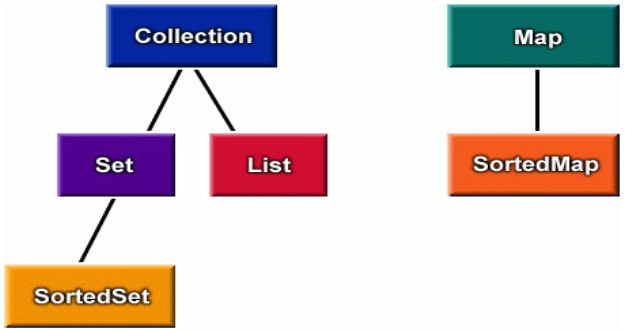
Colecciones

- En Java 1.2 se introduce un nuevo marco para las colecciones. (En el paquete `java.util`)
- Basado en STL de C++
  - Interfaces.
    - Permiten manipular una colección independientemente de los detalles de implementación.
  - Implementación
    - La estructura real
  - Algoritmos
    - Métodos que permiten realizar cómputos

Beneficios de utilizar el marco de colecciones

- Reduce los esfuerzos de programación
- Incrementa velocidad y calidad
- Ayuda a la interoperabilidad
- Reduce los esfuerzos de aprendizaje
- Reduce los esfuerzos de diseño
- Fomenta la reutilización de software

Interfaces para colecciones



Interfaces para colecciones

- Permiten manipular las estructuras
- Hay operaciones opcionales, es decir, una implementación concreta pudiera no tenerla.
- Todas las colecciones del JDK implementan todas las operaciones opcionales
- Si se invoca una operación no soportada se lanza la excepción `UnsupportedOperationException`

Interface `Collection`

- Proporciona un protocolo mínimo para una colección.
- Suelen usarse `Set` y `List` que son interfaces más especializados

5

Interface `Set`, `List` y `SortedSet`

- `Set`
  - No permite tener elementos duplicados
- `SortedSet`
  - Versión de `Set` que mantiene ordenados a los elementos
- `List`
  - También llamado secuencia
  - Controla dónde se inserta cada elemento
  - Es muy parecido al antiguo `Vector`

6

Interface `Map` y `SortedMap`

- `Map`
  - Empareja claves con valores
  - Cada clave puede emparejarse con a lo sumo un valor
  - No puede haber claves duplicadas
  - Muy parecido al antiguo `Hashtable`
- `SortedMap`
  - Versión de `Map` que mantiene las claves ordenadas

7

Interface `Collection`

```
public interface Collection {
   // Operaciones basicas
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element);       // Opcional
   boolean remove(Object element);    // Opcional
   Iterator iterator();
   // Operaciones completas
   boolean containsAll(Collection c);
   boolean addAll(Collection c);      // Opcional
   boolean removeAll(Collection c);   // Opcional
   boolean retainAll(Collection c);   // Opcional
   void clear();                      // Opcional
   // Operaciones con Arrays
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```

8

Interface `Iterador`

- Tiene una conducta mejorada que la del antiguo `Enumeration`
- Ahora se permite quitar elementos de la colección con el iterador

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

9

Un ejemplo de `Iterator`

- Filtrar los elementos que no cumplen una condición

```
static void filtro(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (!cond(i.next()))
            i.remove();
}
```

- Es un código genérico para cualquier colección que admita la eliminación de elementos

10

Interfaces e implementación

- Cada interface puede estar implementada por más de una clase
  - Podemos utilizar una referencia a la interface para señalar a un objeto de una clase concreta

```
Collection c = new ArrayList();
```

  - Todas estas clases tienen un constructor que admite un objeto que implemente la interface `Collection`

```
Collection d = new ArrayList(c);
```

  - ¿Cómo actúan estos constructores?

11

Interface `Set` I

- La interface es igual a la de `Collection`
- No permite elementos duplicados
  - El control se hace por el método `equals()`
- Hay dos implementaciones en JDK
  - `HashSet`
    - Guarda los datos en una tabla hash
  - `TreeSet`
    - Guarda los datos en un árbol

12

Interface `Set` II

```
public interface Set {
   // Operaciones basicas
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element);       // Optional
   boolean remove(Object element);    // Optional
   Iterator iterator();
   // Operaciones completas
   boolean containsAll(Collection c);
   boolean addAll(Collection c);      // Optional
   boolean removeAll(Collection c);   // Optional
   boolean retainAll(Collection c);   // Optional
   void clear();                      // Optional
   // Operaciones con Arrays
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```


Ejemplo de Interface `Set`

```
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
                    (" duplicado: "+args[i]);

        System.out.println(s.size()+
                    " palabras detectadas: "+s);
    }
}
```


Un ejemplo de uso de `Set`

- ¿Cómo eliminar de una colección `c` todos los elementos duplicados?

```
Collection noDups = new HashSet(c);
```

Ejercicios.
- ¿Cómo hacer la unión, intersección y diferencia de conjuntos?
- Crear un programa como el anterior pero que mantenga un conjunto con las repetidas y otro con las no repetidas
- Llaves y cerraduras

Interface `List` I

- Una colección de elementos también llamado secuencia
  - Puede haber elementos repetidos
  - Acceso a la posición
  - Devuelve la posición de un elemento buscado
  - Permite iterar de una forma más especializada (Con un `ListIterator`)
  - Realiza operaciones con rango de índices

Interface `List` II

- Hay dos implementaciones en JDK
  - `ArrayList`
    - Consultas en cualquier posición
    - Añadir por los extremos
  - `LinkedList`
    - Consultas por los extremos
    - Añadir en cualquier posición
- El antiguo `Vector` implementa esta interface

Interface `List` III

```
public interface List extends Collection {
    // Acceso Posicional
    Object get(int index);
    Object set(int index, Object element);          // Opt.
    void add(int index, Object element);            // Opt.
    Object remove(int index);                       // Opt.
    abstract boolean addAll(int index, Collection c);// Opt.

    // Busqueda
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteracion
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Vista de rango
    List subList(int from, int to);
```


Ejemplo de interface `List`

```
private static void swap(List a, int i, int j) {
    Object tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
public static void shuffle(List list, Random rnd) {
    for (int i=list.size(); i>1; i--)
        swap(list, i-1, rnd.nextInt(i));
}
    import java.util.*;
    public class Shuffle {
        public static void main(String args[]) {
            List l = new ArrayList();
            for (int i=0; i<args.length; i++)
                l.add(args[i]);
            Collections.shuffle(l, new Random());
            System.out.println(l);
        }
    }
```


Un array visto como una lista

- En `java.util.Arrays` existe el método

```
static List asList(array)
```

```
import java.util.*;

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

Interface `ListIterator`

```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove();          // Optional
    void set(Object o);     // Optional
    void add(Object o);     // Optional
}
```


Algoritmos de la clase `Collections` aplicados a `List`

- Métodos de clase

```
sort(List)
suffle(List)
reverse(List)
fill(List,Object)
copy(List dest, List src)
binarySearch(List,Object)
```


Métodos especiales de `LinkedList`

- Permiten el acceso por los extremos

```
void addFirst(Object)
void addLast(Object)
Object getFirst()
Object getLast()
Object removeFirst()
Object removeLast()
```

- Para poder utilizarse

```
LinkedList l = new LinkedList();
```

  - Implementar una clase Pila y una clase Cola

Interface `Map` I

- Mantiene claves y valores asociados
- Las claves no pueden repetirse.
  - Se controla por `equals()`
- A lo sumo puede haber un valor por clave
- Hay dos implementaciones en JDK
  - `HashMap`
  - `TreeMap`
- La antigua `Hashtable` implementa el interfaz Map

Interface `Map` II

```
public interface Map {
    // Operaciones basicas
    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();

    // Operaciones completas
    void putAll(Map t);
    void clear();

    ...
```


Interface `Map` III

```
....

// Vista como colecciones
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface para los elementos de entrada
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```


Ejemplo de uso de `Map`

```
import java.util.*;
public class Frecuencia {
    private static final Integer UNO = new Integer(1);

    public static void main(String args[]) {
       Map m = new HashMap();
       // Inicializa la tabla desde la linea de cmd
       for (int i=0; i<args.length; i++) {
           Integer freq = (Integer) m.get(args[i]);
           m.put(args[i],
                 (freq==null ? UNO :
                  new Integer(freq.intValue() + 1)));
       }
       System.out.println(m.size()+" pal. distintas");
       System.out.println(m);
    }
}
```


Ordenación I

- **Orden natural** de algunos tipos:

Class	Natural Ordering
`Byte`	signed numerical
`Character`	unsigned numerical
`Long`	signed numerical
`Integer`	signed numerical
`Short`	signed numerical
`Double`	signed numerical
`Float`	signed numerical
`String`	lexicographic
`Date`	chronological

Ordenación II

- Si se intenta ordenar un tipo que no sabe ordenarse se produce una excepción
  - `ClassCastException`
  - Es posible definir una ordenación para cualquier tipo. Dos maneras:
    - Definiendo en la clase la interface `Comparable` (Ordenación natural)
      - `int compareTo(Object o)`
    - Definiendo la interface `Comparator`
      - `int compare(Object o1, Object o2)`

Ordenación por `Comparable`

```
public interface Comparable {
    public int compareTo(Object o);
}
```

- Define el **orden natural** para los objetos de una clase

positivo	Si receptor > o
0	Si recentor = o
negativo	Si receptor < o

  - Si no es posible se produce la excepción:
    `ClassCastException`
- Este método no debe entrar en contradicción con `equals()`

Ordenación Natural

- Los objetos de una clase que implementen un orden natural pueden utilizarse para:
  - Elementos en un set ordenado (`SortedSet`)
  - Claves en un map ordenado (`SortedMap`)
  - En listas que utilicen el método `Collection.sort()`
- Todos las clases envoltorios (wrapper) tienen definido un orden natural

Ejemplo de `Comparable`

```
import java.util.*;
public class Persona implements Comparable {
    private String  nombre;
    private int     edad;
    public Name(String nombre, int edad) {
        if (nombre==null)
            throw new NullPointerException();
        this.nombre = nombre;
        this.edad   = edad;
    }
    public int compareTo(Object o) {
        Nombre n = (Nombre)o;
        if (edad > n.edad) return 1
        else if (edad == n.edad) return 0
        else return -1;
    }
}
```

Ordenación por `Comparator`

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

- Existen versiones de todas las funciones de ordenación que toman un argumento más:
  - Un objeto que implementa la interface `Comparator`
- Son válidas las consideraciones hechas anteriormente

Interface `SortedSet` I

- Es un `Set` que mantiene sus elementos ordenados en orden ascendente.
  - El orden es:
    - Natural o
    - Definido en una `Comparator` en el momento de la creación
  - Por mantener los datos ordenados puede proporcionar nuevos métodos
  - Lo implementa `TreeSet`

Interface `SortedSet` II

```
public interface SortedSet extends Set {
 // Vistas de rangos
 SortedSet subSet(Object fromElement, Object toElement);
 SortedSet headSet(Object toElement);
 SortedSet tailSet(Object fromElement);

 // elementos finales
 Object first();
 Object last();

 // acceso al Comparator
 Comparator comparator();
}
```


Interface `SortedSet` III

- Para las colecciones que implemente la interface `SortedSet` hay dos nuevos constructores
  - Uno toma un objeto que implemente la interface `Comparable` y devuelve un `SortedSet` vacío
  - Otro toma un `SortedSet` y devuelve otro `SortedSet` con los mismos elementos y en el mismo orden

Interface `SortedMap` I

- Es muy parecido a `SortedSet`
- Un `SortedMap` es un `Map` que mantiene sus claves en orden (natural o dado en la creación)
- Añade nuevos métodos
- Lo implementa `TreeMap`

Interface `SortedMap` II

```
public interface SortedMap extends Map {
    Comparator comparator();

    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    Object first();
    Object last();
}
```


Ejemplo de `SortedMap`

```
import java.util.*;
public class Frecuencia {
    private static final Integer UNO = new Integer(1);

    public static void main(String args[]) {
       SortedMap m = new HashMap();
       // Inicializa la tabla desde la linea de cmd
       for (int i=0; i<args.length; i++) {
           Integer freq = (Integer) m.get(args[i]);
           m.put(args[i],
                 (freq==null ? UNO :
                  new Integer(freq.intValue() + 1)));
       }
       System.out.println(m.size()+" pal. distintas");
       System.out.println(m);
    }
}
```


Decoradores

- Clases envoltorios que añaden funcionalidad a las colecciones
  - De solo lectura
  - De acceso sincronizado
  - Para crear colecciones unitarias

Colecciones de solo lectura

- Si se intenta una operación de modificación

  `UnsupportedOperationException`

```
Collection unmodifiableCollection(Collection collection)
List unmodifiableList(List list)
Map unmodifiableMap(Map map)
Set unmodifiableSet(Set set)
SortedMap unmodifiableSortedMap(SortedMap map)
SortedSet unmodifiableSortedSet(SortedSet set)
```

41

Implementaciones

		Implementaciones			
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**
Interfaces	**Set**	HashSet		TreeSet	
	SortedSet			TreeSet	
	List		ArrayList Vector		LinkedList
	Map	HashMap HashTable		TreeMap	
	SortedMap			TreeMap	

42