v 1.5

Tema 6
Genericidad y estructuras de datos

Programación Orientada a Objetos

José Francisco Vélez Serrano

Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Índice

La genericidad

La genericidad permite definir variables para los tipos de manera que los algoritmos puedan operar con diferentes tipos de datos. En esta sección veremos el soporte que proporciona Java a la genericidad.

Antes de los tipos genéricos

Una de las principales estructuras de datos genéricas proporcionadas por Java es ArrayList.

Hasta Java 5, las estructuras de datos se basaban en el uso polimórfico de Object.

El uso de Object tiene el problema del casting que hay que hacer a los datos devueltos, ya que la estructura de datos no sabe de qué tipo es el dato almacenado y esto puede ocasionar problemas.

ArrayList
+get(int):Object +add(Object) +set(int,Object) +subList():Object +indexOf(Object):int +remove(int) +contains(Object):bool

```
import java.util.ArrayList;


class Coche {

}


... metodoExterno() {
  ArrayList l = new ArrayList();
  Coche c = new Coche();
  l.add(c);
  Coche c2 = (Coche) l.get(0);
}
```

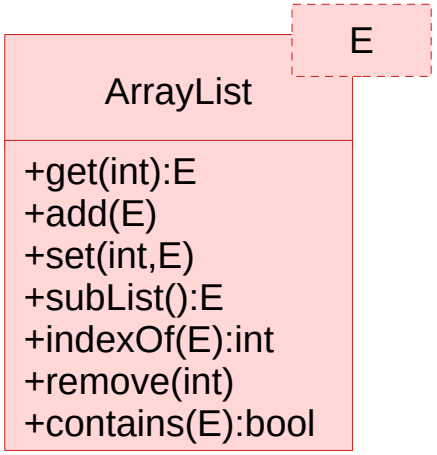
Tipos genéricos y estructuras de datos

Desde Java 5, las estructuras de datos de Java se basan en el uso de tipos genéricos.

Al declarar una estructura de datos se debe indicar de qué tipo es.

Por ello, el compilador impide insertar elementos en la estructura de datos que no sean del tipo indicado.

Además, se hace innecesario el casting al obtener elementos.



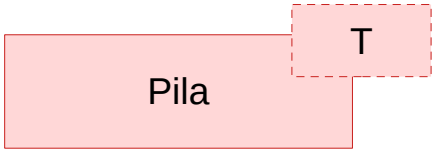

```
import java.util.ArrayList;


class Coche {

}


... metodoExterno() {
  ArrayList <Coche> l = new ArrayList<>();
  Coche c = new Coche();
  l.add(c);
  Coche c2 = l.get(0);
}
```

Concepto de genericidad

La genericidad se define como la capacidad que permite a un lenguaje declarar tipos mediante parámetros variables.

```
[Modificador de clase] class <ident_clase> [parámetros] [herencia] [excepciones]{
     [método|propiedad|clase|bloque inicialización]*
}

parametros ::= ['<'ident_tipo [extends tipo][implements tipo[&tipo]* '>']
               [,'<'ident_tipo [extends tipo][implements tipo[&tipo]*]]*'>']
```

En UML una clase que usa genéricos se representa con un rectángulo punteado en su esquina derecha.

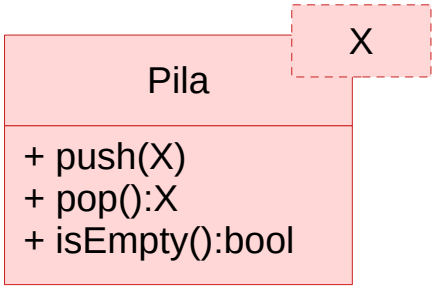



```
class Pila <T> {

  public void push(T valor) {
    ...
  }

  public T pop() {
    ...
  }

  public boolean isEmpty() {
    ...
  }
}
```

Ejemplo de genericidad

```
public class Pila <X> {

  private class Nodo {
    Nodo siguiente;
    X valor;
  }

  private Nodo cima;

  public void push(X valor) {
    Nodo aux = new Nodo();
    aux.valor = valor;
    aux.siguiente = cima;
    cima = aux;
  }

  public X pop() {
    if (cima == null)
      throw new RuntimeExpception("pila vacia");
    Nodo aux = cima;
    cima = cima.siguiente;
    return aux.valor;
  }

  public boolean isEmpty() {
    return cima == null;
  }
}
```

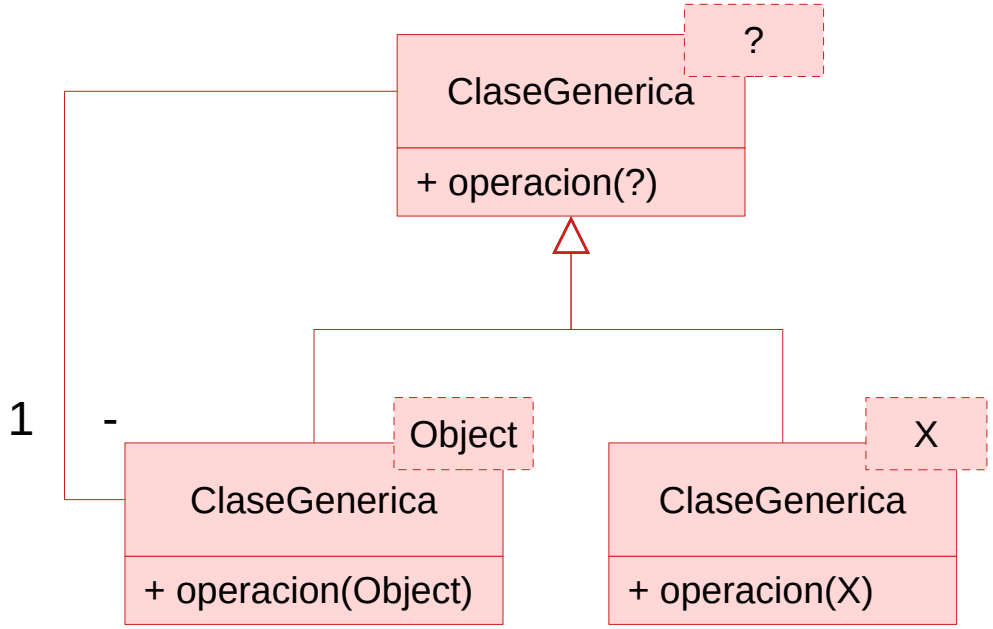
La genericidad se utiliza principalmente para construir estructuras de datos independientes del tipo.




```
class Coche {
}

public static void main(String[] args)

Pila <Coche> p = new Pila<>();
Coche c = new Coche();
p.push(c);
Coche c2 = p.pop();
```

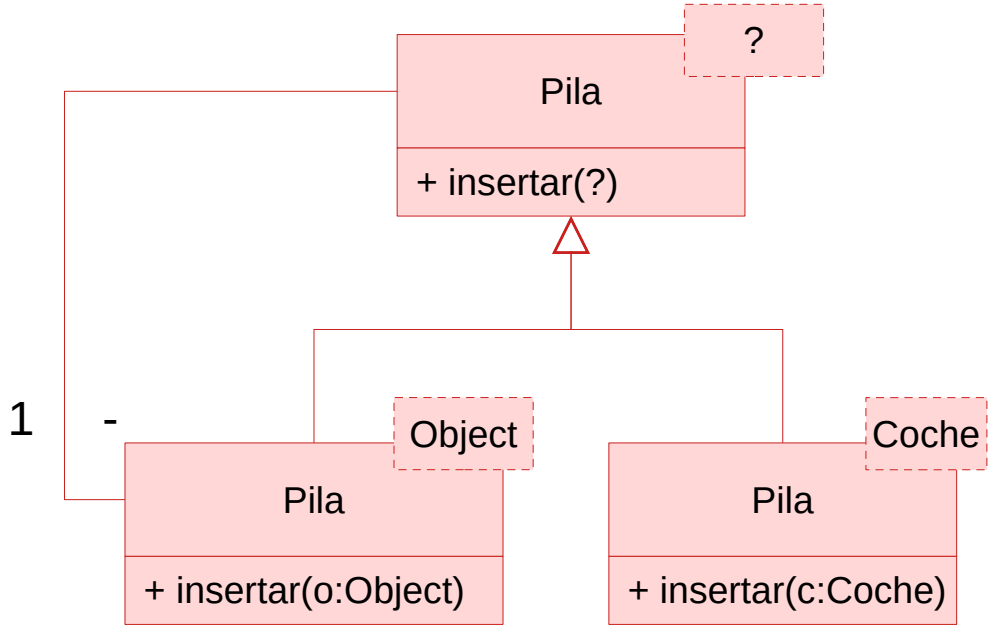
Genericidad sobre polimorfismo

En Java la genericidad se ha implementado usando internamente polimorfismo.



Genericidad sobre polimorfismo

Por ejemplo, una Pila genérica dentro contiene de manera privada una Pila de Object.

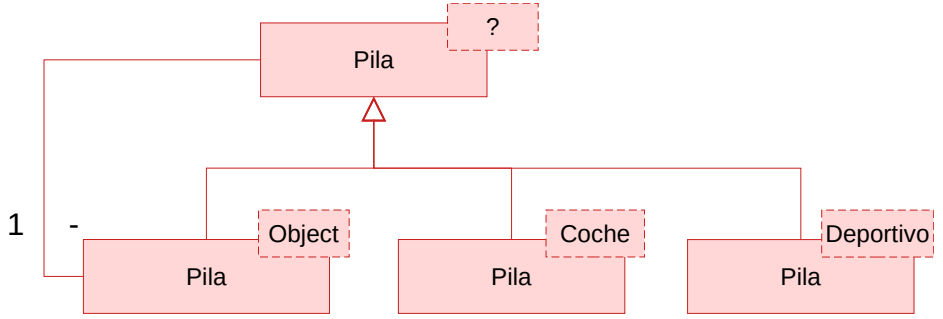


Observar que no se hereda de Pila de Object porque implicaría que Pila de Coches tendría el método insertar(Object). Por eso aparece el concepto del comodín o wildcard (?).

Los genéricos son invariantes

Los genéricos de java son invariantes. Esto quiere decir que no puedes asignar objetos de subclases genéricas a referencias de super clases genéricas. Es decir que el siguiente código da error:

```
Pila <Coche> = new Pila<Deportivo>();
```



Esto limita la mezcla de generícos y polimorfismo. Por ello en los genéricos de Java disponen de los comodines.

```
Pila <? extends Coche> = new Pila<Deportivo>();
```

Los genéricos son invariantes

Por otro lado, la clase array de Java es Covariante. Es decir, se puede asignar un array de una subclase a una referencia de un array de una superclase.

```
Coche [] arrayDeCoches = new Deportivo [25];
```

Esta diferencia entre arrays y genéricos impide crear array de genéricos.

```
Pila <Coche> [] arrayDeCoches = new Pila <Coche> [25];
```

```
Pila <Coche> [] arrayDeCoches = new Pila [25];
arrayDeCoches[0] = new Pila<>();
...
arrayDeCoches[24] = new Pila<>();
```

Las envolturas

Java dispone de una clase por cada tipo primitivo. Estas clases se llaman envolturas.

Clases de envoltura

Un tipo genérico puede ser instanciado con cualquier clase o interfaz de Java, pero no por un tipo primitivo.

Para solventar el problema de usar tipos primitivos en tipos genéricos Java ha creado las clases de envoltura.

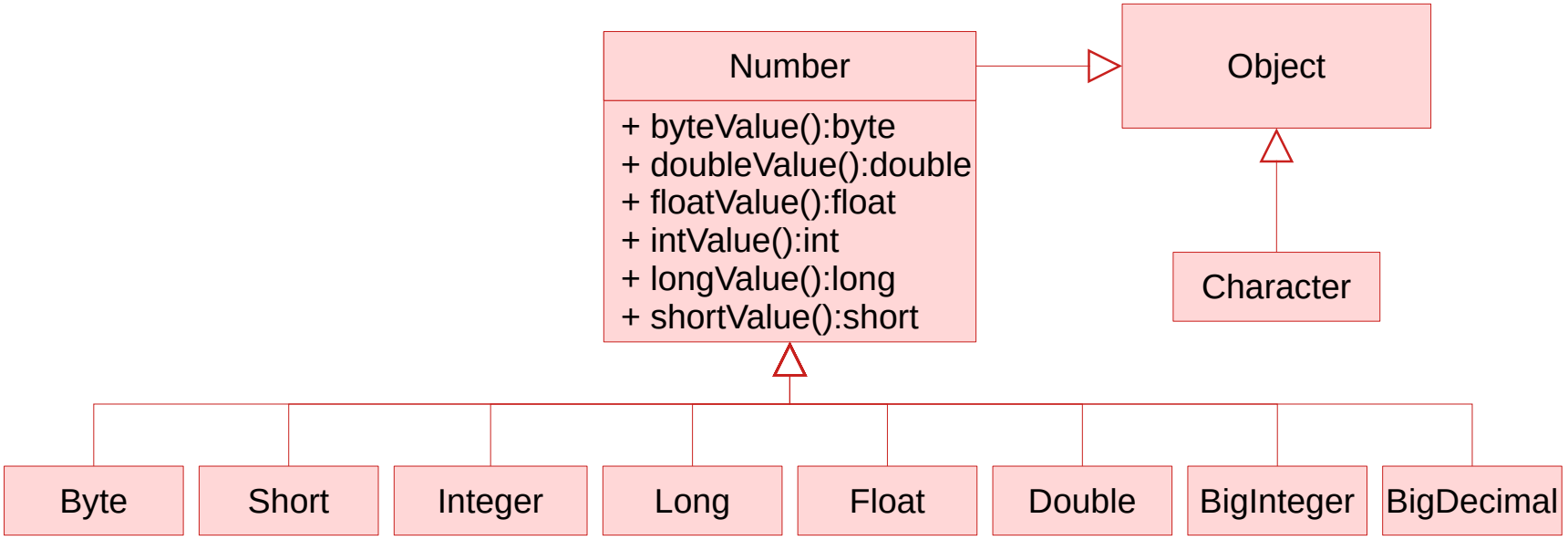
Hay una clase de envoltura por cada tipo primitivo: Character, Byte, Short, Integer, Long, Float y Double.

```
import java.util.ArrayList;

...metodoExterno() {
  ArrayList <Integer> l = new ArrayList<>();
  l.add(5);
  int v = l.get(0);
}
```

Clases de envoltura

Menos Character, todas las envolturas de tipos primitivos derivan de Number.

Además, derivando de Number hay otras clases de interés como BigDecimal o BigInteger.



Autoboxing y autounboxing

Con las envolturas de Java no es necesario invocar al constructor de los objetos. Java lo invoca automáticamente en lo que se conoce como el autoboxing.

```
Integer n = new Integer(12);
```

Tampoco es necesario invocar a métodos para extraer el tipo primitivo de un envoltorio. Java lo hace automáticamente en lo que se conoce como autoinvoxing.

```
Integer n = 12;     //Autoboxing de 12

int x = n;   //Autounboxing de n

n++; //Autounboxing, suma de uno y autoboxing
```

Las envolturas

Pobre rendimiento respecto a tipos primitivos

Hay que evitar el uso de envolturas por la penalización en rendimiento que suponen las operaciones de autoboxing y autounboxing.

```
public class VelocidadEnvoltorios {
  public static void main(String[] args) {
    long start1 = System.nanoTime();

    Integer a = 0;
    for (Integer x = 1; x < 100000; x++) {
      a++;
    }
    long finish1 = System.nanoTime();
    System.out.println("Con envoltorio " + Long.toString(finish1 - start1)+"
(ns)");

    long start2 = System.nanoTime();
    int b = 0;
    for (int x = 1; x < 100000; x++) {
      b++;
    }
    long finish2 = System.nanoTime();
    System.out.println("Sin envoltorio " + Long.toString(finish2 - start2)+"
(ns)");
  }
}
```

```
c:/Tmp> javac EjemploUsoSet.java
c:/Tmp> java EjemploUsoSet
Con envoltorio 8931645 (ns)
Sin envoltorio 251694 (ns)
```

Operaciones de utilidad del tipo asociado

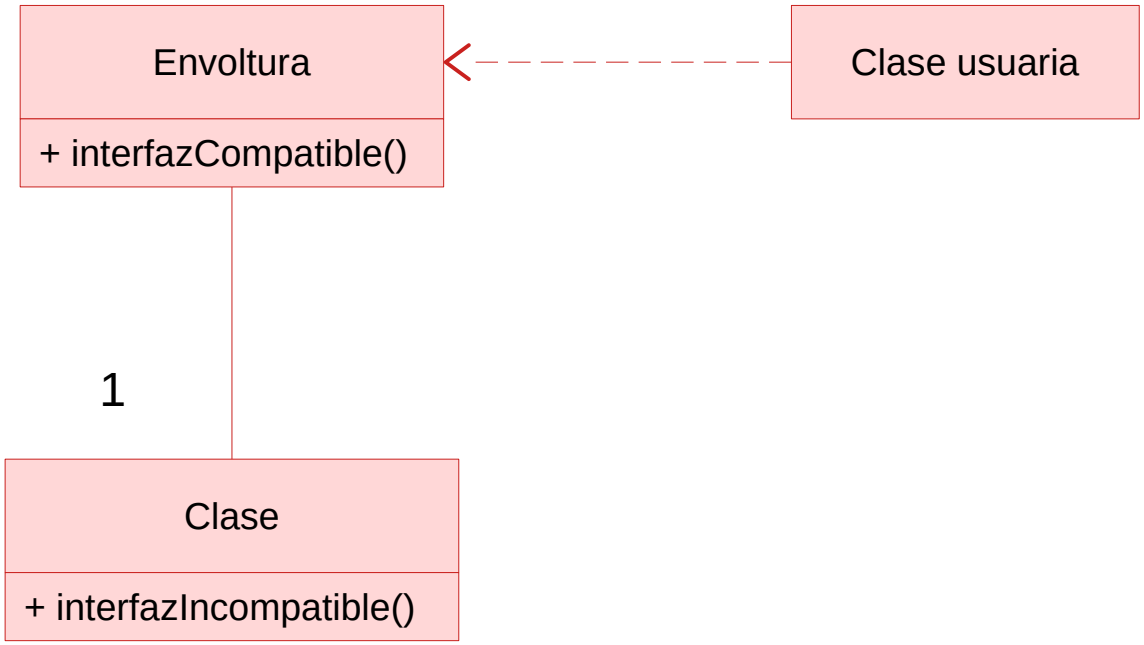
Las clases de envoltura también han sido utilizadas por Java para contener diversos métodos de utilidad como:

- Conversión desde String y a String
- Conversión a otras bases (hexadecimal, binario, octal…)
- Rotaciones e inversiones
- Obtención de signo
- ...

Integer
int MAX_VALUE int MIN_VALUE int SIZE
Integer(int value) Integer(String s) int bitCount(int) byte byteValue() int compare(int x, int y) int compareTo(Integer anotherInteger) Integer decode(String nm) double doubleValue() boolean equals(Object obj) float floatValue() Integer getInteger(String nm) int hashCode() int highestOneBit(int) int intValue() long longValue() int lowestOneBit(int) int numberOfLeadingZeros(int) int numberOfTrailingZeros(int) int parseInt(String s) int reverse(int) int reverseBytes(int) int rotateLeft(int i, int distance) int rotateRight(int i, int distance) short shortValue() int signum(int) String toBinaryString(int) String toHexString(int) String toOctalString(int) String toString() String toString(int) Integer valueOf(int) Integer valueOf(String s)

Patrón de diseño envoltura

Las envolturas de Java son un ejemplo de un patrón de diseño conocido como Wrapper o Adapter.



Los iteradores

Un iterador es un objeto que facilita recorrer cualquier estructura de datos.

Concepto de iterador

En POO es habitual que se puedan recorrer todos los elementos de cualquier estructura de datos.

Para recorrerlas se usan unos objetos que se conocen como iteradores (Iterator).

La interfaz Iterator está en el paquete java.util.Iterator.

Un iterador es un objeto que solo tiene dos métodos:

- Dame el siguiente (next)
- ¿Hay siguiente? (hasNext)

En Java todas las estructuras de datos son iterables.

Recorriendo un objeto iterable

Para recorrer un objeto iterable solo hay que:

1. Obtener un iterador al objeto con `iterator()`
2. Preguntar si hay siguiente con `hasNext()`
3. Mientras haya siguiente obtener el valor con `next()`

```
metodoExterno() {
  ArrayList <Integer> lista = new ArrayList<>();
  lista.push(1);
  lista.push(1);
  lista.push(2);
  lista.push(3);
  lista.push(5);

  Iterator <Integer> it = lista.iterator();
  while (it.hasNext()) {
    int i = it.next();
    System.out.println(i);
  }
}
```

Los iteradores

for each

En la versión 5 de Java se modificó el lenguaje para simplificar el recorrido de las estructura de datos.

Se modifico el bucle “for” para que admitiese recorrer objetos iterables. A estos bucles se les conoce como for-each.

```
metodoExterno() {
  ArrayList <Integer> lista = new ArrayList<>();
  lista.push(1);
  lista.push(1);
  lista.push(2);
  lista.push(3);
  lista.push(5);

  for (int i : lista) {
    System.out.println(i);
  }
}
```

Los iteradores en los arrays y en los enum

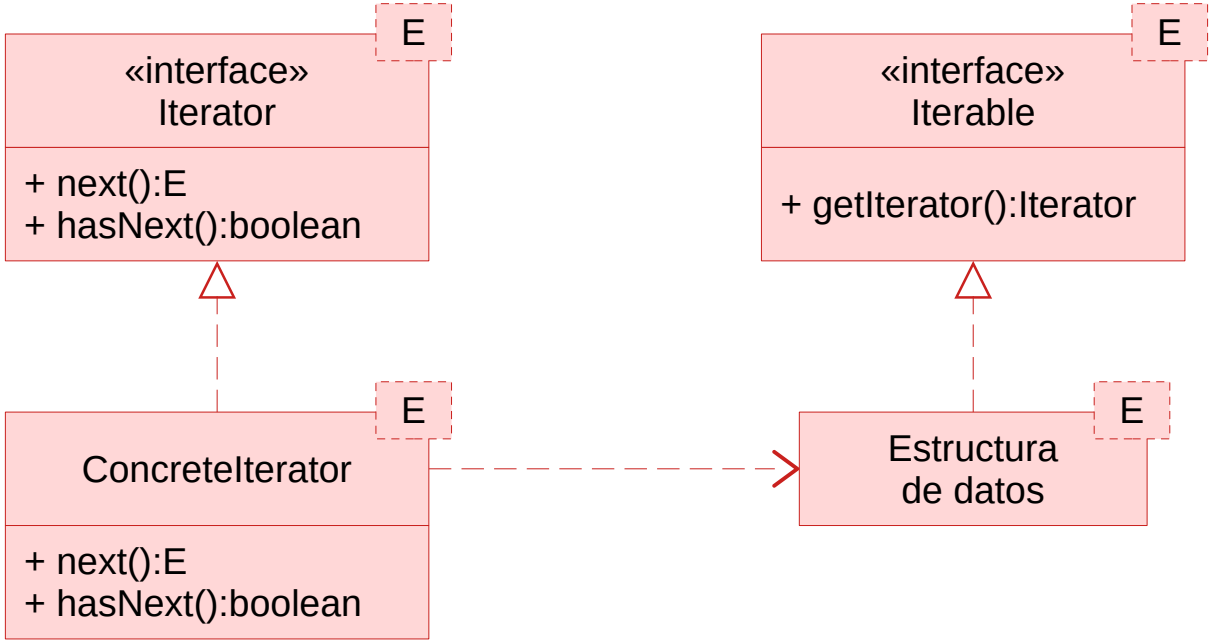
Los arrays y los enum también se pueden recorrer con un for-each.

```
public class EjemplosForEach {

  enum Dia {L, M, X, J, V, S, D}

  public static void main(String[] args) {
    Dia d = Dia.S;

    for(Dia c: Dia.values()) {
      System.out.println(c);
    }

    int [] vector = {1,1,2,3,5,8,13};

    for (int n : vector) {
      System.out.println(n);
    }
  }
}
```

Patrón de diseño iterador

Los iteradores de Java son un ejemplo de un patrón de diseño conocido como Iterator.

Un iterador permite recorrer los datos de otro objeto llamando al método `next()`.



Ejemplo de iterador para la pila

```java
import java.util.Iterator;

public class Pila <E> implements Iterable <E>{

  private class Nodo {
    Nodo siguiente;
    E valor;
  }

  private Nodo cima;

  public void push(E valor) {
    Nodo aux = new Nodo();
    aux.valor = valor;
    aux.siguiente = cima;
    cima = aux;
  }

  public E pop() {
    if (cima == null)
      throw new RuntimeExpception("Vacia");
    E = cima.valor;
    cima = cima.siguiente;
    return valor;
  }

  public boolean isEmpty() {
    return cima == null;
  }
```

```java
  @Override
  public Iterator<E> iterator() {
    return new PilaIterator(this);
  }



  private class PilaIterator implements Iterator<E>{
    private Nodo nodo;

    public PilaIterator(Pila <E> pila) {
      nodo = pila.cima;
    }

    @Override
    public boolean hasNext() {
      return nodo != null;
    }

    @Override
    public E next() {
      E aux = nodo.valor;
      nodo = nodo.siguiente;
      return aux;
    }
  }  //Fin de la clase PilaIterator

} //Fin de la clase Pila
```

Las listas

Las estructuras de datos lineales de Java se encuentran en el paquete “util”. En esta sección se analizan las listas que implementa dicho paquete.

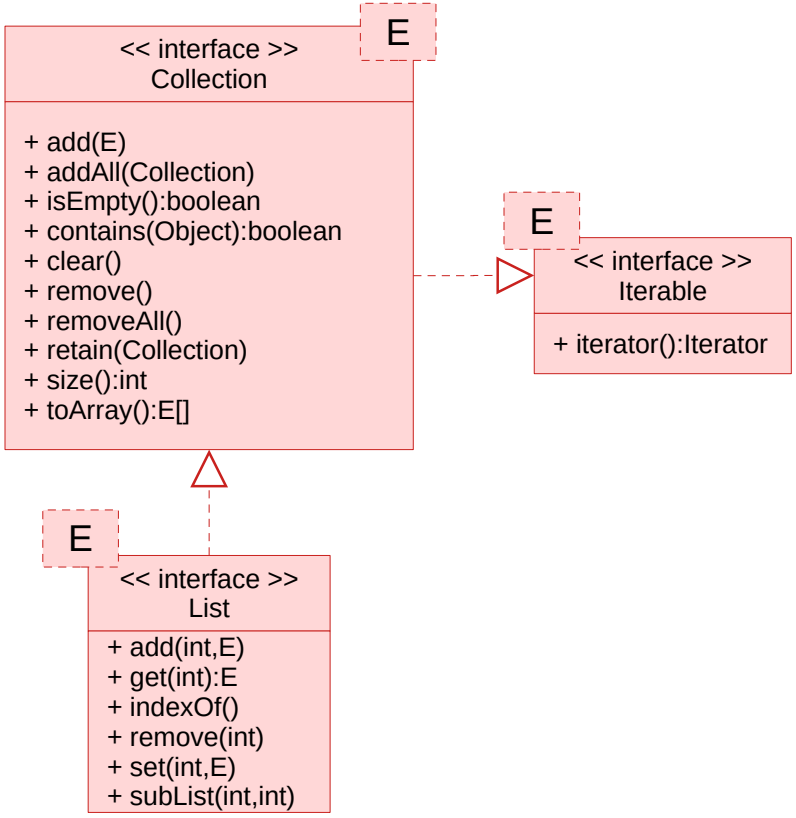
La interfaz Collection

Las Listas de Java implementan la interfaz Collection.

Todos las clases que implementan la interfaz Collection se caracterizan por constituir grupos de objetos con una relación de orden entre ellos.

Todas los objetos de tipo Collection comparten diferentes métodos, como size o clear.

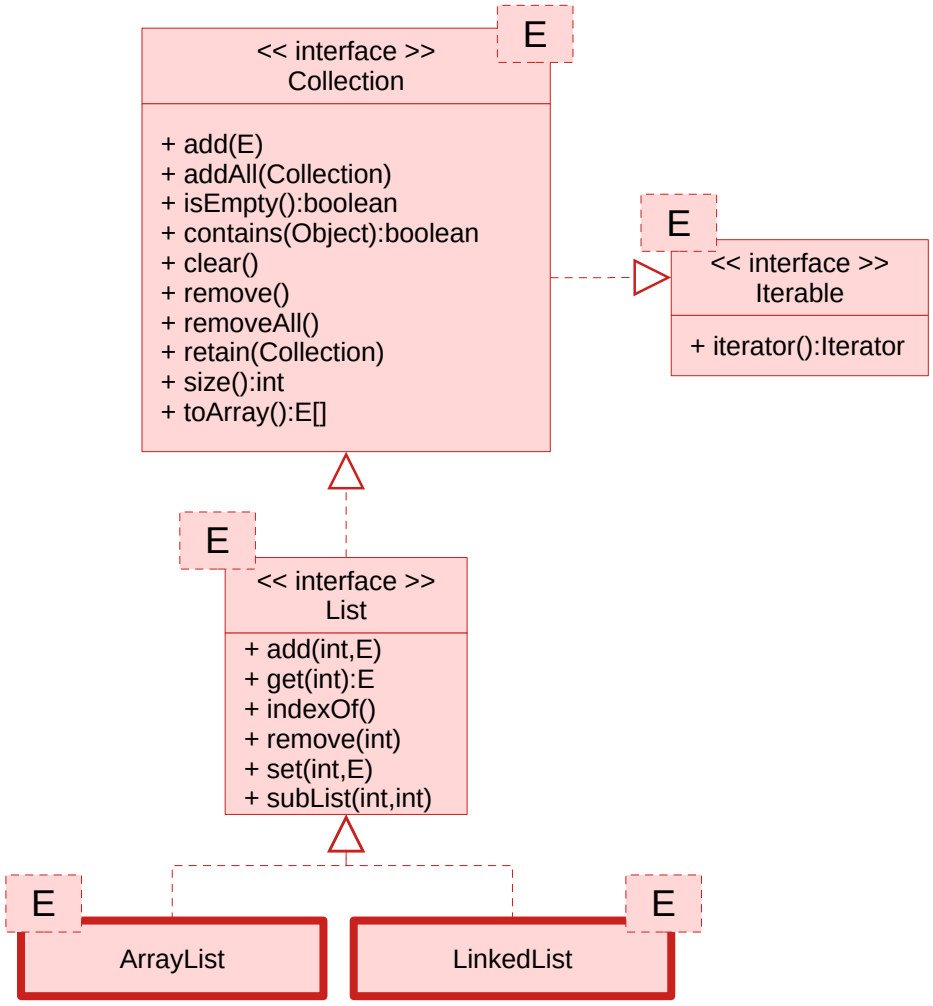
Además, toda Collection cumple Iterable.



Derivados de List

Entre los derivados de List se pueden destacar:

- ArrayList.- Clase que implementa List usando un array dinámico que crece cuando es necesario.
- LinkedList.- Clase que implementa List usando nodos doblemente enlazados.

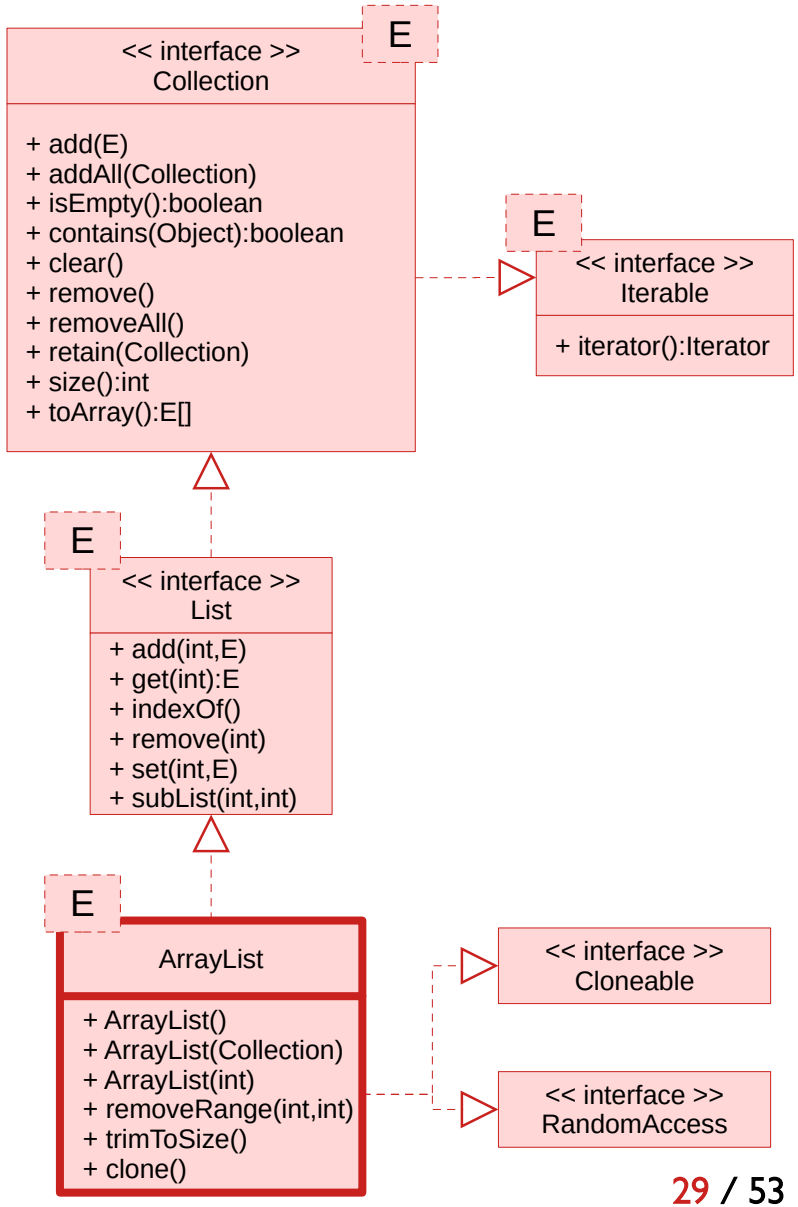


ArrayList

ArrayList es una lista, por tanto debe usarse cuando importa el orden de inserción.

Insertar en posiciones intermedias tiene complejidad $O(n)$ debido a que debe desplazar los elementos del array.

Tiene complejidad $O(1)$ al acceder a sus elementos por índice, ya que está basado en un array.



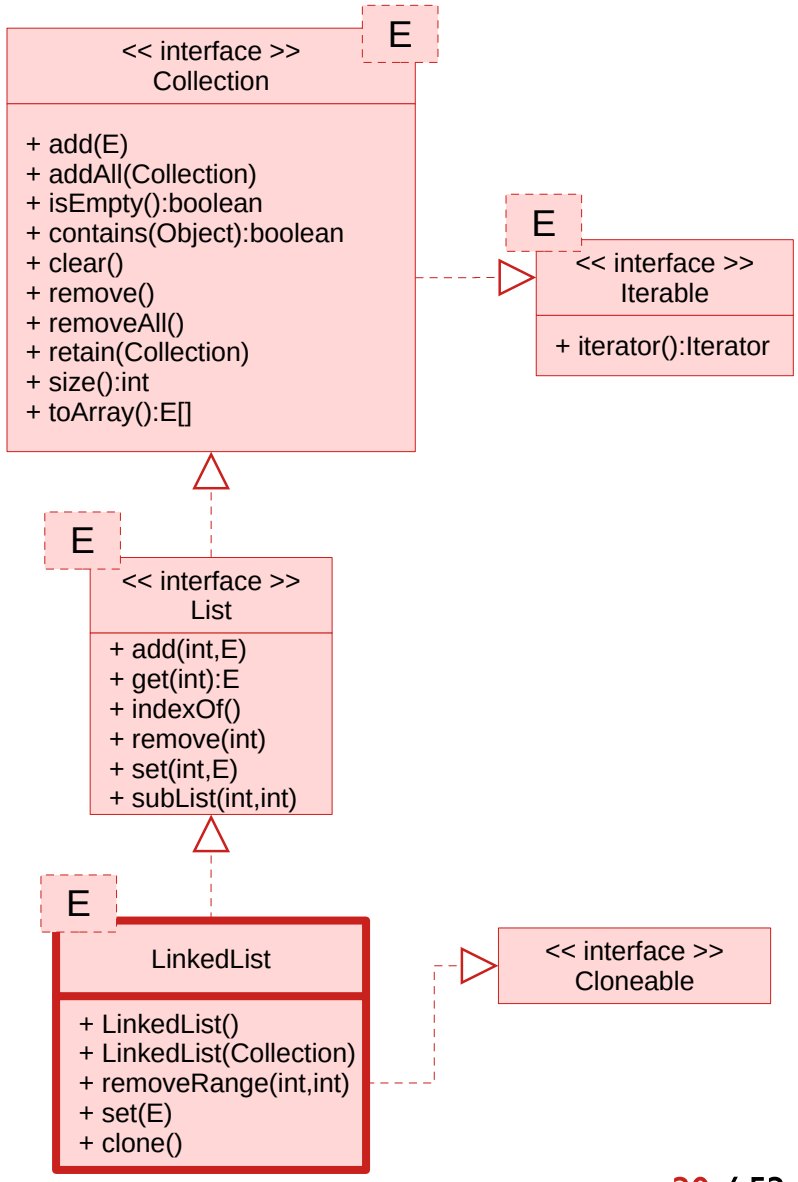
Las listas

LinkedList

LinkedList es una lista, por tanto debe usarse cuando importa el orden de inserción.

El acceso por índice a los elementos de esta estructura es $O(n)$.

La inserción de elementos se realiza en tiempo constante independientemente de su posición. Por ello aporta métodos para insertar por el principio y el final con complejidad $O(1)$.



Ejemplo de uso de listas

Obsérvese la diferencia con las definiciones formales. El método `add()` inserta elementos por el final, aunque se puede insertar en cualquier posición. Y si se recorre con un iterador los elementos aparecen en el mismo orden en que fueron insertados.

```
import java.util.List;
import java.util.ArrayList;

public class EjemploUsoList {

  public static void main(String[] args) {
    List <String> listado = new ArrayList<>();
    listado.add("Pedro");
    listado.add("Ana");
    listado.add("Juan");
    listado.add("Elena");
    listado.add("Paco",2);

    for (String nombre : listado) {
      System.out.println(nombre);
    }
  }
}
```

```
c:\Tmp> javac EjemploUsoList.java
c:\Tmp> java EjemploUsoList
Pedro
Ana
Paco
Juan
Elena
```

Preferencias

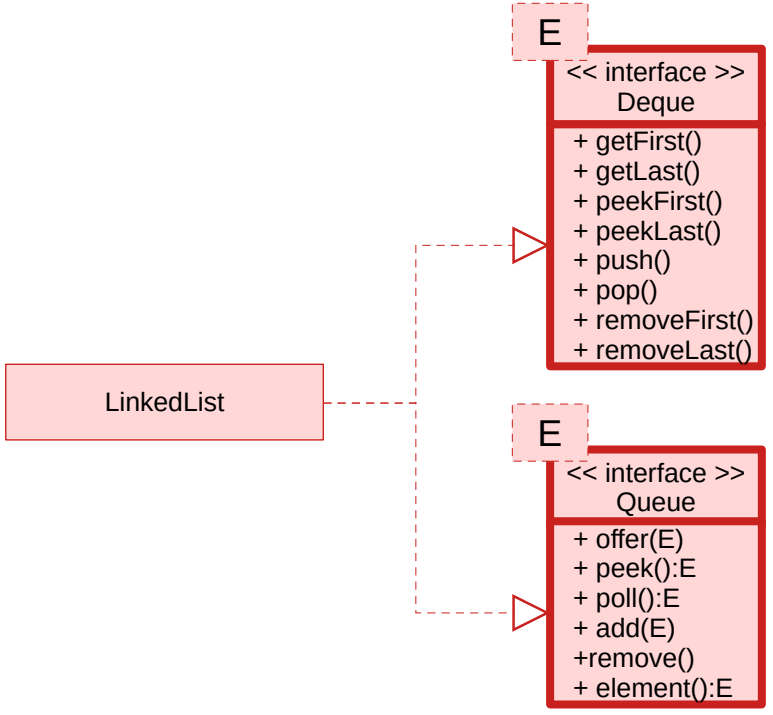
	Array List	Linked List
Cuando usar	Cuando es preciso acceder a elementos de manera aleatoria	Cuando se debe insertar frecuentemente a mitad de la lista
Cuando no usar	Cuando se debe insertar frecuentemente a mitad de la lista	Cuando no es preciso acceder a elementos de manera aleatoria

En general es preferible usar ArrayList, sobre todo teniendo en cuenta que las modernas CPU permiten movimiento de bloques de memoria a muy alta velocidad.

Colas

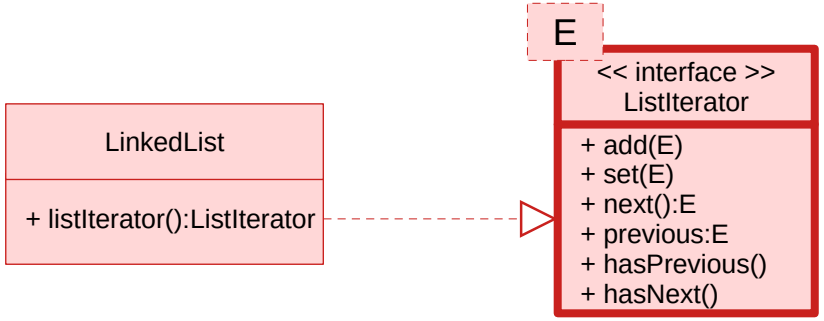
Si se desea el formalismo de una cola, LinkedList implementa las interfaces de Queue y Deque.

```
import java.util.Queue;
import java.util.ArrayList;

public class EjemploUsoQueue {

  public static void main(String[] args) {
    Queue <Integer> cola = new LinkedList<>();
    cola.offer(23);
    int v = cola.poll();
  }
}
```



ListIterator

Si se desea insertar en mitad de una LinkedList con O(1) mientras se recorre con un iterador se puede usar una variedad de iteradores que lo permite ListIterator.



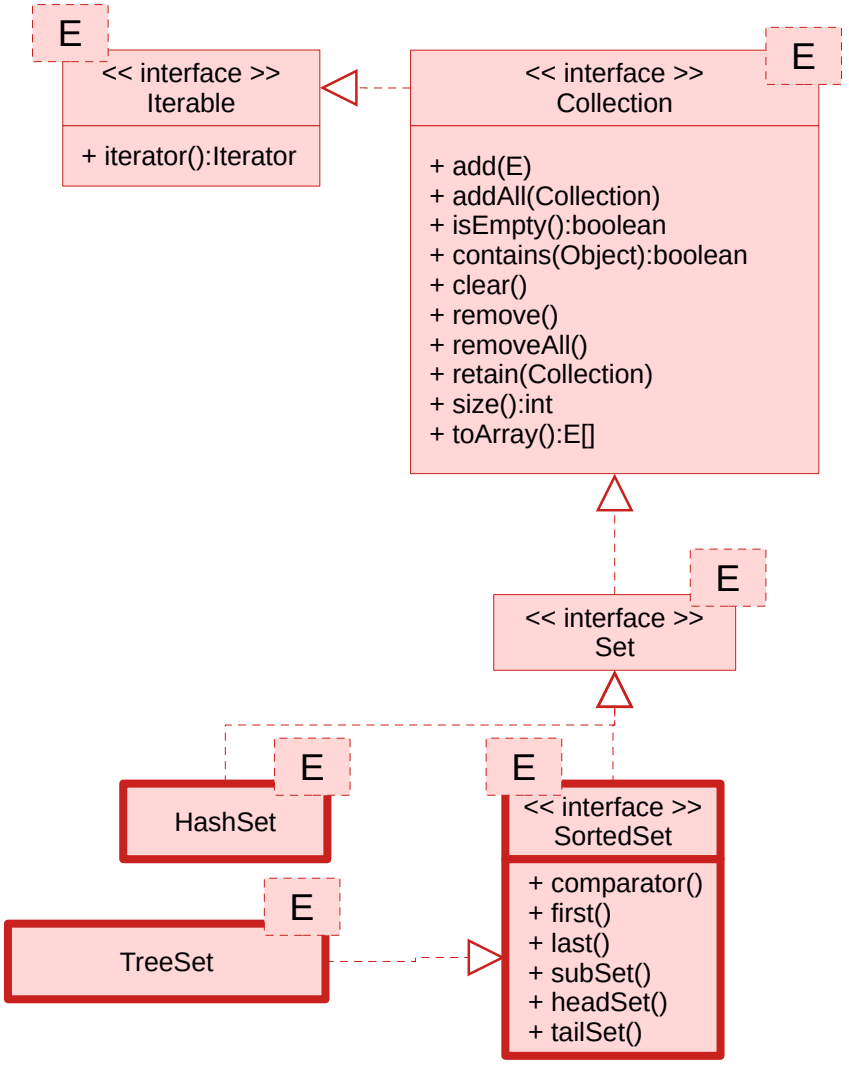
Los conjuntos

La otra interfaz a estructura de datos lineal importante que se encuentra en el paquete util es Set. En esta sección se analiza dicha interfaz y las clases derivdas.

Derivados de Set

Entre los derivados de Set destacan:

- SortedSet.- Interfaz derivada de Set que solo admite elementos derivados de Comparable y añade algunos métodos.
- TreeSet.- Derivado instanciable de SortedSet en el que los elementos se almacenan ordenados en un árbol binario de búsqueda.
- HashSet.- Clase derivada de Set que usa una tabla hash para contener los elementos.



TreeSet

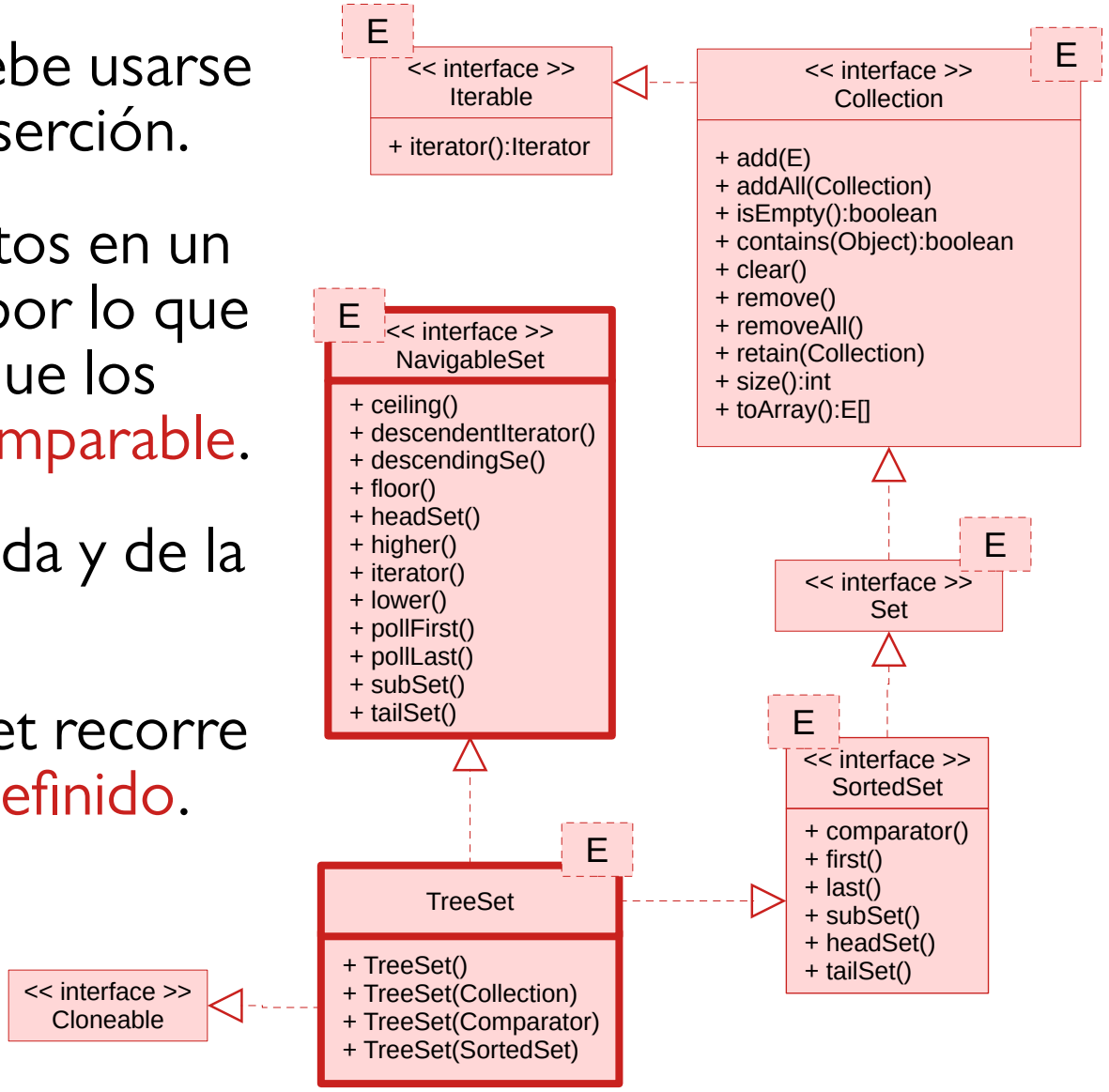
TreeSet es un conjunto, y debe usarse si no importa el orden de inserción.

TreeSet, ordena sus elementos en un árbol binario de búsqueda, por lo que necesita un Comparator o que los elementos implementen Comparable.

La complejidad de la búsqueda y de la inserción es $O(\log(n))$.

Un iterador sobre un TreeSet recorre sus elementos en el orden definido.

También, permite obtener subconjuntos por rangos.

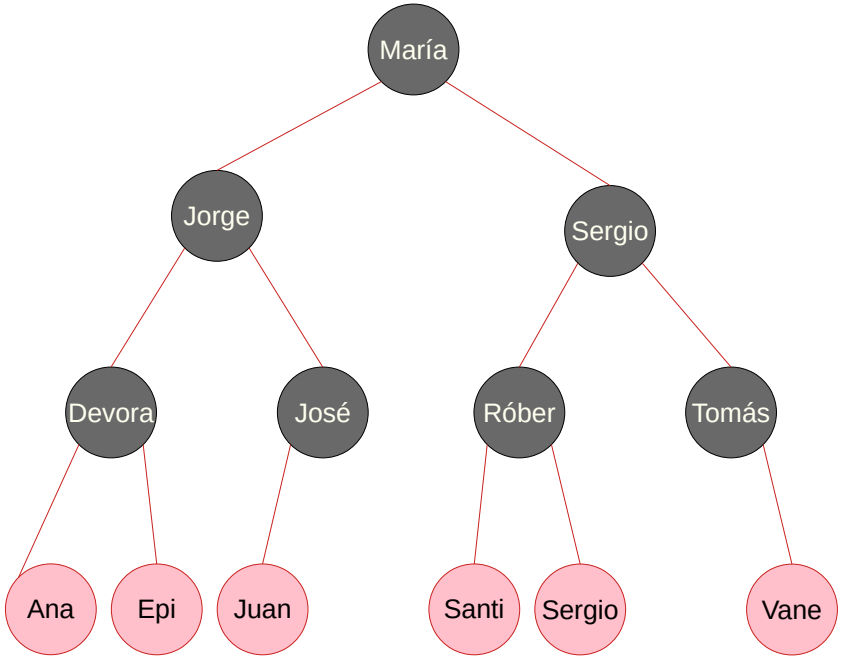


El interior de TreeSet

TreeSet utiliza internamente un árbol binario de búsqueda con reequilibrado Rojo-Negro para almacenar los valores.

Debido al uso del árbol binario de búsqueda se pueden localizar los elemento que contiene realizando menos operaciones.

Por ejemplo, si contiene 13 elementos, para saber si contiene a Pedro solo habría que realizar como mucho 4 comparaciones, mientras que usando una lista no ordenada habría que realizar13.



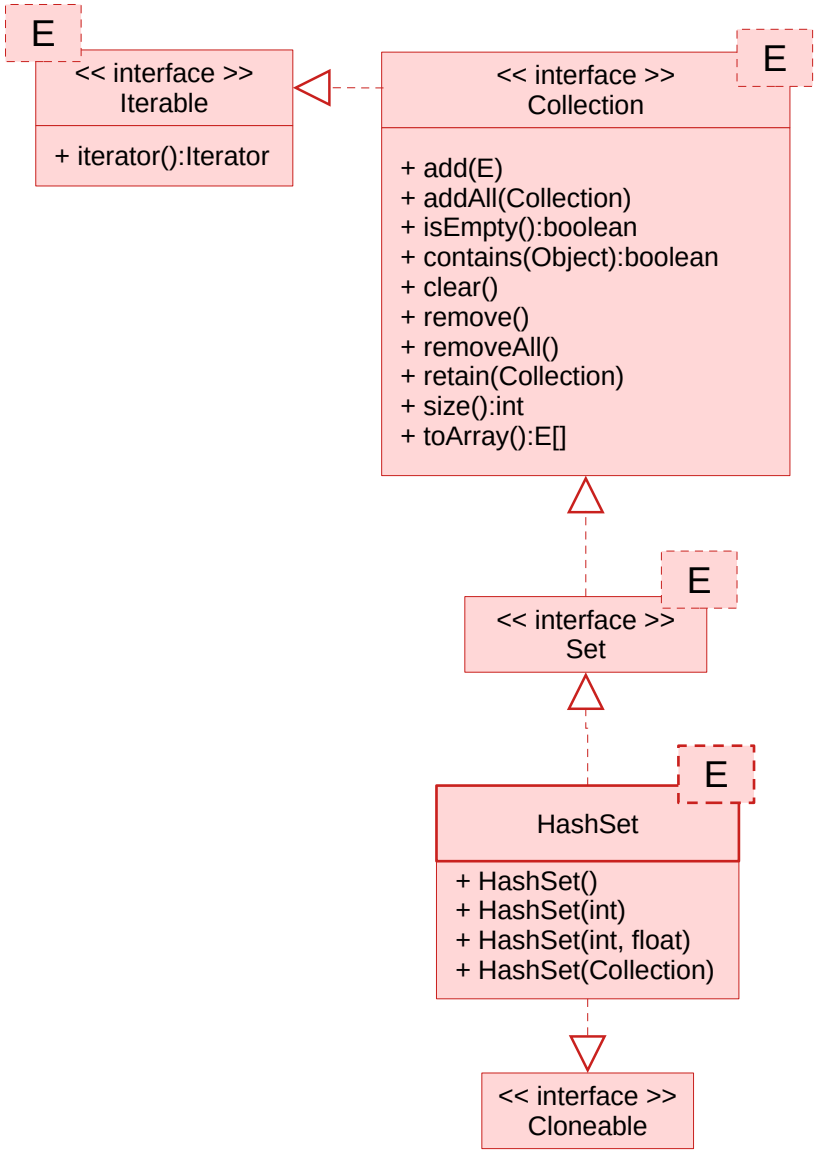
HashSet

HashSet es un conjunto, por tanto debe usarse cuando no importa el orden de inserción.

HashSet utiliza una tabla hash que usa como clave el valor devuelto por el método hashCode (de la clase Object o sobrecargado luego).

Las operaciones de búsqueda eliminación e inserción en media tienen complejidad $\Omega(1)$.

El HashSet se puede recorrer, pero el orden es aleatorio.

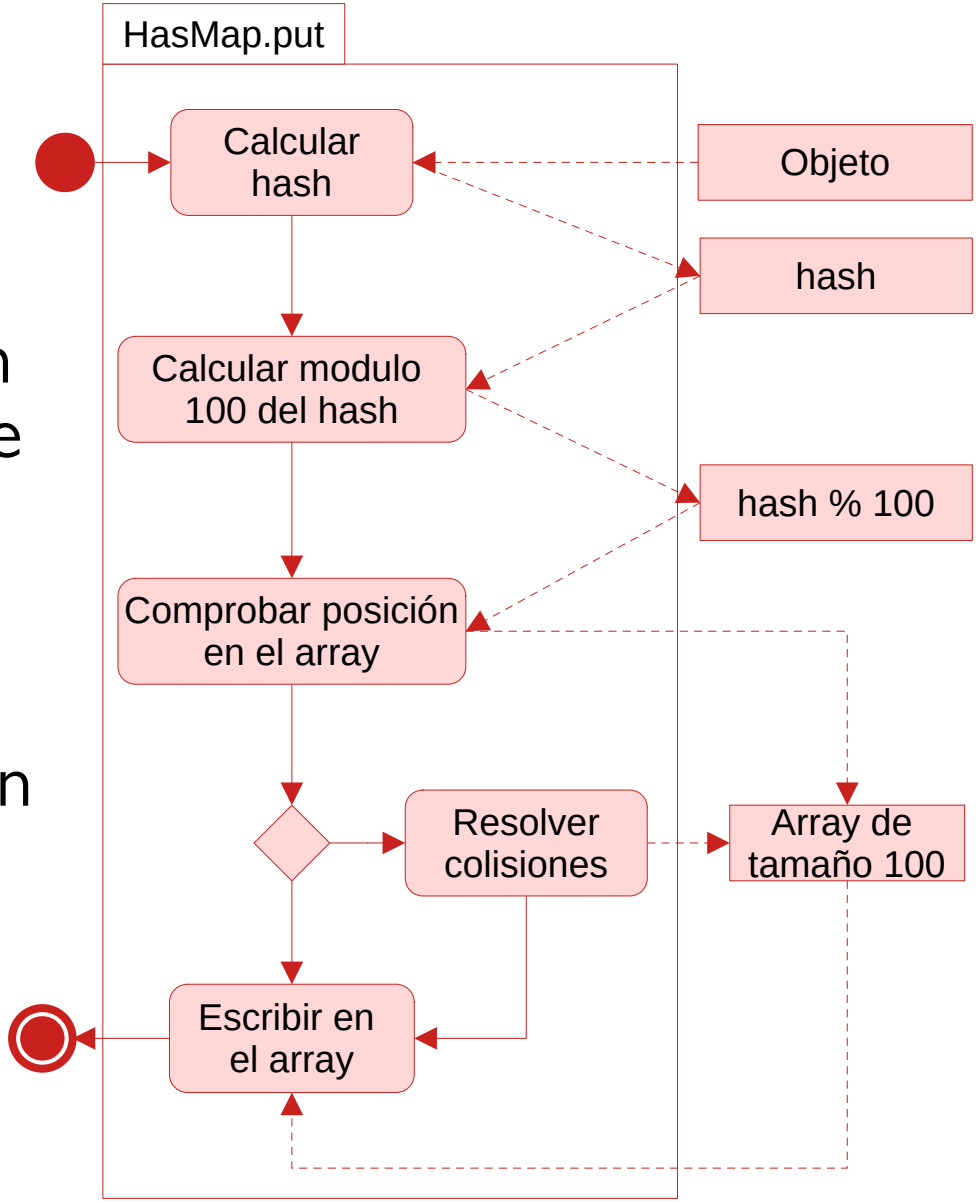


El interior de HashSet

HashSet utiliza internamente el código hash de los objetos para indexarlos en un array.

Debido a que internamente usa un array, en media, las operaciones de consulta e inserción de un valor aleatorio se resuelven en tiempo constante.

Algunas veces, cuando se producen colisiones este tiempo se amplia ligeramente. Por eso en el peor caso el tiempo de acceso e inserción se considera lineal.



Ejemplo de uso de los conjuntos


```
import java.util.Set;
import java.util.TreeSet;


public class EjemploUsoSet {

  public static void main(String[] args) {
    Set <String> listin = new TreeSet<>();
    listin.add("Pedro");
    listin.add("Ana");
    listin.add("Juan");
    listin.add("Ana");

    for (String nombre : listin) {
      System.out.println(nombre);
    }
  }
}
```

```
c:\Tmp> javac EjemploUsoList.java
c:\Tmp> java EjemploUsoList
Ana
Juan
Pedro
```

Los mapas

En esta sección se explora la estructura de datos Map de Java.

Mapa, Diccionario o array asociativo

Los mapas son unas estructuras de datos que permiten almacenar claves y asociar un valor a cada una de dichas claves.

Ejemplo de la agenda de personas:

```
agenda["maria"] = new Contacto("María Vela", 652323871, "c/Galia", "Fuenlabrada");
agenda["juan"]  = new Contacto("Juan Pérez", 916891213, "c/César", "Alcorcón");
agenda["pedro"] = new Contacto("Pedro Gómez", 642323872, "c/Asuranceturix", "Móstoles");
...
```

Ejemplo del diccionario de palabras:

```
diccionario["coche"] ="Objeto con habitaculo provisto de ruedas motorizadas que permite desplazarse";
diccionario["patata"]="Planta hervacea de la famila de las solanaceas de bulbo carnoso y comestible";
diccionario["lápiz"] ="Útil de escritura basado en un núcleo de carbono que deja señal al rozarse";
...
```

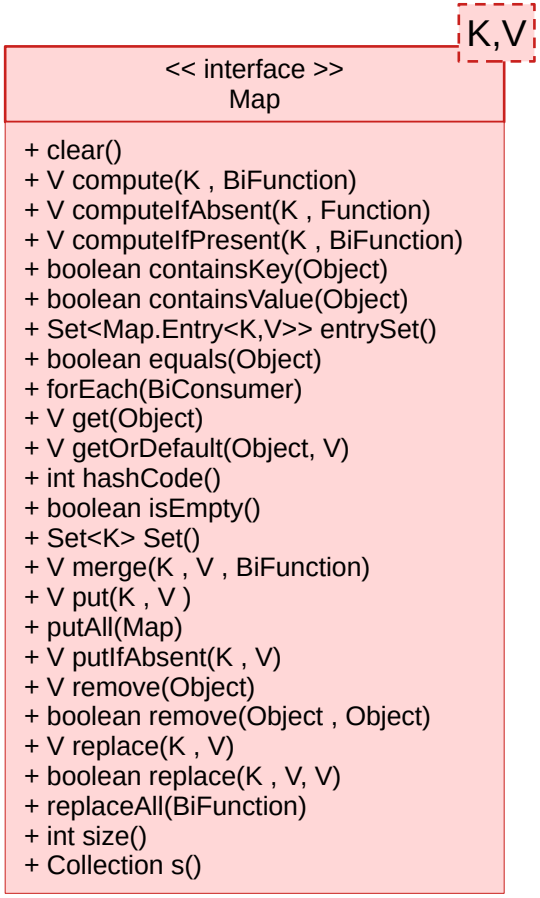
Los mapas también se llaman arrays asociativos o diccionarios.

Interfaz de Map

Los dos métodos principales de la interfaz Map son get y put.

Además, Map tiene métodos para:

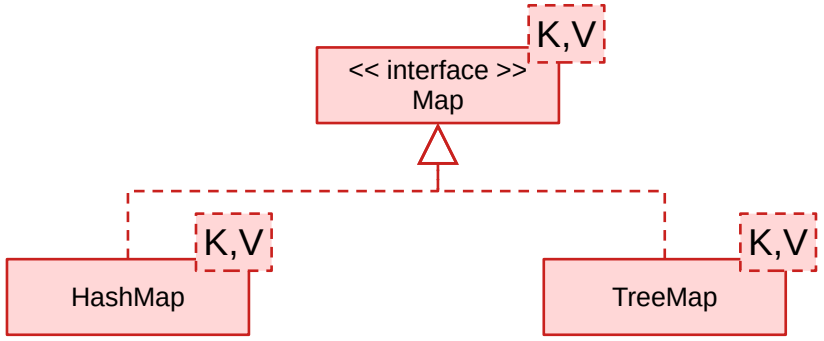
- Obtener el conjunto de claves o valores.
- Comprobar el tamaño
- Buscar valores y claves.
- Eliminar claves
- Reemplazar...



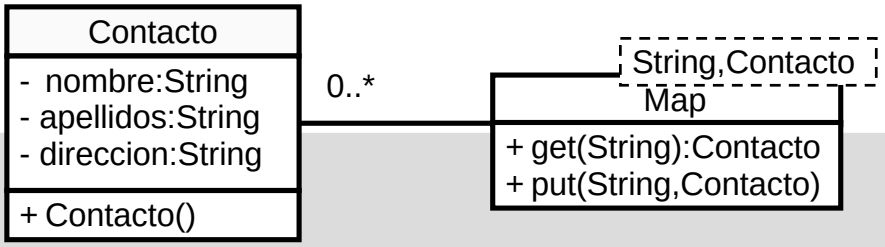
HashMap y TreeMap

Las dos soluciones para construir mapas en Java se basan en usar un HashSet o un TreeSet para las claves.

- HashMap es un mapa que usa un HashSet para las claves.
- TreeMap es un mapa que usa un TreeSet para las claves.



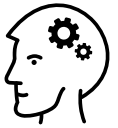
Ejemplo de uso de mapas




```
import java.util.Map;
import java.util.HashMap;

class Contacto {
  private String nombre;
  private String apellidos;
  private String direccion;
  public Contacto(String nombre, String apellidos, String direccion) {
    this.nombre    = nombre;
    this.apellidos = apellidos;
    this.direccion = direccion;
  }
}

public class EjemploUsoMap {

  public static void main(String[] args) {
    Map <String,Contacto> agenda = new HashMap<>();
    agenda.put("916511212", new Contacto("Juan", "Pérez", "c/Asuranceturix 3");
    agenda.put("660047855", new Contacto("Ana", "Gómez", "c/Galias 5");
    Contacto c = agenda.get("660047855");
  }
}
```

Los mapas

LinkedHashMap

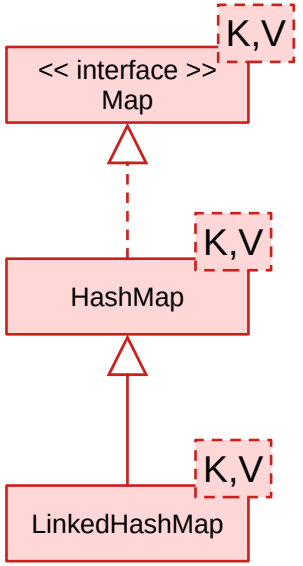


Los mapas almacenan un valor asociado a cada clave.

Los multimapas permiten almacenar varios valores asociados a cada clave.

Se puede construir fácilmente un multimapa insertando una lista en cada valor del mapa, aunque Java introduce la clase LinkedHashMap para facilitar esa tarea.

Algunos autores distinguen a los diccionarios de los mapas diciendo que los diccionarios son multimapas.

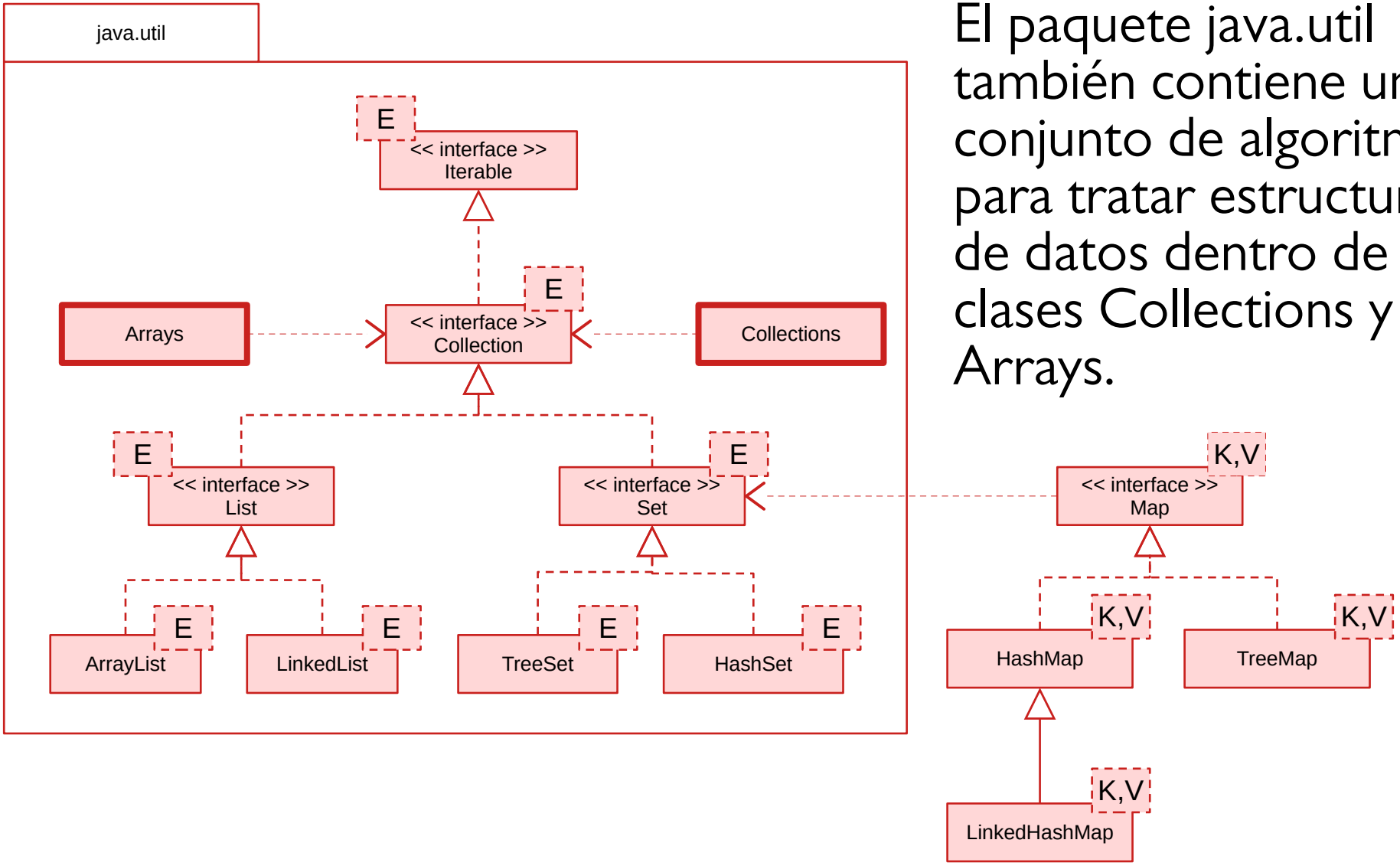


El paquete util

En esta sección se exploran otros elementos del paquete java.util relacionadas con las Collection.

El paquete java.util

Resumen de las estructuras vistas



El paquete java.util también contiene un conjunto de algoritmos para tratar estructuras de datos dentro de las clases Collections y Arrays.

Los algoritmos de la clase Collections

La clase Collections permite realizar multitud de operaciones con objetos de tipo Collection y con objetos de tipo Map como:

- Ordenar, barajar, rotar e invertir
- Rellenar de un valor
- Copiar
- Convertir a solo lectura
- Buscar y obtener máximos y mínimos
- Calcular la frecuencia...

Collections
+ addAll(Collection, E...) + asLifoQueue(Deque) + binarySearch(List,E) + checkedCollection(Collection, Class):Collection + checkedMap(Map,Class):Map + checkedSet(Set,Class):Set + checkedSortedSet(Set,Class):SortedSet + copy(List,List) + disjoint(Collection, Collection) + emptyList():List + emptyMap():Map + emptySet():Set + enumeration(Collection) + fill(List, E) + frequency(Collection) + max(Collection) + min(Collection) + newSetFromMap(Map):Set + nCopies(int, E):List + reverse(List) + rotate(List) + shuffle(List) + singleton(E):Set + singletonList(List):List + singletonMap(k,V):Map + sort(list, Comparator) + sort(list) + swap(List,int,int) + synchronizedList(List):List + synchronizedSet(Set):Set + synchronizedSet(SortedSet):SortedSet + synchronizedMap(Map):Map + unmodifiableList(List):List + unmodifiableMap(Map):Map + unmodifiableSet(Set):Set + unmodifiableSortedSet(SortedSet):SortedSet

Los algoritmos de la clase Arrays

La clase Arrays tiene un metodo que permite crear ArrayList directamente desde valores. Además, Arrays tiene métodos para realizar operaciones sobre arrays:

- Ordenar
- Rellenar de un valor
- Copiar
- Buscar
- Comparación

Arrays
+ asList(T...):List + binarySearch(int[]):int ... + binarySearch(Object[]):int + copy(int[]) ... + copy(Object[]) + copyRange(int[],int,int) ... + copyRange(Object[],int,int) + deepEquals(Object[],Object[]):boolean + equals(int[],int[]):boolean ... + equals(Object[],Object[]):boolean + fill(int[]) ... + fill(Object[]) + sort(int[]) ... + sort(Object[])

Uso de algoritmos sobre Collection

El siguiente ejemplo muestra como crear un List a partir de un conjunto de valores, como obtener el máximo, el mínimo y como invertir los elementos de la lista.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class EjemploUsoCollections {

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,1,2,3,5,8,13,21);
        int max = Collections.max(l);
        int min = Collections.min(l);
        Collections.reverse(l);
        System.out.println(l);
    }
}
```

Referencias

Para saber más de las estructuras de datos del JDK:

- The Java Collections Framework (web)

Además de las estructuras de datos del JDK hay otras:

- Google GUAVA library (web)
- Apache Commons Collections (web)