Tema 5 – Genericidad y Colecciones

Programación Orientada a Objetos

Curso 2014/2015

DIS
Departamento de
Informática y Sistemas

UNIVERSIDAD DE
MURCIA

Contenido – Parte I

- **Genericidad**:
  - Definición de clases genéricas.
  - Declaración y construcción de tipos genéricos.
  - Genericidad restringida.
  - Genericidad y herencia.
  - Genericidad y sistema de tipos.
  - Genericidad y máquina virtual.
  - Métodos genéricos.

Genericidad

- Facilidad de un lenguaje de programación para definir **clases, interfaces y métodos parametrizados con tipos** de datos.
- Resultan de utilidad para la implementación de **tipos de datos contenedores** como las **colecciones**.
- La genericidad sólo tiene sentido en **lenguajes con comprobación estática de tipos**, como Java.
- **La genericidad permite escribir código reutilizable.**

Genericidad

- Una **clase genérica** es una clase que en su declaración utiliza un tipo variable (**parámetro**), que será establecido cuando sea utilizada.

- Al **parámetro** de la clase genérica se le proporciona un nombre (`T`, `K`, `J`, etc.) que permite utilizarlo como **tipo de datos en el código de la clase**.

- Sobre las variables cuyo tipo sea el parámetro (`T`, `K`, `J`, etc.) **sólo es posible aplicar métodos de la clase `Object`**:
  - → dado que representan "cualquier dato" sólo podemos aplicar operaciones disponibles en todos los tipos de datos del lenguaje Java.

Clase genérica Contenedor

```
public class Contenedor<T> {

        private T contenido;

        public void setContenido(T contenido) {

                this.contenido = contenido;
        }

        public T getContenido() {

                return contenido;
        }
}
```

Operaciones disponibles

- Las operaciones aplicables sobre cualquier objeto (**métodos públicos de la clase `Object`**)

- Podemos aplicar la **asignación** (=) y la comparación de **identidad** (== o !=).

- Dentro de la clase genérica, **NO es posible construir objetos de los tipos parametrizados**:

  - ```
    T contenido = new T(); // No compila
    ```
    ```

Uso de una clase genérica

- La **parametrización** de una clase genérica se realiza en la **declaración** de una variable y en la **construcción de objetos**.

```
Contenedor<String> contenedor =
                new Contenedor<String>();

contenedor.setContenido("hola");
```

Genericidad y tipos primitivos

- Las clases genéricas **no pueden ser parametrizadas a tipos primitivos**.
- Para resolver este problema el lenguaje define **clases envoltorio** de los tipos primitivos:
  - **Integer**, **Float**, **Double**, **Character**, **Boolean**, etc.
- El compilador transforma automáticamente tipos primitivos en clases envoltorio y viceversa: **autoboxing**.

```
Contenedor<Integer> contenedor =
                        new Contenedor<Integer>();
contenedor.setContenido(10);
int valor = contenedor.getContenido();
```

Genericidad restringida

- **Objetivo**: limitar los tipos a los que puede ser parametrizada una clase genérica.

- Al restringir los tipos obtenemos el **beneficio** de poder **aplicar métodos sobre los objetos del tipo parametrizado**.

- Una clase con genericidad restringida sólo permite ser parametrizada con tipos **compatibles con el de la restricción** (clase o interfaz).

Genericidad restringida

- **Ejemplo**: la clase `CarteraAhorro` sólo puede ser parametrizada con tipos compatibles con `Deposito`.
- Podemos aplicar métodos disponibles en la clase `Deposito`.

```
public class CarteraAhorro<T extends Deposito> {

        private LinkedList<T> contenido;

        public void liquidar() {
                for (T deposito : contenido)
                        deposito.liquidar();
        }…
}
```

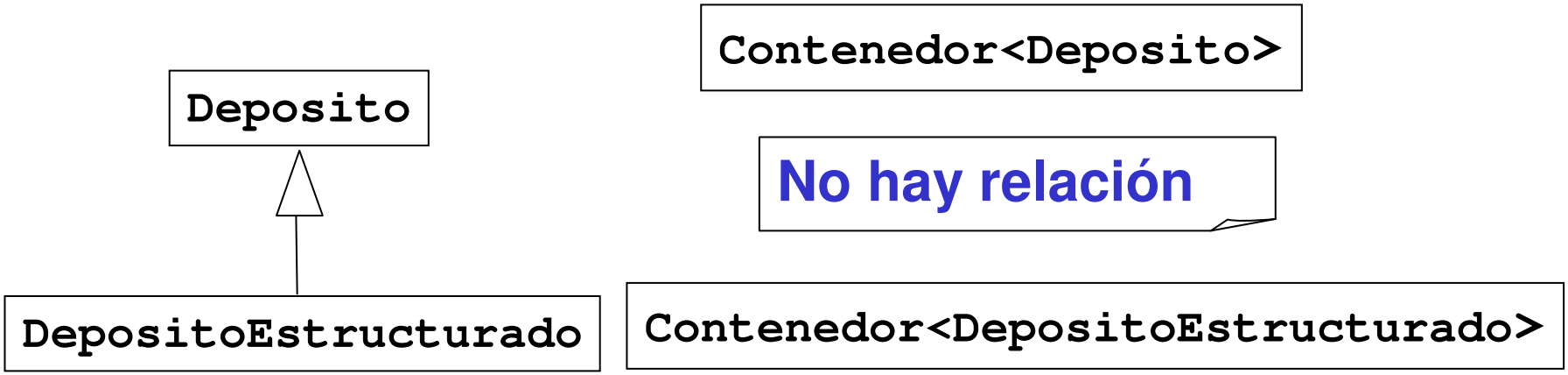
Genericidad restringida

- Una clase genérica puede estar **restringida por varios tipos**:

```
public class Contenedor<T extends Deposito & Amortizable>
```

- ➔ Las operaciones disponibles para objetos de tipo `T` es la unión de todos los tipos de la restricción.
  - En el ejemplo, todas las operaciones de la clase `Deposito` y la interfaz `Amortizable`.

Genericidad y sistema de tipos




```
Contenedor<Deposito> cDeposito;
Contenedor<DepositoEstructurado> cEstructurado;
…
cDeposito = cEstructurado; // No compila
```

Genericidad y sistema de tipos

- Las reglas del **polimorfismo** se mantienen entre clases genéricas.
- Sin embargo, en una asignación polimórfica **no está permitido que tengan distintos parámetros**.
- En el ejemplo, las dos variables son del mismo tipo (`Contenedor`), pero han sido parametrizadas a tipos distintos (`Deposito` y `DepositoEstructurado`).
  - No importa que `Deposito` y `DepositoEstructurado` sean tipos compatibles.
- Es una **limitación en el paso de parámetros**.

Genericidad y sistema de tipos

- **Problema**: el método sólo permite objetos de tipo `List<Deposito>`.

```
public double posicionGlobal(List<Deposito> depositos) {

        double posicion = 0;
        for (Deposito deposito : depositos) {

                posicion += deposito.getCapital();
        }
        return posicion;
}
```

- ¿Cómo podemos pasar una variable de tipo `List<DepositoEstructurado>`?

Genericidad y sistema de tipos

- **Solución**: **tipo comodín**.

```
public double posicionGlobal(
                          List<? extends Deposito> depositos)
```

- En el ejemplo significa: permite cualquier lista genérica parametrizada a la clase *Depósito* o a un tipo compatible (subclase).
- El tipo comodín se puede usar **también para declarar variables locales o atributos**.
- **No se puede utilizar para construir objetos**.
- Si se indica simplemente <?>, significa “cualquier tipo”.

Genericidad – Características avanzadas

- Dentro de una clase genérica se pueden utilizar otras clases genéricas.
- Una clase genérica puede tener **varios parámetros**.
- Una **interfaz** también puede declarar parámetros:
  - Un ejemplo son las interfaces que definen las colecciones.

```
public class ContenedorDoble <T,K> {
   private String nombre;
   private Contenedor<T> clave;
   private K valor; … }
```

```
ContenedorDoble<String, Cuenta> contenedor = …
```

Genericidad – Características avanzadas

- Es posible utilizar una clase genérica y no establecer sus parámetros (**tipo puro**).
- En **tiempo de ejecución** no se puede consultar el parámetro al que fue instanciada una clase genérica.
- Se puede aplicar **herencia** con clases genéricas.

Genericidad – Tipo puro

- Cuando se declara una variable cuyo tipo se corresponde con una clase genérica y no se especifica el parámetro se asigna el **tipo puro** (*raw*) que corresponde a:
  - Sin genericidad restringida, la clase `Object`.
  - Con genericidad restringida, la clase a la que se restringe.

```
Contenedor contenedor = new Contenedor();     // Object
CarteraAhorro cartera = new CarteraAhorro(); // Deposito
```

  - Siendo:
    - Clase `Contenedor<T>`
    - Clase `CarteraAhorro<T extends Deposito>`

Genericidad – Tiempo de ejecución

- **En tiempo de ejecución** se pierde la información sobre el tipo utilizado para parametrizar la clase genérica.
  - Todo tipo genérico (clase genérica parametrizada) se transforma a un **tipo puro**.
- Con el operador `instanceof` sólo podemos preguntar por el nombre de la clase.

```
// No compila
if (contenedor instanceof Contenedor<Deposito>) { … }

// Sí compila
if (contenedor instanceof Contenedor) { … }
```

Genericidad y herencia

- **Una clase puede heredar de una clase genérica.**
- La nueva clase tiene las **opciones**:
  - **Mantener la genericidad** de la clase padre.

```
public class CajaSeguridad<T> extends Contenedor<T>
```

  - **Restringir la genericidad.**

```
public class CajaSeguridad<T extends Valorable>
             extends Contenedor<T>
```

  - **No ser genérica y especificar un tipo concreto.**

```
public class CajaSeguridad extends Contenedor<Valorable>
```

Métodos genéricos

- Un método que declara una variable de tipo se denomina **método genérico**.
- Antes de la declaración del tipo de retorno del método se indica una variable que representa el tipo (`<T>`).
- El alcance de la variable de tipo (`<T>`) es local al método, esto es, puede aparecer en la signatura del método y en el cuerpo del método.
- Es posible definir métodos genéricos incluso en clases que no son genéricas.

Ejemplo 1

- Método que acepta una secuencia de valores de cualquier tipo y lo convierte en una lista:

```
public static <T> List<T> asList (T... datos) {
    List<T> lista =
       new ArrayList<T>(datos.length);

    for (T elemento : datos)
        lista.add(elemento);

     return lista;
}
```

Ejemplo 1

- El método `asList` se podría invocar como sigue:

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

    List<Integer> listaEnteros = asList(1,2,3);

    String[] arrayPalabras = {"hola", "ciao", "hello"};

    List<String> listaPalabras = asList(arrayPalabras);
}
```

- El valor de `T` se infiere a partir del tipo de los argumentos.

Ejemplo 2

- Añade una secuencia variable de elementos a una lista:

```
public static <T> void addAll (List<T> lista,
                               T... elementos) {
     for (T elemento : elementos)
               lista.add(elemento);
}
```

```
List<Integer> enteros = new ArrayList<Integer>();
addAll(enteros, 1, 2, 3);
```

Ejemplo 3

```
public static <T> T getElementoAleatorio(List<T> lista){
    Random random = new Random();

    int index = random.nextInt(lista.size());

    return lista.get(index);
}
```

```
List<Integer> enteros = new ArrayList<Integer>();

addAll(enteros, 1, 2, 3);

int entero = getElementoAleatorio(enteros);
```

Seminario 3

- El **Seminario 3** trata la genericidad en Java e incluye los siguientes **ejemplos**:
  - Genericidad basada en `Object`.
  - Definición de una clase genérica.
  - Genericidad restringida.

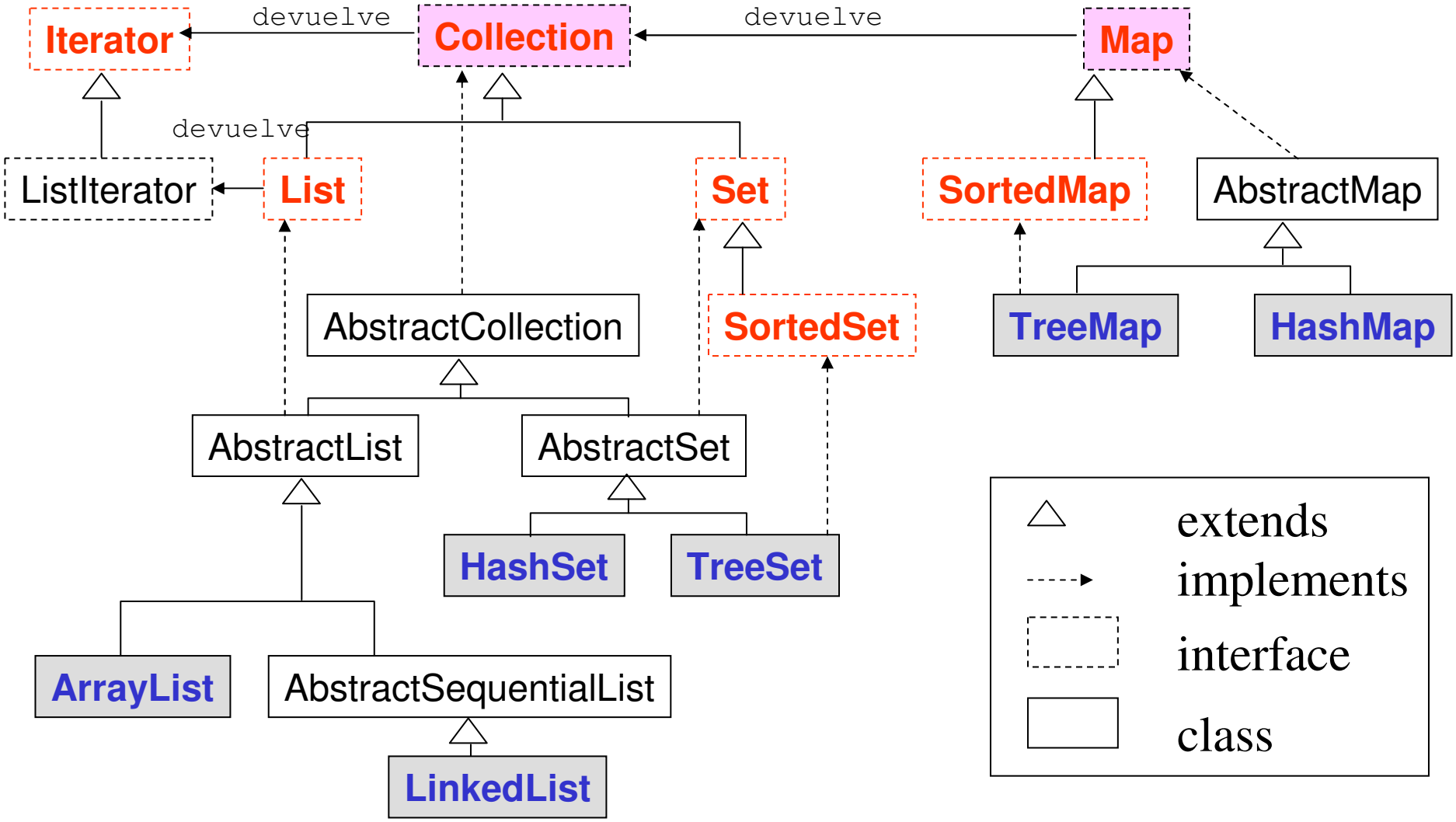
Contenido – Parte II

- Colecciones (paquete `java.util`):
  - Interfaz `Collection<T>`
  - Interfaz `List<T>`
  - Interfaz `Set<T>`
  - Interfaz `Map<K,T>`
  - Copia colecciones.
  - Orden objetos.
  - Iteradores.
  - Recomendaciones.

Colecciones en Java

- Las colecciones en Java son un **ejemplo** destacado de implementación de **código reutilizable** utilizando un lenguaje orientado a objetos.
- Todas las colecciones son **genéricas**.
- Los tipos abstractos de datos se definen como **interfaces**.
- Se implementan **clases abstractas** que permiten factorizar el comportamiento común a varias implementaciones.
- Un mismo **TAD** puede ser implementado por varias clases
  → **`List`**: **`LinkedList`**, **`ArrayList`**

Colecciones en Java



Interfaz `Collection<T>`

- Define las operaciones comunes a todas las colecciones de Java.
- Permite usar colecciones basándonos en su interfaz en lugar de en la implementación.
- Los tipos básicos de colecciones son (subtipos de `Collection<T>`) :
  - Listas, definidas en la interfaz `List<T>`
  - Conjuntos, definidos en la interfaz `Set<T>`

Interfaz `Collection<T>`

- **Operaciones básicas de consulta**:
  - `size()`: devuelve el número de elementos.
  - `isEmpty()`: indica si tiene elementos.
  - `contains(Object e)`: indica si contiene el objeto pasado como parámetro utilizando el método `equals`.

Interfaz `Collection<T>`

- **Operaciones básicas de modificación**:
  - **`add`**`(T e)`: añade un elemento a la colección.
    - Retorna un booleano indicando si acepta la inserción.
  - **`remove`**`(Object e)`: intenta eliminar el elemento.
    - Retorna un booleano indicando si ha sido eliminado.
    - Utiliza el método `equals` para localizar el objeto.
  - **`clear`**`()`: elimina todos los elementos.
  - **`addAll`**`(Collection<? extends T> col)`: añade todos los elementos de la colección `col`
  - **`removeAll`**`(Collection<?> col)`: elimina todos los objetos contenidos en `col`

Interfaz `List<T>`

- La interfaz **`List<T>`** define secuencias de elementos a los que se puede acceder atendiendo a su posición.

- Las posiciones van de `0` a `size()-1`.
  - El acceso a una posición ilegal produce la excepción `IndexOutOfBoundsException`
- El método **`add`**`(T e)` añade al final de la lista.
- Añade a las operaciones de `Collection` métodos de acceso por posición como:
  - `T` **`get`** `(int index)`
  - `T` **`set`** `(int index, T element)`
  - `void` **`add`** `(int index, T element)`
  - `T` **`remove`** `(int index)`

Clases que implementan `List<T>`

- **`ArrayList<T>`**
  - Implementación basada en **arrays redimiensionables**.
  - Operaciones de inserción y modificación ineficientes.
  - Operaciones de creación y consulta rápidas.

- **`LinkedList<T>`**
  - Implementación basada en **listas doblemente enlazadas**
  - Inserciones y modificaciones rápidas, especialmente en el principio y el final:
    - Métodos no disponibles en `List<T>`: `addFirst`, `addLast`, `removeFirst`, `removeLast`
  - Acceso aleatorio a elementos ineficiente.
  - Acceso eficiente al principio y al final de la lista:
    - `getFirst` y `getLast`

Interfaz `Set<T>`

- La interfaz **`Set<T>`** define conjuntos de elementos no repetidos.
- Implementaciones de conjuntos:
  - **`HashSet<T>:`**
    - Guarda los elementos del conjunto en una tabla *hash*.
    - Para evitar la inserción de elementos repetidos, la igualdad de los objetos se comprueba comparando los **`hashCode`**, si son iguales se compara con **`equals`**.
  - **`TreeSet<T>:`**
    - Implementación de **conjuntos ordenados** basada en árboles binarios balanceados.
    - Para su funcionamiento es necesario definir un **orden** (se estudia más adelante).
- Las operaciones de búsqueda y modificación son más lentas en `TreeSet` que en `HashSet`

Interfaz `Map<K,V>`

- La interfaz **`Map<K,V>`** define el tipo de datos que representa pares *<clave, valor>*
  - Un mapa no puede tener claves duplicadas.
  - Cada clave sólo puede tener un valor asociado.
- Un mapa no es una colección, sin embargo contiene distintas colecciones:
  - Conjunto de claves (`Set<K>`)
  - Colección de valores (`Collection<V>`)
  - Conjunto de pares <clave, valor> (`Map.Entry<K,V>`)
- Las implementaciones disponibles son:
  - **`HashMap<T>`**: implementación basada en una tabla *hash*
  - **`TreeMap<T>`**: implementación basada en árboles balanceados.
    - Las claves están ordenadas (`SortedSet<K>`).

Interfaz `Map<K,V>`

- **Métodos básicos**:
  - `V put(K clave, V valor)`: inserta una asociación en el mapa.
    - Retorna el valor de la antigua asociación, si la hubiera.
  - `V get(clave)`: retorna el valor asociado a una clave.
    - Si la asociación no existe, devuelve nulo.
  - `Set<K> keySet()`: devuelve el conjunto de claves.
  - `Collection<V> values()`: devuelve la colección de valores.
  - `boolean containsKey(key)`: indica si existe una clave.
  - `Set<Map.Entry<K, V>> entrySet()`: devuelve el conjunto de todas las asociaciones, `Map.Entry<K, V>`:
    - `getKey()`: consultar la clave.
    - `getValue()`: consultar el valor.

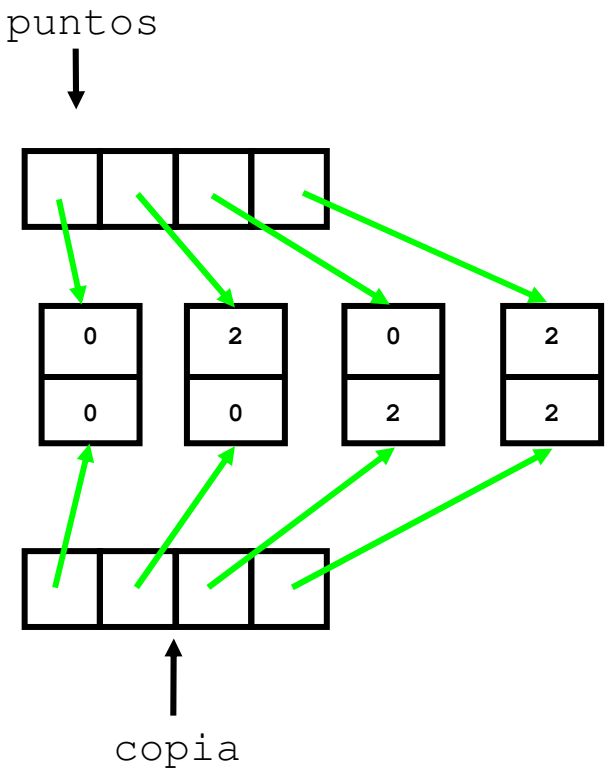
Copia de colecciones

- Todas las clases que implementan colecciones ofrecen un constructor de copia y el método clone.
- En ambos casos construye una **copia superficial** del objeto receptor.

```
LinkedList<Punto> puntos;
…
LinkedList<Punto> copia;

// Opción 1: copia con clone
copia = (LinkedList<Punto>)puntos.clone();

// Opción 2: constructor de copia
copia = new LinkedList<Punto>(puntos);
```



Orden de los objetos

- El orden utilizado por las colecciones ordenadas (`SortedSet, SortedMap`) puede ser el ***orden natural*** de los objetos (por defecto) o el **criterio de ordenación** que se establece en el constructor.
- La **interfaz `Comparable`** impone el orden natural de los objetos de las clases que la implementan.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- El método `compareTo` devuelve un entero positivo si la relación es “mayor que”, negativo si es “menor que” y cero si son iguales.

Orden natural de `Cuenta`

```
public class Cuenta implements Comparable<Cuenta>{
 …
 public int compareTo(Cuenta otraCta) {
    if (this.codigo > otraCta.codigo)

          return 1;

    else if (this.codigo < otraCta.codigo)

          return -1;

    else return 0;
  }
}
```

TreeSet<Cuenta> con orden natural

```
public class Persona {
    …
    private TreeSet<Cuenta> misCuentas;

    public Persona(String dni, String nombre){
       …
       //TreeSet utiliza el orden natural de la clase Cuenta
       misCuentas = new TreeSet<Cuenta>();
    }

    /**
     * Añade una cuenta a la colección de la persona que es titular
     * @param cta Cuenta a añadir en la colección
     * @return true si la cuenta se ha añadido y false en caso contrario
     */
    public boolean addCuenta(Cuenta cta){
         return misCuentas.add(cta);
    }
}
```

Criterios de ordenación

- Para definir un criterio de ordenación hay que implementar la **interfaz `Comparator`**.

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

- El método `compare` devuelve un entero positivo si la relación es "mayor que", negativo si es "menor que" y cero si son iguales.

- Se utiliza un criterio de ordenación cuando los objetos que queremos ordenar no tienen orden natural o ese orden no interesa usarlo.

Criterios de ordenación para `Cuenta`

```
public class OrdenSaldo implements Comparator<Cuenta>{
   public int compare(Cuenta o1, Cuenta o2) {
       if (o1.getSaldo() > o2.getSaldo())
            return 1;
       else if (o1.getSaldo() < o2.getSaldo())
               return -1;
            else
               return 0;
   }
}
```

Criterios de ordenación para `Cuenta`

```
public class OrdenTitular implements Comparator<Cuenta> {
    public int compare(Cuenta o1, Cuenta o2) {
        return (o1.getTitular().getNombre().
                    compareTo(o1.getTitular().getNombre()));

    }
}
```

TreeSet<Cuenta> con criterio de ordenación

```
public class Persona {
    …
  private TreeSet<Cuenta> misCuentas;


  public Persona(String dni, String nombre){
        …
     misCuentas = new TreeSet<Cuenta>(new OrdenTitular());

      //TreeSet utiliza el orden establecido
      //en la clase OrdenTitular para ordenar las cuentas
  }

}
```

Iteradores

- Las colecciones de Java son iterables, es decir, podemos recorrer todos sus elementos.
- Se utilizan iteradores para que el código que realiza el recorrido no conozca las particularidades de la estructura de datos: lista enlazada, lista basada en arrays, etc.

```
public double posicionGlobal(List<Deposito> depositos) {
    double posicion = 0;
    for (Deposito deposito : depositos) {
        posicion += deposito.getCapital();
    }
    return posicion;
}
```

Iteradores

- Java proporciona la interfaz **`Iterable<T>`** que debe ser implementada por aquellas clases sobre las que se pueda iterar:

```
public interface Iterable<T> {

    Iterator<T> iterator();
}
```

- A los objetos iterables se les exige que creen objetos iterador (**`Iterator`**) para realizar la iteración.

- Los **arrays** y la interfaz **`Collection`** son iterables.

Iteradores

- Interfaz **`Iterator<T>`**:
  - **`hasNext`**`()` : indica si quedan elementos en la iteración.
  - **`next`**`()` : devuelve el siguiente elemento de la iteración.
  - **`remove`**`()` : elimina el último elemento devuelto por el iterador.

```
public interface Iterator<T> {

    boolean hasNext();

    T next();

    void remove();
}
```

Recorrido *for each*

- El recorrido **for each** permite recorrer objetos iterables sin manejar un objeto iterador.
- Es la opción más común de recorrido.

```
public double posicionGlobal(List<Deposito> depositos) {
    double posicion = 0;
    for (Deposito deposito : depositos) {
        posicion += deposito.getCapital();
    }
    return posicion;
}
```

Recorrido explícito con iterador

- Interesa manejar un iterador cuando queremos eliminar algún elemento de la colección.
  - En Java sólo se puede modificar una colección que se está recorriendo utilizando explícitamente el iterador.

```
public double filtrar(List<Deposito> depositos) {
    Iterator<Deposito> it = depositos.iterator();
    while (it.hasNext()) {
            Deposito deposito = it.next();
            if (deposito.getCapital() < 1000)
               it.remove();
    }
}
```

Recomendaciones

- **Programar hacia el TAD**
  - En constructores y métodos públicos, el tipo de retorno y el tipo de los parámetros se especifica utilizando la interfaz (por ejemplo `List` en lugar de `LinkedList`)

- Las colecciones tienen más funcionalidad que los arrays.

- Podemos obtener una lista a partir de un array:
  - ```
    List<Deposito> lista =
       Arrays.asList(depositos);
    ```
    ```

Recomendaciones

- Los colecciones suelen producir **aliasing** incorrectos.
- Soluciones:
  - **Copiar la colección** (clone o constructor de copia).
  - Devolver una ***vista* no modificable** de la colección:
    - `Collections.unmodifiableList(depositos);`
    - Existe una operación análoga para cada interfaz de las colecciones.
    - Es recomendable documentar que se devuelve una *vista* no modificable.
    - Es más eficiente que construir una copia.