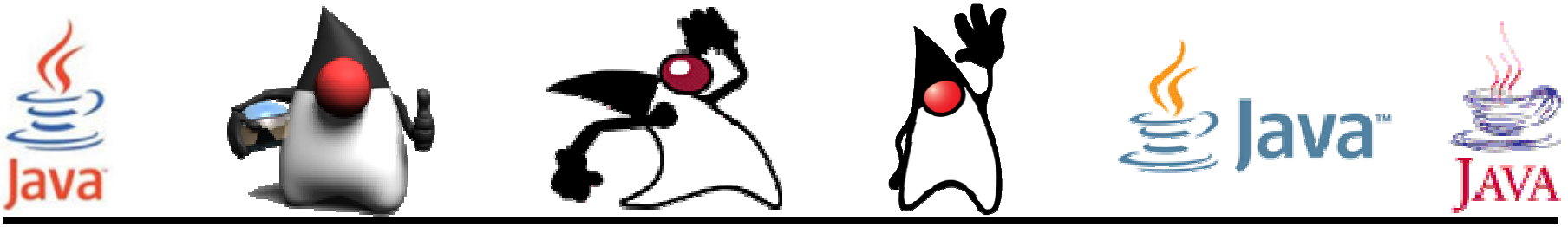


Collections

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Objetivos

- Conocer y justificar el uso de las clases Collection
- Aprender a usar colecciones

Índice

- Qué son y porqué las colecciones
- Núcleo de interfaces de colecciones
- Implementaciones
- Algoritmos

Qué es una colección?

- Un objeto "collection", también llamado contenedor, es un objeto que agrupa múltiples elementos en una sola unidad.
- Las colecciones se usan para almacenar, recuperar, manipular y comunicar datos agregados
  - Típicamente, representan datos que forman un grupo natural. Por ejemplo: colección de naipes, colección de cartas, lista de nombres y números (guía telefónica)

Qué es una estructura collection?

- Una estructura collection es una arquitectura unificada para representar y manipular colecciones
- Todas las estructuras de colecciones contienen lo siguiente:
  - Interfaces
  - Implementaciones
  - Algoritmos

Beneficios de la estructura collection

- Reduce el esfuerzo de programación
- Incrementa la velocidad y calidad de un programa
- Permite interoperabilidad entre APIs no relacionadas
- Reduce el esfuerzo para aprender y usar nuevas APIs
- Reduce el esfuerzo para diseñar nuevas APIs
- Fomenta la reutilización de software

Interfaces de la estructura collection

- Las interfaces Collection son tipos de datos abstractos que representan colecciones
- Las interfaces permiten a las colecciones ser manipuladas independientemente de los detalles de representación
  - Comportamiento polimórfico
- En Java, y otros lenguajes orientados a objetos, las interfaces forman una jerarquía
  - Uno selecciona la que cumple las necesidades como tipo

Implementaciones de la estructura collection

- Son las implementaciones concretas de las interfaces, que son estructuras de datos reusables
- Hay varios tipos de implementaciones:
  - Implementaciones de propósito general.
  - Implementaciones de propósito especial
  - Implementaciones concurrentes
  - Implementaciones envolventes
  - Implementaciones de conveniencia
  - Implementaciones abstractas

Implementaciones de propósito general

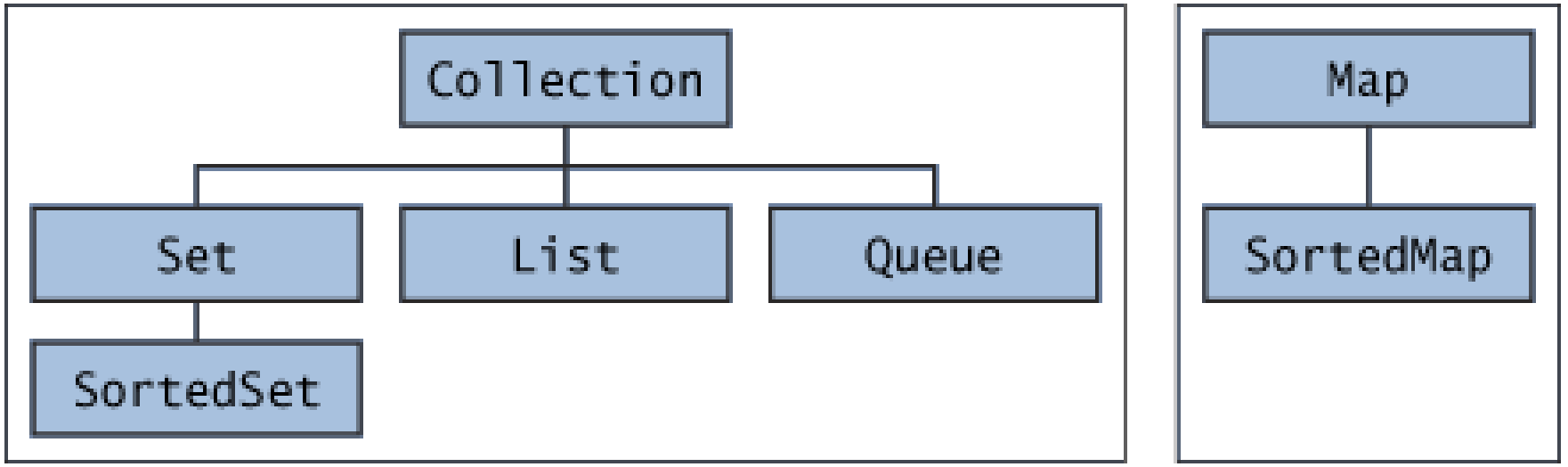
Interfaces	Implementaciones				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Algoritmos

- Son métodos polimórficos que realizan computaciones útiles en objetos que implementan las interfaces collection
- Los algoritmos más comunes son:
  - Ordenamiento (sorting)
  - Barajar (shuffling)
  - Manipulaciones de rutina (invertir, rellenar, copiar, intercambiar, añadir)
  - Búsqueda (searching)
  - Composición (frequency, disjoint)
  - Valores extremos (min, max, etc.)

Núcleo de Interfaces Collection

- El núcleo de interfaces Collection encapsula diferentes tipos de colecciones que permite la manipulación de colecciones independiente de los detalles de representación
- Jerarquía:



Núcleo de Interfaces Collection

- Todas las colecciones son genéricas. Ejemplo de declaración:

```
public interface Collection<E>...
```

donde E es el tipo de objeto contenido en la colección

- La interface Collection proporciona la funcionalidad básica común a las colecciones, tales como métodos add y remove.
- La interface Map crea una correspondencia las llaves y valores

Interface Collection

- Es la raíz de la jerarquía de colecciones
- Es el denominador común que todas las colecciones implementan
- Se usa para pasar colecciones de objetos (elementos) y manipularlos

Interface Collection

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();
    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional
    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Recorrido de colecciones: dos formas

- for-each
  - Sintaxis similar a for, sin acceso al elemento recorrido

```
for (Object o : collection)
    System.out.println(o);
```

- Iterator
  - Es un objeto que permite recorrer una colección y eliminar elementos de la colección de forma selectiva

```
public interface Iterator {
  boolean hasNext();
  Object next();
  void remove(); //optional
}
```

Iterator

- Este método se usa si se requiere eliminar elementos y se itera sobre colecciones múltiples en paralelo
  - El siguiente método muestra cómo usar Iterator para filtrar una Collection, eliminando elementos específicos

```
static void filter(Collection<?> c) {
  for (Iterator<?> it =c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
```

Operaciones masivas

- Son operaciones que realizan una operación sobre toda la colección

`containsAll()` retorna true si contiene todos los elementos de la Collection especificada

`addAll()` añade todos los elementos de la Collection especificada

`removeAll()` elimina de la Collection objetivo todos los elementos que también están contenidos en la Collection especificada

`retainAll()` elimina de la Collection objetivo todos los elementos que no están contenidos en la Collection especificada

`clear()` elimina todos los elementos de la Collection

Operaciones array

- Los métodos toArray son un puente entre collections y APIs antiguas que esperan arrays como entrada
- Las operaciones array permiten que el contenido de una Collection sea convertido a un array.
- Por ejemplo, si c es una Collection. Fragmento de código vuelca el contenido de c en un array de Object

```
Object[] a = c.toArray();
```

- Si c contiene strings (pe Collection<String>).

```
String[] a = c.toArray(new String[0]);
```

Otras interfaces del Core Collection

- `Set` collection que no puede contener elementos duplicados. Modela los conjuntos matemáticos
- `List` collection ordenada (secuencia) que puede contener duplicados. Se tiene control preciso de la inserción en la lista de cada elemento y se puede acceder por su índice entero (posición)
- `Queue` contiene múltiples elementos previo a su proceso y operaciones adicionales como inserción, extracción, e inspección. Típicamente se ordena los elementos como una lista FIFO (first-in, first-out)

Otras interfaces del Core Collection

- `Map` proyecta llaves a valores. Un Map no puede contener llaves duplicadas; cada llave puede proyectar como mucho un valor
- `SortedSet` es una versión ordenada de Set que mantiene los elementos en orden ascendente. Proporciona operaciones adicionales para aprovechar el orden
- `SortedMap` versión ordenada de Map, mantiene las proyecciones en orden ascendente de llaves

Implementaciones

- Las implementaciones son los objetos de datos usados para almacenar las colecciones, que implementan las interfaces
- El Java Collection Framework proporciona implementaciones de propósito general de las interfaces del núcleo:
  - Para la interface Set, HashSet es la más usada
  - Para la interface List, ArrayList es la más usada
  - Para la interface Map, HashMap es la más usada
  - Para la interface Queue, LinkedList es la más usada

Interface List: métodos

- add(i,o) Insert o at position i
- add(o) Append o to the end
- get(i) Return the i-th element
- remove(i) Remove the i-th element
- remove(o) Remove the element o
- set(i,o) Replace the i-th element with o

Interface Map: métodos

- Clear() Remove all mappings
- containsKey(k) Whether contains a mapping for k
- containsValue(v) Whether contains a mapping to v
- SetentrySet() Set of key-value pairs
- get(k) The value associated with k
- isEmpty() Whether it is empty
- keySet() Set of keys
- put(k,v) Associate v with k
- remove(k) Remove the mapping for k
- size() The number of pairs
- values() The collection of values

Algoritmos

- Son métodos estáticos proporcionados en la clase Collection. La mayoría operan con instancias List
- Algoritmos principales:
  - Ordenación (Sorting)
  - Barajado (Shuffling)
  - Manipulación de datos de rutina
  - Búsqueda (Searching)
  - Composición
  - Búsqueda de valores extremos

Ordenamiento

- El algoritmo de ordenamiento reordena una List en orden ascendente de acuerdo a una relación de orden
- Dos formas de operaciones
  - toma una lista y lo ordena de acuerdo al ordenamiento natural de los elementos
  - toma un Comparator además de la List y ordena los elementos según el Comparator

Ordenamiento natural

```
import java.util.*;

public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Ejecución:

```
> java Sort uno dos tres cuatro cinco
[cinco, cuatro, dos, tres, uno]
```

Ejemplo: Ordenamiento por Comparator

```
import java.util.*;
import java.io.*;

public class Anagrams2 {
    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);

        // Lee palabras desde fichero y los pone en un multimap simulado
        Map<String, List<String>> m = new HashMap<String, List<String>>();
        try {
            Scanner s = new Scanner(new File(args[0]));
            while (s.hasNext()) {
                String word = s.next(); String alpha = alphabetize(word);
                List<String> l = m.get(alpha);
                if (l == null)
                    m.put(alpha, l=new ArrayList<String>());
                l.add(word);
            }
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        }
```

Ejemplo: Ordenamiento por Comparator

```
// Construye lista de con grupos de permutacion por umbral de tamaño
        List<List<String>> winners = new ArrayList<List<String>>();
        for (List<String> l : m.values())
            if (l.size() >= minGroupSize)
                winners.add(l);

        // Ordena grupos de permutacion de acuerdo al tamaño
        Collections.sort(winners, new Comparator<List<String>>() {
            public int compare(List<String> o1, List<String> o2) {
                return o2.size() - o1.size();
            }});

        // Imprime grupos de permutacion
        for (List<String> l : winners ) {
            System.out.println(l.size() + ": " + l);
        }
    }

    private static String alphabetize(String s) {
   char[] a = s.toCharArray(); Arrays.sort(a); return new String(a);
    }
}
```

Ejemplo: Ordenamiento por Comparator

```
>java  Anagrams2 dictionary.txt 8
12: [apers, apres, asper, pares, parse, pears, prase, presa, rapes, reaps, spare
, spear]
11: [alerts, alters, artels, estral, laster, ratels, salter, slater, staler, ste
lar, talers]
10: [least, setal, slate, stale, steal, stela, taels, tales, teals, tesla]
9: [anestri, antsier, nastier, ratines, retains, retinas, retsina, stainer, stea
rin]
9: [capers, crapes, escarp, pacers, parsec, recaps, scrape, secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats, septal, staple, tepals]
9: [estrin, inerts, insert, inters, niters, nitres, sinter, triens, trines]
8: [enters, nester, renest, rentes, resent, tenser, ternes, treens]
8: [carets, cartes, caster, caters, crates, reacts, recast, traces]
8: [earings, erasing, gainers, reagins, regains, reginas, searing, seringa]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [arles, earls, lares, laser, lears, rales, reals, seral]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
8: [aspers, parses, passer, prases, repass, spares, sparse, spears]
```

Barajado (shuffling)

- El algoritmo shuffle hace lo opuesto de sort, destruyendo cualquier traza de orden que puede estar presente en List.
- El algoritmo produce un reordenamiento tal que todas las permutaciones ocurre con igual probabilidad
- El algoritmo tiene dos formatos:
  - toma una lista y usa una fuente de aleatoriedad por defecto
  - la otra requiere proporcionar una fuente de aleatoriedad basada en el objeto Random

Ejemplo: Barajado (shuffling)

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}

Ejecución:
>java  Shuffle uno dos tres cuatro cinco
[cuatro, uno, tres, cinco, dos]

>java  Shuffle uno dos tres cuatro cinco
[cuatro, dos, tres, uno, cinco]

>java  Shuffle uno dos tres cuatro cinco
[tres, uno, cinco, dos, cuatro]
```

Manipulación de datos de rutina

- La clase Collection proporciona cinco algoritmos para realizar manipulación de datos de rutina sobre objetos List.
  - `reverse` invierte los elementos de una List
  - `fill` sobreescribe cada elemento en List con el valor especificado
  - `copy` copia los elementos de una List fuente en otra de destino
  - `swap` intercambia los elementos especificados en una List
  - `addAll` añade todos los elementos a una Collection

Búsqueda

- El algoritmo binarySearch realiza una búsqueda sobre una List ordenada
- Tiene dos formatos:
  - El primero toma una List y un elemento a buscar (clave). Esta formato asume que List está en orden ascendente
  - El segundo formato toma un Comparator además de la List y la clave, y asume que List está ordenado de forma ascendente de acuerdo al Comparator especificado
- El valor retornado es el índice del elemento, sino un elemento negativo

Ejemplo: búsqueda binaria

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

public class BinarySearchArrayListExample {
  public static void main(String args[]) {
    ArrayList arrayList = new ArrayList();
    // inserta elementos
    arrayList.add("uno"); arrayList.add("dos");
    arrayList.add("tres"); arrayList.add("cuatro");
    arrayList.add("cinco"); arrayList.add("seis");
    arrayList.add("siete");
    Collections.sort(arrayList); // Ordenacion
    System.out.println("Lista ordenada: " + arrayList);
    // Busqueda de elemento en la lista
    int index = Collections.binarySearch(arrayList, "cinco");
    System.out.println("Elemento encontrado en posicion " + index);
    // Busqueda de elemento no en la lista
    index = Collections.binarySearch(arrayList, "ocho");
    System.out.println("No encontrado " + index);
  }
}
```

Composición

- Los algoritmos de composición comprueban algún aspecto de la composición de una o más Collections
  - Collections.frequency(l) – cuenta el número de veces que los elementos especificadas ocurren en la colección especificada
  - Collections.disjoint(l1, l2) – determina si dos Collections son disjuntos, esto es, si no hay elementos comunes

Búsqueda de valores extremos

- Los algoritmos max y min retornan, respectivamente, el elemento máximo y mínimo contenido en una Collection especificada.
- Ambas operaciones tienen dos formatos:
  - La forma más simple toma una Collection y retorna el máximo (mínimo) de acuerdo al orden natural de los elementos.
  - El segundo formato toma un Comparator además de la Collection y retorna el máximo (mínimo) de acuerdo al orden especificado en Comparator

Ejemplo: Cuenta número de palabras y palabras diferentes en un texto de entrada

```
import java.util.*;
import java.io.*;
public class CountWords {
  static public void main(String[] args) {
    HashSet words = new HashSet();
    BufferedReader in = new BufferedReader(
                        new InputStreamReader(System.in));
    String delim = " \t\n.,:;?!-/()[]\"\'"; String line; int count = 0;
    try {
      while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line, delim);
        while (st.hasMoreTokens()) {
          count++;
          words.add(st.nextToken().toLowerCase());
        }
      }
    } catch (IOException e) {}
    System.out.println("Numero total de palabras: " + count);
    System.out.println("Numero de palabras diferentes: " + words.size());
  }
}
```

Ejemplo: Cuenta número de ocurrencias de cada palabra

```
import java.util.*;
import java.io.*;
public class WordFrequency {
  static public void main(String[] args) {
    HashMap words = new HashMap(); String delim = " \t\n.,:;?!-/()[]\"\'";
    BufferedReader in = new BufferedReader(new
                        InputStreamReader(System.in));
    String line, word; Count count;
    try {
      while ((line = in.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(line, delim);
      while (st.hasMoreTokens()) {
        word = st.nextToken().toLowerCase();
        count = (Count) words.get(word);
        if (count == null) {
          words.put(word, new Count(word, 1));
        } else { count.i++;
      }
     }
    }
   } catch (IOException e) {}
```

Ejemplo: Cuenta número de ocurrencias de cada palabra

```
    Set set = words.entrySet();
    Iterator iter = set.iterator();
    while (iter.hasNext()) {
      Map.Entry entry = (Map.Entry) iter.next();
      word = (String) entry.getKey();
      count = (Count) entry.getValue();
      System.out.println(word +
                  (word.length() < 8 ? "\t\t" : "\t") +
                  count.i);
    }
  }

  static class Count {
    Count(String word, int i) {
      this.word = word;
      this.i = i;
    }
    String word;
    int i;
  }
}
```

Ejemplo: Cuenta ocurrencias de cada palabra e imprime en orden alfabético

```
import java.util.*;
import java.io.*;
public class WordFrequency2 {
  static public void main(String[] args) {
    TreeMap words = new TreeMap(); String delim = " \t\n.,:;?!-/()[]\"\'";
    BufferedReader in = new BufferedReader(new
                        InputStreamReader(System.in));
    String line, word; Count count;
    try {
      while ((line = in.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(line, delim);
      while (st.hasMoreTokens()) {
        word = st.nextToken().toLowerCase();
        count = (Count) words.get(word);
        if (count == null) {
          words.put(word, new Count(word, 1));
        } else { count.i++;
      }
     }
    }
   } catch (IOException e) {}
```

Ejemplo: Cuenta ocurrencias de cada palabra e imprime en orden alfabético

```
    Set set = words.entrySet();
    Iterator iter = set.iterator();
    while (iter.hasNext()) {
      Map.Entry entry = (Map.Entry) iter.next();
      word = (String) entry.getKey();
      count = (Count) entry.getValue();
      System.out.println(word +
                  (word.length() < 8 ? "\t\t" : "\t") +
                  count.i);
    }
  }

  static class Count {
    Count(String word, int i) {
      this.word = word;
      this.i = i;
    }
    String word;
    int i;
  }
}
```

Ejemplo: Cuenta ocurrencias de cada palabra e imprime en orden de frecuencia

```
import java.util.*;
import java.io.*;
public class WordFrequency4 {
  static public void main(String[] args) {
    Map words = new HashMap(); String delim = " \t\n.,:;?!-/()[]\"\'";
    BufferedReader in = new BufferedReader(new
                        InputStreamReader(System.in));
    String line, word; Count count;
    try {
      while ((line = in.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(line, delim);
      while (st.hasMoreTokens()) {
        word = st.nextToken().toLowerCase();
        count = (Count) words.get(word);
        if (count == null) {
          words.put(word, new Count(word, 1));
        } else { count.i++;
      }
     }
    }
   } catch (IOException e) {}
```

Ejemplo: Cuenta ocurrencias de cada palabra e imprime en orden de frecuencia

```
    List list = new ArrayList(words.values());
    Collections.sort(list, new CountComparator());
    Iterator iter = list.iterator();
    while (iter.hasNext()) {count = (Count)iter.next(); word = count.word;
      System.out.println(word +
                  (word.length() < 8 ? "\t\t" : "\t") + count.i);
    }
  }
  static class Count {
    Count(String word, int i) {
      this.word = word; this.i = i;
    }
    String word; int i;
  }
  static class CountComparator implements Comparator {
    public int compare(Object o1, Object o2) {
      if (o1 != null && o2 != null && o1 instanceof Count &&
          o2 instanceof Count) {
         Count c1 = (Count) o1; Count c2 = (Count) o2;
         return (c2.i - c1.i);
      } else { return 0; } } } }
```