

Tipos genéricos y el Framework de Colecciones de JAVA

- ① Tipos genéricos
- ② Las interfaces Centrales
 - Collection**
 - **Set y SortedSet**
 - **List**
 - **Queue**
 - Map y SortedMap**
- ③ Las interfaces Secundarias
 - Iterator y ListIterator**
- ④ Las implementaciones convencionales
 - HashSet, HashMap, ArrayList, LinkedList, TreeSet, TreeMap, etc.**

Tipos Genéricos

Introducción

- En JAVA se denomina **Genéricos** a la capacidad del lenguaje de definir y usar tipos (clases e interfaces) y métodos genéricos.
- Los tipos y métodos genéricos difieren de los "regulares" porque contienen tipos de datos como parámetros formales.
- Los **tipos genéricos** se incorporaron en el JAVA para proveer chequeo de tipos en compilación.

`ListaEnlazadaGenerica<T>` es un tipo genérico

`T` es un **parámetro formal** que denota el tipo

Los elementos que se almacenarán en la lista serán de algún tipo `T`

- Los **tipos parametrizados** se forman al asignarle tipos reales a los parámetros formales que denotan un tipo de datos.

`ListaEnlazadaGenerica<String> lista;`

- El **framework de colecciones** del paquete `java.util` es genérico a partir de Java 5.0


Tipos Genéricos

Qué problemas resuelven los genéricos?

Se evitan los errores en ejecución causados por el uso de *casting*.

```
List list = new ArrayList();  
list.add("abc");  
list.add(new Integer(5));  
  
for(Object obj : list){  
    String str=(String)obj;  
}
```

El casteo de tipos dispara el siguiente error en ejecución:
ClassCastException



Los “Genéricos” proveen una mejora para el sistema de tipos: permite operar sobre objetos de múltiples tipos y provee seguridad en compilación pudiendo detectar *bugs* en compilación.

```
List<String> list1 = new ArrayList<String>();  
list1.add("abc");  
//list1.add(new Integer(5)); //error de compilación  
for(String str: list1){  
    System.out.print(str.length());  
}
```

No es necesario
hacer casting



La programación genérica produce código altamente general y reusable.

Tipos Genéricos

Declaración

Un tipo genérico es un tipo referencial que tiene uno o más tipos de datos como parámetros. En la definición de los **tipos genéricos** la sección correspondiente a los parámetros continúa al nombre del tipo (clase, interface). Es una lista separada por comas y delimitada por los símbolos `<>`.

```
package genericos.definicion;
public class ParOrdenado <X,Y> {
    private X a;
    private Y b;
    public ParOrdenado (X a, Y b){
        this.a=a;
        this.b=b;
    }
    public X getA() {
        return a; }
    public void setA(X a) {
        this.a = a; }
    public Y getB() {
        return b; }
    public void setB(Y b) {
        this.b = b; }
}
```

El alcance de los identificadores **X** e **Y** es toda la clase **ParOrdenado**.

En el ejemplo, **X** e **Y** son usados en la declaración de variables de instancia y en los argumentos y tipos de retorno de los métodos de instancia.


Tipos Genéricos

Instanciación

Para usar un **tipo genérico** se deben especificar los **argumentos** que reemplazarán a los **parámetros formales** que denotan tipos de datos. Los argumentos pueden ser referencias a **tipos concretos** como String, Long, Date, etc o también **instanciaciones comodines**.

```
package genericos.definicion;
public class TestParOrdenado {
public static void main(String[] args){
    ParOrdenado<String, Long> par = new ParOrdenado<>("hola", 23L);
    System.out.println("(" + par.getA() + ", " + par.getB() + ")");
    ParOrdenado<String,Color> nombreColor=
        new ParOrdenado<String,Color>("Rojo",Color.RED);
    System.out.println("(" + nombreColor.getA() + ", " + nombreColor.getB() + ")");
    ParOrdenado<Double,Double> coordenadas = new ParOrdenado<Double,Double>(17.3,42.8);
    System.out.println("(" + coordenadas.getA() + ", " + coordenadas.getB() + ")");
}
}
```

Autoboxing: el 23 es automáticamente convertido a Long



La **instanciación** de ParOrdenado<String,Long>, ParOrdenado<String,Color>, ParOrdenado<Double,Double> son **tipos parametrizados concretos** y se usan como un tipo regular. Podemos usar **tipos parametrizados** como argumentos de métodos, para declarar variables y en la expresión **new** para crear un objeto.

Tipos Genéricos

Seguridad de tipos

En JAVA se considera que un programa es seguro respecto al tipado si compila sin errores ni advertencias y en ejecución NO dispara ningún **ClassCastException**.

Un programa bien formado permite que el compilador realice suficientes chequeos de tipos basados en información estática y que no ocurran errores inesperados de tipos en ejecución. **ClassCastException** sería un error inesperado de tipos que se produce en ejecución sin ninguna expresión de casting visible en el código fuente.

→ los tipos genéricos permiten hacer una detección temprana de errores de tipo.

El framework de Colecciones

Una colección es un objeto que representa a un grupo de objetos. Se usa para almacenar, recuperar y manipular un conjunto de datos.

Un *framework* de colecciones permite representar y manipular colecciones de una manera unificada, independientemente de los detalles de implementación. El *frameworks* de colecciones de JAVA cuentan con:

- **Interfaces:** son tipos de datos abstractos que representan colecciones y que permiten manejarlas en forma independiente de su implementación. Forman jerarquías de herencia.
- **Implementaciones:** son implementaciones concretas de las interfaces. Son estructuras de datos.
- **Algoritmos:** son métodos de clase que realizan operaciones útiles (búsquedas y ordenamientos) sobre objetos que implementan alguna de las interfaces de colecciones.

Java incluye en su librería, implementaciones de las estructuras de datos más comunes. Esta parte de la librería es conocida como API de colecciones.

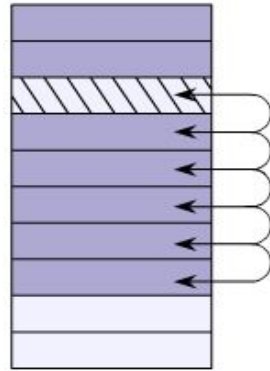
Colecciones

Tecnologías de almacenamiento

Existen cuatro tecnologías de almacenamiento básicas disponibles para almacenar objetos: arreglo, lista enganchada, árbol y tabla de hash.

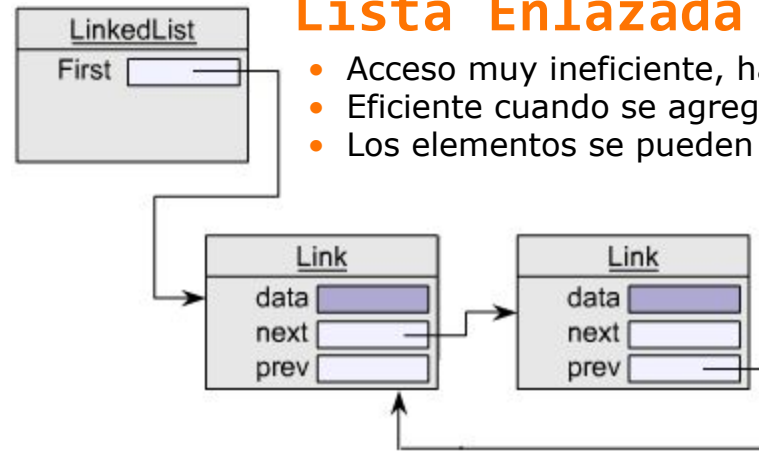
Arreglo

- El acceso es muy eficiente.
- Es ineficiente cuando se agrega/elimina un elemento.
- Los elementos se pueden ordenar.



Lista Enlazada

- Acceso muy ineficiente, hay que recorrer la lista.
- Eficiente cuando se agrega/elimina un elemento.
- Los elementos se pueden ordenar.



Arbol

- Almacenamiento de valores en orden ascendente.
- Búsqueda eficiente.

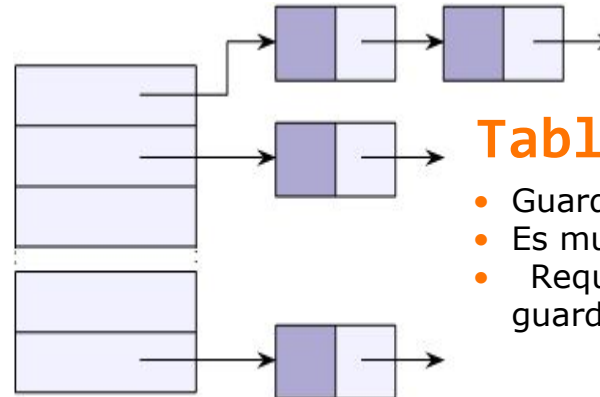
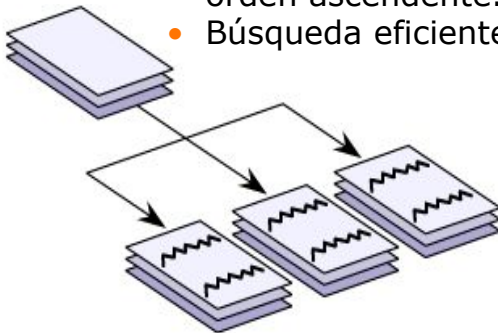


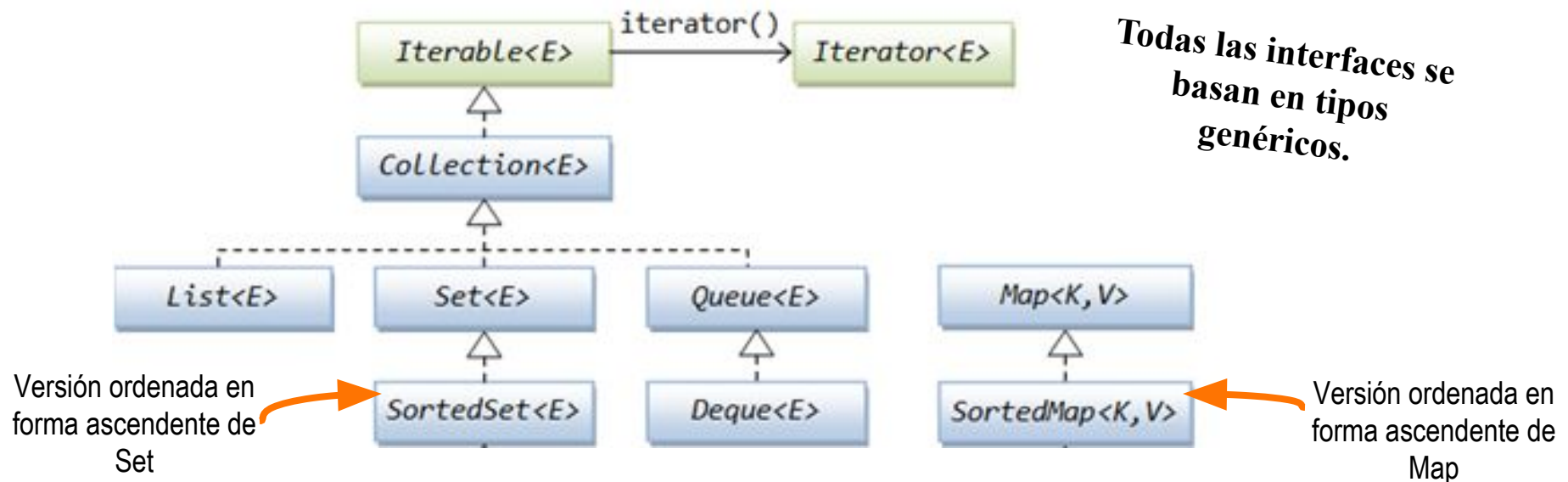
Tabla de Hash

- Guarda asociaciones (clave, valor).
- Es muy rápido, accede por clave
- Requiere memoria adicional para guardar las claves (tabla).

Colecciones

Jerarquías de Interfaces

El framework incluye una jerarquías de interfaces que podemos clasificarlas en: *interfaces centrales* en relación a que sirven para representar distintos contenedores de elementos e *interfaces secundarias* que se usan para recorrer estas colecciones.



Las interfaces centrales son `Collection` y `Map`, raíces de dos jerarquías de interfaces que constituyen el fundamento del framework de colecciones de JAVA. Estas interfaces están ubicadas en el paquete `java.util` de la distribución estándar de JAVA.

Colecciones

Implementaciones de las Interfaces

La tabla muestra las implementaciones de propósito general que vienen con la plataforma java, las cuales siguen una convención de nombre, combinando la estructura de datos subyacente con la interface del framework:

Interfaces	Tecnologías de almacenamiento				
	Tabla de Hashing	Arreglo de tamaño variable	Árbol	Lista Encadenada	Tabla de Hash + Lista Encadenada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayBlockingQueue		LinkedBlockingQueue	
Map	HashMap		TreeMap		LinkedHashMap

Implementación de SortedSet

Implementación de SortedMap

Todas las implementaciones de propósito general:

- Tienen implementado el método **toString()**, el cual retorna a la colección de una manera legible, con cada uno de los elementos separados por coma.
- Tienen por convención al menos 2 constructores el nulo y otro con un argumento **Collection**: como por ejemplo:

TreeSet() y **TreeSet(Collection c)**

LinkedHashSet() y **LinkedHashSet(Collection c)**

Colecciones

La interfaces Collection

La interface Collection representa un conjunto de objetos de cualquier tipo. Esta interface se usa cuando se necesita trabajar con grupos de elementos de una manera muy genérica. La plataforma Java no provee una implementación directa para la interface Collection, pero si para sus subinterfaces Set, List y Queue.

```
public interface Collection<E> extends Iterable<E> {  
  
    // operaciones básicas  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator iterator();  
  
    // operaciones en "masa"  
    boolean containsAll(Collection<E> c);  
    boolean addAll(Collection<E> c);  
    boolean removeAll(Collection<E> c);  
    boolean retainAll(Collection<E> c);  
    void clear();  
  
    // operaciones de Arreglos  
    Object[] toArray();  
}
```

Las clases que implementan esta interface pueden iterarse con un objeto **Iterator**.

devuelve un iterador para recorrer los elementos de la colección. La interface no especifica el orden en que los elementos serán recorridos.

Convierte la colección a un arreglo

Colecciones

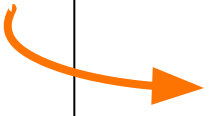
La interfaces List


Un objeto List es una secuencia de elementos donde puede haber duplicados. Además de los métodos heredados de Collection, define métodos para recuperar y modificar valores en la lista por posición como:

```
E get(int index);  
E set(int index, E element);  
boolean add(E element);  
void add(int index, E element);  
E remove(int index)
```

La plataforma java provee 2 implementaciones de la interface List, que son ArrayList y LinkedList

```
import java.util.*;  
public class DemoIterador{  
    public static void main(String[] args){  
        List<Integer> lista = new ArrayList<Integer>();  
        lista.add(1);  
        lista.add(new Integer(2));  
        lista.add(190);  
        lista.add(90);  
        lista.add(7);  
        lista.remove(new Integer(2));  
        System.out.print(lista.toString());  
    }  
}
```

 Se podría reemplazar por `new LinkedList<Integer>()` y todo sigue funcionando

 **salid**

[1, 190, 90, 7]

Colecciones

La interface Set

Un objeto Set es una colección que no contiene elementos duplicados. Tiene exactamente los mismos métodos que la interface Collection, pero agrega la restricción de no mantener duplicados.

La plataforma java provee implementaciones de propósito general para Set. Por ejemplo HashSet (mejor performance, almacena los datos en una tabla de hash) y TreeSet (más lento pero ordenados).

```
public class DemoIterador{
    public static void main(String[] args) {
        Set<String> instrumentos= new HashSet<String>();
        instrumentos.add("Piano");
        instrumentos.add("Saxo");
        instrumentos.add("Violin");
        instrumentos.add("Flauta");
        instrumentos.add("Flauta");
        System.out.println(instrumentos.toString());
    }
}
```

[Violin, Piano, Saxo, Flauta] **salida**

La interface Set podría ser usada para crear colecciones sin duplicados desde una colección c con duplicados. Supongamos que c tiene duplicados

**Set<String> sinDup =
new TreeSet<String>(c);**

Implementa **SortedSet**

Cambiando únicamente la instanciación por un objeto **TreeSet()**, obtenemos una colección ordenada:

Set<String> instrumentos= new TreeSet<String>();

[Flauta, Piano, Saxo, Violin]

En este caso el compilador chequea que los objetos que se insertan (add()) sean **Comparables!!**

Colecciones

La interface Map

Un objeto Map mapea claves con valores. No puede contener claves duplicadas y cada clave mapea con a lo sumo un valor.

La plataforma java provee implementaciones de propósito general para Map que se diferencian principalmente por el orden en que se mantienen los elementos en la estructura.

```
public interface Map <K,V> {  
    // Operaciones Básicas  
    V put(K clave, V valor);  
    V get(K clave);  
    V remove(K clave);  
    boolean containsKey(K clave);  
    boolean containsValue(V valor);  
    int size();  
    boolean isEmpty();  
    // Operaciones en "masa"  
    void putAll(Map <K,V> t);  
    void clear();  
    // Vistas  
    Set<K> keySet();  
    Collection<V> values();  
    . . .  
}
```

Las clases más comúnmente usadas son:

- La clase **HashMap** no garantiza el orden en que se mantienen los elementos en la colección. Un cambio en su contenido, como la inserción de un nuevo valor, puede cambiar completamente el orden del map.
- La clase **TreeMap** al ser una implementación de **SortedMap** debe mantener a sus elementos ordenados de acuerdo al "orden natural" de las claves, respetando la reglas de orden establecidas por el método `compareTo()` de la interface `Comparable` o por un `Comparator` suministrado al objeto `TreeMap`.
- La clase **LinkedHashMap** mantiene sus elementos en forma secuencial, respetando el orden en que los elementos fueron insertados a la colección.

Colecciones

La interface Map

Un objeto Map como vimos mapea claves con valores. No puede contener claves duplicadas y cada clave mapea con a lo sumo un valor. En el ejemplo se crea una **HashMap**

```
public class DemoMap {  
    public static void main(String[] args) {  
        Map<String, Integer> numeros = new HashMap<String, Integer>();  
        numeros.put("uno", new Integer(1));  
        numeros.put("dos", new Integer(2));  
        numeros.put("tres", new Integer(3));  
        System.out.println(numeros.toString())  
    }  
}
```

salida → {tres=3, uno=1, dos=2}

Cambiando únicamente la instanciación del objeto numero creando un objeto **TreeMap**, obtenemos una colección ordenada:

```
public class DemoMap {  
    public static void main(String[] args) {  
        Map<String, Integer> numeros = new TreeMap<String, Integer>();  
        . . .  
        System.out.println(numeros.toString())  
    }  
}
```

salida → {dos=2, tres=3, uno=1}

Implementa
SortedSet

En este caso el compilador chequea que los objetos que se insertan/agregan a la colección sean de tipo **Comparable**

Colecciones

La interface Queue

Un objeto **Queue** es una colección diseñada para mantener elementos que esperan por procesamiento. Además de las operaciones de **Collection**, provee operaciones para insertar, eliminar e inspeccionar elementos. No permite elementos nulos.

La plataforma java provee varias implementaciones de **Queue** como **PriorityQueue**, **DelayQueue** y **BlockingQueue** que implementan diferentes tipos de colas, ordenadas o no, de tamaño limitado o ilimitado, etc.

```
public interface Queue<E> extends Collection<E> {  
    //Recupera, pero no elimina la cabeza de la cola.  
    E peek();  
    //Inserta el elemento en la cola si es posible  
    boolean offer(E e);  
    // Recupera y elimina la cabeza de la cola.  
    E poll();  
}
```

El compilador chequea que los objetos que se agregan a la cola sean de tipo **Comparable**

```
public class DemoQueue{  
    public static void main(String[] args) {  
        PriorityQueue<String> pQueue  
            = new PriorityQueue<String>();  
        pQueue.offer("Buenos Aires");  
        pQueue.offer("Montevideo");  
        pQueue.offer("La Paz");  
        pQueue.offer("Santiago");  
        System.out.println(pQueue.peek());  
        System.out.println(pQueue.poll());  
        System.out.println(pQueue.peek());  
    }  
}
```

Buenos Aires
Buenos Aires
La Paz

Colecciones

Mecanismos para recorrerlas

Hay dos maneras de recorrer una colección :

1) Usando la construcción: **for-each**


```
ArrayList<Integer> lista = new ArrayList<Integer>();  
lista.add(1);  
lista.add(2);  
for (int i: lista)  
    System.out.println(i);  
}
```

2) Usando la interface **Iterator**

Un objeto **Iterator**, permite recorrer una colección y eliminar elementos selectivamente durante la recorrida. Siempre es posible obtener un **iterador** para una colección, dado que la interface **Collection** extiende la interface **iterable**.

```
package java.lang;  
import java.util.Iterator;  
  
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

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



Colecciones

Mecanismos para recorrerlas

Iterando sobre objetos `Collection`.

```
public class IterandoListas {  
  
    public static void main(String[] args) {  
        ArrayList<Character> lista = new ArrayList<Chacater>();  
        lista.add('1'); lista.add('6'); lista.add('H');  
        lista.add('3'); lista.add('0'); lista.add('L');  
        lista.add('7'); lista.add('A');  
        char nro='';  
  
        Iterator<Character> it1 = lista.iterator();  
        while (it1.hasNext()) {  
            nro = it1.next();  
            if (Character.isDigit(nro))  
                it1.remove();  
        }  
        it1 = lista.iterator();  
        while (it1.hasNext())  
            System.out.print(it1.next());  
    }  
}
```

Imprime **HOLA**

Si no se define al **iterador de tipo** `<Character>`, hay que castear cuando se recupera el objeto, dado que es de tipo `Object` y no lo puede asignar a un `char`.

```
for (Character dato: lista) {  
    System.out.print(dato);  
}
```

Colecciones


Mecanismos para recorrerlas

Los objetos **Map** no implementan la interface **Iterable**, pero proveen vistas como objetos **Collection**, a partir de las cuales se puede tratar a un map como una colección (*)

```
public class DemoIterador {
    public static void main(String args[]){
        Map<Integer, Alumno> tablaAlu
            = new HashMap<Integer, Alumno>();
        Alumno[] arregloAlumno =
            {new Alumno(7892, "Gomez", "Juana"),
             new Alumno(3321, "Perez", "Sol"),
             new Alumno(3421, "Rusciti", "Maria"),};
        // Se llena tablaAlu con los alumnos del arreglo
        for (Alumno unAlu: arregloAlumno)
            tablaAlu.put(unAlu.getLegajo(), unAlu);

        Collection<Alumno> listAlu = tablaAlu.values(); (*)
        for (Alumno unAlu: listAlu) {
            unAlu.setApe(unAlu.getApe().toUpperCase());
            System.out.println("Alumno:" + unAlu);
        }
    }
}
```

```
public class Alumno {
    private int leg;
    private String ape, nom;
    . . .
    public void setApe(String a){
        ape=a;
    }
    public int getLeg(){
        return leg;
    }
    public String toString(){
        return "Alumno:" + leg + "-" +
            ape + ", " + nom;
    }
}
```



Alumno: 3321- PEREZ, Sol
Alumno: 3421- RUSCITI, Maria
Alumno: 7892- GOMEZ, Juana

Colecciones

Las clases Arrays y Collections

Las clases `java.util.Collections` y `java.util.Arrays` son clases utilitarias que consisten exclusivamente de métodos estáticos que trabajan con colecciones. Estas clases contienen algoritmos polimórficos de búsqueda binaria, de ordenamiento, permutaciones aleatorias, duplicado de datos, etc.

```
package pruebas;
import java.util.Arrays;

class Prueba {
    public static void main(String[] args) {
        char[] letras = { 'd', 'e', 'j', 'a', 'v', 'a', 'c', 'l', 'i', 'c', 'k', 't', 'e', 'd' };
        System.out.println("Letras desordenadas: " + new String(letras));

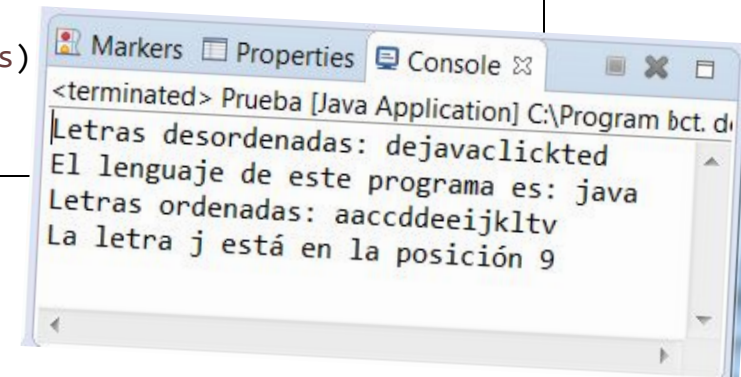
        char[] lenguaje = Arrays.copyOfRange(letras, 2, 6);
        System.out.println("El lenguaje de este programa es: " + new String(lenguaje));

        Arrays.sort(letras);
        System.out.println("Letras ordenadas: " + new String(letras));

        char letra = 'j';
        int pos = Arrays.binarySearch(letras, letra);
        System.out.println("La letra "+letra+" está en la posición "+ pos);
    }
}
```

copyRange es útil para obtener arreglos a partir de un rango de elementos de otro arreglo

binarySearch permite hacer una búsqueda dicotómica en un arreglo ordenado



La clase **Collections**, provee funcionalidades equivalentes a la clase **Arrays** pero trabaja colecciones en lugar de arreglos.