3.
Conjuntos de datos

Arrays, colecciones, listas, conjuntos y mapas

© **Manuel Lama** (manuel.lama@usc.es)

Conjuntos de datos

- Una buena parte de la programación consiste en manejar conjuntos de datos sobre los que se realizan operaciones CRUD de búsqueda, inserción, modificación y borrado
- Típicamente la mayoría de los lenguajes de programación soportan la representación de conjuntos de datos a través del concepto de array
- Los arrays tienen muchas limitaciones a la hora de acceder a los datos, tanto en la lectura como en la escritura
  - Se han propuesto una buena cantidad de formas de representar conjunto de datos que intentan dar respuesta a estas limitaciones
  - Java soporta directamente varios tipos de conjuntos de datos

Conjuntos de datos

- Arrays — Elementos accesibles a través de un índice
- Colección — Grupo de objetos
  https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
- Iterador — Objeto que itera sobre una colección
  https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
- Lista — Colección ordenada de objetos
  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Conjunto — Colección sin objetos duplicados
  https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
- Mapa — Objeto que relaciona claves con valores
  https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Arrays

- El concepto de array en Java es el mismo que en los lenguajes procedimentales, es decir, un conjunto de elementos del mismo tipo que ocupan posiciones consecutivas de memoria para facilitar su acceso de forma sencilla a través de índice
- En Java un array de datos es un objeto
  - Se debe usar **new** para reservar memoria, **no** es suficiente con declararlo
  - Hereda todos los métodos de la clase **Object** y, además, tiene como **atributo público la longitud del array** (length)
- Son las únicas estructuras que permiten almacenar tipos de datos primitivos

Arrays

- El acceso a los elementos de un array en Java es muy parecido al de los lenguajes procedimentales
  - Se accede a los elementos a través de un **índice**
  - Índice del primer elemento = **0**
  - Índice del último elemento = **length-1**
  - Si se intenta acceder a una posición > length-1 se genera un error **ArrayIndexOutOfBoundsException**

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

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

```
for(int i=0;i<numEjercitos.length;i++)
    numEjercitos[i]= 14 + i;
```

El acceso se realiza a través del **índice** usando **[i]**

Arrays

- Los arrays tienen importantes limitaciones desde el punto de vista de las operaciones que se pueden realizar sobre los datos
  - Tienen un **tamaño fijo**, que no se puede modificar
  - No es posible **borrar** datos cuando son tipos primitivos

```
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

Arrays

- Buenas prácticas de programación (XIV)

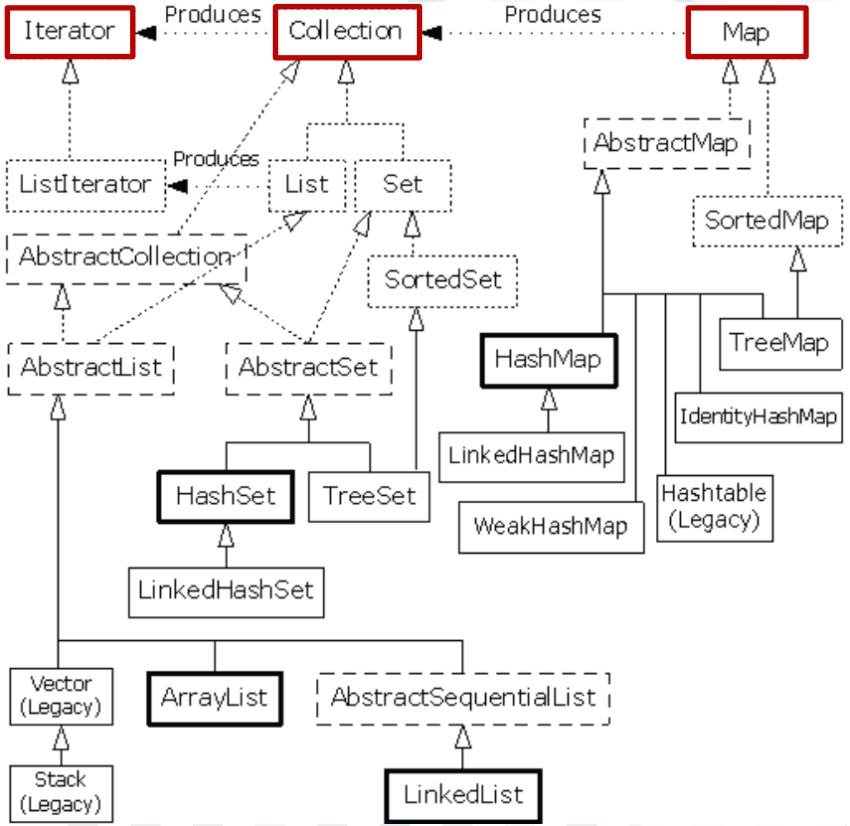
  El uso de arrays se debería limitar a situaciones en las que el tamaño de los datos a almacenar es fijo y no es necesaria la eliminación de los datos

- Buenas prácticas de programación (XV)

  Los arrays deberían de utilizar únicamente cuando se invocan métodos de otras clases que devuelven arrays y convertir esos arrays a otras estructuras es ineficiente

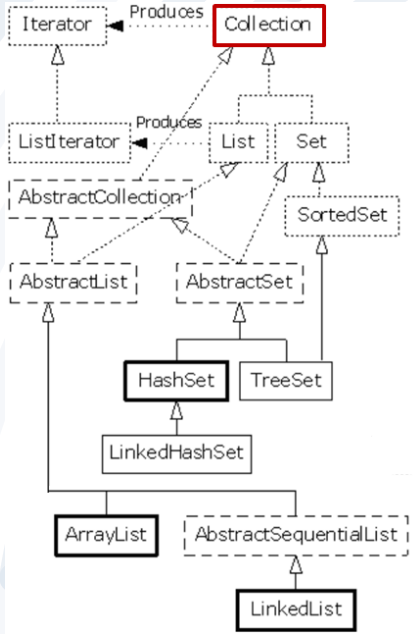
Colecciones, listas, iteradores y mapas

- Todos los tipos de conjuntos de datos son mapas (Map), colecciones (Collection) o iteradores (Iterator)
- Los otros tipos de conjuntos de datos tienen características que particularizan el tipo del que hereda
- Los **ArrayList** y los **HashMap** son tipos de listas y de mapas, respectivamente, que se usan con mucha frecuencia en los programas de Java



Colecciones: Collection

- Las colecciones son grupos o conjuntos de datos o elementos sobre los que se definen operaciones de inserción, borrado y actualización
  - Collection es un interface que define las operaciones que debe tener una colección
  - No se puede reservar memoria para un objeto de tipo Collection, ya que no es una clase, de modo que son otras clases las que implementan el comportamiento/métodos de una colección
- No se puede recorrer una colección a través de un índice, ya que no es un grupo ordenado de elementos



Colecciones: Collection

- Métodos de Collection más frecuentemente usados
  - **boolean add**(**E ele**), permite añadir el elemento **ele** a la colección
  - **boolean contains**(**Object obj**), comprueba si el objeto **obj** se encuentra en la colección
  - **boolean isEmpty**( ), indica si la colección está vacía
  - **void remove**(**Object obj**), eliminar el objeto **obj** a la colección
  - **Iterator**<E> **iterator**( ), genera un iterador que se puede usar para recorrer la colección
- Los métodos **contains** y **remove** hacen uso del método **equals** para comprobar si el objeto se encuentra en la lista

Colecciones: Collection

- Algunas implementaciones del interface Collection realmente no soportan todos sus métodos, generando una exception del tipo UnsupportedOperationException
  - Ejemplo: La implementación del método values( ) de la clase HashMap devuelve una Collection cuyo método para añadir elementos (add) no está realmente soportado

```
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

Colecciones: Collection

- Una de las formas de recorrer todos los elementos de una colección es a través de un bucle for-each

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);
```

colContinentes es la colección que se recorrerá en el bucle **for-each** y que contiene todos los elementos a los que se desea acceder

continente es el objeto que se extrae de la colección en cada ejecución del bucle **for-each**, de modo que en la ejecución completa del bucle no se repite ninguno de los elementos

Colecciones: Collection

- Las colecciones **no se pueden modificar** mientras se están recorriendo, en cuyo caso se genera una excepción del tipo ConcurrentModificationException

```
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$HashIterator.nextNode(HashMap.java:1445)
        at java.util.HashMap$ValueIterator.next(HashMap.java:1474)
        at risketse.Mapa.borrarContinente(Mapa.java:82)
        at risketse.Menu.crearMapa(Menu.java:300)
        at risketse.Menu.<init>(Menu.java:53)
        at risketse.RiskETSE.main(RiskETSE.java:14)
```

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

Colecciones: Collection

```
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

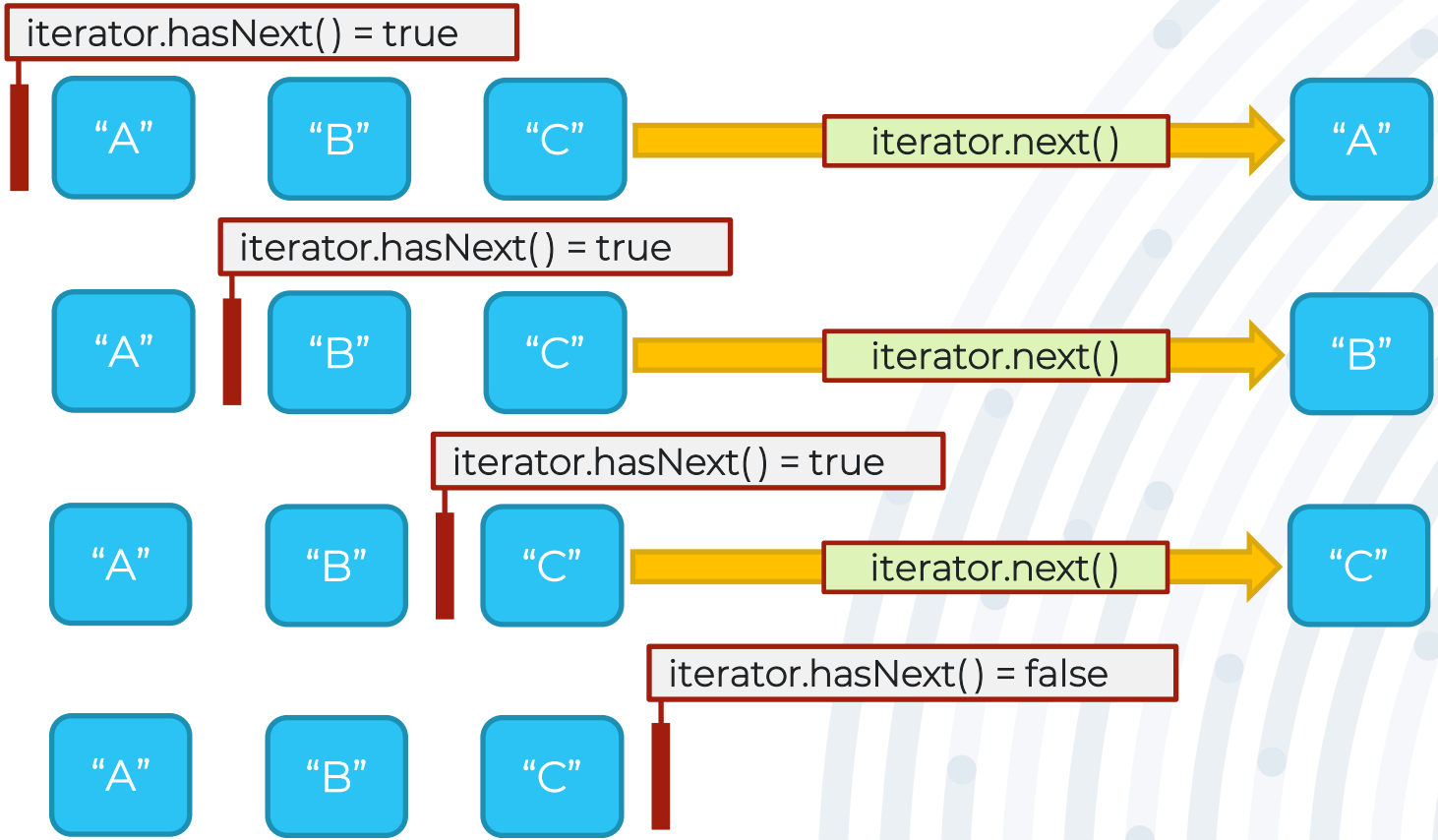
```
java.lang.UnsupportedOperationException
        at java.util.AbstractCollection.add(AbstractCollection.java:262)
        at risketse.Menu.crearMapa(Menu.java:303)
        at risketse.Menu.<init>(Menu.java:54)
        at risketse.RiskETSE.main(RiskETSE.java:14)
```

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

Colecciones: Iteradores

- Un Iterator es un interface en el que se definen un conjunto de operaciones que se pueden usar para recorrer los elementos de una colección
- Para recorrer todos los elementos se usa un puntero o iterador que señala al siguiente elemento de la colección, de modo que las operaciones básicas de un Iterator son
  - boolean hasNext( ), indica existe un elemento en la siguiente posición en la que se encuentra el puntero
  - E next( ), obtiene el elemento que se encuentra en la posición siguiente y actualiza el puntero a la posición siguiente
  - remove( ), elimina el elemento que está en la posición actual

Colecciones: Iteradores



Colecciones: Iteradores

continentes es un objeto de HashMap que, a su vez, genera un objeto que es una colección de continentes (**colContinentes**), a partir del cual **se genera un iterator** (**itContinentes**)

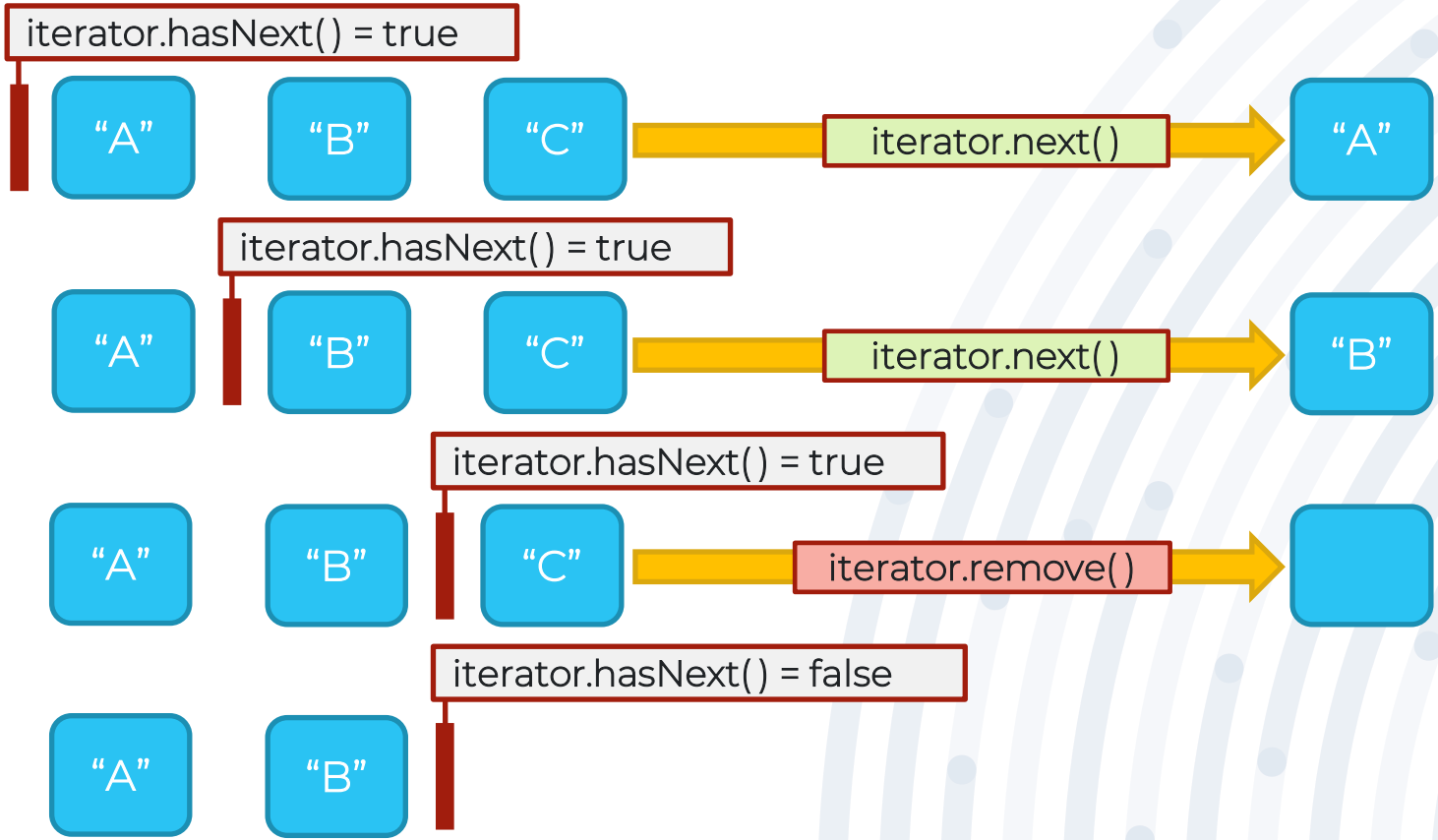
En cada iteración se debe invocar **una única vez a next()**, ya que si se invoca varias veces, el puntero del iterator **se desplazará a lo largo de la colección** tantas veces como haya tenido lugar la invocación

```
private void generarPaises() {
    Collection<Continente> colContinentes= continentes.values();
    Iterator<Continente> itContinentes= colContinentes.iterator();
    while(itContinentes.hasNext()) {
        Continente continente= itContinentes.next();
        System.out.println("Añadir países de continente " + continente);
        for(Pais paisContinente : continente.getPaises()) {
            paises.put(paisContinente.getAbreviatura(), paisContinente);
            System.out.println("País añadido -> " + paisContinente);
        }
    }
}
```

Colecciones: Iteradores

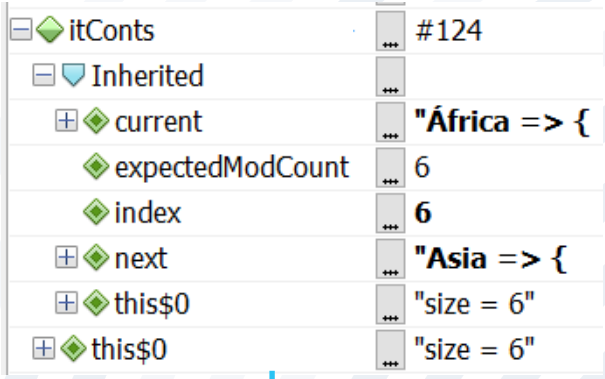
- Un iterador es una forma alternativa a for-each para recorrer los elementos de una colección
  - A diferencia de for-each, un iterador permite la eliminación de los elementos mientras se recorre la colección, eliminado tanto los elementos de la colección como del iterador
  - Un iterador solamente se puede usar una única vez para recorrer los elementos de la colección, ya que el puntero del iterador habrá llegado al final de la colección y hasNext devolverá siempre falso
  - El rendimiento de un iterador y un for-each es el mismo, ya que en realidad el compilador interpreta un for-each como si fuese un iterator

Colecciones: Iteradores



Colecciones: Iteradores

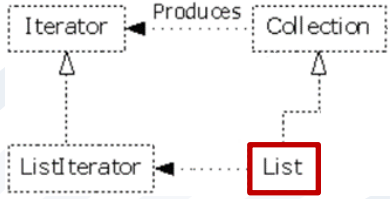
```
public Collection borrarContinente(Continente aBorrar) {
    Collection<Continente> colConts= continentes.values();
    Iterator<Continente> itConts= colConts.iterator();
    while(itConts.hasNext()) {
        Continente continente= itConts.next();
        if(continente.equals(aBorrar))
            itConts.remove();
    }
    return colConts;
}
```



Al invocar itConts.remove( ) no se genera ningún error porque el valor del puntero del iterador se **actualiza** al valor que tenía antes de haber invocado al método next( ), que es el que provoca que el valor del puntero aumente una posición

next( ) obtiene tanto el objeto actual como el siguiente, mientras que **hashNext( )** comprueba si existe el siguiente

Listas: List



- Las listas son colecciones en las que los elementos tienen un orden que está relacionado con la secuencia en la que dichos elementos han ido siendo introducidos durante la ejecución del programa
  - Una lista **mantiene siempre el orden** en el cual se han introducido los elementos en la colección, de modo que a cada elemento se le asigna un índice que indica el lugar en el que ha sido introducido durante la ejecución del programa
  - List es un interface que, además de soportar las operaciones de una colección, define las operaciones que debe tener una lista
- Una lista se puede recorrer usando un índice para acceder a cada elemento de la colección ordenada

Listas: List

- Una lista, al ser una colección, también se puede recorrer a través de un bucle de tipo **for-each**
- Una lista puede contener objetos duplicados, es decir, una referencia a un objeto puede aparecer en más de una ocasión en la colección
- El interface **List** añade otros métodos a **Collection**, que están pensados para acceder a los datos a través de un índice
  - **E get(int i)** , obtiene el objeto que está en la posición i
  - **void set(int i, E ele)**, actualiza el objeto que está en la posición i
  - **E remove(int i)**, eliminar el objeto que está en la posición i

Listas: ArrayList

- La clase ArrayList es un tipo de lista que soporta la redimensión dinámica cuando el número de datos que se desea almacenar es mayor que la **capacidad** de almacenamiento de la lista
  - La **redimensión es automática**, de manera que no será necesario invocar ni programar ningún método para que esta redimensión tenga lugar
  - Si la redimensión se realiza con mucha frecuencia, se penalizará el rendimiento de la lista, ya que internamente esta redimensión supone incrementar el tamaño de un array de objetos ([ ]) a través de una operación de caché de datos (Arrays.copyOf)
- Un objeto de ArrayList puede contener objetos de tipo null

Listas: ArrayList

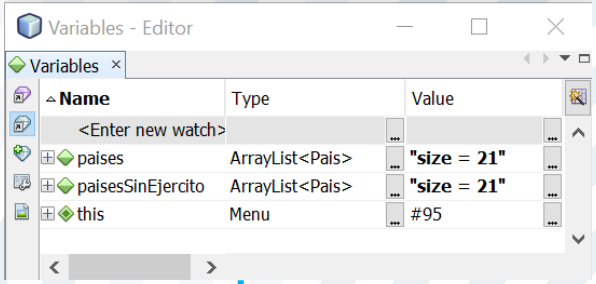
- El constructor sin argumentos de ArrayList reserva espacio para 10 elementos del tipo almacenado en la lista
  - Se diferencia entre **tamaño** del conjunto de datos que se desea almacenar y la **capacidad** de la lista
  - Cuando la lista alcanza el límite de su capacidad, por defecto **aumenta** dicha capacidad **en 10 objetos**
- Si la lista se recorre a través del índice, es posible modificar su contenido, aunque se deberá tener en cuenta que un índice mayor que el tamaño de la lista no se asigna a ningún objeto
- Al recorrer la lista es necesario comprobar que los objetos almacenados en ella son distintos de null

Listas: ArrayList

Se reserva memoria para **paisesSinEjercito**, con capacidad inicial de 10

No se reserva memoria para los elementos que se introducirán en la lista

```
ArrayList<Pais> paisesSinEjercito= new ArrayList<>();
for(int i=0;i<paises.size();i++) {
    if(paises.get(i)!=null)
        if(paises.get(i).getEjercito()==null) {
            paisesSinEjercito.add(paises.get(i));
            paises.remove(i);
        }
}
```

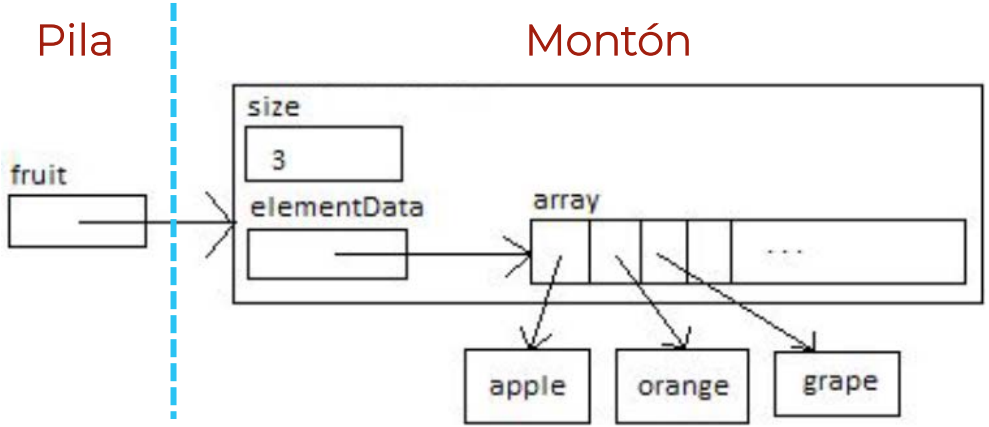


En la lista **paisesSinEjercito** se introducen los países que no tienen asignado un ejército. En la ejecución actual esta condición la cumplen 21 países de los 42 que están en la lista paises, de modo que la lista **paisesSinEjercito** se ha redimensionado de forma automática en 2 ocasiones

Los países sin ejército se elimina de la lista **países** y en cada iteración **i** se comprueba el tamaño de la lista, de modo que el índice no supera nunca el tamaño de los datos

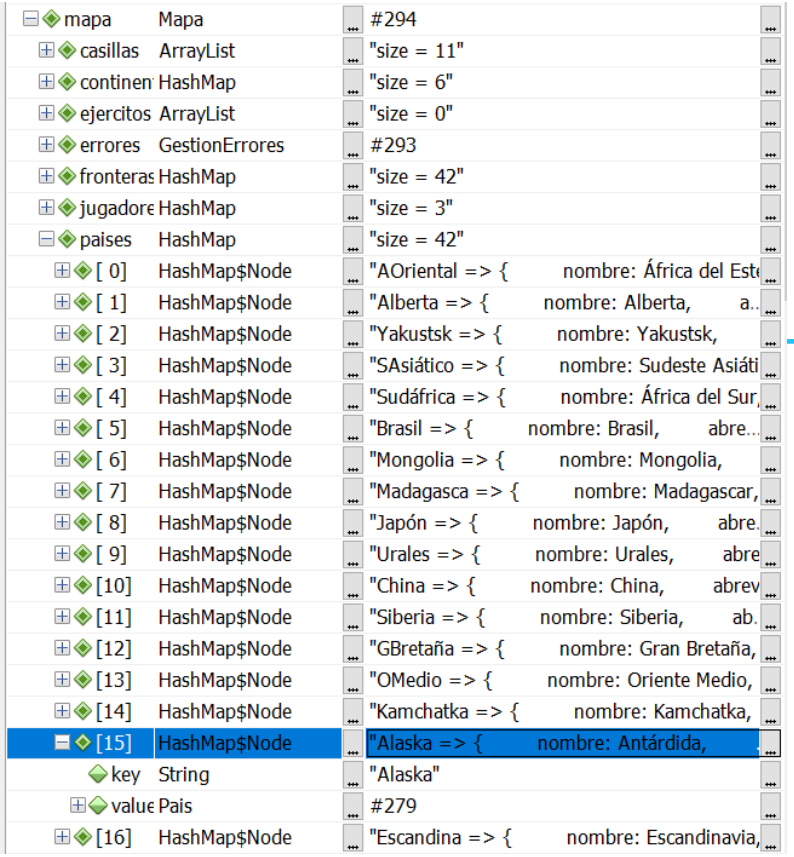
Listas: ArrayList

- La case ArrayList encapsula un array de objetos al que se accede a través de los métodos definidos en el interface List
- La referencia al array de objetos (**fruit**) se almacena en la pila, mientras que el array de objetos (**elementData**) y los otros atributos de la clase (**size**) se almacenan en el montón



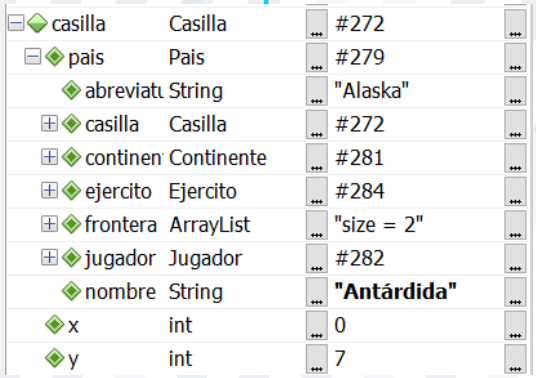
- Las operaciones de acceso a los objetos del ArrayList tienen lugar en tiempo lineal con el número de objetos almacenados en el array (O(n))

Listas: ArrayList

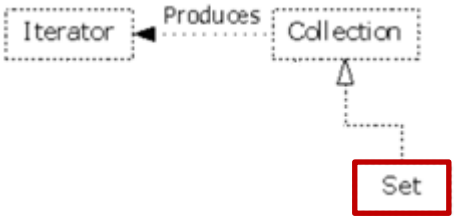


- El aliasing tiene un mayor impacto al usar conjuntos de datos, ya que se pueden modificar sus elementos sin cambiar la dirección del conjunto

```
ArrayList<ArrayList<Casilla>> casillas= mapa.getCasillas();
Casilla casilla= casillas.get(0).get(7);
Pais pais= casilla.getPais();
pais.setNombre("Antárdida");
```



Conjuntos: Set



- Los conjuntos son colecciones en las que no se repiten los datos, es decir, los datos son todos diferentes de acuerdo con el criterio de igualdad definido para los datos (método equals)
  - Set es un interface en el que se definen exactamente las mismas operaciones que en una Collection
  - Las implementaciones de Set deberán asegurar que para cada par de objetos de la colección se cumple la siguiente condición

```
Object e1, e2;
if(!e1.equals(e2)) return true;
```

  - En un Set puede haber, a lo sumo, un único null, aunque no todas las implementaciones del interface permiten almacenar null

Conjuntos: Set

- Al ser una colección, para recorrer todos los elementos de un conjunto de hace uso de un iterador o de un bucle for-each
- El criterio de igualdad es crítico en la gestión de los datos de los conjuntos, lo que en la práctica obliga a sobrescribir el método equals en las clases a las que pertenecen los objetos del conjunto
- Uno de los factores que se deben controlar en un conjunto es que una modificación de un objeto no dé como resultado que dicho objeto sea igual a alguno de los objetos guardados en el conjunto
  - Se generan inconsistencias cuando estas modificaciones no se realizan con los métodos de Set

Conjuntos: Set

- 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

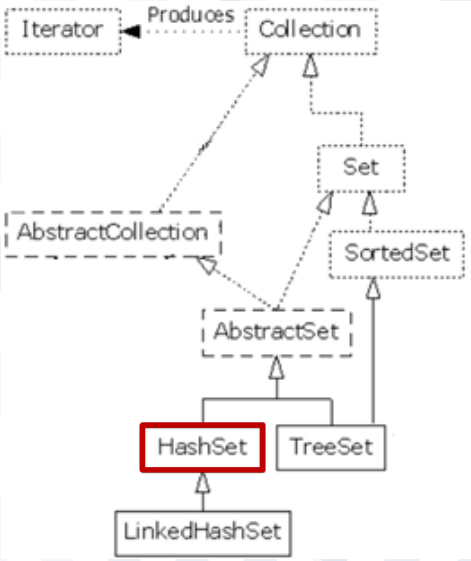
```
32  System.out.println("Países antes de modificación");
33  for(Pais pais : conjuntoPaises)
34      System.out.println(pais.getNombre());
35
36  System.out.println("Países después de modificación");
37  for(Pais pais : conjuntoPaises) {
38      if(pais.getNombre().equals("Siberia"))
39          pais.setNombre("Rusia");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

```
Países antes de modificación
Siberia
GBretaña
Rusia
```

```
Países después de modificación
Rusia
GBretaña
Rusia
```

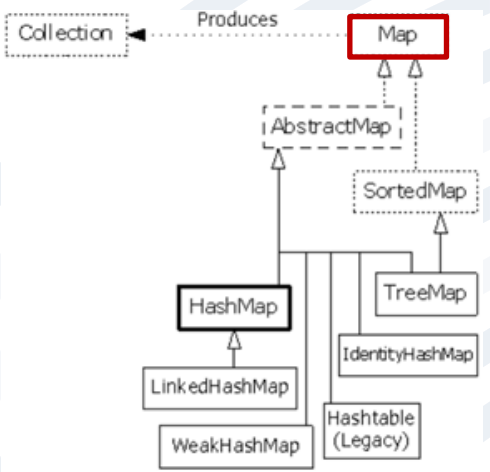
Conjuntos: HashSet

- La clase HashSet implementa el interface Set
- La clase HashSet es un wrapper a la clase HashMap, de manera que internamente los datos se almacenan como las claves de un objeto HashMap
- No se asegura que los datos se guarden en el mismo orden en el que se insertan, sino que el orden de inserción está basado en su hash code

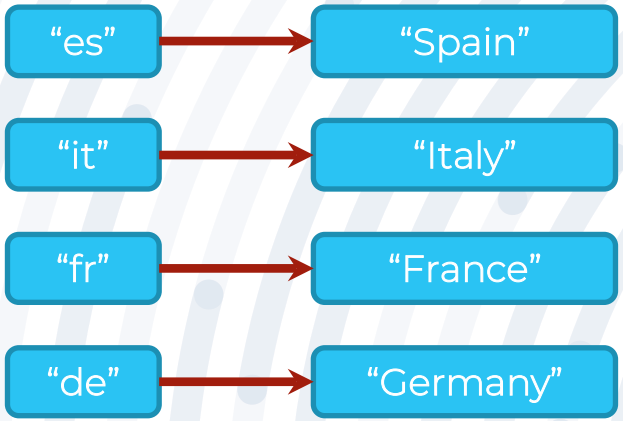


- Las operaciones de acceso a los objetos del HashSet tienen lugar en **tiempo constante**, de modo que no depende del número de objetos almacenados (O(1))

Mapas: Maps



- Un Map es un interface que relaciona una clave con un valor, de manera que a partir de la clave se obtiene el valor que está asociado a ella
  - Tanto las claves como los valores **son objetos**, es decir, no pueden ser tipos primitivos
  - Las **claves no pueden estar repetidas**
  - Los **objetos sí que pueden estar repetidos**, de modo que varias claves pueden tener asociado un mismo valor
  - Algunas implementaciones **permiten almacenar** claves y valores nulos



Mapas: Maps

- La clave de un mapa puede ser cualquier tipo de objeto
  - Para localizar y obtener la clave con la que se recupera el valor se hará uso de equals, que deberá estar sobrescrito con un criterio de igualdad definido para la clase a la que pertenece la clave
  - La clave debería de ser un objeto inmutable, es decir, que no cambia una vez se realiza la reserva de memoria
    - Si el objeto no es inmutable, entonces es necesario controlar que no se modifiquen aquellos atributos usados en equals para comprobar la igualdad entre dos objetos
  - Algunas implementaciones del interface Map realmente no soportan todos sus métodos, generando una exception del tipo UnsupportedOperationException

Mapas: Maps

- Métodos de Map más frecuentemente usados para acceder directamente a las claves y a los valores
  - V get(Object key), devuelve el valor asociado a una clave dada
  - V put(K key, V value), almacena un valor al que se le asocia una clave dada, quedando vinculado
  - boolean containsValue(Object value), indica si un valor está almacenado en el mapa, es decir, si tiene asociada una clave
  - boolean containsKey(Object key), indica si existe un dato que está asociado a una clave dada
  - V remove(Object key), elimina la dupla <clave, valor> a partir de una clave dada

Mapas: Maps

- Los objetos almacenados en un mapa no se pueden recorrer directamente, ya que el objetivo de los mapas es acceder a los datos a partir de las claves, no realizar una búsqueda de datos usando otros criterios diferentes de las claves
- El interface Map define tres métodos que convierten las claves y los datos a colecciones (Collection) o a conjuntos (Set)
  - Collection<V> values( ), devuelve una colección que contiene los valores del mapa
  - Set<K> keySet( ), devuelve un conjunto con las claves del mapa
  - Set<Map.Entry<K, V>> entrySet( ), devuelve un conjunto con todas las duplas <clave, valor> del mapa

Mapas: Maps

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 **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(Pais paisContinente : continente.getPaises())
            paises.put(paisContinente.getAbreviatura(), paisContinente);
    }
}
```

Mapas: Maps

- Map.Entry es un interface que define métodos para acceder a una entrada de un mapa, es decir, a la dupla <clave, valor>
  - 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

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(Pais paisContinente : continente.getPaises())
            paises.put(paisContinente.getAbreviatura(), paisContinente);
    }
}
```

Mapas: HashMap

- La clase HashMap implementa el interface Map
- Los objetos que son las claves del mapa están asociados a un código hash (un entero) que se usa para comparar si dos claves son iguales, complementando así al método equals
- Cuando **se comparan dos objetos en un HashMap** se aplica el siguiente procedimento (lo mismo que para un HashSet)
  - Primero se comprueba si los objetos tienen el mismo hash code, de modo que si no es el mismo, se considerará que los objetos son diferentes
  - Si el hash code es igual, se invoca al método equals para aplicar el criterio de igualdad y comprobar que los objetos son iguales

Mapas: HashMap

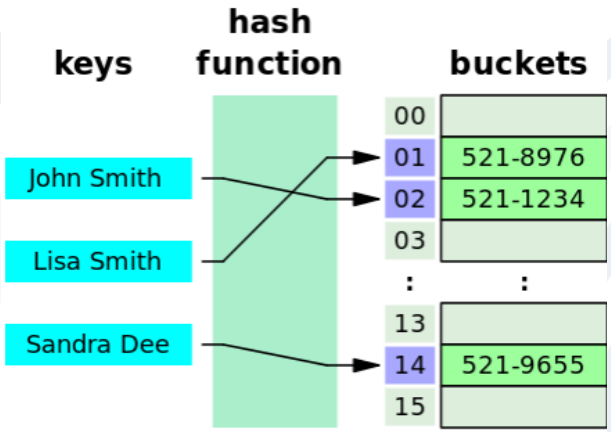
- Se asume que aunque dos objetos no sean iguales, los hash code pueden ser iguales; pero si dos objetos son diferentes, entonces los hash code deben de ser diferentes
- El uso de hash code simplifica y hace más eficiente el acceso a los datos en objetos HashMap (y HashSet), ya que en primer lugar se compara entre enteros y solo en el caso de que sean iguales tiene lugar una comparación más compleja
- Para generar el hash code, en Java es necesario sobrescribir el método hasCode( ), que toda clase hereda de la clase Object
  - La implementación de hashCode( ) en la clase Object es un método nativo (implementación específica para cada JVM)

Mapas: HashMap

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

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

- La clase HashMap implementa una tabla hash en la que los datos se almacenan de forma desordenada de acuerdo con una función hash
- Las operaciones de acceso a los objetos del HashMap tienen lugar en tiempo constante, de modo que no depende del número de objetos almacenados (O(1))



Colecciones, listas y mapas

- Buenas prácticas de programación (XVI)

  Para hacer uso de conjunto de datos es muy conveniente sobrescribir el método **equals** y, en el caso de objetos hash, también el método **hashCode**

- Buenas prácticas de programación (XVII)

  Los métodos **equals** y **hashCode** deben de usar objetos mutables para comprobar la igualdad entre objetos y generar el código hash, respectivamente

Conjuntos de datos: Comparativa

- Las características de los datos que se almacenan orientan la selección del tipo de conjunto de datos que se usarán en el programa

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

Conjuntos de datos: Comparativa

- 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

Colecciones, listas y mapas

- Buenas prácticas de programación (XVIII)

  Si se van a almacenar datos que no se pueden repetir y no se necesita acceder a ellos en el orden en el que se guardaron, lo más conveniente es hacer uso de un objeto **HashSet**

- Buenas prácticas de programación (XIX)

  Si se necesita acceder a los datos en el orden en el que se han almacenado, lo más conveniente es usar un objeto **ArrayList**

Colecciones, listas y mapas

- Buenas prácticas de programación (XX)

  Si existen restricciones de memoria debido a la cantidad de datos que se debe almacenar y la eficiencia no es la prioridad, lo más conveniente es usar un objeto **ArrayList**

- Buenas prácticas de programación (XXI)

  Si no se necesita acceder a los datos en el orden en el que se han almacenado y se dispone de un identificador unívoco, lo más conveniente es usar objeto **HashMap**