

Jerarquías de clases

Definición básica de subclases

```
class Persona {  
    String nombre;  
    int edad;  
    String asString () {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}  
  
class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
}  
  
class Directivo extends Empleado {  
    int categoria;  
    ArrayList<Empleado> equipo = new ArrayList<Empleado> ();  
    void promocionar () { categoria++; }  
}
```

2

Jerarquía de tipos (I)

```
Persona x1, x2;  
Empleado y, emp = new Empleado ();  
Directivo z, dir = new Directivo ();
```

- Conversión automática implícita (generalización)

```
x1 = emp; // Empleado → Persona  
x2 = dir; // Directivo → Persona  
y = dir; // Directivo → Empleado
```

Un Directivo puede hacer automáticamente el papel de Empleado y de Persona

- Conversión explícita (especialización), responsabilidad del programador

```
z = x2; // Error de compilación  
z = (Directivo) x2; // Persona → Directivo  
z = (Directivo) x1; // Error de ejecución:  
// x1 no es un Directivo  
z = (Perro) x1; // Error de compilación (a menos que Perro  
// fuese subclase o superclase de Persona)
```

Una Persona puede hacer el papel de Directivo si realmente es un Directivo

Jerarquía de tipos (II)

```
class C {  
    void f (Empleado p) { ... }  
    void g (Directivo p) { ... }  
}
```

```
Directivo dir = new Directivo ();  
Empleado x = dir, emp = new Empleado ();  
C c = new C ();
```

- Conversión implícita

```
c.f (dir); // Directivo → Empleado
```

- Conversión explícita

```
c.g (x); // Error de compilación  
c.g ((Directivo) x); // Empleado → Directivo  
c.g ((Directivo) emp); // Error de ejecución: emp no es  
// un Directivo
```


Sobreescritura de variables y métodos

- Redefinición de variables y métodos de una clase padre en una subclase
- La definición de la subclase ensombrece a la de la superclase
- La definición de la superclase es accesible desde la subclase con **super**
- Sobreescritura de métodos (especialización)
 - El método se redefine con los mismos argumentos y tipo de retorno
 - **Nota:** desde Java 1.5, el tipo de retorno puede ser una subclase del original
 - Si no coinciden los tipos de los argumentos, se trata de una sobrecarga
 - No se puede aumentar la privacidad de los métodos sobreescritos
 - Evita la proliferación de identificadores
 - Permite la ligadura dinámica
- Sobreescritura de variables
 - Se reserva un espacio de memoria para cada definición
 - El tipo no tiene por qué coincidir
 - Ligadura estática
 - En general es preferible evitar la sobreescritura de variables

7

Sobreescritura de variables

```
class Musico extends Persona {
    String nombre;
    void mostrarNombres () {
        System.out.println ("Musico: " + nombre);
        System.out.println ("Persona: " + super.nombre);
    }
}

// Bloque main
Musico m = new Musico ();
Persona p = m;
m.nombre = "Stevie Wonder"; // nombre de Musico
p.nombre = "Stevland Morris"; // nombre de Persona
```

8

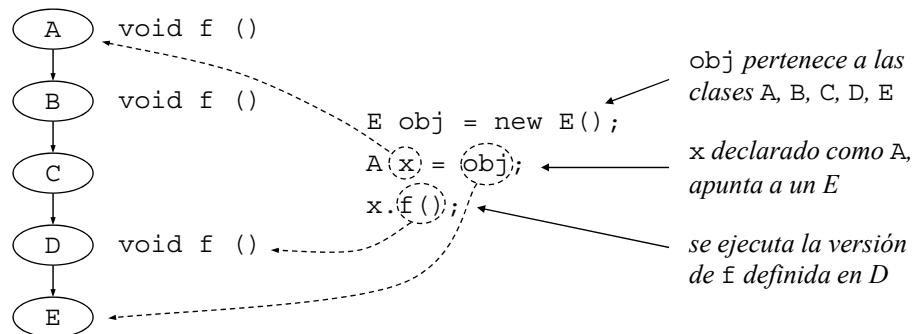
Sobreescritura de métodos

```
class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    String asString () {  
        return "Nombre: " + nombre + "\nEdad: " + edad +  
            "\nSueldo: " + sueldoBruto + "\nJefe: " +  
            ((jefe == null)? nombre : jefe.nombre);  
    }  
}  
  
// Bloque main  
Empleado emp = new Empleado ();  
Persona p = emp;  
emp.asString (); // asString de Empleado  
p.asString ();   // asString de Empleado
```

9

Ligadura dinámica

- La sobreescritura de métodos se resuelve por ligadura dinámica en tiempo de ejecución
- Se ejecuta la definición del método de la clase más específica del objeto, independientemente de cómo se ha declarado la referencia al objeto



- Los métodos estáticos tienen ligadura estática

10

Ligadura dinámica: ejemplo

```
class Persona {
    String nombre;
    int edad;
    String asString () {
        return "Nombre: " + nombre + "\nEdad: " + edad;
    }
}

class Empleado extends Persona {
    long sueldoBruto;
    Directivo jefe;
    String asString () {
        return "Nombre: " + nombre + "\nEdad: " + edad +
            "\nSueldo: " + sueldoBruto + "\nJefe: " +
            ((jefe == null)? nombre : jefe.nombre);
    }
}
```

11

```
class Directivo extends Empleado {
    int categoria;
    ArrayList<Empleado> equipo = new ArrayList<Empleado> ();
    String asString () {
        return "Nombre: " + nombre + "\nEdad: " + edad +
            "\nSueldo: " + sueldoBruto + "\nJefe: " +
            ((jefe == null)? nombre : jefe.nombre) +
            "\nCategoria: " + categoria;
    }
    void promocionar () { categoria++; }
}
```

12

```
// Bloque main
Directivo dir = new Directivo ();
Empleado emp = new Empleado ();
Empleado z = dir;
Persona p = new Persona ();
Persona x = emp;
Persona y = z;

p.asString (); // asString de Persona
emp.asString (); // asString de Empleado
dir.asString (); // asString de Directivo
x.asString (); // asString de Empleado
y.asString (); // asString de Directivo
z.asString (); // asString de Directivo
y.promocionar (); // ERROR
```

13

La ligadura respecto a los argumentos es estática

```
class A {
    void f (Persona p) {
        System.out.println ("Clase Persona");
    }
    void f (Empleado emp) {
        System.out.println ("Clase Empleado");
    }
}
```

Se ejecuta la definición compatible más específica

```
// Bloque main
A a = new A ();
Directivo dir = new Directivo (...);
Persona p = dir;
a.f (dir);
a.f (p);
Animal x = p;
a.f (x); // ERROR
```

Suponiendo que Persona es subclase de Animal

14

Herencia y constructores

Los constructores no se heredan ni se sobrescriben

- Al crear un Empleado, se invoca al constructor de Persona
- Invocación automática implícita
 - Se invoca al constructor de la clase padre sin argumentos
 - Si no está definido → error
- Invocación explícita
 - Invocación a `super (...)` en la primera línea del constructor de Empleado
- Invocación a otros constructores de la misma clase: `this (...)`

15

Herencia y constructores: ejemplo

```
class Persona {
    String nombre;
    int edad;
    Persona (String str, int i) {
        nombre = str;
        edad = i;
    }
    String asString () {
        ...
    }
}

// Error al crear un Empleado: el constructor por
// defecto Empleado () invoca al constructor por
// defecto Persona (), que ya no está definido
```

16

```
class Empleado extends Persona {
    long sueldoBruto;
    Directivo jefe;
    Empleado (String str, int i, long sueldo, Directivo dir) {
        nombre = str; } → super (str, i);
        edad = i;
        sueldoBruto = sueldo;
        jefe = dir;
    }
    String asString () {
        ...
    }
}

// Error al crear un Empleado: se sigue invocando
// automáticamente a Persona () por defecto

// Error al crear un Directivo: el constructor por
// defecto Directivo () invoca al constructor por
// defecto Empleado (), que ya no está definido
```

17

```
class Empleado extends Persona {
    long sueldoBruto;
    Directivo jefe;
    Empleado (String str, int i, long sueldo, Directivo dir) {
        super (str, i);
        sueldoBruto = sueldo;
        jefe = dir;
    }
    String asString () {
        ...
    }
}
```

18

```
class Directivo extends Empleado {
    int categoria;
    ArrayList<Empleado> equipo
        = new ArrayList<Empleado> ();
    Directivo (String str, int i, long n, int j) {
        this (str, i, n, null, j);
    }
    Directivo (String str, int i, long n,
        Directivo dir, int j) {
        super (str, i, n, dir);
        categoria = j;
    }
    String asString () {
        ...
    }
    void promocionar () { categoria++; }
}
```

Siempre en la primera línea

19

Control de acceso: private

```
class Persona {
    private String nombre;
    private int edad;
    String asString () {
        return "Nombre: " + nombre + "\nEdad: " + edad;
    }
}

class Empleado extends Persona {
    long sueldoBruto;
    Directivo jefe;
    String asString () {
        return "Nombre: " + nombre + "\nEdad: " + edad +
            "\nSueldo: " + sueldoBruto + "\njefe: " +
            ((jefe == null)? nombre : jefe.nombre);
    }
}
```

Error: nombre y edad son privados

20

Control de acceso: protected (I)

```
class Persona {
    protected String nombre;
    protected int edad;
    protected String asString () {
        return "Nombre: " + nombre + "\nEdad: " + edad;
    }
}

class Empleado extends Persona {
    long sueldoBruto;
    Directivo jefe;
    String asString () {
        return super.asString () +
            "\nSueldo: " + sueldoBruto + "\nJefe: " +
            ((jefe == null)? nombre : jefe.nombre);
    }
}
```

Error: sólo puede ser protected o public

Correcto incluso si Empleado y Persona en distinto package

En distinto package habría error si jefe fuera Persona pero no Empleado

21

Control de acceso: protected (II)

```
package personal;

public class Persona {
    protected String nombre;
    protected int edad;
    protected String asString () {
        return "Nombre: " + nombre + "\nEdad: " + edad;
    }
}
```

```
package personal;

...
// En cualquier clase
Persona p = new Persona ();
p.asString ();
...
```

```
package X;

...
// En cualquier clase
personal.Persona p =
    new personal.Persona ();
p.asString (); // Error
...
```

Control de acceso: protected (III)

```
package p1;  
  
public class A {  
    int w; // package  
    private int x;  
    protected int y;  
    public int z;  
}
```

```
package p2;  
  
class C {  
    void h () {  
        p1.A a = new p1.A ();  
        a.w = 2; // Error  
        a.x = 6; // Error  
        a.y = 8; // Error  
        a.z = 3;  
    }  
}  
  
class D extends p1.A {  
    void h () {  
        p1.A a = new p1.A ();  
        w = 2; a.w = 2; // Error  
        x = 2; a.x = 6; // Error  
        z = 3; a.z = 3;  
        a.y = 8; // Error  
        y = 8;  
        D d = new D ();  
        d.y = 8;  
    }  
}
```

Clases final y miembros final

```
final class A {  
}  
  
class B extends A { // Error: A no se puede derivar  
    final int x; // Error: obligatorio inicializar x  
    final double f (int x) {  
        return (x-1)/(x+1);  
    }  
}  
  
class C extends B {  
    int x;  
    double f (int x) { // Error: f no se puede sobrescribir  
        return (x-2)/(x+2);  
    }  
}
```

Abstracción de clases e interfaces

Clases abstractas, métodos abstractos

- Clases abstractas
 - No se pueden crear objetos
`new A ()` → ERROR si A abstracta
 - Se pueden definir subclases
- Métodos abstractos
 - Métodos sin código, se declaran pero no se definen
 - Deben definirse en alguna subclase
- Una clase abstracta puede tener métodos no abstractos
- Un método abstracto debe pertenecer a una clase abstracta
- Si una subclase no implementa un método abstracto heredado, debe ser abstracta también

26

Clases abstractas: ejemplo

```
abstract class Figura {
    abstract double area ();
}

class Circulo extends Figura {
    Punto2D centro; double radio;
    double area () {
        return Math.PI * radio * radio;
    }
}

class Triangulo extends Figura {
    Punto2D a, b, c;
    double area () {
        return Math.abs ((b.x-a.x)*(c.y-a.y) -
            (b.y-a.y)*(c.x-a.x))/2;
    }
}
```

27

Utilidad de los métodos abstractos

```
class Grupo {
    private List<Figura> figuras = new ArrayList<Figura>();
    void agregarFigura (Figura fig) {
        figuras.add (fig);
    }
    double area () {
        double a = 0;
        for (Figura f : figuras)
            a += f.area ();
        return a;
    }
}
```

28

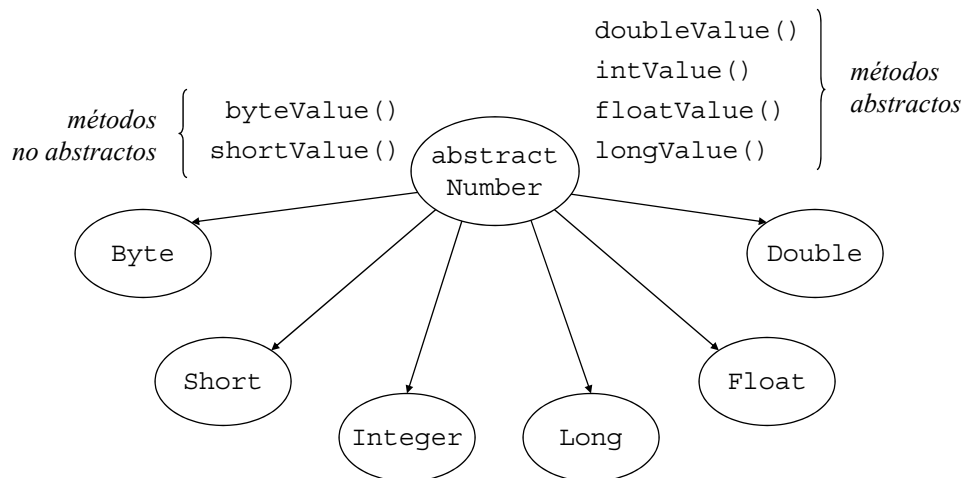
Métodos heredados abstractos

```
abstract class Figura {  
    abstract double area ();  
}  
class Poligono extends Figura {  
    // No se implementa area ()  
}  
class Triangulo extends Poligono {  
    Punto2D a, b, c;  
    double area () {  
        return Math.abs ((b.x-a.x)*(c.y-a.y) -  
                           (b.y-a.y)*(c.x-a.x))/2;  
    }  
}
```

Error: Poligono tiene que ser abstract

29

java.lang.Number es una clase abstracta



30

Interfaces

Motivación

```
// Algoritmo de ordenacion

public class Algoritmos {
    public static void ordenar (double valores[]) {
        int i, j;
        for (i = 0; i < valores.length; i++)
            for (j = valores.length-1; j > i; j--)
                if (valores[j] < valores[j-1])
                    intercambiar (valores, j, j-1);
    }
}
```

32

```
// Generalizacion

public class Algoritmos {
    public static void ordenar (Ordenable valores[]) {
        int i, j;
        for (i = 0; i < valores.length; i++)
            for (j = valores.length-1; j > i; j--)
                if (valores[j].menorQue (valores[j-1]))
                    intercambiar (valores, j, j-1);
    }
}
```

Se necesita:

- Una forma genérica de referirse a datos ordenables
- Una definición de la relación de orden para cada tipo de datos ordenables

33

```
interface Ordenable {
    boolean menorQue (Ordenable valor);
}

class Directivo extends Empleado implements Ordenable {
    ...
    public boolean menorQue (Ordenable dir) {
        return categoria < ((Directivo) dir).categoria;
    }
}

class Figura implements Ordenable {
    ...
    public boolean menorQue (Ordenable fig) {
        return area () < ((Figura) fig).area ();
    }
}
```

34

```
public static void main (String args[]) {
    Directivo jefes[] = {
        new Directivo (...),
        new Directivo (...),
        new Directivo (...)
    };
    Figura objcompuesto[] = {
        new Triangulo (...),
        new Circulo (...),
        new Rectangulo (...)
    };
    Algoritmos.ordenar (jefes);
    Algoritmos.ordenar (objcompuesto);
}
```

35

De hecho...

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

public class Collections {
    public static void sort (List list)
    public static int binarySearch (List list, Object key)
    public static Object min (Collection col)
    public static Object max (Collection col)
    ...
}
```

36

Ejemplos de interfaces en la librería estándar de Java

- Comparable (java.util)
- Event listeners (java.awt.event)
- TableModel (javax.swing.table)
- Remote (java.rmi)
- Serializable (java.io)
- ...

37

¿Qué es una interfaz?

- Colección de métodos sin definir y valores constantes
- Similar a una clase abstracta con todos los métodos abstractos y públicos y todas las variables public static final
- Derivar de una clase → implementar una interfaz
- Una clase puede implementar varias interfaces

38

¿Para qué sirve una interfaz?

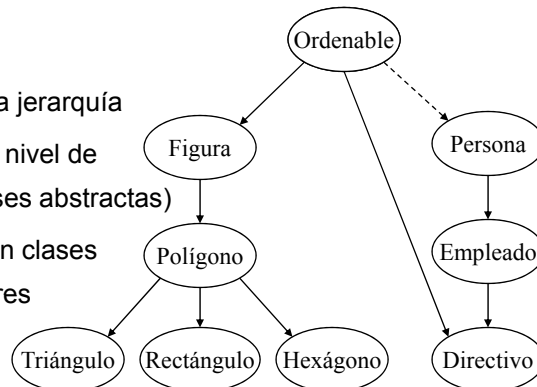
- Una interfaz impone un protocolo de métodos a implementar
 - Es un buen medio de “comunicación” entre programadores
- Una interfaz introduce un nuevo tipo
Las clases que implementan la interfaz son compatibles con este tipo
- Las interfaces posibilitan la herencia múltiple en cuanto a jerarquía de tipos:
Una interfaz define un tipo \Rightarrow una clase puede tener múltiples supertipos
- Sin embargo una interfaz no tiene contenido que heredar:
 - los métodos no tienen código
 - las variables se heredan pero son inamovibles (son estáticas y constantes)
- Una interfaz no proporciona funcionalidad a un objeto, sino la posibilidad de ser objeto de la funcionalidad de otros objetos

39

¿Por qué una interfaz y no una clase?

Si Ordenable es una clase abstracta:

- Obliga a derivar desde la cima de la jerarquía
- Obliga a definir el método `menor` a nivel de `Persona` y de `Empleado` (si no, clases abstractas)
- Problema si `Persona` y `Directivo` son clases definidas por distintos programadores

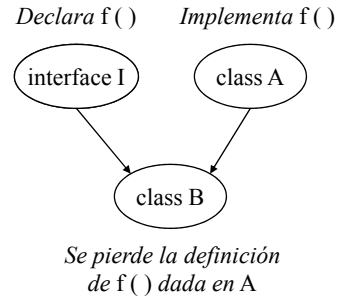


En general, siempre que sea posible y tenga sentido, mejor interfaz que clase

40

Interfaces: otros detalles

- Una interfaz puede derivar de varias otras interfaces con `extends`
- Las interfaces también pertenecen a un package
- Las interfaces pueden ser `public` o del package
- Los métodos implementados deben ser `public` en las clases que implementan las interfaces
- Al implementar una interfaz, la clase pierde las definiciones de los métodos que coincidan con métodos definidos en sus superclases



41

Ejemplo de abstracción de clases: Framework de colecciones Java

- Arquitectura para representar y manipular colecciones de datos
- Interfaces: jerarquía de funcionalidades
- Implementaciones: distinto comportamiento
- Clases abstractas: implementaciones parciales
- Algoritmos: polimórficos, operaciones sobre las interfaces

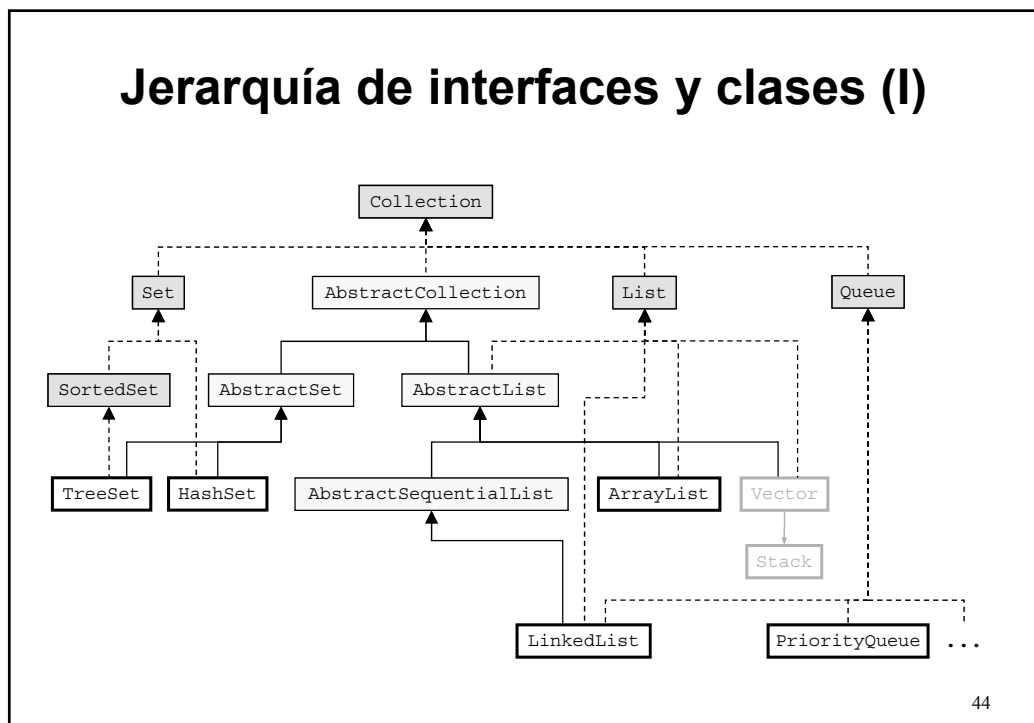
42

Tipos de colección

- Listas
 - Colecciones de objetos con posición
 - Útil cuando interesa que exista un orden
 - Implementación típica: `ArrayList`
- Conjuntos
 - Colecciones de objetos sin posición
 - No se repiten los elementos
 - Implementación típica: `TreeSet`
- Maps
 - Pares clave/valor
 - La solución más eficiente para almacenar relaciones entre dos conjuntos
 - Implementación típica: `HashMap`

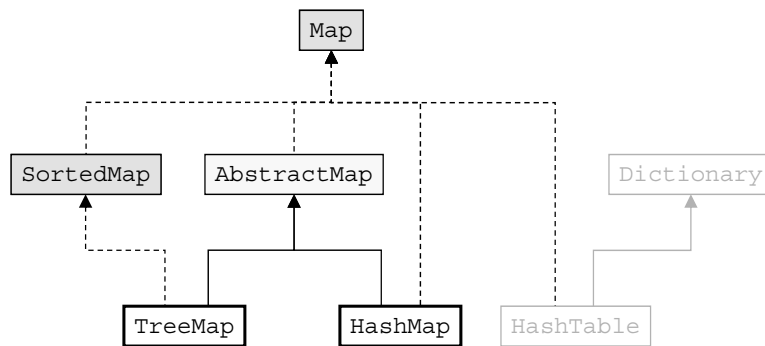
43

Jerarquía de interfaces y clases (I)



44

Jerarquía de interfaces y clases (II)



45

Interfaz java.util.Collection

- Colección genérica
- Insertar, eliminar, pertenencia, iterar

46

```
public interface Collection<E> extends Iterable<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T a[]);
}

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

Interfaz java.util.Set

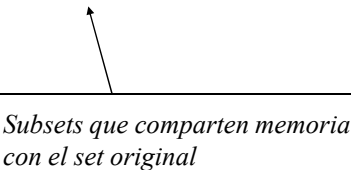
- Mismos métodos que Collection
- Los elementos no pueden repetirse
Esto influye en add, containsAll, addAll, retainAll, removeAll
- set1.equals (Object set2) puede devolver true aunque set2 no sea del mismo tipo de implementación que set1
- Implementaciones
 - HashSet: tabla hash, máxima eficiencia
 - TreeSet: árbol red-black, asegura orden de iteración

Interfaz java.util.SortedSet

- Conjunto ordenado
- Con `iterator()` se itera por orden “natural” (ver interfaz `Comparable`)
- `toArray()` devuelve los elementos en orden
- `toString()` muestra los elementos en orden
- Se puede especificar una relación de orden (función de comparación) en el constructor (ver implementaciones)
- Implementaciones: `TreeSet`

49

```
public interface SortedSet<E> extends Set<E> {  
    // Range-view  
    SortedSet<E> subSet(E fromElement, E toElement);  
    SortedSet<E> headSet(E toElement);  
    SortedSet<E> tailSet(E fromElement);  
  
    // Endpoints  
    E first();  
    E last();  
  
    // Comparator access  
    Comparator<? super E> comparator();  
}
```



Subsets que comparten memoria con el set original

50

Interfaz java.util.List

- Colección con posiciones asociadas a los elementos
 - Operaciones posicionales
 - add al final, remove el primero
 - list1.equals (Object list2) puede devolver true aunque list2 no sea del mismo tipo de implementación que list1
 - Implementaciones
 - ArrayList: encapsula un array dinámico (acceso cte., inserción/eliminación lineal)
 - LinkedList: encapsula una lista enlazada (acceso lineal, inserción/eliminación cte.)
- ```
addFirst (Object obj) addLast (Object obj)
getFirst () getLast ()
removeFirst () removeLast ()
```

51

```
public interface List<E> extends Collection<E> {
 // Positional Access
 E get(int index);
 E set(int index, E element);
 void add(int index, E element);
 E remove(int index);
 abstract boolean addAll(int index, Collection<? extends E> c);

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

 // Backward iteration
 ListIterator<E> listIterator();
 ListIterator<E> listIterator(int index);

 // Range-view
 List<E> subList(int from, int to);
}
```

Sublista que comparte memoria con la lista original  
Si la original cambia, resultado impredecible

52

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

 boolean hasPrevious();
 E previous();

 int nextIndex();
 int previousIndex();

 void set (E e);
 void add (E e);
}
```

53

## Interfaz java.util.Map

- Pares clave / elemento
- Las claves no se repiten
- `map1.equals (Object map2)` puede devolver true aunque `map2` no sea del mismo tipo de implementación que `map1`
- Implementaciones
  - `HashMap`: tabla hash, máxima eficiencia
  - `TreeMap`: árbol red-black, asegura orden (natural) de iteración

54

```
public interface Map<K,V> {
 // Basic Operations
 V put(K key, V value);
 V get(K key);
 V remove(K key);
 boolean containsKey(Object key);
 boolean containsValue(Object value);
 int size();
 boolean isEmpty();

 // Bulk Operations
 void putAll(Map<? extends K, ? extends V> t);
 void clear();

 // Collection Views
 public Set<K> keySet();
 public Collection<V> values();
 public Set<Map.Entry<K,V>> entrySet();
 ...
```

Siempre se devuelve el  
mismo objeto Collection.  
remove afecta, add no

55

```
...
// Interface for entrySet elements
public interface Entry<K,V> {
 K getKey();
 V getValue();
 V setValue(V value);
}
}
```

56

## Interfaz java.util.SortedMap

- Map ordenado por clave
- `first`, `last`, rangos (submaps)
- Con `iterator()` se itera por orden de clave
- `toArray()` devuelve los elementos en orden de clave
- `toString()` muestra los elementos en orden
- Permite cambiar la relación de orden (función de comparación)
- Implementaciones: `TreeMap`

57

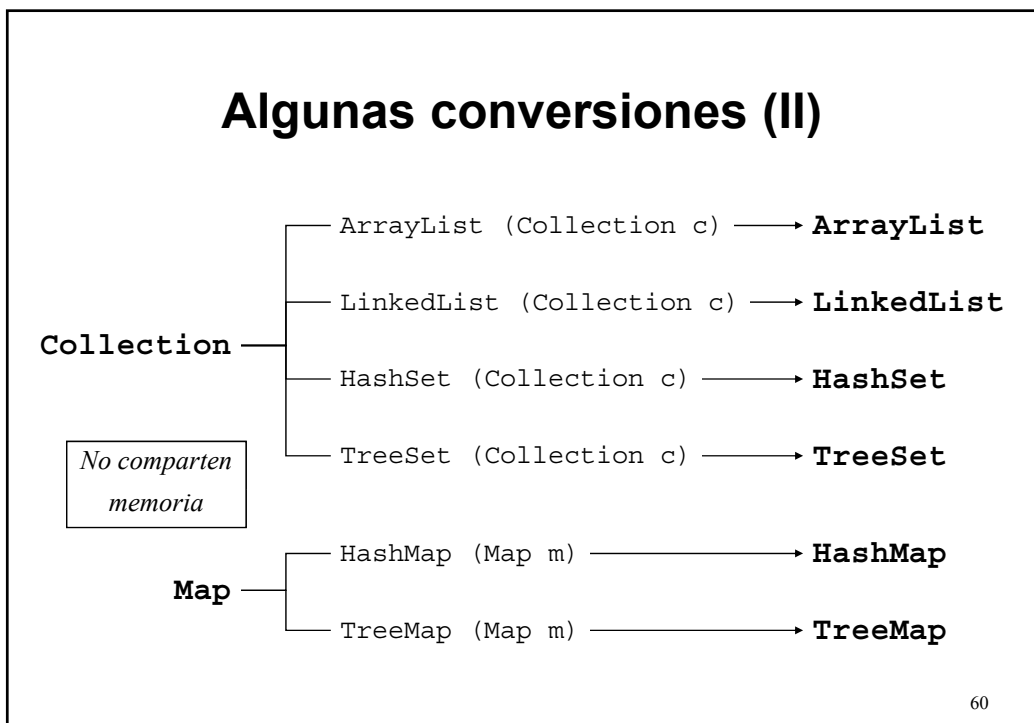
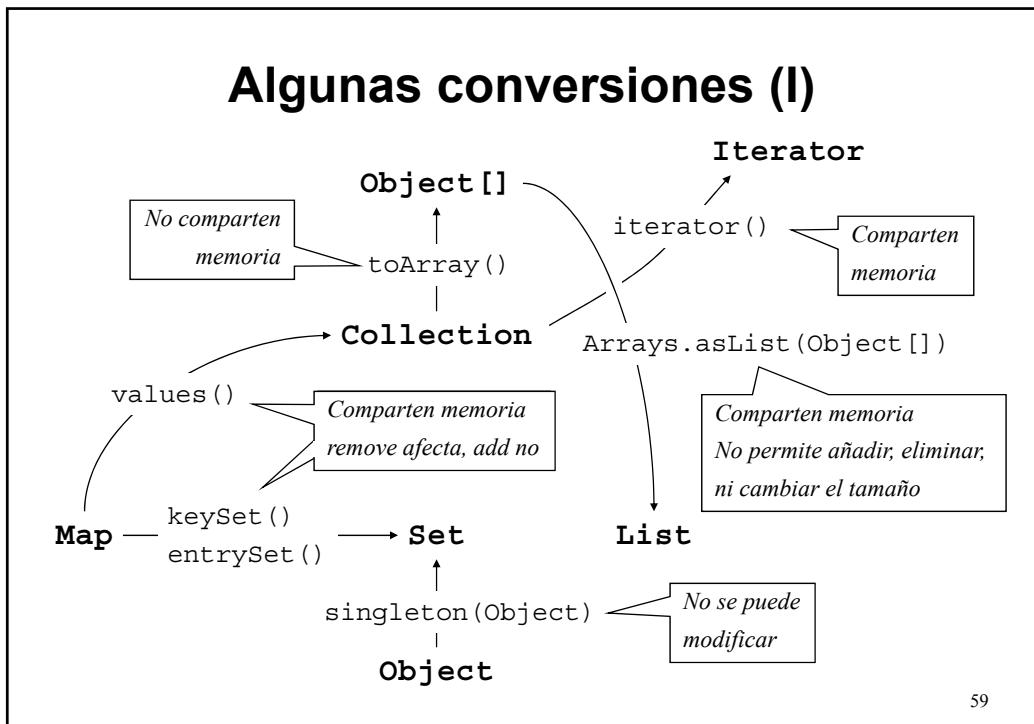
```
public interface SortedMap<V,K> extends Map<V,K> {
 Comparator<? super K> comparator();

 K firstKey();
 K lastKey();

 // Range view
 SortedMap<V,K> subMap(K fromKey, K toKey);
 SortedMap<V,K> headMap(K toKey);
 SortedMap<V,K> tailMap(K fromKey);
}
```

*Submaps que comparten memoria  
con el map original*

58



## Algoritmos sobre colecciones Interfaz java.util.Collections

```
class Collections
 public static void sort (List list)
 public static void sort (List list, Comparator c)
 public static int binarySearch (List list, Object key)
 public static int binarySearch (List list, Object key,
 Comparator c)
 public static Object min (Collection col)
 public static Object min (Collection col, Comparator c)
 public static Object max (Collection col)
 public static Object max (Collection col, Comparator c)
 public static List reverse (List list)
 ...
```

*Los elementos de las colecciones tienen que implementar Comparable*

61

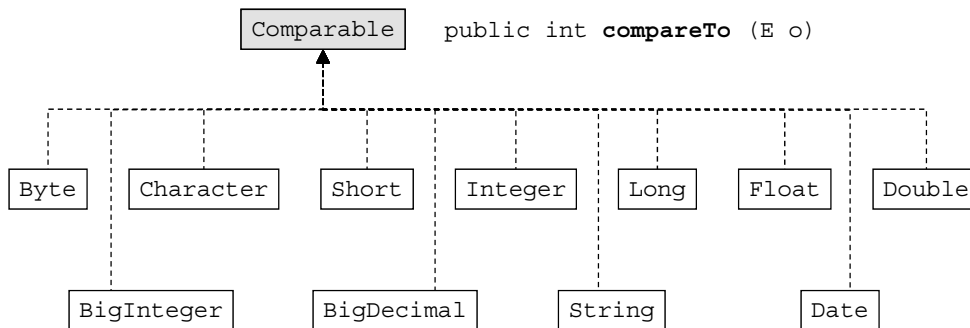
## Interfaz java.lang.Comparable

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

- Hacer que una clase pueda ser objeto de comparación (establecer orden natural, definir relación de orden total en una clase) p.e.:
  - Para introducirlas en colecciones ordenadas SortedSet ó SortedMap
  - Para algoritmos de búsqueda y ordenación
- ClassCastException si se comparan Comparable's de distinta clase
- Conviene que sea consistente con equals para SortedSet y SortedMap
- Algunas clases Java implementan Comparable

62

## Clases Java que implementan java.lang.Comparable



63

## Interfaz java.util.Comparator

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

- Saltarse la relación de orden natural de un Comparable
  - En colecciones ordenadas SortedSet ó SortedMap
  - Para algoritmos de búsqueda y ordenación
- Conviene que sea consistente con equals para SortedSet y SortedMap

64

## Diseño de clases basadas en colecciones

- Métodos que manipulan colecciones
  - Argumentos de entrada: interfaces, lo más genéricas posibles
  - Valores de retorno: la implementación más específica posible
- Nuevas implementaciones de colecciones
  - Escoger la clase abstracta apropiada
    - `AbstractCollection`
    - `AbstractSet`
    - `AbstractList`
    - `AbstractSequentialList`
    - `AbstractMap`
  - Implementar todos los métodos abstractos como mínimo

65

## Reflexión: conocimiento dinámico de las clases

## La clase `java.lang.Object`

- Clase padre por defecto si no se especifica otra  
`class A {...}` es equivalente a `class A extends Object {...}`
- Tratamiento genérico de objetos: vectores, etc.
- Proporciona definición por defecto para métodos como `equals`,  
`toString`, `getClass`
- Los arrays son también subclases de `Object`

67

## Metainformación sobre las clases en tiempo de ejecución

### El operador `instanceof`

```
Triangulo t = new Triangulo;
Figura fig = t;
t instanceof Triangulo → true
t instanceof Poligono → true
t instanceof Figura → true
t instanceof Circulo → false
fig instanceof Triangulo → true
```

68

## Metainformación de clase: ejemplo

```
class A {
 void f (Persona p) {
 if (p instanceof Empleado)
 System.out.println ("Clase Empleado");
 else if (p instanceof Persona)
 System.out.println ("Clase Persona");
 }
}
```

```
// Bloque main
A a = new A ();
Directivo dir = new Directivo (...);
Persona p = dir;
a.f (dir);
a.f (p);
Animal x = p;
a.f (x); // ERROR
```

69

## Metainformación sobre las clases en tiempo de ejecución

### La clase `java.lang.Class`

- Obtener un objeto `Class`  

```
Class c1 = t.getClass ();
Class c2 = fig.getClass ();
Class c3 = Class.forName ("Triangulo");
```

*método de la clase Object*

$\left. \begin{array}{l} c1 == c2 \\ c1 == c3 \end{array} \right\}$
- Métodos de `java.lang.Class`  

*método estático de la clase Class*

  - `getName ()` → `String`
  - `isArray ()`, `isPrimitive ()`, `isInterface ()`
  - `getConstructors ()`, `getFields ()`,  
`getMethods ()`, `getSuperclass ()`
  - `newInstance ()`
- `Class` es subclase de `Object`

70