Clase 8 – La clase Object – Colecciones – Tipos Genéricos

La Clase Object – Colecciones - 2022

La clase Object

El método equals

El método hashcode

Colecciones

Tipos de colecciones: HashSet, LinkedHashSet, TreeSet, ArayList, LinkedList, PriorityQueue.

Iterators

Excepciones comunes

Tipos genéricos

Restricción de tipos genéricos

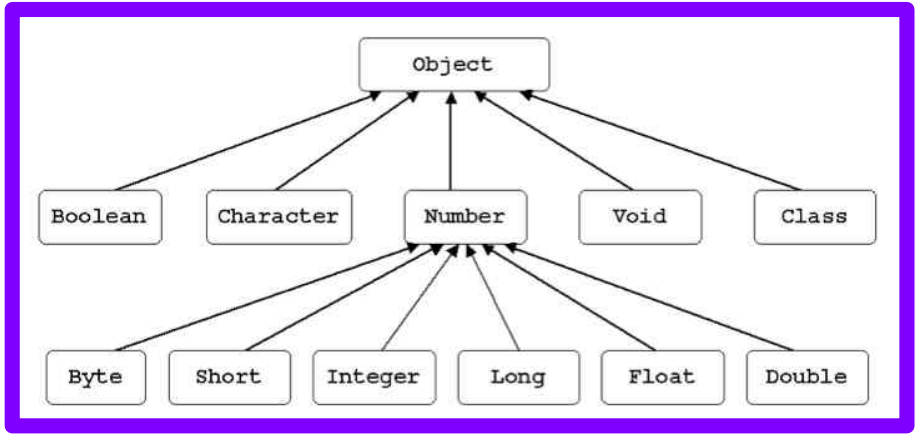
Instanciando genericidad en una subclase

Colecciones genéricas

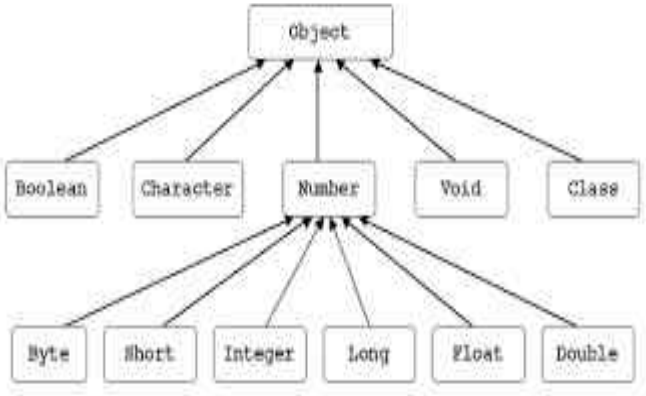
Herramientas para programación genérica

Método genérico de inserción directa

Implementación de la interfaz comparable.



La clase Object



Es la raíz de la jerarquía de clases.

Todas las clases extienden directa o indirectamente de la clase Object y por lo tanto una variable de tipo Object puede referirse a cualquier objeto, sea este la instancia de una clase o un array.

La clase object define algunos métodos que son heredados por todos los objetos:

```
public boolean equals(Object obj)
public int hashcode()
protected Object clone() thows CloneNotSupportedException
public final Class getClass()
protected void finalize() throws Throwable
public String toString()
```

public boolean equals(Object obj)

Compara el objeto receptor con el objeto referido por obj para ver si son iguales, devolviendo true si tienen el mismo valor y false si no es así.

Si deseamos determinar si dos **referencias** se refieren al mismo objeto, podemos compararlas utilizando **==** o **!=**.

El método equals pregunta por la igualdad de valores. La implementación por defecto supone que un objeto sólo es igual a sí mismo, comprobando si **this == obj**.

```
public class Socio
{
    private int legajo;
    private String nya;

    public Socio(int legajo, String nya)
    {
        super();
        this.legajo = legajo;
        this.nya = nya;
    }

    @Override
    public boolean equals(Object object)
    {
        if (this == object)
        {
            return true;
        }
        if (!(object instanceof Socio))
        {
            return false;
        }
        final Socio other = (Socio) object;
        if (legajo != other.legajo)
        {
            return false;
        }
        return true;
    }
```

public boolean equals(Object obj)

```java
public class Socio
{
    int legajo;
    String nombre;

    public Socio(int legajo, String nombre)
    {
        this.legajo = legajo;
        this.nombre = nombre;
    }

    @Override
    public boolean equals(Object object)
    {
        if (this == object)
        {
            return true;
        }
        if (!(object instanceof Socio))
        {
            return false;
        }
        final Socio other = (Socio) object;
        if (legajo != other.legajo)
        {
            return false;
        }
        return true;
    }
```

```java
public static void main(String[] args)
{
    Socio s1 = new Socio(1000, "Pedro");
    Socio s2 = new Socio(1001, "Juan");
    Socio s3 = new Socio(1000, "Marta");
    Socio s4 = s2;
    System.out.println("s1 es igual a s3 ?: "+s1.equals(s3));
    System.out.println("s1 es igual a s2 ?: "+s1.equals(s2));
    System.out.println("s2 es igual a s3 ?: "+s2.equals(s3));
    System.out.println("s2 es igual a s4 ?: "+s2.equals(s4));
}
```

```
s1 es igual a s3 ?: true
s1 es igual a s2 ?: false
s2 es igual a s3 ?: false
s2 es igual a s4 ?: true
```

`public int hashcode()`

HashCode es un identificador de 32 bits, cuyo valor depende del estado del objeto o de algunos de sus campos

`HashCode()` permite recuperar este valor.

Se sobrescribe el método para que se comporte de forma acorde a la que lo hace .**`equals()`**, es decir, si el método .**`equals()`** dice que dos objetos son iguales, estos han de tener el mismo valor hash.

public int hashcode()

Hashcode para objetos de tipo **`String`** `(primeras versiones)` $h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

Hashcode para un objeto Empleado basándonos en su ID, el hash code de su nombre y de su departamento

```java
public class Empleado {
    int         IDempleado;
    String      nombre;
    Departamento dept;

    // Métodos de la Clase

    @Override
    public int hashCode() {
        int hash = 1;
        hash = hash * 17 + IDempleado;
        hash = hash * 31 + nombre.hashCode();
        hash = hash * 13 + ((dept == null) ? 0 : dept.hashCode());
        return hash;
    }
}
```

Consideraciones

Los métodos *hashCode* y *equals* se pueden redefinir si se desea proporcionar una **noción de igualdad diferente** de la implementación por defecto.

Si nuestra clase tiene una noción de igualdad en la que dos objetos diferentes pueden ser equal, los dos objetos deberían devolver el mismo valor de hashcode.

Por ejemplo, la clase String redefine equals devolviendo true si dos objetos String tienen el mismo contenido, y redefine también a hashcode para que devuelva un hash basado en el contenido del String, de forma que dos cadenas con el mismo contenido tengan el mismo hashcode

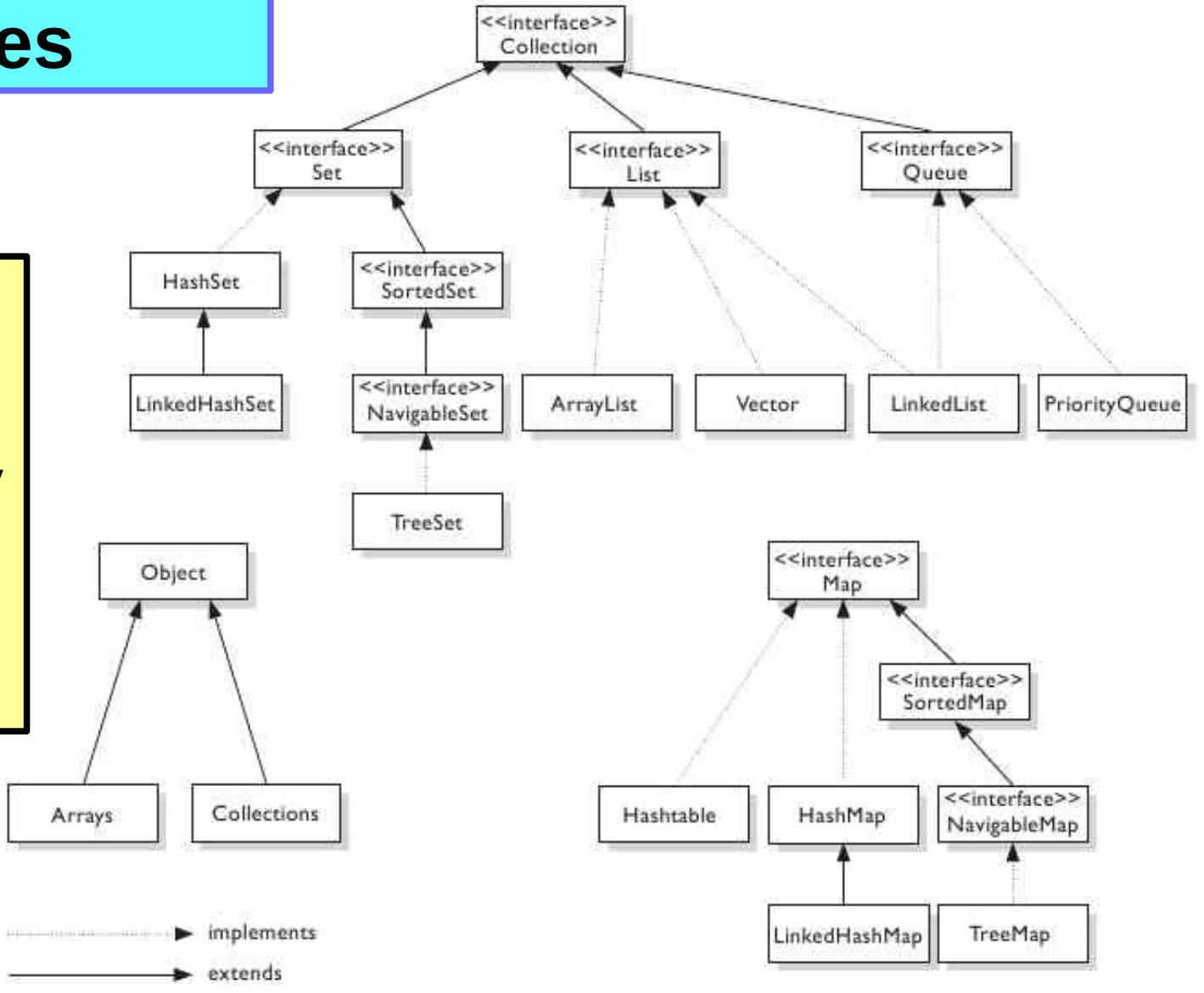
Equals y Hashcode

```java
public class Socio
{
    int legajo;
    String nombre;

    public Socio(int legajo, String nombre)
    {
        this.legajo = legajo;
        this.nombre = nombre;
    }

    @Override
    public boolean equals(Object object)
    {
        if (this == object)
        {
            return true;
        }
        if (!(object instanceof Socio))
        {
            return false;
        }
        final Socio other = (Socio) object;
        if (legajo != other.legajo)
        {
            return false;
        }
        return true;
    }
```

```java
@Override
public int hashCode()
{
    final int PRIME = 37;
    int result = 1;
    result = PRIME * result + legajo;
    return result;
}
```

Colecciones

Las colecciones (contenedores) son receptáculos que permiten almacenar y organizar objetos de forma útil para un acceso eficiente.

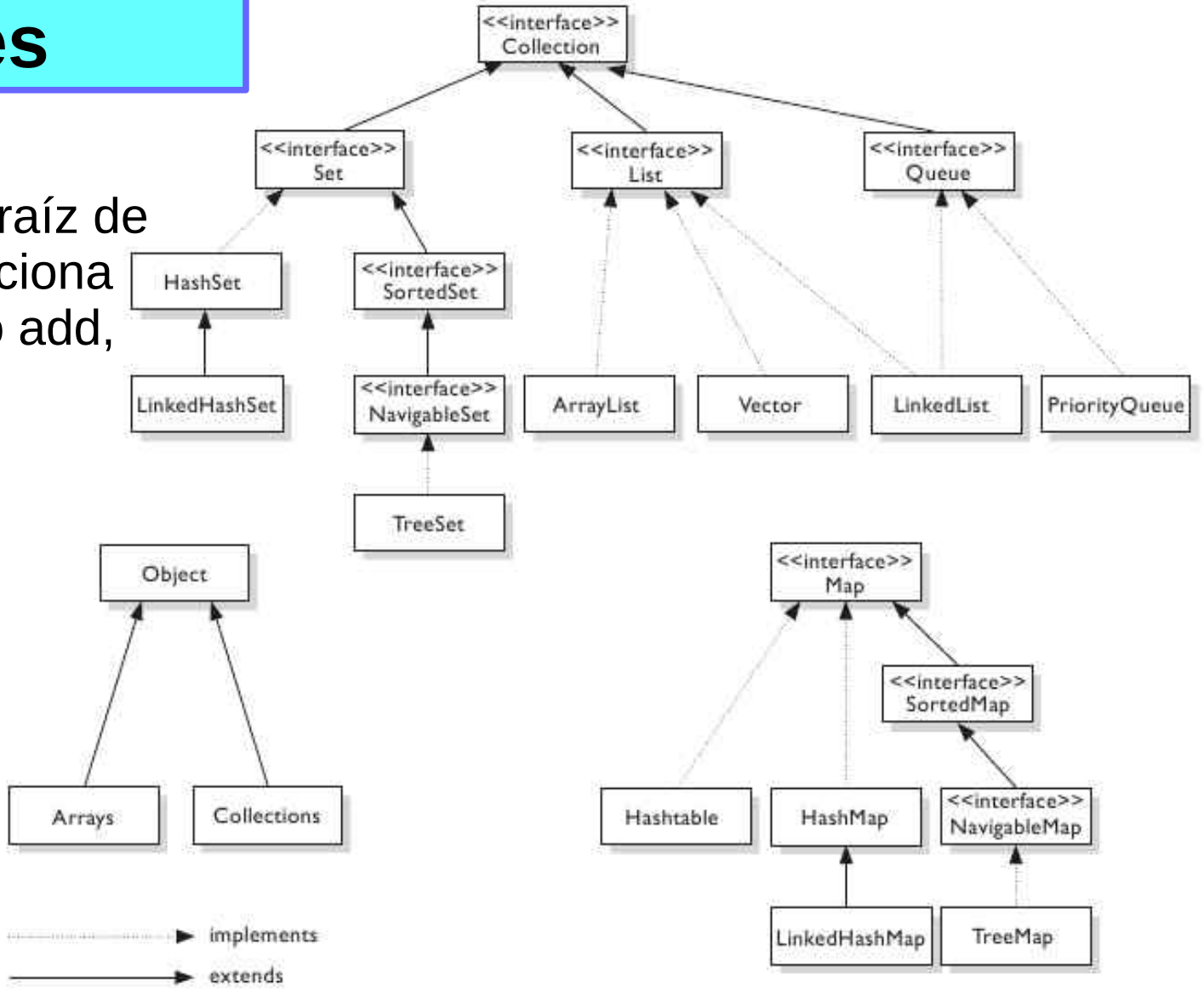


Colecciones

Collection: La interfaz raíz de las colecciones. Proporciona diversos métodos como add, remove, size e iterator (importante).

Set: Colección sin duplicados y sin orden de almacenamiento.

SortedSet: Conjunto ordenado.



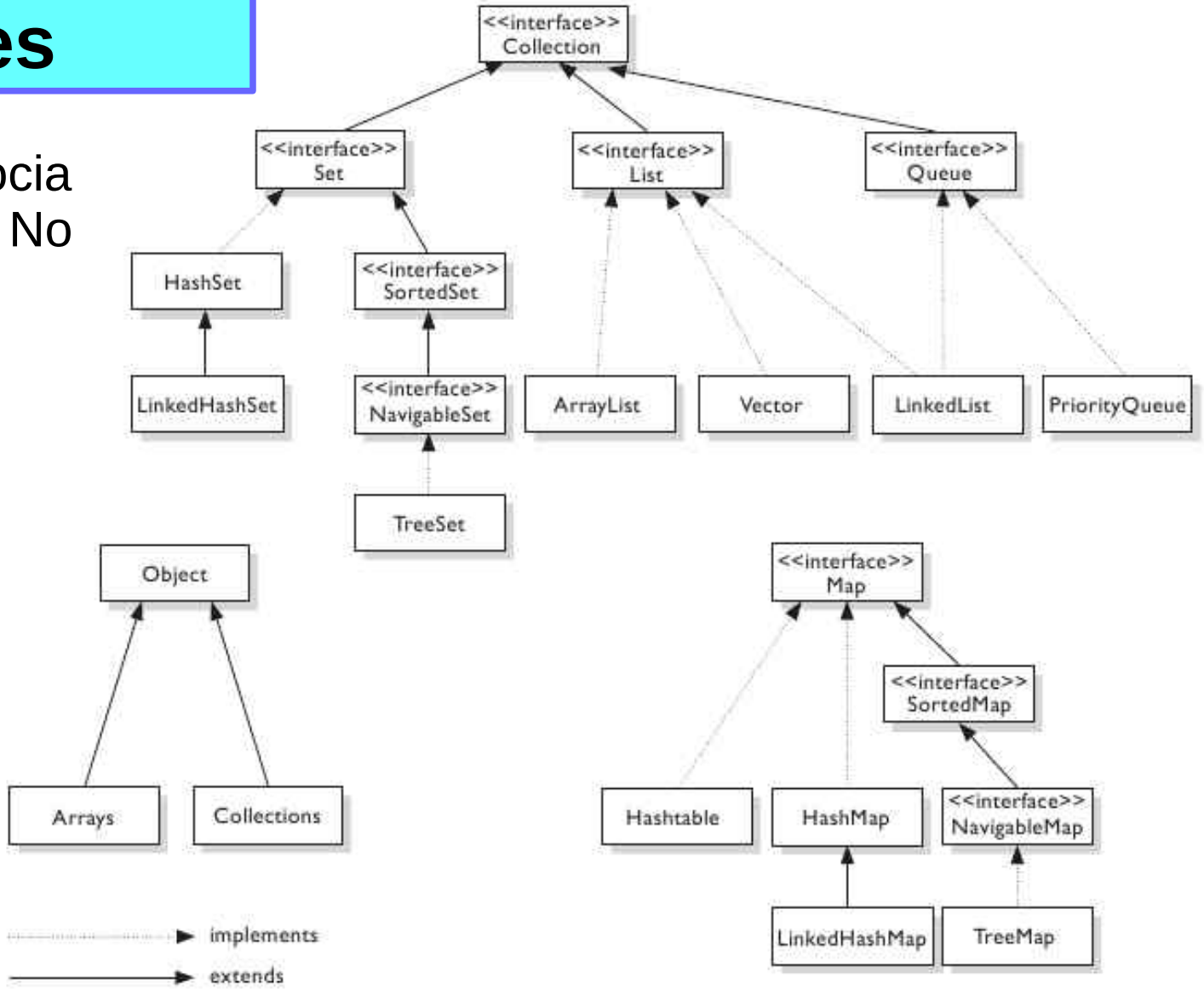
Colecciones

Map: Un mapa que asocia una clave con un valor. No extiende de Collection.

SortedMap: Un mapa cuyas claves están ordenadas (extiende a Map).

Iterator: Una interfaz que devuelve elementos de una colección de uno en uno.

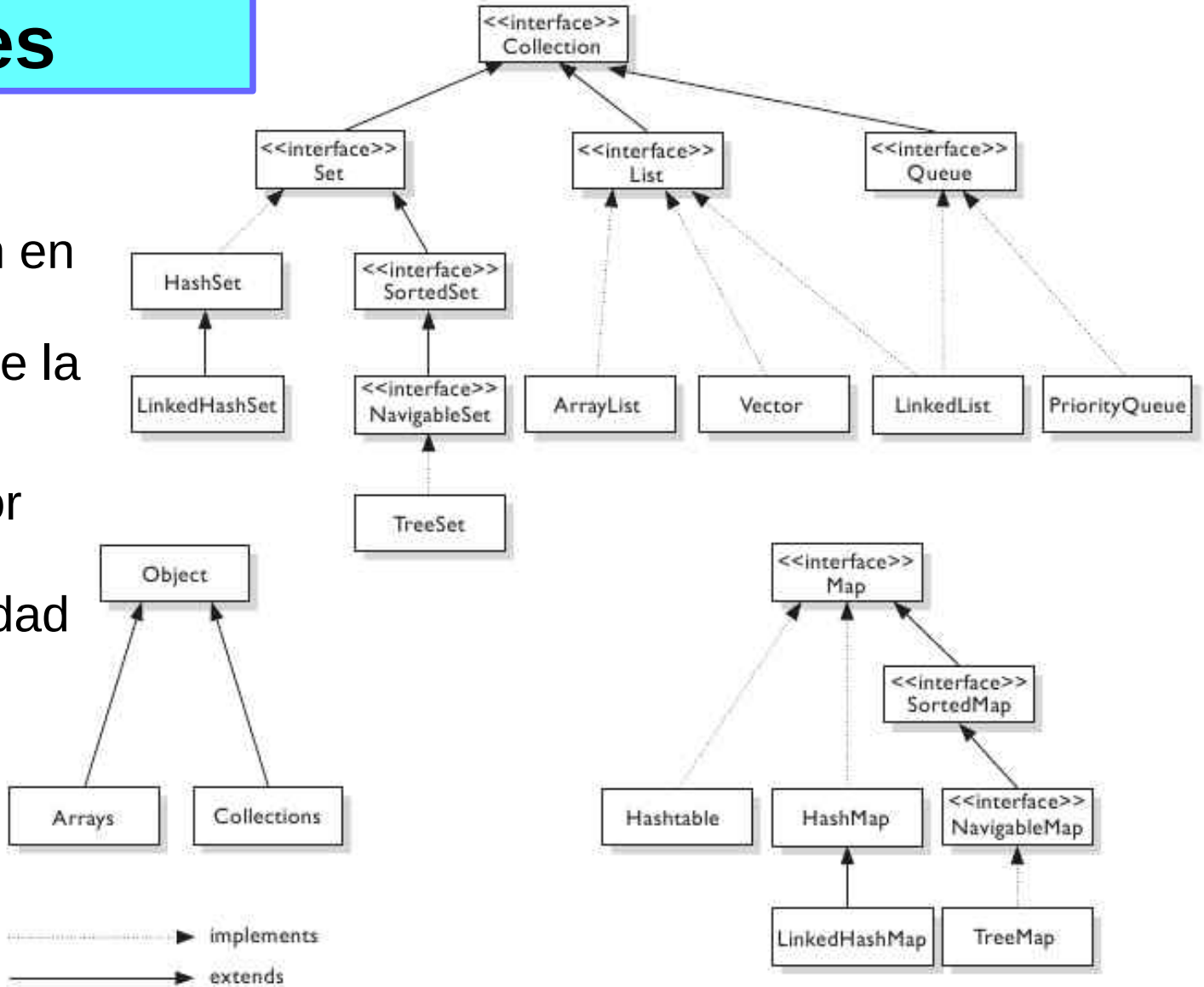
`Collection.iterator()`:



Colecciones

List: Colección cuyos elementos permanecen en un orden particular a menos que se modifique la lista.

ListIterator: Un iterador para objetos List que añade métodos de utilidad relacionados con listas.



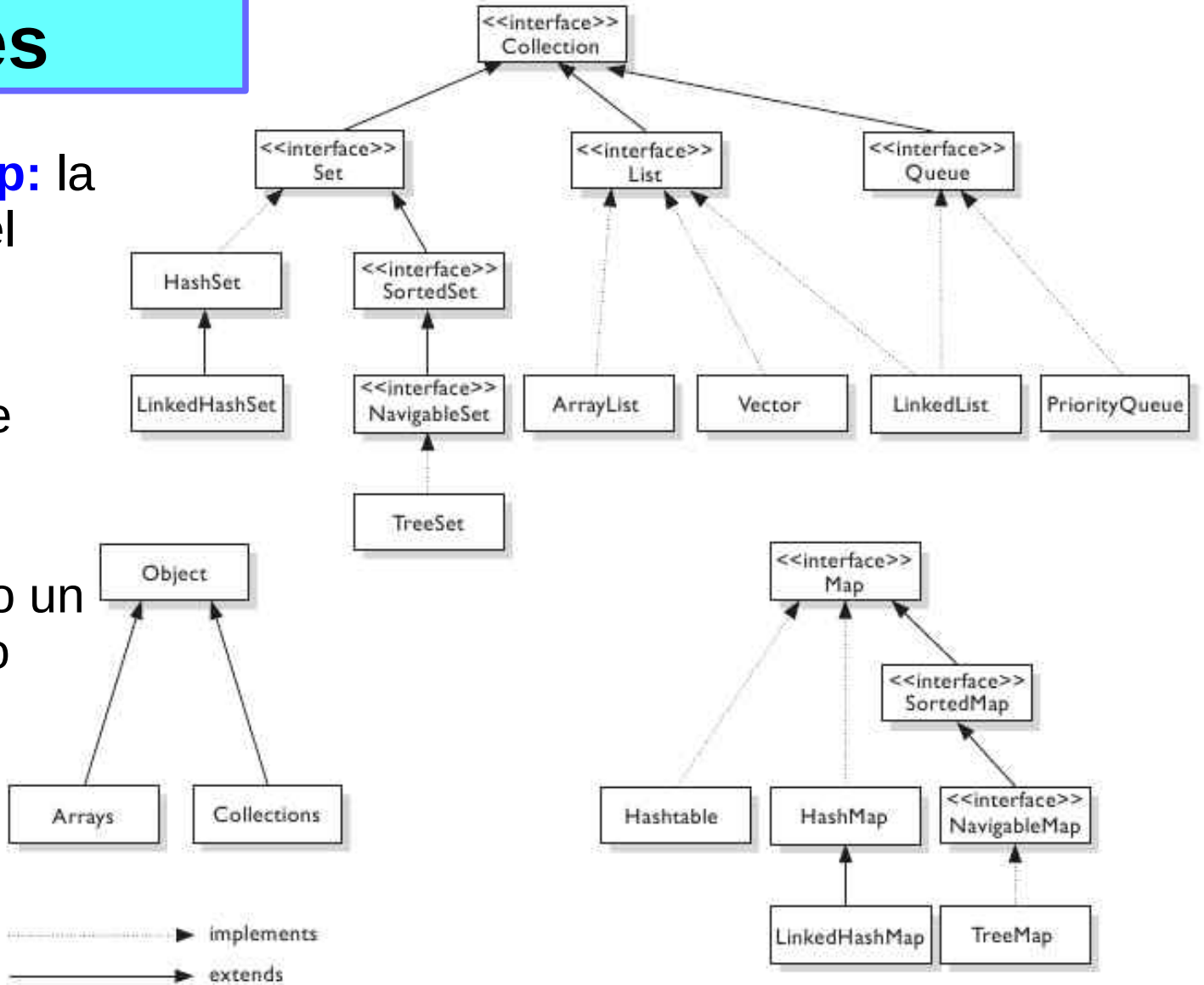
Colecciones

SortedSet y SortedMap: la iteración se realiza en el orden correspondiente.

HashSet: Un set Implementado mediante una tabla hash.

TreeSet: Un SortedSet implementado utilizando un árbol binario equilibrado

ArrayList: Una lista implementada utilizando un array de dimensión modificable.

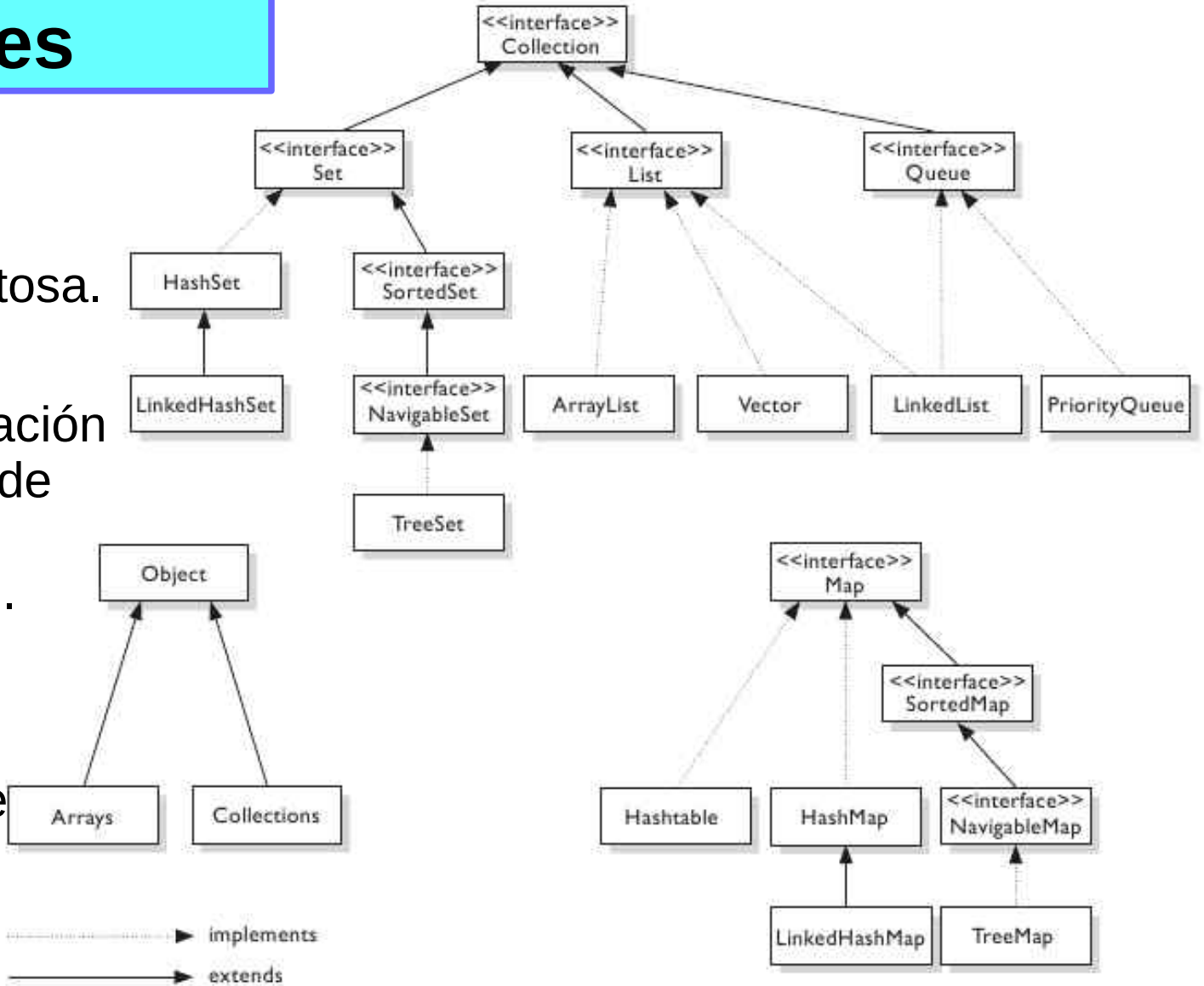


Colecciones

LinkedList: Lista doblemente enlazada. Modificación poco costosa. Útil para filas.

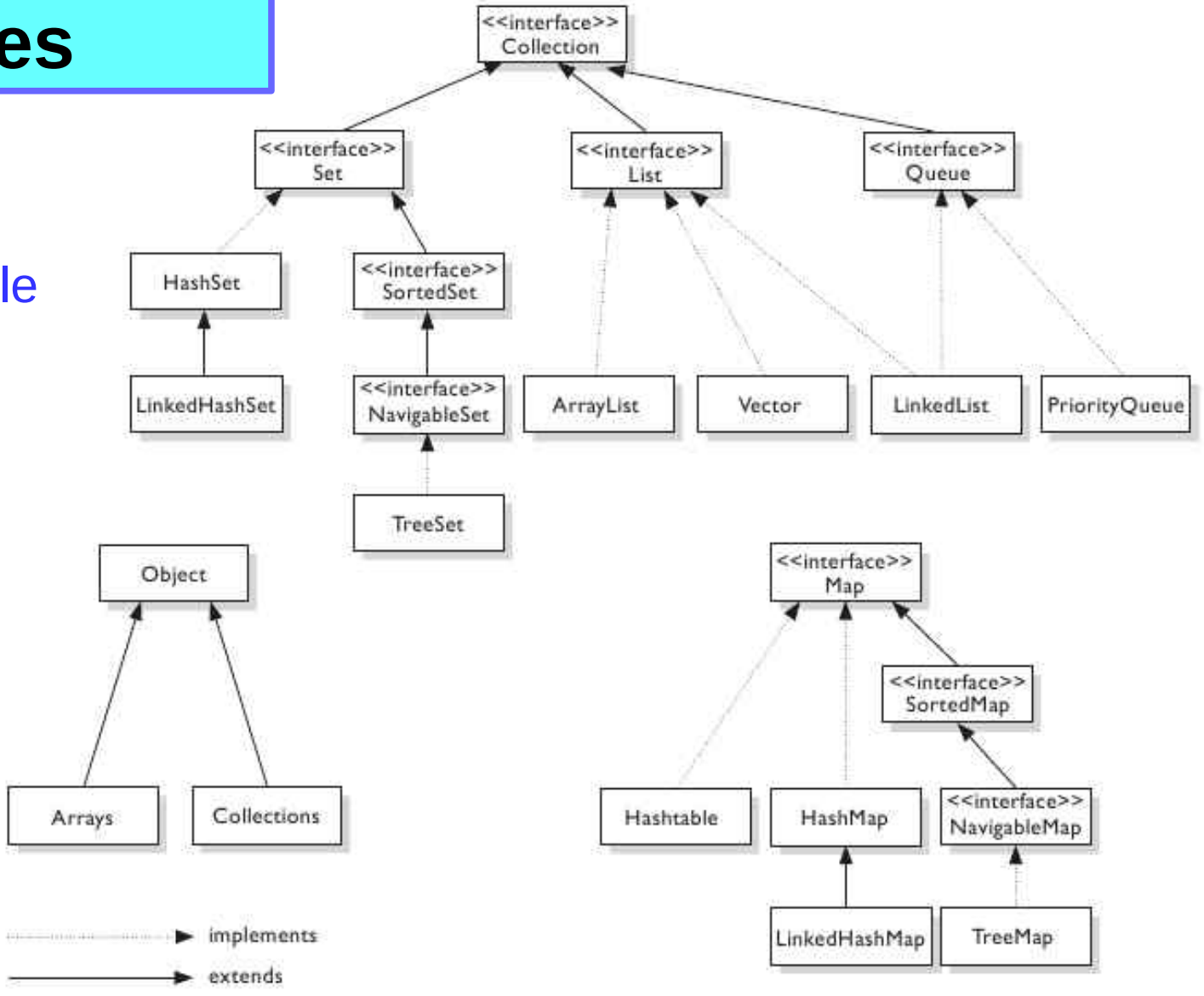
HashMap: Implementación de Map con una tabla de Hash. Rápida para búsquedas e inserción.

TreeMap: SortedMap utilizando un árbol binario equilibrado que mantiene sus elementos ordenados por clave.



Colecciones

Todas estas implementaciones son Cloneable y Serializable



Colecciones - excepciones comunes

UnsupportedOperationException: los métodos opcionales en la implementación de una interfaz lanzan esta excepción cuando no están implementados.

ClassCastException: Se lanza cuando el elemento a agregar a la colección no es del tipo apropiado para la colección.

IllegalArgumentException: Idem anterior pero considerando el valor, no el tipo.

NoSuchElementException: Se lanza al retornar un elemento cuando la colección está vacía.

NullPointerException: se lanza cuando se recibe por parámetro una referencia null.

Ver Gosling: pág 409

Colecciones - iteración

La interfaz Collection define un método iterador (**`iterator()`**) que devuelve un **objeto que implementa** la interfaz Iterator.

`public boolean hasNext()`
devuelve true si la iteración tiene más elementos.

`public Object next()`
devuelve el siguiente elemento de la iteración. Si no hay elemento siguiente, se lanza una excepción NoSuchElementException

`public void remove()`
Elimina de la colección el elemento devuelto en último lugar por la iteración. Se puede llamar una sola vez por cada llamada a next(). Si no se cumple esto lanza la excepción IllegalStateException. (Opcional)

Colecciones - iteración

Ejemplo: eliminar de una colección todas las cadenas de texto cuya longitud es mayor que un valor determinado

```
public void eliminarStringsLargas(Collection col, int longMax)
{
        Iterator it = col.itertor();
        while(it.hasNext())
        {
                String cad = (String) it.next();
                if(cad.length() > longMax)
                        it.remove();
        }
}
```

Referencias

Se recomienda leer el capítulo 16, libro Gosling, en donde se encuentra el listado completo de colecciones con la explicación de sus métodos.

Tipos Genéricos – Programación Genérica

La Programación Genérica es un mecanismo para la reutilización de software.

El objetivo es independizar el proceso del tipo de datos sobre los que se aplica.

Ejemplo

Sea la clase Caja, con un atributo de tipo entero.

```java
public class Caja
{
    private Integer dato;

    public Caja()
    {
        super();
    }

    public Integer quita()
    {
        return dato;
    }

    public void pon(Integer d)
    {
        dato = d;
    }
}
```

```java
public class TestCaja
{
    public static void main(String args[])
    {
        Caja c = new Caja();
        c.pon(new Integer(46));
        Integer x = c.quita();
    }
}
```

Ejemplo

Es posible generalizar la Caja para que permita trabajar con cualquier tipo de datos.

```java
public class Caja
{
    private Object dato;

    public Caja()
    {
        super();
    }

    public Object quita()
    {
        return dato;
    }

    public void pon(Object d)
    {
        dato = d;
    }
}
```

```java
public class TestCaja
{
    public static void main(String args[])
    {
        Caja c = new Caja();
        c.pon(new Manzana());
        Manzana x = (Manzana) caja.quita();
    }
}
```

La utilización de diversos tipos de datos únicamente puede detectarse en tiempo de ejecución.

Requiere un Casting tras obtener el valor de la caja.

Ejemplo

A partir de la versión 1.5, Java introduce mecanismos explícitos para gestionar la genericidad

```java
public class Caja<T>
{
    private T dato;

    public Caja()
    {
        super();
    }

    public T quita()
    {
        return dato;
    }

    public void pon(T d)
    {
        dato = d;
    }
}
```

```java
public class TestCaja
{
    public static void main(String args[])
    {
        Caja<Integer> caja = new Caja<Integer>();
        caja.pon(new Integer(46));
        Integer x = caja.quita();
    }
}
```

Permite la comprobación de tipos correctos en tiempo de compilación.

Declaración explícita del tipo de datos a utilizar.

No es necesario el casting tras obtener el valor.

Entonces...

El uso de Object como una referencia genérica **es potencialmente inseguro** y no se puede hacer nada para que el programador no cometa un error equivocando el tipo de objeto esperado. El cual es descubierto al momento de realizar un casteo y lanzarse la excepción ClassCastException.

Se pueden declarar clases genéricas, que luego se especialicen (para un determinado objeto de dato) al momento de usarla.

Mensaje de error

Permite la detección de errores de tipos en tiempo de compilación y evita algunos errores de ejecución.

```
2  public class Caja<T>
3  {
4      private T dato;
5
6      public Caja()
7      {
8          super();
9      }
10
11     public T quita()
12     {
13         return dato;
14     }
15
16     public void pon(T d)
17     {
18         dato = d;
19     }
20 }
```

```
23 public class TestCaja
24 {
25     public static void main(String args[])
26     {
27         Caja<Integer> caja = new Caja<Integer>();
28         caja.pon("HOLA !! ");
29         Integer x = caja.quita();
30     }
31 }
```

ERROR

Mensaje del compilador: *pon(java.lang.Integer) in Caja<java.lang.Integer> cannot be applied to (java.lang.String).*

Para tener en cuenta

Una declaración de tipos genéricos puede tener múltiples tipos parametrizados, separados por comas.

```
public class CajaDoble<T, Q>
```

Todas la invocaciones de clases genéricas son expresiones de **una clase**.

Al instanciar una clase genérica **no se crea** una nueva clase.

```
Caja<String> caja1 = new Caja<String>();

Caja<Integer>caja2 = new Caja<Integer>();

boolean iguales = ( caja1.getClass() == caja2.getClass() );

//la variable iguales es true (no se crea una nueva clase)
```

Consecuencias

No se puede usar **T** (el tipo de datos parametrizado) como tipo de un campo **estático** o en cualquier lugar dentro de un método estático o inicializador estático.

Dentro de una definición de clases genérico, un tipo parametrizado puede aparecer en cualquier declaración no estática donde se pondría cualquier tipo de datos concreto

Consecuencias

No se puede usar un tipo de datos parametrizado en la **creación de objetos y arreglos**, por ejemplo:

```
class ListaSimple<E>
{//...
   public E[] toArray()
   {
      int size = 0;
      for(Nodo<E> nodo = primero; nodo != null; nodo = nodo.getSiguiente())
         size ++;
      E[] arreglo = new E[size];  //INVALIDO: no compila
      //... Copiar los datos...
      ...
   }
```

Consecuencias

No se puede instanciar o crear un array de **T** por la misma razón que no se puede crear un campo estático de tipo **T**.

No hay manera de que el compilador sepa el tipo de datos en que va a instanciarse **T**.

Esta forma de notación puede pensarse como una forma sintética y optimizada de castear un tipo Object.

Consecuencias – en síntesis

1. Los parámetros de tipo no pueden ser instanciados

No es posible crear una instancia de un parámetro de tipo. Por ejemplo, considere esta clase:

```
01.  //No se puede crear una instancia de T
02.      class Gen<T>{
03.          T ob;
04.          Gen(){
05.              ob= new T(); //Ilegal;
06.          }
07.      }
```

Aquí, es ilegal intentar crear una instancia de T. La razón debería ser fácil de entender: el compilador no tiene forma de saber qué tipo de objeto crear. T es simplemente un marcador de posición.

Habría que asignar valor al atributo **`T ob`** a través de un setter.

Consecuencias – en síntesis

2. Restricciones en miembros estáticos

Ningún miembro estático puede usar un parámetro de tipo declarado por la clase envolvente. Por ejemplo, ambos miembros estáticos de esta clase son ilegales:

```
01.  class Incorrecto<T>{
02.          //Incorrecto, no hay variables estáticas de tipo T.
03.          static T ob;
04.          //Incorrecto, no hay métodos estáticos de tipo T.
05.          static T getOb(){
06.              return ob;
07.          }
08.      }
```

Consecuencias – en síntesis

3. Restricciones de arrays genéricos

No se puede instanciar una array cuyo tipo de elemento es un parámetro de tipo.

```
03.  class Gen <T extends Number>{
04.      T ob;
05.      T vals[]; //Ok
06.
07.      Gen(T o,T[] nums){
08.          ob=o;
09.
10.          //Esta declaración es ilegal
11.          // vals=new T[10]; //No se puede crear un array de T
12.
13.          //Esta declaración es Ok
14.          vals=nums; //OK para asignar referencia al array existente
15.      }
16.  }
```

Consecuencias – en síntesis

3. Restricciones de arrays genéricos

Hay dos restricciones genéricas importantes que se aplican a los arrays.

Primero, no puede instanciar una array cuyo tipo de elemento es un parámetro de tipo.

En segundo lugar, no puede crear un array de referencias genéricas específicas de tipo. El siguiente programa corto muestra ambas situaciones:

Consecuencias – en síntesis

```
public class Gen<T extends Number>
{
    T ob;
    T vals[];
    public Gen(T o, T[] nums)
    {
        ob = o;
        vals = nums;
        vals = new T[10]; //Error. No se puede crear un array de T
    }
    public static void main(String[] args)
    {
        Integer n[] = { 1, 2, 3, 4, 5 };
        Gen<Integer> iOb = new Gen<Integer>(50, n); // se crea un objeto de tipo Gen instanciado en Integer
        // con los parámetros correctos del constructor

        //no puede crear un array o referencias genéricas específicas de tipo
        Gen<Integer> gens[]=new Gen<Integer>[10]; //Error.
        //el new es para el arreglo o para Gen ??

        Gen<?> gens2[] = new Gen<?>[10]; //Ok. Se crea un arreglo de un tipo de datos que contiene genéricos.
        Integer p[]={1,4,5};
        gens2[0] = new Gen <Integer>(20,p); // se crean los objetos del arreglo y se los asigna
    }
}
```

Restricción de Tipos Genéricos

Es posible restringir el tipo genérico para trabajar con un tipo específico (y sus subtipos).

```
public class CajaNumeros<T extends Number>
{
    private T dato;

    public T quita()
    {
        return dato;
    }

    public void pon(T dato)
    {
        this.dato = dato;
    }
}
```

```
CajaNumeros<Double> caja = new CajaNumeros<Double>(); //OK
CajaNumeros<String> caja2 = new CajaNumeros<String>(); //ERROR
```

Instanciando la Genericidad en una Subclase

```java
public class Caja<T>
{
    protected T dato;

    public Caja()
    {
        super();
    }

    public T quita()
    {
        return dato;
    }

    public void pon(T d)
    {
        dato = d;
    }
}
```

```java
public class Manzana
{
    private int sabor;

    public Manzana(int sabor)
    {
        this.sabor = sabor;
    }

    public int getSabor()
    {
        return sabor;
    }
}
```

Es posible particularizar una clase genérica en una subclase para que utilice un tipo de datos determinado.

La clase CajaManzanas deja de ser genérica.

```java
public class CajaManzanas extends Caja<Manzana>
{
    public int getSabor()
    {
        return dato.getSabor();
    }
}
```

Colecciones genéricas

Las colecciones son genéricas

Collection<E>

Set<E>

Map<K,V>
- HashMap<K,V>
- HashTable<K,V>
- LinkedHashMap<K,V>
- TreeMap<K,V>

List<E>
- ArrayList<E>
- Vector<E>
- LinkedList<E>

Queue<E>

La interface **`Comparable<T>`** también !

Teórico práctico

- ✔ Programación Genérica a Partir de la Herencia
- ✔ Reutilizar código requiere:
  - Independizar un algoritmo del tipo de datos sobre el que se aplica, i.e. realizar una Programación Genérica.

Ejemplo: Problema de la Ordenación de un array de elementos

Diversos algoritmos de ordenación:

Inserción directa

Selección directa

Intercambio directo

Se debe conocer el tipo base del array si es primitivo, como int, u objeto como Figura o Integer.

Inserción Directa: Dos Métodos Según el Tipo

```java
// Ordenación del array int a[]
for( int i = 1; i < a.length ; i++ )
{
   int elemAInsertar = a[i];
   int posIns = i ;
   for(; posIns>0 && elemAInsertar < a[posIns-1]; posIns--)
       a[posIns]=a[posIns-1];
   a[posIns] = elemAInsertar;
}

// Ordenación del array Figura a[]
for( int i = 1; i < a.length ; i++ )
{
   Figura elemAInsertar = a[i];
   int posIns = i ;
   for(; posIns>0 && elemAInsertar.area() < a[posIns-1].area(); posIns--)
       a[posIns]=a[posIns-1];
   a[posIns] = elemAInsertar;
}
```

Herramientas para Programación Genérica

¿Qué haría falta para desarrollar un método genérico?

- Un tipo o clase genérica compatible con el de los elementos a ordenar.
- Un tipo o clase genérica con un único método abstracto que permita comparar objetos compatibles entre sí:
  - La interfaz **`Comparable<E>`**, cuyo único método es:

```
int compareTo(E o)
```

Método Genérico de Inserción Directa

Un código genérico de ordenación requiere:

- Restringir el tipo de datos de los elementos del vector a aquellos que sean comparables.
- La clase de los elementos del vector debe implementar la interfaz **`Comparable<T>`.**

```
public static <T extends Comparable<T>> void insercionDirecta( T a[] )
{
   for( int i = 1; i < a.length ; i++ )
   {
       T elemAInsertar = a[i];
       int posIns = i ;
       for(; posIns>0 && elemAInsertar.compareTo(a[posIns-1]) < 0; posIns--)
       {
           a[posIns]=a[posIns-1];
       }
       a[posIns] = elemAInsertar;
    }
}
```

Uso de insercionDirecta

Uso del método estático insercionDirecta en la clase Ordenacion del paquete ordenacionArray:

```
public class TestOrdenacionInteger
{
        public static void main(String args[])
        {
                Integer a[] = new Integer[4];
                .... //Rellenar el vector a
                Ordenacion.insercionDirecta(a);
                .... //El vector a está ordenado
        }
}
```

La clase base (o tipo) del array debe implementar el interfaz Comparable<T>, proporcionando código al método compareTo.

Implementación de la Interfaz Comparable

```
public abstract class Figura implements Comparable<Figura>
{
    ...
    public int compareTo(Figura otra)
    {
        int rta = 0;
        double areaf = otra.area();
        double areathis = this.area();
        if ( this.area() < otra.area() )
            rta = -1;
        else if ( this.area() > otra.area() )
            rta = 1;
        return rta;
    }
}
```