Tema I.1 Introducción

angel.sanchez@urjc.es
jose.velez@urjc.es
mariateresa.gonzalezdelena@urjc.es

abraham.duarte@urjc.es
raul.cabido@urjc.es

Descripción de la asignatura

- **Tema 1: Introducción** — Bloque 1
- Tema 2: Árboles generales — Bloque 2
- Tema 3: Mapas y diccionarios — Bloque 2
- Tema 4: Mapas y Diccionarios ordenados — Bloque 2
- Tema 5: Grafos — Bloque 2
- Tema 6: EEDD en memoria secundaria — Bloque 3

Resumen

- Repaso de las ED lineales ya conocidas…
  - Invariantes de las ED, operaciones soportadas, complejidad de las mismas, posibles implementaciones…
- Implementación en Java
  - Genericidad
  - *Collections*

Objetivos

- Dado un problema, **identificar** la estructura de datos más adecuada
- **Uso** de las estructuras de datos disponibles en el ***framework Collections***

Tipo Abstracto de Datos

- Es un modelo matemático que permite definir un tipo de datos junto con las funciones que operan sobre él
- La mayoría de los lenguajes de programación actuales permiten implementar TADs mediante el uso de clases
- Las clases definen propiedades (representación interna del TAD) y métodos (operaciones soportadas por el TAD)
- Las instancias de un TAD son los objetos de una clase

Tipos de Estructuras de Datos

- Cualquier estructura que permite almacenar datos durante la ejecución de un programa

- Se puede hablar de:
  - ED dinámicas o estáticas
    - Su uso de memoria puede cambiar (o no) durante la ejecución de un programa
  - ED lineales o no lineales
    - Cada elemento tiene a lo sumo dos vecinos, uno que le precede y otro que le sigue. Únicamente es posible un tipo de recorrido (en dos sentidos).

- Operaciones sobre las estructuras de datos, dos tipos:
  - Consulta: search, minimum, maximum, sucessor, predecessor,…
  - Modificación: insert, delete, …

Estructuras de datos lineales

Pila

Colección de elementos homogéneos dispuestos en orden tal que se recuperan en orden inverso a como se introdujeron (política *LIFO*)

Dos posibles implementaciones:

- Mediante vectores: `ArrayStack`
- Mediante nodos enlazados: `LinkedStack`

Operaciones básicas:

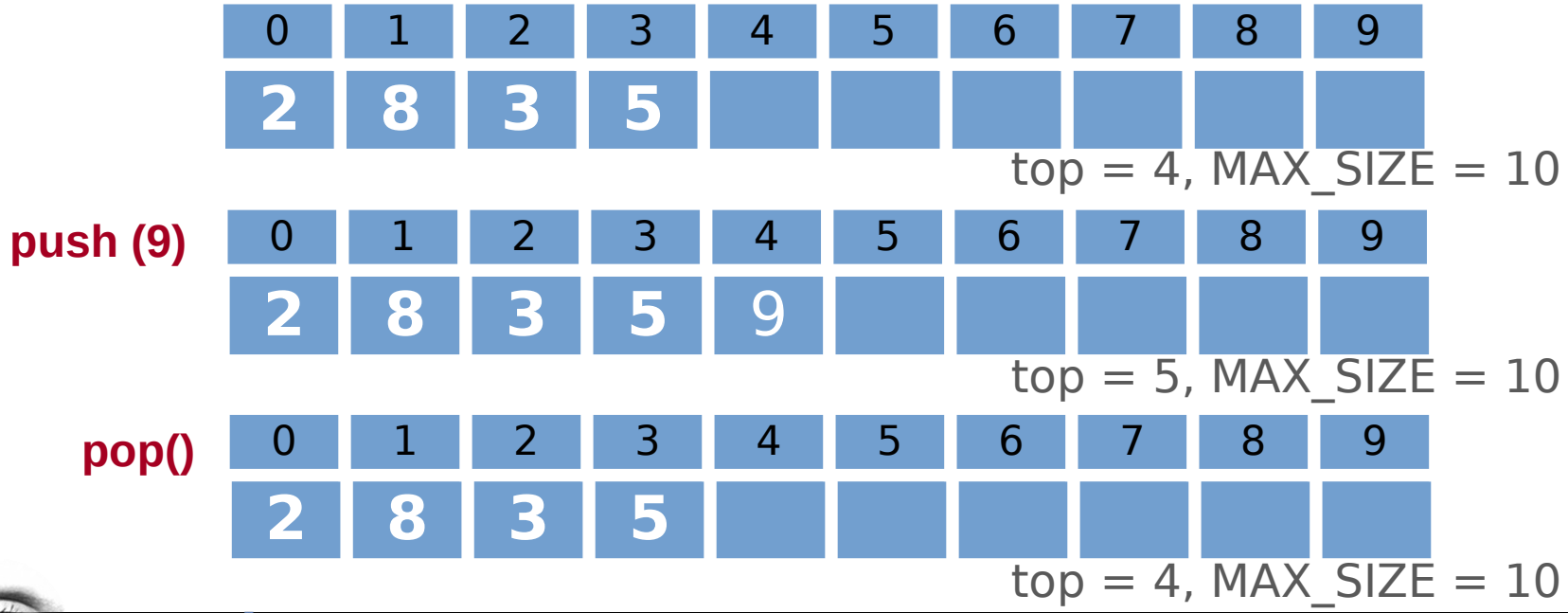
`isEmpty(), push() y pop()`

Pila – Ejemplo de uso

Operación	Salida	Stack
`create`	-	-
`push(5)`	-	(5)
`push(3)`	-	(3,5)
`pop()`	3	(5)
`push(7)`	-	(7,5)
`pop()`	7	(5)
`top()`	5	(5)
`pop()`	5	()
`pop()`	Error	()
`isEmpty()`	true	()

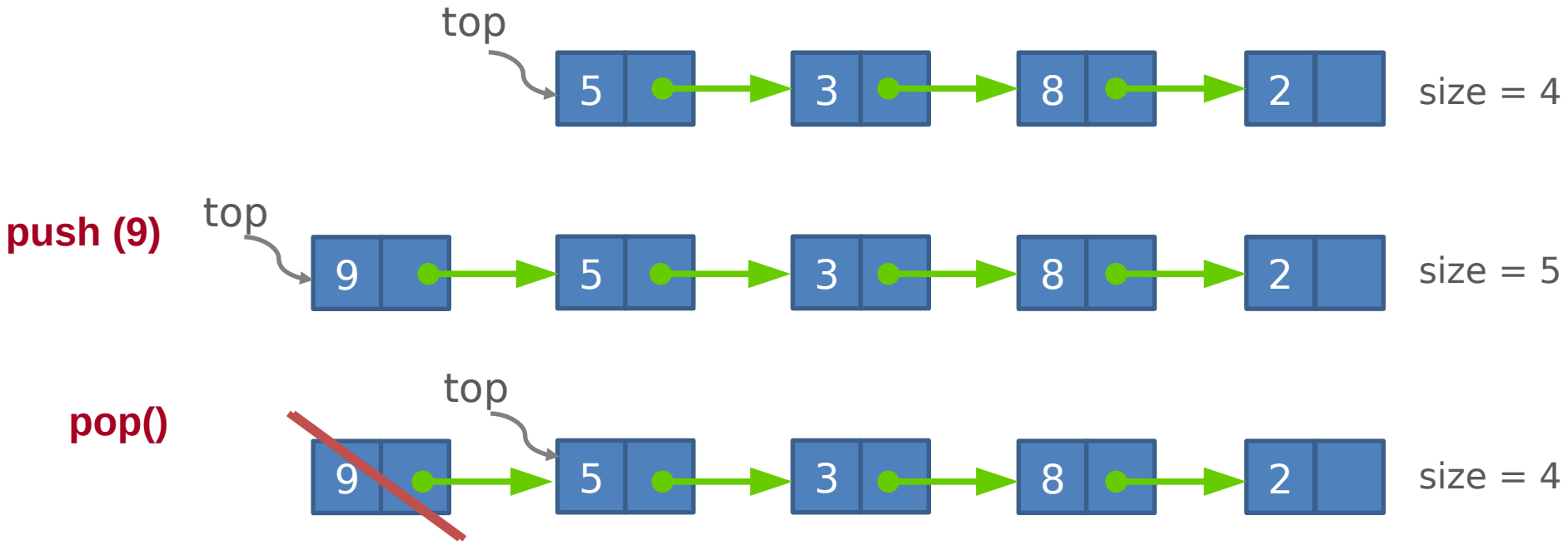
Pila – Implementación usando Array

Utilizando un array estático. Para lograr una ED dinámica, si el número de elementos alcanza el MAX_SIZE debe crearse un nuevo array de mayor capacidad y copiar los elementos del antiguo al nuevo.



Pila – Implementación usando Nodos

Utilizando una clase nodo para cada elemento de la pila. Cada nodo contiene un valor y un puntero al siguiente elemento de la pila. La ED precisa un puntero a la cabeza.



Pila – Complejidad

Operación	Complejidad en ArrayList	Complejidad en LinkedList
isEmpty()	O(1)	O(1)
push()	O(n)	O(1)
pop()	O(1)	O(1)

Cola

Colección de elementos homogéneos dispuestos en orden tal que se recuperan en igual orden a como se introdujeron (política *FIFO*)

Dos posibles implementaciones:

- Mediante vectores: `ArrayQueue`
- Mediante nodos: `QLinkedQueue`

Operaciones básicas:

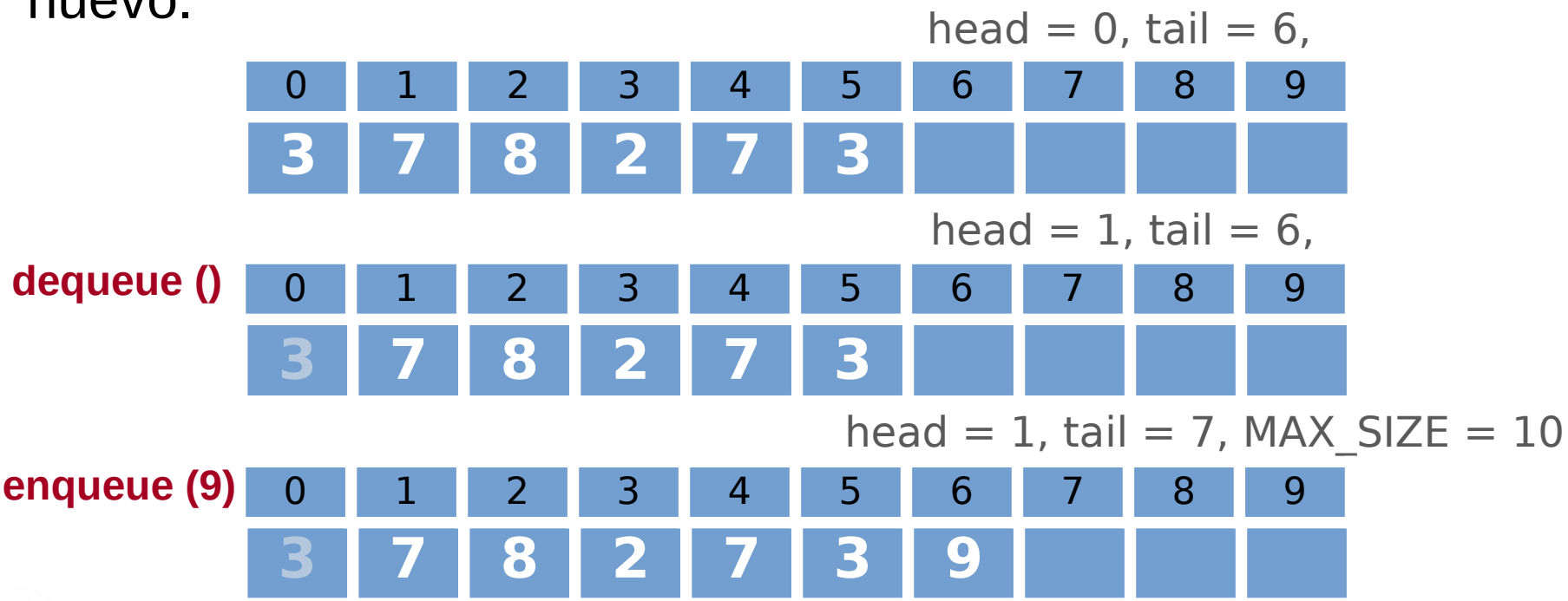
```
isEmpty(), front()
enqueue(E e), dequeue()
```

Cola – Ejemplo de uso

Operación	Salida	queue
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(3, 5)
`dequeue()`	5	(3)
`enqueue(7)`	-	(7,3)
`dequeue()`	3	(7)
`front()`	7	(7)
`dequeue()`	7	()

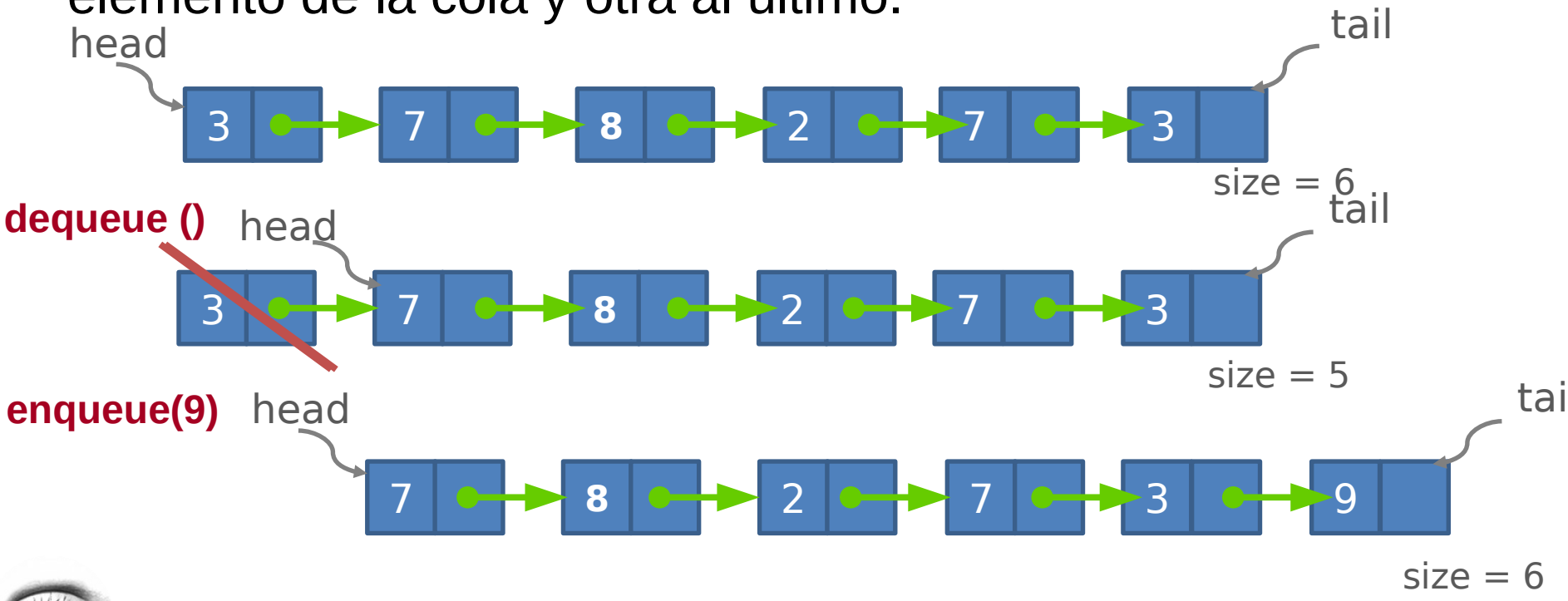
Cola – Implementación basada en array

Usando un array estático de forma circular. Para lograr una ED dinámica, si se alcanza MAX_SIZE debe crearse un nuevo array mayor y copiar los elementos del antiguo al nuevo.



Cola – Implementación basada en nodos

Utilizando una clase nodo para cada elemento de la pila. Cada nodo contiene un valor y un puntero al siguiente elemento de la cola. La ED contiene un puntero al primer elemento de la cola y otra al último.



Cola – Complejidad

Operación	Complejidad en ArrayQueue	Complejidad en LinkedQueue
isEmpty()	O(1)	O(1)
enqueue()	O(n)	O(1)
dequeue()	O(1)	O(1)

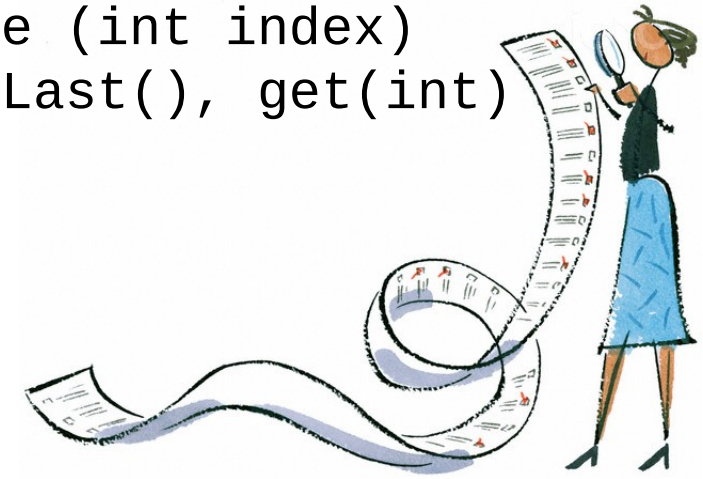
Lista

Colección de elementos homogéneos dispuestos en un orden.

Cualquier elemento tiene un predecesor (excepto el primero) y un sucesor (excepto el último).

Operaciones básicas:

```
addFirst(E e), addLast(E e), add(E e, int index)
removeFirst(), removeLast(), remove (int index)
isEmpty(), size(), getFirst(), getLast(), get(int)
```



Dos posibles implementaciones:

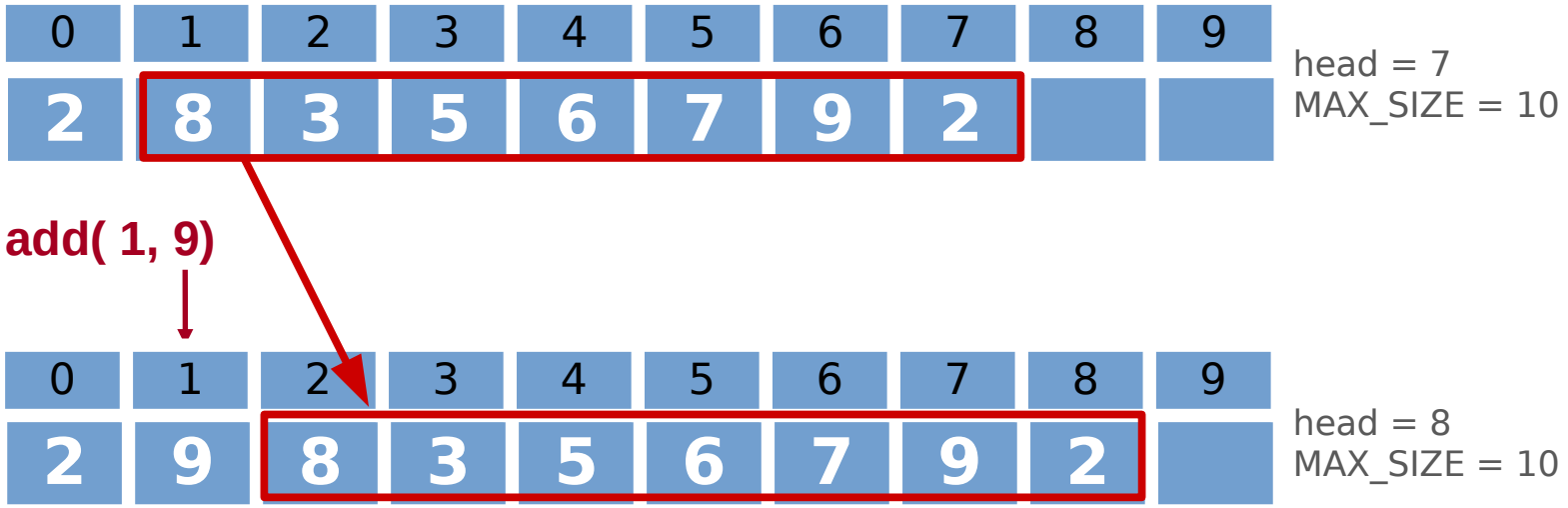
- Mediante vectores: `ArrayList`
- Mediante nodos enlazados: `LinkedList`

Lista – Ejemplo de uso

Operación	Salida	List
add(5)	-	(5)
add(3)	-	(5,3)
add(9)	-	(5,3,9)
addLast(7)	-	(5,3,9,7)
remove(3)	9	(5,3,7)
contains(10)	false	(5,3,7)
get(1)	5	(5,3,7)
getLast()	7	(5,3,7)

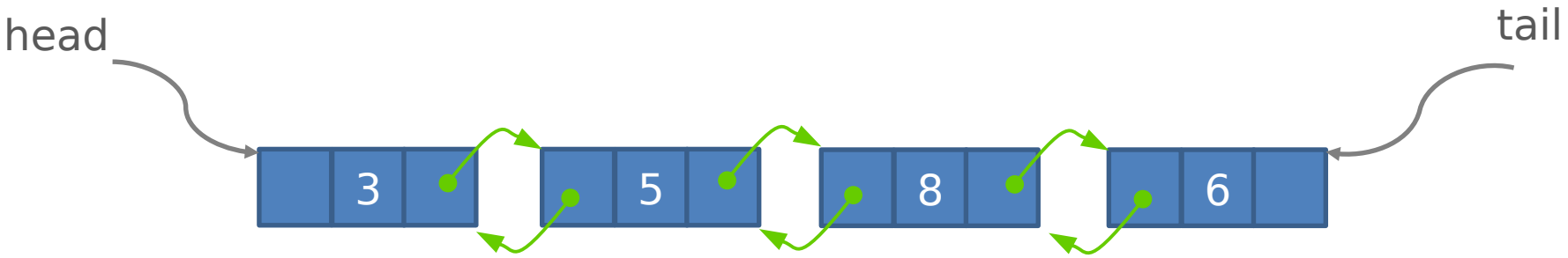
Lista – Implementación usando Array

Usando un array estático. Para lograr una ED dinámica, si se alcanza MAX_SIZE debe crearse un nuevo array mayor y copiar los elementos del antiguo al nuevo. La inserción puede implicar desplazamientos.



Lista – Implementación basada en nodos

Utilizando una clase nodo para cada elemento de la pila. Cada nodo contiene un valor y dos punteros: al siguiente elemento y al anterior. La ED contiene un puntero al primer elemento de la lista y otro al último.

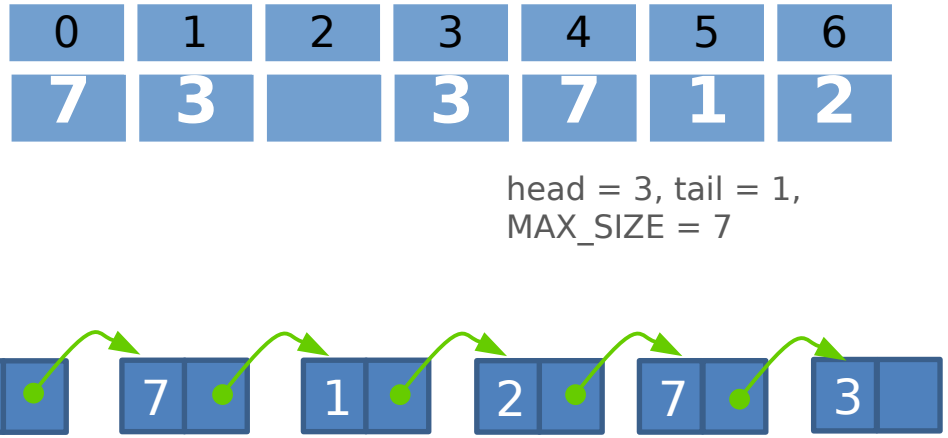


Lista – Complejidad

Operación	Complejidad en ArrayList	Complejidad en LinkedList
addFirst()	O(n)	O(1)
addLast()	O(n)	O(1)
add()	O(n)	O(1)
getFirst	O(1)	O(1)
getLast	O(1)	O(1)
get	O(1)	O(n)
removeFirst	O(1)	O(1)
removeLast	O(1)	O(1)
remove	O(n)	O(n)
size	O(1)	O(1)

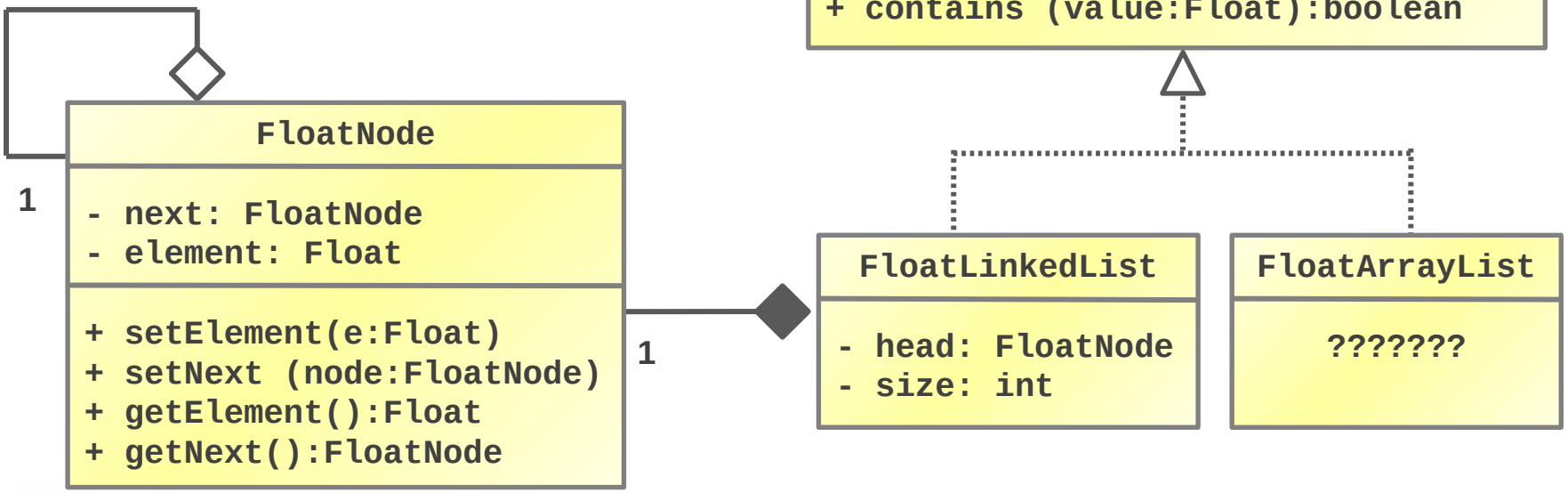
Implementación en Java: `list`

- Implementar en Java el TAD ListaFlotantes (*FloatList*)
  - Definir un conjunto de clases que permitan trabajar con los dos tipos de implementaciones vistas:
    - `FloatArrayList`
    - `FloatLinkedList`
  - Tipo de lista: simple
    - Inserción y borrado por cabecera
  - Operaciones:
    - `size()`, `isempty()`
    - `add()`, `remove()`
    - `get()`
    - `search()`, `contains()`



Implementación en Java: `list`

¿Cuáles son las **propiedades privadas** que permiten implementar el FloatArrayList?



Implementación en Java: `list`

¿Cómo podemos solucionar el problema de la **capacidad fija** del FloatArrayList?

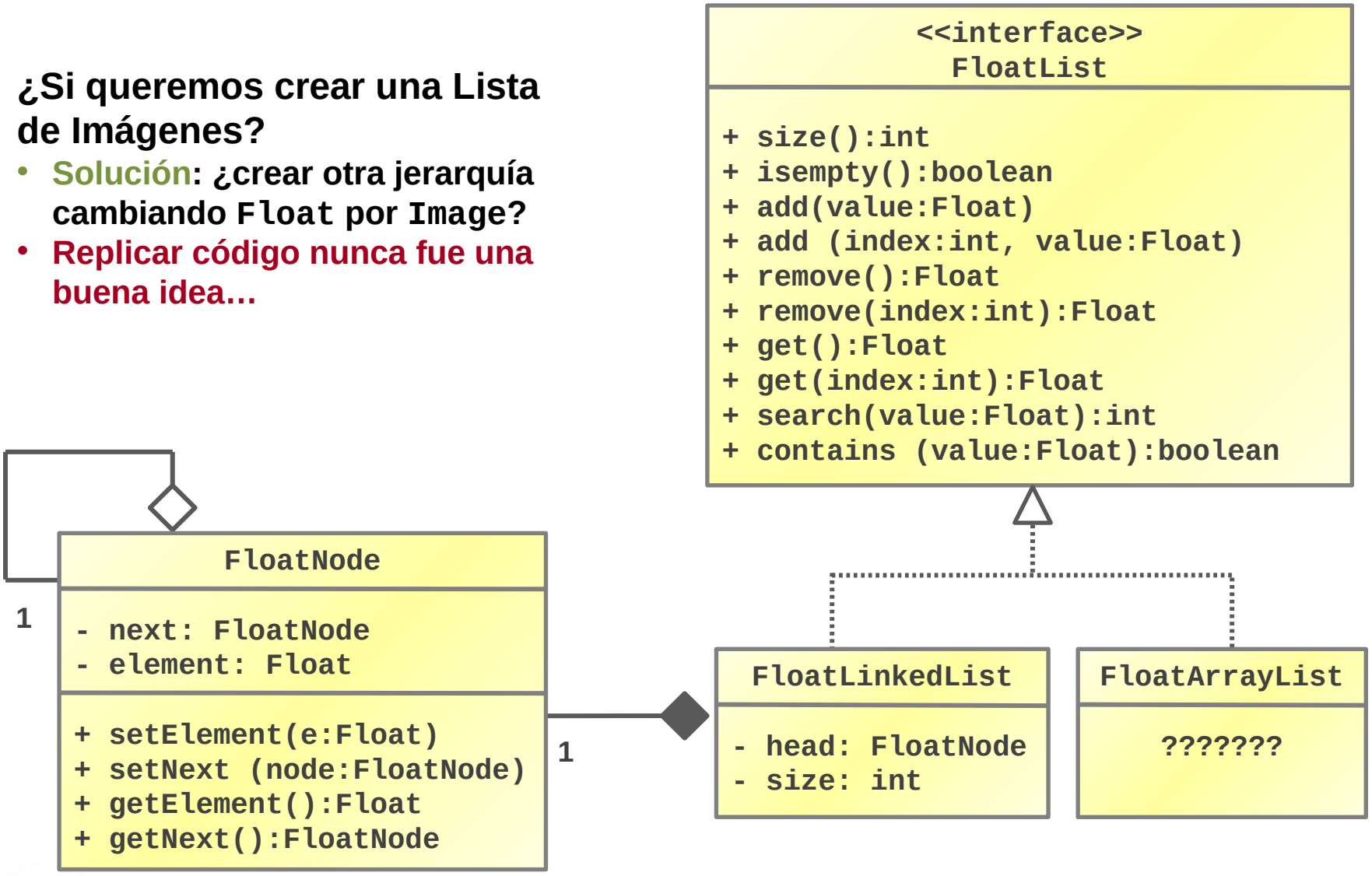
Implementación en Java: `list`

¿Hemos realizado algún tipo de **prueba** que nos permita garantizar que el **comportamiento** de nuestra ED es el **esperado**?

Implementación en Java: `list`

¿Si queremos crear una Lista de Imágenes?

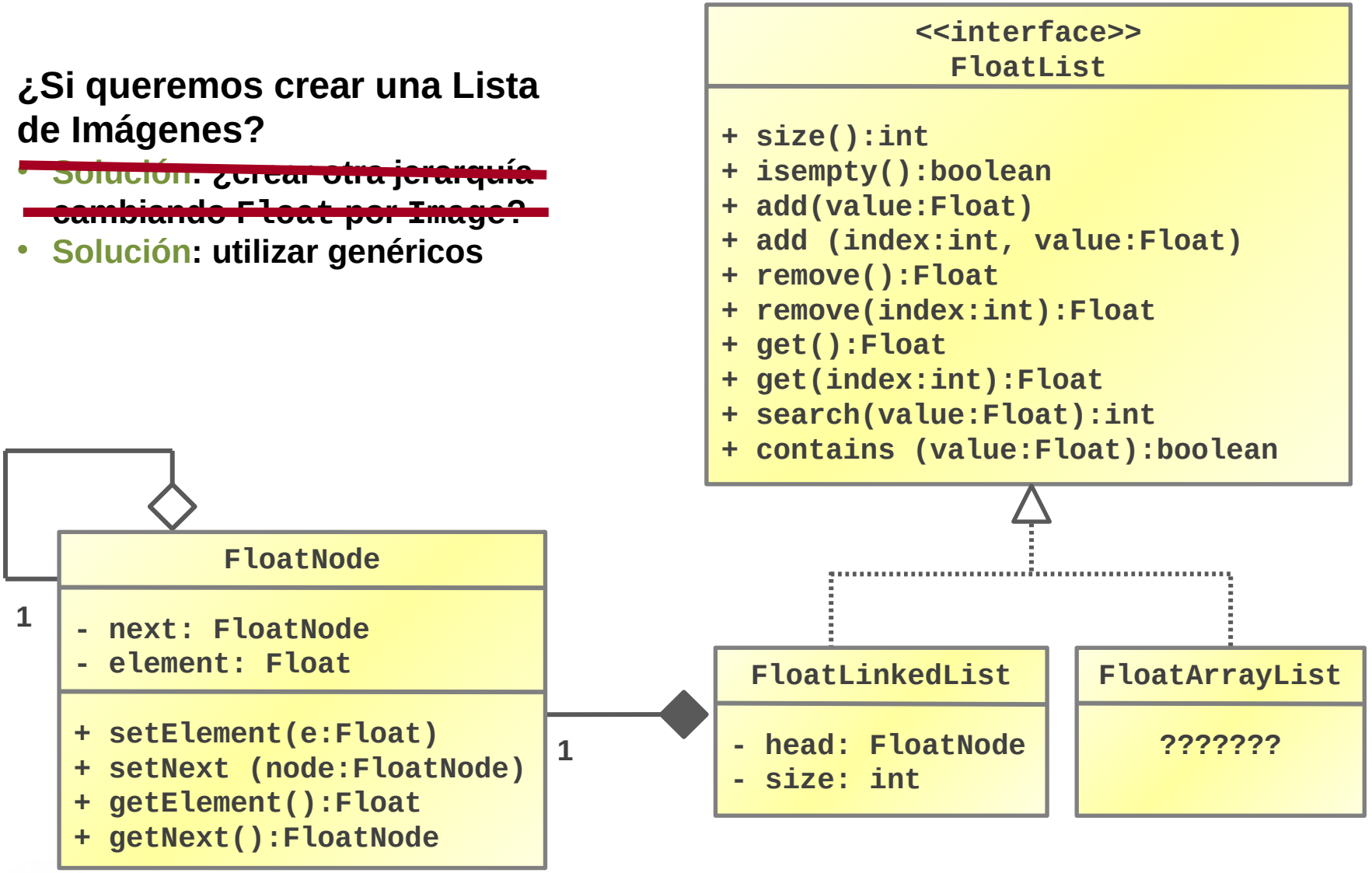
- **Solución: ¿crear otra jerarquía cambiando `Float` por `Image`?**
- **Replicar código nunca fue una buena idea…**



Implementación en Java: `list`

¿Si queremos crear una Lista de Imágenes?

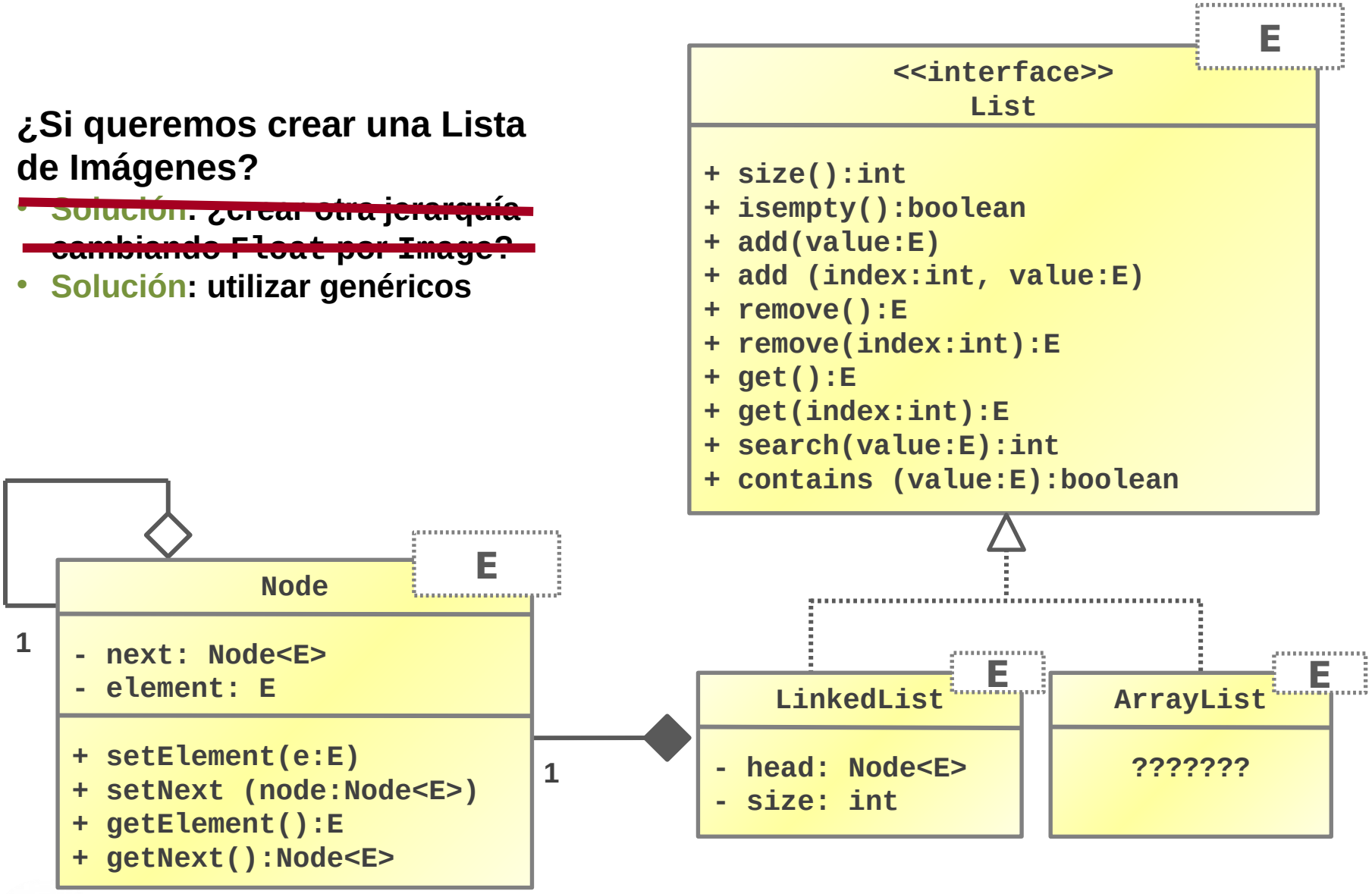
- ~~Solución: ¿crear otra jerarquía cambiando `Float` por `Image`?~~
- Solución: utilizar genéricos



Implementación en Java: `list`

¿Si queremos crear una Lista de Imágenes?

- ~~Solución: ¿crear otra jerarquía cambiando `Float` por `Image`?~~
- Solución: utilizar genéricos



Genéricos

Genéricos (*Generics*)

- Los genéricos son un mecanismo utilizado en los lenguajes de programación con tipado estático para especificar el tipo de los elementos de una estructura de datos
- En C++ se les denomina plantillas (*templates*)
- Aparte de las estructuras de datos, también puede utilizarse en otros contextos
- Los genéricos se incorporaron en Java 5

Genéricos (*Generics*)

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- Clase pila de fracciones implementada con un array

```
class PilaFracciones {

   private Fraccion[] elementos;
   private int numElementos;

   public PilaFracciones(int tope) {
      this.elementos = new Fraccion[tope];
      this.setNumElementos(0);
   }

   public Fraccion getElemento(int indice) {
      return this.elementos[indice];
   }

   public void addElemento(Fraccion fraccion) {
      if(numElementos < elementos.length){
         this.elementos[numElementos] = fraccion;
         numElementos++;
      }
   }
   ...
}
```

- Uso de la clase pila de fracciones

```
public static void main(String[] args) {

   PilaFracciones pila = new PilaFracciones(5);

   Fraccion fraccion1 = new Fraccion(1,2);
   Fraccion fraccion2 = new Fraccion(1,3);
   Fraccion fraccion3 = new Fraccion(1,4);

   pila.addElemento(fraccion1);
   pila.addElemento(fraccion2);
   pila.addElemento(fraccion3);

   ...
}
```

Genéricos (*Generics*)

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- ¿Si queremos implementar pila de **Intervalos**?
- ¿Creamos **otra** clase cambiando `Fracción` por `Intervalo`?
- Duplicar código es malo

```
class PilaIntervalos {

   private Intervalo[] elementos;
   private int numElementos;

   public PilaIntervalos(int tope) {
      this.elementos = new Intervalo[tope];
      this.setNumElementos(0);
   }

   public Intervalo getElemento(int indice) {
      return this.elementos[indice];
   }

   public void addElemento(Intervalo intervalo) {
      if(numElementos < elementos.length){
         this.elementos[numElementos] = intervalo;
         numElementos++;
      }
   }
   ...
}
```

- Uso de la clase pila de intervalos

```
public static void main(String[] args) {

   PilaIntervalos pila = new PilaIntervalos(5);

   Intervalo intervalo1 = new Intervalo(1,2);
   Intervalo intervalo2 = new Intervalo(1,3);
   Intervalo intervalo3 = new Intervalo(1,4);

   pila.addElemento(intervalo1);
   pila.addElemento(intervalo2);
   pila.addElemento(intervalo3);

   ...
}
```

Genéricos (*Generics*)

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- ¿Si queremos implementar pila de Intervalos?
- Podemos usar **polimorfismo**
- ¿Hacemos que los elementos sean de tipo **`Object`**?

```
class PilaObjects {

   private Object[] elementos;
   private int numElementos;

   public PilaObjects(int tope) {
      this.elementos = new Object[tope];
      this.setNumElementos(0);
   }

   public Object getElemento(int indice) {
      return this.elementos[indice];
   }

   public void addElemento(Object object) {
      if(numElementos < elementos.length){
         this.elementos[numElementos] = object;
         numElementos++;
      }
   }
   ...
}
```

- Usar Object funciona
- Pero el compilador no nos ayuda
- Se puede escribir código incorrecto

```
//Queremos que la pila contenga Intervalos
PilaObjects pila = new PilaObjects(5);

pila.addElemento(new Intervalo(2,4));
pila.addElemento(new Intervalo(2,6));
…
//Nos equivocamos y el compilador no da error
//No se genera una excepción en ejecución
pila.addElemento(new Fraccion(1,2));


…
//Siempre que sacamos intervalos tenemos
//que hacer cast. Puede generar una excepción
Intervalo intervalo = (Intervalo) pila.getElemento();
```

- Lo ideal sería definir el tipo de los elementos cuando se usa la pila, no cuando se implementa la pila

```
public static void main(String[] args) {

   PilaIntervalo pilaIntervalo;
   PilaFraccion  pilaFraccion;

   ...
}
```

- A partir de Java 5 se puede implementar la pila con elementos genéricos, sin definir su tipo

- El tipo se define cuando se usa la pila
  - Al declarar una variable
  - Al instanciar un objeto pila

Genéricos (*Generics*)

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

```
class PilaIntervalos {

  public PilaIntervalos(int tope) { ... }

  public Intervalo getElemento(int indice) { ... }

  public void addElemento(Intervalo intervalo) { ... }
}
```

```
class Pila<E> {

  public Pila(int tope) { ... }

  public E getElemento(int indice) { ... }

  public void addElemento(E elem) { ... }
  ...
}
```

Genéricos (*Generics*)

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

Sin usar genéricos

```
//Queremos que la pila contenga Intervalos
PilaIntervalos pilaIntervalos = new PilaIntervalos(5);

Intervalo i1 = new Intervalo(2,4);
Intervalo i2 = new Intervalo(2,6);

pilaIntervalos.addElemento(i1);
pilaIntervalos.addElemento(i2);
…
Intervalo i3 = pilaIntervalos.getElemento(1);
```

Usando genéricos

```
//Queremos que la pila contenga Intervalos
Pila<Intervalo> pilaIntervalos = new Pila<Intervalo>(5);

Intervalo i1 = new Intervalo(2,4);
Intervalo i2 = new Intervalo(2,6);

pilaIntervalos.addElemento(i1);
pilaIntervalos.addElemento(i2);
…
Intervalo i3 = pilaIntervalos.getElemento(1);
```

Genéricos (*Generics*)

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

Sin usar genéricos

```
PilaIntervalos pilaIntervalos = new PilaIntervalos(5);
PilaFracciones pilaFracciones = new PilaFracciones(5);

…

Intervalo i3 = pilaIntervalos.getElemento(1);
Fraccion fraccion = pilaFracciones.getElemento(1);
```

Usando genéricos

```
Pila<Intervalo> pilaIntervalos = new Pila<Intervalo>(5);
Pila<Fraccion> pilaFracciones = new Pila<Fraccion>(5);

…

Intervalo intervalo = pilaIntervalos.getElemento(1);
Fraccion fraccion = pilaFracciones.getElemento(1);
```

- En Java 7 se ha simplificado la creación de objetos genéricos

Antes de Java 7

```
Pila<Intervalo> pilaIntervalos = new Pila<Intervalo>(5);
Pila<Fraccion> pilaFracciones = new Pila<Fraccion>(5);
```

En Java 7

```
Pila<Intervalo> pilaIntervalos = new Pila<>(5);
Pila<Fraccion> pilaFracciones = new Pila<>(5);
```

- El tipo de los elementos se infiere de la declaración de la variable y se puede omitir de la construcción del objeto

Iteradores

Iteradores

- Formas de recorrer una colección de elementos
  - Usando un bucle `for` con acceso por posición
  - Usando iteradores
  - Usando el `for` mejorado

Iteradores

- Usando un bucle `for` con acceso por posición

```
List<String> ciudades = new ArrayList<String>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

for (int i=0; i < ciudades.size(); i++) {
   String ciudad = ciudades.get(i);
   System.out.println(ciudad + "\n");
}
```

  - No se recomienda, especialmente en ED donde el acceso a los elementos no es directo
    - Ejemplo: `LinkedList<String>`
      - **Recorrido: O(n^2)**

Iteradores

- Usando…¿iteradores?
- Iterador
  - Patrón de diseño software que abstrae el recorrido de una colección elemento a elemento
  - Encapsula el concepto de “posición actual” y “elemento siguiente” dentro de una colección

Proporciona una forma general de acceder a los elementos de una colección (independientemente de su organización interna)

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove();
}
```

Iteradores

- Usando iteradores

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove();
}
```

```
List<String> ciudades = new ArrayList<String>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

Iterator<String> it = ciudades.iterator();
while (it.hasNext()){
    String s = it.next();
    System.out.println(s + "\n");
}
```

Proporciona una forma general de acceder a los elementos de una colección (independientemente de su organización interna)

Iteradores

- Usando iteradores

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove();
}
```

```
List<String> ciudades = new LinkedList<String>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

Iterator<String> it = ciudades.iterator();
while (it.hasNext()){
    String s = it.next();
    System.out.println(s + "\n");
}
```

Proporciona una forma general de acceder a los elementos de una colección (independientemente de su organización interna)

Iteradores

- Usando iteradores

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove();
}
```

```
HashSet<String> ciudades = new HashSet<String>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

Iterator<String> it = ciudades.iterator();
while (it.hasNext()){
    String s = it.next();
    System.out.println(s + "\n");
}
```

Proporciona una forma general de acceder a los elementos de una colección (independientemente de su organización interna)

Iteradores

- `for` mejorado de Java
  - Internamente usa un iterador

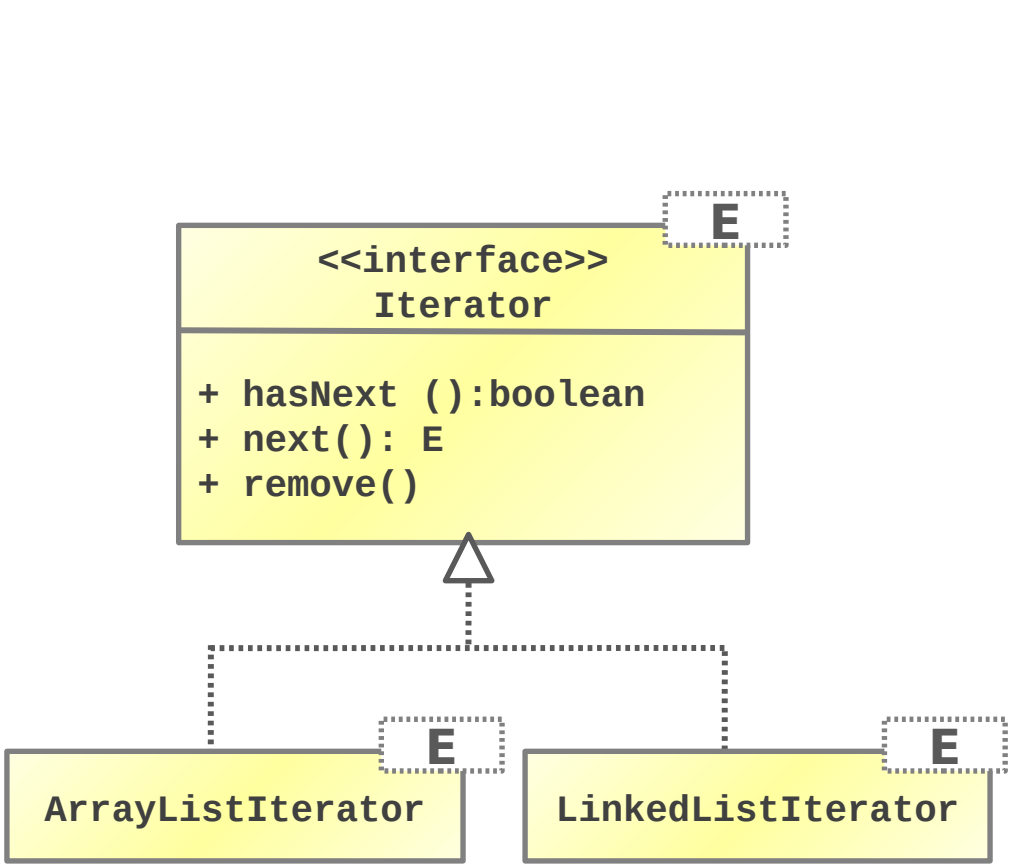
```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove();
}
```

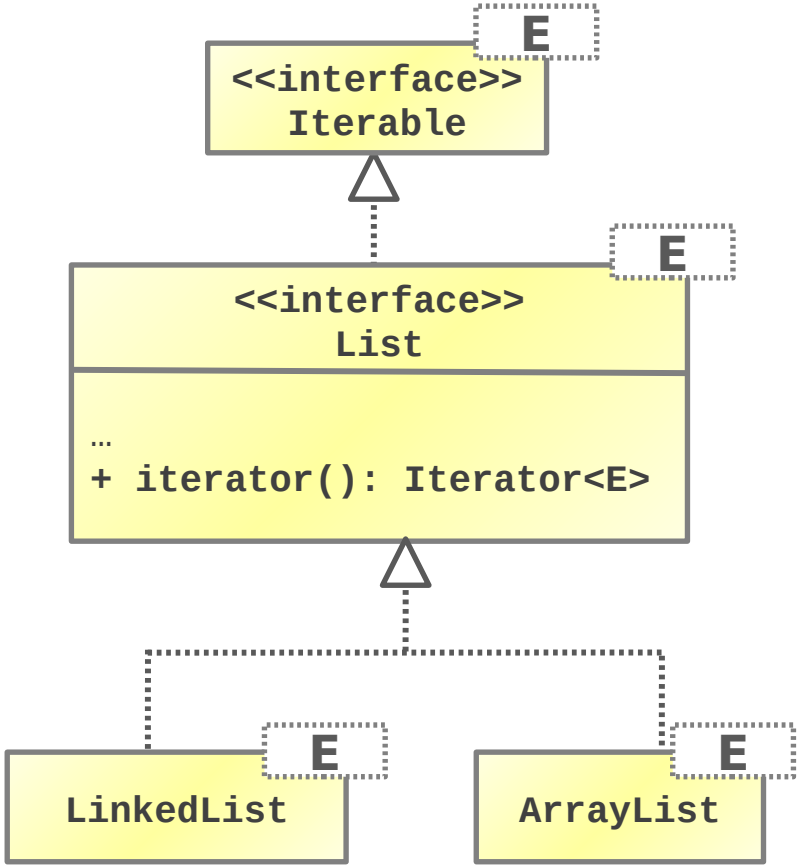
```
HashSet<String> ciudades = new HashSet<String>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

Iterator<String> it = ciudades.iterator();
while (it.hasNext()){
    String s = it.next();
    System.out.println(s + "\n");
}
```

```
for (String ciudad: ciudades) {
    System.out.println(ciudad + "\n");
}
```

Requisito: la colección debe implementar la interfaz Iterable

Implementación en Java: `list` (iteradores)





Consejo: Iteradores como nested classes

Interfaz position

Interfaz *position (LinkedPositionList)*

- Operaciones de actualización en una lista enlazada
  - Manera de indicar dónde se realiza = índice
    - `add(int index, E value)` // acceso secuencial
    - `remove (int index)` // acceso secuencial

O(n)

```
List<String> ciudades = new LinkedList<String>();
…

ciudades.add(i, "Madrid");
```

O(n)

```
List<String> ciudades = new LinkedList<String>();
…

ciudades.remove(i);
```

Interfaz *position (LinkedPositionList)*

- Operaciones de actualización en una lista enlazada
  - Manera de indicar dónde se realiza = índice
    - `add(int index, E value)` // acceso secuencial
    - `remove (int index)` // acceso secuencial

¿Alguna solución?

O(n)

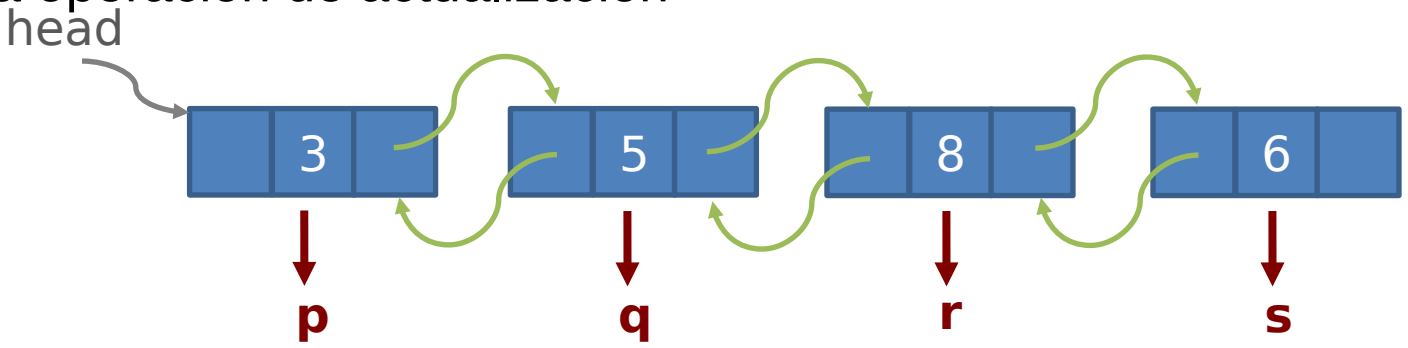
```
List<String> ciudades = new LinkedList<String>();
…

ciudades.add(i, "Madrid");
```

O(n)

```
List<String> ciudades = new LinkedList<String>();
…

ciudades.remove(i);
```

Interfaz *position (LinkedPositionList)*

- Operaciones de actualización en una lista enlazada
  - En una lista enlazada, parece más natural (y eficiente) utilizar referencias a nodo en lugar de índices para indicar donde realizar la operación de actualización



Índice O(n)	Referencia O(1)
`list.add(2, 123)`	`list.addBefore(r, 123)`
`list.add(2, 123)`	`list.addAfter(q, 123)`
`list.remove (2)`	`list.remove (r)`

Interfaz *position (LinkedPositionList)*

- Operaciones de actualización en una lista enlazada
  - En una lista enlazada, parece más natural (y eficiente) utilizar referencias a nodo en lugar de índices para indicar donde realizar la operación de actualización

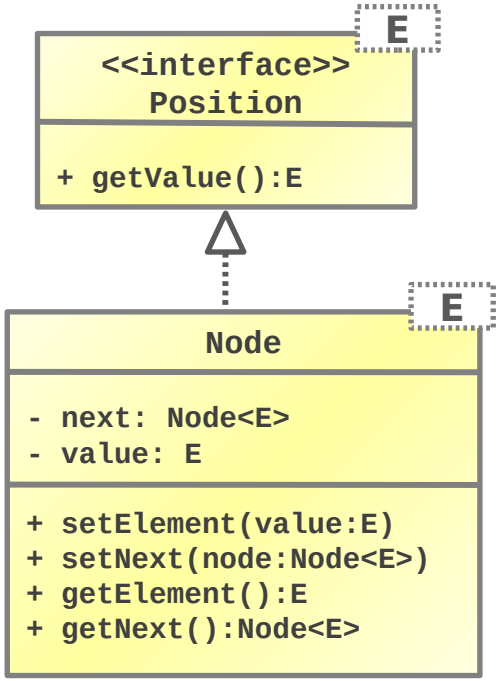
```
//void add(int index, E value)
Node<E> addBefore (Node<E> node, E value);
Node<E> addAfter  (Node<E> node, E value);

//E remove (int index)
E remove (Node<E> node)
```

  - Problema
    - Exponemos la representación interna de la lista al usuario (desencapsulación)
    - El usuario podría romper las invariantes de la ED
      - A través de las referencias a nodo podría modificar la estructura interna de la lista, sin nuestro conocimiento

Interfaz *position (LinkedPositionList)*

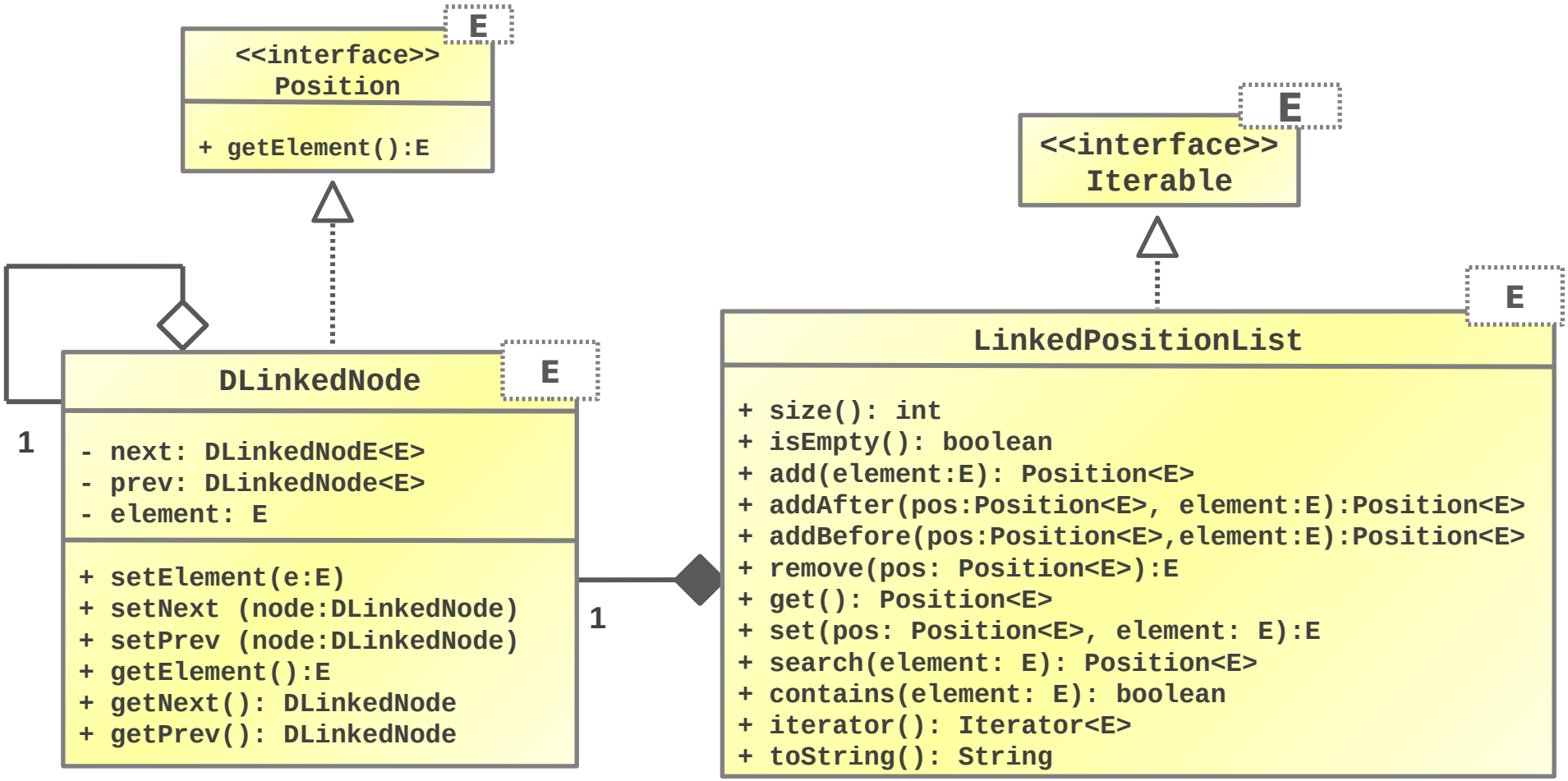
- Operaciones de actualización en una lista enlazada
  - Solución:
    - Devolver nodos en “modo lectura”: `Position`

```
//void add(int index, E value)
Node<E> addBefore (Node<E> node, E value);
Node<E> addAfter  (Node<E> node, E value);

//E remove (int index)
E remove (Node<E> node)
```




```
//void add(int index, E value)
Position<E> addBefore (Position<E> pos, E value);
Position<E> addAfter  (Position<E> pos, E value);

//E remove (int index)
E remove (Position<E> pos)
```

Interfaz *position (LinkedPositionList)*

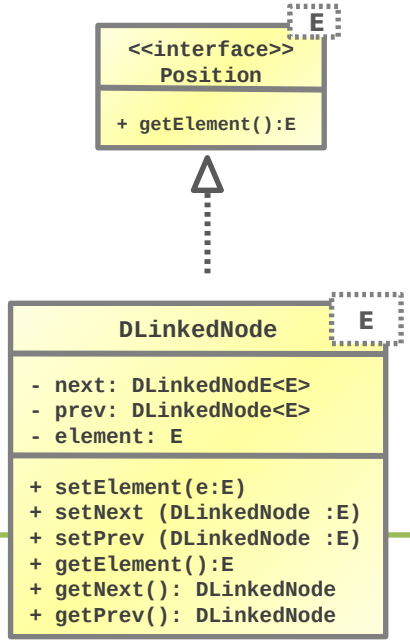


Operaciones como `remove` y `addBefore` requieren que la lista sea doblemente enlazada

Interfaz *position (LinkedList)*

- Operación de actualización `addBefore`

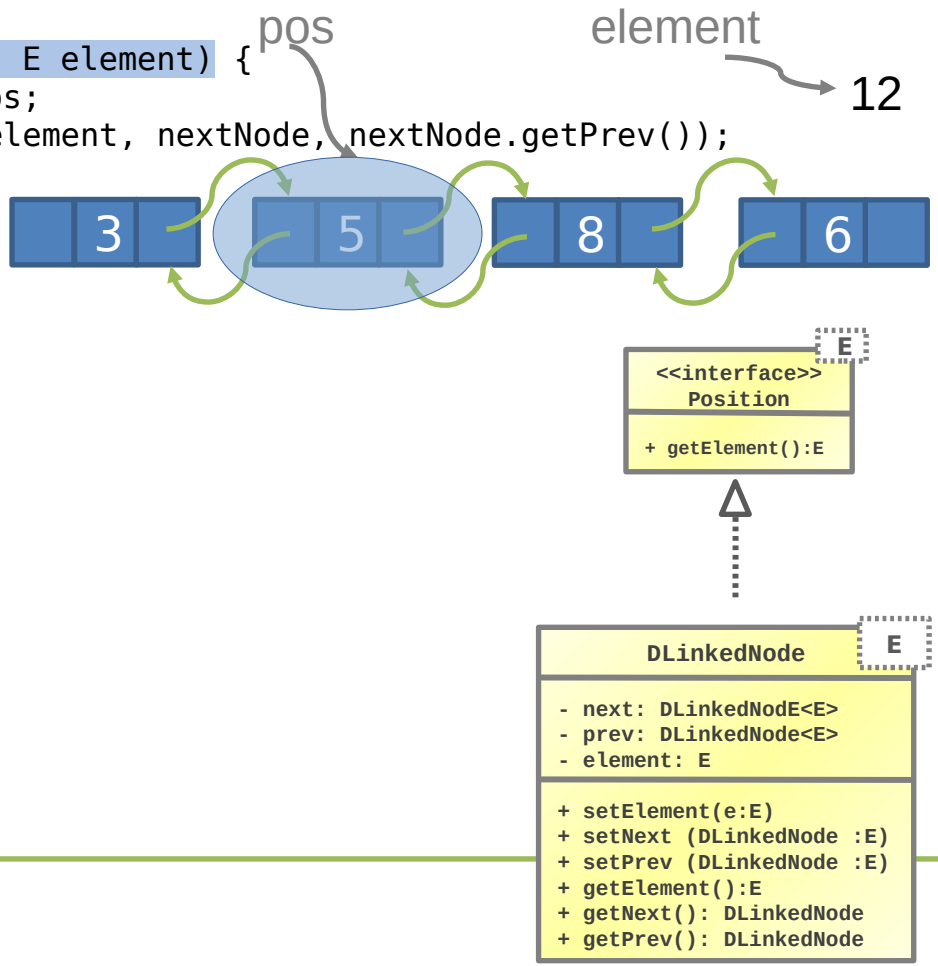
```
public Position<E> addBefore(Position<E> pos, E element) {
DLinkedNode<E> nextNode = (DLinkedNode<E>) pos;
DLinkedNode<E> newNode = new DLinkedNode<E>(element, nextNode, nextNode.getPrev());

if (this.head==nextNode){
     this.head = newNode;
}
else{
     nextNode.getPrev().setNext(newNode);
}

nextNode.setPrev(newNode);

this.size++;
return (Position<E>) newNode;
}
```



Interfaz *position (LinkedList)*

- Operación de actualización `addBefore`

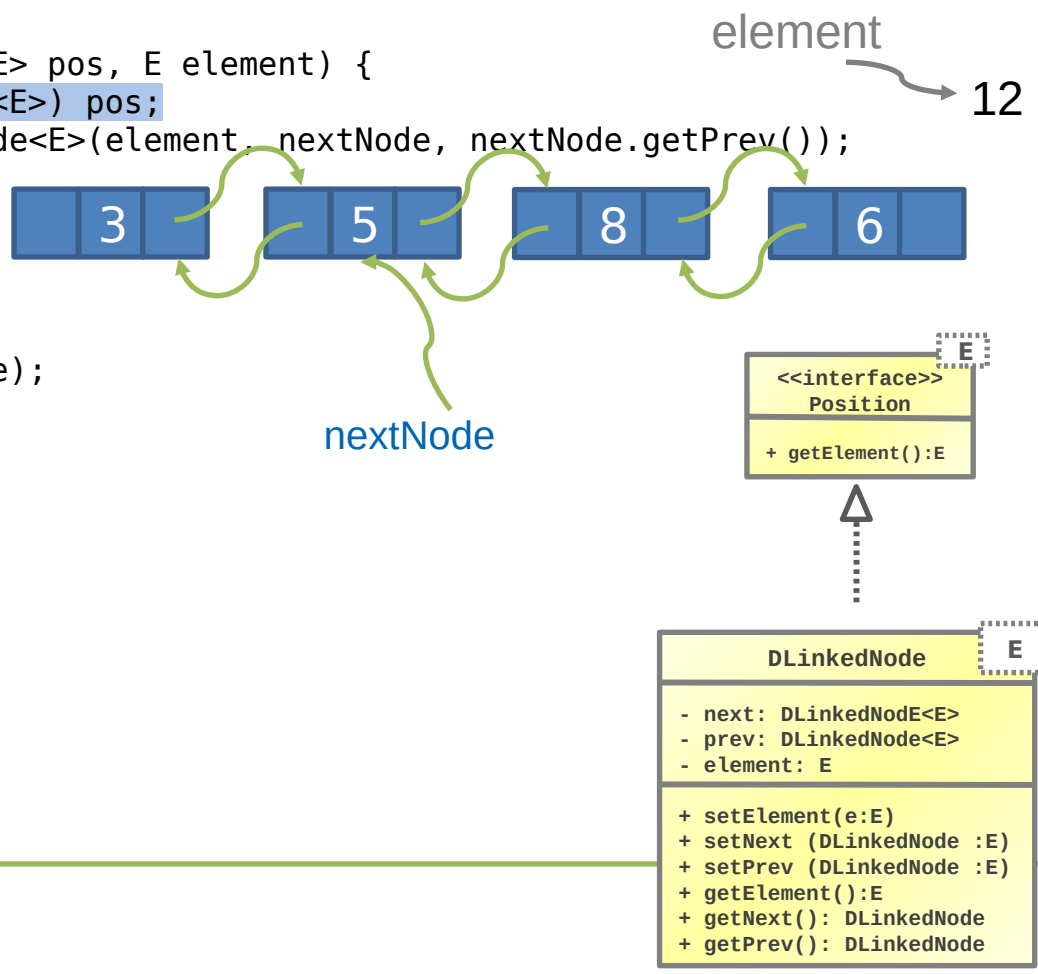
```
public Position<E> addBefore(Position<E> pos, E element) {
DLinkedNode<E> nextNode = (DLinkedNode<E>) pos;
DLinkedNode<E> newNode = new DLinkedNode<E>(element, nextNode, nextNode.getPrev());

if (this.head==nextNode){
     this.head = newNode;
}
else{
     nextNode.getPrev().setNext(newNode);
}

nextNode.setPrev(newNode);

this.size++;
return (Position<E>) newNode;
}
```



Interfaz *position (LinkedList)*

- Operación de actualización `addBefore`

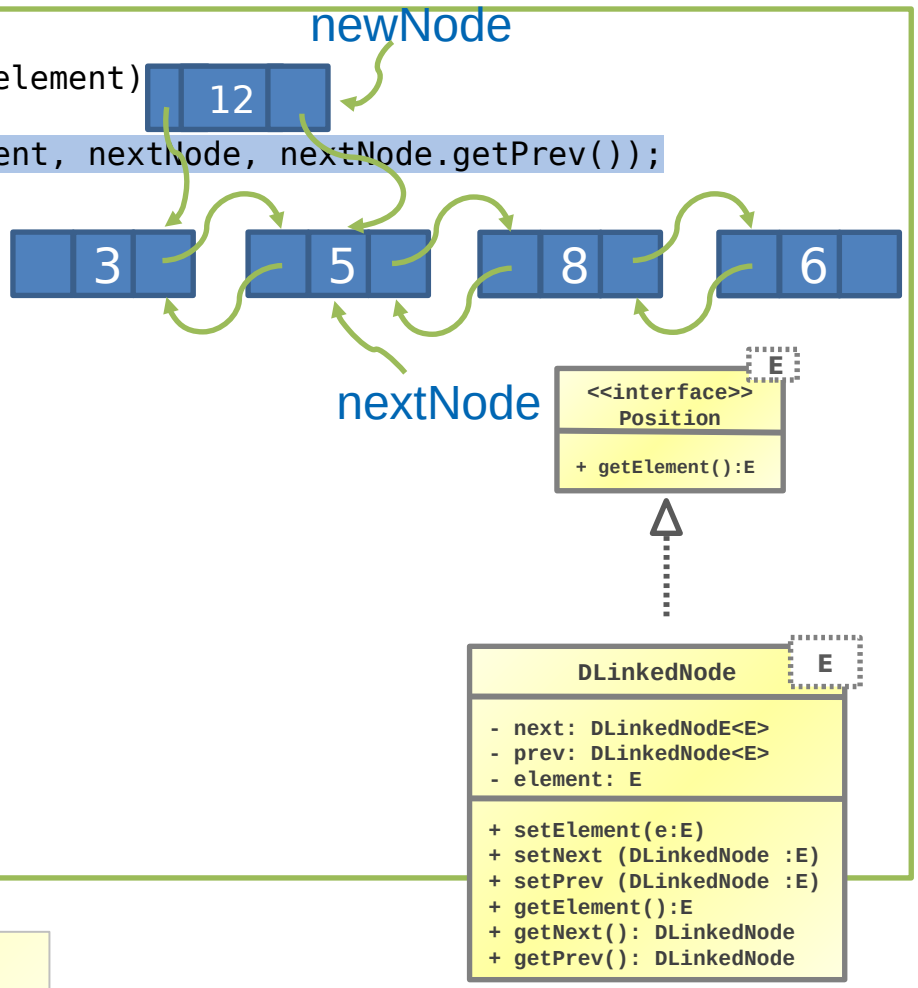
```
public Position<E> addBefore(Position<E> pos, E element) {
DLinkedNode<E> nextNode = (DLinkedNode<E>) pos;
DLinkedNode<E> newNode = new DLinkedNode<E>(element, nextNode, nextNode.getPrev());

if (this.head==nextNode){
     this.head = newNode;
}
else{
     nextNode.getPrev().setNext(newNode);
}

nextNode.setPrev(newNode);

this.size++;
return (Position<E>) newNode;
}
```



Interfaz *position (LinkedList)*

- Operación de actualización `addBefore`

```
public Position<E> addBefore(Position<E> pos, E element)
DLinkedNode<E> nextNode = (DLinkedNode<E>) pos;
DLinkedNode<E> newNode = new DLinkedNode<E>(element, nextNode, nextNode.getPrev());

if (this.head==nextNode){
     this.head = newNode;
}
else{
     nextNode.getPrev().setNext(newNode);
}

nextNode.setPrev(newNode);

this.size++;
return (Position<E>) newNode;
}
```



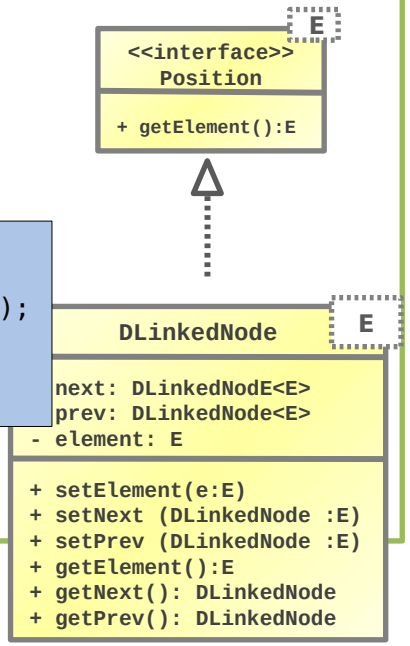
¿Y si pos no es una posición válida?

Interfaz *position (LinkedList)*

- Operación de actualización `addBefore`

```
public Position<E> addBefore(Position<E> pos,E element) throws InvalidPositionException{
DLinkedNode<E> nextNode = this.checkPosition(pos);
DLinkedNode<E> newNode = new DLinkedNode<E>(element, nextNode, nextNode.getPrev());

if (this.head==nextNode){
     this.head = newNode;
}
else{
     nextNode.getPrev().setNext(newNode);
}

nextNode.setPrev(newNode);

this.size++;
return newNode
}
```

```
private DLinkedNode<E> checkPosition(Position<E> p) {
        if (p == null || !(p instanceof DLinkedNode)) {
            throw new InvalidPositionException("The position is invalid");
        }
        return (DLinkedNode<E>) p;
    }
```



Framework Collections

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- A las estructuras de datos en Java se las denomina colecciones

- Colecciones: objetos que mantienen una colección de elementos, independientemente de su estructura interna

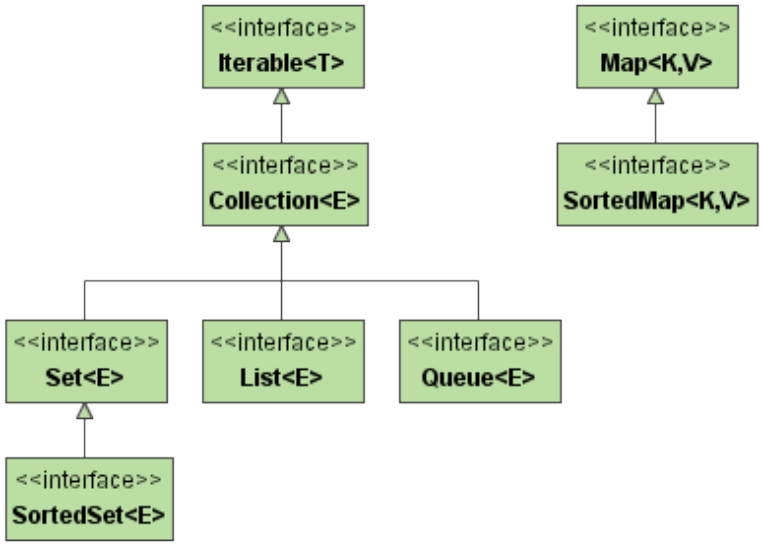
- *Framework Collections*
  - Arquitectura unificada para representar y manejar colecciones
  - Separa la parte pública (interfaz) de los detalles de implementación internos
  - Se conforma de interfaces, implementaciones y algoritmos

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- Interfaces:
  - Permiten que las colecciones puedan manipularse independientemente de los detalles de implementación

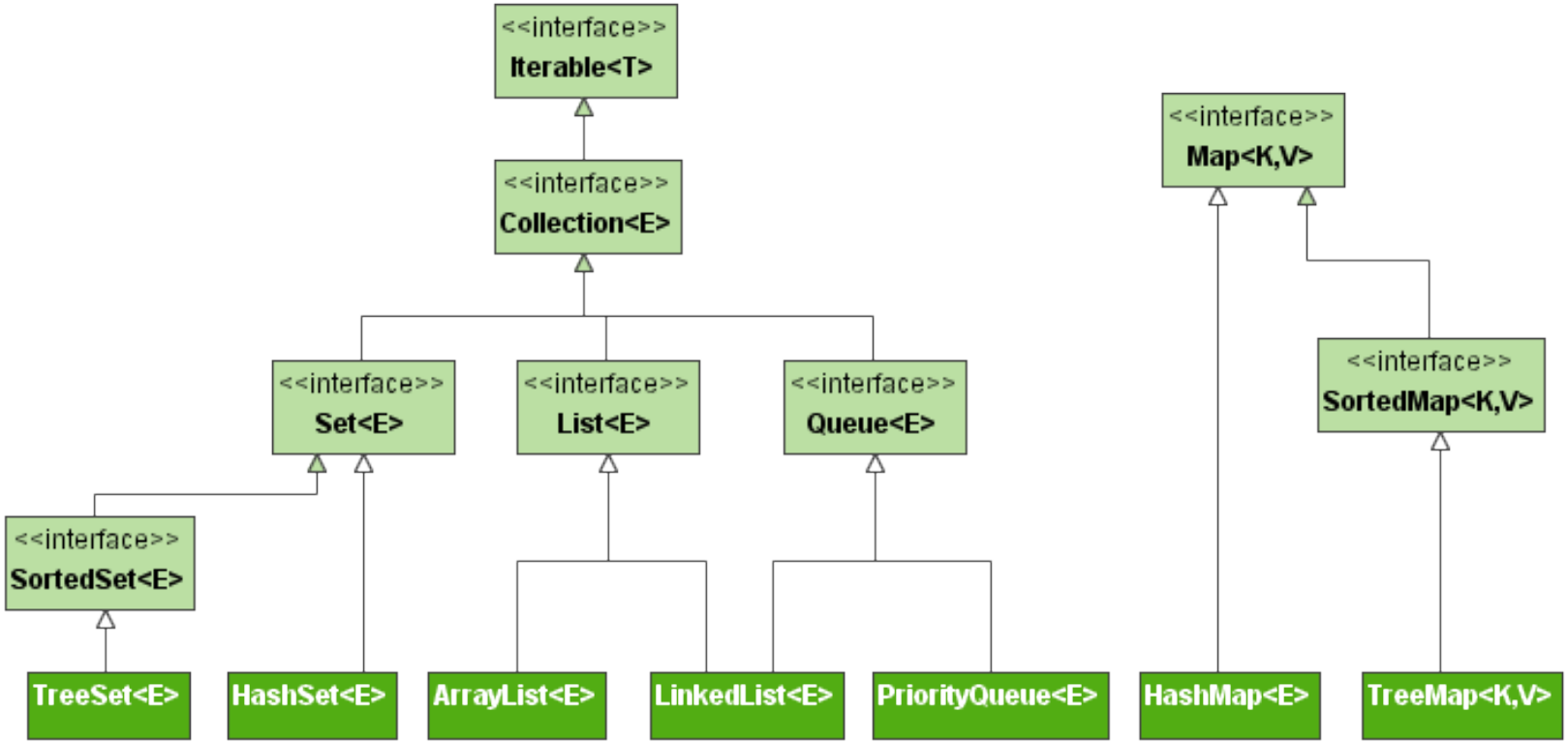
```
T max(Collection<? extends T> coll){…}

void reverse(List<?> list){…}
```

  - Definen la funcionalidad, no cómo debe implementarse esa funcionalidad



Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- Implementaciones:
  - Clases que implementan los interfaces que definen los tipos de colecciones (listas, mapas y conjuntos)



Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- Algoritmos
  - Métodos que realizan algún cómputo concreto sobre colecciones de elementos
  - Se dice que son polimórficos
    - Mismo método puede utilizarse con diferentes implementaciones de una colección
  - `java.util.Collections`
    - Búsqueda, ordenación, etc.

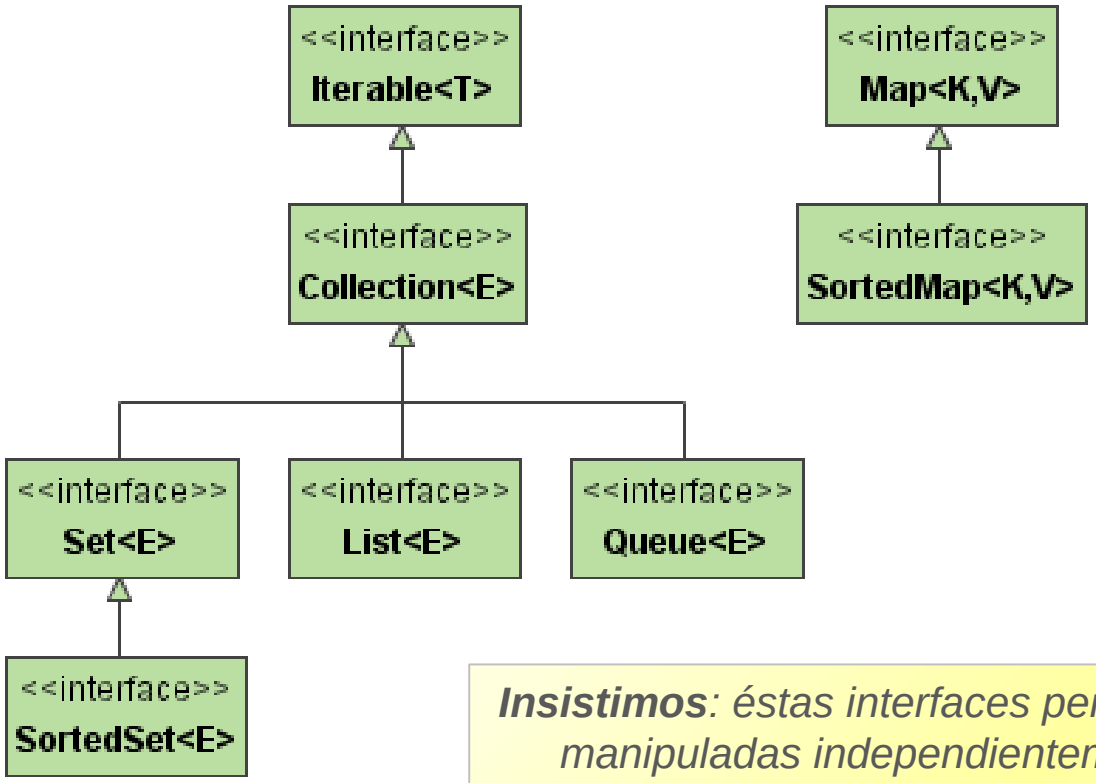
```
T max(Collection<? extends T> coll){…}

void reverse(List<?> list){…}
```

Framework Collections: Interfaces

Collections: Interfaces

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- *Core collection interfaces*
  - *Encapsula diferentes tipos de colecciones*

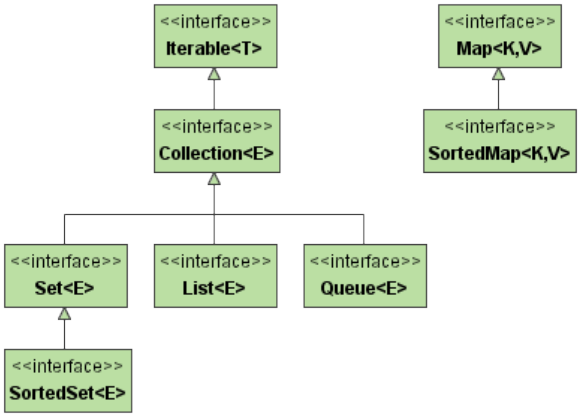


***Insistimos**: éstas interfaces permiten a las colecciones ser manipuladas independientemente de sus detalles de implementación*

- **`Iterable<T>`**

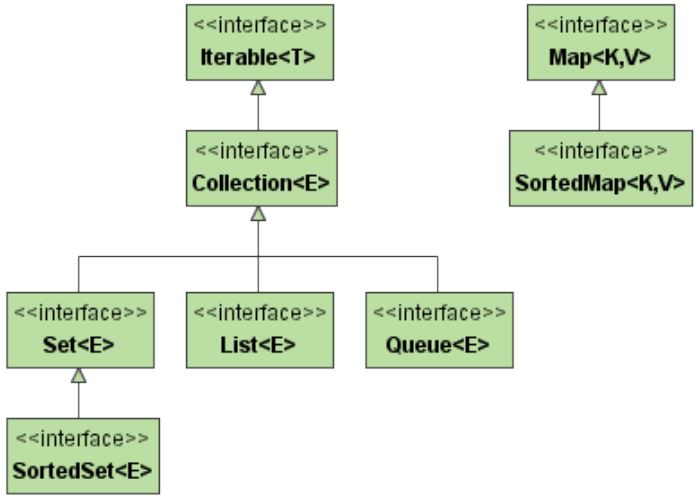
  - Representa la expresión mínima de una colección de elementos

  - Sólo permite recorrer los elementos
    - No permite consultar cuantos son
    - No permite añadir elementos




```
public static void print (Iterable<?> iterableCollection){
    Iterator<?> it = iterableCollection.iterator();
    while (it.hasNext()){
        System.out.println(it.next().toString());
    }
}
```

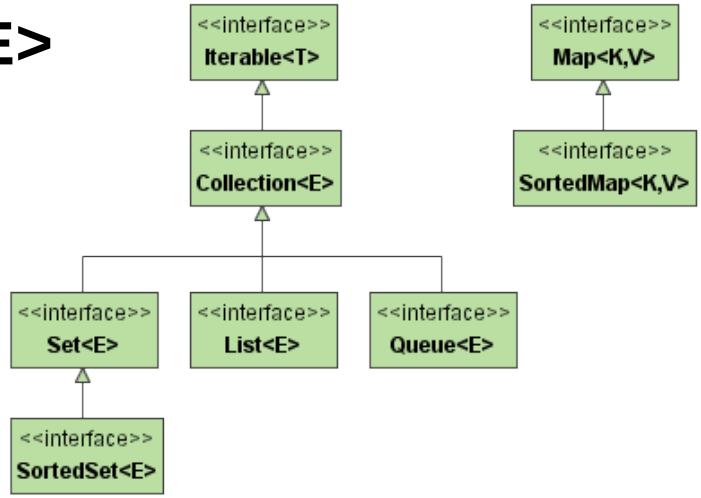
- **`Collection<E>`**
  - Representa a una colección de objetos
  - Añade nueva funcionalidad:
    - Consultar vacía
    - Consultar número de elementos
    - Añadir elemento/s
    - ...

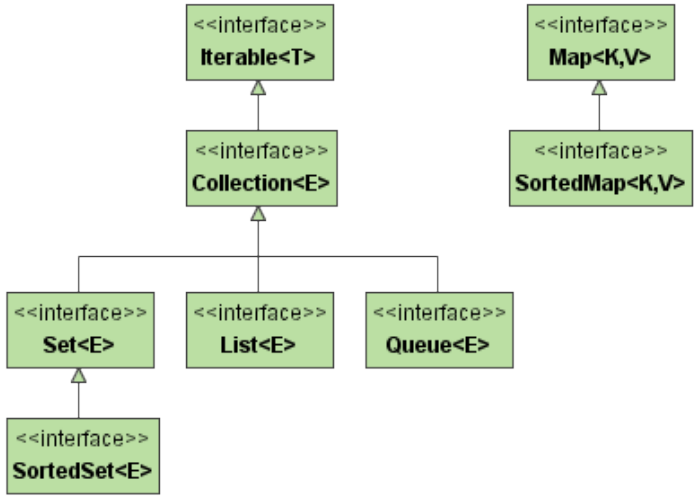


  - Clase padre de las colecciones con acceso por posición y de forma secuencial

  - Dependiendo de las interfaces hijas
    - Hay colecciones que permiten elementos duplicados y otras no
    - Hay colecciones ordenadas o desordenadas
    - Hay colecciones que permiten el valor `null`, otras no

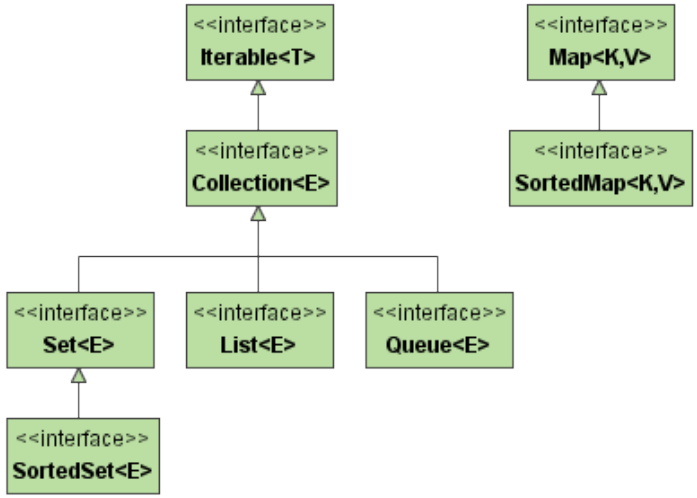
- Algunos métodos de **`Collection<E>`**
  - Para agregar y eliminar elementos
    - `boolean add(E e)`
    - `boolean remove(Object o)`
  - Para realizar consultas
    - `int size()`
    - `boolean isEmpty()`
    - `boolean contains(Object o)`
  - Para realizar varias operaciones de forma simultánea
    - `boolean containsAll(Collection<?> collection)`
    - `void clear()`
    - `boolean removeAll(Collection<?> collection)`





- **`Set<E>`**
  - Colección que no mantiene el orden de inserción y que no puede tener dos o más objetos iguales
- **`List<E>`**
  - Colección que sí mantiene el orden de inserción y que puede contener elementos duplicados
- **`Queue<E>`**
  - Colección para almacenar múltiples elementos antes de ser procesados
    - Elementos ordenados bajo un criterio FIFO
      - Excepciones: colas de prioridad

- **`SortedSet<E>`**
  - Conjunto que mantiene todos los elementos ordenados
    - Orden ascendente
- **`Map<K,V>`**
  - Estructura que guarda los elementos (valores) asociados a una clave
- **`SortedMap<K,V>`**
  - Mapa que mantiene sus claves ordenadas
    - Orden ascendente



Los mapas se verán en profundidad en temas posteriores

Framework Collections: Implementaciones

Collections: Implementaciones

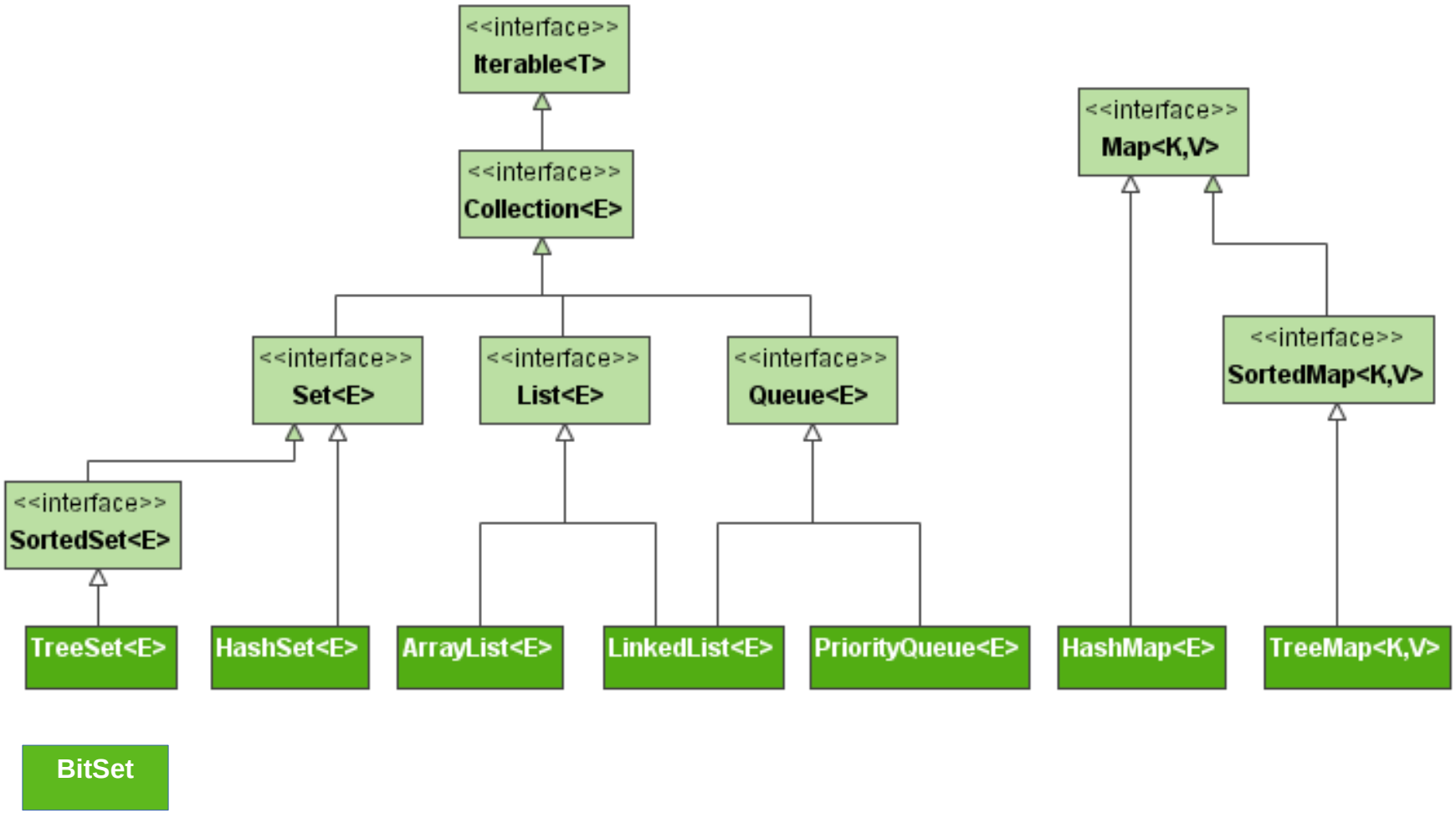
Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- El *framework Collections* proporciona implementaciones de propósito general para las interfaces presentadas

- Más comunmente utilizadas
  - `Set<E>` → **`HashSet<E>`**
  - `List<E>` → **`ArrayList<E>`**
  - `Map<K,V>` → **`HashMap<K,V>`**
  - `SortedSet<E>` → **`TreeSet<E>`**
  - `SortedMap<K,V>` → **`TreeMap<K,V>`**
  - `Queue<E>`
    - **`LinkedList<E>`**: Cola FIFO
    - **`PriorityQueue<E>`**: Ordena sus elementos antes de ser procesados

Collections: Implementaciones

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas



Listas (*List<E>*)

- Colección que mantiene el orden de inserción y que puede contener elementos duplicados
- Similar a un array pero que crece de forma dinámica
- Se accede a los elementos indicando su posición (tipo `int`)
- Algunos métodos:

```
void add(int index, E element)
boolean addAll(int index,  Collection<? extends E> c)
E get(int index)
E remove(int index)
```

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

Listas (*List<E>*)

- Es la estructura de datos más usada
- Es la estructura de datos más eficiente para la inserción de elementos (al final)
- No obstante, no es muy eficiente para búsquedas (porque son secuenciales)

Listas (*List<E>*)

- Interfaces *vs.* Implementaciones
  - Las variables, parámetros y atributos se declaran con el tipo de las interfaces
  - La clase de implementación sólo se usa para instanciar los objetos
  - Se abstrae lo más posible de la implementación concreta (y se puede cambiar fácilmente en el futuro)

ArrayList<E> es la clase por defecto que implementa List<E>

```
List<Intervalo> listaIntervalos = new ArrayList<Intervalo>();
List<Fraccion> listaFracciones = new ArrayList<Fraccion>();

listaIntervalos.add(new Intervalo(2,4));
listaFracciones.add(new Fraccion(2,6));
…
Intervalo intervalo = listaIntervalos.get(0);
Fraccion fraccion = listaFracciones.get(0);
```

Conjuntos (*Set<E>*)

- No mantiene el orden de inserción
- No es posible recuperar los elementos en el orden en que fueron insertados
- No admite elementos duplicados
  - Si se añade un objeto al conjunto y ya había otro igual, no se produce ningún cambio en la estructura
- Es la estructura de datos **más eficiente buscando elementos**

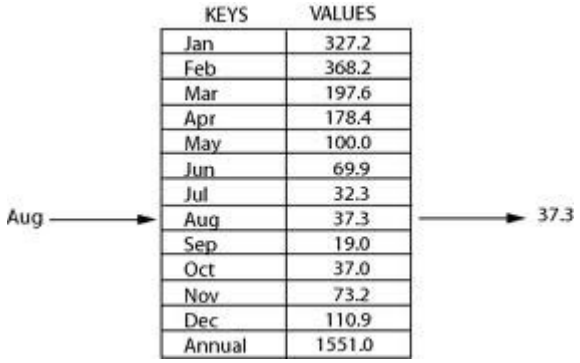
Conjuntos (*Set<E>*)

- `HashSet<E>` es la implementación por defecto de `Set<E>` y se implementa utilizando una tabla hash
  - Tiempo constante para la mayor parte de sus operaciones
    - `add()`, `remove()`, `contains()` y `size()`
  - No mantiene orden de inserción

```
//Declaro la variable del tipo de la interfaz,
//y le asigno un objeto del tipo de la clase de
//implementación.
Set<Intervalo> intervalos = new HashSet<Intervalo>();

Intervalo intervalo = new Intervalo(2,4);
intervalos.add(intervalo);
//Esta inserción no tiene efecto
intervalos.add(intervalo);
int numIntervalos = intervalos.size(); // Devuelve 1
```

Mapas (Map<*K*,*V*>)

- Define una estructura de datos que asocia o "mapea" claves con valores
  - `<idContacto, teléfono>`
  - `<palabra, significado>`
  - `<ciudad, litros de agua>`
  - …



- No permite claves repetidas
- Varias claves distintas pueden estar asociadas al mismo valor
- La búsqueda de un valor asociado a una clave es muy eficiente
  - Tiempo constante

Mapas se verán con detalle en temas posteriores

Mapas (Map<*K*,*V*>)

- Algunos métodos:
  - `V put(K key, V value)` : insertar un valor asociado a la clave
  - `V get(Object key)` : obtener un valor asociado a la clave
  - `Collection<V> values()` : devuelve la colección de valores
  - `Set<K> keySet()`: devuelve el conjunto de claves
  - `Entry<K,V> entrySet()`: devuelve el conjunto de pares clave-valor (entradas del mapa)

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

Mapas (Map<*K*,*V*>)

- `HashMap<K,V>` es la implementación por defecto de `Map<K,V>` que implementa el conjunto de datos utilizando una tabla hash

```
Map<String, Coche> propietarios = new HashMap<String, Coche>(5);

Coche toledo = new Coche("Seat", "Toledo", 110)
Coche punto = new Coche("Fiat", "Punto", 90);

propietarios.put("M-1233-YYY", toledo);
propietarios.put("M-1234-ZZZ", punto);
…

Coche c = propietarios.get("M-1234-ZZZ");
```

Estructuras de datos ordenadas

- Cuando los elementos son comparables entre sí, puede ser útil insertar de forma ordenada los elementos
  - `SortedSet<E>`, `SortedMap<E>`

- Un objeto es comparable si:

  - Implementa la interfaz `Comparable<T>`, orden natural

  - ¿Y si quiero ordenar en base a otro criterio que no sea el marcado como orden natural?
    - Proporcionar un `Comparator<T>`

Lectura obligada:
http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

Estructuras de datos ordenadas

- **`SortedSet<E>`**
  - Ordena los elementos de manera ascendente
  - Permite realizar consultas basadas en rango
    - Dado un texto, dame las palabras que empiecen por “de”. Muéstralas ordenadas de manera ascendente.
  - Implementación `TreeSet<E>`
    - `add(), remove(), contains()` $\in O(\log n)$

- **`SortedMap<K,V>`**
  - Ordena las claves de forma ascendente
  - Permite realizar consultas basadas en rango
    - Comité olímpico: ciudades con nota mayor a 6 (<nota, ciudades>)
  - Implementación `TreeMap<E>`
    - `containsKey(), get(), put(), remove()` $\in O(\log n)$

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

Otras implementaciones

- Aparte de las implementaciones por defecto, existen otras implementaciones para situaciones especiales
  - De propósito general
  - De propósito específico
  - Para soporte de concurrencia
  - Combinadas
  - *Wrappers*
  - etc…

Más info:
http://docs.oracle.com/javase/tutorial/collections/implementations/index.html

Framework Collections: Colecciones de tipos primitivos

Recorrer una colección

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- Hay tres posibles formas de recorrer una colección
  - Usando un bucle for con acceso por posición
  - Usando los iteradores (`Iterator<E>`) de forma secuencial
  - **Usando el for mejorado**

```
List<String> ciudades = new ArrayList<String>();
ciudades.add("Ciudad Real");
ciudades.add("Madrid");
ciudades.add("Valencia");

for (String ciudad: ciudades) {
    System.out.println(ciudad + "\n");
}
```

Colecciones con tipos primitivos

- Existen clases que se comportan como los tipos primitivos (clases de envoltura, *wrapper*)
  - `Integer`, `Double`, `Float`, `Boolean`, `Character`…
- El *autoboxing* y *autounboxing* es la capacidad de conversión automática entre un valor de un tipo primitivo y un objeto de la clase correspondiente

```
int numero = 3;
Integer numObj = numero;
int otroNum = numObj;
```

Colecciones con tipos primitivos

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- Esto permite usar las colecciones con tipos primitivos
- Hay que ser consciente de que se tienen que realizar conversiones y eso es **costoso**

```
List<Integer> enteros = new ArrayList<Integer>();
enteros.add(3);
enteros.add(5);
enteros.add(10);

int num = enteros.get(0);
```

Colecciones con tipos primitivos

- Existen implementaciones de terceros con estructuras de datos especialmente diseñadas para tipos primitivos

- Deben usarse cuando se utilizan mucho en un programa y las conversiones sean muy numerosas

  - http://trove4j.sourceforge.net/
  - http://fastutil.dsi.unimi.it/
  - http://commons.apache.org/primitives/

Framework Collections: Algoritmos

Collections: Algoritmos

Autoría: Micael Gallego Carrillo y Francisco Gortázar Bellas

- `java.util.Collections`
- Colección de algoritmos polimórficos
  - Métodos estáticos
    - Primer argumento: colección sobre la que operar
  - La mayoría trabajan sobre instancias de `List` y `Collection`
- Clasificación
  - *Sorting*
  - *Shuffling*
  - *Data manipulation*
  - *Searching*
  - *Composition*
  - *Finding extreme values*

- *Sorting*
  - Reordena la lista en orden ascendente
    - En base a su orden natural → `Comparable<E>`

```
List<String> nombres = new ArrayList<String>();
nombres.add("Pepe");
nombres.add("Juan");
nombres.add("Antonio");

Collections.sort(nombres);
System.out.println(nombres);
```

nombres = [Antonio, Juan, Pepe]

- *Sorting*
  - Reordena la lista en orden ascendente
    - En base a su orden natural → `Comparable<E>`
    - En base a un criterio diferente → `Comparator<E>`

```
List<String> nombres = new ArrayList<String>();
nombres.add("Juanin");
nombres.add("Pepe");
nombres.add("Antonio");

Collections.sort(nombres, new Comparator<String>(){
   public int compare(String o1, String o2) {
      return o1.length() - o2.length();
   }
});
```

nombres = [Pepe, Juanin, Antonio]

- *Shuffling*
  - `suffle`: reordena los elementos de una lista de forma aleatoria
- *Routine data manipulation*
  - `reverse`: invierte orden de los elementos en una lista
  - `fill`: cada elemento se sustituye por el elemento especificado
  - `copy`: copia los elementos de una lista origen en una lista destino
  - `swap`: intercambio de elementos en las posiciones especificadas de una lista
  - `addAll`: añade los elementos especificados a una colección

- *Searching*
  - La forma más eficiente de saber si un elemento está o no en una estructura de datos es usar un `Set` o un `Map`
  - Si usamos listas
    - Si la lista está desordenada, usamos el método `indexOf(E e)`
    - Si la lista está ordenada, se puede usar una búsqueda binaria
      - Si el elemento está en la lista, devuelve su posición
      - Si el elemento no está en la lista, devuelve el lugar en el que debería estar

- *Searching*

```
List<String> nombres = new ArrayList<String>();
nombres.add("Pepe");
nombres.add("Juan");
nombres.add("Antonio");

Collections.sort(nombres);

//int pos = nombres.indexOf("Mario");
int pos = Collections.binarySearch(nombres, "Mario");
if (pos < 0){
    //El nombre no está en la lista
    int insertPos = -pos-1;
    System.out.println("No está. Debería estar en: "+insertPos);
} else {
    System.out.println("Está en la posición: "+pos);
}
```

- *Composition*
  - `frequency`: número de veces que aparece un determinado elemento dentro de una colección
  - `disjoint`: determina si dos colecciones son disjuntas (ningún elemento en común)
- *Finding Extreme Values*
  - `min`
  - `max`

Ventajas de las colecciones

- Reducción del esfuerzo del programador
- Incremento de la velocidad y calidad
- Interoperabilidad entre APIs no relacionadas
- Menor esfuerzo de aprendizaje y uso de otras APIs
- Fomenta la reutilización del software

Ventajas de las colecciones

- Aunque las estructuras de datos de la API son muy completas, existen librerías de terceros que las complementan
  - Google Guava:
    - http://code.google.com/p/guava-libraries/
  - Otras:
    - http://java-source.net/open-source/collection-libraries