Objetivo general del taller

El objetivo general de este taller es practicar algorítmica sobre varias estructuras de datos en Java. El desarrollo de este taller necesariamente incluye el uso de algunos conceptos de herencia y el uso de genéricos.

Objetivos específicos del taller

Durante el desarrollo de este taller se buscará el desarrollo de las siguientes habilidades:

1. Recorrer, crear y manipular arreglos de tipos simples y de objetos
2. Recorrer, crear y manipular Listas
3. Recorrer, crear y manipular árboles y conjuntos
4. Recorrer, crear y manipular mapas
5. Poner en práctica diferentes mecanismos para recorrer estructuras, incluyendo los iteradores.
6. Reconocer la utilidad de diferentes estructuras de datos para resolver problemas específicos

Instrucciones generales

1. Descargue de Bloque Neón el archivo n`Taller2-Estructuras_esqueleto.zip` y descomprímalo en una carpeta dentro de su computador a la que pueda llegar con facilidad.
2. Lea con cuidado este documento: en este documento encontrará información importante para completar el programa mientras que va entendiendo lo que está haciendo.
3. Ejecute las pruebas que se encuentran dentro de la carpeta test.
4. Implemente los métodos de las clases que se encuentran en la carpeta src y al final asegúrese de que todas las pruebas corren correctamente.

El taller debe realizarse de forma individual.

Descripción del taller: Sandbox por estructura de datos

En este taller usted tendrá que implementar métodos en cuatro clases: SandboxArreglo, SandboxListas, SandboxConjuntos y SandboxMapas. Cada una de estas clases tiene unos atributos ya creados y varios métodos: unos pocos métodos ya están implementados, pero la mayoría están vacíos.

Usted debe implementar todos los métodos de estas clases, utilizando los atributos de la clase. Las clases son totalmente independientes entre ellas.

Cada una de las clases se concentra en una estructura de datos diferente: aproveche el taller para generar destreza en el uso de cada una y descubrir en qué caso cada una de estas funciona mejor.

Existen cuatro clases de prueba que verifican la correcta implementación de los métodos: ayúdese de esas pruebas para facilitar el desarrollo del taller.

Temas relevantes

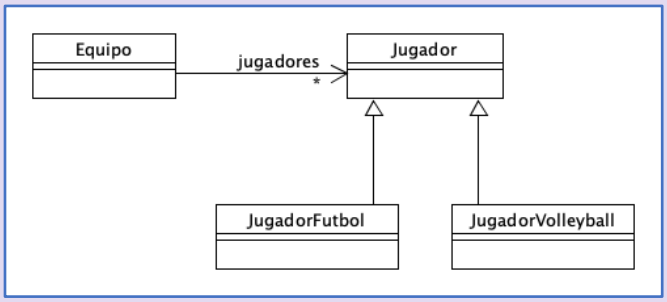
A continuación, encontrará un resumen de algunos temas relacionados con el taller: revíselos con cuidado y úselos como punto de partida para hacer búsquedas sobre temas complementarios.

Los temas tratados son:

- Generics en Java
- Listas y conjuntos en Java Collections
- Mapas en Java Collections
- El patrón Iterador

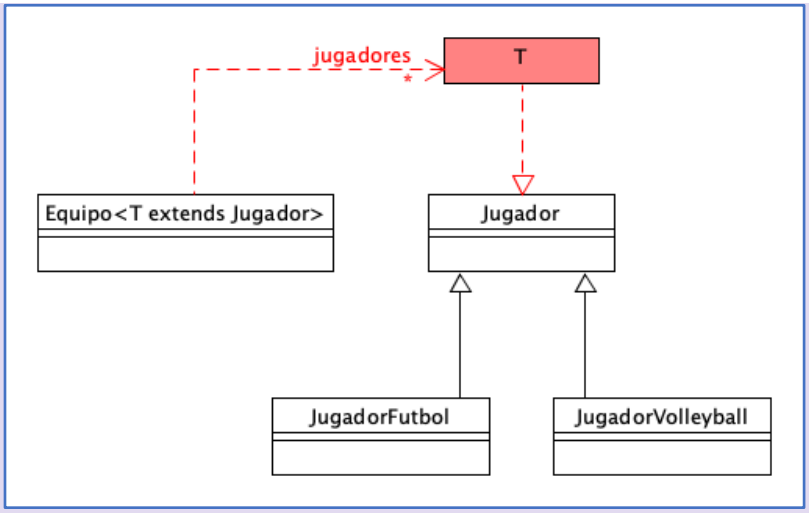
Teoría: Generics

En Java, cada objeto tiene que construirse a partir de una clase (cada objeto es una instancia de una clase) y puede tener relaciones con objetos que sean instancias de otra clase. Consideremos el siguiente diagrama:



Este diagrama nos indica que podemos tener equipos y que cada equipo puede tener jugadores en él. El diagrama también nos muestra que los jugadores pueden ser jugadores de fútbol o de volleyball. El problema es que esta estructura tiene un problema gravísimo porque hace posible que tengamos equipos que combinen jugadores de fútbol y jugadores de volleyball. Una solución posible sería extender la clase `Equipo` para que ahora tengamos `EquipoFutbol` y `EquipoVolleyball`. Sin embargo, esto haría necesario tener una clase de equipo nueva por cada nuevo tipo de jugador.

La otra alternativa es utilizar el mecanismo de **generics** que ofrece Java, el cual permite parametrizar los tipos de las relaciones. En este caso, **generics** nos permitiría hacer que cada objeto de tipo `Equipo` especifique qué tipo de `Jugador` va a contener, para que de ahí en adelante el compilador verifique que sólo agreguemos y extraigamos jugadores del tipo correcto. Para ver esto gráficamente, debemos abusar de la sintaxis de UML que no está preparada para soportar **generics**, aunque muchos lenguajes orientados a objetos ofrezcan este mecanismo.



En el diagrama vemos que ahora los equipos no van a tener instancias de `Jugador`, sino van a tener objetos de tipo `T`. `T` no es una clase real, es una ***variable de tipo***. En este caso decimos que `T` puede ser cualquier clase siempre y cuando herede de `Jugador`. En el código, esto está especificado en la declaración de la clase `Equipo`, donde se define que `T` debe ser alguna especialización de `Jugador`. Más abajo, en todos los lugares donde se haga referencia al tipo de los jugadores en el equipo, se debe utilizar `T` como tipo:

```
public class Equipo<T extends Jugador>
{
    private T[] jugadores;
    …
    public void agregarJugador(T nuevoJugador)
    …
    public T buscarJugador(String nombre)
    …
}
```

Como la clase `Equipo` ahora usa **generics**, al crear una nueva instancia estamos obligados a especificar el valor real de `T`, es decir el tipo de jugador que va a contener el equipo. La ventaja de esto es que en las instrucciones que agregan o extraen elementos del equipo, el compilador de Java puede verificar que no estemos cometiendo ningún error. Por ejemplo, las siguientes instrucciones deberían funcionar correctamente porque estamos respetando los tipos:

```
Equipo<JugadorFutbol> equipoFutbol = new Equipo<JugadorFutbol>();
equipoFutbol.agregarJugador(new JugadorFutbol("Pele"));
equipoFutbol.agregarJugador(new JugadorFutbol("Maradona"));
equipoFutbol.agregarJugador(new JugadorFutbol("Rossi"));
JugadorFutbol jugador = equipoFutbol.buscarJugador("Pele");
```

Por el contrario, las siguientes dos instrucciones no compilarían porque no superarían la verificación de tipos:

```
equipoFutbol.agregarJugador(new JugadorVolleyball("Karch"));
JugadorVolleyball jugador = equipoFutbol.buscarJugador("Karch");
```

Teoría: Framework de colecciones (conjuntos y listas)

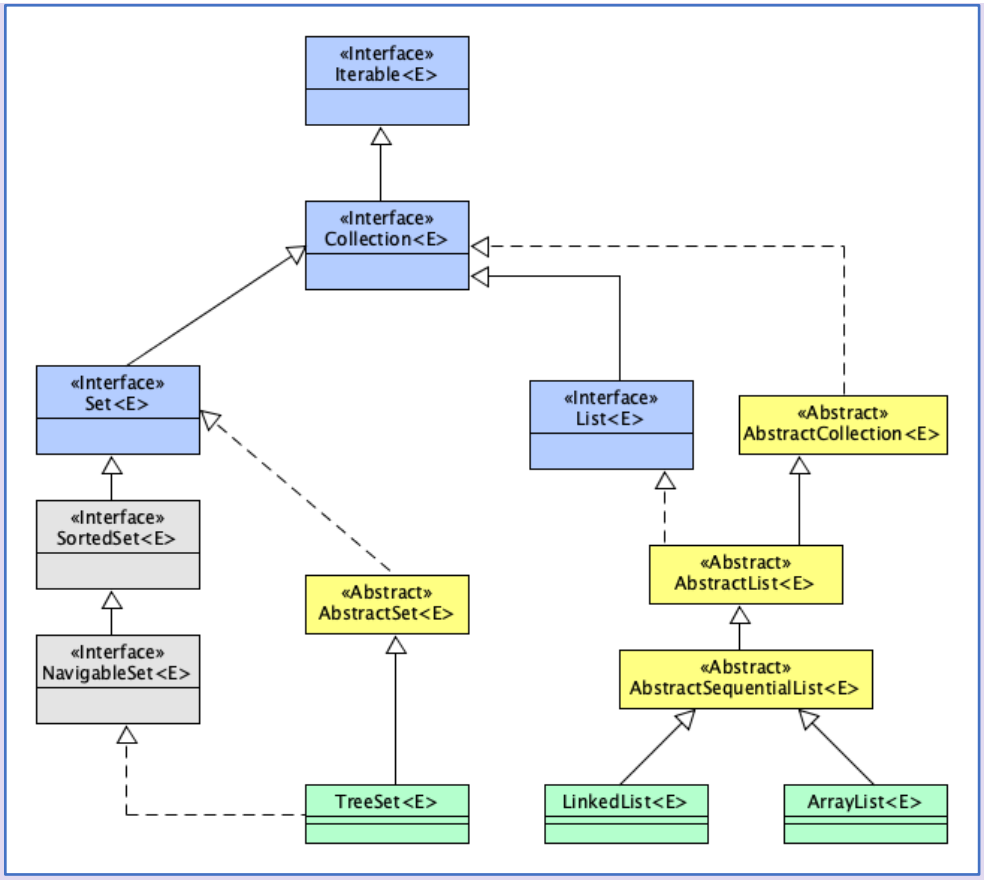
La librería estándar de Java ofrece el "Framework de colecciones", que es un conjunto de clases, clases abstractas e interfaces para el manejo de estructuras de datos. A continuación, presentamos un resumen de los elementos de este framework que necesita conocer para este taller. Estos elementos muy posiblemente le serán de mucha utilidad siempre que esté trabajando con Java.

En primer lugar, recordemos algunos conceptos básicos de programación orientada a objetos y de herencia:

- Un ***contrato*** es la promesa visible de lo que debe ser capaz de hacer un objeto. Un contrato se define a través de los métodos públicos de una clase o de una interfaz.
- Las ***interfaces*** definen un contrato (comportamiento) pero no tienen una implementación. Es decir, anuncian que quien ***implemente*** la interfaz debe ser capaz de hacer algo, pero no especifican cómo debe hacerlo (los métodos no tienen un cuerpo).
- Las ***clases abstractas*** definen también un contrato, pero pueden tener una implementación para uno o varios de los métodos definidos en el contrato. Las clases abstractas no se pueden instanciar, así que tienen que ser extendidas para que ese comportamiento pueda usarse.
- Si una clase **implementa** una interfaz, significa que la clase ofrece una implementación para todos los métodos definidos en la interfaz.
- Si una clase **extiende** una clase abstracta, tiene acceso a todos los métodos implementados en la clase abstracta y tiene la obligación de implementar los métodos que no tuvieran implementación en la clase abstracta. Si no los implementa todos, entonces también tiene que ser una clase abstracta.
- **Polimorfismo** hace referencia a la capacidad que tenemos de referirnos al tipo de un objeto utilizando su clase o cualquiera de sus superclases o de las interfaces que implementa. Por ejemplo, si tenemos la clase Rectángulo que extiende a la clase Polígono, si tenemos un objeto que sabemos que es un Rectángulo, podemos referirnos a él bien sea como Rectángulo o como Polígono.

En la imagen que se encuentra más abajo ilustramos los principales elementos que hacen parte del framework de colecciones. En realidad, sólo debería preocuparse por las interfaces de color azul y por las clases (concretas) de color verde. Las clases abstractas sabemos que existen porque están documentadas y porque tenemos acceso al código fuente, pero en la enorme mayoría de los casos no necesitamos ni siquiera conocer de su existencia.

- **Iterable**: esta interfaz simplemente indica que las colecciones van a ser *iterables*, es decir que se pueden recorrer con un iterador (ver el bloque de teoría sobre iteradores más adelante).
- **Collection**: esta interfaz indica que tenemos algo que contiene a otros objetos y que podemos realizar algunas operaciones básicas como agregar y quitar elementos, contar los elementos, o saber si un elemento está o no dentro de la colección.
- **Set**: esta interfaz indica que una colección es un conjunto. Esto quiere decir que no tendrá elementos repetidos y que además el concepto de orden no tiene sentido. Si intentamos agregar un elemento repetido a un conjunto, el conjunto no cambiará. Si recorremos los elementos del conjunto, no podemos predecir el orden de los elementos (no necesariamente estará relacionado con el orden en que se agregaron).
- **List**: esta interfaz indica que una colección es una estructura de datos lineal de una dimensión. Esto quiere decir que los elementos dentro de estas colecciones tienen una posición bien definida, identificable por un número entre 0 y la cantidad de elementos menos 1. En una lista podemos entonces consultar posiciones específicas, como la primera o la última, podemos agregar elementos al final o podemos insertar elementos en cualquier posición intermedia.



Hay 3 clases concretas que nos van a interesar para este taller:

- **TreeSet**: es la clase concreta más sencilla que implementa la interfaz `Set`. El nombre nos indica que la implementación está basada en un árbol.
- **LinkedList**: esta clase implementa la interfaz `List` utilizando una estructura de tipo lista encadenada.
- **ArrayList**: esta clase implementa la interfaz `List` utilizando un arreglo como base.

Finalmente, vale la pena mencionar que el framework de colecciones utiliza ampliamente el mecanismo de *generics*. Por esto todas las clases e interfaces de este framework están marcadas con `<E>` al final de su nombre, indicando que al momento de crear instancias de las clases concretas debería indicarse el tipo de datos que se va a almacenar en ellas. Esto puede verse en el siguiente ejemplo, en el que también ilustramos el concepto de ***polimorfismo***:

```
ArrayList<String> listaPalabras = new ArrayList<String>();
Collection<String> texto = new LinkedList<String>();
Set<String> palabras = new TreeSet<String>();
```

En el primer caso estamos creando un `ArrayList` que va a contener `String` y lo almacenamos en una variable de tipo `ArrayList<String>`. Es decir que el compilador de Java va a saber que `listaPalabras` es un `ArrayList` y nos va a permitir usar todos los métodos de esta clase.

En el segundo caso, estamos creando un `LinkedList` que va a contener `String` y lo estamos almacenando en una variable de tipo `Collection<String>`. Es decir que el compilador de Java sólo nos va a permitir usar los métodos de `Collection` y no los métodos de `LinkedList`. Por ejemplo, vamos a poder usar el método `add(…)` pero no vamos a poder usar el método `addLast(…)` que está definido en `LinkedList`.

Finalmente, en el tercer caso estamos creando un `TreeSet` que va a contener `String` y lo estamos almacenando en una variable de tipo `Set<String>`. Es decir que el compilador de Java nos va a permitir usar los métodos de `Set` y no los de `TreeSet`.

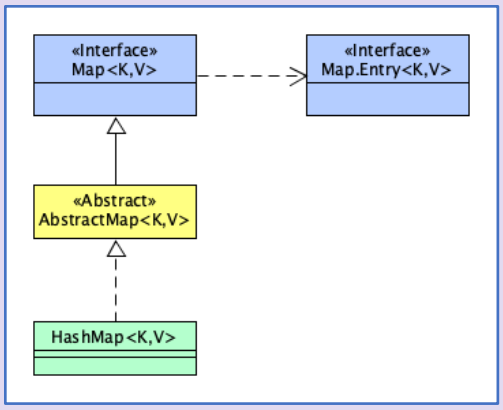
¿Qué ganamos con el polimorfismo? En este caso, desacoplamiento. Por ejemplo, en el tercer caso sabemos que en `palabras` hay un conjunto de cadenas, pero sin tener que saber si ese conjunto está implementado sobre una lista, una tabla de hashing, un arreglo, un árbol o cualquier otro mecanismo que alguien pudiera imaginarse. Esto significa que en `palabras` no puede haber cadenas repetidas, mientras que en `texto` (que es de tipo `Collection`) sí podríamos esperarnos repeticiones. Este desacoplamiento permitirá cambiar más adelante el tipo concreto utilizado por la estructura de datos, sin tener un impacto significativo en el código.

¿Qué ganamos con los *generics*? En primer lugar, mayor control sobre los elementos que metemos dentro de cada estructura. Gracias a este mecanismo no corremos el riesgo de combinar elementos de diferentes tipos dentro de la misma estructura y luego confundir sus tipos cuando los vayamos a utilizar.

En segundo lugar, usar generics nos permite simplificar un poco nuestro código. Por una parte, no tenemos que introducir instrucciones adicionales para protegernos de errores con los tipos y verificar qué es cada cosa cada vez que lo vayamos a usar. Por otra parte, no tenemos que estar haciendo *casting* permanentemente para convertir de tipos genéricos a tipos más específicos.

Teoría: Framework de colecciones (mapas - diccionarios)

La otra parte del framework de colecciones es la que tiene que ver con mapas, que son los elementos más parecidos a los diccionarios que existen en Python [1]. Los mapas son todas estructuras de datos que contienen parejas de llaves y valores: para cada llave en el mapa, hay un valor correspondiente. Además, los mapas están construidos usando también *generics*, así que se debe indicar el tipo que deben tener las llaves y el tipo que deben tener los valores almacenados en el mapa [2].



Como vemos, los mapas no están relacionados de ninguna forma con la interfaz `Collection`, aunque se considera que hacen parte del llamado framework de colecciones. En este diagrama hemos incluido sólo el subconjunto de clases e interfaces relevante para este taller, de las cuales nos interesan:

---

[1] En realidad, en el JDK existe una clase llamada `Dictionary` pero está deprecada desde hace más de 15 años y no debería usarse.

[2] A diferencia de Python, donde las llaves y los valores pueden ser de cualquier tipo, en Java tanto las llaves como los valores deben ser objetos, todas las llaves de un mapa deben ser del mismo tipo y todos los valores deben ser del mismo tipo. Si esto es una limitación, la solución es utilizar la superclase más específica que se pueda para las llaves y valores (que en el peor caso es `Object`), aunque en la mayoría de los casos no es una buena idea.

- **Map<K,V>**: esta interfaz define el comportamiento básico que tiene que implementar un mapa, es decir almacenar, buscar y eliminar parejas llave – valor. La interfaz tiene dos parámetros: `K`, que se refiere al tipo de las llaves, y `V`, que se refiere al tipo de los valores que se almacenarán en el mapa.
- **Map.Entry<K,V>**: esta interfaz está definida dentro de la interfaz `Map` (de ahí que su nombre tenga el prefijo `Map.`) y sirve para representar precisamente una pareja de una llave y un valor.
- **HashMap<K,V>**: se trata de la implementación concreta de `Map` más utilizada.

Veamos ahora un pequeño programa que usa mapas.

```
Map<String, Integer> precios = new HashMap<String, Integer>();
precios.put("Sal", 1200);
precios.put("Vinagre", 2205);
precios.put("Lentejas", 4050);
precios.put("Sal", 1300);

int precioSal = precios.get("Sal");
System.out.println("La sal vale " + precioSal);  // La sal vale 1300

for (Map.Entry<String, Integer> pareja : precios.entrySet())
{
   String producto = pareja.getKey();
   int precio = pareja.getValue();
   System.out.println(producto + " vale " + precio);
}
```

En este programa en primer lugar se construye un `HashMap` para almacenar parejas conformadas por una cadena de caracteres y un número entero. Como `int` es un tipo simple, tenemos que usar la clase equivalente (`Integer`). Después de guardar algunos valores dentro del diccionario, incluyendo uno repetido, extraemos un valor y lo imprimimos.

Finalmente, usamos una estructura de tipo `for-each` para recorrer el mapa. Sin embargo, los mapas no se pueden recorrer directamente así que es necesario recorrer el conjunto con las parejas.

Teoría: El patrón *Iterator*

Un patrón es una solución conocida a un problema recurrente. En este caso, el problema recurrente es tener que recorrer una estructura de datos, desde el principio hasta el final, y tener que cambiar la forma del recorrido dependiendo de la implementación de la estructura de datos. Esto hace que en muchos casos cambiar la implementación de una estructura requiera un esfuerzo enorme.

El ***patrón Iterador*** nos ayuda a solucionar este problema. Nos permite recorrer estructuras de datos diferentes usando exactamente el mismo código, y esto hace posible cambiar con mucha facilidad las implementaciones de nuestras estructuras de datos. Puede imaginarse que un Iterador es un guía que le va a servir para recorrer un museo y al cual usted puede pedirle únicamente que lo lleve al siguiente punto de interés: a usted no le importa cómo hace el guía para saber cuál es el siguiente sitio; a usted sólo le interesa que lo lleven hasta allá. Si usted le pide repetidamente al guía que lo lleve al siguiente punto, después de un tiempo habrá recorrido todo el museo.

El patrón Iterador está soportado en el framework de colecciones, así que usar el patrón será absolutamente natural con estas estructuras. Puede haber pequeñas diferencias en la implementación del patrón, así que acá vamos a limitarnos a explicar cómo está implementado el patrón dentro de este framework.

Participantes:

- **Estructura de datos iterable**: esta es la estructura de datos que se va a recorrer y tiene que implementar la interfaz *Iterable*. Esta interfaz define sólo un método que nos interesa, llamado `iterator()`, el cual retorna un objeto que implementa la interfaz `Iterator`.
- **Iterador**: este es el elemento que nos va a permitir recorrer la estructura de datos. No nos interesa saber cuál es la clase concreta de este elemento y sólo sabemos que implementa la interfaz `Iterator`. En el framework de colecciones, la interfaz Iterator sólo define dos métodos que nos interesan: `hasNext()`, que nos indica si todavía falta algún elemento por visitar en la estructura de datos; y `next()`, que nos retorna el siguiente elemento en la estructura de datos.

Veamos ahora un par de ejemplos de uso de los iteradores. Primero veremos cómo se usa un iterador usando un `while`:

```
Collection<Integer> numeros = new ArrayList<Integer>(listaNumeros);

Iterator<Integer> iterador = numeros.iterator();
while (iterador.hasNext())
{
    Integer siguienteNumero = iterador.next();
    System.out.println("Número: " + siguienteNumero);
}
```

Ahora veamos un programa equivalente, pero esta vez utilizando la estructura `for`:

```
Collection<Integer> numeros = new ArrayList<Integer>(listaNumeros);
for (Iterator<Integer> iterator = numeros.iterator(); iterator.hasNext();)
{
    Integer siguienteNumero = iterator.next();
    System.out.println("Número: " + siguienteNumero);
}
```

Estos dos ejemplos nos muestran los dos aspectos importantes del patrón Iterador:

- No importa cuál sea la estructura de datos, el recorrido se hace de la misma manera. Fíjese que dentro del while y del for no sabemos cómo está implementado `numeros`. Sólo sabemos que es una colección, pero no sabemos si es una lista, un conjunto, una pila, una cola de prioridades, etc.
- Como no nos interesa la implementación concreta de la colección, podemos cambiarla sin que implique cambios al código que estamos usando para hacer los recorridos. Observe ahora el siguiente ejemplo de recorrido que no usa un iterador:

```
ArrayList<Integer> numeros = new ArrayList<Integer>(listaNumeros);

for(int i = 0; i<numeros.size(); i++)
{
    Integer siguienteNumero = numeros.get(i);
    System.out.println("Número: " + siguienteNumero);
}
```

Aunque este programa no se ve tan diferente, tiene una diferencia crucial: el llamado al método `get(int)` implica necesariamente que `numeros` sea una lista (no puede ser ni un conjunto, ni una pila, ni ninguna otra estructura que no implemente la interfaz List). El recorrido que estamos haciendo acá está acoplado a la estructura de datos y no nos servirá para ninguna estructura que no sea una lista.

Entrega

1. Cree un repositorio privado donde quedarán sus talleres y deje ahí todos los archivos relacionados con sus entregas.

2. Entregue a través de Bloque Neón el URL para el repositorio, en la actividad designada como "**Taller 2**".