Objetivo: mostrar algunas de las colecciones más básicas de la jerarquía de clases que ofrece Java, y algunos métodos, para manipular objetos de tipo colección. Se espera motivar al alumno para que acceda por sus propios medios a la documentación de la API de java, en la que se encuentra en detalle la jerarquía completa, contenida en el paquete java.util.

Guardar Objetos - Contenedores

Java tiene varias formas de "guardar" o contener objetos (mejor dicho, referencias a objetos). Los contenedores proporcionan formas sofisticadas de guardar e incluso manipular objetos. El tipo de contenedor embebido en el compilador es el array. Por otra parte, la biblioteca de utilidades de Java (java.util) tiene un conjunto completo de clases contenedoras, del cual las mas usadas son: List, Set y Map, que complementan al array y sus desventajas.

1. Estructura Estática: array

Un array es una secuencia de elementos que pueden contener: ó datos primitivos u objetos (en realidad, referencias a objetos), empaquetados juntos bajo un único identificador (o nombre). Todos los elementos del array deben ser del mismo tipo.
Hay dos aspectos que distinguen a los arrays de otros tipos de contenedores: la eficiencia y el tipo. El array es la forma más eficiente que proporciona Java para almacenar y acceder al azar a una secuencia de objetos. Es una secuencia lineal simple, que hace rápido el acceso a los elementos, pero esa velocidad se paga: cuando se crea un objeto array, su tamaño es fijo y no puede variarse durante la vida de ese objeto array. También, el tipo de datos de los elementos es homogéneo.
Los arrays de objetos y los arrays de tipos primitivos son casi idénticos en uso. La única diferencia es que los arrays de objetos almacenan referencias, mientras que los arrays de primitivas guardan los valores primitivos directamente.
El identificador de array es, de hecho, una referencia a un objeto verdadero. Éste objeto mantiene las referencias a los otros objetos contenidos en él. El único atributo al que se puede acceder es el miembro length (de sólo lectura) que dice cuántos elementos pueden almacenarse en ese objeto array. El operador [ ] (corchetes) es el otro acceso que se tiene al objeto array, que permite su declaración.
El ejemplo siguiente muestra las distintas formas de inicializar un array, y cómo se pueden asignar las referencias a distintos objetos array.

```
public class Numero {
    private static long contador;
    private final long id = contador++;
    public String toString() { return "Entero "+id; }
}

public class PruebaArray {
     public static void main (String[] args) {
        // Arrays de objetos:
        Numero[] a; // Referencia Null
        Numero[] b = new Numero[5]; // Referencias Null
        Numero[] c = new Numero[4];
        for (int i = 0; i < c.length; i++){
            c[i] = new Numero();
        }
        // Inicialización de agregados:
        Numero [] d = {new Numero(), new Numero(), new Numero()};

        // Inicialización dinámica de agregados:
        a = new Numero[] {new Numero(), new Numero()};
        System.out.println("a.length = " + a.length);
        System.out.println("b.length = " + b.length) ;
        // Las referencias internas del array se inicializan automáticamente a null
        for (int i = 0; i < b. length; i++){
            System.out.println ("b[" + i + "] = " + b[i]);
        }
        System.out.println ("c.length = " + c.length) ;
        System.out.println ("d.length = " + d.length) ;
        a = d;
        System.out.println ("a.length = " + a.length) ;
        // Arrays de datos primitivos:
        int[] e; // Referencia null
        int[] f = new int[5];
        int[] g = new int[4];
        for(int i = 0; i < g.length; i++){
            g[i] = i*i;
        }
        int[] h = { 11, 47, 93 } ;

        // La siguiente línea presenta error de compilación, debido a que la variable e no está inicializada
```

```
        System.out.println("e.length=" + e.length);
        // Los datos primitivos de dentro del array se inicializan automáticamente a cero
        for (int i = 0; i < f .length; i++){
            System.out.println ("f[" + i + "] = " + f[i]);
        }
        System.out.println ("g.length = " + g.length) ;
        System.out.println ("h.length = " + h.length) ;
        e = h;
        System.out.println ("e.length = " + e.length) ;
        e = new int[] { 1, 2 };
        System.out.println("e.length = " + e.length);
    }
}
```

Nota: No se puede saber directamente cuántos elementos hay contenidos en un array, puesto que length dice sólo cuántos elementos puede contener el array. Sin embargo, cuando se crea un objeto array sus referencias se inicializan automáticamente a *null*, por lo que se puede ver si una posición concreta del array tiene un objeto, comprobando si es o no es null. De forma análoga, un array de tipos primitivos se inicializa automáticamente a *cero* en el caso de datos numéricos, *(char)0* (representación de *cero* como caracter) en el caso de caracteres, y *false* si se trata de valores lógicos.

Resumen de manipulación de objetos array:

- Se puede declarar y definir un contenedor en una misma línea
  ```
  tipo_de_dato[] var = new tipo_de_dato[4];
  ```
- Asignación de valores a los elementos de un contenedor
  ```
  var [ índice ] = valor;
  ```
- Java verifica que el índice no sea mayor o igual que la dimensión del array (dado que comienza en 0), ni negativo.
- Se pueden declarar, definir e inicializar en una misma línea, del siguiente modo
  ```
  int[] numeros ={2, -4, 15, -25};
  String[] nombres ={"Juan", "José", "Miguel", "Antonio"};
  ```
- Para saber el número de elementos del contenedor, se utiliza *length*, que proporciona la dimensión del array.
  ```
  int canElemen = var.length;
  ```

En el momento de crear un array, se debe especificar el tipo de los valores que almacenará. Los arrays permiten guardar elementos de tipo primitivo o referencias a objetos.

- Hay arrays de enteros, arrays reales, arrays de strings, etc.

```
int[] numeros = new int[4];

for(int i=0; i<numeros.length; i++){
        numeros[i] = i*i+3;
 }
```

- También es posible crear un array de referencias a objetos de una clase creada por el usuario. Por ejemplo, para crear un array de tres objetos de la clase *Punto* se escribe

```
Punto[] puntos = new Punto[3];

puntos [0]= new Punto(10, 35);
puntos [1]= new Punto(30, 40);
puntos [2]= new Punto(50, 80);
```

- Para imprimir cada punto según el diseño del método mostrar() definido en la clase Punto:

```
public class ArrayDePuntos{
   public static void main(String [] args){
        Punto [] puntos = new Punto[3];
        puntos [0]= new Punto(10, 35);
        puntos [1]= new Punto(30, 40);
        puntos [2]= new Punto(50, 80);
        for(int i=0; i<puntos.length; i++){
             puntos[i].mostrar();
        }
    }
 }
```

La sentencia **for** es la estructura de bucle más frecuentemente utilizada con los objetos array. Para una versión más corta ver foreach mas adelante.

2. Estructuras dinámicas: colecciones

Los arrays en Java son muy eficientes cuando se trata de almacenar elementos cuya cantidad máxima se conoce de antemano. Sin embargo, algunas veces no se sabe cuantos elementos se guardarán. Para estas ocasiones, el paquete **java.util** ofrece, mediante el *Collection Framework*, un conjunto de interfaces y clases bien diseñados para manipular contenedores dinámicos. Contenedores que denominamos colecciones: Set, List, Map -conjunto, lista y mapa- son los mas comunes. Presentan distintas características, pero todos permiten aumentar su capacidad en forma dinámica, durante la ejecución del programa salvando así la limitación de los arrays.

El *Collection Framework* adopta el concepto de "contener objetos" y lo divide en dos conceptos distintos: Colecciones y Mapas

1. Colección: una secuencia de elementos individuales con una o mas reglas que se aplican a ellos. Una lista (List) debe contener a sus elementos en el orden en que fueron agregados y un conjunto (Set) no puede tener elementos duplicados.
2. Mapa: un grupo de pares Clave.Valor (Key.Value) que permiten buscar un valor usando una clave. Es también llamado diccionario porque se busca un valor usando una clave como se busca una definición usando una palabra.

Todos estos contenedores contienen elementos de tipo **Object.** No permiten almacenar tipos primitivos. Para guardar tipos primitivos habrá que usar clases especiales, conocidas como wrapper (envoltura): Byte para byte, Short para short, Integer para int, Long para long, Boolean para boolean, Float para float, Double para double y Character para char.

Ejemplo:

```
ArrayList numeros = new ArrayList();
Integer miEntero = new Integer(12);   // crea una referencia a un entero
miLista.add(miEntero);               // añade a la colección una referencia a un int
 miLista.add(new Double(40.00));     // añade a la colección una referencia a un double
 miLista.add(new Long(55));          // añade a la colección una referencia a un long
```

Los contenedores de java son considerados polimórficos, porque permiten almacenar objetos de diferentes tipos: unaCasa, unAlumno, unaPera, etc.
Este tipo de polimorfismo se denomina **polimorfismo de subtipo**, ya que se basa en la jerarquía de tipos. Implica que se deberán moldear explícitamente los elementos durante la recuperación, a través de una operación de cast.

Por ejemplo:

```
ArrayList puntos = new ArrayList();     // crea una colección de puntos
 Punto miPunto = new Punto(10, 20);// crea un objeto de la clase Punto
 puntos.add (miPunto);              // agrega el punto a la colección
 puntos.add (new Punto(10, 10));    // agrega el punto a la colección
 . . .
 Punto otroPunto = puntos.get(0);  // recupera → el compilador lo rechazará
```

La sentencia que añade **miPunto** es correcta, pues el método add espera un objeto, y Punto es una subclase de **Object** (en java, todas las clases descienden de Object). La siguiente sentencia crea el un nuevo objeto de tipo **Punto antes de agregarlo al contenedor.** La sentencia que recupera **otroPunto** es errónea; ya que el tipo devuelto por la sentencia puntos.get(0) devuelve un **Object**, y no se puede asignar un **Object** a una referencia del tipo Punto, ya que Punto es subtipo de Object y no viceversa. Por tanto, es necesario realizar una operación de **downcast**, para lo que hay que escribir el siguiente código:

```
Punto otroPunto = (Punto)puntos.get(0);
```

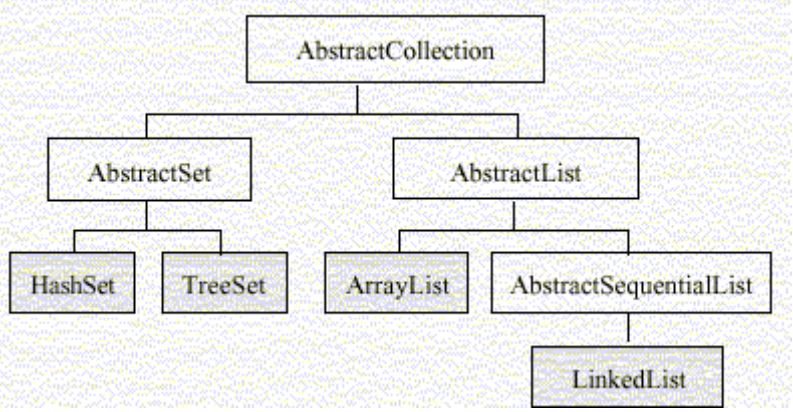
El efecto de la operación de **downcast** es realizar una **verificación en tiempo de ejecución**. En este caso, si el elemento alojado en la posición 0 de la colección es efectivamente un punto, la ejecución continuará normalmente. En caso de que falle, porque el tipo devuelto no es el correcto, se lanzará una excepción ClassCastException y no se realizará la asignación.
Es importante entender que la operación de cast **no** realiza una mutación del objeto devuelto por el método, es decir, no lo convierte. Los objetos llevan su tipo en tiempo de ejecución, y si un objeto se creó con un constructor de la clase Punto, siempre tendrá ese tipo.
El polimorfismo de subtipo ofrece cierta flexibilidad, dado que permite crear contenedores heterogéneos que contengan diferentes tipos de elementos.
Esto es común entre las aplicaciones como las tiendas online. Un cliente añade una mercancía a su carro de compra, y detrás de la escena, los ítems son almacenados y eliminados automáticamente.
Para esta clase de grupos de datos crecientes y menguantes, se pueden usar, entre otras, las clases ArrayList y HashMap del paquete java.util.

2.1. ArrayList

ArrayList es un contenedor que guarda sus elementos como una lista. Existe una secuencia en los elementos del ArrayList. Es muy eficiente al acceder a un elemento cualquiera pero es poco eficiente para insertar o remover elementos en medio de la lista. Por contraposición, el contenedor LinkedList, es poco eficiente al acceder a un elemento cualquiera y muy eficiente para insertar o remover elementos en medio de la lista.
ArrayList es la clase mas usada del *Collection Framework*, y los métodos más utilizados son add() para agregar elementos, get() para recuperarlos e iterator() o for() para recorrerlo.

Class ArrayList

```
java.lang.Object
  └java.util.AbstractCollection
      └java.util.AbstractList
          └java.util.ArrayList
```



Constructor Summary	
`ArrayList()`	Constructs an empty list with an initial capacity of ten.
`ArrayList(int initialCapacity)`	Constructs an empty list with the specified initial capacity.

Algunos de los más utilizados de esta clase son:

Method Summary	
void	`add(int index, Object element)` Inserts the specified element at the specified position in this list.
boolean	`add(Object o)` Appends the specified element to the end of this list.
void	`clear()` Removes all of the elements from this list.
boolean	`contains(Object elem)` Returns `true` if this list contains the specified element.
Object	`get(int index)` Returns the element at the specified position in this list.
int	`indexOf(Object elem)` Searches for the first occurence of the given argument, testing for equality using the `equals` method.
boolean	`isEmpty()` Tests if this list has no elements.
int	`lastIndexOf(Object elem)` Returns the index of the last occurrence of the specified object in this list.
boolean	`remove(Object elem)` Removes first occurence of the given argument .
protected void	`removeRange(int fromIndex, int toIndex)` Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
Object	`set(int index, Object element)` Replaces the element at the specified position in this list with the specified element
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
int	`size()` Returns the number of elements in this list.

Nota: ver todos los métodos disponibles de esta clase en http://java.sun.com/javase/6/docs/api/index.html

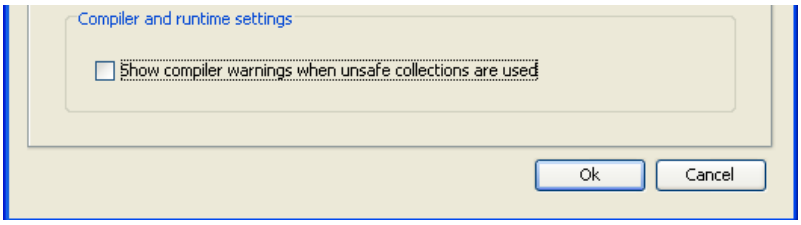
Ejemplo: Se desea manipular una agenda de contactos

```
public class Contacto{
    private String nombre;
    private String mail;

    public Contacto(String p_nombre, String p_mail){
       this.setNombre(p_nombre);
       this.setMail(p_mail);
    }

    public void setNombre(String p_nombre){
       this.nombre = p_nombre;
    }
     public void setMail(String p_mail){
       this.mail = p_mail;
    }
     public String getNombre(){
       return this.nombre;
    }
     public String getMail(){
       return this.mail;
    }

    public String mostrarContacto(){
        return this.getNombre() + " - " + this.getMail();
    }
}

import java.util.*;

public class Agenda{
    private ArrayList contactos;

    public Agenda(Contacto p_contacto) {
         this.setContactos(new ArrayList());
         this.agregarContacto(p_contacto);
        // la agenda se inicia con 1 contacto
    }

    public ArrayList getContactos(){
         return this.contactos;
    }

    public void setContactos(ArrayList p_contactos) {
         this.contactos = p_contactos;
    }

    public boolean agregarContacto(Contacto p_contacto){
         return this.getContactos().add(p_contacto);
    }

    public boolean quitarContacto(Contacto p_contacto){
         return this.getContactos().remove(p_contacto);
    }

    public int cuantosContactos(){
         return this.getContactos().size();
    }

    public void mostrarAgenda(){
       // para acceder a un elemento se usa el método get(i), indicando la posición de dicho elemento
      // se debe realizar un cast para recuperar un elemento (moldearlo a un tipo)
         for (int i=0;i<this.getContactos().size();i++){
           System.out.println(((Contacto)this.getContactos().get(i)).mostrarContacto());
    }
}
```

```
import java.util.*;
public class UsarAgenda{

        public static void main(String[] args) {
                // crea un contacto
                Contacto juan = new Contacto("Juan Perez", "jp_98@hotmail.com");

                // crea una agenda
                Agenda miAgenda = new Agenda(juan);

                // crea otro contacto
                Contacto maria = new Contacto("Maria Lopez", "ml_85@gmail.com");

                //agrega un contacto a la agenda
                miAgenda.agregarContacto(maria);
            System.out.println("\nLa agenda tiene " + miAgenda.cuantosContactos() + "
contactos");
                miAgenda.mostrarAgenda();

System.out.println("\nIndice del elemento Maria " + " --> " +
miAgenda.getContactos().indexOf(maria));

                System.out.println("\nElimina Maria y muestra nuevamente");
                miAgenda.quitarContacto(maria);
                miAgenda.mostrarAgenda();
        }
}
```

Nota: desde la versión 1.5 del JDK, al compilar, señala un error de tipo warning, debido a que no se está usando genéricos, la nueva característica de java, que se expondrá más adelante. Para evitar este aviso, se puede proceder de dos maneras:

- En BlueJ, en Opciones\Preferencias\Miscelaneas quitar el tilde de **Show compiler …**



- En general, usando una sentencia especial de java, también se evita este mensaje. Se coloca al inicio de la clase.

```
@SuppressWarnings("unchecked")
```

Resultado de la ejecución

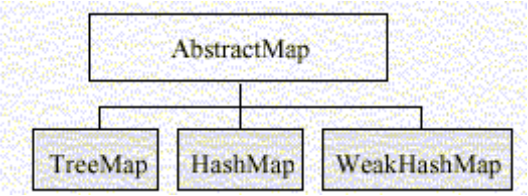
```
La agenda tiene 2 contactos
Juan Perez - jp_98@hotmail.com
Maria Lopez - ml_85@gmail.com

Indice del elemento Maria --> 1

Elimina Maria y muestra nuevamente
Juan Perez - jp_98@hotmail.com
```

2.2. HashMap

HashMap, otra de las clases del **Collection Framework**, es apropiada para almacenar y manipular **pares (clave.valor)** de datos sin orden. Cada objeto tiene algún atributo que permite identificarlo, por ejemplo un "idCliente" o "nroDocumento". Entonces se asocia cada objeto insertado en la colección (valor) a otro objeto identificador (clave), el cual es usado para recuperar el objeto insertado. Esta clase por lo tanto es utilizada cuando es necesario recuperar puntualmente algún elemento de la colección, sin necesidad de recorrerla completa. Esto ocurre frecuentemente en todas las aplicaciones.
La manera más eficiente de iterar a lo largo de los elementos (pares clave.valor) contenidos en un mapa es usando el método entrySet(). Esté método devuelve un conjunto (Set) de objetos Map.Entry. Este conjunto podrá iterarse y tratarse como cualquier otro contenedor de tipo Set. Un objeto Map.Entry contiene un par clave.valor asociado a un elemento del mapa, más adelante se indican los métodos más comúnmente utilizados con Map.Entry.

Class HashMap

java.lang.Object
  └ java.util.AbstractMap
    └ **java.util.HashMap**



Constructor Summary
HashMap() Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
HashMap(int initialCapacity) Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

Method Summary	
void	**clear**() Removes all mappings from this map.
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a set view of the mappings contained in this map.
Object	**get**(Object key) Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set	**keySet**() Returns a set view of the keys contained in this map.
Object	**put**(Object key, Object value) Associates the specified value with the specified key in this map.
Object	**remove**(Object key) Removes the mapping for this key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.
Collection	**values**() Returns a collection view of the values contained in this map.

Nota: todos los métodos disponibles de esta clase en http://java.sun.com/javase/6/docs/api/index.html

2.2.1. Map.Entry

Method Summary	
boolean	**clear**() Removes all mappings from this map.
K	**getKey**() Returns the key corresponding to this entry.
V	**getValue**() Returns the value corresponding to this entry
Int	**hashCode**() Returns the hash code value for this map entry
V	**setValue**(V value) Replaces the value corresponding to this entry with the specified value (optional operation)

Nota: todos los métodos disponibles de esta clase en http://java.sun.com/javase/6/docs/api/index.html

Ejemplo: Una empresa mantiene una colección de pares (Dni, Cliente).

```
/**
 * Clase Empresa mantiene una colección de clientes
 * @author Catedra POO
 * @version 2011-01
 */
 import java.util.*;
 public class Empresa{
    HashMap clientes;
    public Empresa(){
       this.setClientes(new HashMap());
    }
    // Retorna la colección de clientes de la empresa
    public HashMap getClientes() {
           return this.clientes;
    }
    // Asigna una colección de clientes a la empresa
    public void setClientes(HashMap p_clientes){
             this.clientes = p_clientes;
    }
    // Agrega un cliente a la coleccion de la empresa
    public void agregarCliente(Cliente p_cliente){
             this.getClientes().put(new Integer(p_cliente.getNroDni()), p_cliente);
    //                               ^ -> wrapper para insertar un int como un objeto
    }
    // Quita un cliente según DNI de la coleccion de la empresa. Retorna el objeto eliminado
    public Cliente quitarCliente(int p_nroDni){
             return (Cliente)this.getClientes().remove(new Integer(p_nroDni));
    }
    // Quita un cliente de la coleccion de la empresa. No retorna nada
    public void quitarCliente(Cliente p_cliente){
             this.getClientes().remove(new Integer(p_cliente.getNroDni()));
    }
    // Retorna el cliente que corresponde al DNI pasado como parámetro
    public Cliente buscarCliente(int p_nroDni){
        return (Cliente)this.getClientes().get(new Integer(p_nroDni));
    }
    /**
     * Verifica si el cliente es un cliente de esta empresa – Busqueda por clave
     * @return verdadero o falso
     */
    public boolean esCliente(int p_nroDni){
       return this.getClientes().containsKey(new Integer(p_nroDni));
     // Atención !! a partir de Java 5 se puede colocar el primitivo (int). El compilador en forma automática lo convierte a Integer
     //  -->  return this.getClientes().containsKey(p_nroDni);
    }

  // Metodo sobrecargado: busca por objeto cliente (valor), y no por DNI
   public boolean esCliente(Cliente p_cliente){
        return this.getClientes().containsValue(p_cliente);
   }
}

import java.util.Scanner;

public class PruebaEmpresa {
  public static void main(String[] args) {
     char sigue = 'S';
     int a_nroDni=0;
     Empresa emp1 = new Empresa();
     Scanner teclado = new Scanner(System.in);
     System.out.println("1 - Ingresa datos, instancia objetos de la clase Cliente y los adiciona a la colección");
     while(sigue == 'S') {
            System.out.print("    1.1 - Ingrese DNI:      ");
            a_nroDni = teclado.nextInt();
            System.out.print("    1.2 - Ingrese nombre:   ");
            String a_nombre = teclado.next();
            System.out.print("    1.3 - Ingrese apellido: ");
            String a_apellido = teclado.next();
```

```
            System.out.print("    1.4 - Ingrese saldo:   ");
            double a_saldo = teclado.nextDouble();

            Cliente cli1 = new Cliente(a_nroDni, a_nombre, a_apellido, a_saldo);
            emp1.agregarCliente(cli1);

            System.out.print("Mas clientes ? S/N :");
            sigue=(teclado.next()).charAt(0);
      }//cierra while

      System.out.println("\n 2 - Ingrese DNI del cliente buscado");
      a_nroDni = teclado.nextInt();

      System.out.println("\n 3 - Recupera el cliente y lo muestra");
      Cliente miCli = emp1.buscarCliente(a_nroDni);
      miCli.mostrar();

      System.out.println("\n 4 - Elimina el mismo cliente");
      emp1.quitarCliente(miCli);

      System.out.println("\n 5 - Verifica si existe el cliente (por DNI) --> " +
                         unaEmpresa.esCliente(a_nroDni)
      );

      System.out.println("\n 6 - Verifica si existe el cliente (por objeto) --> "   +
                         unaEmpresa.esCliente(miCli)
      );
   }
}
```

Resultado de la ejecución

1 - Ingresa datos, instancia objetos de la clase Cliente y los adiciona a la colección
  1.1 - Ingrese DNI: 21434556
  1.2 - Ingrese nombre: Juan
  1.3 - Ingrese apellido: Perez
  1.4 - Ingrese saldo: 1000
Mas clientes ? S/N :S
  1.1 - Ingrese DNI: 45637277
  1.2 - Ingrese nombre: Maria
  1.3 - Ingrese apellido: Gonzalez
  1.41 - Ingrese saldo: 233
Mas clientes ? S/N :N

2 - Ingrese DNI del cliente buscado
21434556

3 - Recupera el cliente y lo muestra
Nombre y Apellido: Juan Perez (21434556)
Saldo: $1000.0

4 - Elimina el mismo cliente

5 - Verifica si existe el cliente (por DNI) --> false

6 - Verifica si existe el cliente (por objeto) --> false

3. Genéricos

La característica de los contenedores dinámicos es que permiten agregar objetos de cualquier tipo (Object). Esto puede considerarse una ventaja ó un inconveniente. Por ejemplo, consideremos un cajón de Manzanas, usando el tipo de contenedores ArrayList. Se agregarán Manzanas y Peras al contenedor. Luego se accederá a cada elemento del contenedor.

```
public class Manzana {
    private static long contador;
    private final long id = contador++;
    public long id() { return id; }
}

public class Pera {
      // no hace nada
}

import java.util.*;
public class ManzanasYPerasSinGenerics {
    @SuppressWarnings("unchecked") // sentencia especial de java

    public static void main(String[] args) {
        ArrayList cajon = new ArrayList ();

        for(int i = 0; i < 6; i++){
            cajon.add(new Manzana());
        }

        // Se pueden agregar Peras en lugar de Manzanas porque es un Object
        cajon.add(new Pera());

        for(int i = 0; i < cajon.size(); i++){
            System.out.println(
                    ( (Manzana) cajon.get(i) ).id()
            );
        }
    }
}
```

Las clases Pera y Manzana son distintas. Sólo tienen en común que ambas son subtipo de Object. Al usar ArrayList para el cajón, mediante **add()** no solo se pueden agregar objetos Manzana sino cualquier tipo de objeto, como por ejemplo una Pera, ***sin recibir ninguna advertencia ni en tiempo de compilación, ni en tiempo de ejecución.***
Al recuperar usando **get()**, se obtiene una referencia a un objeto de tipo Object que deberá ser "mutado" (cast) a Manzana antes de llamar al método id() de Manzana. Caso contrario, ***se recibe un mensaje de error en tiempo de compilación***. Por lo tanto, se debe encerrar toda esa expresión entre paréntesis para forzar el cast, y luego enviar el mensaje id().
Al ejecutar, cuando recupera un objeto Pera y quiere hacer el cast a Manzana, aparece ***un error en tiempo de ejecución,*** porque la clase subyacente del objeto no es Manzana.

java.lang.ClassCastException: Pera cannot be cast to Manzana
at ManzanasYPerasSinGenerics.main(ManzanasYPerasSinGenerics.java:14)

Este inconveniente se resuelve usando programación genérica. Este tipo de programación es compleja sin embargo, aplicarla para el uso en contenedores es más directo.
En el ejemplo, para el cajón de manzanas deberíamos haber usado ArrayList<Manzana> en lugar de solo ArrayList.
Los paréntesis angulares indican el tipo de dato que se podrá agregar al contenedor, así en tiempo de compilación, si queremos agregar algún objeto que no sea del tipo de éste, el compilador dará error.

```
public class ManzanasYPerasConGenerics {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        ArrayList<Manzana> cajon = new ArrayList<Manzana>();

        for(int i = 0; i < 6; i++){
            cajon.add(new Manzana());
        }

        // NO se pueden agregar Peras en lugar de Manzanas:
        cajon.add(new Pera());

        for(int i = 0; i < cajon.size(); i++)
            // no es necesario hacer cast
            System.out.println(cajon.get(i).id());
    }
} /* (Ejecutar para ver el resultado) *///:~
```

5 – Reutilizacion de clases predefinidas - Colecciones en Java

Programación Genérica (PG): es un estilo de programación en la cual los algoritmos son escritos en términos de tipos "a ser especificados después" (to-be-specified-later types), que son instanciados cuando se los necesita para tipos específicos provistos como parámetros.
Esta mejora permite escribir funciones comunes que difieren solamente en el conjunto de tipos sobre los cuales operan cuando se los usa, reduciendo la duplicación de código.
La **PG** se refiere a características de ciertos lenguajes de programación tipados estáticamente que permiten algún tipo de código para solucionar deficiencias por falta de dinamismo.

El primer lenguaje que implementó este tipo de programación fue ADA (1983). Otros son Eiffel, Java, C#.
C++: El código genérico se implementa mediante plantillas de clases.
C# y .NET: Fueron agregados como parte del .NET Framework 2.0 (2005). Similar a JAVA.
JAVA: soporta genéricos (o containers-of-type-T), subconjunto de la programación genérica que fue agregada en el 2004 como parte del J2SE 5.0. En Java los genéricos son **chequeados en tiempo de compilación**, verificando las clases. La información de tipo genérica es removida mediante un proceso llamado type erasure, y no disponible en tiempo de ejecución.

Definición General
Para trabajar con tipos genéricos se debe proceder de la siguiente manera:

- **Declaración**: ClaseColeccion <ClaseObjetos_a_ser_contenidos> nombreColección;
- **Definición**: nombreColeccion = new ClaseColeccion <ClaseObjetos_a_ser_contenidos> (parámetros del constructor)

Ejemplo 1: trabajando con la clase ArrayList. Colección de objetos de las clases Integer. En tiempo de compilación ya se sabe el tipo de objeto de los elementos de la colección.

```
import java.util.*;
public class ArrayListGenerico{
   public static void main(String[] args) {
       ArrayList <Integer> miLista = new ArrayList <Integer>(10);
       Integer miEntero = 0;

       System.out.println(" 1 - Genera automaticamente enteros y los adiciona a la colección");
       for(int i=0; i< 4; i++){
            miEntero = new Integer(i*3);
            miLista.add(miEntero);
       }

       System.out.println("\n 2 - Métodos varios *********");
       System.out.println("    2.1 - Cantidad de elementos: " + miLista.size());
       System.out.println("    2.2 - Indice del elemento " + miEntero + " --> "+
                                    miLista.indexOf(miEntero));

       // Para acceder a un elemento se usa el método get(i), indicando la posición de dicho elemento
       // No es necesario un casteo para recuperar un elemento (asignarlo a un tipo)
       Integer elEntero = miLista.get(3);
       System.out.println("    2.3 - Recuperado un entero en particular " +" --> "+ elEntero);

       System.out.println("\n 3 - Foreach permite recuperar todos los elementos de la lista");
       for(Integer x: miLista){
             System.out.println("Elemento " + x);
       }
   }
}
```

Resultado de la ejecución

```
1 - Genera automaticamente enteros y los adiciona a la colección

 2 - Métodos varios *********
    2.1 - Cantidad de elementos: 4
    2.2 - Indice del elemento 9 --> 3
    2.3 - Recuperado un entero en particular  --> 9

 3 - Foreach permite recuperar todos los elementos de la lista
Elemento 0
Elemento 3
Elemento 6
Elemento 9
```

Ejemplo 2: trabajando con la clase HashMap.

```
import java.util.*;
public class EmpresaGenerico{
    HashMap <Integer, Cliente> clientes;

    public EmpresaGenerico(){
        this.setClientes(new HashMap<Integer, Cliente>());
    }
    public HashMap <Integer, Cliente> getClientes() {
        return this.clientes;
    }

    // Asigna una colección de clientes a la empresa
    public void setClientes(HashMap<Integer, Cliente> p_clientes){
            this.clientes = p_clientes;
    }
    public void agregarCliente(Cliente p_cliente){
        this.getClientes().put(new Integer(p_cliente.getNroDni()),p_cliente);
    }
     public void quitarCliente(Cliente p_cliente){
            this.getClientes().remove(new Integer(p_cliente.getNroDni()));
    }
    public Cliente buscarCliente(int p_nroDni){
        return this.getClientes().get(new Integer(p_nroDni));
    }
    public void mostrarTodos(){
        for (Map.Entry <Integer, Cliente> e : clientes.entrySet()){
            System.out.println(e.getKey() + " " + e.getValue().getNombre());
        }
    }
}

import java.util.Scanner;
public class PruebaEmpresaGenerica {
    public static void main(String[] args) {
        char sigue = 'S';
        int a_nroDni=0;
        EmpresaGenerico emp1 = new EmpresaGenerico();
        Scanner teclado = new Scanner(System.in);
        System.out.println("1 - Ingresa datos, instancia objetos de la clase Cliente y los adiciona a la colección");
        while(sigue == 'S') {
            System.out.print("    1.1 - Ingrese DNI:      ");
            a_nroDni = teclado.nextInt();
            System.out.print("    1.2 - Ingrese nombre:   ");
            String a_nombre = teclado.next();
            System.out.print("    1.3 - Ingrese apellido: ");
            String a_apellido = teclado.next();
            System.out.print("    1.4 - Ingrese saldo:   ");
            double a_saldo = teclado.nextDouble();

            Cliente cli1 = new Cliente(a_nroDni, a_nombre, a_apellido, a_saldo);
            emp1.agregarCliente(cli1);

            System.out.print("Mas clientes ? S/N :");
            sigue=(teclado.next()).charAt(0);
        }//cierra while

        System.out.println("\n 2 - Ingrese DNI del cliente buscado");
        a_nroDni = teclado.nextInt();

        System.out.println("\n 3 - Recupera el cliente y lo muestra");
        Cliente miCli = emp1.buscarCliente(a_nroDni);
        miCli.mostrar();

        System.out.println("\n 4 - Foreach recupera todos los clientes y los muestra");
        emp1.mostrarTodos();
    }
}
```

Resultado de la ejecución

```
1 - Ingresa datos, instancia objetos de la clase Cliente y los adiciona a la colección
    1.1 - Ingrese DNI:      34565876
    1.2 - Ingrese nombre:   Pipo
    1.3 - Ingrese apellido: Pescador
    1.41 - Ingrese saldo:   233
Mas clientes ? S/N :S
    1.1 - Ingrese DNI:      23453765
    1.2 - Ingrese nombre:   Juana
    1.3 - Ingrese apellido: Gomez
    1.4 - Ingrese saldo:   243
Mas clientes ? S/N :N

 2 - Ingrese DNI del cliente buscado
23453765

 3 - Recupera el cliente y lo muestra
Nombre y Apellido: Juana Gomez (23453765)
 Saldo: $243.0

 4 - Foreach recupera todos los clientes y los muestra
23453765 Pipo
34565876 Juana
```

4. Recorrer colecciones

En ocasiones se precisa recorrer una colección para realizar alguna operación con cada elemento del contenedor. Existen dos formas de hacerlo: foreach o Iterator. La forma foreach es mas breve y directa pero, si se necesita eliminar al menos un elemento de la lista, la programación se vuelve muy compleja. En ese caso es mejor usar el concepto Iterator.

4.1. Recorrer la colección para cada elemento (foreach)

Es la forma más breve de la sentencia for para uso con contenedores. No se necesita crear un índice para contar a través de la secuencia de elementos, el foreach trae el elemento i-ésimo del contenedor en cada iteración.
Ejemplo: se desea guardar una lista de 10 números entre 0 y 9 generados al azar, para luego mostrarlos por consola:

```
import java.util.*;

public class pruebaForEach {
    public static void main(String [] args){
        //la semilla aleatoria
        Random rand = new Random();

        //Crea la lista
        ArrayList<Integer> numeros = new ArrayList<Integer>();

        //Carga la lista. For tradicional para asegurar los 10 elementos
        for (int i=0; i<10; i++){
            numeros.add ( new Integer(rand.nextInt(9)));
        }
        //Muestra la lista. Foreach
        int ord=0;
        for(Integer e:numeros){
            System.out.println(""+ord++ +"- Numero: "+e);
        }
    }
}
```

4.2. Recorrer la colección mediante Iterator

Iterator es un objeto cuyo trabajo es moverse a través de un contenedor y traer el siguiente elemento en ese contenedor, sin que al programador le preocupe cual es el tipo de contenedor. Con un objeto Iterator no se puede hacer mucho más que:

1. Pedirle a un contenedor que nos dé un objeto Iterator con el método **iterator()**. Dicho objeto estará listo para traer el primer elemento en el contenedor.
2. Traer el siguiente elemento en la secuencia del contenedor con **next()**.
3. Ver si hay mas elementos en el contenedor con **hasNext()**.

Iterator tiene un subtipo ListIterator que puede ser producido únicamente por contenedores del tipo List. Las ventajas que da este subtipo son: recorrer la lista en ambas direcciones, reemplazar el elemento que se esta visitando y agregar nuevos elementos. Para eso ListIterator agrega los siguientes métodos:

1. Perdirle al contenedor que nos dé su ListIterator() con **listIterator()** o con **listIterator(n)** si queremos que el primero elemento de esta secuencia sea el n-ésimo.
2. Pedir el elemento previo con **previous()**.
3. Saber si existe un elemento previo con **hasPrevious()**.
4. Reemplazar el elemento con **set(E)**.
5. Pedir el siguiente número de índice con **nextIndex()**.
6. Pedir el índice previo con **previousIndex()**.
7. Agregar un nuevo elemento con **add(E)**.

En el ejemplo anterior quedaría:

```
import java.util.*;

public class pruebaIterator {
    public static void main(String [] args){
        //la semilla aleatoria
        Random rand = new Random();

        //Crea la lista
        ArrayList<Integer> numeros = new ArrayList<Integer>();

        //Carga la lista. For tradicional para asegurar los 10 elementos
        for (int i=0; i<10; i++){
            numeros.add ( new Integer(rand.nextInt(9)));
        }
        //Muestra la lista. Iterator
        int ord=0;
        Iterator iteL = numeros.iterator(); //Pide al contenedor que le pase su iterador
        while (iteL.hasNext()){
            System.out.println("" + ord++ + "- Numero: " + iteL.next());
        }
        //Muestra la lista desde el quinto elemento hacia el primero. ListIterator
        ListIterator lIte = numeros.listIterator(5); //Pide su listIterador
        while (lIte.hasPrevious()){
            System.out.println(""+ lIte.previousIndex()+ " <- Numero: "+ lIte.previous());
        }
    }
}
```

5. Utilidades generales

En java.util encontramos clases que fueron diseñadas para uso auxiliar complementario de los contenedores. Estas son las clases **Collections** y **Arrays**. Ambas clases permiten realizar operaciones masivas sobre una colección o un array respectivamente. Sus métodos son de clase (estáticos), es decir que no hará falta instanciarlas para hacer uso de sus servicios.

5.1. La clase Collections

Esta clase tiene métodos de clase (estáticos) que operan exclusivamente con, o devuelven colecciones. Todos los métodos lanzan la excepción NullPointerException si los objetos pasados son null.

Method Summary	
static <T> boolean	addAll(Collection<? super T> c, T... elements) Adds all of the specified elements to the specified collection.
static <T> int	binarySearch(List<? extends T> list, T key, Comparator<? super T> c) Searches the specified list for the specified object using the binary search algorithm.
static <T> void	copy(List<? super T> dest, List<? extends T> src) Copies all of the elements from one list into another.
static <T> List<T>	emptyList() Returns the empty list (immutable).
static	emptyMap()

<K,V> Map<K,V>	Returns the empty map (immutable).
static <T> void	fill(List<? super T> list, T obj) Replaces all of the elements of the specified list with the specified element.
static <T> T	max(Collection<? extends T> coll, Comparator<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T> List<T>	nCopies(int n, T o) Returns an immutable list consisting of n copies of the specified object.
static <E> Set<E>	newSetFromMap(Map<E,Boolean> map) Returns a set backed by the specified map.
static <T> boolean	replaceAll(List<T> list, T oldVal, T newVal) Replaces all occurrences of one specified value in a list with another.
static void	reverse(List<?> list) Reverses the order of the elements in the specified list.
static <T> void	sort(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.

5.2. La clase Arrays

Tiene varios métodos de clase (estáticos) para manipular arrays (como ordenar y buscar). También tiene un "fábrica" que permite ver al array como una Lista. Estos son algunos de los métodos:

Method Summary	
static <T> List<T>	**asList**(T... a) Returns a fixed-size list backed by the specified array.
static int	**binarySearch**(Object[] a, Object key) Searches the specified array for the specified object using the binary search algorithm.
static boolean	**equals**(Object[] a, Object[] a2) Returns true if the two specified arrays of Objects are *equal* to one another
static void	**sort**(Object[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements.
static <T> void	**sort**(T[] a, int fromIndex, int toIndex, Comparator<? super T> c) Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
static String	toString(Object[] a) Returns a string representation of the contents of the specified array.

- asList: public static <T> List<T> asList(T... a)

Devuelve una lista de tamaño fijo basada en el array especificado (Los cambios realizados en la lista retornada son escritos en el array de origen). Este método actúa como un puente entre arrays y colecciones en combinación con Collections.toArray(). Este método puede usarse también para crear listas de tamaño fijo inicializadas:

```
List<String> chiflados= Arrays.asList("Larry", "Moe", "Curly");
```

Parametros:
a – el array por en la que se basa la lista
Retorna:
a una vista de lista del array especificado

- binarySearch: public static int binarySearch(Object[] a, Object key)
Recorre el array especificado buscando el objeto indicado usando el algoritmo de busqueda binario. El array deberá estar ordenado en forma ascendente o descendente de acuerdo al orden natural del sus elementos. (sort(Object[])).

Parametros:
a – el array a ser recorrido
key – el valor buscado
Retorna:
El indice del elemento que contiene el objeto buscado, si no lo encuentra un valor menor que cero.
Excepciones:
ClassCastException – si la clave buscada no es comparable con los elementos del array.

- equals: public static boolean equals(Object[] a, Object[] a2)
Devuelve true si los dos arrays especificados son iguales entre sí. Es decir si ambos contienen los mismos elementos en el mismo orden o si ambos son nulos.
Parametros:
a – un array
a2 – el otro array
Retorno:
true si los dos arrays son iguales

- sort: public static void sort(Object[] a, int fromIndex, int toIndex)
Ordena el rango indicado de los elementos del array especificado, en orden ascendente de acuerdo con el orden natural de los elementos. El rango incluye a fromIndex y excluye a toIndex. Todos los elementos a ser ordenados deben implementar la interface Comparable. Ademas, dichos elementos deben ser mutuamente comparables.

Parametros:
a – el array a ser ordenado
fromIndex – el indice del primer elemento (incluido) a ser ordenado
toIndex - el indice del ultimo elemento (excluido) a ser ordenado
Excepciones:
IllegalArgumentException - si fromIndex > toIndex
ArrayIndexOutOfBoundsException - si fromIndex < 0 o toIndex > a.length
ClassCastException – si el array contiene elementos que no son mutuamente comparables (por ejemplo String e int).

- sort: public static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)
Ordena el rango indicado de los elementos del array especificado, en orden ascendente de acuerdo con objeto comparator especificado. El rango incluye a fromIndex y excluye a toIndex. Todos los elementos a ser ordenados deben ser mutuamente comparables por el objeto comparator especificado. (significa que c.compare(e1, e2) no debe lanzar una ClassCastException para ningún elemento e1 y e2 del rango)

Parametros:
a – el array a ser ordenado
fromIndex – el indice del primer elemento (incluido) a ser ordenado
toIndex - el indice del ultimo elemento (excluido) a ser ordenado
c – el objeto comparator que determina el criterio de orden el array un valor null indica que debe usarse el orden natural.

Excepciones:
IllegalArgumentException - si fromIndex > toIndex
ArrayIndexOutOfBoundsException - si fromIndex < 0 o toIndex > a.length
ClassCastException – si el array contiene elementos que no son mutuamente comparables (por ejemplo String e int).

- toString: public static String toString(Object[] a)
Devuelve una representación en texto del contenido del array especificado. Si el array contiene otros array como elementos, son convertidos a strings mediante el método Object.toString() lo que muestra sus identidades y no su contenido.

Parametros:
a – el array del que se devolvera la cadena de texto
Retorno:
Una cadena de texto que representa al array

6. Comparar y Ordenar contenedores

Dos contenedores son iguales si tienen los mismos elementos y en el mismo orden. O si ambos son null. Para saberlo se lo recorre y se compara cada elemento usando equals(). Ejemplo: elementoColeccion1.equals(elementoColeccion2).
Java tiene dos formas de comparar elementos:

6.1. Orden natural

Se implementa en una clase con la interfase java.lang.Comparable. Esta es una simple interfase con un solo método compareTo(). Este método toma otro objeto del mismo tipo como argumento devolviendo un valor negativo si el objeto es menor que el argumento, 0 si son iguales y un valor positivo si el objeto es mayor que el argumento.

Por ejemplo, supongamos una clase Nombre, a la cual queremos hacerla comparable y que su "orden natural" esté dado por Apellido+Nombre:

```
public class Nombre implements Comparable<Nombre> {
    private final String nombreDePila, apellido;

    public Nombre(String nombreDePila, String apellido) {
        if (nombreDePila == null || apellido == null){
             throw new NullPointerException();
        }
        this.nombreDePila = nombreDePila;
        this.apellido = apellido;
    }
    public String getNombreDePila() { return nombreDePila; }
    public String getApellido()  { return apellido;  }
    public boolean equals(Object o) {
       if (!(o instanceof Nombre)){
           return false;
        }
        Nombre n = (Nombre) o;
        return n.getNombreDePila().equals(nombreDePila) &&
               n.getApellido.equals(apellido);
    }
    public int hashCode() {
        return 31*nombreDePila.hashCode() + apellido.hashCode();
    }
    public String toString() {
          return nombreDePila + " " + apellido;
    }
    public int compareTo(Nombre n) {
        // Primero compara el apellido del objeto contra el apellido del argumento
        // si los apellidos son iguales, compara los nombres
        int lastCmp = apellido.compareTo(n.getApellido());
        return (lastCmp != 0 ? lastCmp :
                nombreDePila.compareTo(n.getNombreDePila()));
    }
}
```

6.2. Comparator

La segunda forma de ordenamiento es mediante un objeto que implementa la interfase Comparator. Esta interfase tiene dos métodos: equals() y compare(). Sin embargo, en contadas excepciones habrá que re-definir el método equals() ya que se usa el definido en la clase Object. Un objeto que implemente la interfase Comparator será un objeto que contenga el/los criterios de orden que se podrán utilizar para determinado tipo de objetos.

Supongamos que el orden natural de un objeto Empleado es su nombre, pero para un determinado listado se prefiere ordenado por DNI:

```
public class Empleado implements Comparable<Empleado> {
    private Nombre nombre;
    private int dni;
       ...
    public int compareTo(Empleado emp) {
        return (getNombre().compareTo(emp.getNombre()));
    }
}
```

La clase que arma el listado será:

```
import java.util.*;
public class ReportesEmpleados {
    //CRITERIO DE ORDENAMIENTO POR DNI
    static final Comparator<Empleado> ORDEN_DNI =
        new Comparator<Empleado>() {
            //Redefine el método compare del objeto Comparator
            public int compare(Empleado e1, Empleado e2) {
                               return e2.getDni().compareTo(e1.getDni());
            }
        };

    public static void reportePorDNI(Collection<Empleado> empleados) {
        //Crea una copia e de la lista de empleados
        List<Empleado>e = new ArrayList<Empleado>(empleados);
       //Ordena la copia según el criterio ORDEN_DNI
        Collections.sort(e, ORDEN_DNI);
        //Imprime, en la salida estándar, la copia ordenada
        System.out.println(e);
    }
}
```