```
LinkedList<String> LL = new LinkedList<>();

LL.add("Hola");
LL.add( " a");
LL.add( " todos :)");

System.out.println("Encabezado de la lista : " +LL.peek());

System.out.println("Primer elemento de la lista : " +LL.peekFirst
    ());

System.out.println("Ultimo elemento de la lista : " + LL.peekLast
    ());
```

- **clone():** Se utiliza para crear una copia superficial de la lista vinculada. No necesita de un parámetro para su funcionamiento, puesto que solo devuelve una copia de la instancia de la lista.

  Ejemplo:

```
LinkedList<String> list = new LinkedList<String>();

list.add("Esto");
list.add(" es");
list.add(" una prueba");
list.add(" 10");

LinkedList sec_list = new LinkedList();

sec_list = list.clone();
```

- Vectores (Vector):

Para el caso de los vectores, estos llegan a ser semejantes a las matrices, donde en ellos se puede acceder a los elementos del objeto vectorial, esto por medio de un índice en el vector. Como tal, el vector implementa una matriz dinámica, como también, dicho vector no limita su tamaño.

Si nos damos cuenta, el vector es similar a ArrayList, pero con 2 diferencias:

1. El vector está sincronizado.
2. El vector contiene muchos métodos heredados que no forman parte del marco de las colecciones.

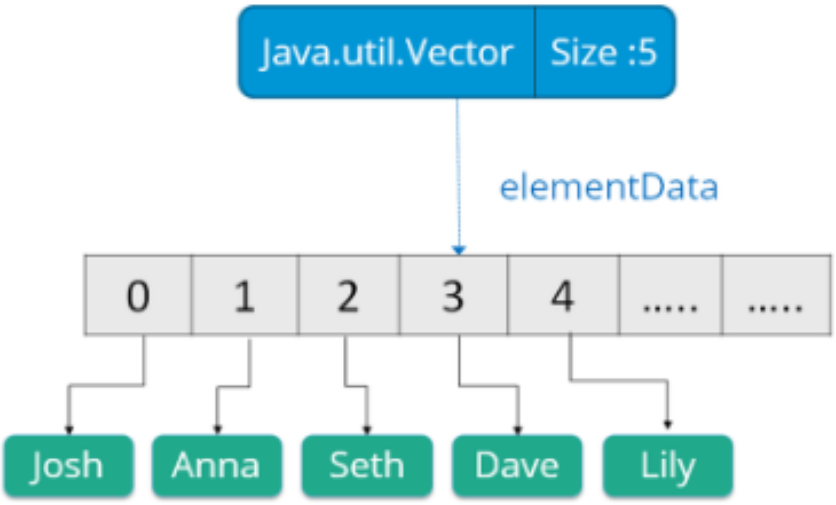

Figure 5: Ejemplo gráfico de la utilería Java Vector.

Forma para crear un Vector

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.Vector;

A continuacion se presenta la forma para declarar un LinkedList:

$Vector < [Tipodedato] > [Identificador] = newVector < [Tipodedato] > ([Tamano]);$

Donde se tiene que:

- **Tipo de dato:** Se definirá el tipo de dato contendrá cada uno de los elementos del Vector.
- **Identificador:** Se definirá su nombre.
- **Tamano:** Se le designará el tamaño que tendrá.

Métodos principales de Vector

- **add():** Nos permite añadir un elemento especificado al final de un vector creado, todo esto mediante el aumento del tamaño del vector por 1.
- **add(int index, Object):** El método inserta un elemento en un índice especificado en el vector, de forma que desplaza el elemento que se encuentra actualmente en esa posición, y cualquier elemento posterior a la derecha.

  Ejemplo:

```
Vector<String> vec_tor = new Vector<String>();

vec_tor.add("Hola, ");
vec_tor.add("esto es ");
vec_tor.add("otra prueba :) ");
vec_tor.add(1, " . Como estas? ");
```

- **clear():** Se utiliza para eliminar todos los elementos de un vector, es necesario mencionar que el método solo borra los elementos mas no el vector.

  Ejemplo:

```
Vector<String> vec_tor = new Vector<String>();

vec_tor.add("Hola, ");
vec_tor.add("esto es ");
vec_tor.add("otra prueba :) ");

vec_tor.clear();

System.out.println("El vector se encuentra vacio: " + vec_tor);
```

- **remove():** Nos permite eliminar un elemento de un vector mediante una posición o índice dado.

  Ejemplo:

```
Vector<String> vec_tor = new Vector<String>();

vec_tor.add("Hola, ");
vec_tor.add("esto es ");
vec_tor.add("otra prueba :) ");

vec_tor.remove(2);
```

- **contains():** Nos permite verificar si un elemento en específico se encuentra en el vector. Como parámetros se le da un elemento del tipo de vector por buscar. Retorna verdadero en caso de encontrar el elemento, caso contrario devuelve falso.

  Ejemplo:

```
Vector<String> vec_tor = new Vector<String>();

vec_tor.add("Bienvenidos ");
vec_tor.add("sean ");
vec_tor.add("todos ");

if(vec_tor.contains(":)") == true){
    System.out.println("El elemento esta en el vector :) ");
}
else{
        System.out.println("El elemento no esta en el vector :( ")
            ;
}
```

- **size():** Se utiliza para conocer el tamaño o el número de elementos presentes en el vector. No necesita de un parámetro para su funcionamiento. Retorna el tamaño o la cantidad de elementos en el vector.

  Ejemplo:

```
Vector<Integer> vec_tor = new Vector<Integer>();

vec_tor.add(15);
vec_tor.add(4);
vec_tor.add(98);
vec_tor.add(32);

System.out.println("Tamano del vector:  " + vec_tor.size());
```

- **indexOfObject():** Nos permite verificar y encontrar la ocurrencia de un elemento dentro del vector. Como parámetro se envía el elemento del tipo vector. Si el elemento está presente, se devuelve el índice de su primera aparición en el vector, en caso contrario devuelve -1.

  Ejemplo:

```
Vector<Integer> vec_tor = new Vector<Integer>();

vec_tor.add(15);
vec_tor.add(4);
vec_tor.add(98);
vec_tor.add(32);
vec_tor.add(4);

System.out.println("Indice de la primera aparicion del elemento
    4: :" + vec_tor.indexOf(4) );
```

- Stack (Pila):

En el marco de Java Collection, se proporciona la clase Stack (Pila) que modela e implementa una estructura de datos Stack. Un Stack es una estructura de datos lineal que solo tienen un único punto de acceso fijo por el cual se añaden, eliminan o se consultan elementos. El modo de acceso a los elementos es de tipo LIFO (Last In First Out, último en entrar, primero en salir).

También se puede decir que la clase extiende Vector y trata la clase como una pila, de esta manera, la clase también puede denominarse subclase de Vector.

Forma para crear un Stack

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.*;

Para este caso en particular donde la clase Stack, se deriva de la clase Vector, se debe realizar lo siguiente:

$$publicclassStack < E > extendsVector < E >$$

A continuación se presenta la forma para declarar un Stack:

$$Stack < [Tipodedato] > [Identificador] = newStack < [Tipodedato] > ([Tamano]);$$

Donde se tiene que:

- **Tipo de dato:** Se definirá el tipo de dato contendrá cada uno de los elementos del Stack.
- **Identificador:** Se definirá su nombre.
- **Tamano:** Se le designará el tamaño que tendrá.

Métodos principales de Stack

- **push():** Permite agregar un elemento a la pila, de forma que coloca el elemento en la parte superior de la pila. Como parámetro acepta un elemento de tipo Pila y se refiere al elemento del cual será insertado.

  Ejemplo:

```
Stack<String> pila = new Stack<String>();

pila.push("Hola ");
pila.push("esto es ");
pila.push("una prueba :) ");

System.out.print(pila);
```

- **search():** Permite buscar un elemento dentro de la pila, de tal forma que se obtiene su distancia desde la parte superior. Como parámetro se ingresa el elemento del cual se buscará. El método devuelve la posición del elemento si se encuentra en la pila, en caso de no encontrarlo, se devolverá un -1.

  Ejemplo:

```
Stack<String> pila = new Stack<String>();

pila.push("Hola ");
pila.push("esto es ");
pila.push("una prueba :) ");

System.out.println("Posicion del elemento 'Hola ': " + pila.search
    ("Hola"));
```

- **pop():** Permite extraer un elemento de la pila. El elemento se saca de la parte superior de la pila y se quita de la misma. El método no necesita de un parámetro para su funcionamiento. De esta forma, devuelve el elemento del cual será quitado, para posteriormente eliminarlo de la pila.

  Ejemplo:

```
Stack<String> pila = new Stack<String>();

pila.push("Hola ");
pila.push("esto es ");
pila.push("una prueba :) ");

System.out.println("Se ha quitado el elemento: " + pila.pop());

System.out.print(pila);
```

- **empty():** Permite verificar si una pila está vacía o no. El método es de tipo booleano y devuelve verdadero si la pila está vacía, de lo contrario, devolverá falso.

  Ejemplo:

```
Stack<String> pila = new Stack<String>();

System.out.println("La pila se encuentra vacia? " + pila.empty())
    ;
```

2 Colecciones de Java: Cola (Queue)

En Java, la estructura de Cola ordena los elementos de la manera FIFO (First In, First Out ó Primero en entrar, Primero en salir). En la Cola, el primer elemento ingresado se elimina primero, y el último se elimina al final.

Los métodos básicos de la Cola existe de 2 formas:v

1. Se lanza una excepción si se falla la operación.
2. Devuelve un valor especial.

	Throws Exception	Returns Special Value
Insert	Add(e)	Offer(e)
Remove	Remove()	Poll()
Examine	Element()	Peek()

Figure 6: Pie de pagina.

Para este caso en cuestión, la cola prioridad se implementa en la interfaz de Cola, de esta manera, cada elemento se ordena de acuerdo con su orden natural, o por un comparador proporcionado en el momento de la construcción de la cola.

Forma para crear un Queue

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.Queue;

A continuación se presenta la forma para declarar un LinkedList:

$Queue < [Tipodedato] > [Identificador] = newQueue < [Tipodedato] > ([Tamano]);$

Donde se tiene que:

- **Tipo de dato:** Se definirá el tipo de dato contendrá cada uno de los elementos del Queue.
- **Identificador:** Se definirá su nombre.
- **Tamano:** Se le designará el tamaño que tendrá.

Métodos principales de Cola (Queue)

- **add():** Inserta un elemento pasado en el parámetro al final de la cola en caso de haber espacio. Si la cola al momento de insertar un elemento se encuentra en su capacidad límite, devolverá una excepción.

  Ejemplo:

```
Queue<Integer> cola = new LinkedList<Integer>();

cola.add(444);
cola.add(356);
cola.add(890);

System.out.println("Cola: " + cola);
```

- **offer():** Permite insertar un el elemento especificado en la cola si es posible hacerlo inmediatamente sin violar las restricciones de su capacidad. Este método es preferible al método add(), ya que este método no lanza una excepción cuando la capacidad del contenedor está llena, puesto que solo devuelve un falso.

  Ejemplo:

```
Queue<Integer> cola = new LinkedList<Integer>(3);

cola.add(444);
cola.add(356);
cola.add(890);

if(cola.offer(250)){
    System.out.println("Se ha insertado el elemento 250 en la
        cola :) ");
}
else{
    System.out.println("La cola se encuentra llena :( ");
}
```

- **remove():** Devuelve y elimina el elemento al frente de la cola. El método regresa una excepción en caso de que la cola esté vacía.

  Ejemplo:

```
Queue<Integer> cola = new LinkedList<Integer>();

cola.add(444);
cola.add(356);
cola.add(890);
cola.add(777);

System.out.println("Elemento eliminado de la cabeza de la cola: "
    + cola.remove());
```

- **poll():** Similar al método anterior, devuelve y elimina el elemento al frente de la cola. La diferencia de este método respecto a remove(), se debe a que este no lanza una excepción cuando la cola está vacía, en su lugar devuelve un valor nulo.

Ejemplo:

```
Queue<Integer> cola = new LinkedList<Integer>();

cola.add(444);
cola.add(356);
cola.add(890);
cola.add(777);

System.out.println("Elemento eliminado de la cabeza de la cola: "
    + cola.poll());
```

- **element():** Devuelve el elemento que se encuentre al frente de la cola, es necesario mencionar que dicho elemento no lo elimina. Su funcionamiento es semejante a peek(), pero su diferencia radica en que solo lanza una excepción si la cola está vacía.

  Ejemplo:

```
Queue<Integer> cola = new LinkedList<Integer>();

cola.add(444);
cola.add(356);
cola.add(890);
cola.add(777);

System.out.println("Elemento en la cabeza de la cola: " + cola.
    element());
```

3 Colecciones de Java: Conjuntos (Sets)

Un conjunto se refiere a una colección que no puede contener elementos duplicados. Su uso se enfoca principalmente en modelar la abstracción de conjuntos matemáticos. Para este caso Set tiene su implementación en varias clases como:

1. **Hashsests**
2. **LinkedHashet**
3. **TreeSet**

- HashSet

En Java, la clase HashSet crea una colección que usa una tabla hash para su almacenamiento. HashSet solo puede contener elementos únicos, a su vez, hereda la clase AbstractSet e implementa la interfaz Set.

Forma para crear un HashSet

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.HashSet;

A continuación se presenta la forma para declarar un HashSet:

$$HashSet < [Tipodedato] > [Identificador] = newHashSet < [Tipodedato] > ([Tamano]);$$

Donde se tiene que:

- **Tipo de dato:** Se definirá el tipo de dato contendrá cada uno de los elementos del HashSet.
- **Identificador:** Se definirá su nombre.
- **Tamano:** Se le designará el tamaño que tendrá.

Métodos principales de Hashset

- **add():** Permite añadir o insertar elementos al Hashset, sin embargo, la orden de inserción no se conserva. Es necesario tener en cuenta que los elementos duplicados no están permitidos, por lo que cada elemento duplicado que se encuentre se ignorará.

  Ejemplo:

```
HashSet<String> hs = new HashSet<String>();

hs.add("A");
hs.add("B");
hs.add("C");
hs.add("D");
```

- **contains():** Verifica si un elemento específico se encuentra dentro del HashSet. Como parámetro se envía un elemento del tipo HashSet. Retorna verdadero si el elemento se encuentra, en caso contrario devuelve falso.

  Ejemplo:

```
HashSet<String> hs = new HashSet<String>();

hs.add("A");
hs.add("B");
hs.add("C");
hs.add("D");

if(hs.contains("D") == true){
    System.out.println("La letra 'D' esta presente :) ");
}
else{
        System.out.println("La letra 'D' no esta :( ");
}
```

- **clear():** Permite eliminar todos los elementos de un HashSet. El uso de este método solo borra todos los elementos del conjunto, mas no elimina al conjunto.

  Ejemplo:

```
HashSet<String> hs = new HashSet<String>();

hs.add("A");
hs.add("B");
hs.add("C");
hs.add("D");

System.out.println("Conjunto despues de usar clear(): " + hs.
    clear());
```

- **isEmpty():** Permite verificar si un Hashset se encuentra vacío. Este devuelve verdadero si el HashSet está vacío, en caso contrario devolverá un falso.

  Ejemplo:

```
HashSet<String> hs = new HashSet<String>();

hs.add("A");
hs.add("B");
hs.add("C");
hs.add("D");

if(hs.isEmpty() == true){
    System.out.println("El conjunto esta vacio :) ");
}
else{
    System.out.println("El conjunto tiene elementos :( ");
}
```

- **remove():** Permite eliminar un elemento especificado dentro del HashSet.

  Ejemplo:

```
HashSet<String> hs = new HashSet<String>();

hs.add("A");
hs.add("B");
hs.add("C");
hs.add("D");

hs.remove("D");
```

- **clone():** Se utiliza para devolver una copia superficial del conjunto de hash. No necesita de ningún parámetro para su funcionamiento.

  Ejemplo:

```
HashSet<Integer> hs = new HashSet<Integer>();

hs.add(1);
hs.add(2);
hs.add(3);
hs.add(4);

HashSet hs_2  = new HashSet();

hs_2 = hs.clone();

System.out.println("Nuevo conjunto clonado: " + hs_2);
```

- **size():** Permite obtener el tamaño, o conocer la cantidad de elementos que se encuentran en el HashSet.

  Ejemplo:

```
HashSet<Integer> hs = new HashSet<Integer>();

hs.add(1);
hs.add(2);
hs.add(3);
hs.add(4);

System.out.println("Tamano del conjunto: " + hs.size());
```

- LinkedHashSet (HashSet vinculado)

En Java, HashSet vinculado o LinkedHashSet es una tabla Hash con una implementación de lista vinculada de la interfaz Set. Contiene elementos únicos como HashSet, también, proporciona cada uno de los métodos HashSet, manteniendo el orden de inserción.

Forma para crear un LinkedHashSet

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.LinkedHashSet;

A continuación se presenta la forma para declarar un LinkedHashSet:

$$LinkedHashSet < [Tipodedato] > [Identificador] = newLinkedHashSet < [Tipodedato] > ([Tamano]);$$

Donde se tiene que:

- **Tipo de dato:** Se definirá el tipo de dato contendrá cada uno de los elementos del LinkedHashSet.
- **Identificador:** Se definirá su nombre.
- **Tamano:** Se le designará el tamaño que tendrá.

- TreeSet

La clase TreeSet implementa la interfaz Set, del cual usa un árbol binario para su almacenamiento. Cada uno de los objetos se almacenan en orden ascendente. Dicha clase hereda la clase AbstractSet e implementa la interfaz NavigableSet. Al igual que las anteriores clases, este solo puede contener elementos únicos. La clase TreeSet llega a ser mucho más rápida en el tiempo de acceso y recuperación de los elementos.

Forma para crear un TreeSet

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.Set;

A continuación se presenta la forma para declarar un TreeSet:

$$Set < [Tipodedato] > [Identificador] = newTreeSet < [Tipodedato] > ([Tamano]);$$

Donde se tiene que:

- **Tipo de dato:** Se definirá el tipo de dato contendrá cada uno de los elementos del TreeSet.
- **Identificador:** Se definirá su nombre.
- **Tamano:** Se le designará el tamaño que tendrá.

Métodos principales de TreeSet

- **add():** Permite agregar un elemento en específico dentro de un TreeSet. El método agrega al elemento sólo si el elemento no se encuentra, caso contrario el método devuelve un falso.
- **addAll():** Permite agregar todos los elementos de la colección al conjunto creado. Cada uno de los elementos se agregan aleatoriamente sin seguir ningún orden específico.

  Ejemplo:

```
TreeSet<String> tree = new TreeSet<String>();

tree.add("Hola ");
tree.add("esto es ");
tree.add("una prueba ");

TreeSet<String> tree_2 = new TreeSet<String>();

tree_2.add("Bienvenidos ");
tree_2.add("sean todos ");

tree.addAll(tree_2);

System.out.println("TreeSet: " + tree);
```

- **contains():** Se utiliza para comprobar si un elemento específico está presente en el TreeSet. Devuelve verdadero si el elemento se encuentra dentro del TreeSet, en caso contrario devuelve un falso.

- **containsAll():** Permite comprobar si dos conjuntos contienen los mismos elementos o no. Toma un conjunto como parámetro y devuelve verdadero si todos los elementos de este conjunto están presentes en el otro conjunto

  Ejemplo:

```
TreeSet<Integer> set = new TreeSet<Integer>();

set.add(12);
set.add(99);
set.add(42);
set.add(30);

System.out.println("El conjunto contiene al 99? "+ set.contains
    (99));

TreeSet<Integer> set_2 = new TreeSet<Integer>();

set_2.add(12);
set_2.add(99);
set_2.add(42);
set_2.add(30);

System.out.println("El conjunto 1 contiene al conjunto 2?  " +
    set.containsAll(set_2));
```

- **isEmpty():** Permite comprobar y verificar si un TreeSet se encuentra vacío. Retorna verdadero si el TreeSet está vacío, en caso contrario devuelve falso.

  Ejemplo:

```
TreeSet<String> tree = new TreeSet<String>();

if(tree.isEmpty() == true){
     System.out.println("El conjunto esta vacio :) ");
}
else{
        System.out.println("El conjunto tiene elementos :( ");
}
```

- **remove():** Permite eliminar un elemento en particular de un TreeSet. Como parámetros se le da el elemento del cual se eliminará del conjunto. Retorna verdadero si se elimina el elemento dado del conjunto, de lo contrario devuelve falso.

  Ejemplo:

```
TreeSet<Integer> set = new TreeSet<Integer>();

set.add(12);
set.add(99);
set.add(42);
set.add(30);

sett.remove(42);

System.out.println("Nuevo conjunto despues de haber eliminado el
    elemento 42: " + set);
```

- **clear():** Permite eliminar todos los elementos de un TreeSet. Al usar este método borra únicamente los elementos del conjunto, mas no el conjunto.

  Ejemplo:

```
TreeSet<Integer> set = new TreeSet<Integer>();

set.add(12);
set.add(99);
set.add(42);
set.add(30);

set.clear();

System.out.println("Conjunto despues haber eliminado sus
    elementos: " + set);
```

- **clone():** Devuelve una copia superficial del conjunto de arboles.

  Ejemplo:

```
TreeSet<String> tree = new TreeSet<String>();

tree.add("Hola ");
tree.add("esto es ");
tree.add("una prueba ");

TreeSet tree_2 = new TreeSet();

tree_2 = tree.clone();

System.out.println("Conjunto clonado: " + tree_2);
```

- **fist():** Permite devolver el primero de los elementos de un TreeSet. El primer elemento aquí se refiere al más bajo de los elementos del conjunto, puesto que nos encontramos en un conjunto de árboles. Es necesario mencionar que si los elementos son de tipo entero, se devuelve el número entero más pequeño; en caso de ser elementos de tipo string, los elementos se verifican en orden alfabético.

- **last():** Permite devolver el último elemento de un TreeSet. El último elemento aquí se refiere al más alto de los elementos del conjunto. Es necesario mencionar

que si los elementos son de tipo entero, se devuelve el entero más grande; en caso de ser elementos de tipo string, los elementos se verifican en orden alfabético

Ejemplo:

```
TreeSet<Integer> tree = new TreeSet<Integer>();

tree.add(500);
tree.add(12);
tree.add(210);
tree.add(98);
tree.add(7);

System.out.println("El primer elemento del conjunto es: " + tree.
    first());

System.out.println("El ultimo elemento del conjunto es: " + tree.
    last());
```

- **size():** Nos permite obtener el tamaño del conjunto de árboles, o la cantidad de elementos presentes en él. No necesita de un parámetro para su funcionamiento.

  Ejemplo:

```
TreeSet<Integer> tree = new TreeSet<Integer>();

tree.add(500);
tree.add(12);
tree.add(210);
tree.add(98);
tree.add(7);

System.out.println("Tamano del conjunto: " + tree.size());
```

+ Interfaz de Mapa

La interfaz Mapa no pertenece como un subtipo de la interfaz de Colección. Como tal la interfaz Map, representa un mapeo entre una clave y un valor. Por lo tanto, esta se comporta distinta al resto de tipos de colecciones de Java. Es necesario mencionar que un mapa solo contiene claves únicas.

Forma para crear un HashMap

Se debe tomar en cuenta que para su creación se debe importar la biblioteca:

import java.util.HashMap;

A continuación se presenta la forma para declarar un Map

$$Map < [Clave][Valor] > [Identificador] = newHashMap < [Tipodedato] > ([Tamano]);$$

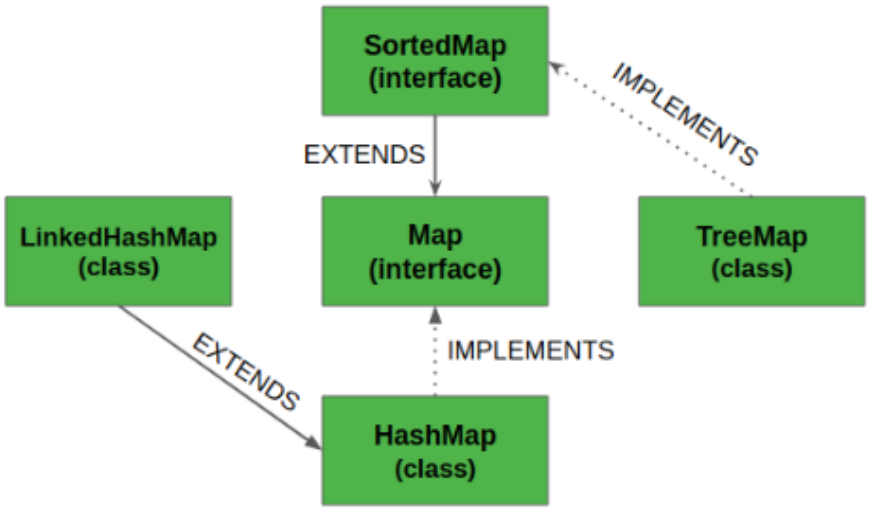

Figure 7: Relación de la interfaz de Mapas con las colecciones de Java.

Donde se tiene que:

- **Clave:** El tipo de clave que tendrá el mapa.
- **Valor:** El tipo de valores mapeados.
- **Tamaño:** Se le designará el tamaño que tendrá.
- **Identificador:** Se definirá su nombre.

Métodos principales de HashMap

- **put():** Permite asociar un valor especificado con una clave del mapa. Como parámetros, se requiere de 2 argumentos (clave y valor), donde la clave es el primer argumento ingresado, y el segundo argumento es el valor correspondiente de la clave en el mapa. A su vez, el método devuelve el valor anterior asociado con la clave si esta se encuentra presente, en caso de no serlo, devolverá un -1.

  Ejemplo:

```
Map<Integer, String> map = new HashMap<>();

map.put(1, "Uno");
map.put(2, "Dos");
map.put(3, "Tres");

System.out.println(map);
```

- **remove():** Permite eliminar el mapeo de una clave. Como argumento, se le asigna la clave del argumento, del cual se procederá a eliminar.

  Ejemplo:

```
Map<Integer, String> map = new HashMap<>();

map.put(1, "Uno");
map.put(2, "Dos");
map.put(3, "Tres");

map.remove(2);

System.out.println(map);
```

- **containsKey():** Permite verificar si una clave en particular se encuentra mapeado, de forma que toma el elemento de la clave como parámetro y devuelve verdadero si ese elemento está mapeado, en caso contrario devolverá un falso.

  Ejemplo:

```
Map<Integer, String> map = new HashMap<>();

map.put(12, "Hola");
map.put(24, "Esto es");
map.put(31, "una prueba");

System.out.println("Se encuentra la clave 24? "+map.containsKey
    (24));
```

- **clear():** Permite eliminar todos los elementos de una colección de mapas.

  Ejemplo:

```
Map<Integer, String> map = new HashMap<>();

map.put(12, "Hola");
map.put(24, "Esto es");
map.put(31, "una prueba");

map.clear();

System.out.println(map);
```

- **hashCode():** Permite generar un código hash para el mapa dado de claves y valores. No necesita de argumentos para su funcionamiento. El método devuelve el valor HashCode para el mapa especificado.

  Ejemplo:

```
Map<Integer, String> map = new HashMap<>();

map.put(12, "Hola");
map.put(24, "Esto es");
map.put(31, "una prueba");
map.put(7, ":D");

int hash = map.hashCode();

System.out.println("Codigo Hash del mapa: "+hash);
```

- **entrySet():** Permite crear un conjunto de los mismos elementos del mapa especificado. No necesita de parámetros para su funcionamiento. El método devuelve un conjunto que tiene los mismos elementos que el mapa especificado.

  Ejemplo:

```
Map<Integer, String> map = new HashMap<>();

map.put(12, "Hola");
map.put(24, "Esto es");
map.put(31, "una prueba");
map.put(7, ":D");

System.out.println(map.entrySet());
```

- **equals():** Permite verificar la igualdad entre 2 mapas, de forma que verifica si los elementos de un mapa pasados como parámetro son iguales a los elementos del mapa especificado. Como parámetro, se ingresa el objeto de tipo mapa del cual se verifica la igualdad. El método devuelve verdadero si la igualdad existe entre ambos mapas, en caso contrario devolverá un falso.

  Ejemplo:

```
Map<Integer, String> map_1 = new HashMap<>();
Map<Integer, String> map_2 = new HashMap<>();

map_1.put(12, "Hola");
map_1.put(24, "Esto es");
map_1.put(31, "una prueba");
map_1.put(7, ":D");

map_2.put(1, "Uno");
map_2.put(2, "Dos");
map_2.put(3, "Tres");

System.out.println("El mapa 1 es igual al mapa 2? "+map_1.equals(
    map_2));
```

Análisis del programa: Escuderia Puma

En el proyecto presente se nos propuso la elaboración de un programa que hiciera uso de las utilerías existentes en Java, siendo que el enfoque estaba dirigido hacia el uso de listas, pilas, colas y tablas hash.

Necesariamente, la propuesta de codificación tenía que incluir la posibilidad de realizar las siguientes operaciones:

- Crear un Campeonato:
  - Registrar Pilotos (con sus respectivos equipos)
  - Registrar Pistas
  - Calendario de carreras
- Iniciar una carrera (asociar una pista con los pilotos, y asignar una clave única)
  - Al finalizar la carrera se sumarán los puntos dependiendo de la posición de los pilotos
- Consultar carreras anteriores
- Opciones para Mostrar Elementos:
  - Equipos con Información de los pilotos
  - Resultados de las carreras
  - Posiciones del campeonato

Por ello, nuestro código realiza una propuesta para cubrir dichos requerimientos.

Para iniciar con el código, fue necesario presentar mediante la impresión en pantalla una introducción pertinente para el usuario, la cual cuenta con las siguientes opciones a su disposición:

(1) - Crear un campeonato

(2) - Consultar carreras anteriores

(3) - Ver info. General (Corredores, Escuderias, Carreras y Campeonatos)

(4) - Salir

- **Creación de datos iniciales:**

  De inicio se tienen valores predefinidos dentro de las carreras y los corredores, esto para que el usuario no tenga que ser el encargado de crear todos los corredores, dentro del programa iniciar.java es en donde se asignan estos valores, tenemos la creación de 10 objetos escudería, que son 2 corredores dentro de cada una, por lo cual se crean 20 objetos corredor y 22 objetos carrera, una por cada carrera que actualmente estuvo vigente en el campeonata de la f1 de este año.

  Cada uno de estos objetos tiene su clase en la que se tiene la clase constructora correspondiente así como sus setters y getters, estos documentos son:

  - Carreras
  - CalendarioCarreras
  - Corredores
  - Escuderias
  - Pistas

  Dentro de estas es lo estándar, se solicitan diferentes atributos para cada una de ellas.

  - **Carreras:**

    Dentro de carreras se solicita el nombre de la carrera, un objeto calendario Carrera que especificara la fecha de inicio y de final de la carrera, un objeto pista que vendrá con la información de la misma y la lista de escuderías que van a participar en la carrera.

  - **CalendarioCarerras:**

    Se solicita la fecha de inicio y la fecha final de cada carrera, esto se solicita de manera de String del formato aaaa-mm-dd, de ahí se llama a una función que a través del método parse lo convierte en objeto localdate, para que al momento de que el usuario decida crear una carrera se compruebe que la fecha ingresada de inicio o final no exista ya dentro de otra carrera.

  - **Corredores:**

    Los atributos solicitados en corredores son:

    * nacionalidad
    * nombre
    * apellido
    * número
    * peso
    * estatura

    Los cuales son los mismos que se muestran cuando el usuario desea saber información de la escudería.

– **Escuderias:**

Se solicita el nombre de la escudería así como la lista con los dos corredores de la misma, esto para un manejo más sencillo

– **Pistas:**

Finalmente en pistas se solicita el país, la ciudad, y la longitud de la pista como datos extra para tener información precisa de donde van a desempeñarse los corredores.

Teniendo todo esto desde un inicio le permite al usuario consultar la información de las escuderías y las carreras antes de siquiera iniciar el campeonato, de igual forma va a poder agregar corredores y pistas.

Se muestra de esta manera la información de las escuderías así como una vez creado el campeonato se puede acceder a la información de las carreras creadas. Estos datos mostrados son los mismos que se le van a solicitar al usuario cuando este desee crear una carrera o crear un corredor, el corredor lo puede crear dentro de una escudería existente, o dentro de una nueva escudería.

```
       [ Menu del simulador ]
(1) - Crear un campeonato
(2) - Consultar carreras anteriores
(3) - Ver info. General (Corredores, Escuderias, Carreras y Campeonatos)
(4) - Salir

Seleecione una opcion: 3

       [ Ver info. General ]
(1) - Info. Corredores y escuderias designadas
(2) - Info. carreras
(3) - Resultados de las carreras
(4) - Posiciones del campeonato
(5) - Salir

Seleecione una opcion: 2

------------------------------------------------------------

       [ Carrera 2: STC Saudi Arabian GP ]

( Fecha de la carrera )

Inicio de la carrera: 2022-03-25

Fin de la carrera: 2022-03-27


( Info. Pista )

Pais: Arabia Saudita

Ciudad: Yeda

Longitud: 6175.0 km

------------------------------------------------------------
```

- **Registrar pilotos y pistas:**

  Para registrar pilotos se tienen dos opciones: crear el corredor dentro de una nueva escudería o crear el corredor en una escudería existente y reemplazarlo con uno de los dos corredores de dicha escudería.

  Si se decide crear el corredor con una nueva escudería se la va a solicitar los atributos tanto de la escudería como del corredor y se va a insertar a la lista ya creada de escuderías, actualizando así los valores, mientras que si se decide dentro de la escudería existente, se le va a mostrar las escuderías existentes al usuario así como cada corredor de la escudería que este seleccione, y según el corredor que desee reemplazar se le va a actualizar los valores para que se tenga el corredor nuevo en dicha escudería.

- **Registrar carrera:**

  Para el registro de la carrera se le va a solicitar los atributos de la carrera, la pista y la fecha, que es tanto la de inicio como la de final, comprobando claro que esta no se empalme con una carrera ya existente, nuevamente se mandan los constructores de los objetos correspondientes en orden (Primero fecha, luego pista y finalmente la carrera) y se actualiza la lista de carreras para que al crear el campeonato se tengan los nuevos valores.

- **Crear campeonato:**

  Ahora si, al crear el campeonato e iniciarlo el usuario tendrá que avanzar por todas las carreras y en esta le va a mostrar el nombre y número de carrera que se encuentra así como la lista de las posiciones en las que quedaron los corredores:

```
Ha iniciado el campeonato F1, buena suerte a todos :)

------------------------------------------------------------
        [ Carrera No. 1 - Gulf Air Bahrein GP ]

Fecha de inicio: 2022-03-18  Fecha de cierre: 2022-03-20

Resultados de la carrera:

1  Verstappen Max Nacionalidad: Paises Bajos Puntos: 25
2  Ocon Esteban Nacionalidad: Francia Puntos: 18
3  Alonso Fernando Nacionalidad: España Puntos: 15
4  Hamilton Lewis Nacionalidad: Gran Bretaña Puntos: 12
5  Zhou Guanyu Nacionalidad: China Puntos: 10
6  Albon Alexander Nacionalidad: Tailandia Puntos: 8
7  Vries Nyck Nacionalidad: Paises Bajos Puntos: 6
8  Vettel Sebastian Nacionalidad: Alemania Puntos: 4
9  Rusell George Nacionalidad: Gran Bretaña Puntos: 2
10  Schumacher Mick Nacionalidad: Alemania Puntos: 1
11  Pérez Sergio Nacionalidad: Mexico Puntos: 0
12  Magnussen Kevin Nacionalidad: Dinamarca Puntos: 0
13  Ricciardo Daniel Nacionalidad: Australia Puntos: 0
14  Tsunoda Yuki Nacionalidad: Japon Puntos: 0
15  Bottas Valtteri Nacionalidad: Finlandia Puntos: 0
16  Gasly Pierre Nacionalidad: Francia Puntos: 0
17  Leclerc Charles Nacionalidad: Monaco Puntos: 0
18  Stroll Lance Nacionalidad: Canada Puntos: 0
19  Sainz Carlos Nacionalidad: España Puntos: 0
20  Norris Lando Nacionalidad: Gran Bretaña Puntos: 0

Ingrese 1 para avanzar a la siguiente carrera:
```

Para conseguir esto se creó una función dentro del documento simulación llamada posiciones, en esta función se extraen los corredores de la lista creada anteriormente que tiene los valores de las escuderías. De ahí estos valores se van a guardar dentro de un treemap al cual se le va a dar como llave un número aleatorio del 1 al 100 000, esto con la intención de que sean ordenados automáticamente, se escogió que fueran esos numero aleatorios por que dentro del treemap si dos llaves son iguales se ignoran los valores:

```
TreeMap<Integer, Corredores> campeonato = new TreeMap<Integer, Corredores>()

for(i=0; i<corredores.size(); i++){
    Integer random = (int)Math.floor(Math.random()*100000);
    campeonato.put(random, corredores.get(i));
}
```

Posteriormente estos valores van a insertarse en un objeto diferente llamado posiciones, en este se va a guardar la posición, el corredor y se le darán los puntos de acuerdo a los puntos establecidos por la F1, esto comprobando en qué orden se encuentran dentro del treemap.

```
for(i = 1; i<=posicion.size(); i++){
    if(i == 1){
        Posiciones lugar1 = new Posiciones(i, posicion.get(i), puntos: 25);
        posiciones.add(lugar1);
    }
    else if(i==2){
        Posiciones lugar2 = new Posiciones(i, posicion.get(i), puntos: 18);
        posiciones.add(lugar2);

    }
    else if(i==3){
        Posiciones lugar3 = new Posiciones(i, posicion.get(i), puntos: 15);
        posiciones.add(lugar3);

    }
```

Cabe aclarar que después del décimo lugar todos los lugares reciben 0 puntos.

Finalmente cuando se hayan recorrido todas las carreras y se hayan dado todas las posiciones dentro de estas se va a mostrar la posición final de todos los corredores junto con sus puntos:

```
1 George Rusell 230
2 Daniel Ricciardo 181
3 Lance Stroll 148
4 Sergio Pérez 136
5 Lando Norris 130
6 Kevin Magnussen 128
7 Carlos Sainz 116
8 Alexander Albon 113
9 Valtteri Bottas 112
10 Lewis Hamilton 107
11 Fernando Alonso 104
12 Charles Leclerc 84
13 Sebastian Vettel 82
14 Pierre Gasly 79
15 Esteban Ocon 78
16 Guanyu Zhou 75
17 Mick Schumacher 66
18 Yuki Tsunoda 65
```

Para poder obtener esto dentro de la lista ya creada anteriormente que contiene las posiciones, se va a recorrer carrera por carrera y posición por posición de cada una con la intención de que cada que encuentre una similitud en cada posición con algún corredor a este se le inserten los puntos correspondientes, llevando así el control de todas las carreras.

```
for(i=0; i<carreras_f1.size(); i++){
    for(j=0; j<corredores.size(); j++){
        for(k = 0; k<corredores.size(); k++){
            if(posicionesFinales.get(k).corredores.getApellido().equals(campeonato.get(i).posiciones.get(j).corredores.getApelli
                posicionesFinales.get(k).setPuntos(campeonato.get(i).posiciones.get(j).puntos);
            }
        }
    }
```

Posteriormente se realiza un nuevo ordenamiento con un nuevo treemap y se repite el proceso para poder imprimirlo y guardar los puntos, cabe aclarar que al tener en este caso como llave los puntos finales de cada corredor, para poder acceder a ellos dentro de un ciclo for se utiliza la colección set, en la cual obtenemos las llaves del treemap y posteriormente estas llaves se convierten a un arreglo para poder insertarlas en el objeto posiciones.

```
Integer[] array = new Integer[ordenarPosiciones.size()];

Set<Integer> keys = ordenarPosiciones.keySet();
array = keys.toArray(array);
```

- **Consultar carreras:**

  Una vez que se termina el campeonato es posible acceder a ver los resultados de las carreras, se va a mostrar una lista de las carreras en las que se participó y se le va a solicitar al usuario de cual carrera desea ver los resultados, una vez seleccionada se van a mostrar las posiciones de las mismas:

```
Ingrese la carrera por consultar: 20

Ha seleccionado la carrera 'Aramco United States GP'

Resultados de la carrera:

1  Stroll Lance Nacionalidad: Canada Puntos: 25
2  Leclerc Charles Nacionalidad: Monaco Puntos: 18
3  Ocon Esteban Nacionalidad: Francia Puntos: 15
4  Albon Alexander Nacionalidad: Tailandia Puntos: 12
5  Ricciardo Daniel Nacionalidad: Australia Puntos: 10
6  Vries Nyck Nacionalidad: Paises Bajos Puntos: 8
7  Pérez Sergio Nacionalidad: Mexico Puntos: 6
8  Rusell George Nacionalidad: Gran Bretaña Puntos: 4
9  Tsunoda Yuki Nacionalidad: Japon Puntos: 2
10  Magnussen Kevin Nacionalidad: Dinamarca Puntos: 1
11  Gasly Pierre Nacionalidad: Francia Puntos: 0
12  Sainz Carlos Nacionalidad: España Puntos: 0
13  Alonso Fernando Nacionalidad: España Puntos: 0
14  Bottas Valtteri Nacionalidad: Finlandia Puntos: 0
15  Zhou Guanyu Nacionalidad: China Puntos: 0
16  Verstappen Max Nacionalidad: Paises Bajos Puntos: 0
17  Vettel Sebastian Nacionalidad: Alemania Puntos: 0
18  Schumacher Mick Nacionalidad: Alemania Puntos: 0
19  Norris Lando Nacionalidad: Gran Bretaña Puntos: 0
20  Hamilton Lewis Nacionalidad: Gran Bretaña Puntos: 0
```

Referencias electronicas:

- ArrayList en Java – Acervo Lima. (s. f.). Recuperado 24 de septiembre de 2022, de https://es.acervolima.com/arraylist-en-java/
- Clase de vector en Java – Acervo Lima. (s. f.). Recuperado 24 de septiembre de 2022, de https://es.acervolima.com/clase-de-vector-en-java-1/
- COLECCIONES JAVA: INTERFAZ, LISTA, COLA, CONJUNTOS EN JAVA CON EJEMPLOS - BLOG. (s. f.). Recuperado 24 de septiembre de 2022, de https://es.quish.tv/java-collections-interface
- Interfaz de mapa en Java – Acervo Lima. (s. f.). Recuperado 30 de septiembre de 2022, de https://es.acervolima.com/interfaz-de-mapa-en-java/
- Interfaz de cola en Java – Acervo Lima. (s. f.). Recuperado 24 de septiembre de 2022, de https://es.acervolima.com/interfaz-de-cola-en-java/
- LinkedList en Java – Acervo Lima. (s. f.). Recuperado 24 de septiembre de 2022, de https://es.acervolima.com/linkedlist-en-java-1/