Sumario

Collections y Generics

Genéricos

Convertir tipos es el arma más importante al utilizar polimorfismo. Sin embargo esto deja abierta una puerta peligrosa, sobre todo cuando se utiliza Object , porque cualquier tipo puede ser convertido a esta clase.
Los genéricos restringen al código para que los tipos se conviertan en tiempo de compilación en lugar de tiempo de ejecución. Dado este hecho, si un tipo no es “convertible” se detecta en tiempo de compilación y no de ejecución.
Los generics aseguran un mejor diseño de software. La razón es que aquello que es diseñado para ser polimórfico se utilice como tal y aquello que no, se especifique con que tipo se quiere utilizar.
A partir de Java 1.5 se agregaron una serie de extensiones al lenguaje Java y una de ellas fueron los genéricos. Estos son similares a los templates de C++. Los genéricos permiten abstraerse de los tipos de datos. Los ejemplos más comunes de esto son las clases contenedoras, como por ejemplo la cadena de herencia Collection.

```
//Código de Ejemplo sin Generic
public class Pila{
      public void add(Object o){}
      public Object get(int indice){}
}

Pila pila=new Pila();
pila.add(“Hola”);
pila.add(23);     //La pila es Object.


//Código de Ejemplo con Generic
public class Pila <E>{
      public void add(E e){}
      public E get(int indice){}
}

Pila<String>pila=new Pila();
pila.add(“Hola”); //el método add esta limitado a recibir String
pila.add(23);     //error por que no es String.
```

Genéricos y tipos base

Una pregunta válida es ¿se pueden utilizar genéricos con los tipo base como int o long ? La respuesta es un categórico NO.

Sin embargo, la duda persiste porque el lenguaje posee la capacidad de autoboxing, ¿no debería manejar automáticamente su capacidad de convertir los tipos base? Nuevamente la respuesta es NO.

Las razones de estas respuestas no son simples y residen mucho en la filosofía de diseño del lenguaje. Primero, los tipos base son valores numéricos, con lo cual no hay forma de verificar sus tipos salvo por su tamaño, lo cual implicaría un crecimiento interno notable de reglas de validación para el compilador y sus respectivas consecuencias en el rendimiento. Además, incrementaría el número de reglas a aplicar en todo lo que intervenga un tipo base, como promociones automáticas o conversiones de tipo. Por sus consecuencias es evidente que no se incluyó por estos u otros motivos.

Se puede definir entonces que **Los genéricos en Java sólo funcionan con objetos como parámetros.**

Por otra parte, esto debe quedar en claro porque los parámetros genéricos no se comportan como tipos base en las declaraciones. Por ejemplo, si se intenta declarar un vector de un tipo genérico el compilador dará un mensaje de error cuando se intente crear las referencias mediante new . La razón es simple, se puede crear una referencia a un tipo de objeto sin necesidad de conocer su estructura, lo cual no es válido en un tipo base.

El operador diamante

A partir de la versión 1.7, se puede declarar un parámetro genérico en un tipo y no repetir la declaración “nuevamente” a la derecha del operador new . Esto, aunque parece una tontería, reduce notablemente la cantidad de código escrito.

```
A<Double> a2 = new A<>();
// de esta forma:
A<Double> a2 = new A();
```

Limitación en los parámetros de tipo

Existen situaciones en las cuales se desea limitar los parámetros que se puedan asignar a un tipo cuando se declara un objeto. Para esto, Java provee un mecanismo a través de las cadenas de herencia que permite establecer un límite de manera que un parámetro se pueda asignar a un tipo sólo si pertenece a la cadena de herencia declarada.
Para declarar el límite para un parámetro de tipo, se debe escribir el nombre de éste, seguido de la palabra clave extends , seguida de su límite superior (superclase de la cadena o sub cadena de herencia), que en el siguiente ejemplo es Number . Nótese que, en este contexto, extends se utiliza en un sentido general para significar "extender" (como en las clases) o "implementar" (como en las interfaces).

```
public class Test<T> {
      public <U extends Number> void inspeccionar(U u) {
            System.out.println("T: " + t.getClass().getName());
            System.out.println("U: " + u.getClass().getName());
      }
}
```

Invariancia

El uso de genéricos asegura el tipo. Es más, su principal objetivo es asegurar el tipo utilizado por un determinado objeto. Pero, como se puede asignar cualquier tipo al parámetro genérico, ¿qué pasa si se asigna a un tipo a un parámetro cuando se crea un objeto y se trata de asignar a una referencia de ese tipo, que se creó con un parámetro diferente?
Es claro que nunca estos objetos pueden ser “exactamente del mismo tipo” porque el compilador asignó diferentes tipos al parámetro genérico y es como si con declaraciones estándar se hubiesen hecho dos clases diferentes. Al mecanismo que asegura el tipo de esta forma se lo llama invariancia y su principal función es asegurar el tipo.

```
public class VerificaLimitesParametros {
      public static void main(String[] args) {
            Z<A> z1 = new Z<>();
            Z<B> z2 = new Z<>();
```

```
            Z<C> z3 = new Z<>();
            z1 = z2; // ERROR
            z2 = z3; // ERROR
      }
}
```

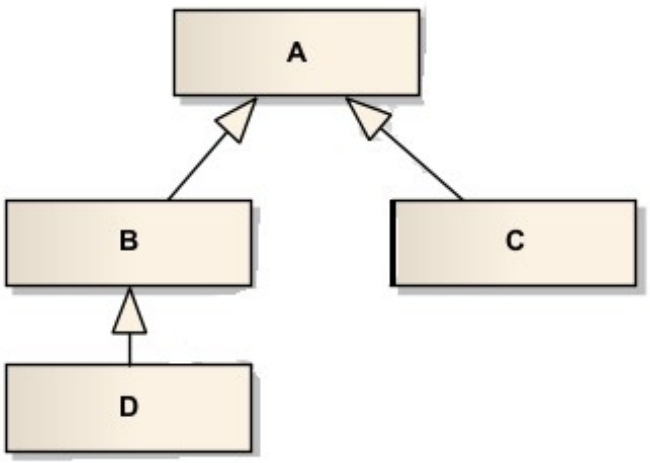
z1 se creó con un parámetro genérico del tipo A , por lo tanto, intentar asignar un objeto que se creó con un parámetro del tipo B es incompatible.

Covarianza

Se define a la covarianza de la siguiente manera:

> *Cuando en las asignaciones se preserva el orden natural de los subtipos respecto a sus súper tipos en una asignación (de lo más específico a lo más genérico) la asignación es covariante*

```
public class Z <T>{
      private T t;
      public void agregar(T t) {
            this.t = t;
      }
      public T obtener() {
            return t;
      }
      public <U extends A> void inspeccionar(U u) {
            System.out.println("T: " + t.getClass().getName());
            System.out.println("U: " + u.getClass().getName());
      }
}
```




```
public class VerificaLimitesParametros {
      public static void main(String[] args) {
            Z<A> z1 = new Z<>();
            Z<B> z2 = new Z<>();
            Z<C> z3 = new Z<>();
            Z<D> z4 = new Z<>();
            z1.agregar(new A());
            z1.inspeccionar(new A());
            z1.inspeccionar(new B());
            z1.inspeccionar(new C());
            z1.inspeccionar(new D());
            // La siguiente línea da Error porque el parámetro
            // del tipo para z2 es B y A es superclase
            // z2.agregar(new A());
            z2.agregar(new B());
            // z2.agregar(new C());
```

```
                z2.agregar(new D());
                z2.inspeccionar(new A());
                z2.inspeccionar(new B());
                z2.inspeccionar(new C());
                z2.inspeccionar(new D());
                // z3.agregar(new A()); // Error por ser superclase
                // z3.agregar(new B()); // Error por estar en otra cadena de
                                        // herencia
                z3.agregar(new C());
                // La siguiente línea da Error porque el parámetro
                // del tipo para z3 es C y D esta en otra cadena de herencia
                // z3.agregar(new D());
                z3.inspeccionar(new A());
                z3.inspeccionar(new B());
                z3.inspeccionar(new C());
                z3.inspeccionar(new D());
                z4.agregar(new D());
                z4.inspeccionar(new A());
                z4.inspeccionar(new B());
                z4.inspeccionar(new C());
                z4.inspeccionar(new D());
        }
}
```

Parámetros desconocidos declarados con caracteres comodines

Al codificar se presentan situaciones como las siguientes cuando se quiere crear un parámetro genérico limitado al declarar una referencia (este tipo de declaraciones se utilizan para indicar que tipos admiten que se le asignen una determinada referencia):

- Se desconoce la superclase del parámetro
- Se desconoce la subclase del parámetro

Java admite estos dos estilos de declaraciones usando un carácter de comodín, el símbolo ?.
Cuando se desconoce la subclase, pero se quiere declarar el parámetro la declaración es:

```
<? extends A>
```

Esta declaración debe leerse de la siguiente manera:

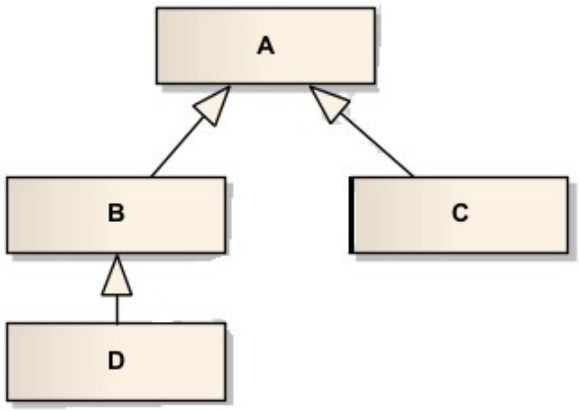
El parámetro es un tipo desconocido que es A o una subclase de A
Cuando lo que se desconoce es la superclase, la declaración se transforma en:

```
<? super D>
```

Que deberá leerse como

- El parámetro es un tipo desconocido que es D o una superclase de D.

Si se pretendiera declarar referencias a la clase Z de esta manera, las mismas deberán respetar el límite impuesto.



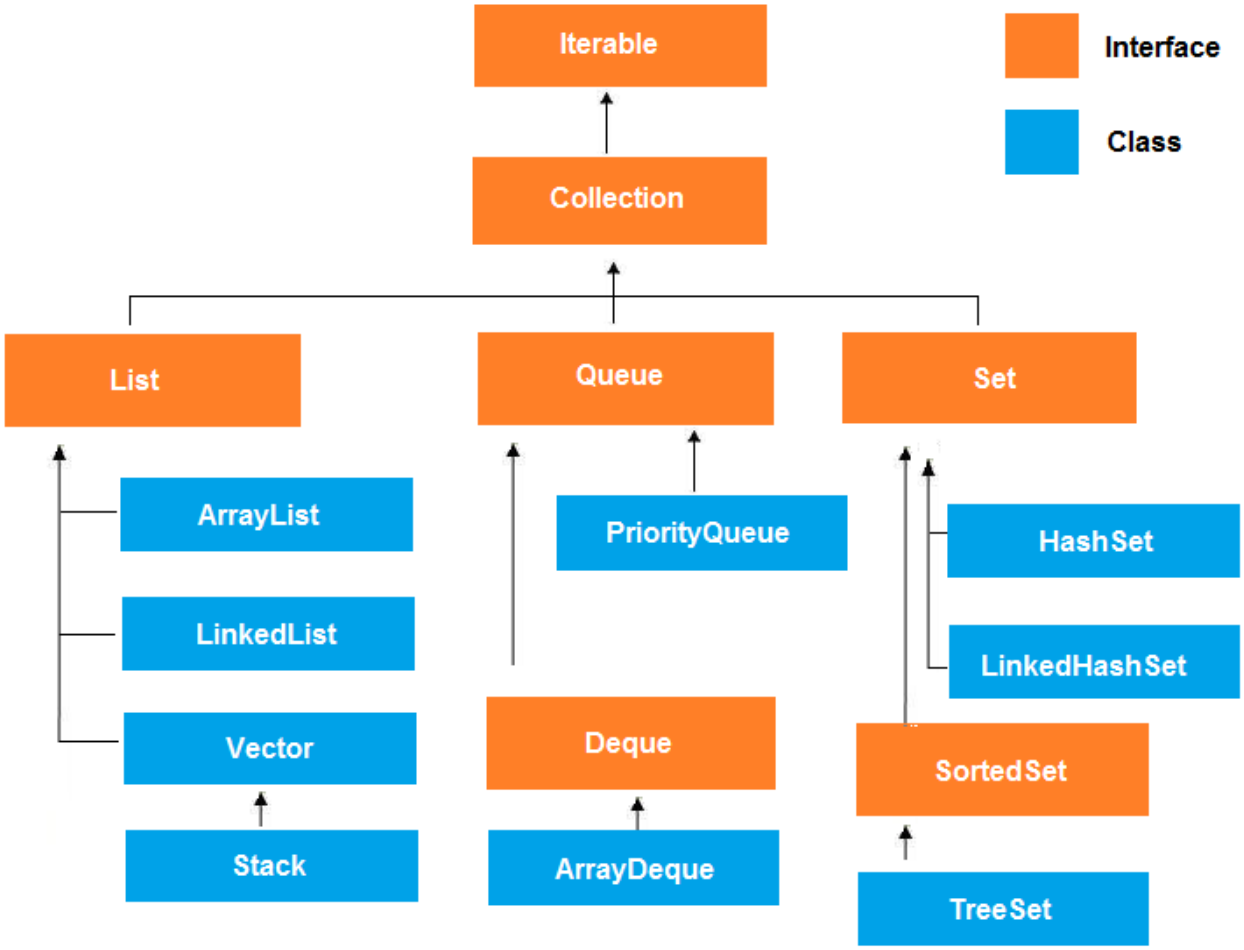

```
Z<? extends A> z1 = new Z<B>();
Z<? extends A> z2 = new Z<C>();
Z<? extends A> z3 = new Z<D>();
Z<? super C> z4 = new Z<A>();
Z<? super C> z5 = new Z<B>(); // Error
Z<? super C> z6 = new Z<A>();
z1.agregar(new A()); // Error
z1.agregar(new B()); // Error
z1.agregar(new C()); // Error
z1.agregar(new D()); // Error
z1.agregar(new A()); // Error
z2.agregar(new B()); // Error
z2.agregar(new C()); // Error
z2.agregar(new D()); // Error
z2.agregar(new A()); // Error
```

El método agregar tiene un prototipo como el siguiente

```
public void agregar(T t)
```

Si fuera posible agregar de la misma manera en Z un tipo A como un subtipo de A , la declaración de la variable genérica de Z , que define su estado, podría ser B , C o D , lo cual implica que la definición de Z puede ser un objeto de cualquiera de estos tipos, y esto es incongruente, ya que un objeto puede asignarse por conversión, pero siempre sobre el mismo tipo de variable.

La API de Colecciones



Antecedentes

Las colecciones son un concepto intuitivo implementado desde los comienzos del lenguaje en diversas clases, algunas de uso interno y otras provistas por las API.
La estructura de colecciones está pensada como un conjunto de interfaces de las cuales especializan clases concretas para su uso. Este formato brinda flexibilidad en el manejo de toda la estructura para la declaración y gestión de referencias en las asociaciones con otras clases. Otro beneficio adicional de esta forma de implementación, es que los cambios que se puedan suscitar con las diferentes versiones no afecten a las implementaciones existentes, ya que ante la necesidad de funcionalidades específicas, se puede implementar las interfaces necesarias y especializar la clase según el requerimiento en particular.

Por lo tanto, para entender el framework de colecciones (estructura de clases e interfaces para el trabajo con colecciones) es fundamental comprender la filosofía de diseño y el funcionamiento esperado a través de las interfaces.

Las colecciones se pueden interpretar como un conjunto, recordando que este último es una serie de elementos únicos agrupados, por lo tanto, no contiene duplicados o como listas, que si contienen duplicados pero conservan el orden en que se agregaron los elementos. Las colecciones, como son un concepto genérico, debe admitir en su diseño ambas posibilidades. En Java por cuestiones de diseño, se separan los conceptos en dos interfaces independientes que admiten estas funcionalidades distintas: Set y List . La primera no admite duplicados mientras que la segunda sí.

La API Collection contiene interfaces que agrupan objetos como:

- **Collection:** Un grupo de objetos llamados elementos. Cualquier orden o la falta de este y la posibilidad de duplicados se especifica en cada implementación.
- **Set:** Una colección sin orden. No se permiten duplicados.
- **List:** Una colección ordenada. Se permiten duplicados.

La interfaz Iterator

Esta interfaz es similar a Enumeration (la cual se explicará posteriormente, aunque está en desuso), sólo cambian los métodos utilizados. Las operaciones definidas soportan la navegación de una colección, la recuperación de un elemento y su remoción.

Las operaciones de la interfaz Collection

Esta interfaz soporta operaciones básicas como insertar y remover objetos. Por ejemplo, si se quiere remover objetos, con invocar al método apropiado se saca tan sólo una instancia del mismo si existiese.

Los métodos para agregar y remover son:

```
boolean add(Object element)
boolean remove(Object element)
```

Otros métodos de utilidad para consultas sobre el estado de la colección o su gestión son:

```
int size()
boolean isEmpty()
boolean contains(Object element)
Iterator iterator()
```

Operaciones grupales

```
boolean containsAll(Collection collection)
boolean addAll(Collection collection)
void clear()
void removeAll(Collection collection)
void retainAll(Collection collection)
```

Set

La interface Set, representa una lista que no permite valores duplicados.
Un ejemplo de la vida real de conjuntos puede ser el siguiente:

- Las letras minúsculas de la “a” a la “z”
- Los número naturales
- Los protocolos de una red
- Las medidas de longitud

De los ejemplos citados, se pueden definir una serie de propiedades respecto de los conjuntos, como ser:

- Existe una instancia de cada ítem (como en los ejemplos anteriores)
- Pueden ser finitos (protocolos de una red) o infinitos (números naturales)
- Pueden definir conceptos abstractos (medidas de longitud)

Como se mencionó anteriormente, Set es una interfaz que aquellas clases que la implementen **Deben ser colecciones que no admitan duplicados.**

```
//Ejemplo de uso
Set set;
//Seleccionar una implementación  HashSet -  TreeSet - LinkedHashSet
set = new HashSet();
//set = new TreeSet();
//set = new LinkedHashSet();
set.add("uno");
set.add("segundo");
set.add("3ro");
set.add(new Integer(4));
set.add(new Float(5.0F));
set.add("segundo ");
set.add(new Integer(4)); // duplicado, no se agrega
System.out.println(set);
```

Implementaciones clásicas de SET

- HashSet(): Es la más veloz, pero no garantiza el orden de los elementos.
- TreeSet(): Almacena los elementos en un árbol por orden natural.
- LinkedHashSet(): Almacena los elementos en una lista enlazada por orden de ingreso.

Nota: *ninguna implementación de Set permite ingreso de valores duplicados.*

List

La interface List representa una lista que permite valores duplicados. Ejemplo de uso:

```
//Emplo de uso de un List
List list;
list = new ArrayList();
//list = new LinkedList();
list.add("uno");
list.add("segundo");
list.add("3ro");
list.add(new Integer(4));
list.add(new Float(5.0F));
list.add("segundo"); // duplicado, se agrega
list.add(new Integer(4)); // duplicado, se agrega
System.out.println(list);
```

Ejemplos de recorrido de una lista

```
list=new ArrayList();
list.add("Lunes");
list.add("Martes");
list.add("Martes");
list.add("Miércoles");
list.add("Jueves");
list.add("Viernes");

//Recorrido con indices
for(int a=0;a<list.size();a++) System.out.println(list.get(a));

//Recorrido forEach
for(String st:list) System.out.println(st);

//Recorrido usando el método default de java8
list.forEach(item->System.out.println(item));

//Recorrido abreviado java8
```

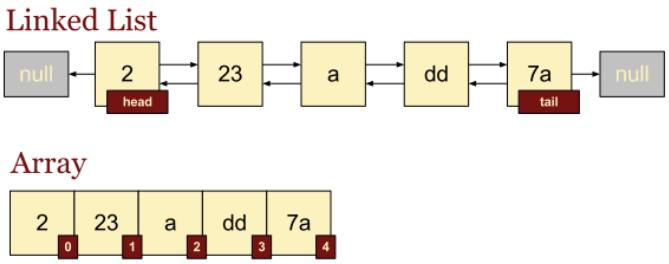
```
list.forEach(System.out::println);
```

Implementaciones clásicas de List

Consideramos Vector una implementación historica y obsoleta de List y descartamos su uso.

Diferencia entre ArrayList y LinkedList

LinkedList y ArrayList son dos diferentes implementaciones de la interfaz List. LinkedList usa internamente una lista doblemente ligada, mientras que ArrayList usa un arreglo que cambia de tamaño dinámicamente.
Podemos visualizarlas de la siguiente forma:



LinkedList permite **eliminar e insertar elementos en tiempo constante** usando iteradores, pero el acceso es secuencial por lo que encontrar un elemento toma un tiempo proporcional al tamaño de la lista.
Normalmente la complejidad de esa operación promedio sería O(n/2) sin embargo usar una lista doblemente ligada el recorrido puede ocurrir desde el principio o el final de la lista por lo tanto resulta en O(n/4).
Por otro lado ArrayList ofrece **acceso en tiempo constante** O(1), pero si quieres añadir o remover un elemento en cualquier posición que no sea la última es necesario mover elementos. Además si el arreglo ya está lleno es necesario crear uno nuevo con mayor capacidad y copiar los elementos existentes.

Ventajas y desventajas

LinkedList

Ventajas	Desventajas
Añadir y remover elementos con un iterador	Uso de memoria adicional por las referencias a los elementos anterior y siguiente
Añadir y remover elementos al final de la lista	El acceso a los elementos depende del tamaño de la lista

ArrayList

Ventajas	Desventajas
Añadir elementos	Costos adicionales al añadir o remover elementos
Acceso a elementos	La cantidad de memoria considera la capacidad definida para el ArrayList, aunque no contenga elementos

Tip: Si especificas el tamaño de un ArrayList al momento de crearlo se disminuye la cantidad de memoria usada porque no necesita aumentar de tamaño y copiar los elementos.

¿Cuál debo usar?

En la práctica la mayoría de las ocasiones es mejor usar ArrayList porque el tiempo de las operaciones y uso de memoria es menor que en LinkedList, de manera simple: **si no sabes cual usar usa ArrayList**.
Eso no quiere decir que LinkedList nunca se utilice, existen algunos casos muy específicos donde es la mejor opción, por ejemplo la pila de llamadas del lenguaje C está implementada usando una estructura de datos con estas características.

La interfaz Enumeration

Esta interfaz permite iterar sobre los elementos de una colección.
La nueva estructura de colecciones la remplazó por la interfaz Iterator la cual es utilizada en la mayoría de las situaciones. Sin embargo, muchas clases de utilidad todavía no soportan el uso de Iterator , por eso aún se sigue usando.
Ambas interfaces cumplen una función similar y conceptualmente son lo mismo, simplemente cambian los nombres de los métodos a utilizar.

Iteradores

Cuando un programa tiene un ciclo de ejecución, itera sobre las instrucciones definidas para dicho ciclo. El concepto de iterador en una colección es similar, salvo que se aplica a los elementos que posee, por lo tanto se puede definir para colecciones como:

- La iteración es el proceso de recuperar cada elemento en una colección

La interfaz Iterator

```
//Código de la interfaz Iteraror JDK8
public interface Iterator<E extends Object> {
    public boolean hasNext();
    public E next();
    public default void remove() {
        // compiled code
    }
    public default void forEachRemaining(Consumer<? super E> cnsmr) {
        // compiled code
    }
}
```

La Interface Iterable

La interface iterable está incluida en el api de Java, en concreto en el paquete **java.lang**. Es una interfaz de uso habitual al igual que el caso de las interfaces Cloneable y Comparable que ya hemos explicado. Implementar **Iterable** tan sólo obliga a sobreescribir un método que es **iterator().** Este método debe devolver un objeto de tipo Iterator. Vamos a explicar la terminología porque puede parecer confusa.

Lo primero es tener claro que hay que distinguir el método **iterator** (en minúsculas) y el tipo definido en el api de Java **Iterator** (con mayúsculas). Iterator con mayúsculas es un tipo definido por la Interface Iterator, igual que List es un tipo definido por la interface List. Por el contrario, iterator() con minúsculas es un método igual que puede ser toString() o cualquier otro. Esto hay que tenerlo claro desde el principio para no llevar a confusiones.

¿Qué es un objeto de tipo Iterator y cómo se implementa el método iterator()? Eso es lo que vamos a explicar a continuación, con detenimiento y a través de un ejemplo para entenderlo mejor.

Lo primero que vamos a recordar es que una interface es un tipo abstracto: no puede ser instanciado porque carece de constructor. Sin embargo, puede definirse un objeto del tipo definido por la interface si se instancia en una clase que implementa la interface. Esto puede parecer complicado pero con un ejemplo lo veremos claramente:

List <Persona> miListaDePersonas = new List<Persona> (); es erróneo ¿Por qué? Porque List es una interface y carece de constructor. En cambio sí sería una escritura correcta definir como de tipo List una colección que creamos instanciándola con una clase que tiene implementada la interface List como es ArrayList: List <Persona> miListaDePersonas = new ArrayList<Persona> ();

De la misma manera que no podemos usar un constructor de List, tampoco podremos usar un constructor para Iterator porque igualmente se trata de una interface sin constructor.

```
//Código de la interface Iterable JDK 8
public interface Iterable<T> {
    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }

    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

Método default forEach JDK8

Java 8 incorpora en la interface Iterable el método default forEach que permite realizar un autorecorrido de la collection. Poco a poco las expresiones Lambda se comienzan a utilizar. Una de las novedades es el uso de iteradores forEach en Java 8 .

Vamos a explicar como estos funcionan. Para ello vamos a partir de un ejemplo con un bucle forEach clásico que recorre una colección de Personas:

```
ArrayList<Persona> milista= new ArrayList<Persona>();
milista.add(new Persona("Miguel"));
milista.add(new Persona("Alicia"));

for (Persona p: milista) {
      System.out.println(p.getNombre());
}
```

Esta operación la podemos realizar de similar forma utilizando el método forEach de Java 8 que las colecciones soportan a través del interface Iterable. Así pues el nuevo código sería :

```
ArrayList<Persona> milista = new ArrayList<Persona>();
milista.add(new Persona("Miguel"));
milista.add(new Persona("Alicia"));

//Recorrido forEach
milista.forEach(item→System.out.println(item));

//Recorrido imprimiendo solo el nombre de la persona
milista.forEach(item->System.out.println(item.getNombre()));

//Recorrido forEach Abreviado usando el operador 4 puntos ::
milista.forEach(item→System.out::println);
```

Método de la interface List agregado en JDK 8

Java 8 agrego algunos métodos en la interface List, replaceAll() permite reemplazar todos los elementos de la lista usando una expresión lambda.

```
List<Integer> list = Arrays.asList(1, 2, 3);
list.replaceAll(x -> x*2);
System.out.println(list); // [2, 4, 6]
```

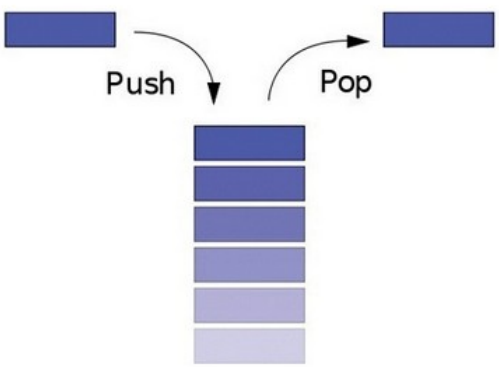
Vectores

Los vectores en Java están definidos para ser una colección de tipos primitivos de tamaño fijo, el cual es definido por una variable entera (Atención: utilizar otro tipo de datos para definir el tamaño de un vector genera un error, inclusive otro tipo entero como long).
El hecho que los vectores son objetos se hace evidente porque se puede acceder a propiedades de los mismos, como length o más aún, a los métodos que se obtienen por heredar de Object .

Las clases Vector y Stack

Vector es una clase histórica de Java. Permite la implementación de un vector de dimensión extensible por la incorporación de nuevos elementos. Esta clase fue remplazada en las nuevas versiones por **ArrayList** , sin embargo no fue depreciada en la actualidad y sigue vigente.

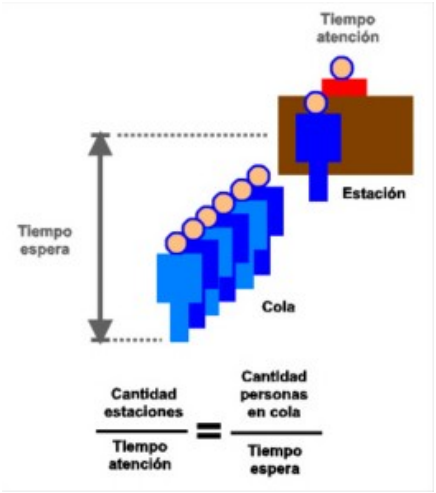
Por otro lado, la clase **Stack** implementa una pila y tiene como superclase a Vector . Si bien la operatoria de la pila se lleva a cabo con los métodos push y pop , se debe tener precaución en su forma de uso porque los métodos públicos de Vector siguen activos.
El concepto de pila se lo conocer como **LIFO**(por la sigla en ingles de “Last In – First Out”) - “El Ultimo en Entrar es el Primero en Salir”



Ejemplo de uso de una pila

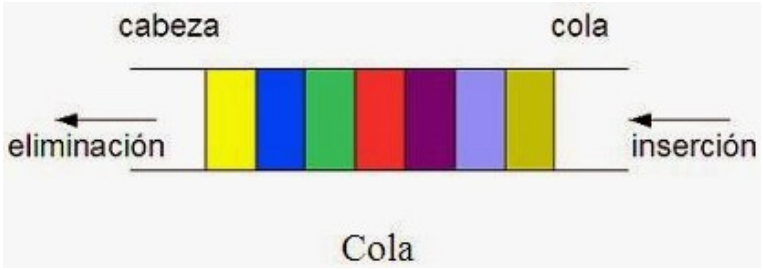
```
Stack<String> pila=new Stack();
pila.push("Lunes");             //método para apilar un elemento
pila.push("Martes");
System.out.println(pila.pop()); //método para desapilar un elemento.
```

Colas en Java Las clases Queue y Deque



Definición

Se conoce como cola a una colección especialmente diseñada para ser usada como almacenamiento temporario de objetos a procesar. Las operaciones que suelen admitir las colas son “encolar”, “obtener siguiente”, etc. Por lo general las colas siguen un patrón que en computación se conoce como **FIFO** (por la sigla en inglés de “First In - First Out” - “lo que entra primero, sale primero”), lo que no quiere decir otra cosa que lo obvio: El orden en que se van obteniendo los “siguientes” objetos coincide con el orden en que fueron introducidos en la cola. Esto análogo a su tocaya del supermercado: La gente que hace la cola va siendo atendida en el orden en que llegó a ésta.
Hasta hace poco, para implementar una cola FIFO en Java la única opción provista por la biblioteca de colecciones era LinkedList. Como ya se dijo más arriba, esta implementación ofrece una implementación eficiente de las operaciones “poner primero” y “sacar último”. Sin embargo, aunque la implentación es la correcta, a nivel de interfaz dejaba algo que desear. Los métodos necesarios para usar una LinkedList como una cola eran parte solamente de la clase LinkedList, no existía ninguna interfaz que abstraiga el concepto de “cola”. Esto hacía imposible crear código genérico que use indistintamente diferentes implementaciones de colas (que por ejemplo no sean FIFO sino que tengan algún mecanismo de prioridades). Esta situación cambió recientemente a partir del agregado a Java de dos interfaces expresamente diseñadas para el manejo de colas: La interfaz Queue tiene las operaciones que se esperan en una cola. También se creó Deque, que representa a una “double-ended queue”, es decir, una cola en la que los elementos pueden añadirse no solo al final, sino también “empujarse” al principio.
La implementación adecuada para Queue es PriorityQueue existe desde java 5.



Ejemplo de uso

```
Queue<String> cola=new PriorityQueue();
cola.offer("Lunes");             //método para encolar elementos
cola.offer("Martes");
System.out.println(cola.poll()); //método para descolar elementos
```

Mapas

Que es un mapa?

Un mapa es un caso particular del conjunto, donde cada elemento es un par “clave=valor”, definiendo así una relación bidireccional entre los componentes de cada elemento, ya que cada componente está “mapeado” al otro.

Algunos ejemplos simples de mapas pueden ser:

- Las direcciones IP y los nombres ingresados en un servidor DNS
- Las claves de una tabla y los registros asociados a cada una de ellas
- El nombre de una palabra y la cantidad de veces que aparece en un texto
- Los títulos de un libro y el número de página donde se encuentran
- Las conversiones de base numérica

Algunas propiedades que se pueden derivar de estos ejemplos son:

- Existe un solo par clave – valor en el mapa (todos los ejemplos anteriores)
- Pueden ser finitos (direcciones IP y nombres) o infinitos (conversiones de base numérica)
- Pueden definir conceptos abstractos (conversiones de base numérica)

Por definición, los objetos del tipo Map no admiten claves duplicadas y una clave solo puede asignarse a un valor. Si se proporciona otro valor para una clave existente, éste sobrescribe al anterior.
La interfaz Map proporciona tres métodos que permiten ver el contenido de un mapa como colecciones:

entrySet: devuelve una variable Set que contiene todos los pares formados por una clave y un valor.

keySet: devuelve una variable Set con todas las claves del mapa.
values: devuelve una variable Collection con todos los valores contenidos en el mapa.

```
//Métodos mas cómunes de la interfaz Map
public interface Map<K extends Object, V extends Object> {
      public int size();
```

```
    public boolean isEmpty();
    public boolean containsKey(Object o);
    public boolean containsValue(Object o);
    public V get(Object o);
    public V put(K k, V v);
    public V remove(Object o);
    public void putAll(Map<? extends K, ? extends V> map);
    public void clear();
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Entry<K, V>> entrySet();
    public boolean equals(Object o);
    public int hashCode();
}
```

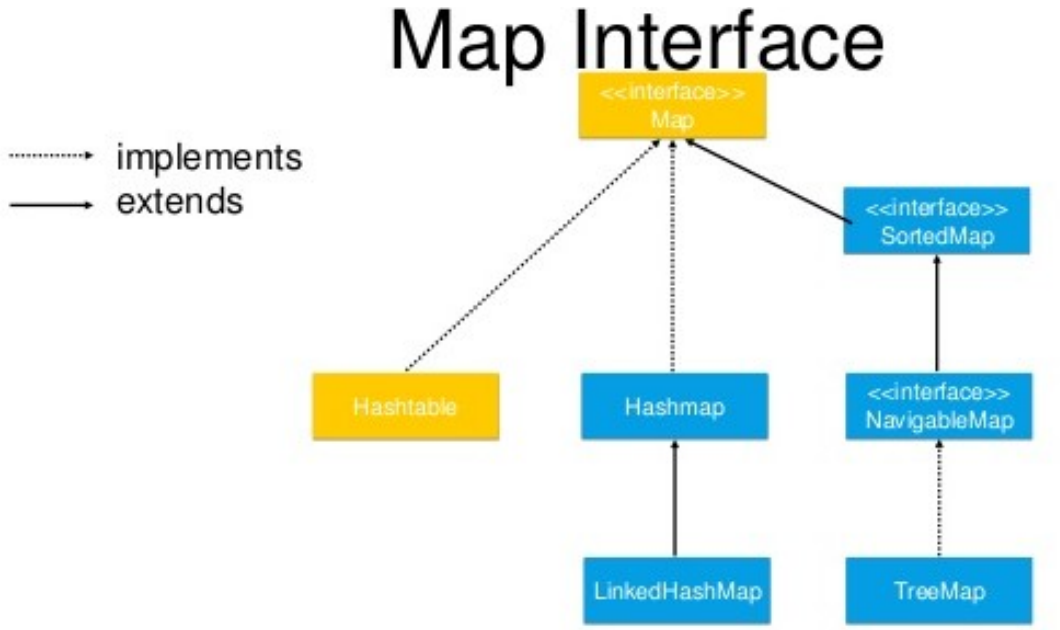
La interfaz Map no hereda la interfaz Collection porque representa asignaciones y no una colección de objetos. La interfaz SortedMap hereda de la interfaz Map. Algunas de las clases que implementan la interfaz Map son HashMap , TreeMap , IdentityHashMap y WeakHashMap .
El orden que presentan los iteradores de estas implementaciones de colección de mapas es específico de cada iterador.

```
//Ejemplo de uso de Map
Map mapa = new HashMap();
mapa.put("uno", "1ro");
mapa.put("segundo", new Integer(2));
mapa.put("tercero", "3o");
// Sobrescribe la asignación anterior
mapa.put("tercero", "III");
// Devuelve el conjunto de las claves
Set conjunto1 = mapa.keySet();
// Devuelve la vista Collection de los valores
Collection coleccion = mapa.values();
// Devuelve el conjunto de las asignaciones de claves a valores
Set conjunto2 = mapa.entrySet();
System. out .println(conjunto1 + "\n" + coleccion + "\n" +
      conjunto2);
```

Implementaciones clásicas de Map

HashMap: Es la más veloz no garantiza el orden de los elementos.
TreeMap: Alamacena los elementos en un arbol por orden natural de la clave.
LinkedHashMap: Almacena los elementos en una linsta enlazada por orden de ingreso.

Las Clases Dictionary, Hashtable, Properties.



La clase Dictionary es una clase abstracta llena de métodos abstractos, lo cual, a nivel de diseño indica claramente, que debería haber sido una interfaz. Esta clase se creó originalmente para mantener conjuntos de pares clave – valor y fue remplazada por Map .
Las clases que heredan de Dictionary son Hashtable y Properties y conforman parte del conjunto original de diseño de colecciones que poseía Java.
Hashtable es una clase que cumple las funciones de un diccionario permitiendo almacenar objetos y sus claves asociadas (excepto null). La versión nueva de esta clase es HashMap que implementa la interfaz Map al igual que Hashtable.
La clase Properties es una implementación de Hashtable para utilizar tan sólo cadenas de caracteres, por lo tanto, no se necesitan conversiones de tipo para extraer los elementos porque están manejados por su tipo (String) y no son un tipo Object (el resto de las colecciones clásicas se manejan con elementos de tipo Object , lo que obliga siempre a convertir el tipo al del elemento que se gestiona con dicha colección). Esta colección, como se mencionó anteriormente, permite la asignación de valores desde una corriente de ingreso, así como también guardar sus elementos en una corriente de salida.
Su uso más frecuente es a través de acceder al Singlenton cuya referencia se obtiene de System.getProperties() . Java utiliza Properties durante el inicio de un programa para guardar inicialmente las propiedades de la máquina virtual.

Se debe tener en cuenta que las versiones originales de clases para el manejo de colecciones siguen vigentes por cuestiones de compatibilidad y se pueden utilizar, sin embargo, en caso de ser posible, es conveniente utilizar las nuevas implementaciones.
A las clases antiguas se las nombra como **LegacyClass** o clases de legado.

Estas clases son thread-safe y por lo tanto es “importante” conocerlas y apreciar la diferencia respecto de las nuevas implementaciones que no lo son y dejan la responsabilidad de sincronizar memoria compartida al programador (tema que se verá posteriormente).

```
//Encabezado de la clase Hashtable
public class Hashtable<K extends Object, V extends Object> extends Dictionary<K,
V> implements Map<K, V>, Cloneable, Serializable {

      //Encabezado método put
      public synchronized V put(K k, V v) {
            // compiled code
    }
```

```
      //Encabezado método get
      public synchronized V get(Object o) {
            // compiled code
    }
}


//Encabezado de la clase HashMap
public class HashMap<K extends Object, V extends Object> extends AbstractMap<K,
V> implements Map<K, V>, Cloneable, Serializable {

      //Encabezado método put
      public V put(K k, V v) {
            // compiled code
      }

      //Encabezado método get
      public V get(Object o) {
            // compiled code
      }
}


      //Ejemplo de uso de un Hashtable
      Hashtable<String,String>table=new Hashtable();
      table.put("red", "Rojo");
      String valor1=table.get("red");

      Ejemplo de uso de un HashMap
      HashMap<String,String>mapa=new HashMap();
      mapa.put("red", "Rojo");
      String valor2=mapa.get("red");
```

Nota: Existe una diferencia muy importante respecto de las nuevas clases. Cuando se diseñó la estructura de colecciones, se decidió que la sincronización de métodos debe ser responsabilidad del programador, por lo tanto, si se quiere una versión “thread safe”, se deben sincronizar los métodos a utilizar, contrariamente a las clases históricas cuyos métodos están sincronizados.

Diferencias entre Hashtable y HashMap

Ambas clases implementan la interfaz Map y funcionan de manera muy similar puesto que utilizan los métodos equals y hashCode de la clase llave para obtener la ubicación del objeto del valor en su estructura de datos interna.
La gran diferencia entre ambas clases es que los métodos de Hashtable están todos sincronizadas mientras que las operaciones de HashMap no lo están. Los métodos de Hashtable, al ser sincronizados, tienen un golpe de rendimiento en cada llamada puesto que internamente la JVM debe utilizar un mecanismo de bloqueo para la sincronización de la operación en la ejecución del hilo o hilos donde se comparte la instancia de Hashtable. HashMap, por el contrario, no sufre de este golpe de rendimiento puesto que sus operaciones no son sincronizadas. Cabe resaltar que HashMap no es una clase que se deba utilizar para operaciones en ambiente multi hilos, pero Hashtable **tampoco**. Tal como se recomienda en la misma documentación de Hashtable , es mejor utilizar ConcurrentHashMap.

```
//Encabesado de ConcurrentHashMap.
```

```
public class ConcurrentHashMap<K extends Object, V extends Object> extends
AbstractMap<K, V> implements ConcurrentMap<K, V>, Serializable {

	public V putIfAbsent(K k, V v) {
		// compiled code
	}
	public V get(Object o) {
		// compiled code
	}
}


	//Ejemplo de uso de la interfaz ConcurrentMap y clase ConcurrenHahMap.
	ConcurrentMap<String, String> mapaConcurrente = new ConcurrentHashMap();
	mapaConcurrente.putIfAbsent("red","Rojo");
	String valor3=mapaConcurrente.get("red");
```

Interface Collections

Las **colecciones sincronizadas** son aquellas en que **múltiples hilos pueden acceder** pero en realidad los **accesos no son simultáneos**. **Solo un hilo** lee o escribe en un **instante** dado. Como por ejemplo los ***Hashtable***.

Las **colecciones no sincronizadas pueden corromperse irreparablemente** si **más** de **un hilo** accede a ella. Sin embargo, **pueden convertirse en sincronizadas** utilizando ***wrappers***:

- *`Collections.synchronizedSet`*`( … );`
- *`Collections.synchronizedList`*`( … );`
- *`Collections.synchronizedMap`*`( … );`

Ejemplos de uso General

```
	//Ejemplo de uso de Collections actuales

	//List Permite Duplicados
	List<String> list;

	//Implementación ArrayList (usa una lista vector) no Safe Thread
	list=new ArrayList();
	list.add("Lunes");
	list.add("Martes");
	list.add("Martes");
	list.add("Miércoles");
	list.add("Jueves");
	list.add("Viernes");
	System.out.println("-------------------------------------------------");
	list.forEach(System.out::println);

	//Implementación LinkedList (usa una lista enlazada) no Safe Thread
	list=new LinkedList();
	list.add("Lunes");
	list.add("Martes");
	list.add("Martes");
	list.add("Miércoles");
	list.add("Jueves");
	list.add("Viernes");
	System.out.println("-------------------------------------------------");
	list.forEach(System.out::println);
```

```
 //Implementación ArrayList (usa una lista vector) Safe Thread
list=Collections.synchronizedList(new ArrayList());
list.add("Lunes");
list.add("Martes");
list.add("Martes");
list.add("Miércoles");
list.add("Jueves");
list.add("Viernes");
System.out.println("-------------------------------------------------");
list.forEach(System.out::println);

//Implementación LinkedList (usa una lista enlazada) Safe Thread
list=Collections.synchronizedList(new LinkedList());;
list.add("Lunes");
list.add("Martes");
list.add("Martes");
list.add("Miércoles");
list.add("Jueves");
list.add("Viernes");
System.out.println("-------------------------------------------------");
list.forEach(System.out::println);

//Set no permite duplicados.
Set<String> set;

//Implementacion HashSet no Safe Thread
//La más veloz sin orden
set=new HashSet();
set.add("Lunes");
set.add("Martes");
set.add("Martes");
set.add("Miércoles");
set.add("Jueves");
set.add("Viernes");
System.out.println("-------------------------------------------------");
set.forEach(System.out::println);

//Implementacion TreeSet no Safe Thread
//Almacena en un arbol por orden natural
set=new TreeSet();
set.add("Lunes");
set.add("Martes");
set.add("Martes");
set.add("Miércoles");
set.add("Jueves");
set.add("Viernes");
System.out.println("-------------------------------------------------");
set.forEach(System.out::println);

//Implementacion LinkedHashSet no Safe Thread
//Almacena en una lista enzada por orden de ingreso
set=new LinkedHashSet();
set.add("Lunes");
set.add("Martes");
set.add("Martes");
set.add("Miércoles");
set.add("Jueves");
set.add("Viernes");
System.out.println("-------------------------------------------------");
set.forEach(System.out::println);


//Implementacion HashSet Safe Thread
```

```
//La más veloz sin orden
set=Collections.synchronizedSet(new HashSet());
set.add("Lunes");
set.add("Martes");
set.add("Martes");
set.add("Miércoles");
set.add("Jueves");
set.add("Viernes");
System.out.println("-------------------------------------------------");
set.forEach(System.out::println);

//Implementacion TreeSet Safe Thread
//Almacena en un arbol por orden natural
set=Collections.synchronizedSet(new TreeSet());
set.add("Lunes");
set.add("Martes");
set.add("Martes");
set.add("Miércoles");
set.add("Jueves");
set.add("Viernes");
System.out.println("-------------------------------------------------");
set.forEach(System.out::println);

//Implementacion LinkedHashSet Safe Thread
//Almacena en una lista enzada por orden de ingreso
set=Collections.synchronizedSet(new LinkedHashSet());
set.add("Lunes");
set.add("Martes");
set.add("Martes");
set.add("Miércoles");
set.add("Jueves");
set.add("Viernes");
System.out.println("-------------------------------------------------");
set.forEach(System.out::println);

//Map
Map<String,String> map;

//Implementación LinkedHashMap no Safe Thread
//Es la más veloz y desordenada
map=new LinkedHashMap();
map.put("lunes", "monday");
map.put("martes", "tuesday");
map.put("miércoles", "wednesday");
map.put("jueves", "thursday");
map.put("viernes", "friday");
System.out.println("-------------------------------------------------");
map.forEach((k,v)->System.out.println(k+" - "+v));

//Implementación TreeMap no Safe Thread
//Almacena en un arbol por orden natural
map=new TreeMap();
map.put("lunes", "monday");
map.put("martes", "tuesday");
map.put("miércoles", "wednesday");
map.put("jueves", "thursday");
map.put("viernes", "friday");
System.out.println("-------------------------------------------------");
map.forEach((k,v)->System.out.println(k+" - "+v));

//Implementación HashMap no Safe Thread
//Almacena en una lista enlazada por orden de ingreso
map=new HashMap();
map.put("lunes", "monday");
```

```
map.put("martes", "tuesday");
map.put("miércoles", "wednesday");
map.put("jueves", "thursday");
map.put("viernes", "friday");
System.out.println("-------------------------------------------------");
map.forEach((k,v)->System.out.println(k+" - "+v));

//Implementación LinkedHashMap Safe Thread
map=new LinkedHashMap();
map.put("lunes", "monday");
map.put("martes", "tuesday");
map.put("miércoles", "wednesday");
map.put("jueves", "thursday");
map.put("viernes", "friday");
System.out.println("-------------------------------------------------");
map.forEach((k,v)->System.out.println(k+" - "+v));

//Implementación TreeMap Safe Thread
map=Collections.synchronizedMap(new TreeMap());
map.put("lunes", "monday");
map.put("martes", "tuesday");
map.put("miércoles", "wednesday");
map.put("jueves", "thursday");
map.put("viernes", "friday");
System.out.println("-------------------------------------------------");
map.forEach((k,v)->System.out.println(k+" - "+v));

//Implementación HashMap Safe Thread
map=Collections.synchronizedMap(new HashMap());
map.put("lunes", "monday");
map.put("martes", "tuesday");
map.put("miércoles", "wednesday");
map.put("jueves", "thursday");
map.put("viernes", "friday");
System.out.println("-------------------------------------------------");
map.forEach((k,v)->System.out.println(k+" - "+v));
```

Otras utilidades de Collections

```
//Método sort() ordena la collection
List<Integer> numeros = Arrays.asList(9,7,5,3);
Collections.sort(numeros);
numeros.forEach(System.out::println);

//Método reverse() invierte la collection
Collections.reverse(numeros);
numeros.forEach(System.out::println);


//Método binarySearch() busca un elemento en la lista.
List<Integer> numeros = Arrays.asList(9,7,5,3);
System.out.println(Collections.binarySearch(numeros, 9)); // 0
System.out.println(Collections.binarySearch(numeros, 3)); // 3
System.out.println(Collections.binarySearch(numeros, 2)); // -1
```

Ordenamiento de colecciones

Los métodos de Object y sus consecuencias en las colecciones.

Mientras que el lenguaje Java no proporciona soporte directo a los vectores asociativos (aquellos

que pueden tomar cualquier objeto como un índice), la presencia del método hashCode() en la clase Object anticipa claramente el uso de HashMap (y su predecesor, Hashtable) para resolver problemas que involucran claves y valores asociados. En condiciones ideales, los contenedores basados en hash ofrecen tanto inserción como búsqueda eficaz. Soportar hash directamente en el modelo de objetos facilita el desarrollo y el uso de contenedores basados en este algoritmo.
Dos objetos Integer sólo son iguales si contienen el mismo valor entero. Esto, junto con el hecho que sea inmutable, determina que sea práctico utilizar un número entero como clave en un HashMap . Este enfoque basado en el valor almacenado para determinar la igualdad es utilizado por todas las clases de envoltorio primitivas en las clases del marco de trabajo de Java, tales como Integer , Float , Character y Boolean , así como también lo hace String (dos objetos de tipo cadena son iguales si contienen la misma secuencia de caracteres). Debido a que estas clases son inmutables e implementan los métodos hashCode() y equals() con sensatez, todas ellas son buenas claves hash (esto tiene suma importancia al ordenar objetos en base a otros que actúen de clave).
¿Qué pasaría si Integer no rescribiera equals() y hashCode() ? Nada, si nunca se usa un entero como una clave en un HashMap o de otro tipo de colección basada en hash. Sin embargo, si se tuviera que utilizar un objeto entero como una clave en un HashMap , este no se sería capaz de recuperar con fiabilidad el valor asociado correspondiente, a menos que se utilice exactamente la misma instancia de Integer en la llamada a get() como con la que se invocó a put() . Para ello sería necesario asegurarse de que sólo se usa una sola instancia del objeto del tipo Integer correspondiente a un determinado valor entero en todo el programa. No es necesario decir que este enfoque sería incómodo y propenso a errores.
El contrato de interfaz para el objeto requiere que si dos objetos son iguales de acuerdo a equals() , entonces ellos deben tener el mismo valor de hashCode() . ¿Por qué la clase Object necesita hashCode() , cuando su capacidad de discriminar totalmente es equals() ? El método hashCode() existe exclusivamente por motivos de eficiencia. Los arquitectos de la plataforma Java anticiparon la importancia de las clases de colección basadas en algoritmos hash, como Hashtable , HashMap y HashSet , en las aplicaciones típicas de Java, y comparar entre sí muchos objetos con equals() puede ser costoso. Tener todos los objetos de Java como soporte a hashCode() permite el almacenamiento y la recuperación eficiente utilizando colecciones basadas en algoritmos hash.
La especificación de Object ofrece una directriz vaga acerca de equals() y hashCode() para que sea consistente, "sus resultados serán los mismos para las subsiguientes invocaciones”, dado que “ninguna información utilizada en la comparación del objeto mediante equals() se modificará". Esto suena algo así como "el resultado del cálculo no debe cambiar, a menos que lo haga". Esta declaración vaga generalmente se interpreta en el sentido de que los cálculos de la igualdad y el valor hash deben ser una función determinista del estado de un objeto y nada más.

¿Qué es igualdad?

Los requisitos para equals y hashCode impuestos por la especificación de la clase Object son bastante fáciles de seguir. Decidir si, y cómo, sobrescribir equals requiere un poco más de criterio. En el caso de clases de valores inmutables simples, como Integer (y de hecho, para casi todas las clases inmutables), la elección es bastante obvia - la igualdad debe basarse en la equivalencia del estado del objeto subyacente (el valor en su interior que determina su estado). En el caso de Integer , el único estado del objeto es el valor entero que almacena en su interior, por lo tanto, el subyacente.
Para los objetos que pueden mutar, la respuesta no es siempre tan clara. ¿Se debe basar la igualdad de un objeto en la igualdad de referencias como en la implementación por defecto o se debe basar en el estado del objeto (como Integer y String)? No hay una respuesta fácil – depende del uso previsto de la clase. Para los contenedores como List y Map , se podría haber hecho un argumento razonable de cualquiera. La mayoría de las clases en las clases del marco de trabajo de Java, incluyendo clases contenedoras, sólo ofrecen un equals y hashCode basados en Object.
Si el valor hashCode de un objeto puede cambiar en función de su estado, entonces hay que tener

cuidado al usar tales objetos como claves en las colecciones basadas en hash para asegurarse de no permitir que su estado cambie cuando se utilizan como claves hash. Todas las colecciones basadas en hash suponen que el valor calculado de hash para un objeto no cambia mientras está en uso como clave de una colección. Si el código hash de la clave fuera a cambiar mientras se encuentra en una colección, podrían surgir consecuencias imprevisibles y confusas. Esto no suele ser un problema en la práctica - no es un uso común el de un objeto mutable, como una lista, para que sea una clave en un HashMap .
Para las clases más complejas, el comportamiento de equals y hashCode , pueden, incluso, ser impuestas por la especificación de una superclase o interfaz. Por ejemplo, List requiere que un objeto del tipo List es igual a otro objeto del mismo tipo si y sólo si el otro también es una lista y contiene los mismos elementos (definido por el Object.equals en dichos elementos) en el mismo orden para ambos. Los requisitos para hashCode se definen aún más específicos.

```
	//Calculo del hashCode de una lista
	hashCode = 1;
	Iterator i = list.iterator();
	while (i.hasNext()) {
		Object obj = i.next();
		hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
	}
```

No sólo es el valor de hash depende de los contenidos de la lista, sino que se establece también el algoritmo específico para combinar los valores hash de los elementos individuales. (La clase String especifica un algoritmo similar para ser utilizado para calcular el valor hash de una cadena).

Escribiendo métodos equals y hashCode propios

Remplazar el valor por defecto de equals es bastante fácil, pero remplazar un método equals que ya se ha sobrescrito puede ser extremadamente difícil de hacer sin violar la simetría o la exigencia de la transitividad. Al remplazar equals , siempre se debe incluir algunos comentarios en el Javadoc sobre éste para ayudar a aquellos que quieran extender la clase, para que lo hagan correctamente.

```
public class A {
	final B valorNoNulo = new B();
	C otrovalorNoNulo = new C();
	int valorQueNoDeterminaEstado;
	public boolean equals(Object otro) {
		if (this == otro) return true;
		if (!(otro instanceof A)) return false;
		A otroA = (A) otro;
		return (valorNoNulo.equals(otroA.valorNoNulo))
		&& ((otrovalorNoNulo == null)? otroA.otrovalorNoNulo == null
			: otrovalorNoNulo.equals(otroA.otrovalorNoNulo));
	}
	public int hashCode() {
		int hash = 1;
		hash = hash * 31 + valorNoNulo.hashCode();
		hash = hash * 31 + (otrovalorNoNulo == null ? 0 :
					otrovalorNoNulo.hashCode());
		return hash;
	}
}
```

Nótese que ambas implementaciones delegan una porción del cálculo de equals o hashCode en los campos de estado de la clase (sus variables de instancia). Dependiendo de la clase, es posible que

también delegar una parte del cálculo de equals o hashCode a la superclase. Para los campos de estado de tipo base, hay funciones de ayuda en las clases de envoltorios asociadas que pueden asistir en la creación de valores de hash, como Float.floatToIntBits .

Posibles mejoras

La construcción de hash en la clase Object de las clases del marco de trabajo de Java era un compromiso de diseño muy sensible - se facilita el uso de contenedores basados en hash que es mucho más fácil y eficiente. Sin embargo, varios han hecho críticas del enfoque y la aplicación de algoritmos hash y la igualdad en dichas clases Java. Los contenedores de hash basados en java.util son muy convenientes y fáciles de usar, pero puede no ser adecuado para aplicaciones que requieren un rendimiento muy alto. Mientras que la mayoría de ellos nunca será cambiada, vale la pena tener en cuenta al diseñar las aplicaciones que dependen en gran medida de la eficiencia de los contenedores basados en hash. Estas críticas incluyen:

- Rango muy chico para el uso de hash. El uso de int , en lugar de long , para el tipo de retorno de hashCode aumenta la posibilidad de colisiones de hash.
- La mala distribución de los valores hash. Los valores hash para cadenas cortas y enteros pequeños en sí son números enteros pequeños, y están cerca los valores hash unos de otros. Una función más adecuada hash sería distribuir los valores hash de manera más uniforme en toda el rango de hash posible.
- No se definen las operaciones de hash. Mientras que algunas clases, como String y List , definen un algoritmo de hash que se utiliza en combinación con los valores hash de sus elementos constitutivos como un valor hash único, la especificación del lenguaje no define ningún medio aprobado de la combinación de los valores hash de varios objetos en un nuevo valor de hash. El truco utilizado por List , String , o la clase de ejemplo que se discutió anteriormente es simple, pero muy lejos de las matemáticas ideales. Tampoco las implementaciones de biblioteca de clases ofrecen cualquier algoritmo conveniente de hash que simplifique la creación de implementaciones más sofisticadas de hashCode .
- Dificultad para escribir equals cuando se extiende una clase instanciable que ya rescribió equals . Las maneras "obvias" de definir equals cuando se hereda de una clase concreta que ya rescribió este método, fallan todas al no cumplir en general con los requisitos de la simetría o la transitividad que exige el método. Esto significa que se debe entender los detalles de la estructura y la implementación de clases que se extienden al rescribir equals , e inclusive se puede tener que exponer a los campos privados de la clase base como protegidos para hacerlo, lo cual viola los principios de buen diseño orientado a objetos.

Uso de Interfaces para hacer comparaciones

Las interfaces Comparable y Comparator resultan útiles para ordenar colecciones. La interfaz Comparable define un orden natural para las clases que la implementan. La interfaz Comparator se emplea para especificar la relación de orden. También permite anular el orden natural. Estas interfaces resultan útiles para ordenar los elementos de una colección.

La interfaz Comparable

La interfaz Comparable pertenece al paquete java.lang . Cuando se declara una clase, la implementación de JVM no tiene manera de determinar el orden que debe aplicarse a los objetos de la clase.

La interfaz Comparable permite definir el orden de los objetos de cualquier clase. Es posible ordenar las colecciones que contienen objetos de clases que implementan la interfaz Comparable . Algunos ejemplos de clases de Java que implementan la interfaz Comparable son Byte , Long , String , Date y Float . Las clases que representan números emplean una implementación numérica, la clase String utiliza una implementación alfabética y la clase Date emplea una implementación cronológica. Al pasar una lista de tipo ArrayList que contiene elementos de tipo String al método estático sort de la clase Collections , se genera una lista en orden alfabético. Una lista que contiene elementos de tipo Date se clasificará por orden cronológico, mientras que una lista con elementos de tipo Integer se clasificará por orden numérico.
Para escribir tipos Comparable personalizados, es necesario implementar el método compareTo de la interfaz Comparable.

```
//Código de la Interface Comparable
public interface Comparable<T extends Object> {
    public int compareTo(T t);
}

//Ejemplo de uso de Comparable
public class Horario implements Comparable<Horario> {
      private int dia = 0;
      private int horaComienzo = 0;
      private int minutosComienzo = 0;
      private int horaFin = 0;
      private int minutosFin = 0;
      private int turnosPorHora = 0;

      public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + dia;
            result = prime * result + horaComienzo;
            result = prime * result + horaFin;
            result = prime * result + minutosComienzo;
            result = prime * result + minutosFin;
            result = prime * result + turnosPorHora;
            return result;
      }
      public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null) return false;
            if (getClass() != obj.getClass()) return false;
            Horario other = (Horario) obj;
            if (dia != other.dia) return false;
            if (horaComienzo != other.horaComienzo) return false;
            if (horaFin != other.horaFin) return false;
            if (minutosComienzo != other.minutosComienzo) return false;
            if (minutosFin != other.minutosFin) return false;
            if (turnosPorHora != other.turnosPorHora) return false;
            return true;
      }
      public int compareTo(Horario o) {
            if (dia > o.dia || horaComienzo > o.horaComienzo
                  || minutosComienzo > o.minutosComienzo || horaFin > o.horaFin
                  || minutosFin > o.minutosFin) return 1;
            else if (dia < o.dia || horaComienzo < o.horaComienzo
                  || minutosComienzo < o.minutosComienzo || horaFin < o.horaFin
                  || minutosFin < o.minutosFin)
                  return -1;
            else return 0;
      }
}
```

```
//Ejemplo de uso
TreeSet conjuntoDeHorarios = new TreeSet();
conjuntoDeHorarios.add(new Horario(2, 10, 30, 15, 0, 4));
conjuntoDeHorarios.add(new Horario(2, 8, 0, 12, 0, 4));
conjuntoDeHorarios.add(new Horario(3, 9, 0, 13, 0, 4));
conjuntoDeHorarios.add(new Horario(3, 8, 0, 13, 0, 3));
Object[] estudianteArray = conjuntoDeHorarios.toArray();
Horario s;
for (Object obj : estudianteArray) {
      s = (Horario) obj;
      System. out .printf("Día = %s Comienza a las %s:%s y termina a las
            %s:%s\n", s.getDia(), s.getHoraComienzo(),
            s.getMinutosComienzo(), s.getHoraFin(), s.getMinutosFin());
}

//Salida por consola del programa
Día = 2 Comienza a las 8:0 y termina a las 12:0
Día = 2 Comienza a las 10:30 y termina a las 15:0
Día = 3 Comienza a las 8:0 y termina a las 13:0
Día = 3 Comienza a las 9:0 y termina a las 13:0
```

El ejemplo ordena los objetos del tipo Horario por día, hora de comienzo, minutos de comienzo, hora de fin, y minutos de fin. Se debe notar que la variable de instancia que almacena la cantidad de turnos no se la considera significativa para ordenar los horarios. Esta es una decisión de diseño que establece el “orden natural” de los objetos de tipo Horario .
Observe que los horarios se comparan según sus variables de instancia, que definen su estado. Esto ocurre porque al ordenar los elementos de la colección TreeSet , la cual comprueba si los objetos se encuentran ordenados usando el método compareTo para compararlos.
Algunas colecciones, como TreeSet , se ordenan. La implementación de TreeSet necesita saber cómo ordenar los elementos. Si los elementos tienen un orden natural, TreeSet emplea el orden natural. En caso contrario, es necesario ayudarle. Por ejemplo, la clase TreeSet incluye el siguiente constructor, que admite la interfaz Comparator como parámetro.

```
TreeSet(Comparator comparator)
```

Este constructor crea un conjunto de árbol vacío, ordenado según el parámetro Comparator especificado. La siguiente sección ofrece una explicación detallada del uso de la interfaz Comparator.

La interfaz Comparator

La interfaz Comparator ofrece una mayor flexibilidad a la hora de ordenar. Por ejemplo, en el caso de la clase Horario descrita anteriormente, los horarios sólo se ordenaron mediante una comparación de sus días, horas y minutos en los cuales están comprendidos los objetos de su tipo. No se ordenó de ninguna otra forma ni se usó algún otro criterio. Esta sección muestra cómo aumentar la flexibilidad de la ordenación usando la interfaz Comparator .
La interfaz Comparator forma parte del paquete java.util . Se utiliza para comparar los objetos según el orden personalizado (cuales atributos se usan para la comparación) en lugar del orden natural (que se suele implementar con Comparable). Por ejemplo, permite ordenar los objetos según un orden distinto del orden natural.
También se emplea para ordenar objetos que no implementan la interfaz Comparable .
Para escribir un comparador Comparator personalizado, es necesario agregar una implementación del método compare de esta interfaz en la clase concreta:

```
int compare(Object o1, Object o2)
```

Este método compara los dos argumentos y devuelve un entero negativo si el primer argumento es menor que el segundo. Si ambos son iguales, devuelve un cero y si el primer argumento es mayor que el segundo, devuelve un entero positivo, como lo hace la interfaz Comparable.
El en caso supuesto que se quiera generar una forma de comparar los mismos horarios pero sólo teniendo en cuenta las horas, pero nada más, habría que cambiar la implementación de Comparable. En lugar de hacer esto se crea una clase que manejará la comparación a través de implementar Comparator.

```
//Ejemplo de uso de Comparator
public class ComparaHorasHorario implements Comparator {
      public int compare(Object o1, Object o2) {
            if(((Horario)o1).getHoraComienzo()==((Horario)o2).getHoraComienzo())
                  if (((Horario) o1).getHoraFin()==((Horario) o2).getHoraFin())
                        return 0;
                  else if(((Horario)o1).getHoraFin()<((Horario)o2).getHoraFin())
                        return 1;
                  else
                        return -1;
            else if(((Horario)o1).getHoraComienzo()  <
                        ((Horario)o2).getHoraComienzo())
                  return -1;
            else
                  return 1;
      }
}
```

Es posible crear varias clases para comparar los horarios. Por ejemplo, se puede volver a utilizar la interfaz para crear otro criterio de comparación, en este caso, por horas y minutos, lo cual cambiaría el criterio para ordenar los elementos internamente.
Se puede concluir entonces que se pueden hacer tantos ordenamientos como se quiera para luego aplicarlos en los parámetros de construcción de la colección y así determinar que el ordenamiento se haga según lo especificado en estas clases.
Los efectos de utilizar diferentes criterios de ordenamiento se pueden ver inmediatamente si se crea una colección que admita a Comparator como argumento en el constructor (lo cual indica el criterio de ordenamiento que utilizará la colección). Por ejemplo, si se utilizan las dos clases que se mostraron anteriormente como ejemplos, se obtendría dos formas diferentes de ordenar dicha colección.

```
ComparaHorasHorario ch = new ComparaHorasHorario();
ComparaHorasYMinutosHorario cm = new ComparaHorasYMinutosHorario();
TreeSet horarioSetPorHora = new TreeSet(ch);
TreeSet horarioSetPorMinuto = new TreeSet(cm);
```

La palabra clave final en una collection

Cuando definimos una collections con el modificador final, la misma no puede redefinirse pero si se puede cambiar su contenido, se pueden agregar y quitar elementos de la misma.

```
//Colleciones con el modificador final.
final List<String> lista=new ArrayList();
lista.add("hola");
lista.forEach(System.out::println);
//lista=new ArrayList();                    //Esta linea arroja error.
```

Genéricos y colecciones

Las clases de las colecciones tradicionales utilizan el tipo Object para admitir contener diferentes tipos de entradas y salidas. Es necesario convertir el tipo de objeto explícitamente para poder recuperarlo y esto no garantiza la seguridad del tipo.
Aunque la infraestructura de colecciones existente admite las colecciones homogéneas (es decir, colecciones con un tipo de objeto específico, por ejemplo Date), no había ningún mecanismo para evitar la inserción de otros tipos de objetos en la colección. Para poder recuperar un objeto, casi siempre había que convertirlo.
Este problema se ha resuelto con la funcionalidad de los genéricos. Dicha funcionalidad se ha introducido a partir de la plataforma Java SE 5.0. Para ello, se proporciona información para el compilador sobre el tipo de colección utilizado mediante el parámetro genérico con el que se va a utilizar la colección. Así, la comprobación del tipo se resuelve de forma automática antes del tiempo de ejecución. Esto elimina la conversión explícita de los tipos de datos para su uso en la colección durante la ejecución del programa. Gracias al autoboxing de los tipos primitivos, es posible utilizar tipos genéricos para escribir código más sencillo y comprensible. Sin embargo, se debe evitar en lo posible usar autoboxing porque es costoso a nivel de recursos cada vez que convierte un objeto o lo hace seleccionable para el recolector de basura.

```
//Código si usar Generics
ArrayList lista = new ArrayList();
lista.add(0, 42);
int total = ((Integer)lista.get(0)).intValue();
```

En este código, se necesita una clase de envoltorio Integer para la conversión del tipo mientras se recupera el valor del número entero que existe en lista. En el momento de la ejecución, el programa debe controlar el tipo para lista.
Al aplicar los tipos genéricos, ArrayList debe declararse como ArrayList<Integer> , para informar al compilador sobre el tipo de colección que se va a utilizar. Al recuperar el valor, no se necesita una clase envoltorio Integer .

```
//Evitando la conversión de tipo al utilizar genéricos
ArrayList<Integer> list = new ArrayList();
list.add(0, 42);
int total = list.get(0).intValue();
```

La función de autoboxing encaja muy bien con el API de tipos genéricos. Con autoboxing.

Nota: *Los tipos genéricos se habilitan de forma predeterminada a partir de la plataforma Java SE 5.0.*
Los tipos genéricos son ideales para utilizar en cualquier clase que no tenga dependencia con su estado para prestar sus servicios. Todos los algoritmos fundamentales son candidatos ideales para esto.

La API de colecciones genéricas

Las colecciones tradicionales fueron actualizadas a partir de Java 1.5 para utilizar genéricos. Los parámetros se definen para las nuevas colecciones según la siguiente convención:

E - Elemento (usado extensamente por el Framework de colecciones de Java)
K - Clave
N - Número
T - Tipo
V - Valor

S,U,V etc. – tipos 2o, 3o, 4o…

Cuando se trata de interfaces se suele indicar el uso del genérico dentro de las declaraciones. En cambio, cuando se trata de clases concretas, se suele indicar resaltando en un rectángulo superpuesto el parámetro a recibir.

Invariancia con genericos
Debido a que los tipos están asegurados, la pregunta siguiente es ¿qué pasa cuando los tipos son subtipos de los parámetros genéricos en la declaración de una colección?

```
List<A> la;
List<C> lc = new ArrayList<C>();
List<D> ld = new ArrayList<D>();
//si lo siguiente fuese posible...
la = lc;
la.add(new C());
//lo siguiente también debería ser posible...
la = ld;
la.add(new D());
//por tanto...
C sa = ld.get(0); //¡No!!
```

La asignación incorrecta en este caso es la = lc , porque como se mencionó anteriormente, si bien C y D son subclases de A , ArrayList<C> y ArrayList<D> no son subclases de List<A> . A lo sumo se puede afirmar que, por el principio de invariancia antes mencionado:

ArrayList<A> es subclase de List<A>
ArrayList<C> es subclase de List<C>
ArrayList<D> es subclase de List<D>

Para que la garantía de seguridad de tipo siempre sea válida, debe ser imposible asignar una colección de un tipo a una colección de un tipo distinto, incluso si el segundo tipo es una subclase del primero.
Esto va en contra del polimorfismo tradicional y, a primera vista, parece restar flexibilidad a las colecciones genéricas.

Covarianza

Los comodines ofrecen una cierta flexibilidad al trabajar con colecciones genéricas.

```
public static void main(String[] args) {
      List<C> lc = new ArrayList<C>();
      List<D> ld = new ArrayList<D>();
      imprimeNombreClase (lc);
      imprimeNombreClase (ld);
      // pero...
      List<? extends Object> lo = lc; // Bien
      lo.add(new C());//¡Error de compilación!
}
public static void imprimeNombreClase(List<? extends A> la) {
      for (int i = 0; i < la.size(); i++) {
            System. out .println(la.get(i).getClass());
      }
}
```

El método imprimeNombreClase se declara con un argumento que incluye un comodín. El comodín " ? " de List <? extends A> podría interpretarse como "cualquier tipo de lista de elementos

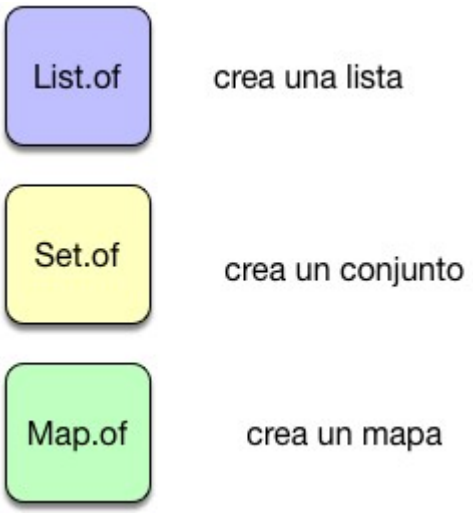
desconocidos que sean de tipo A o de una subclase de A ". El límite superior (A) indica que los elementos de la colección pueden asignarse de forma segura a una variable A . Por tanto, las dos colecciones de los subtipos A pueden pasarse al método imprimeNombreClase .
Esta respuesta de la covarianza está designada para su lectura, no para que se escriba en ella debido a que el tipo puede ser convertido en una asignación para acceder, pero nunca para escribir porque puede suceder una asignación incorrecta, como por ejemplo, un súper tipo a un subtipo. Debido al principio de la invariancia, es ilegal agregar a una colección que contenga un comodín con la palabra clave extends .

Genéricos: refactorización de código no genérico existente

Con las colecciones genéricas, es posible especificar tipos genéricos sin argumentos de tipo, denominados tipos básicos. Esta característica garantiza la compatibilidad con el código no genérico.
En el momento de la compilación, se elimina toda la información genérica del código genérico. Lo que queda es un tipo básico. Esto permite la interoperabilidad con el código tradicional, ya que los archivos de clase generados por el código genérico y por el código tradicional serían los mismos.
Durante la ejecución, ArrayList<String> y ArrayList<Integer> se traducen como ArrayList , lo cual es un tipo básico.
El uso del compilador a partir de Java SE 5.0 o posterior con código no genérico más antiguo generará una advertencia.

Java 9 Collections y sus novedades

Las Novedades e**n Java 9 Collections son muchas**. Java 8 introdujo las expresiones Lambda y los Streams que fueron un salto muy importante en Java. Sin embargo como una tecnología nuev**a quedaron muchas cosas por añadir y fue únicamente un primer paso.** Java 9 hace uso del framework de colecciones **de una forma mucho más natural.** Hoy vamos a hablar de los nuevos métodos estáticos **of** que soportan las diferentes colecciones y que hacían mucha falta para simplificar la creación de objetos.



Java 9 Collections y List.of

El primer método que vamos a abordar es el método List.of que nos permite de una forma sencilla crear una lista de elementos.

```
package com.arquitecturajava;

import java.util.List;
import java.util.Map;
import java.util.Set;

public class Principal {
	public static void main(String[] args) {
		List<String> lista=List.of("hola","que","tal","estas");
		lista.forEach(System.out::println);
	}
}
```

Era algo que hacía mucha falta ya que anteriormente había que usar el método de la clase Arrays.toList para conseguir algo similar. El resultado se muestra en la consola.

```
<terminated> Principal (7) [Java
hola
que
tal
estas
```

Java 9 Collections y Set.of

De igual manera podemos trabajar con los Set

```
Set<Integer> lista2=Set.of(1,2,3,4,5,6);

lista2.forEach(System.out::println);
```

Que nos imprimirá la lista de números.

```
6
5
4
3
2
1
```

El resultado es es curioso , los elementos no salen en orden . Recordemos que **la implementación por defecto de un Set es un HashSet** , una tabla Hash. Estas estructuras no aseguran un orden.

Java 9 Collections y Map.of

Por último aunque desde mi punto de vista el más interesante , tenemos el método of a nivel de los mapas que nos permite crear un mapa de elementos de una forma muy directa.

```
Map<Integer,String> mapa=Map.of(1,"cecilio",2,"antonio",3,"gema");

System.out.println(mapa.get(1));
```

En este caso acabamos de crear un mapa que tiene como clave un número y como valor un String . Si solicitamos el número 1 nos imprimirá por la consola Cecilio.

```
cecilio
```

API Stream

Introducción.

En español la palabra Stream quiere decir **flujo** o **secuencia** y según la documentación oficial de Java, la interfaz Stream representa una secuencia de elementos, pero ¿qué clase de elementos?, elementos del tipo que indiquemos. La interfaz Stream es genérica, por lo tanto para recuperar un stream de nuestros productos podemos hacer esto.

```
class Persona{
      private int id;
      private String nombre;
      private int edad;
      public Persona(int id, String nombre, int edad) {
          this.id = id;
          this.nombre = nombre;
          this.edad = edad;
      }
      @Override public String toString() {
          return "Persona{" + "id=" + id + ", nombre=" + nombre + ",
          edad=" + edad + '}';
      }
      public int getId()          { return id; }
      public String getNombre()   { return nombre; }
      public int getEdad()        { return edad; }
  }

  Stream<Persona> personasStream = List.of(
          new Persona(1,"Ana",32),
          new Persona(2,"Javier",41),
          new Persona(3,"Carlos",22),
          new Persona(4,"Estela",55),
          new Persona(5,"Raul",27)
  ).stream(); //flujo de elementos Persona creados a partir de un List
  personasStream.forEach(System.out::println); // imprime la lista de
            //personas
```

Nota: Es importante aclarar que Stream **implementa la interface AutoCloseable**, tema que trataremos en el capitulo de **Exceptions.**

Realizaremos algunas operaciones que aplicamos comúnmente a una base de datos relacional usando sql, pero las haremos sobre listas en Java usando Stream, por lo tanto a continuación te presento una tabla donde se muestran algunos métodos de la interfaz`Stream` junto a una posible equivalencia en sql.

SQL	**Interfaz Stream**
from	stream()
select	map()
where	filter() (antes de un collecting)
order by	sorted()
distinct	distinct()
having	filter() (después de un collecting)

SQL	Interfaz Stream
join	flatMap()
union	concat().distinct()
offset	skip()
limit	limit()
group by	collect(groupingBy())
count	count()

Consultas simples

Considera la siguiente consulta **sql:**

```
select nombre from personas;
```

El equivalente usando Java Streams es:

```
List<Persona> listaPersona=List.of(
        new Persona(1,"Ana",32),
        new Persona(2,"Javier",41),
        new Persona(3,"Carlos",22),
        new Persona(4,"Estela",55),
        new Persona(5,"Raul",27)
);

Stream<String> stream=listaPersona.stream().map(Persona::getNombre);
stream.forEach(System.out::println);

//en forma verbosa o abreviada:
     listaPersona
           .stream()
           .map(Persona::getNombre)
           .forEach(System.out::println);
```

- Con el método `stream()` obtenemos una secuencia de elementos de tipo `Persona`. Este es el **from**.
- Con el método `map` recuperamos solo el atributo *name*. Este es el **select**.

El punto clave es obtener un Stream mediante el método `stream()` y a partir de ahí ejecutar las operaciones como filtrados, agrupaciones, etc.

Filtrado

Recuperar los nombres de personas que tengan una edad menor a 30 años.

En **sql:**

```
select name from personas where edad < 30;
```

Con Java **Stream:**

```
listaPersona
      .stream()
      .filter(p -> p.getEdad()<30)
```

```
            .map(Persona::getNombre)
            .forEach(System.out::println);
```

Es importante notar el órden en el que aparecen los métodos, primero se encuentra filter y después map. ¿Qué ocurre si colocamos primero a map y luego a filter?

```
      listaPersona
            .stream()
            .map(Persona::getNombre)
            .filter(p -> p.getEdad()<30)
            .forEach(System.out::println);
```

Obtendremos un error de compilación. ¿Por qué? Porque el método map devuelve el nombre de la persona que es un String y la clase String no tiene un método que se llame filter.

El método `filter()` recibe un predicado. Un predicado es solo una función que devuelve un valor boolean y la instrucción `p.getEdad()<30` es una expresión booleana.

Ordenación

Obtener los nombres de personas que tengan una edad menor a 30 años pero ordenados de forma ascendente por edad, es decir, de menor edad a mayor edad.

En **sql:**

```
      select name from personas where edad < 30 order by edad;
```

Con Java **Stream:**

```
              listaPersona
                .stream()
                .filter(p -> p.getEdad()<30)
                .sorted(Comparator.comparingInt(Persona::getEdad))
                .map(Persona::getNombre)
                .forEach(System.out::println);
```

El método `sorted` recibe un `Comparator`. Ésta misma interfaz `Comparator` tiene algunos métodos que nos serán de gran ayuda

- `comparingInt()` Permite comparar elementos de tipo int
- `comparingDouble()` Permite comparar elementos de tipo double
- `comparingLong()` Permite comparar elementos de tipo long
- `thenComparing()` Permite anidar comparaciones. Útil cuándo deseamos ordenar por más de 1 atributo (ejemplo más adelante)

Lo mejor será revisar la documentación de la interfaz Comparator

Si deseamos ordenar en forma descendente necesitamos aplicar un reverso,

```
        listaPersona
                .stream()
                .filter(p -> p.getEdad()<30)
                .sorted(Comparator.comparingInt(Persona::getEdad).reversed())
```

```
            .map(Persona::getNombre)
            .forEach(System.out::println);
```

Otra forma diferente de ordenar, es que nuestra clase Product implemente a la interfaz **`Comparable`**

```
class Persona implements Comparable<Persona>{
    private int id;
    private String nombre;
    private int edad;
    public Persona(int id, String nombre, int edad) {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
    }
    @Override public String toString() {
        return "Persona{" + "id=" + id + ", nombre=" + nombre + ",
                edad=" + edad + '}';
    }
    public int getId()                      { return id;            }
    public void setId(int id)               { this.id = id;         }
    public String getNombre()               { return nombre;        }
    public void setNombre(String nombre)    { this.nombre = nombre; }
    public int getEdad()                    { return edad;          }
    public void setEdad(int edad)           { this.edad = edad;     }
    @Override public int compareTo(Persona p) {
        //En este método escribimos la estrategia de ordenamiento
        //estandar
        return this.toString().compareTo(p.toString());
    }
}
```

Ahora al método sorted() es invocado sin argumentos:

```
listaPersona
        .stream()
        .filter(p -> p.getEdad()<30)
        .sorted()
        .map(Persona::getNombre)
        .forEach(System.out::println);
```

¿Cómo ordenamos en forma descendente? Usando Comparator.reverseOrder()

```
listaPersona
        .stream()
        .filter(p -> p.getEdad()<30)
        .sorted(Comparator.reverseOrder())
        .map(Persona::getNombre)
        .forEach(System.out::println);
```

Agrupado

En SQL las operaciones como sum, max, min, avg, group by, partition by, etc., se llaman funciones de agregado. En Java, se especifican en el método `collect`

Obtener la cantidad de personas agrupados por estadoCivil.

En **sql:**

```
select estadoCivil, count(*) cantidad from personas GROUP BY estadoCivil;
```

Con Java **Streams**:

```
            Map<String, Long> collect= listaPersona
                .stream()
                .collect( //en el metodo collect se especifican las funciones de
                             //agregacion
                        Collectors.groupingBy( // deseamos agrupar
                                Persona::getEstadoCivil, //agrupamos por estado
                                Collectors.counting()    // realizamos el conteo
                        )
                );

        collect.forEach((s, c) -> System.out.printf("Estado Civil: %s: cantidad:
%s \n", s,c));
```

en forma abreaviada y verbosa:

```
        listaPersona
                .stream()
                .collect(
                        Collectors.groupingBy(
                                Persona::getEstadoCivil, Collectors.counting())
                ).forEach((s, c) -> System.out.printf("Estado Civil: %s:
                                        cantidad: %s \n", s,c));
```

Dado que en el método collect especificamos funciones de agregado, casi siempre nos auxiliaremos de la clase Collectors la cuál nos proporciona varios métodos de funciones de agregado. En este ejemplo, usamos el método groupingBy

Si deseamos filtrar todas las productos que tengan menos de 30 años y agrupados por estado civil:

```
listaPersona
                .stream()
                .filter(p -> p.getEdad() < 30)
                .collect(
                        Collectors.groupingBy(
                                Persona::getEstadoCivil, Collectors.counting()
                        )
                )
                .forEach((s, c) -> System.out.printf("Estado Civil: %s:
                                    cantidad: %s \n", s,c));
```

Sumas

Obtener la suma de las edades agrupadas por estado civil:

```
        listaPersona
                .stream()
                .collect(
```

```
                        Collectors.groupingBy(
                                Persona::getEstadoCivil,
                                Collectors.summingInt(Persona::getEdad)
                        )
                )
                .forEach((s, c) -> System.out.printf("Estado Civil: %s:
cantidad: %s \n", s,c));
```

Having

Tomando el ejemplo anterior, le agregaremos algo más:

> Obtener la suma de las edades agrupadas por estado civil, pero solo obtener aquellos registros cuya suma sea mayor a 50.

```
 listaPersona
                .stream()
                .collect(
                        Collectors.groupingBy(
                                Persona::getEstadoCivil,
                                Collectors.summingInt(Persona::getEdad)
                        )
                )
                .entrySet()
                .stream() //volvemos a generar un stream
                .filter(p -> p.getValue() > 50) //filtramos (simula el having)
                .collect(Collectors.toList())
                .forEach(list -> System.out.printf("Estado Civil: %s: cantidad:
                        %s \n", list.getKey(),list.getValue()));
```

Más operaciones

Promedio de existencias en almacen:

```
      Double average = products.stream()
                .collect(Collectors.averagingInt(Product::getUnitsInStock));
      System.out.printf("Promedio de existencias en almacen: %s",average );

      // Promedio de existencias en almacen: 40.54545454545455
```

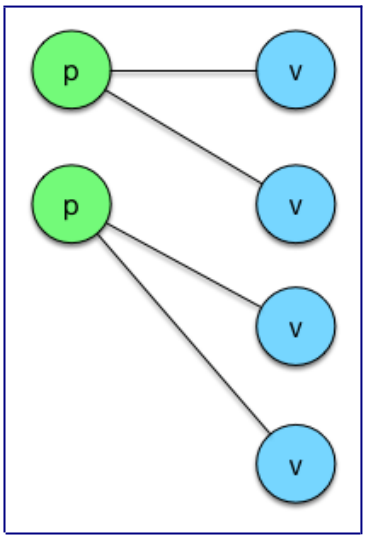
Producto con el precio unitario más alto

```
      Optional<Product> product =
      products.stream().max(Comparator.comparing(Product::getUnitPrice));
      System.out.println(product.get());

      // Product{id=38, name='Côte de Blaye', supplier=18, category=1,
      // unitPrice=263.5, unitsInStock=17}
```

Podemos obtener el **count, sum, min, max** y **average** con una sola operación. Por ejemplo si queremos obtener estas estadísticas respecto al precio unitario

```
      DoubleSummaryStatistics statistics =
      products
```




```
        .stream()
        .collect(Collectors.summarizingDouble(Product::getUnitPrice));
System.out.println(statistics);

// DoubleSummaryStatistics{count=77, sum=2222.710000, min=2.500000,
// average=28.866364, max=263.500000}
```

Limitar el numero de productos devueltos

```
products.stream().limit(50); // limitamos a 50 productos
```

Saltar hasta el elemento indicado y a partir de ahí devolver todos los elementos

```
Stream<Product> skip = products.stream().skip(5); //obtenemos los productos a
partir del 6 (inclusive)
skip.forEach(System.out::println);

// Product{id=6, name='Grandma's Boysenberry Spread', supplier=3, category=2...}
// Product{id=7, name='Uncle Bob's Organic Dried Pears', supplier=3,
category=7...}
// Product{id=8, name='Northwoods Cranberry Sauce', supplier=3, category=2...}
// …
```

Resumen

Hemos visto como a partir de Java 8 podemos ejecutar sobre colecciones potentes operaciones de agregación como en SQL. Para mejor comprensión debes escribir estos ejemplos tu mismo y ver los resultados. Si lo deseas incluso puedes cargar el archivo csv a una base de datos relacional para comprobar los resultados.

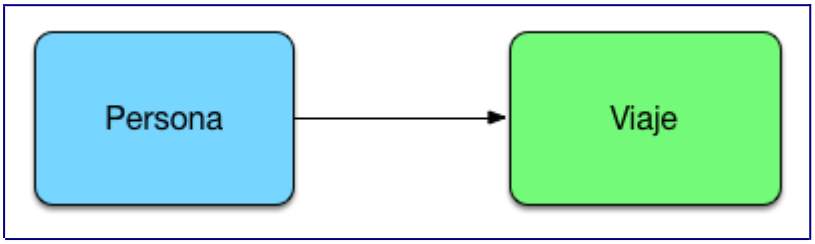
Podemos decir que,

- **`Stream`** Representa un flujo de elementos y podemos aplicar métodos tales como
  - **`filter`** para filtrar los elementos. Éste método recibe un predicado como argumento.
  - **`map`** para devolver solo el atributo indicado. Es como el select de sql.
  - **`sorted`** para ordenar nuestros elementos. Recibe un `Comparator`.
  - **`min`, `max`, `count`** que permiten obtener el minimo, máximo y conteo de elementos respectivamente.

    - **`collect`** aquí es donde definiremos nuestros funciones de agregado, principalmente agrupados, particiones, etc.
- **`Comparator`** nos proporciona métodos útiles para realizar ordenamientos. Lo usaremos en conjunto con el método **`sorted`**
- **`Collectors`** Nos proporciona métodos útiles para agrupar, sumar, promediar, obtener estadísticas. Lo usaremos en conjunto con el método **`collect`**

Java 8 FlatMap y Streams

El uso de **Java 8 FlatMap** es algo que en muchas ocasiones cuesta entender . La programación funcional en Java 8 esta empezando y para la mayor parte **de la gente es algo muy nuevo**. Vamos a crear un ejemplo sencillo de flatMap, partiremos de dos clases relacionad**as Personas y Viajes.**



Una persona realiza varios viajes.

```
import java.util.ArrayList;
import java.util.List;
public class Persona {
      private String nombre;
      private List<Viaje> lista= new ArrayList<Viaje>();
      public Persona(String nombre)            { this.nombre = nombre;      }
      public String getNombre()                { return nombre;             }
      public void addViaje(Viaje v)            { lista.add(v);              }
      public List<Viaje> getLista()            { return lista;              }
      public void setNombre(String nombre)     { this.nombre = nombre;      }
}


public class Viaje {
      private String pais;
      public Viaje(String pais)          { this.pais = pais;    }
      public String getPais()            { return pais;         }
      public void setPais(String pais)   { this.pais = pais;    }
}
```

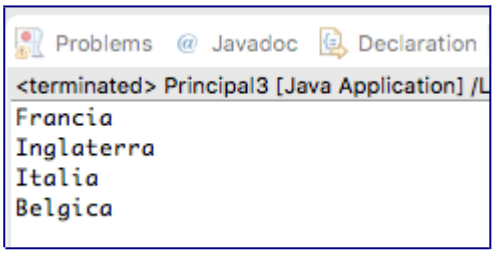
La estructura es anidada.

Es momento de crear y recorrer la estructura en código:

```
import java.util.ArrayList;
```

```
import java.util.List;
public class Principal {
	public static void main(String[] args) {
		Persona p= new Persona("pedro");
		Viaje v= new Viaje("Francia");
		Viaje v2= new Viaje("Inglaterra");
		p.addViaje(v);
		p.addViaje(v2);
		Persona p1= new Persona ("gema");
		Viaje v3= new Viaje("Italia");
		Viaje v4= new Viaje("Belgica");
		p1.addViaje(v3);
		p1.addViaje(v4);
		List<Persona> lista= new ArrayList<Persona>();
		lista.add(p);
		lista.add(p1);
		for(Persona persona:lista) {
			for (Viaje viaje: persona.getLista()) {
				System.out.println(viaje.getPais());
			}
		}
	}
}
```

La consola nos mostrará:



El problema es que en ningún caso hemos usado programación funcional para recorrer la lista, simplemente dos bucles anidados.

Usando Java 8 Map

El primer paso **va a ser usar la función map** para que a través de programación funcional nos imprima los nombres de las personas.

```
lista.stream().map(persona->persona.getNombre()).forEach(new Consumer<String>()
	{
		@Override public void accept(String s) {
			System.out.println(s);
		}
	});
```

Hemos convertido la lista en un Stream y a través de map() **hemos convertido el Stream de Personas en un Stream de "Strings"** que imprime los nombres.

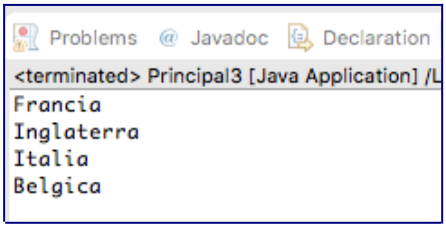
Lamentablemente no es lo que queríamos **ya que necesitamos imprimir el nombre de los países.**

Usando Java 8 FlatMap

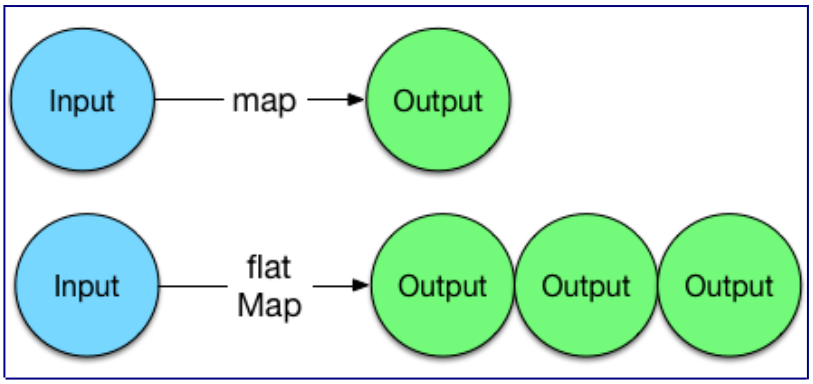
Para conseguir la lista de países debemos operar de otra manera **y usar la función flatMap.**

```
lista.stream().map(persona->persona.getLista())
      .flatMap(viajes->viajes.stream())
      .forEach(new Consumer<Viaje>() {
            @Override public void accept(Viaje t) {
                  System.out.println(t.getPais());
            }
      });
```

El resultado es :



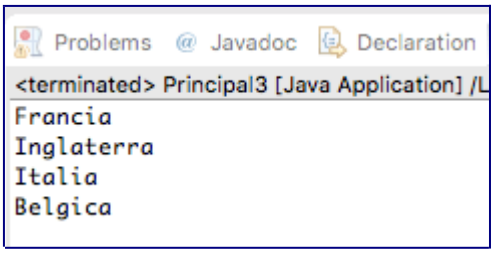
Ahora bien ,**¿Cómo funciona exactamente flatMap? .** FlatMap es una función que recibe una entrada y **devuelve varias salidas para esa entrada** . Esa es la diferencia con **Map que tiene una entrada y devuelve una única salida. En este caso hemos convertido el array en un stream.**



Podemos todavía optimizar **algo más nuestro código y volver a usar la función map**.

```
lista.stream().map(persona -> persona.getLista())
      .flatMap(viajes -> viajes.stream())
      .map(viaje->viaje.getPais()).forEach(System.out::println);
```

El resultado será idéntico:

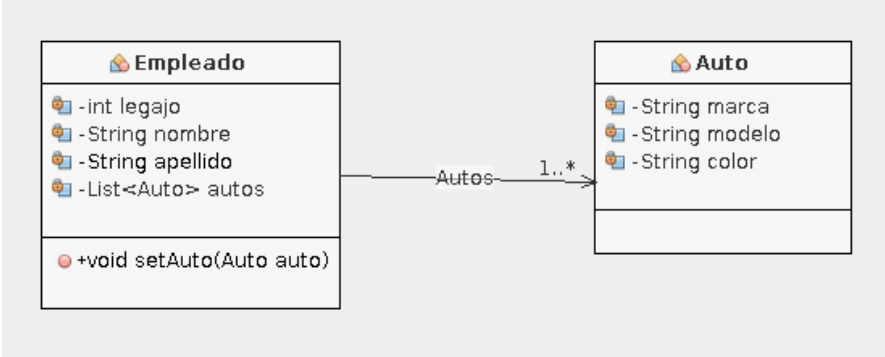


Java 8 FlatMap nos ha ayudado a eliminar todo tipo de bucle del programa.

Relaciones Múltiples, colaboración de clases

A continuación se verán ejemplos de comunicación múltiple entre clases. Estos ejemplos no fueron mostrados en la unidad inicial de objetos por no conocer sobre colecciones.

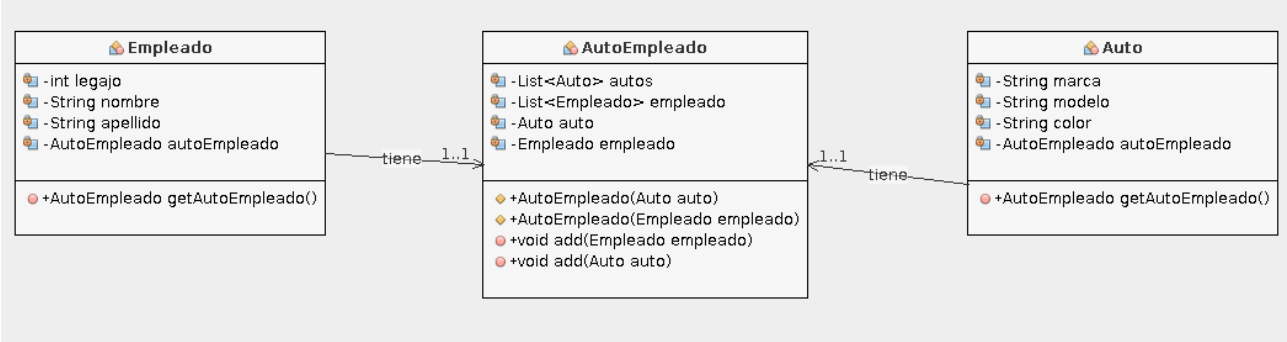
Relación de agregación 1 a Muchos.



```
public class Auto {
    private String marca;
    private String modelo;
    private String color;
    public Auto(String marca, String modelo, String color) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
    }
    @Override public String toString() {
        return "Auto{" + "marca=" + marca + ", modelo=" + modelo + ", color=" +
color + '}';
    }
}

import java.util.ArrayList;
import java.util.List;
public class Empleado {
    private int legajo;
    private String nombre;
    private String apellido;
    private List<Auto>autos;
    public Empleado(int legajo, String nombre, String apellido) {
        this.legajo = legajo;
        this.nombre = nombre;
        this.apellido = apellido;
        this.autos = new ArrayList();
    }
    @Override public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
apellido=" + apellido + '}';
    }
    public void addAuto(Auto auto){
        autos.add(auto);
    }
    public void removeAuto(Auto auto){
        autos.remove(auto);
    }
    public List<Auto> getAutos() { return autos; }
}
```

Relación de agregación Muchos a Muchos.



Es este gráfico se representa una relación muchos a muchos agregando la Clase AutoEmpleado que contiene un mapa de relaciones, de esta manera se desacopla totalmente el mantenimiento de las listas de relaciones en una tercer clases.

```
public class Auto {
    private String marca;
    private String modelo;
    private String color;
    private AutoEmpleado autoEmpleado;
    public Auto(String marca, String modelo, String color) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.autoEmpleado = new AutoEmpleado(this);
    }
    @Override public String toString() {
        return "Auto{" + "marca=" + marca + ", modelo=" + modelo + ", color=" +
color + '}';
    }
    public AutoEmpleado getAutoEmpleado() { return autoEmpleado; }
}
```

```
public class Empleado {
    private int legajo;
    private String nombre;
    private String apellido;
    private AutoEmpleado autoEmpleado;
    public Empleado(int legajo, String nombre, String apellido) {
        this.legajo = legajo;
        this.nombre = nombre;
        this.apellido = apellido;
        this.autoEmpleado = new AutoEmpleado(this);
    }
    @Override public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
apellido=" + apellido + '}';
    }
    public AutoEmpleado getAutoEmpleado() { return autoEmpleado; }
}
```

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

```
public class AutoEmpleado {
    List<Empleado>empleados;
    List<Auto>autos;
    Empleado empleado;
    Auto auto;
    public AutoEmpleado(Empleado empleado) {
        this.empleado = empleado;
        this.empleados = new ArrayList();
        this.autos = new ArrayList();
    }
    public AutoEmpleado(Auto auto) {
        this.auto = auto;
        this.empleados = new ArrayList();
        this.autos= new ArrayList();
    }
    public void addAuto(Auto auto){
        autos.add(auto);
        auto.getAutoEmpleado().empleados.add(empleado);
    }
    public void addEmpleado(Empleado empleado){
        empleados.add(empleado);
        empleado.getAutoEmpleado().addAuto(auto);
    }
}


public class Mock {
    public static void main(String[] args) {
        Auto auto1=new Auto("Ford", "Fiesta", "Rojo");
        Auto auto2=new Auto("Fiat","Idea","Blanco");
        Auto auto3=new Auto("Renault","Clio","Verde");

        Empleado empleado1=new Empleado(1, "Juan", "Perez");
        Empleado empleado2=new Empleado(2, "Ana", "Gomezz");
        Empleado empleado3=new Empleado(3, "Daniel", "Monte");

        auto1.getAutoEmpleado().addEmpleado(empleado1);
        auto1.getAutoEmpleado().addEmpleado(empleado2);
        empleado1.getAutoEmpleado().addAuto(auto3);

        System.out.println(empleado1);
        empleado1.getAutoEmpleado().autos.forEach(System.out::println);


    }
}
```