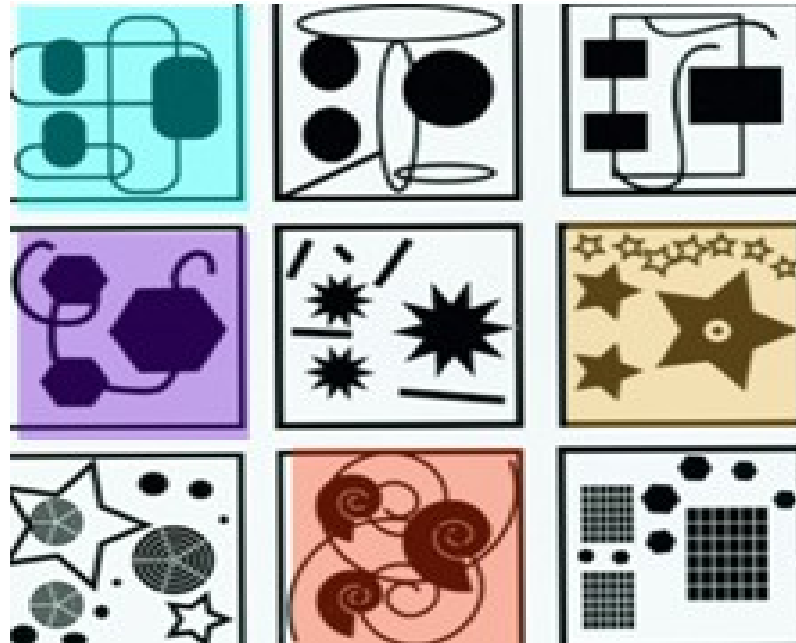


Tema 4



Conceptos complementarios en
Orientación a Objetos

Lección 4.3

Colecciones y copia de objetos

Objetivos de aprendizaje



- Conocer los distintos criterios de clasificación de colecciones
- Conocer la funcionalidad de las colecciones de objetos
- Conocer las propiedades de arrays, listas, conjuntos y diccionarios
- Saber implementar las colecciones anteriores tanto en Java como en Ruby
- Conocer las distintas formas de iterar por colecciones y saber usarlas en Java y en Ruby
- Ser capaz de usar la funcionalidad para ordenar objetos en colecciones con Java y con Ruby
- Conocer los distintos tipos de copias y saber usarlas en Java y Ruby
- Conocer y saber usar los métodos para comparación de objetos que usan las colecciones en Java y Ruby como criterios de ordenación

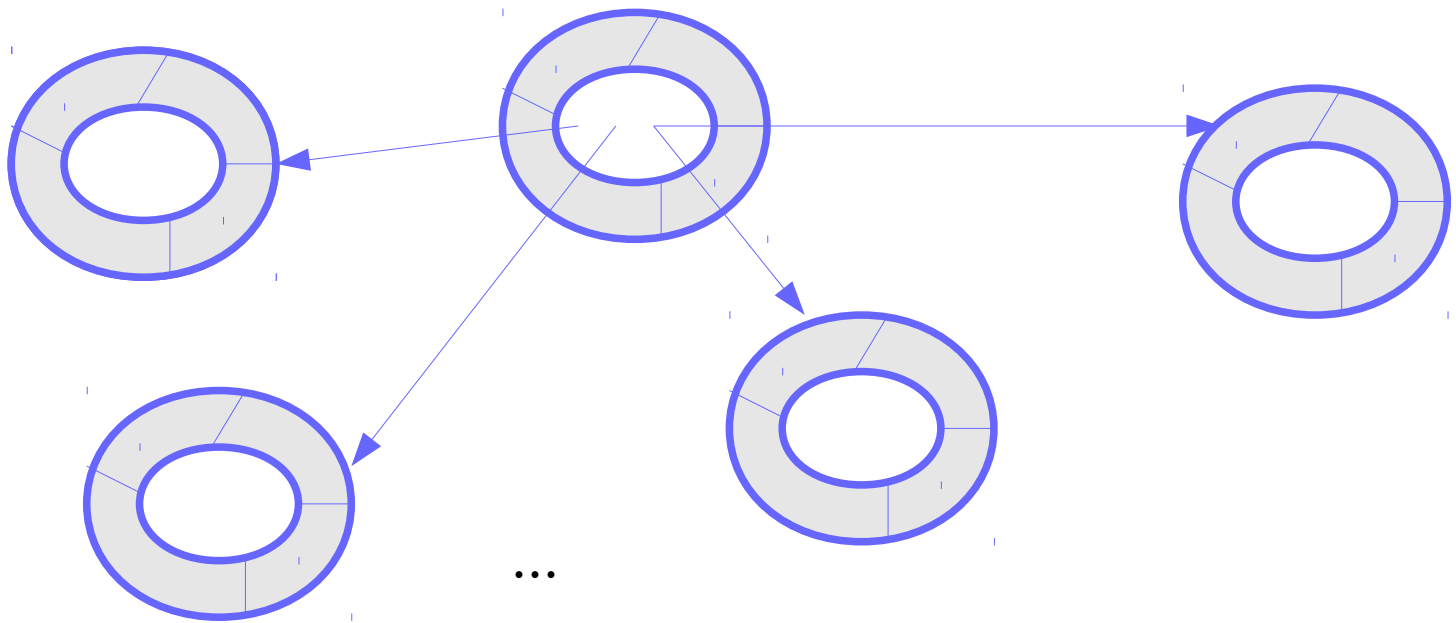
Contenidos



1. Concepto de colección
2. Criterios de clasificación
3. Funcionalidad
4. Arrays
5. Listas
6. Conjuntos
7. Diccionarios
8. Implementación de colecciones en Java y Ruby
9. Colecciones de objetos: iteradores y ordenación
10. Comparación de objetos
11. Copia de objetos
12. Constructor de copia
13. Clonación de objetos
14. Copia defensiva
15. Copia por serialización

1. Concepto de colección (repaso)

- Cuando el estado de un objeto viene determinado por un conjunto de objetos iguales o parecidos, se dice que ese objeto es una **colección de objetos**.



2. Criterios de clasificación (repaso)

- Las colecciones de objetos pueden **clasificarse** según:
 - **Tamaño:** fijas o variables.
 - **Contenido:** homogéneas y heterogéneas.
 - **Orden de elementos:** con o sin orden predeterminado.

Fijas y homogéneas -----> Eficientes.

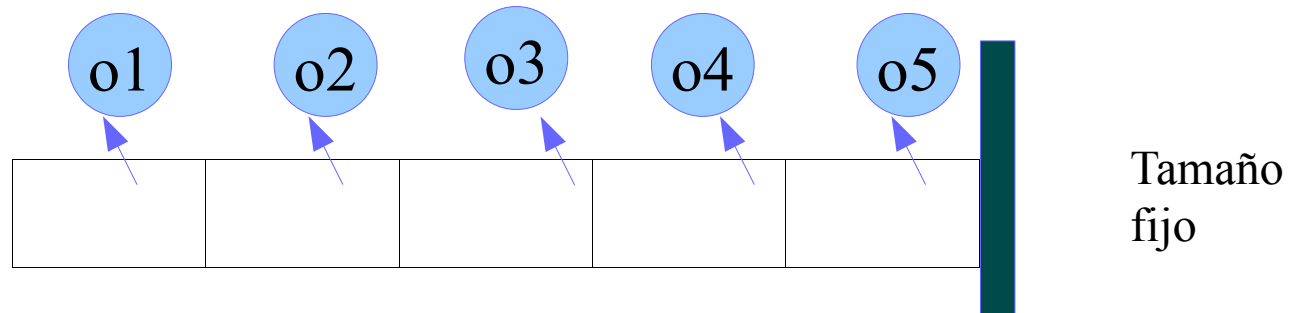
Variables y heterogéneas -----> Flexibles.

3. Funcionalidad (repaso)

- La **funcionalidad** general de una colección de objetos es:
 - Incluir uno o varios objetos.
 - Eliminar uno o varios objetos.
 - Comprobar la existencia de un determinado objeto.
 - Obtener un determinado elemento.
 - Obtener el número de elementos.
 - Iterar sobre todos sus elementos.

4. Arrays

- **Array:**
 - Colección homogénea (que contiene elementos del mismo tipo o clase) y de tamaño fijo.
 - Operaciones básicas: consultar el tamaño del array, acceder al elemento en una posición dada para consultarlo o modificarlo.



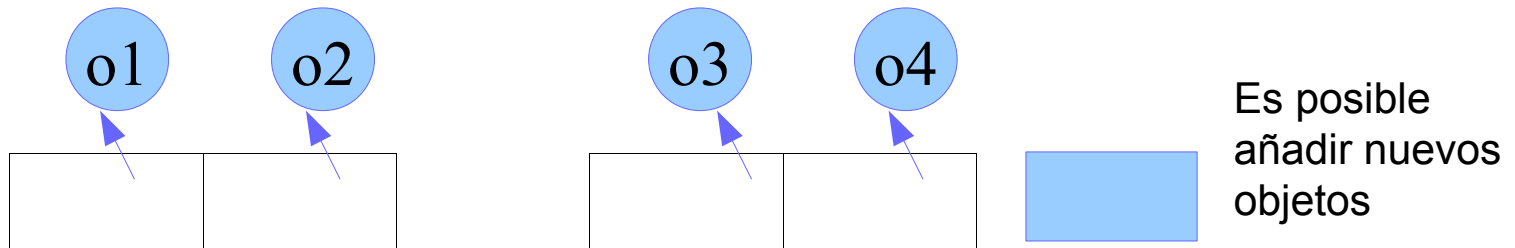
Ruby: no existe

Java:

```
int[] numeros = new int[100];  
numeros[4]=7;  
int tam = numeros.length;
```


5. Listas

- **Lista:**
 - Colección homogénea o heterogénea de tamaño variable. Admite elementos duplicados.
 - Operaciones básicas: insertar un elemento al final de la lista, consultar o borrar el elemento en una posición concreta de la lista, consultar la posición que ocupa un elemento en la lista.

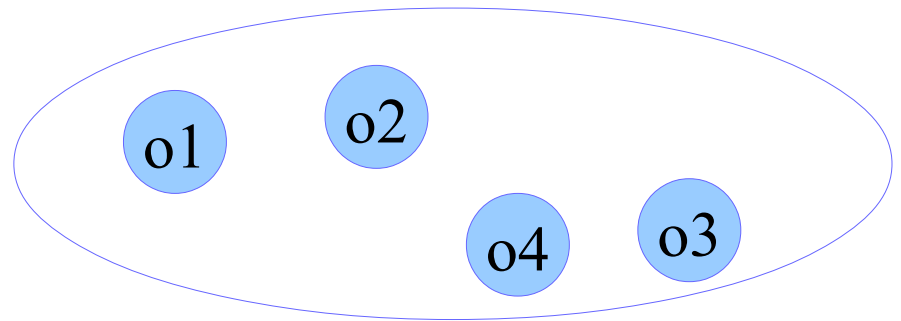


Ruby: `monstruos=Array.new`

Java: `ArrayList<Monstruo>monstruos= new ArrayList()
LinkedList<Monstruo>monstruos= new LinkedList()
// doblemente enlazada`

6. Conjuntos

- **Conjunto:**
 - Colección homogénea o heterogénea, de tamaño variable, no ordenada y sin elementos duplicados.
 - Operaciones básicas: pertenencia de un elemento al conjunto, eliminación de un elemento particular e inserción de un elemento (sin indicar posición).



Ruby: `monstruos=Set.new`

Java: `HashSet<Monstruo>monstruos= new HashSet()
TreeSet<Monstruo>monstruos= new TreeSet()
// elementos ordenados`

6. Conjuntos

- **Diferencias** entre TreeSet y HashSet:

- **Ordenación:**

El concepto matemático de conjunto es una colección no ordenada. En la realidad algunas implementaciones garantizan un orden definido por defecto o por el usuario (como TreeSet). HashSet a diferencia de TreeSet no garantiza que el orden se mantenga constante.

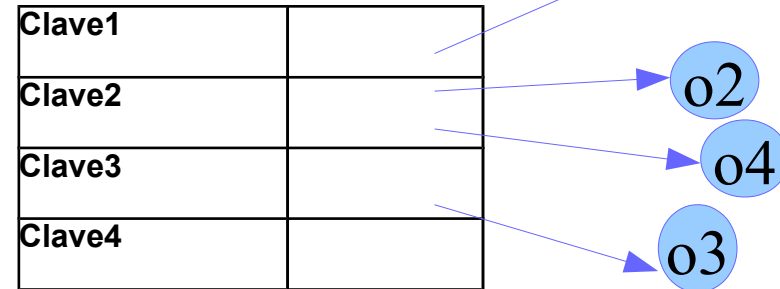
- **Eficiencia:**

HashSet es mucho más eficiente que TreeSet.

7. Diccionarios

- Diccionario:**

- Colección de objetos indexados mediante claves, los objetos pueden estar repetidos pero no las claves.
- Conjunto de pares clave-elemento donde todas las claves son de la misma clase A, y todos los elementos de la misma clase B (homogénea y sin elementos duplicados en la clave).
- Operaciones básicas: guardar un elemento con una clave, extraer un elemento dada su clave, borrar un par clave-elemento, consultar todas las claves, ordenar por clave, añadir clave-valor.



Ruby: `monstruos=Hash.new`

Java: `HashMap<Monstruo,String> monstruos=
 new HashMap<Monstruo,String>()
 TreeMap monstruos= new TreeMap() //orden`

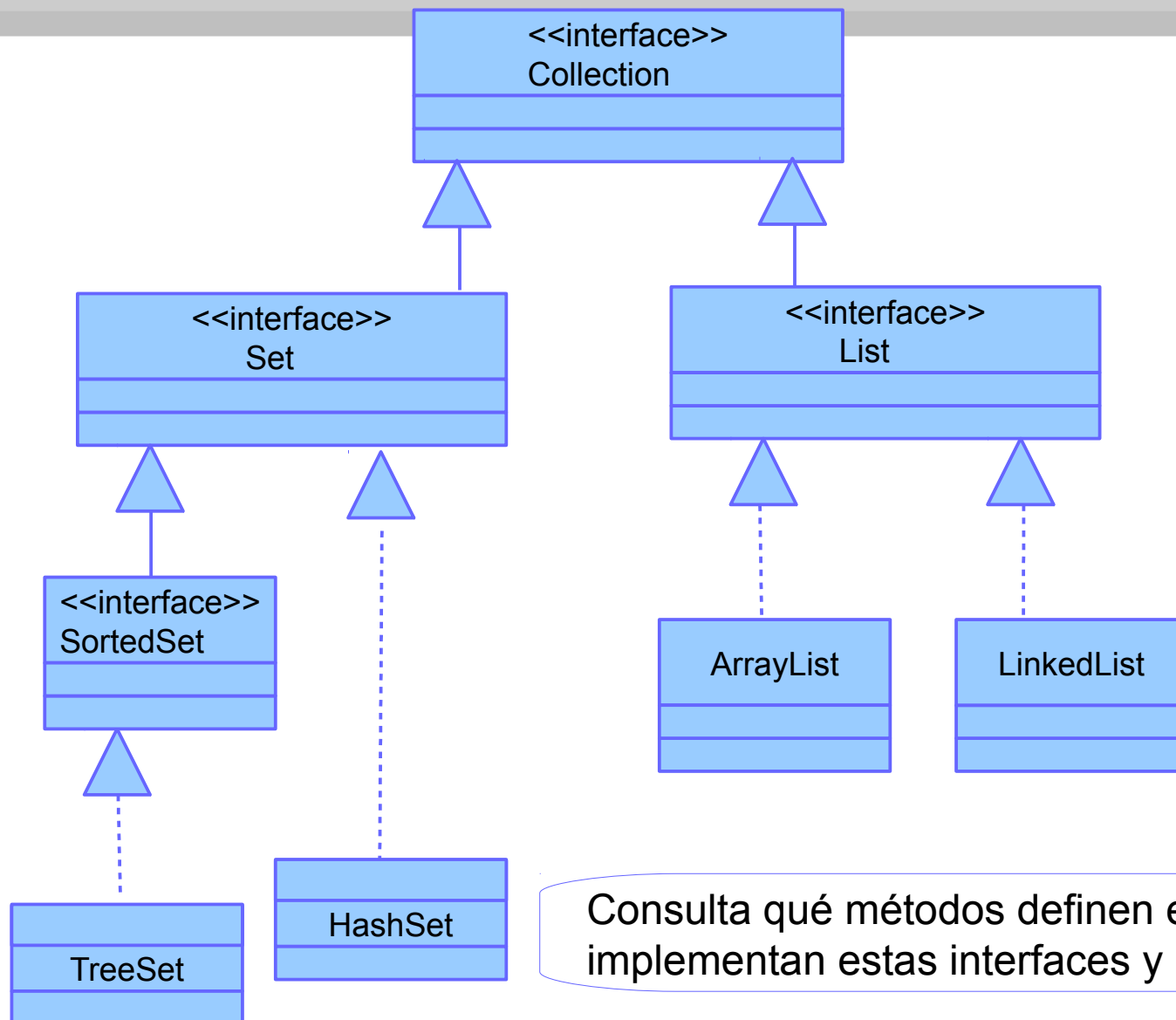
8. Implementación de colecciones en Java y Ruby

	Listas	Conjuntos	Diccionarios
Propiedad	<ul style="list-style-type: none">• Sin orden• Con duplicados	<ul style="list-style-type: none">• Sin duplicados	<ul style="list-style-type: none">• Cada elemento es un par: (key,value)• Sin duplicados en su key
Java	Clases: <ul style="list-style-type: none">• ArrayList• LinkedList	Clases: <ul style="list-style-type: none">• HashSet (sin orden)• TreeSet (ordenados según la relación de orden definida entre sus elementos)	Clases: <ul style="list-style-type: none">• HashMap (sin orden)• TreeMap (ordenado por la key y según la relación de orden definida en la clase a la que pertenece key)
Ruby	Clases: <ul style="list-style-type: none">• Array	Clases: <ul style="list-style-type: none">• Set (sin orden)	Clases: <ul style="list-style-type: none">• Hash (sin orden)

Entender y manipular los ejemplos proporcionados sobre Colecciones[Java|Ruby] en EjemplosTema4.zip



4. Interfaz y sus relaciones: ejemplo



Consulta qué métodos definen e implementan estas interfaces y clases



9. Colecciones de Objetos: Iteradores

Un **iterador** es un mecanismo de **abstracción de control** que **permite recorrer todos o parte de los elementos de una colección** para llevar a cabo alguna operación con cada uno de ellos, sin necesidad de conocer la estructura interna ni la funcionalidad de la colección.

La **funcionalidad básica del iterador** es:

- Construcción del iterador, colocándolo en el primer elemento de la colección.
- Siguiente elemento al que está posicionado.
- Consulta para saber si hay más elementos por recorrer.

Con un iterador podemos:

- Acceder a todos los elementos de la colección.
- Acceder a parte de esos elementos, mientras que se cumpla una condición.
- Seleccionar durante el recorrido los elementos que cumplan con una determinada condición.

Todos los lenguajes de programación proporcionan distintos tipos de iteradores o formas de recorrer colecciones de objetos.

9. Colecciones de Objetos: Iteradores

Java: Construcción y uso de un iterador

- Construcción de la colección sobre la que iterar:

```
ArrayList<MiClase> misObjetos = new ArrayList();
```

- Construcción del iterador:

```
Iterator<MiClase> itMiClase = misObjetos.iterator();
```

- Siguiente elemento del iterador:

```
MiClase miObjeto = itMiClase.next();
```

- Consulta para saber si quedan elementos en el iterador:

```
boolean hayElementos = itMiClase.hasNext();
```

Los iteradores se usan en las estructuras cíclicas: `for` y `while`

A continuación se estudia cómo recorrer los elementos de una colección.

Ruby: no proporciona la posibilidad de definir un iterador, pero sí métodos de iteración o de recorrido de elementos de una colección, como se verá más adelante.

9. Colecciones de Objetos: Iteradores

Listas y conjuntos:



// Construcción

```
ArrayList<MiClase> miLista = new ArrayList(); // Construcción
```

// Mediante el iterador

```
for (Iterator<MiClase> it = miLista.iterator(); it.hasNext();)
{
    MiClase miC = it.next();
    // código para usar miC
    ... }

```

// Mediante un for-each

```
for (MiClase miC:miLista)
{ // código para usar miC
    ... }

```

9. Colecciones de Objetos: Iteradores



Diccionarios:

// Construcción

```
HashMap<KeyClase, ValueClase> miMap = new HashMap();
```

// Mediante el iterador

```
Map<KeyClase, ValueClase> map = new HashMap<KeyClase, ValueClase>();  
for (Iterator<Map.Entry<Integer, Integer>> it =  
    map.entrySet().iterator(); it.hasNext())  
{  
    Map.Entry<KeyClase, ValueClase> entry = it.next();  
    KeyClase unaKey = entry.getKey();  
    ValueClase unValue = entry.getValue();  
    // código para usar unaKey y unValue;  
    ...}
```

// Mediante un for-each

```
for (Map.Entry<KeyClase, ValueClase> miKV : miMap.entrySet()) {  
    KeyClase unaKey = miKV.getKey();  
    ValueClase unValue = miKV.getValue();  
    // código para usar unaKey y unValue  
    ... }
```

9. Colecciones de Objetos: Iteradores



Recorrido de los elementos de una colección en Ruby

- **Listas y conjuntos**

```
miLista = Array.new # Construcción de la colección
miLista.each { |elemento|
  # código para usar elemento
}
for elemento in miLista      # Otra sintaxis alternativa
  # código para usar elemento
end
```

- **Diccionarios**

```
miMap = HashMap.new # Construcción del diccionario
miMap.each { |key value|
  # código para usar key y value
}
miMap.each_key { |key|
  #t código para usar key
}
# Igual para el value
```

9. Colecciones de Objetos: Ordenación

- Lo normal es que en una colección los objetos estén ordenados según el orden de inserción.
- Para ordenarlos según otro criterio hay que recurrir a la funcionalidad proporcionada por el lenguaje para ello.
- En algunos lenguajes, p. ej. Java, existen determinadas colecciones que permiten ir insertando sus elementos de forma ordenada y según la relación de orden existente entre ellos. Por ejemplo, las clases que implementan las interfaces SortedTree y SortedMap como son TreeSet y TreeMap respectivamente.

9. Colecciones de Objetos: Ordenación



- **Listas** (usando el método de clase `sort()` de la clase `Collections`):

```
ArrayList<MiClase> miLista = new ArrayList(); // Construcción
```

// Primera forma usando `compareTo()`:

```
Collections.sort(miLista) ; //En MiClase debe estar definido compareTo()
```

// Segunda forma usando la clase comparadora:

```
Collections.sort(miLista, new MiClaseComparadora());
```

El resultado de las dos formas es *miLista* ordenada según la relación de orden definida entre sus elementos.

Ver y entender los ejemplos de
Colecciones en Java de
EjemplosTema4.zip



- **Conjuntos:**

Para tener un conjunto ordenado usamos la clase *TreeSet*, en la que sus elementos se van incluyendo de forma ordenada conforme a la relación de orden definida entre ellos. La ordenación se hace igual que con las listas.

- **Diccionarios:**

Para tener un diccionario ordenado usamos la clase *TreeMap*, en la que sus elementos se van incluyendo de forma ordenada conforme a la relación de orden definida entre los elementos de la key.

9. Colecciones de Objetos: Ordenación



- **Listas:**

Es la propia clase (Array) la que proporciona los métodos de instancia para ordenar los elementos de una colección, siempre usando la relación de orden que se defina entre sus elementos.

```
miLista = Array.new // Construcción de la colección
listaOrdenada = miLista.sort
listaOrdenada = miLista.sort {|ele1,ele2| ele1 <=> ele2 }
```

El resultado en los dos casos es miLista ordenada en listaOrdenada según la relación de orden definida por <=> entre sus elementos.

- **Conjuntos:**

Cuando ordenamos un objeto de la clase Set, internamente se pasa un objeto Array que es el que se ordena y el resultado se devuelve en un nuevo objeto Set ordenado.

- **Diccionarios:**

Ocurre igual que con los objetos de la clase Set.

Ver y entender los ejemplos de Colecciones en Ruby en EjemplosTema4.zip, donde hay más ejemplos de todo el tema



10. Comparación de objetos: identidad y estado

- A la hora de comparar objetos podemos comparar identidad o estado.
- Dependiendo de cómo se hayan construido los objetos a comparar, así será el resultado de la comparación de identidad o estado.

	Comparando identidad ¿objetos idénticos?	Comparando estado ¿objetos iguales?
a es idéntico a b	true	true
a es copia de b	false	true
a no es ni idéntico ni copia de b	false	true o false

- Todos los lenguajes de programación proporcionan funcionalidad para comparar identidad e igualdad (estado) de objetos.

10. Comparación de objetos: identidad y estado



`obj1 == obj2`, `obj1 != obj2` y `obj1.equals(obj2)`

- Comparan identidad por defecto y devuelven true o false
- Para comparar estado se redefine `equals(obj)`
- `obj1 != obj2` es equivalente a `!(obj1 == obj2)`

Ejemplo

```
class MiClase{
    private String saludo;
}
MiClase mc1 = new MiClase("hola");
MiClase mc2 = new MiClase("hola");
MiClase mc3 = mc1;
```

Sin redefinir equals(obj2)

```
mc1 == mc2 // false
mc1 == mc3 // true
mc2 == mc3 // false
mc1.equals(mc2) // false
mc1.equals(mc3) // true
mc2.equals(mc3) // false
```

Entender y manipular el
código en
EjemplosTema43Java



Redefiniendo equals(obj2)

```
mc1 == mc2 // false
mc1 == mc3 // true
mc2 == mc3 // false

mc1.equals(mc2) // true
mc1.equals(mc3) // true
mc2.equals(mc3) // true
```


10. Comparación de objetos: identidad y estado

Código recomendado para redefinir **equals(obj)**



```
@Override
public boolean equals(Object obj) {

    if (obj == null)
        return false;

    if (obj == this)
        return true;

    if (!(obj.getClass().getSimpleName().equals("MiClase")))
        return false;

    MiClase mc = (MiClase) obj;
    if (!saludo.equals(mc.saludo))
        return false;

    /  / Y así para todos los atributos de obj
    return true;
}
```

Cabecera ya proporcionada y que no podemos cambiar (@Override)

Consulta el nombre de la clase a la que pertenece obj

Referenciar obj por una variable, mc, que sea de tipo MiClase

10. Comparación de objetos: identidad y estado



`obj1 == obj2`, `obj1 != obj2`, `obj1.equal?(obj2)` y `obj1.eql?(obj2)`

- Comparan identidad por defecto y devuelven true o false
- `obj1 != obj2` equivalente a `!(obj1 == obj2)`
- Recomendaciones para la redefinición de estos métodos:
 - Para comparar estado redefinir `obj1 == obj2`
 - No redefinir `equal?(obj2)` ya que se usa internamente para determinar identidad
 - Mantener el mismo significado de `==` y `eql?(obj2)`: para Hash compara keys

```
class MiClase
  attr_reader :saludo
end

mc1 = MiClase.new("hola")
mc2 = MiClase.new("hola")
mc3 = mc1
```

Sin redefinir `obj1==obj2`

```
mc1 == mc2 # false
mc1 == mc3 # true
mc2 == mc3 # false

mc1.equal?(mc2) # false
mc1.equal?(mc3) # true
mc2.equal?(mc3) # false
```

Entender y manipular el
código en
EjemplosTema43Ruby



Redefiniendo `obj1==obj2` (comparar estado)

```
mc1 == mc2 # true
mc1 == mc3 # true
mc2 == mc3 # true

mc1.equal?(mc2) # false
mc1.equal?(mc3) # true
mc2.equal?(mc3) # false
```

10. Comparación de objetos: identidad y estado

Código recomendado para redefinir == (obj)



```
def ==obj

  if (obj == nil)
    return false
  end

  if obj.class.name.split('::').last != 'MiClase'
    return false
  end

  if @saludo != obj.saludo
    return false
  end

  # y así para todos los atributos de obj
  return true

end
```

Consulta el nombre
de la clase a la que
pertenece obj

10. Comparación de objetos: orden



Para poder comparar el orden de los objetos, debe establecerse o existir una relación de orden entre ellos.

En Java puede hacerse de dos formas:

- con **compareTo** en la misma clase
- con **compare** en una clase nueva **comparadora**.

El **resultado de la comparación** debe ser:

- 0 si son iguales (estado).
- -1 si el objeto receptor es menor que objeto argumento.
- 1 si el objeto receptor es mayor que el argumento.

10. Comparación de objetos: orden



Primera forma: mediante el método: `compareTo(Object obj)`

Ejemplo

```
class MiClase implements Comparable<MiClase>{  
    private String saludo;  
    public int compareTo(MiClase obj){  
        if (saludo.compareTo(obj.saludo)==0) return 0;  
        if (saludo.compareTo(obj.saludo)>0) return 1;  
        if (saludo.compareTo(obj.saludo)<0) return -1;  
    }  
}
```

Método (compareTo())
ya definido en la
interfaz Comparable

```
MiClase mc1 = new MiClase("hola");  
MiClase mc2 = new MiClase("hola");  
MiClase mc3 = new MiClase ("tola")  
MiClase mc4 = new MiClase ("bola")  
mc1.compareTo(mc2) // Resultado --> 0  
mc1.compareTo(mc3) // Resultado ---> -1  
mc1.compareTo(mc4) // Resultado ---> 1
```

Ver en EjemplosTema4.zip
porqué hay que hacer esto



10. Comparación de objetos: orden

Segunda forma: definiendo una clase comparadora de objetos y ahí redefiniendo el método: **compare(MiClase mc1, MiClase mc2)**



Ejemplo

```
class MiClaseComparador implements Comparator<MiClase>{  
    public int compare(MiClase mc1, MiClase mc2){  
        return (mc1.compareTo(mc2));  
    }  
}
```

```
MiClase mc1 = new MiClase("hola");  
MiClase mc2 = new MiClase("hola");  
MiClase mc3 = new MiClase ("tola");  
MiClase mc4 = new MiClase ("bola");
```

```
MiClaseComparador mcCom = new MiClaseComparador();  
mcCom.compare(mc1,mc2); // Resultado -->0  
mcCom.compare(mc1,mc3); // Resultado --> -1  
mcCom.compare(mc1,mc4); // Resultado --> 1
```

El método `compareTo()` ya está definido en `MiClase`. Si no lo estuviera, habría que definir aquí la relación de orden

10. Comparación de objetos: orden



Proporciona toda la funcionalidad para comparar objetos

Ejemplo

```
class MiClase

  include Comparable
  attr_reader :saludo
  def <=> obj
    @saludo <=> obj.saludo
  end

end

mc1 = MiClase.new("hola")
mc2 = MiClase.new("hola")
mc3 = MiClase.new("tola")
mc4 = MiClase.new("bola")

mc1<=> mc2 // Resultado --> 0
mc1<=>mc3 // Resultado ---> -1
mc1 <=> mc4 // Resultado ---> 1
```

Tenemos que asegurarnos que la clase del objeto que referencia @saludo debe tener definido <=>

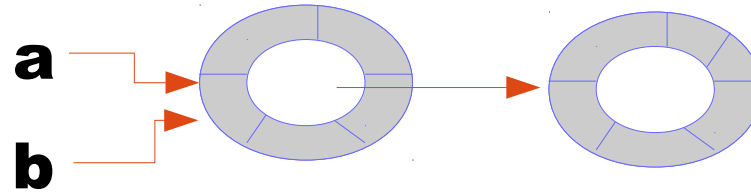
Entender y manipular el ejemplo dado para Ruby
EjemplosTema4.zip



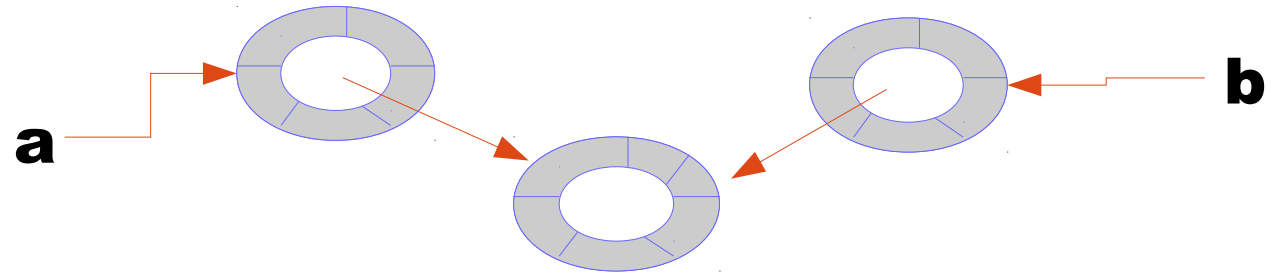
11. Copia de objetos

Al copiar un **objeto** (p. ej. **b** es copia de **a**) podemos hacer distintos tipos de copia:

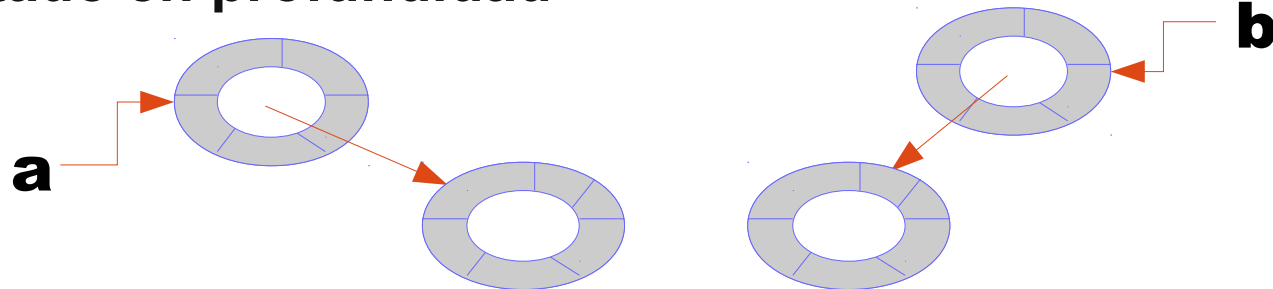
1. De identidad



2. De estado superficial



3. De estado en profundidad

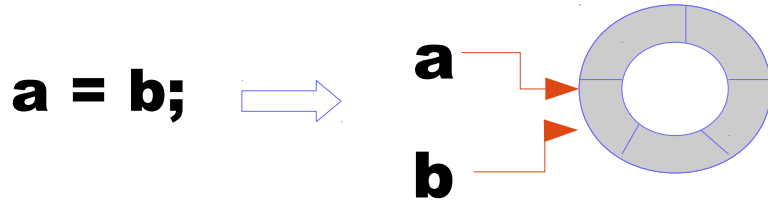


11. Copia de objetos: Copiando identidad

La **asignación** (=)

En el montículo o **heap** (memoria dinámica no automática)

- es una copia de identidad, por ejemplo en Java y en Ruby.



El paso de parámetro a métodos, tanto en Java como en Ruby, es por valor. Como las variables son punteros, lo que se copia es la dirección del objeto, diciéndose que se hace copia de identidad.

- Recordad de primer curso los conceptos de paso por valor y por referencia

En la pila o **stack** (memoria dinámica automática) (por ejemplo C++)

- es una copia de estado bien en superficie, bien en profundidad, dependiendo de que los atributos que forman parte del objeto se definan nuevamente en la pila (copia profunda) en el montón (copia superficial).
- Recordad de primer curso los conceptos de tipos de memoria y asignación de memoria

11. Copia de objetos: Copiando estado

Todos los lenguajes de programación proporcionan la funcionalidad adecuada para **copia de estado de objetos**. Los mecanismos usuales para copiar estado son:

- **Constructores de copia.**
- **Clonación** de objetos.

La **semántica** de los dos formas es **la misma**, obtener un objeto a partir de otro.

12. Constructor de copia

- Es posible copiar objetos creando un constructor que acepte como parámetro objetos de la misma clase. Este constructor se encargaría de hacer la copia profunda.
- Este esquema presenta algunos problemas cuando se utilizan jerarquías de herencia.

12. Constructor de copia

Constructor que acepta como parámetro un objeto de la misma clase y hace una copia profunda

Java

- **Definición** del constructor de copia

```
class MiClase{
    MiClase(MiClase mc){
        //Definición del tipo de copia tomando mc como objeto a copiar
    }
}
```

- **Uso** del constructor de copia

```
MiClase mc = new MiClase(..);
MiClase mcCopia = new MiClase(mc);
```

Ruby

- **Definición** del constructor de copia

```
class MiClase{
    ...
    def self.newCopia(mc)
        new(..) # usando mc para pasar valores a copiar
    end
end
```

- **Uso** del constructor de copia

```
mc = MiClase.new(..)
mcCopia = MiClase.newCopia(mc);
```

Entender y manipular el código en
EjemplosTema4[Java|Ruby]



12. Constructor de Copia

Ejemplo de declaración de dos constructores de copia



```
class A {  
    private int x;  
    private int y;  
    public A(int a,int b) { x=a; y=b;}  
    public A(A a) { x=a.x; y=a.y; }  
    @Override  
    public String toString()  
        return "(" + Integer.toString(x) + ","  
            + Integer.toString(y) + ") ";  
}
```

```
class B extends A {  
    private int z;  
    public B(int a,int b,int c) { super(a,b); z=c; }  
    public B(B b){ super(b); z=b.z; }  
    @Override  
    public String toString() {  
        return super.toString() + ("+" + Integer.toString(z) + ") ";  
    }  
}
```

12. Constructor de Copia

Ejemplo de uso de los dos constructores de copia

```
A a1=new A(11,22);
```

```
A a2=new B(111,222,333); // se supone que B hereda de A
```

```
A a3;
```

```
//a3=a1;
```

```
a3=a2;
```

```
A a4=null;
```

```
//a4=new A(a3);
```

```
a4=new B((B) a3);
```

```
System.out.print(a4);
```



El constructor de copia a utilizar viene condicionado según la asignación que se haya hecho antes

- Dependiendo de la clase del objeto a ser copiado es necesario utilizar un constructor de copia u otro.
- En general esta información no está disponible hasta la ejecución del programa.

13. Clonación de objetos


Justificación: objetos que contienen referencias a otros objetos y el estado de éstos no debería ser alterado (tendrían que ser inmutables).

La clonación permite trabajar con una copia de esos objetos sin modificar los originales.

Ejemplo:

```
class Numero{
    private Integer i;

    public Numero(Integer a){
        i=a;
    }
    public void inc() {
        i++;
    }
    @Override
    public String toString(){
        return i.toString();
    }
}
```



```
class Compleja {
    private ArrayList<Numero> numeros;

    public Compleja() {
        numeros=new ArrayList();
    }

    public void add(Numero n) {
        numeros.add(n);
    }

    ArrayList<Numero> getNumeros() {
        return numeros;
    }
}
```

13. Clonación de objetos

Ejemplo (cont.)



```
Compleja c1=new Compleja();

c1.add(new Numero(3));
c1.add(new Numero(2));
c1.add(new Numero(1));

c1.getNumeros().clear(); /***
c1.add(new Numero(8));
for( Numero n : c1.getNumeros() )
{
    System.out.println(n); // 8
}

c1.getNumeros().get(0).inc();
/***
for( Numero n : c1.getNumeros() )
{
    System.out.println(n); // 9
}
```

Problemas

- Se puede obtener acceso sin restricciones a la lista de números que contienen los objetos de la clase Compleja y alterar esta lista o los elementos de la lista.
- Los consultores devuelven referencias a objetos que forman parte del estado interno de otro objeto. Esos objetos que devuelve el consultor pueden modificarse.

Solución: usar clone para copia superficial o profunda de un objeto.

13. Clonación de objetos: Clone y la interfaz Cloneable

Método **clone()** definido en `Object` como funcionalidad general de todos los objetos, su significado por defecto es la copia superficial.

Declaración y uso del método `clone()` para duplicar objetos simples:



```
class MiClase implements Cloneable {
    private int atr;

    @Override
    public Object clone() {
        Object obj;
        try {
            obj = super.clone();
        } catch (CloneNotSupportedException e) {
            // Tratamiento de la excepción
        }
        return obj;
    }

    MiClase mc = new MiClase(...);
    MiClase mcCopia = (MiClase)mc.clone();
    //mcCopia es copia superficial de mc
}
```

13. Clonación de objetos: Clone y la interfaz Cloneable

- Al utilizar este mecanismo se redefine:

```
protected Object clone() throws CloneNotSupportedException
```



- Se suele redefinir de la siguiente forma:

```
public MiClase clone() throws CloneNotSupportedException
```

- El método creado debe crear una copia base (`super.clone()`). En **objetos compuestos**, debe crear copias de los atributos no inmutables, lo cual también puede hacerse usando el método `clone` sobre esos atributos.

```
class MiClase implements Cloneable {
    private ClasedeAtributo1 atr1;
    @Override
    public Object clone() {
        Object obj;
        try {
            obj = super.clone();
            ((MiClase) obj).atr1 =
                (ClasedeAtributo1) ((MiClase) obj).atr1.clone();
            return obj;
        } catch (CloneNotSupportedException e) {
            // Tratamiento de la excepción
        }
        return obj;
    }
}
```

13. Clonación de objetos: Clone y la interfaz Cloneable



Según la documentación oficial:

“Creates and returns a copy of this object. The precise meaning of “copy” may depend on the class of the object.

The general intent is that, for any object x, the expression:

1) `x.clone() != x` will be true

2) `x.clone().getClass() == x.getClass()` will be true, but these are not absolute requirements.

3) `x.clone().equals(x)` will be true, this is not an absolute requirement.

.....

.....

By convention, the object returned by this method should be independent of this object (which is being cloned)”

13. Clonación de objetos: Clone y la interfaz Cloneable



- A pesar de que existe la interfaz *Cloneable*, ésta no define ningún método. Este hecho rompe con el significado habitual de una interfaz en Java.

```
class Raro implements Cloneable {  
    //Este código no produce errores  
}
```

- En la clase *Object* se realiza la comprobación de si la clase que originó la llamada a *clone* implementa la interfaz. Si no es así se produce una excepción.

13. Clonación de objetos: clone y dup



La **funcionalidad de *clone* y *dup*** es la misma, copia superficial de objetos, la diferencia es que ***clone*** copia el propio objeto teniendo en cuenta todo lo definido en él y ***dup*** copia el objeto teniendo en cuenta las propiedades definidas en la clase a la que pertenece.

Ejemplo

```
class MiClase
  def saludo
    "Hola"
  end
end
mc = MiClase.new

def mc.saludo
  "Buenos Días"
end
def mc.saludoCordial
  "hola ¿qué tal?"
end

mcClone = mc.clone;
mcDup = mc.dup
```

```
puts mc.saludo           # Buenos Días
puts mcClone.saludo      # Buenos Días
puts mcDup.saludo        # Hola

puts mc.saludoCordial    # hola ¿qué tal?
puts mcClone.saludoCordial # hola ¿qué tal?
puts mcDup.saludoCordial  # ERROR
```

Definición de métodos solo
para un objeto concreto

13. Clonación de objetos: clone y dup



- En Ruby el método *clone* de la clase *Object* también realiza la copia superficial.
- Si se desea realizar la copia profunda debe realizarla el programador redefiniendo *clone* o *dup*.

11. Copia de objetos: Copiando estado

Las **diferencias entre constructor de copia y clone** son:

- En el constructor de copia el objeto a copiar es el parámetro del constructor y en la clonación es el objeto receptor del mensaje.
- Respecto a su definición, los constructores de copia los define el programador y los métodos para clonación los proporciona el lenguaje.
- Respecto al tipo de copia, cuando se define el constructor se especifica si la copia es superficial o profunda, mientras que en la clonación es superficial por defecto. Para realizar copia profunda con la clonación es necesario redefinir el método que el lenguaje proporciona para clonar.

14. Copia defensiva

- Para impedir que, mediante métodos consultores, sea posible alterar el estado interno de un objeto sin utilizar los métodos designados para ello, es recomendable realizar una copia defensiva de los objetos que se devuelven y que no sean inmutables.
- Para ello es necesario realizar copias profundas y no solo copias superficiales.
- En el ejemplo de las páginas 40 y 41 la lista de números no es inmutable porque existen métodos para poder alterarla. Se debería duplicar la lista y devolver esa copia en el consultor *getNumeros()*.

14. Copia defensiva

Solución 1 al ejemplo de las páginas 40 y 41 (**sigue siendo un problema)

```
class ComplejaSegura {  
    private ArrayList<Numero> numeros;  
    public ComplejaSegura() { numeros=new ArrayList(); }  
    public void add(Numero n) {numeros.add(n);}   
    ArrayList<Numero> getNumeros() {  
        return (ArrayList<Numero>)(numeros.clone());}  
}
```



```
uso  
ComplejaSegura c1=  
    new ComplejaSegura();  
c1.add(new Numero(3));  
c1.add(new Numero(2));  
c1.add(new Numero(1));  
c1.getNumeros().clear();
```

```
for(Numero n: c1.getNumeros()) {  
    System.out.println(n);  
} // Resultado: 3 2 1  
  
c1.getNumeros().get(0).inc();
```

14. Copia defensiva

- En la copia profunda hay que llegar a nivel requerido en cada caso.
- En el ejemplo anterior no solo hay que duplicar la lista sino también los elementos de la lista. En caso contrario ambas listas compartirán las referencias a los mismos objetos.
- A este tipo de copia en profundidad se le llama también copia defensiva.

14. Copia defensiva

Solución 2 al ejemplo

```
class Numero implements Cloneable{  
    private Integer i;  
    public Numero(Integer a) {  
        i=a;  
    }  
    public void inc() {  
        i++;  
    }  
    @Override  
    public Numero clone() throws CloneNotSupportedException  
    {  
        //Los objetos de la clase Integer son inmutables  
        return (Numero) super.clone();  
    }  
}
```



14. Copia defensiva

Solución 2 al ejemplo (Cont.)

```
class ComplejaMasSegura implements Cloneable {  
    ArrayList<Numero> getNumeros() {  
        ArrayList nuevo= new ArrayList();  
        Numero n=null;  
        for(Numero i:this.numeros) {  
            try { n=i.clone(); }  
            catch (CloneNotSupportedException e)  
            { System.err.println("CloneNotSupportedException" ); }  
            nuevo.add(n); }  
        return (nuevo);  
    }  
    @Override  
    public ComplejaMasSegura clone() throws CloneNotSupportedException{  
        ComplejaMasSegura nuevo=(ComplejaMasSegura) super.clone();  
        nuevo.numeros= this.getNumeros();  
        return nuevo;  
    }  
}
```



Se hace un clone de cada uno de sus elementos y se añaden a la nueva lista

Puedo usar getNumeros() aquí porque realiza una copia profunda

15. Copia por serialización

- En Ruby se puede recurrir a la serialización, deserialización para crear una copia profunda

```
b = Marshal.load( Marshal.dump(a) )
```

En este proceso el objeto se convierte a una secuencia de bits y después se construye otro a partir de esta secuencia.

- Esta última técnica también es aplicable a Java y en ambos casos es **poco eficiente, no aplicable en todos los escenarios**. La documentación de Ruby advierte que el uso del método *load* puede llevar a la ejecución de código remoto.