Capítulo 7

Genéricos - Colecciones

Revista Digital - Hecha por estudiantes para estudiantes

Bienvenidos

Nos encontramos nuevamente en otra entrega de JavaWorld. En el capítulo anterior comenzamos a adentrarnos en la API de Java para Escritura/Lectura de archivos y serialización de clases. En este capítulo nos toca la API para Colecciones y clases genéricas.

Tanto el capítulo anterior como este requieren que aprendan como y para que funciona cada clase, como así también, cada uno de sus métodos, de manera que se recomienda realizar mucha ejercitación.

Recuerden “la práctica hace al maestro”.

¿Tenés lo que se necesita para ser un SCJP?

http://valorcreativo.blogspot.com

Métodos del objeto object

Método	Descripcion
boolean equals(Object)	Interpreta cuando dos objetos son equivalentes.
void finalize()	Método invocado por el Garbage Collector (Recolector de basura).
int hashCode()	Devuelve un valor de hash para poder ser utilizado en colecciones que utilizan hashing.
final void notify()	Despierta un hilo de ejecución que estuviera esperando a este objeto.
final void notifyAll()	Despierta todos los hilos de ejecución que estuvieran esperando a este objeto.
final void wait()	Hace que el hilo de ejecución pase a inactividad hasta que otro objeto llame a notify() o notifyAll() sobre el mismo.
String toString()	Genera una representación del objeto en un String.

finalize() fue tratado en el cap. 3.

notify(), notifyAll(), y wait() se verán en el capítulo 9.

El método toString()

El método toString() es el encargado de transformar un objeto en una cadena de texto.

Dado que el método está contenido en java.lang.Object, significa que todos los objetos poseen este método. Además, es posible sobrescribirlo, de manera de modificar su comportamiento.

Normalmente cuando mostramos un objeto, es [nombre de la clase]@[hash code de la clase].

```
public class Persona {
  private int edad;
  private String nombre;

  static public void main(String[] args) {
    System.out.println(new Persona("Venito", 32));
  }
  public Persona(String nombre, int edad) {
    this.edad = edad;
    this.nombre = nombre;
  }
}
```

```
Persona@3e25a5
```

Ahora si sobrescribimos el método toString de la siguiente manera, podemos obtener una representación más legible del objeto:

```
public String toString() {
      return "La persona llamada " + nombre + " tiene " + edad + " año(s).";
}
```

```
La persona llamada Venito tiene 32 año(s).
```

¿Querés ser un SCJP?

Sobrescribiendo el método equals()

El método equals() indica si dos objetos son iguales (contienen los mismos atributos y son del mismo tipo).
Ahora, cual es la diferencia entre == y equals().

- == Compara las referencias entre dos variables, e indica si las mismas apuntan al mismo objeto.
- equals() Compara dos objetos e indica si los mismos son equivalentes con respecto al valor de sus atributos.

Las clases String y todos los wrappers de los tipos primitivos tienen sobrescrito el método equals().

Veamos un poco de código como ejemplo:

```
public class Prueba {
  static public void main(String[] args) {
    Valor obj_1 = new Valor(17);
    Valor obj_2 = new Valor(17);
    Valor obj_3 = obj_2;

    System.out.println("es obj_1 == obj_2    ? " + (obj_1==obj_2));
    System.out.println("es obj_1 equals obj_2? " + (obj_1.equals(obj_2)));

    System.out.println("es obj_2 == obj_3    ? " + (obj_2==obj_3));
    System.out.println("es obj_2 equals obj_3? " + (obj_2.equals(obj_3)));
  }
}

class Valor {
  private int valor;
  public Valor(int valor) {
    this.valor = valor;
  }
  public int getValor() {
    return valor;
  }
  public boolean equals(Object obj) {
    boolean resultado = false;
    if (obj instanceof Valor) {
      if (valor == ((Valor)obj).getValor()) {
        resultado = true;
      }
    }
    return resultado;
  }
}
```

```
es obj_1 == obj_2    ? false
es obj_1 equals obj_2? true
es obj_2 == obj_3    ? true
es obj_2 equals obj_3? True
```

Cuando hacemos obj_1 == obj_2 devuelve false porque los objetos a los que apunta cada referencia son distintos.
En cambio el equals() fue sobrescrito para verificar el campo valor y verificar por una igualdad.

http://valorcreativo.blogspot.com

Recuerda que tanto toString(), equals(), y hashCode() son **public**. El intentar sobrescribirlo con otro modificador de acceso generaría un error de compilación.

Sobrescribiendo el método hashCode()

El hashCode es un identificador, el cual, no es único. Este se utiliza para aumentar la velocidad de búsqueda en grandes colecciones de datos, de manera que la búsqueda se resuma a todos aquellos que tengan el mismo hashing.
Para el examen solo es necesario saber el funcionamiento de este método y que sea una implementación válida, sin importar si es o no eficiente.
Un código hash dentro de una tabla señala solamente un grupo, luego, dentro de ese grupo, hay que buscar el objeto.

Dado que el hash solo marca el grupo contenedor, y luego hay que rastrear el objeto dentro de ese grupo, es necesario que se implemente el método equals().

Dado que el hash se utiliza para almacenar y para obtener, es una condición básica de que siempre devuelva el mismo código para el mismo objeto.
En términos de código hashing, lo más performante sería tener un algoritmo que genere un código hashing distribuido uniformemente para todos los objetos, aunque también es válido tener un hasing que siempre devuelva el mismo número (cumple con la regla de el mismo hashing para el mismo objeto).

Contratos a cumplir

Hashcode

- El mismo objeto debe devolver siempre el mismo hash.
- Si según equals() dos objetos son iguales, la llamada a hashcode para cada uno debe devolver el mismo valor (obj1.hashcode() == obj2.hashcode()).
- Si según equals() dos objetos son diferentes, la llamada a hashcode para cada uno puede llegar a devolver el mismo valor, pero no es obligatorio.

Equals

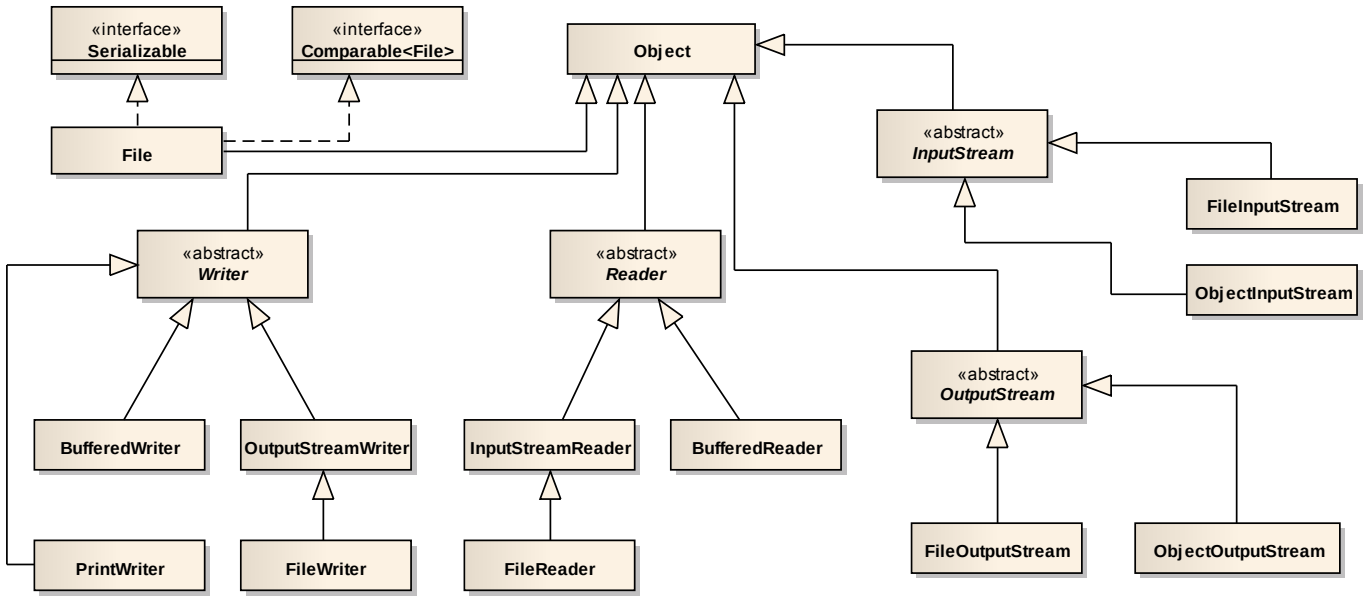
- Si a equals se le pasa el mismo objeto que el que invoca el método, el resultado debe ser siempre true (obj1.equals(obj1)).
- Si la llamada de equals entre dos objetos devuelve true, los operandos pueden ser invertidos(obj1.equals(obj2) == obj2.equals(obj1)).
- Si se compara cualquier objeto con null, este debe devolver false (obj1.equals(null) siempre debe ser false).

Colecciones

Es posible confundirse en el examen con las distintas palabras collection

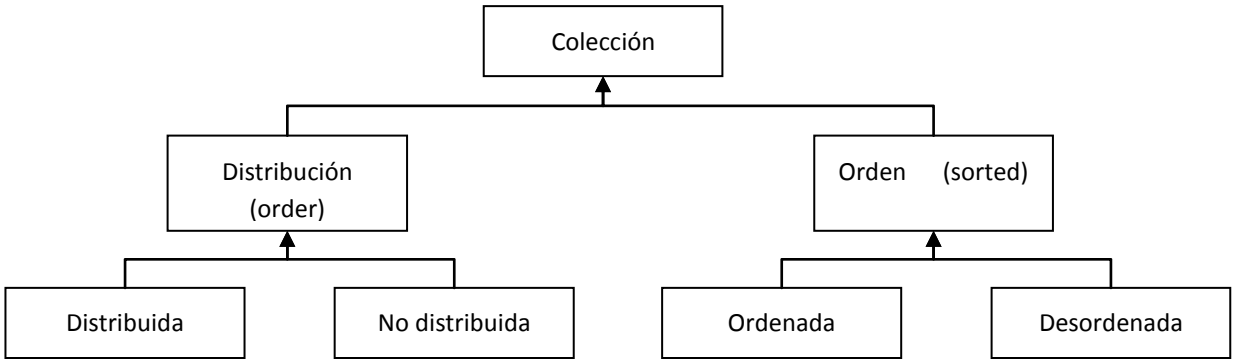
collection (todo en minúsculas)	Nombre para identificar cualquier estructura que contenga una colección de elementos, y permita iterar a través de ellos.
Collection (con C mayúscula)	Interface java.util.Collection.
Collections (con C mayúscula y s al final)	Clase java.util.Collections.

¿Querés ser un SCJP?



Características de cada clase

Una colección puede clasificarse de la siguiente manera:

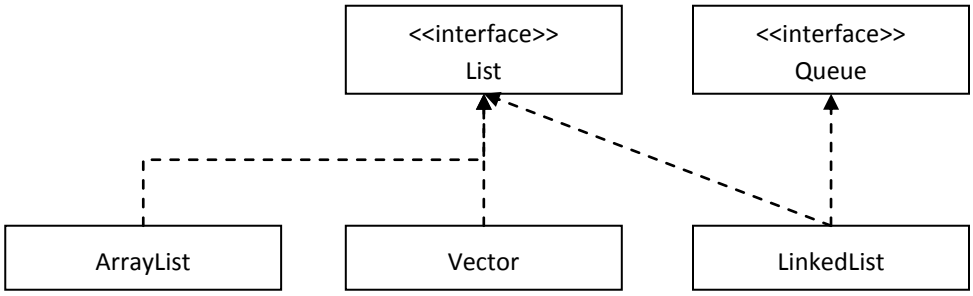


Cada colección va a tener un atributo de Distribución y otro de Orden.

- Distribución: Indica la forma en que los elementos son almacenados en base a las reglas de ordenamiento.
  - Distribuido Los elementos son ordenados según un conjunto de reglas.
  - No distribuido Los elementos no se ordenan de manera definida.
- Orden: Indica la forma en que los elementos son accedidos.
  - Ordenado Se itera en los elementos de una forma definida.
  - Desordenado Se itera en los elementos de una forma que pareciera casi aleatorea.

List <<interface>>

Todas las listas manejan sus colecciones mediante índices.
Permiten agregar un elemento dentro de la colección en un índice específico, obtener un elemento de la colección según el índice, y buscar en que posición se encuentra un objeto en particular.



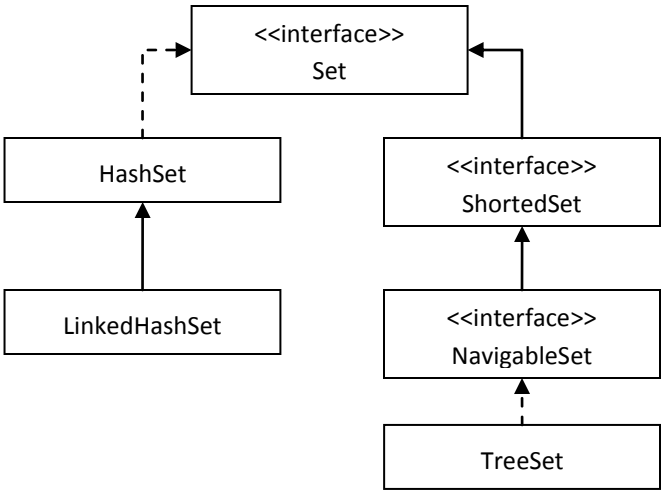
¿Tenés lo que se necesita para ser un SCJP?

http://valorcreativo.blogspot.com

ArrayList	Funciona como un array que puede modificar su tamaño dinámicamente. Permite acceder aleatoriamente a sus elementos.
Vector	Se comporta exactamente igual que el ArrayList, con la salvedad de que es Thread-Safe (multihilo seguro).
LinkedList	Mantiene un índice aparte, de manera que se puede utilizar como una cola, o como un simple ArrayList. Añade métodos como agregar al comienzo o al final.

Set <<interface>>

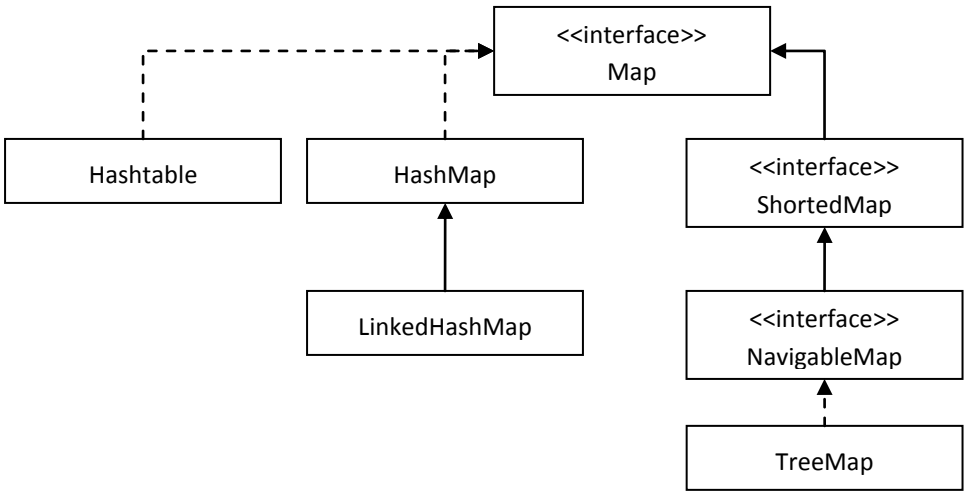
Los sets solo manejan objetos únicos, impidiendo su duplicidad dentro de la colección.
Para saber si un objeto ya se encuentra agregado en la colección, es necesario que se implemente el método equals().



HashSet	Los códigos se almacenan sin un orden definido, utilizando como identificador su código de hash.
LinkedHashSet	Se comporta como HashSet con la salvedad de que esta mantiene otro índice, el cual almacena el orden de los elementos según fueron siendo insertados en la colección.
TreeSet	Mantiene los elementos ordenados según su "orden natural". Los objetos almacenados en esta colección requieren implementar Comparable o Comparator.

Map <<interface>>

Los maps corresponden con la funcionalidad de los sets, con la diferencia de que los maps no tienen restricciones en cuanto a la duplicidad de sus elementos.
Al igual que los sets, requiere que se implemente equals() y hashCode().



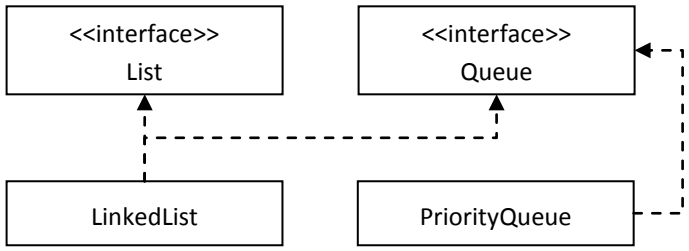
HashMap	Los códigos se almacenan sin un orden definido, utilizando como identificador su código de hash.
Hashtable	Se comporta como HashMap, con la salvedad de que es Thread-Safe (multihilo seguro).
LinkedHashMap	Se comporta como HashMap con la salvedad de que esta mantiene otro índice, el cual almacena el orden de los elementos según fueron siendo insertados en la colección.
TreeMap	Mantiene los elementos ordenados según su "orden natural". Los objetos almacenados en esta colección requieren implementar Comparable o Comparator.

¿Querés ser un SCJP?

http://gustavoalberola.blogspot.com

Queue <<interface>>

Representan un listado de "cosas por hacer". El comportamiento puede variar, pero se comportan por defecto como una cola o FIFO, First-In – First-Out (Primero en entrar – primero en salir).



LinkedList	Ver interface List.
PriorityQueue	Es una lista por defecto FIFO con prioridad. Requiere la implementación de Comparable.

Resumen de las clases de collection

Clase	Map	Set	List	Queue	Distribución (order)	Orden (sorted)
HashMap	X				No	No
Hashtable	X				No	No
TreeMap	X				Por orden (sorted)	Por orden natural o reglas de comparación.
LinkedHashMap	X				Por orden de inserción o último acceso.	No
HashSet		X			No	No
TreeSet		X			Por orden (sorted)	Por orden natural o reglas de comparación.
LinkedHashSet		X			Por orden de inserción	No
ArrayList			X		Por índice	No
Verctor			X		Por índice	No
LinkedList			X	X	Por índice	No
PriorityQueue				X	Por orden (sorted)	Por orden de prioridades.

ArrayList

java.util.ArrayList. Representa un array que puede modificar su tamaño dinámicamente.
Ventajas:

- Puede crecer dinámicamente
- Provee mecanismos de inserción y búsqueda más potentes que los de un Array.

Ahora que conoces las interfaces de las colecciones, lo mejor es utilizarlas. Veamos como instanciar un ArrayList.

```
import java.util.List;
import java.util.ArrayList;
//... código
List myArray = new ArrayList();
//O también podemos hacer uso de genéricos
List<String> myArray = new ArrayList<String>();
```

No te preocupes por los genéricos, ya los veremos en detalle más adelante.

¿Tenés lo que se necesita para ser un SCJP?

http://valorcreativo.blogspot.com

Autoboxing con colecciones

Las colecciones solo pueden majera objetos, no primitivas. En versiones previas a Java 5, si queríamos agregar una primitiva a un Array la teníamos que codificar manualmente dentro de su respectivo Wrapper. A partir de Java 5, de esto se encarga el autoboxing.

```
//Antes de Java 5
miArrayList.add(new Integer(15));

//A partir de Java 5
miArrayList.add(15);
```

Ordenando colecciones

Los métodos que veremos a continuación, solo se encuentran disponibles para las colecciones List.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class PruebasCollections_001 {
  static public void main(String[] args) {
    List<String> myArray = new ArrayList<String>();
    myArray.add("JavaWorld");
    myArray.add("Gustavo Alberola");
    myArray.add("Leonardo Blanco");
    myArray.add("Matias Alvarez");
    //Mostramos el array sin orden
    System.out.println("Desordenado: " + myArray.toString());
    //El ordenamiento se realiza con la clase Collections
    Collections.sort(myArray);
    //Mostramos el array ordenado
    System.out.println("Ordenado   : " + myArray.toString());
  }
}
```

Como puedes observar, el ordenamiento se realizó con el método estático sort de la clase java.util.Collections.

```
Desordenado: [JavaWorld, Gustavo Alberola, Leonardo Blanco, Matias
Alvarez]
Ordenado   : [Gustavo Alberola, JavaWorld, Leonardo Blanco, Matias
Alvarez]
```

Pero, si intentamos hacer este ejercicio con elementos dentro del ArrayList que no sean ni Strings ni Wrappers de primitivas, obtendremos un error.

La interfaz comparable

Si queremos hacer que nuestras propias clases se puedan ordenar, deberemos de implementar dicha interfaz y definir el método compareTo.
La firma de este método es int compareTo(Object).
El valor devuelto debe ser:

- Negativo Si this < Object
- 0 Si this == Object
- Positivo Si this > Object

El método sort utilizará el método compareTo para ordenar el array.

¿Querés ser un SCJP?

Existen dos maneras de declarar el método, mediante genéricos y mediante la notación antigua.

```
//Utilizando genéricos
class ImplementaGenerico implements Comparable<ImplementaGenerico> {
  //... código
  public int comparteTo(ImplementaGenerico obj) {
    //... código
  }
}

//Utilizando notación antigua
class ImplementaGenerico implements Comparable {
  //... código
  public int compareTo(Object obj) {
    if (obj instanceof ImplementaGenerico) {
      ImplementaGenerico tmpObj = (ImplementaGenerico) obj;
      //... código
    }
  }
}
```

Utilizando genéricos, automáticamente podemos especificar como parámetro el mismo tipo de la clase, en cambio, bajo la notación antigua, debemos castear el objeto recibido a dicho tipo.

La interfaz comparator

Bien, vimos la interfaz Comparable, pero, ¿que pasaría si quisiéramos ordenar una Clase que no podemos modificar, u ordenar en más de una forma diferente?

Comparator al rescate!

Si vemos el método sort, podemos encontrar una sobrecarga que recibe como segundo parámetro un Comparator.

Veamos cómo crear uno.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class PruebasCollections_002 {
  private String nombre;
  private String apellido;
  public PruebasCollections_002(String nombre, String apellido) {
    this.nombre = nombre;
    this.apellido = apellido;
  }
  public String getNombre() { return nombre; }
  public String getApellido() { return apellido; }
  public String toString() { return nombre + " " + apellido; }

  static public void main(String[] args) {
    List myArray = new ArrayList();
    myArray.add(new PruebasCollections_002("Gustavo", "Alberola"));
    myArray.add(new PruebasCollections_002("Matias", "Alvarez"));
    myArray.add(new PruebasCollections_002("Leonardo", "Blanco"));
    myArray.add(new PruebasCollections_002("Java", "World"));
    //Mostramos la coleccion tal cual está ahora
    System.out.println("Desordenada          : " + myArray.toString());
    //Ordenamos por nombre
    Collections.sort(myArray, new ComparatorByNombre());
    System.out.println("Ordenada por nombre  : " + myArray.toString());
```

http://valorcreativo.blogspot.com

```
    //Ordenamos por apellido
    Collections.sort(myArray, new ComparatorByApellido());
    System.out.println("Ordenada por apellido: " + myArray.toString());
  }
}

class ComparatorByNombre implements Comparator<PruebasCollections_002> {
  public int compare(PruebasCollections_002 obj1, PruebasCollections_002 obj2) {
    return obj1.getNombre().compareTo(obj2.getNombre());
  }
}

class ComparatorByApellido implements Comparator {
  public int compare(Object obj1, Object obj2) {
    if (obj1 instanceof PruebasCollections_002 && obj2 instanceof
PruebasCollections_002) {
      return
((PruebasCollections_002)obj1).getApellido().compareTo(((PruebasCollections_002)obj2).
getApellido());
    } else {
      return -1;
    }
  }
}
```

```
class ComparatorByNombre implements Comparator<PruebasCollections_002> {
  public int compare(PruebasCollections_002 obj1, PruebasCollections_002 obj2) {
    return obj1.getNombre().compareTo(obj2.getNombre());
  }
}

class ComparatorByApellido implements Comparator {
  public int compare(Object obj1, Object obj2) {
    if (obj1 instanceof PruebasCollections_002 && obj2 instanceof
PruebasCollections_002) {
      return
((PruebasCollections_002)obj1).getApellido().compareTo(((PruebasCollections_002)obj2).
getApellido());
    } else {
      return -1;
    }
  }
}
```

```
Desordenada          : [Gustavo Alberola, Matias Alvarez, Leonardo
Blanco, Java World]
Ordenada por nombre  : [Gustavo Alberola, Java World, Leonardo Blanco,
Matias Alvarez]
Ordenada por apellido: [Gustavo Alberola, Matias Alvarez, Leonardo
Blanco, Java World]
```

¿Querés ser un SCJP?

Comparable vs Comparator

Ya vimos ambas interfaces, entonces, ¿Cuándo utilizar una o la otra?
Cuando solo queramos definir un método de ordenamiento en una clase cuyo código fuente podemos acceder y modificar, utilizamos Comparable.
Cunado queremos implementar varios métodos de ordenamiento, u ordenar una clase cuyo código no tenemos o no podemos modificar, utilizamos Comparator.

Ordenando con la clase Arrays

Las firmas de los métodos son iguales que Collections: sort(Object) y sort(Object, Comparator). El problema radica en que un Array, a diferencia de un List, puede contener elementos de tipo primitivo. ¿y, cual es el problema entonces?
Simple, veamos la firma de Comparator:
public int compare(Object obj1, Object obj2)
¿Sigues sin verlo? Bueno, aquí va una pista: **Object.** Este solo recibe objetos, no primitivas. A menos que se almacenen en sus respectivos Wrappers, podemos utilizar las sobrecargas de sort(Xx) que existen para cada primitivo.

Recuerda, sort(Xx) esta sobrecargado para todos los tipos de primitivas, siendo Xx el tipo, pero cuando utilizas un Comparator sort(Object, Comparator) no puedes jamás utilizar un array con primitivas.

Búsqueda en Arrays y Collections

Ambas clases permiter realizar búsquedas dentro de colecciones. Para ello, necesitamos conocer las siguientes reglas:

- Las búsquedas son realizadas utilizando el método binarySearch().
- Las búsquedas exitosas devuelven un **int** con el índice donde se encuentra el objeto.
- Si la búsqueda no encuentra el valor, devuelve un valor negativo.
- La colección o array sobre la cual buscar, debe de haber sido previamente ordenada.
- **Si no cumples con el paso anterior, el resultado puede ser impredecible**.
- Si la colección o array se encuentra en un orden natural, la búsqueda se debe realizar en un orden natural.
- Si la colección o array fue ordenada con un Comparator, se debe utilizar el mismo Comparator para realizar la búsqueda, pasándolo como segundo argumento al método binarySearch().

Veamos un ejemplo:

```
import java.util.Comparator;
import java.util.Arrays;

public class PruebasCollections_003 {
  static public void main(String[] args) {
    String[] myArray = {"uno", "dos", "tres", "cuatro"};
    for(String s : myArray) {
      System.out.print(s + " ");
    }
    System.out.println();
    System.out.println("Posicion de \"tres\": " + Arrays.binarySearch(myArray,
"dos"));
    NuevoOrdenamiento ordenador = new NuevoOrdenamiento();
    Arrays.sort(myArray, ordenador);
    for(String s : myArray) {
      System.out.print(s + " ");
    }
```

http://valorcreativo.blogspot.com

```
    System.out.println();
    System.out.println("Posicion de \"tres\": " + Arrays.binarySearch(myArray,
"dos"));
    System.out.println("Posicion de \"tres\": " + Arrays.binarySearch(myArray, "dos",
ordenador));
  }
}

class NuevoOrdenamiento implements Comparator<String> {
  public int compare(String obj1, String obj2) {
    return obj2.compareTo(obj1);
  }
}
```

```
uno dos tres cuatro
Posicion de "tres": 1
uno tres dos cuatro
Posicion de "tres": -1
Posicion de "tres": 2
```

Dado que en la segunda búsqueda (luego del nuevo ordenamiento), no utilizamos el mismo Comparator para la búsqueda, nos dio como resultado -1 (resultado inesperado), luego, cuando utilizamos el mismo Comparator, la búsqueda resultó exitosa.

Recuerda, antes de buscar en una colección, esta debe de haber sido previamente ordenada, y la búsqueda debe realizarse en el mismo orden.

Conversion de Arrays a Lists a Arrays

Para pasar de Array a List, tenemos el método Arrays.asList(). La firma del método es:
static public <T> List<T>(T… elementsOrArray)
Algo muy importante para recordar. Si se le pasa un array, generará un List cuyos elementos apuntan a la misma dirección de memoria que los elementos del Array, de manera que si modificamos los valores en uno, se verán reflejados en el otro.
Para pasar de List a Array, tenemos el método toArray(). Este puede generar un array de objetos, o se le puede pasar un array de un tipo determinado.

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
public class PruebasCollections_004 {
  static public void main(String[] args) {
    Integer[] myArrayDeInts = { 1, 3, 12, 5, 47};
    List myListDeInts = Arrays.asList(myArrayDeInts);

    System.out.println(myListDeInts.size());
    System.out.println(myListDeInts);
    myListDeInts.set(4, 456);
    for(int a : myArrayDeInts) {
      System.out.print(a + " ");
    }
    System.out.println();
```

¿Querés ser un SCJP?

```
    Object[] myArrayDeObjects = myListDeInts.toArray();
    Integer[] myArrayDeIntegers = new Integer[5];
    myListDeInts.toArray(myArrayDeIntegers);

    myArrayDeIntegers[2] = 555;

    for(int a : myArrayDeInts) {
      System.out.print(a + " ");
    }
    System.out.println();

    for(int a : myArrayDeIntegers) {
      System.out.print(a + " ");
    }
  }
}
```

```
5
[1, 3, 12, 5, 47]
1 3 12 5 456
1 3 12 5 456
1 3 555 5 456
```

Si intentas convertir un array de primitivas en un List, no generaras un error, pero no se generará correctamente el List. El intentar acceder luego a alguno de sus elementos puede lanzar un ArrayIndexOutOfBoundsException.

http://gustavoalberola.blogspot.com

¿Tenés lo que se necesita para ser un SCJP?

http://valorcreativo.blogspot.com

Utilizando los Lists

Lo más común para iterar a través de los elementos de un List es un iterador (también se puede utilizar un simple for-each, pero con un iterador es menos acoplamiento).
El iterador posee dos métodos simples:

- boolean hasNext() Indica si aun quedan más elementos. No nos desplaza de la posición acutal dentro del array.
- Object next() Obtiene el elemento actual, y desplaza a la siguiente posición del array.

A partir de Java 5, los iteradores también soportan genéricos. Veamos un ejemplo:

```
import java.util.List;
import java.util.Iterator;
import java.util.ArrayList;

public class PruebasCollections_005 {
  static public void main(String[] args) {
    List<String> myLista = new ArrayList();
    myLista.add("JavaWorld");
    myLista.add("Dragon Ball");
    myLista.add("Pampita");
    Iterator<String> it = myLista.iterator();
    while (it.hasNext()) {
      String s = it.next();
      System.out.print(s + " ");
    }
    System.out.println();

    Iterator it2 = myLista.iterator();
    while (it2.hasNext()) {
      String s = (String)it2.next();
      System.out.print(s + " ");
    }
  }
}
```

```
JavaWorld Dragon Ball Pampita
JavaWorld Dragon Ball Pampita
```

Como podrán apreciar, cuando utilizamos el iterador con genéricos, no hubo que castear el tipo de dato, ya que automáticamente se obtiene el String. En el segundo caso, es necesario realizar el casteo explícitamente.

¿Querés ser un SCJP?

Utilizando los Sets

Recordemos que los sets almacenan objetos únicos (no existen dos elementos que respondan a x.equals(y) == true).
En los Sets, los métodos add devuelven un valor de tipo **boolean**, que indica si el valor fue o no agregado a la colección.
Veamos un ejemplo:

```
import java.util.Set;
import java.util.HashSet;
import java.util.TreeSet;

public class PruebasCollections_006 {
  static public void main(String[] args) {
    boolean[] resultados = new boolean[4];
    Set mySet = new HashSet();
    resultados[0] = mySet.add(new Integer(45));
    resultados[1] = mySet.add("b");
    resultados[2] = mySet.add(new Object());
    resultados[3] = mySet.add(45); //Se produce un autoboxing

    System.out.println("Resultados para el HashSet");
    for(boolean res : resultados) {
      System.out.println(res);
    }


    mySet = new TreeSet();
    resultados[0] = mySet.add("a");
    resultados[1] = mySet.add("b");
    resultados[2] = mySet.add("a");
    resultados[3] = mySet.add("c");

    System.out.println("Resultados para el TreeSet");
    for(boolean res : resultados) {
      System.out.println(res);
    }

    mySet.add(new Integer(45)); //Al llegar a esta línea se produce una excepción.
  }
}
```

```
Consola
Resultados para el HashSet
true
true
true
false
Resultados para el TreeSet
true
true
false
true
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
  at java.lang.Integer.compareTo(Unknown Source)
  at java.util.TreeMap.put(Unknown Source)
  at java.util.TreeSet.add(Unknown Source)
  at PruebasCollections_006.main(PruebasCollections_006.java:XX)
```

¿Tenés lo que se necesita para ser un SCJP?

http://valorcreativo.blogspot.com

Dado que un Tree (árbol) tiene por defecto ordenamiento natural, los elementos deben de poder compararse entre sí.

Utilizando los Maps

Los maps se manejan almacenando pares clave/valor. Veamos un poco más a fondo como funciona un Map para inserción y para obtención de datos.
Veamos un ejemplo en código:

```
import java.util.Map;
import java.util.HashMap;

public class PruebasCollections_007 {
  static public void main(String[] args) {
    Map<Object, Object> myMap = new HashMap<Object, Object>();
    Persona p1 = new Persona(45);
    Persona p2 = new Persona(30);
    Animal a1 = new Animal(30);

    myMap.put(p1, "Clave persona1");
    myMap.put(p2, "Clave persona2");
    myMap.put(a1, "Clave animal1");

    System.out.println("1 - " + myMap.get(new Persona(45)));
    System.out.println("2 - " + myMap.get(new Persona(30)));
    System.out.println("3 - " + myMap.get(a1));
    System.out.println("4 - " + myMap.get(new Animal(30)));
    System.out.println("Cant:" + myMap.size());
  }
}

class Persona {
  private int edad;
  public int getEdad(){ return edad; }
  public Persona(int edad){ this.edad = edad; }
  public boolean equals(Object persona) {
    if (persona instanceof Persona && edad == ((Persona)persona).getEdad()) {
      return true;
    } else {
      return false;
    }
  }
  public int hashCode(){ return edad; }
}

class Animal {
  int edad;
  public int getEdad(){ return edad; }
  public Animal(int edad){ this.edad = edad; }
}
```

¿Querés ser un SCJP?

```
1 - Clave persona1
2 - Clave persona2
3 - Clave animal1
4 - null
Cant:3
```

Bien, pero… no entendí nada de nada…
Ok, si estas dentro de este grupo, veamos como funcionan las colecciones Map.

Insersión:

- Se ingresa el par clave/valor
- De la clave se obtiene el int del hashCode.
- Se almacena el valor en un grupo identificado por el hashCode del paso anterior.

Obtención:

- Se especifica la clave
- De esta se obtiene el hashCode y se busca el grupo
- Si el hashCode existe, se comparan mediante equals las claves.
- Si son equivalentes, se devuelve el valor

En el caso de Animal, contiene tanto el método HashCode como equals redefinido, de manera que al crear una nueva clase, se puede encontrar la clave.
Cuando utilizamos la misma instancia de Animal (a1), esta primero genera el mismo HashCode (utilizando el hash por defecto, generando algo como Animal@44532), y luego utiliza el equals. Dado que este último no esta sobrescrito, lo que hace es verificar las posiciones de memoria. Dado que es la misma instancia (esta apuntando al mismo objeto), el resultado es true.
Pero cuando creamos un nuevo objeto de tipo Animal, por más que este tenga las mismas características, al no haber sobrescrito los métodos equals y hashCode, jamás podremos obtener dicho valor, y el mismo quedará perdido dentro del HashMap.
Es muy común ver ejemplos en los que como clave utilizan Strings, bien, recuerda que tanto los Strings como los Wrappers de los tipos primitivos sobrescriben equals y hashCode, de manera que es posible utilizarlos como clave.

http://valorcreativo.blogspot.com

Búsqueda dentro de los TreeSets y TreeMaps

Vimos como navegar a travez de Arrays y Colecciones, ahora veremos como hacerlo a través de los trees.
Para ello utilizamos los métodos de las 2 nuevas interfaces: java.util.NavigableSet y java.util.NavigableMap.
El siguiente cuadro representa un resumen de los métodos encontrados en ambas clases:

Clase	Método	Descripción	Abrebiatura
TreeSet	ceiling(e)	Devuelve el valor más cercano a e que sea valor >= e.	Val >= e
TreeMap	ceilingKey(key)	Devuelve la clave más cercana a key que sea keyDevuelta >= key.	Val >= key
TreeSet	higher(e)	Devuelve el valor más cercano a e que sea valor > e.	Val > e
TreeMap	higherKey(key)	Devuelve la clave más cercana a key que sea keyDevuelta >= key.	Val > key
TreeSet	floor(e)	Devuelve el valor más cercano a e que sea valor <= e.	Val <= e
TreeMap	floorKey(key)	Devuelve la clave más cercana a key que sea keyDevuelta <= key.	Val <= key
TreeSet	lower(e)	Devuelve el valor más cercano a e que sea valor < e.	Val < e
TreeMap	lowerKey(key)	Devuelve la clave más cercana a key que sea keyDevuelta < key.	Val < key
TreeSet	pollFirst()	Devuelve y elimina la primer entrada.	
TreeMap	pollFirstEntry()	Devuelve y elimina el primer par clave/valor.	
TreeSet	pollLast()	Devuelve y elimina la última entrada.	
TreeMap	pollLastEntry()	Devuelve y elimina el último par clave/valor.	
TreeSet	descendingSet()	Devuelve un NavigableSet en orden inverso.	
TreeMap	descendingMap()	Devuelve un NavigableMap en orden inverso.	

¿Querés ser un SCJP?

Colecciones respaldadas

En inglés denominadas “backed collections”, representan un subset de una colección completa el cual no es simplemente una copia del último, sino que quedan enlazados, de manera que ciertos cambios en uno pueden afectar a otro, y viceversa.
Veamos un ejemplo:

```
import java.util.TreeMap;
import java.util.SortedMap;

public class PruebasCollections_008 {
  static public void main(String[] args) {
    TreeMap<String, String> map = new TreeMap<String, String>();
    SortedMap<String, String> sorted = null;
    //Cargamos algunos valores en el map
    map.put("a", "Anaconda");
    map.put("r", "Rambo");
    map.put("t", "Terminator");
    map.put("i", "Indiana Jones");

    sorted = map.subMap("c", "o");

    System.out.println("Map    :" + map);
    System.out.println("Sorted:" + sorted);

    map.put("j", "Jurassik Park");
    map.put("o", "Open Seasson");
    map.put("v", "V for Vendetta");

    System.out.println();
    System.out.println("Map    :" + map);
    System.out.println("Sorted:" + sorted);

    sorted.put("f", "Final Fantasy VII Advent Children");
    //sorted.put("z", "Zorro"); Si ejecutaramos esta línea tendríamos una excepcion
IllegalArgumentException

    System.out.println();
    System.out.println("Map    :" + map);
    System.out.println("Sorted:" + sorted);
  }
}
```

```
Map    :{a=Anaconda, i=Indiana Jones, r=Rambo, t=Terminator}
Sorted:{i=Indiana Jones}

Map    :{a=Anaconda, i=Indiana Jones, j=Jurassik Park, o=Open Seasson,
r=Rambo, t=Terminator, v=V for Vendetta}
Sorted:{i=Indiana Jones, j=Jurassik Park}

Map    :{a=Anaconda, f=Final Fantasy VII Advent Children, i=Indiana
Jones, j=Jurassik Park, o=Open Seasson, r=Rambo, t=Terminator, v=V for
Vendetta}
Sorted:{f=Final Fantasy VII Advent Children, i=Indiana Jones, j=Jurassik
Park}
```

¿Tenés lo que se necesita para ser un SCJP?

http://valorcreativo.blogspot.com

Generamos un subMap del primero de tipo SortedMap, Pero los cambios que hacemos en uno o en otro afectan entre si.
La línea que se encuentra comentada que indica que generaría un error, es debido a que un SortedMap se encuentra limitado al rango de keys con que lo hayamos generado, en nuestro caso de c a o. Al insertar una z, esta no entra en c <= z < o.

Métodos para crear colecciones respaldadas

Método	Descripción
headSet(e, b*)	Devuelve un subset de todos los elementos que sean < e (`int`).
headMap(e, b*)	Devuelve un subset de todas las claves que sean < e (`int`).
tailSet(e, b*)	Devuelve un subset de todos los elementos que sean >= e (`int`).
tailMap(e, b*)	Devuelve un subset de todas las claves que sean >= e (`int`).
subSet(a, b*, e, b*)	Devuelve un subset de todos los elementos que sean >= a (`int`) y < e (`int`).
subMap(a, b*, e, b*)	Devuelve un subset de todas las claves que sean >= a (`int`) y < e (`int`).

Nota: b* representa un valor de tipo boolean, el cual es en realidad una sobrecarga del método. Si no se especifica el parámetro boolean, se devuelve un SortedXxx. En cambio, si se especifica, se devuelve un NavigableXxx.

Cola de prioridades (PriorityQueue)

La PriorityQueue ordena los elementos según reglas predefinidas. Por defecto esta regla es por ordenamiento natural.

Métodos de PriorityQueue

Métodos	Descripción
peek()	Obtiene el elemento actual.
poll()	Obtiene el elemento actual y lo elimian de la colección.
offer(o<T>)	Añade un elemento a la colección,
size()	Al igual que en cualquier colección, devuelve la cantidad de elementos que esta contiene.

Para crear nuestro propio algoritmo de ordenamiento para una ProrityQueue, debemos de utilizar una clase que:

- Implemente (`implements`) Comparator<T>
- Defina el método public `int` compare(Integer uno, Integer dos)

Nota: recuerda que solo los Arrays pueden contener elementos de tipo primitivos, de manera que compare utiliza Integer y no `int`.

Resúmen de métodos para Arrays y Colecciones

java.util.Array	Descripción
static List **asList**(<T>[])	Conviete un Array de tipo <T> en una Lista y enlaza los elementos.
static `int` **binarySearch**(Object[], key) static `int` **binarySearch**(primitivo[], key)	Busca dentro de un Array **ordenado** por un valor, y devuelvel índice en que se encuentra.
static `int` **binarySearch**(<T>[], key, Comparador)	Busca dentro de una Array **ordenado con un comparador** por un valor en particular.
static `boolean` **equals**(Object[], Objet[]) static `boolean` **equals**(primitivo[], primitivo[])	Compara dos Arrays para determinar si su contenido es semejante.
public static void **sort**(Object[]) public static void **sort**(primitivo[])	Ordena los elementos de un Array por orden natural.
public static void **sort**(<T>[], Comparador)	Ordena los elementos de un Array utilizando un Comparador.
public static String **toString**(Object[]) public static String **toString**(primitive[])	Crea una cadena con el contenido del Array (invoca los toString de cada elemento).

¿Querés ser un SCJP?

java.util.Collections	Descripción
static **int** **binarySearch**(List, key) static **int** **binarySearch**(List, key, Comparator)	Busca dentro de un Array **ordenado** por un valor, y devuelve el índice que se encutra.
static **void** reverse(List)	Invierte el orden de los elementos en la lista.
static **Comparator reverseOrder**() static **Comparator reverseOrder**()	Devuelve un Comparador que ordena la colección al revés
static void **sort**(List) static void **sort**(List, Comparator)	Ordena una lista ya sea por orden natural l,

Resumen de métodos para List, Set y Map

Métodos	List	Set	Map	Queue	Descripción
boolean **add**(elemento)	X	X			Añade un elemento. En el caso de los List, puede especificar la posición donde agregarlo.
boolean **add(**índice, elemento)	X				
boolean **contains**(Objet)	X	X			Busca dentro de los elementos de la colección un objeto (o en los Maps una clave), y devuelve como resultado un **boolean**.
boolean **containsKey**(Objet key)			X		
boolean **containsValue**(Objet valor)			X		
Object **get**(índice)	X				Obtiene un objeto de la colección a través del índice o la clave.
Object **get**(key)			X		
int **indexOf**(Object)	X				Obtiene la ubicación (índice) de un objeto dentro de la colección.
Iterator **iterator**()	X	X			Obtiene un iterador para poder recorrer la colección.
Set **keySet**()			X		Genera un Set con todas las claves del Map.
offer(<T>)				X	Añande un elemento a la Queue.
<T> **peek**()				X	Devuelve el objeto actual de la Queue. En el segundo caso también es eliminado de la colección.
<T> **poll**()				X	
put(key, valor)			X		Agrega un par clave/valor a un Map.
remove(indice)	X				Elimina un elemento de la colección.
remove(Object)	X	X			
remove(key)			X		
int **size**()	X	X	X	X	Devuelve el número de elementos en la colección.
Object[] **toArray**()	X	X			Devuelve un array con los elementos de la colección.
<T>[] **toArray**(<T>[])	X	X			

http://gustavoalberola.blogspot.com

¿Tenés lo que se necesita para ser un SCJP?

http://valorcreativo.blogspot.com

Genéricos

Los genéricos son una manera de especificar el tipo de dato de un elemento que por naturaleza es más abstracto. Generalmente es utilizado en las colecciones.
Dado que estas contienen elementos, su naturaleza gira en torno a los diferentes tipos de datos que puedan albergar.
Antes de java 5, un List como un ArrayList, contenía solo tipo Object, o sea, cualquier cosa que no fuera un primitivo. Aunque por un lado parece tentador, por otro lado nos incurre en tener que castear cada elemento para poder utilizar sus métodos, y verificar que el mismo sea de dicho tipo (Is-A).
La solución a esto son los genéricos. Estos permiten especificar el tipo de dato que contendrá, y verificarlo en tiempo de compilación, de manera que te aseguras de que los tipos de datos son válidos para la colección, evitando casteos y verificaciones de por medio.
A su vez, Java ideo una manera que permite la retro compatibilidad, permitiendo utilizar colecciones sin genéricos. En estos, a la hora de compilar la aplicación, nos muestra una advertencia. También veremos que se pueden llegar a dar serior problemas si mezclamos colecciones genéricas con colecciones no genéricas.

Un vistazo a colecciones con y sin genéricos

Primero veamos un ejemplo sin genéricos:

```
import java.util.ArrayList;

class Perro {
  public void ladrar(){ System.out.println("wouf!"); }
}

public class PruebasGenerics_01 {
  static public void main(String[] args) {
    ArrayList lista = new ArrayList();
    lista.add("Hola");
    lista.add(15);
    lista.add(new Perro());

    for(Object o : lista) {
      if (o instanceof Perro) {
        ((Perro)o).ladrar();
      }
    }
  }
}
```

Primero noten la falta de restricción en cuanto a tipos de datos que pudimos agregar en el ArrayList (String, Integer, Perro).
Luego, si lo que queremos es invocar los ladridos de todos los perros de la colección, tenemos primero que verificar que el elemento sea de tipo Perro (Is-A). recién sabiendo esto, podemos castearlo a Perro e invocar el método ladrar.
Por el contrario, si no hubiéramos añadido la verificación instanceof, al primer intento habríamos recibido una excepción en tiempo de ejecución ClassCastException.

¿Querés ser un SCJP?

Ahora, veamos el mismo ejemplo con genéricos:

```
import java.util.ArrayList;

class Perro {
  public void ladrar() { System.out.println("wouf!"); }
}

public class PruebasGenerics_02 {
  static public void main(String[] args) {
    ArrayList<Perro> lista = new ArrayList<Perro>();

    //lista.add("Hola"); //Genera un error de compilación
    //lista.add(11);     //Genera un error de compilación
    lista.add(new Perro());

    for(Perro p : lista) {
      p.ladrar();
    }
  }
}
```

Primera gran diferencia, no se puede añadir ningún elemento a la colección que no corresponda a la relación Is-A Perro. Por esa razón comentamos las líneas en que agregamos el String e Integer, de lo contrario recibiremos un error de compilación.
Segundo, dado que especificamos el tipo, no es necesario castear los elementos de la colección, y de manera que estamos seguros de que todos son perros, no necesitamos realizar la verificación de la condición Is-A Perro.

Como verán, genériocos provee una manera limpia de definir un tipo de dato con el cual se manejará la colección, de manera de minimizar errores.

Mezclando colecciones genéricas y no genéricas

Dado que para conservar la compatibilidad de código anterior a Java 5 se debían de poder seguir utilizando colecciones no genéricas, puede llegar a darse el caso de que ambas se mezclen. Veamos un ejemplo:

```
import java.util.ArrayList;

class ContenedoraArrayNoGenerico {
  static public void agregarElementoEnArray(ArrayList list) {
    list.add("Hola");
    list.add(45);
    list.add(new ArrayList());
  }
}

public class PruebasGenerics_03 {
  static public void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(40);
    list.add(new Integer(63));

    System.out.println("Antes: " + list.toString());
    ContenedoraArrayNoGenerico.agregarElementoEnArray(list);
    System.out.println("Despues: " + list.toString());
  }
}
```

http://valorcreativo.blogspot.com

Ahora, si el ArrayList original es genérico, y está como de tipo Integer, ¿no debería de generar un error de compilación?... Pues no. La salida por pantalla es:

```
Antes: [40, 63]
Despues: [40, 63, Hola, 45, []]
```

Como es fácil apreciar, las limitaciones de tipo de dato en genéricos solo aplican a aquellos que hayan sido declarados como tales, de manera que si estamos utilzando código antiguo, posiblemente un algoritmo que hicimos hace mucho o una librería de terceros, debemos de tener cuidado con esto.

De todas maneras, que el compilador lo permita no quiere decir que esté totalmente de acuerdo y nos muestra una advertencia de compilación. El mensaje que muestra al compilar el código es:

```
Note: PruebasCollections_004.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Traducido al criollo, significa algo como “Hay ciertas operaciones que no son seguras, la ejecución de este código puede llegar a desatar un cataclismo y tu inmediato despido. No digas después que no te avisé!”.

Salvo que en las opciones de una pregunta diga “avertencia”, una colección no genérica no produce errores de compilación, de manera que lee con cuidado antes de contestar este tipo de preguntas.

Polimorfismo y genéricos

Otra cuestión a tener en cuenta es que tipos de datos se pueden declarar y en que tipo de variables de referencia se pueden almacenar. Muy simple, siempre deben de ser del mismo tipo. No vale un hujo o padre.
Si creamos un ArrayList de objetos, no es correcto hacer:

```
ArrayList<Object> list = new ArrayList<String>();
```

Por más que String califica para la condición Is-A Object, este código no es válido. La única manera es:

```
ArrayList<Object> list = new ArrayList<Object>();
```

También podemos asignarlo a un List, pero siempre que sea del mismo tipo:

```
List<Object> list = new ArrayList<Object>();
```

Declaración de genéricos

Comodín	Descripción
<T>	Representa cualquier tipo de dato. Este se debe especificar cuando se cree un objeto concreto.
<E>	Representa cualquier tipo de dato pero solo para colecciones. Este se debe especificar cuando se cree un objeto concreto.
<Xxx>	Representa un tipo de dato concreto. Solo podrá recibir parámetros del mismo tipo.
<? super Xxx>	Representa un tipo de dato concreto. Solo podrá recibir parámetros del mismo tipo o super tipos.
<? extends Xxx>	Representa un tipo de dato concreto. Solo podrá recibir parámetros del mismo tipo o sub-tipos. El extends en genéricos aplica a las interfaces también,

Xxx representa un tipo de dato.

Tanto T como E son solo nomenclaturas diseñadas para generar código que sea legible y fácil de entender bajo un estándar común. De todas maneras, es posible utilizar cualquier letra.
El carácter **?** solo puede ser utilizado para declaración de colecciones. Cuando se utilice en clases o métodos, este debe ser reemplazado por una letra.
Los comodines solo pueden especificarse en ciertos lugares específicos. Introducirlo en otro lugar generará un error de compilación.

¿Querés ser un SCJP?

Lo que se viene

En la próxima entrega estaremos adentrándonos en el mundo de la programación Orientada a objetos.

Conoceremos como se compone una clase, y como es la sintaxis de esta. También veremos interfaces, conceptos como herencia, polimorfismo, y otros.

Aprenderemos sobre las relaciones que existen entre objetos como Is-A (es u) y As-A (tiene un).

Y veremos cómo extender una clase, o implementar una interface.

Comodines en la declaración de clases

```
public class Almacen<T> {
  List<T> lista;

  public Almacen() { lista = new ArrayList<T>; }
  public Almacen(T lista) { this.lista = lista; }

  public T getLista() { return lista; }
  public void addToLista(T item){lista.add(item);}
}
```

Se declara luego del nombre de la clase. De ahí en más, cada tipo de dato que requiera corresponder con este, utiliza la letra sin los <>. Siempre que se cree un objeto que a su vez sea un genérico, se deben de utilizar los caracteres <>.

Comodines en la declaración de métodos

```
public <T> realizarAccion(T item) {
      //...
}

//En cabio, no existe la declaración de un método
public void realizarAccion(<T> item)

//Tampoco se puede hacer un
public void realizarAccion(T item)
//Salvo que la clase se declarara como
NombreClase<T>
```

http://gustavoalberola.blogspot.com

¿Tenés lo que se necesita para ser un SCJP?

Valor Creativo

Equilibrio entre la vida profesional y la vida personal

Un problema común que afecta al ser humano, y por en ende a su vida personal, es el estrés. Para hablar de ello lo definiremos como: “la respuesta de adaptación de la persona a un estímulo que le genera excesiva demanda psicológica o física”.

El stress es necesario para poder mejorar y adaptarse a un nuevo ambiente, y es algo común y necesario en la vida de todo ser vivo para poder sobrevivir. Los efectos negativos del stress se hacen notorios cuando se somete al cuerpo por un largo período de tiempo.

Las presiones del trabajo, la falta de sueño, el no hacer deporte y alimentarse mal es la combinación necesaria para producir un desequilibrio tanto físico como mental en una persona.

En las empresas de software, la calidad de sus productos se basa en la creatividad e ingenio de su capital humano. Los gerentes de RRHH saben muy bien que una persona bien motivada y libre de preocupaciones rinde el doble, y que una persona que trabaja incomodo, que no le gusta su empleo y que además tiene problemas en la casa difícilmente rinda lo que debería rendir.

Veamos un caso muy común de un operario en la ciudad de Buenos Aires. Un operario trabaja 8 horas de lunes a sábado y posee un franco a la semana, en la ciudad las distancias son más largas y por ello tiempo unas 3 horas de viaje entre ida y vuelta al trabajo. Pero esa persona se tiene que preparar para ir a trabajar debe despabilarse, desayunar e higienizarse y ello dura aproximadamente 1 hora y media. En síntesis ese operario le dedica a su trabajo unas 12 horas y media, además ese operario tiene que dormir y ello debería durar unas 8 hs. Por lo tanto le queda unas 3 horas y media para hacerse la comida, pagar los impuestos, lavar la ropa, limpiar la casa, jugar con los hijos, atender al conyugue, etc. **Como conclusión el tiempo no alcanza, la empresa solo percibe 8 horas de trabajo, no ve lo que hay detrás y quiere cada vez más, todo esto genera un ambiente de stress destructivo.** Este es el motivo por el cual la gente escapa de Buenos Aires en los veranos con o sin capital en el bolsillo.

En el mundo de los programadores analizaremos su entorno de trabajo en un ambiente de baja calidad institucional. Un programador que trabaja 8 horas frente a un monitor CRT de 15 pulgadas, sufre cansancio y desgaste de la vista. En asientos no ergonómicos trae aparejados problemas de postura y de columna. En teclados no ergonómicos y la mala altura de los mismos provocan síndrome de túnel carpiano. Estar todo el dia encerrado pensando en lenguaje binario en un ambiente de poca luz afecta la capacidad de relacionarse con otras personas y en su exceso desorden psicológico. El no tener contacto con la luz solar provoca deficiencia de vitamina e, un mineral necesario para regular el sistema digestivo en las personas. Las personas sometidas a estas condiciones de trabajo y que sufren muchas presiones les causa un envejecimiento prematuro, caída de pelo, problemas renales y falta de concentración.

¿Querés ser un SCJP?

Lo que se busca es que el empleado este bien tanto en el ambiente de trabajo como en su vida cotidiana, este es la conciliación que las mejoras empresas aplican y les da resultados envidiables. Un ejemplo es Google, que entendió la importancia de restarles problemas a sus empleados en el trabajo y también los de la vida cotidiana. Lo primero que hizo es eliminar los tiempos de trabajo, allí no se posee un horario fijo de 8 a 16 hs sino que se le asignan tareas con una fecha estimativa de tiempo de entrega y sus empleados lo realizan cuando lo deseen, es decir cuando sus mentes estén lo suficientemente abiertas para hacer el trabajo lo mejor y más eficiente posible. Esto no es lo único que hizo Google empezó a solucionar problemas cotidianos como lo es la comida que incorpora un buffet gourmet, lavar la ropa con una lavandería y tintorería, el transporte pasando a buscar a los empleados con un vehículo con conexión WIFI para que comiencen a trabajar desde el colectivo, les pagan sus impuestos, y se encargan de sus hijos con una guardería y una educación bien paga. Además incorporo un spa, un gimnasio, una pileta, un salón de juegos, una sala de mini cine y hasta un masajista para que despejen la mente. Muchos piensan que Google es una fiesta y yo les digo puede ser, pero por algo es la empresa número uno del 2008 para trabajar según la revista Fortune.

Uno no espera que las empresas de nuestro país en vías desarrollo aplique esta filosofía de cuidar a sus empleados en el corto plazo, pero hay buenas prácticas que se pueden aplicar y que a veces valen más que un aumento de sueldo. Por ejemplo:

- **Pagarles el gimnasio u los partidos de futbol.** El deporte es el principal cable a tierra de las personas y hay que fomentarlo ya que regula la circulación de sangre en el cuerpo e irriga el cerebro para que este más predispuesto a pensar.
- **Semana comprimida.** Esta metodología se esta empezando a usar mucho en la Argentina, que trata que el personal trabaje 10 horas de lunes a jueves. Esto le da un día libre más a las personas para descansar y además se ahorran esas 4 horas y media de preparación y viaje al trabajo.
- **Horario de verano.** En dicha estación del año se cambian los horarios para que el personal pueda disfrutar más del día.
- **Pago de los impuestos del personal.** Esta modalidad no es usada por las empresas y trae muchos beneficios para ambas partes. Por cada peso con cincuenta que pone el empleador, al empleado le llega un peso. Todo esto es debido a las obligaciones impositivas. Pagar los impuestos al empleado implica una percepción directa de un beneficio de la empresa e implica un ahorro de tiempo importante, y para la empresa significa un ahorro del 50% en obligaciones impositivas, mas beneficios por pagar impuestos en cantidad, más que se pueden ingresar en los costos contables de la empresa que significa un ahorro del iba. **Es una simbiosis perfecta que sería muy útil en empresas de mas de 20 empleados, esperemos que algún día se utilice.**

¿Tenés lo que se necesita para ser un SCJP?

Mi consejo

Programadores:

La remuneración y beneficios ofrecidos por una empresa deben ser analizados fríamente antes de dar una respuesta. Porque bien se puede ganar mucho dinero, pero en las consecuencias del stress laboral no hay vuelta atrás (perder años de vida y plata en médicos), si vas a hacer ese sacrificio que sea por una buena causa.

Hay empresas que son flexibles, te pagan bien, te cuidan y saben que el que estés tranquilo hace les rinda más y mejor. Solo hace falta encontrarlas y sino juntarse con compañeros crearlas.

Empresas:

En esta época se va a notar más y más que la tecnología ya es un comodity que lo puede tener cualquiera y por ello lo que diferencia a una empresa de otra son sus personas. Hay que cuidarlas, motivarlas, capacitarlas e integrarlas a la empresa como parte de ellas.

No solo hay que solucionarles problemas en el trabajo sino también en la casa, ya que no existen las personas perfectas maquinas de trabajo. Hay problemas que son universales en la vida cotidiana y se pueden solucionar.

Un mensaje es “no hay mejor empleado que el que se siente parte de la empresa y que no tiene preocupaciones, solo piensa en trabajar y disfrutar la vida”.

Estudiantes:

A veces uno resigna salidas, abandona el deporte y descuida a sus amigos para poder estudiar. Mi consejo es que si queres hacerlo y ves que podes hacerlo porque sino estas tranquilo con vos mismo las cosas te van a costar el doble aprenderlas. Después te vas a arrepentir de no haber hecho esas cosas.

Como mensaje final “todo sacrificio debe tener su recompensa, y si estudiastes entonces descansa y salí a disfrutar la vida, porque ese es tu premio”.

Hasta acá ha llegado el tema de equilibrio en la vida profesional y vida personal espero que les haya gustado. Los invito a visitar el sitio ValorCreativo.blogspot.com en donde pueden encontrar el informe completo por si quieren profundizar en el tema.

Mucha suerte y nos vemos en la próxima edición.

VALOR CREATIVO

¿Querés ser un SCJP?

¿Tenés lo que se necesita para ser un SCJP?

Autores
Gustavo Alberola
Matias Álvarez

Diseño
Leonardo Blanco