

Colecciones y Genéricos



Resumen

- Framework
- Colecciones en Java
 - Jerarquía de interfaces
 - Collection (Set y List), Map (SortedMap)
 - Recorrido de colecciones: Iterator y ListIterator
 - Comparación de elementos: Comparable y Comparator
- Genéricos:
 - Aspectos generales
 - Motivación
 - Definición y uso
 - Comparaciones de elementos con tipos genéricos
 - Uso de genéricos en colecciones



Objetivos

- Conocer y comprender qué son los frameworks
- Conocer y comprender las características principales de las colecciones y los tipos que existen
- Conocer y comprender cómo recorrer una colección y cómo comparar los distintos elementos de una colección
- Utilizar distintos tipos de colecciones en pequeñas aplicaciones Java
- Realizar búsquedas, recorridos y comparaciones de elementos en colecciones
- Conocer y comprender los genéricos en Java y su uso dentro de colecciones
- Realizar pequeñas aplicaciones que utilicen genéricos y colecciones



Colecciones



Framework – Aspectos generales

- **Framework**: Término usado en POO para definir un conjunto de clases que definen un diseño abstracto para solucionar un conjunto de problemas relacionados
- ¿Qué se define en un *framework*?
 - Un conjunto de clases e interfaces (p.e. clases que define distintos tipos de componentes (*JButton*, *JTable*, ...) o interfaces de gestión de eventos (*MouseListener*, ...))
 - Modelos de uso de las clases e interfaces
 - Modelo de funcionamiento del *framework* en tiempo de ejecución, en el que hay que "enganchar" los nuevos objetos añadidos por nosotros (p.e. definición y tratamiento de eventos)



Frameworks - Ejemplos

- **Swing**: *framework* de interfaces gráficas de usuario
- **Jakarta Struts**: *framework* de aplicaciones *web* en Java basado en el patrón MVC
- **JMF (Java Media Framework)**: *framework* para el tratamiento de contenidos multimedia: audio, vídeo, ...
- **JAI (Java Advanced Imaging)**: *framework* para el procesamiento de imágenes en Java



Framework de Colecciones

- Arquitectura unificada para representar y manipular colecciones independiente de los detalles de representación
- Una colección es una agrupación de objetos
- Tipos de colecciones:
 - Acceso por posición: listas, conjuntos, ...
 - Acceso por clave: diccionarios, ...



Framework de Colecciones

- ¿Qué contiene un *framework* de colecciones?
- Interfaces
 - TADs que definen la funcionalidad
- Implementaciones
 - Clases que implementan las interfaces de las colecciones
 - Un TAD (p.e. secuencia o lista) puede tener más de una implementación
- Algoritmos
 - Métodos que realizan computaciones, como búsquedas u ordenaciones, sobre objetos que implementan las interfaces



Colecciones en Java – Aspectos generales

- Una colección es una agrupación de objetos
- Se encuentran en el paquete *java.util*
- Arrays
 - La clase más sencilla, está fuera del *framework* *Collection*
- Iterator
 - Iterator
- Colecciones
 - Collection
 - Set
 - List
 - Map



Programación Orientada a Objetos



Interfaz *Iterator*

- Java proporciona interfaces para recorrer los elementos de las colecciones, sin necesidad de conocer la implementación. La interfaz *Collection* define un método llamado *iterator()* que devuelve un objeto que implementa la interfaz *Iterator*
- Un *iterator* sirve para recorrer / modificar una colección de elementos en Java
- *Iterator* pertenece al *framework* de colecciones de Java
- Métodos:
 - *hasNext()*: Devuelve *true* si tiene más elementos
 - *next()*: Devuelve el siguiente elemento de la colección
 - *remove()*: Borra el último elemento devuelto por el iterador de la colección (llamada al método *next()*)

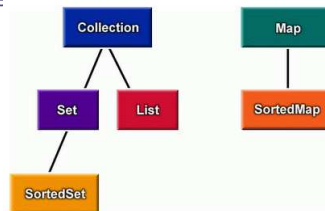


Programación Orientada a Objetos



Colecciones en Java – Aspectos generales

- Las colecciones en Java se encuentran estructuradas en la siguiente jerarquía de interfaces



- Ventajas de definir interfaces:
 - Se separa la especificación de la implementación
 - Es más sencillo reemplazar una clase por otra que implemente la misma interfaz y sea más eficiente



Programación Orientada a Objetos



Colecciones en Java - Tipos

- *Collection*: Representa un grupo de objetos, sin implementaciones directas, agrupando la funcionalidad general que todas las colecciones ofrecen
 - *Set*: Colección que no puede tener objetos duplicados
 - *SortedSet*: *Set* que mantiene los elementos ordenados
 - *List*: Colección ordenada que puede contener objetos duplicados
- *Map*: Colección que mapea claves y valores. No puede tener claves duplicadas
 - *SortedMap*: *Map* que mantiene las claves ordenadas



Programación Orientada a Objetos



La interfaz *Collection* – Aspectos generales

- Interfaz raíz en la jerarquía de colecciones
- Tipos de colecciones:
 - Colecciones con elementos duplicados o no
 - Colecciones con elementos ordenados o no
- Clases que implementen esta interfaz:
 - Constructores:
 - Sin parámetros crea una colección vacía
 - Con un argumento de tipo *Collection*, crea una colección con los mismos elementos de la colección que recibe como argumento
 - Algunas operaciones son opcionales de implementar



La interfaz *Collection* – Operaciones básicas

- `int size();` → Número de elementos de la colección
- `boolean isEmpty();` → `true` si la colección está vacía
- `boolean contains (Object element);` → `true` si la colección contiene un determinado objeto
- `boolean add (Object element);` → `true` si se consiguió añadir el elemento a la colección (`false` si el elemento ya existía y no se admiten repetidos)
- `boolean remove (Object element);` → `true` si borra un determinado objeto de la colección
- `Iterator iterator();` → Iterador sobre los elementos de la colección
- (*) `add` y `remove` son opcionales



La interfaz *Collection* – Operaciones básicas

- `boolean containsAll (Collection c);` → `true` si la colección contiene todos los elementos de la que recibe como argumento
- `boolean equals (Object o);` → Comparación de igualdad entre la colección y el parámetro. Sobreescrbe al método `equals` de la clase *Object*
- `Object[] toArray();` → Devuelve un array con todos los elementos de la colección
- `Object[] toArray (Object a[]);` → Recibe un array donde los objetos de la colección serán almacenados si hay espacio suficiente. Si no existe espacio suficiente, se creará un nuevo array del mismo tipo que "a" y se almacenarán los elementos de la colección en él



La interfaz *Collection* – Otras operaciones

- `boolean addAll (Collection c);` → Añade los elementos de la colección que recibe como parámetro a la colección
- `boolean removeAll (Collection c);` → Elimina los elementos de la colección que sean elementos también de la colección que recibe como parámetro
- `boolean retainAll (Collection c);` → Elimina los elementos de la colección que **NO** sean elementos también de la colección que recibe como parámetro
- `void clear();` → Elimina los elementos de la colección
- La implementación de las operaciones `addAll`, `removeAll`, `retainAll` y `clear` es opcional



La interfaz Set – Implementaciones

- Es similar a *Collection*, con la particularidad de que:
 - No se permiten objetos iguales en la colección
 - Los elementos no se almacenan necesariamente ordenados
- Implementación HashSet
 - Usa una tabla hash
 - No introduce orden en sus elementos
 - Los métodos `add`, `remove` y `contains` son muy eficientes: $O(1)$
- Implementación TreeSet
 - Usa un árbol
 - Introduce un orden en los elementos → útil para recorrer el conjunto
 - Los métodos `add`, `remove` y `contains` son $O(\log(n))$



Programación Orientada a Objetos



La interfaz Set – Ejemplo de *HashSet*

```
import java.util.*;
public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);
        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

```
% java FindDups i came i saw i left
Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```



Programación Orientada a Objetos



Ejercicio 1 - La interfaz Set – *TreeSet*

- ¿Cuál sería la salida del siguiente programa?

```
TreeSet set = new TreeSet();
set.add("b");
set.add("c");
set.add("a");
Iterator it = set.iterator();
while (it.hasNext()) {
    Object element = it.next();
    System.out.println("Elemento: " + element);
}
```



Programación Orientada a Objetos



La interfaz *List* – Aspectos generales

- Colecciones ordenadas en las que cada elemento ocupa una posición identificada por un índice (primer índice es el 0)
- Las listas admiten duplicados

```
public interface List extends Collection {
    Object get (int index);
    Object set (int index, Object element); // Opc
    void add (int index, Object element); // Opc
    Object remove (int index); // Opc
    abstract boolean addAll (int index, Collection c); // Opc
    int indexOf (Object o); // Búsqueda
    int lastIndexOf (Object o);
    ListIterator listIterator(); // Iteración
    ListIterator listIterator (int index);
    List subList (int from, int to); // Sub-listas
}
```



Programación Orientada a Objetos



La interfaz *List* – Implementaciones

- **ArrayList**
 - Usa un array extensible como implementación
 - Permite acceso aleatorio muy rápido a los elementos
 - Realiza con lentitud la inserción y el borrado de elementos en mitad de la lista
 - Se puede usar un `ListIterator` para moverse sobre la lista
- **LinkedList**
 - Basada en una lista doblemente enlazada
 - Operaciones `add` y `remove` en tiempo constante
 - Dispone de los métodos `addLast()`, `getFirst()`, `getLast()`, `removeFirst()` y `removeLast()`, que permiten utilizar esta clase como pila, cola o cola doble



La interfaz *List* – Implementaciones – *ArrayList*

```
ArrayList l1 = new ArrayList();
ArrayList l2 = new ArrayList();
l1.add("Pepe");      l1.add("Juan");      l1.add("Antonio");
l2.add("Maria");    l2.add("Ana");      l2.add("Susana");
```

```
l1.addAll(l2);
Iterator i = l1.iterator();
while (i.hasNext()){
    Object aux = i.next();
    System.out.print(aux + "\t");
}
```

Salida: Pepe Juan Antonio Maria Ana Susana



La interfaz *List* – Implementaciones – *LinkedList*

```
public class MiPila {
    private LinkedList list = new LinkedList();
    public MiPila(){ }
    public void push(Object o){ list.addFirst(o); }
    public Object top(){ return list.getFirst(); }
    public Object pop(){ return list.removeFirst(); }
}

public class Punto2D {
    private int x;
    private int y;
    public Punto2D(){
        Random rand = new Random();
        x = rand.nextInt();
        y = rand.nextInt();
    }
    public void imprimePunto() { System.out.println(x + " / " + y); }
}
```



La interfaz *List* – Implementaciones – *LinkedList*

```
MiPila s = new MiPila();
s.push (new Punto2D());
s.push (new Punto2D());
s.push (new Punto2D());

((Punto2D)s.pop()).imprimePunto();
((Punto2D)s.top()).imprimePunto();
((Punto2D)s.pop()).imprimePunto();
```



La interfaz *Map*

- Un *Map* es un objeto que asocia una clave a un valor
 - No contiene claves duplicadas
- También se denomina Diccionario
- Métodos para añadir y borrar:
 - `put(Object key, Object value)` / `remove(Object key)`
- Métodos para la extracción de objetos:
 - `get(Object key)`
- Métodos para obtener claves, valores y parejas (clave, valor) como conjuntos
 - `keySet()` // Devuelve un *Set*
 - `values()` // Devuelve una *Collection*
 - `entrySet()` // Devuelve un *Set* de *Map.entry*



Programación Orientada a Objetos



La interfaz *Map*

```
public interface Map {  
    Object put (Object key, Object value);  
    Object get (Object key);  
    Object remove (Object key);  
    boolean containsKey (Object key);  
    boolean containsValue (Object value);  
    int size();  
    boolean isEmpty();  
    void putAll (Map t);           // Operaciones masivas  
    void clear();  
    public Set keySet();          // Vistas de colecciones  
    public Collection values();  
    public Set entrySet();  
    public interface Entry { // Interfaz para los eltos de entrySet  
        Object getKey();  
        Object getValue();  
        Object setValue (Object value);  
    }  
}
```



Programación Orientada a Objetos



La interfaz *Map* – Implementaciones - *HashMap*

- Basado en una tabla Hash
- No se realiza ninguna ordenación en las parejas (clave, valor)
- Para cada clave tenemos un valor
- Si añadimos un elemento clave / valor cuando la clave ya existe, se sobrescribe el valor almacenado



Programación Orientada a Objetos



La interfaz *Map* – Implementaciones – *HashMap* – Ejemplo

```
Map hm = new HashMap();  
hm.put("C1","V1"); hm.put("C3","V3"); hm.put("C2","V2");  
Iterator it = hm.entrySet().iterator();  
while (it.hasNext()) {  
    Map.Entry e = (Map.Entry)it.next();  
    System.out.print(e.getKey() + " - " + e.getValue() + "\t");  
} // Salida:   C2 - V2       C3 - V3       C1 - V1  
  
hm.put("C2","V4");  
it = hm.entrySet().iterator();  
while (it.hasNext()) {  
    Map.Entry e = (Map.Entry)it.next();  
    System.out.print(e.getKey() + " - " + e.getValue() + "\t");  
} // Salida:   C2 - V4       C3 - V3       C1 - V1
```



Programación Orientada a Objetos



La interfaz *Map* – Implementaciones – *HashMap* – Ejemplo

```
Map hm = new HashMap();
hm.put("C1","V1");
hm.put("C3","V3");
hm.put("C2","V2");
System.out.println("Valores");
Iterator it2 = hm.values().iterator();
while (it2.hasNext()) {
    System.out.print((String)it2.next() + "\t");
}

System.out.println("Claves");
Iterator it3 = hm.keySet().iterator();
while (it3.hasNext()) {
    System.out.print((String)it3.next() + "\t");
}
```



Programación Orientada a Objetos



La interfaz *Map* – Implementaciones – *TreeMap*

- Basado en un árbol rojo-negro
- Implementa interfaz *SortedMap* → Las parejas (clave, valor) se ordenan sobre la clave



Programación Orientada a Objetos



La interfaz *Map* – Implementaciones – *TreeMap* - Ejemplo

```
Map hm = new HashMap();
hm.put("C1","V1");
hm.put("C3","V3");
hm.put("C2","V2");

TreeMap tm = new TreeMap(hm);
Iterator it = tm.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry e = (Map.Entry)it.next();
    System.out.println( e.getKey()+ "\t" + e.getValue());
}
```

C1	V1
C2	V2
C3	V3



Programación Orientada a Objetos



Interfaz *Iterator* – Interfaz *ListIterator*

- *Collection* define el método *iterator()* que devuelve un objeto que implementa la interfaz *Iterator* → Recorrido de las colecciones
- *ListIterator*:
 - Mejora de la interfaz *Iterator*
 - Permite:
 - Retroceder → `Object previous();`
 - Agregar → `void add(Object o);`
 - Devolver índices → `int previousIndex(); / int nextIndex();`



Programación Orientada a Objetos



Interfaz *Iterator* – Interfaz *ListIterator*

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove(); // Opc: Elimina el elemento del último next()
}
public interface ListIterator extends Iterator {
    boolean hasNext();      Object next();
    boolean hasPrevious(); Object previous();
    int nextIndex();       int previousIndex();
    void remove();        // Opcional
    // Sustituye el último elemento de next o previous por 'o'
    void set (Object o);   // Opcional
    // Inserta un elemento delante del próximo que se devolverá
    void add (Object o);   // Opcional
}
```



Programación Orientada a Objetos



Interfaz *ListIterator* - Ejemplo

```
public void eliminarNumerosPares(ArrayList l){
    Iterador i = l.iterator();
    while (i.hasNext()){
        Numero num = (Numero)i.next();
        if (num.esPar())
            i.remove();
    }
}

public Iterator iteradorInstantaneo(LinkedList l){
    return new ArrayList(l).iterator();
}
```



Programación Orientada a Objetos



Comparaciones

- Dado que ciertas colecciones trabajan ordenando sus elementos, necesitamos alguna forma de compararlos
- Existen dos soluciones posibles:
 - Interfaz *Comparable* → `public int compareTo (Object o)`
 - Interfaz *Comparator* → `public int compare (Object o1, Object o2)`
 - *Comparator* está pensado para implementarlo externo a la clase y *Comparable* para ser implementado en la clase
- Proporcionan un método que devuelve -1, 0 o 1 según el resultado de sea "menor", "igual" o "mayor"
- Si los objetos no son comparables entre si, se lanzará una excepción *ClassCastException*



Programación Orientada a Objetos



Comparaciones – *Comparable* - Ejemplo

```
public class Fraccion implements Comparable {
    private int numer;
    private int denom;
    ...
    public int compareTo (Object otro) {
        Fraccion otroR = (Fraccion) otro;
        if (n * otroR.d > d * otroR.n)      return 1;
        else if (n * otroR.d < d * otroR.n)  return -1;
        return 0;
    }
}
```



Programación Orientada a Objetos



Ejercicio 2 – Comparaciones - *Comparable*

- Una persona tiene un nombre y una edad. Utilizad el siguiente código para ordenar una colección de objetos *Persona* por nombre. Implementar la clase *Persona*

```
List<Persona> datos = new ArrayList();
datos.add(new Persona("Pepe", 33));
datos.add(new Persona("Jose", 22));
Collections.sort(datos);
Iterator it = datos.iterator();
while (it.hasNext()) {
    Persona p = (Persona)it.next();
    p.imprime();
}
```



Programación Orientada a Objetos



Comparaciones – *Comparator*

- Clase externa con la implementación de los métodos de la interfaz *Comparator*

```
public class MiClase implements Comparator{
    public int compare(Object o1, Object o2) {
        if (((Class)o1).getNum() == ((Class)o2). getNum())
            return 0;
        else if (((Class)o1). getNum() > ((Class)o2). getNum())
            return 1;
        return -1;
    }
    public boolean equals(Object obj) { return this == obj;}
}
```

- Para ordenar: `Collections.sort(lista, new MiClase());`



Programación Orientada a Objetos



Ejercicio 3 – Comparaciones - *Comparator*

- Realizar el ejercicio anterior de la clase *Persona*, implementando dos clases externas que contienen los métodos para poder comparar dos personas por:

- Su nombre
- Su edad

Utilizando para ello la interfaz *Comparator*

- Además, ¿qué sería necesario actualizar en el programa principal?



Programación Orientada a Objetos



Genéricos



Programación Orientada a Objetos



Genéricos – Aspectos Generales

- ¿Cómo defino clases, variables y parámetros genéricos?
- Solución 1: Uso de la clase *Object* como "comodín":
 - Mezcla de objetos de clases distintas en un mismo contenedor
 - Realización de castings (conversión de *Object* a otra clase)
- Solución 2: Utilizar *Generics* de Java
 - Se usa cuando el tipo de un dato no afecta al tratamiento del dato
 - Mecanismo similar en C++ o Ada
 - A partir de la versión 1.5



Programación Orientada a Objetos



Genéricos - Motivación

```
class ParNumeros{
    private int num1;
    private int num2;
    public ParNumeros (int a, int b){
        num1 = a;
        num2 = b;
    }
    public void intercambia(){
        int aux = num1;
        num1 = num2;
        num2 = aux;
    }
}

class ParCadenas{
    private String cad1;
    private String cad2;
    public ParCadenas (String a, String b){
        cad1 = a;
        cad2 = b;
    }
    public void intercambia(){
        String aux = cad1;
        cad1 = cad2;
        cad2 = aux;
    }
}
```



Programación Orientada a Objetos



Genéricos - Motivación

- Implementación igual → Abstracción a un tipo T

```
class Par<T>{
    private T e1;
    private T e2;
    public Par (T a, T b){
        e1 = a;
        e2 = b;
    }
    public void intercambia(){
        T aux = e1;
        e1 = e2;
        e2 = aux;
    }
}

Par<Integer> p1 = new Par<Integer>(1,2);
Par<String> p2 = new Par<String>("Hola", "Adios");
p1.intercambia();
p2.intercambia();
```



Programación Orientada a Objetos



Genéricos – Motivación

- ¿Sólo me sirve para enteros y cadenas de caracteres? → NO!

```
Par<Integer> p1 = new Par<Integer>(1,2);
Par<String> p2 = new Par<String>("Hola", "Adios");
Par<Character> p3 = new Par<Character>('a','b');
Persona per1 = new Persona("Juan", 1234);
Persona per2 = new Persona("Pepe", 5678);
Par<Persona> p4 = new Par<Persona>(per1, per2);
p1.intercambia();
p2.intercambia();
p3.intercambia();
p4.intercambia();
```



Programación Orientada a Objetos



Genéricos - Comparaciones

- Queremos añadir a nuestra clase Par un método que nos calcule el máximo de los dos elementos del par
- Posible Solución:

```
public T maximo(){ return ((a > b) ? a : b);}
```
- ERROR!→ Error de compilación: a y b no tienen por qué ser comparables con el operador mayor (p.e. objetos de la clase Persona)
- Solución correcta: Indicar al compilador que el tipo T sólo representará tipos de valores comparables con el símbolo mayor o una función equivalente (compareTo)



Genéricos - Comparaciones

- Clase Par:
 - ```
public class Par<T extends Comparable>
```
  - ```
public T maximo()
```
 - ```
{ return ((e1.compareTo(e2) > 0) ? e1 : e2);}
```
- Clase Persona:
  - ```
public class Persona implements Comparable
```
 - ```
public int compareTo(Object o) {
```
  - ```
Persona aux = (Persona) o;
```
 - ```
return nombre.compareTo(aux.nombre);
```
  - ```
}
```
- Programa principal:
 - ```
Persona res = p4.maximo(); → Pepe
```



## Généricos y Colecciones

```
ArrayList<Integer> lista = new ArrayList<Integer>();
for (int i = 1; i <= 10; i++)
 lista.add(i*10);

for (int num: lista)
 System.out.println(num);
```



## Ejercicio 4 – Généricos y Colecciones

- Partiendo de una clase Persona cuyos atributos son el nombre de la persona y el número de teléfono, realizar el código necesario tanto de la clase Persona como del programa principal, para:
  - Crear un ArrayList de Personas
  - Insertar objetos Persona
  - Ordenar la lista por el nombre de la persona (utilizada la interfaz Comparable)
  - Mostrar los datos de la lista ordenados

