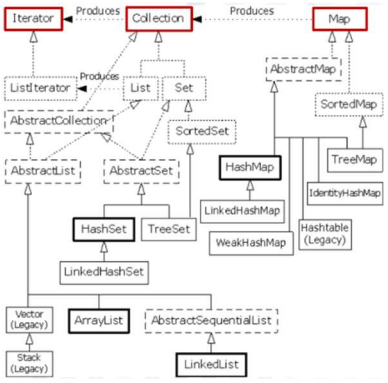
TEMA 3: CONJUNTOS DE DATOS

- Una buena parte de la programación consiste en **manejar conjuntos de datos** sobre los que se realizan operaciones CRUD (creación, lectura, actualización y eliminación).
- La mayoría de los lenguajes de programación soportan la representación de conjuntos de datos a través del concepto de **array**. Sin embargo, los arrays tienen muchas **limitaciones**.
- Por esto, se han propuesto muchas formas de representar conjuntos de datos que resuelven estas limitaciones. Java soporta arrays, colecciones, iteradores, listas, conjuntos, mapas, etc.
  - ↳ Todos los tipos de conjuntos de datos son **mapas, colecciones o iteradores** → el resto particulariza el tipo del cual hereda.



ARRAYS

- El concepto de ARRAY en Java es el mismo que en los lenguajes procedimentales → conjunto de elementos del **mismo tipo** que ocupan posiciones de **memoria consecutivas** para facilitar su acceso de forma sencilla a través de un **índice**.
- En Java, los arrays son **objetos**:
  - ↳ Su clase es $tipodedatos[]$.
  - ↳ Se debe reservar memoria para ellos con new.
  - ↳ Hereda todos los métodos de la clase $Object$.
  - ↳ Tiene como atributo público su longitud ($length$).
- Es la única estructura que permite almacenar **tdd primitivos**.
- Tienen importantes **limitaciones**:
  - Tienen un **tamaño máximo fijo** → no se pueden añadir más elementos una vez se alcanza.
  - **No** se pueden **borrar** elementos cuando el array maneja un **tdd primitivo**.
    - ↳ Cuando maneja objetos, se pueden eliminar elementos escribiendo $null$ en su posición
      Como consecuencia, en todos los posteriores recorridos se deberá tener cuidado con estos valores nulos para evitar un error de ejecución.

```
int[] numEjercitos;
numEjercitos= new int[6];
```

En la reserva de memoria se especifica el tamaño del array, que no puede cambiar

```
String[] colores= { "Amarillo", "Azul",  "Cyan",
                    "Rojo",     "Verde", "Violeta" };
```

El borrado de un dato en un array de objetos se puede interpretar como escribir un null en la posición que se quiere borrar

```
String[] colores= { "Amarillo", null,    "Cyan",
                    "Rojo",     "Verde", "Violeta" };
```

```
int[] numEjercitos= { 14, 15, 18, 20, 24, 28 };
```

En datos primitivos no se puede usar null, entonces, ¿qué valor por defecto se puede usar?

No se puede borrar el dato i= 1

- Sólo se deben usar arrays cuando se sabe que el **número de elementos es fijo** y **no** se va a **eliminar** ninguno.
- Sólo se deben usar cuando se invocan **métodos** de otras clases que **devuelvan un array** y convertirlos a otras estructuras sea ineficiente.

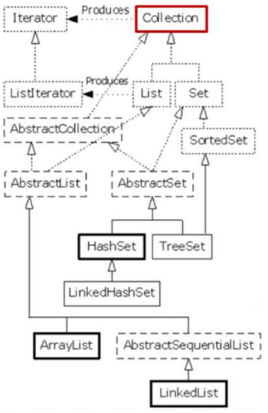
- Una INTERFACE es una especie de "clase abstracta" formada por un conjunto de **métodos abstractos** (sin implementar) y **atributos públicos**.
  - ↳ Las interfaces **no se pueden instanciar**, pues sus métodos no están implementados.

COLECCIONES

- COLECCIONES → grupos de datos **desordenados** sobre los que se definen operaciones de inserción, borrado y actualización.

COLLECTION

- $Collection < E >$ → interface que define las operaciones que debe tener una colección que almacena datos de tipo E.
  - ↳ **No** se puede **reservar memoria** (instanciar) para un objeto de tipo Collection, pues es una interfaz.
- Los elementos de las colecciones están **desordenados**, así que sus elementos **no** se pueden identificar mediante un **índice**.
- **Métodos** de Collection más frecuentemente usados.
  - $boolean\ \boldsymbol{add}(E\ ele)$ → añade el elemento ele a la colección.
  - $boolean\ \boldsymbol{contains}(Object\ obj)$ → comprueba si el objeto obj se encuentra en la colección.
  - $boolean\ \boldsymbol{isEmpty}()$ → indica si la colección está vacía.
  - $void\ \boldsymbol{remove}(Object\ obj)$ → elimina el objeto obj de la colección.
  - $Iterator < E > \boldsymbol{iterator}()$ → genera un iterador que se puede usar para recorrer la colección.
  - ↳ $contains$ y $remove$ usan $equals$ para comprobar si el objeto está en la colección.



- Algunas implementaciones de la interface Collection **no soportan todos sus métodos**.
  Como de todas formas es necesario que los implementen, generan una excepción del tipo $UnsupportedOperationException$.
  - ▷ Por ejemplo, la implementación del método $values()$ de la clase HashMap devuelve una Collection para la cual el método $add()$ no está soportado.
    - ↳ No se debe poder realizar un $add()$ sobre este objeto porque entonces se introduciría un elemento sin clave en el HashMap, rompiendo su estructura.
- Una de las formar de recorrer todos los elementos de una Collection es a través de un **bucle *for-each***. No se puede hacer un *for* normal porque la colección no tiene índices que recorrer.
  - **Problema**: las colecciones **no se pueden modificar** (es decir, no se puede añadirle o eliminarle elementos) mientras se están recorriendo porque esto puede hacer que el iterador que la está recorriendo pierda la pista del estado de la colección. Si se intenta, se genera una excepción del tipo $ConcurrentModificationException$.

```
public boolean add(E e) {
    throw new UnsupportedOperationException();
}
```

Es necesario implementar add porque la clase debe implementar todos los métodos de Collection

La implementación de este método para la colección devuelta por la clase HashMap tiene sentido, ya que así se evita que se modifique la colección, haciéndola inconsistente con los valores almacenados en el objeto HashMap

colContinentes es una colección de continentes generada por el método values() de la clase HashMap

```
Collection<Continente> colContinentes= this.continentes.values();
for(Continente continente : colContinentes)
    System.out.println("Nombre: " + continente);
```

```
HashMap<String,Pais> mapPaises= mapa.getPaises();
Collection<Pais> colPaises= mapPaises.values();
Pais nuevoPais= new Pais("Atlántida", "Atlántida", null, null);
colPaises.add(nuevoPais);
```

Genera una excepción al intentar añadir un nuevo país a la colección

```
java.lang.UnsupportedOperationException
        at java.util.AbstractCollection.add(AbstractCollection.java:262)
        at risketse.Menu.crearMapa(Menu.java:303)
        at risketse.Menu.<init>(Menu.java:54)
        at risketse.RiskETSE.main(RiskETSE.java:14)
```

Usa la implementación del método add que se encuentra en la clase AbstractCollection

```
public Collection borrarContinente(Continente aBorrar) {
    Collection<Continente> colConts= continentes.values();
    for(Continente continente : colConts) {
        if(continente.equals(aBorrar))
            colConts.remove(aBorrar);
    }
    return colConts;
}
```

Elimina el objeto aBorrar de la colección

```
java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextNode(HashMap.java:1445)
```

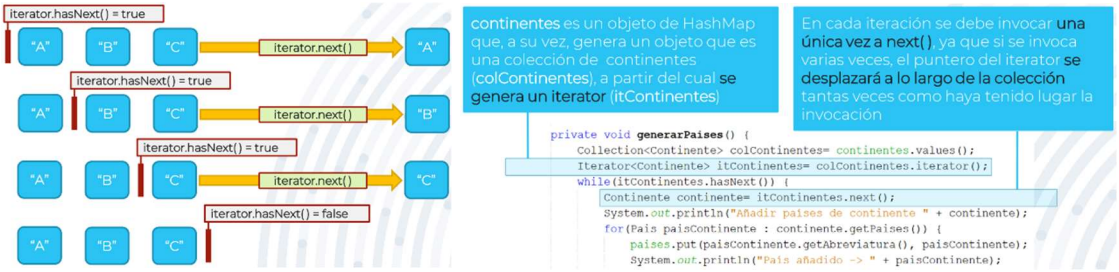
Si la clase Continente no reimplementa equals, remove compara la referencia del argumento aBorrar con las de los elementos de la colección

Si se comparan las referencias, en la mayoría de las ocasiones remove no encontrará el objeto y no lo eliminará de la colección

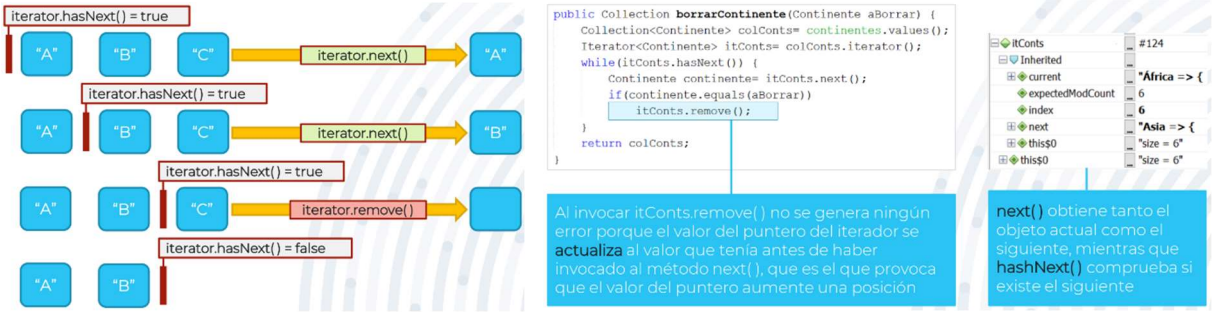
ITERATOR

- $Iterator < E > \rightarrow$ interface que define las operaciones que se pueden usar para recorrer los elementos de una colección de datos de tipo E.
  - ↳ **No** se puede **reservar memoria** (instanciar) para un objeto de tipo Iterator, pues es una interfaz.

- **Métodos** de Iterator más frecuentemente usados.
  - $boolean\ \boldsymbol{hasNext}() \rightarrow$ indica si existe un elemento en la siguiente posición en la que se encuentra el puntero.
  - $E\ \boldsymbol{next}() \rightarrow$ actualiza el puntero a la posición siguiente y obtiene el elemento que se encuentra ahí.
  - $\boldsymbol{remove}() \rightarrow$ elimina el elemento que se encuentra en la posición actual.
  - ↳ Como los punteros sólo se mueven hacia adelante, un iterador **sólo se puede usar una vez** para recorrer los elementos de la colección, ya que después el puntero del iterador estará en el final de la colección y $hasNext()$ devolverá siempre falso.

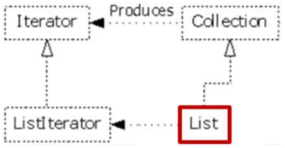


- Un iterador es una forma alternativa al bucle *for-each* para **recorrer los elementos de una colección**.
  - A diferencia del *for-each*, el iterador permite la eliminación de los elementos mientras se recorre la colección, **eliminando tanto los elementos de la colección como del iterador.**
  - ↳ El **rendimiento de un iterador y de un *for-each* es el mismo**, ya que en realidad el compilador interpreta el *for-each* como si fuese un iterador.



LISTAS

LISTAS



- $List < E > \rightarrow$ subinterface de Collection. Además de soportar las operaciones de Collection, soporta las que debe tener una lista.
  - ↳ **No** se puede **reservar memoria** (instanciar) para un objeto de tipo List, pues es una interfaz.

- Los elementos de las List tienen un **orden** que está relacionado con la secuencia en que dichos elementos han sido introducidos durante la ejecución. Este orden permite identificar cada elemento con un **índice** que indica su posición en la List.
- Las List pueden contener **elementos duplicados**.

- **Métodos** de List más frecuentemente usados (además de los de Collection):
  - $E\ \boldsymbol{get}(int\ i) \rightarrow$ obtiene el elemento de índice i.
  - $void\ \boldsymbol{set}(int\ i, E\ ele) \rightarrow$ actualiza el objeto de índice i con el valor ele.
  - $E\ \boldsymbol{remove}(int\ i) \rightarrow$ elimina el objeto de índice i.

- Las List se pueden **recorrer** tanto usando sus **índices**, ya que son ordenadas, como usando un **bucle *for-each***, ya que son Collections.

ARRAYLIST

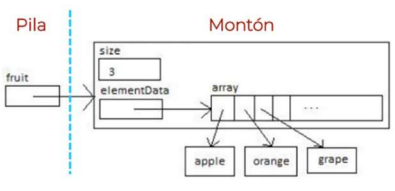
- $ArrayList < E > \rightarrow$ clase que implementa la interface List.
  - ↳ **Sí** se puede **reservar memoria** (instanciar) para un objeto de tipo ArrayList, pues es una clase.

- Tiene un **atributo público,** $\boldsymbol{size}$.
- Las **operaciones de acceso** a sus elementos de tienen complejidad lineal ($\boldsymbol{O(n)}$).
- Permite introducir **valores nulos**.

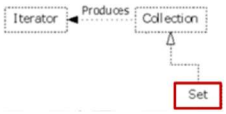
- El **constructor sin argumentos** de ArrayList reserva espacio para 10 elementos.
- ArrayList realiza una **redimensión dinámica** cuando el número de datos que se almacena es **mayor que la capacidad** de almacenamiento de la lista.
  - ↳ Por defecto, se aumentará la capacidad en 10.
    - Si la redimensión se realiza **muy frecuentemente** se penalizará el **rendimiento** de la lista, ya que internamente la redimensión supone incrementar el tamaño de un array normal a través de una operación de caché de datos. ?

- Los ArrayList se **recorren** usando un bucle ***for* normal**.
  - ↳ Al **eliminar** elementos durante el recorrido habrá que tener cuidado con que el **índice no supere el nuevo tamaño**.
  - ↳ Durante el recorrido hay que comprobar que el elemento actual es **distinto de** $\boldsymbol{null}$ antes de operar sobre él.

- En realidad, la clase ArrayList encapsula un **array normal** de objetos al que se accede a través de los métodos de la interface List.
  - ↳ La **pila** almacena → la **referencia al array de objetos**.
  - ↳ El **montón** almacena → el **array de objetos** y los **demás atributos** de ArrayList ($size$).



CONJUNTOS



- CONJUNTOS → colecciones en las que **no se repiten los datos**.
  - Esto significa que los datos serán diferentes de acuerdo al criterio de igualdad definido para ellos (método $equals$).

SET

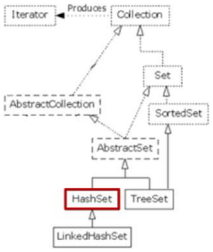
- $Set < E >$ → subinterface de Collection que soporta exactamente las mismas operaciones que una Collection.
  - **No** se puede **reservar memoria** (instanciar) para un objeto de tipo Set, pues es una interfaz.
- Los elementos del conjunto son **únicos**.
  - Para asegurarse de esto, **se debe sobreescribir el método $equals$ de la clase E**.
  - Se debe controlar que la **modificación** de los objetos no se dé como resultado que dicho objeto sea igual a otro de los almacenados.
    - Se generarán estas inconsistencias cuando las modificaciones **no se realizan con los métodos de Set**, si no que se realizan por aliasing.
  - Como consecuencia, permiten **sólo un valor nulo**.
- Como es una colección, se **recorre** con un **bucle *for-each*** o un **iterador**, no con un índice.

- El objeto conjuntoPaises contiene países ,donde el nombre del país país es el criterio de identidad, es decir, dos objetos del conjunto son iguales si sus nombres son los mismos
- El nombre del objeto cuyo nombre es "Siberia" se cambia por "Rusia", en cuyo caso hay dos países que son iguales y no se generan errores

```
32 System.out.println("Países antes de modificación");
33 for(País país : conjuntoPaises)
34     System.out.println(país.getNombre());
35
36 System.out.println("Países después de modificación");
37 for(País país : conjuntoPaises) {
38     if(país.getNombre().equals("Siberia"))
39         país.setNombre("Rusia");
40     System.out.println(país.getNombre());
41 }
```

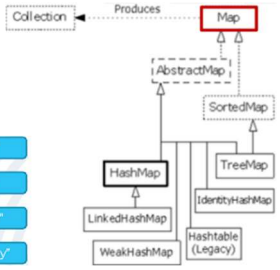
```
Países antes de modificación
Siberia
GBretaña
Rusia

Países después de modificación
Rusia
GBretaña
Rusia
```

HASHSET



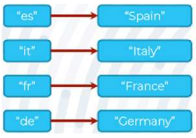
- $HashSet < E >$ → clase que implementa la interface Set.
  - **Sí** se puede **reservar memoria** (instanciar) para un objeto de tipo ArrayList, pues es una clase.
- Internamente, sus datos se almacenan como las **claves de un objeto HashMap.**
  - Sus valores están **desordenados**, pues se almacenan en una **tabla hash** de acuerdo con una **función hash**.
  - Las operaciones de **acceso** a sus elementos se realizan en tiempo constante ($O(1)$).

MAPAS



MAP

- $Map < K, V >$ → interface que relaciona una clave con un valor de manera que **a partir de la clave se obtiene el valor asociado a ella**.
  - **No** se puede **reservar memoria** (instanciar) para un objeto de tipo Map, pues es una interfaz.



- Las **claves no pueden estar repetidas**, pero los **objetos sí** (por tanto varias claves pueden tener asociado un mismo valor).
- Tanto las claves como los valores tienen que ser **objetos**, no pueden ser tdd primitivos.
- Algunas implementaciones permiten almacenar **claves y valores nulos**.
- La **clave** puede ser **cualquier tipo de objeto**, siempre que su clase siga las siguientes **reglas**:
  - El $equals()$ **debe estar sobrescrito**, pues se usará para localizar y obtener la clave con la que recuperar un valor.
  - El objeto clave debe ser **inmutable** (es decir, no cambiará nunca una vez se realiza la reserva de memoria) pues si se cambiara, perderíamos el acceso al valor asociado a la clave, pues el identificador de la entrada (hashCode) se calculó en base al valor de la clave.
    - Si la clave no es inmutable, entonces es necesario controlar que **no se modifiquen aquellos atributos usados en $equals()$** para comprobar la igualdad entre objetos.
- **Métodos** de Map más frecuentemente usados.
  - V **get**$(Object\ key)$ → devuelve el valor asociado a la clave key.
  - V **put**$(K\ key, V\ value)$ → almacena un valor $value$ al que se le asocia una clave key.
  - V **$remove$**$(K\ key)$ → elimina la entrada cuya clave es key.
  - $boolean$ **$containsValue$**$(V\ value)$ → indica si el valor $value$ está almacenado en el mapa, es decir, si tiene asociada una clave.
  - $boolean$ **$containsKey$**$(K\ key)$ → indica si existe algún dato asociado a la clave key.
  - **$containsKey()$ es más eficiente** que $containsValue()$ pues aprovecha la estructura de hashing de la implementación del Map para realizar la búsqueda de claves.
- Algunas implementaciones de la interface Map **no soportan todos sus métodos**.
  Como de todas formas es necesario que los implementen, generan una excepción del tipo $UnsupportedOperationException$.
- Los objetos almacenados en un Map **no se pueden recorrer directamente**, ya que su objetivo es acceder a los valores a partir de las claves, no realizar una búsqueda de datos usando un criterio diferente de las claves.
  - Map define los siguientes métodos para **convertir las claves y valores** a Collections o Sets **para poder recorrerlos**.
    - $Collection < V >$ **$Values$**$()$ → devuelve una Collection con los valores del Map.
      - Devuelve una colección pues cuando se guardan valores no hay manera de saber si lo han hecho de manera consecutiva y/o en el orden en el que se introdujeron.
    - $Set < K >$ **$keySet$**$()$ → devuelve un Set (conjunto) con las claves del Map.
      - Devuelve un conjunto pues las claves no pueden contener valores repetidos.
    - $Set < Map.Entry < K, V >>$ **$entrySet$**$()$ → devuelve un Set (conjunto) con todas las entradas del Map.
      - Devuelve un conjunto pues, ya que no se pueden repetir claves, no puede haber dos entradas iguales.

MAP ENTRY

- $Map.Entry < K, V >$ → interface que define métodos para acceder a una entrada de un mapa.
  - $K\ getKey()$ → permite acceder a la clave de la entrada en el mapa.
  - $V\ getValue()$ → permite acceder al valor de la entrada en el mapa.

continentes es un mapa con clave de tipo String y valores de tipo Continente: Map<String, Continente>

```
continentes.put("África", new Continente("África", Valor.PAIS_VERDE));
continentes.put("Asia", new Continente("Asia", Valor.PAIS_CYAN));
continentes.put("Australia", new Continente("Australia", Valor.PAIS_ROJO));
continentes.put("Europa", new Continente("Europa", Valor.PAIS_AZUL));
```

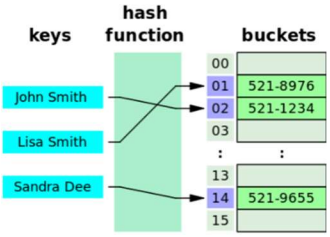
Se usa entrySet() para acceder al conjunto de entradas del HashMap, de modo que se puede obtener o la clave o el dato

```
private void generarPaises() {
    Set<Map.Entry<String,Continente>> meContinentes= continentes.entrySet();
    for(Map.Entry entrada : meContinentes) {
        Continente continente= (Continente) entrada.getValue();
        System.out.println("Añadir países de continente " + continente);
        for(País paisContinente : continente.getPaises())
            países.put(paisContinente.getAbreviatura(), paisContinente);
    }
}
```

Se usa keySet() para generar un conjunto de claves que se recorren a través de un for-each para acceder a todos los continentes

```
private void generarPaises() {
    Set<String> setContinentes= continentes.keySet();
    for(String claveCon : setContinentes) {
        Continente continente= continentes.get(claveCon);
        System.out.println("Añadir países de continente " + continente);
        for(País paisContinente : continente.getPaises())
            países.put(paisContinente.getAbreviatura(), paisContinente);
    }
}
```

HASHMAP



- $HashMap < K, V > \rightarrow$ clase que implementa la interface Map.
  - **Sí** se puede **reservar memoria** (instanciar) para un objeto de tipo HashMap, pues es una clase.
- Los valores de una del HashMap están **desordenados**, pues se almacenan en una **tabla hash** de acuerdo con una **función hash**.
  - Sin embargo, las operaciones de **acceso** a las entradas del HashMap se realizan en tiempo constante ($\boldsymbol{O(1)}$).
- Cada clave del mapa está asociada a un CÓDIGO HASH (hashCode), que es un entero generado aleatoriamente a partir de una semilla.
- Se supone que los hashCode son **únicos** para cada objeto, pero como son generados aleatoriamente es **imposible asegurar** esto.
- Por tanto, **cuando se comparan dos entradas** de un HashMap:
  - Se comparan sus hashCode → si son distintos, se entiende que los objetos son distintos, si son iguales, se pasa al paso 2.
  - Se invoca a $equals()$ para aplicar el criterio de igualdad
    - El uso de los hashCode simplifica y hace más **eficiente el acceso** a los datos en HashMap pues en primer lugar compara enteros y sólo ejecutará $equals()$ (que puede tener una implementación con comparaciones muy complejas) cuando coincidan.
- Para **generar** el hashCode, es necesario **sobreescribir** el método $hashCode()$ heredado de $Object$ usando el valor de los **atributos inmutables de la clave** como semillas.
  - La implementación de $hashCode()$ en $Object$ es un método nativo escrito en C.
  - Se generan hashCodes parciales usando estas semillas en el método $hashCode()$ de la clase $Objects$ (el de $Object$ no funciona con valores nulos).

```java
@Override
public int hashCode() {
    int hash = 3;
    hash = 31 * hash + Objects.hashCode(this.nombre);
    hash = 31 * hash + Objects.hashCode(this.abreviatura);
    return hash;
}
```

```java
public static int hashCode(Object o) {
    return o != null ? o.hashCode() : 0;
}
```

COMPARATIVA

- Según las **características** de los conjuntos de datos escogeremos uno u otro en función de para qué lo necesitamos:

Collection	Ordering	Random Access	Key-Value	Duplicate Elements	Null Element
ArrayList	✅	✅	❌	✅	✅
LinkedList	✅	❌	❌	✅	✅
HashSet	❌	❌	❌	❌	✅
TreeSet	✅	❌	❌	❌	❌
HashMap	❌	✅	✅	❌	✅
TreeMap	✅	✅	✅	❌	❌

- Las **listas** ocupan **menos memoria** que los mapas.
- Los **mapas** son **más rápidos** que las listas.

Collection	Performance	Default capacity	Empty size	10K entry overhead	Accurately sized?	Expansion algorithm
HashSet	O(1)	16	144	360K	No	x2
HashMap	O(1)	16	128	360K	No	x2
Hashtable	O(1)	11	104	360K	No	x2+1
LinkedList	O(n)	1	48	240K	Yes	+1
ArrayList	O(n)	10	88	40K	No	x1.5

- Por lo general:
  - Si hay **restricciones de memoria** respecto a la cantidad de daos a almacenar y **la eficiencia no es la prioridad** → ArrayList.
  - Si **se necesita acceder en el orden de llegada** → ArrayList.
  - Si **no se necesita acceder** a los datos **en el orden en el que se almacenaron** y **se dispone de un identificador unívoco** → HashMap.
  - Si **no se necesita acceder** a los datos **en el orden en el que se almacenaron** y los datos **no se pueden repetir** → HashSet.