Apéndice 1

Cómo implementar estructuras para un tipo de elemento genérico

En este artículo se presentan los puntos más importantes a tener en cuenta para implementar en Java estructuras de datos que acepten cualquier tipo de elemento. Para ello, se evita declarar los tipos de elementos de manera específica (int, String, char, etc) y, en cambio, se definen de un tipo genérico o general que incluye a todos los tipos de dato aceptables para la estructura en cuestión.

En el caso de las estructuras que no requieren ningún tipo de ordenamiento entre sus elementos (Lista, Pila, Cola, Árbol Binario, Árbol Genérico, Hash), la estructura se deberá definir para manipular elementos de tipo *Object*[1]. En cambio, para las estructuras que deben mantener los datos ordenados parcial o totalmente (Lista Ordenada, Heap, Árbol Binario de Búsqueda, AVL, etc), se las definirá para elementos que implementen la interfaz Java *Comparable*.

A continuación se presentarán las clases Wrapper de Java y luego se explicarán los principios y ejemplos de implementación para crear estructuras de datos de tipo genérico.

Las clases envoltorio o Wrapper

Cómo ya se ha visto en las materias anteriores, Java permite manipular variables de dos tipos diferentes: tipos simples o primitivos (int, double, boolean, char, etc.) y clases (todas las clases Java predefinidas, como String y Date, más todas las clases definidas por el usuario). Sin embargo, en muchas ocasiones es conveniente poder tratar los datos de tipos primitivos como clases. Para ello, la API de Java incorpora un conjunto de clases envoltorio (wrapper), que permiten tratar a los datos de tipo primitivo como si fueran objetos.

Las clases wrapper existentes son:

- Byte para el tipo primitivo byte.
- Short para short.
- Integer para int.
- Long para long.
- Boolean para boolean
- Float para float.

[1] En algunos casos, según el dominio de aplicación, puede ser necesario mantener el orden entre elementos de una lista o de los árboles binario y/o genérico. En esos casos, el nodo correspondiente y las operaciones del TDA se podrán definir de también de tipo Comparable en lugar de Object.

- Double para double y
- Character para char.

Observese que las clases wrapper tienen el mismo nombre que el tipo primitivo (o similar en el caso de char) y que la primera letra es una mayúscula, dado que estos son nombres de clases.

Hay que tener en cuenta que las operaciones aritméticas habituales (suma, resta, multiplicación, etc.) están definidas sólo para los datos primitivos, por lo que las clases wrappers no sirven para ese fin.

Las variables de tipo primitivo tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes wrappers siempre que se pueda.

La conversión de tipo primiivo a clase de Java, que debía realizarse de forma explícita en las primeras versiones del lenguaje, en las versiones recientes se realiza automáticamente, por lo que es transparente al usuario. Por ese motivo, en la actualidad, las dos sentencias a continuación son equivalentes:

- Integer i = new Integer(5);
- Integer i = 5;

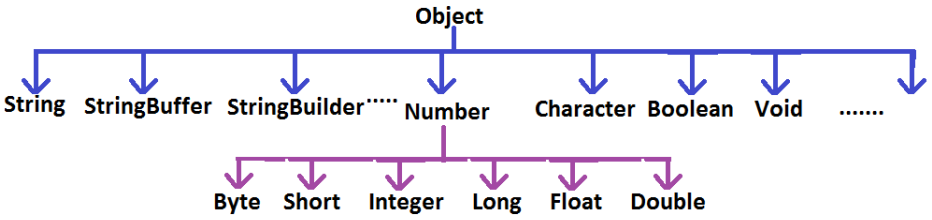
A modo de ejemplo, los métodos de la clase *Integer* más utilizados son los siguientes:

Método	Descripción
Integer(int valor)	constructor a partir de un número int
int intValue()	Devuelve el valor de la clase, convertido al tipo primitivo int
double doubleValue()	Devuelve el valor convertido a tipo primitivo double
boolean equals(Object obj)	Devuelve true si el objeto con el que se compara es un Integer y su valor es el mismo que el del objeto llamador
static int parseInt(String s)	Devuelve un int a partir de un String, en forma estática
String toString()	Devuelve un String a partir de un Integer
static Integer valueOf(String s)	Devuelve un Integer a partir de un String

Además, la clase Integer tiene dos atributos constantes públicos que pueden ser muy útiles al necesitar comparar valores int: el menor valor int posible (Integer.MIN_VALUE) y el mayor valor int posible (Integer.MAX_VALUE). Las otras clases para números enteros también tienen definidas estas constantes públicas.

1.1. Implementando estructuras que no requieren ordenar sus elementos

Es muy importante saber que la clase Object de Java es la clase raíz del árbol de jerarquía de clases de Java. Esto implica que todas las clases Java heredan directa o indirectamente de la clase Object. Como se puede observar en la figura a continuación, la clase String hereda de la clase Object, y también todas las clases wrapper.



De esta manera, una estructura definida para elementos de tipo Object puede ser utilizada para almacenar elementos de cualquier tipo Java, aunque la recomendación es, una vez creada una estructura, guardar en ella siempre elementos del mismo tipo, para que al recuperarlos se sepa (sin necesidad de preguntar) de qué tipo es el objeto recuperado.

1.1.1. Cómo convertir un elemento de tipo Object a su tipo original (casting)

Algo muy importante es recordar que al guardar un elemento en una estructura (por ejemplo mediante el método *apilar* de Pila), no es necesario hacer ninguna conversión explícita entre clases, porque el método apilar está implementado para recibir elementos de tipo Object. Como Object es la raíz de la jerarquía de clases de Java, cualquier elemento de tipo String, Integer o definido por el usuario, son considerados también instancias de la superclase Object y son aceptados automáticamente como tales.

Sin embargo, cuando un objeto almacenado en una estructura como Object debe ser usado como instancia de la subclase original, esta conversión sí debe ser hecha de forma explícita. Dicha conversión se llama "casting", y se logra anteponiendo a la variable, entre paréntesis, el nombre de la subclase a la que se quiere convertir.

Por ejemplo, en el Algoritmo 1.1 se muestra una clase TestPila que crea y guarda varias cadenas (String) en una Pila y luego las recupera para mostrarlas convertidas a mayúsculas. Como puede observarse en el código, al insertar los elementos la conversión es automática (1). Luego, cuando recupera el tope para mostrarlo en mayúsculas (2) es necesario convertirlo a String antes de invocar el método toUpperCase() dado que se trata de un método propio de la clase String.

Resumiendo: la conversión de String a Object (subclase a superclase) es automática al guardar el elemento en la estructura, pero la conversión de un objeto recuperado de la estructura para poder utilizarlo como instancia del tipo original (superclase a subclase) debe ser explícita (casting).

Algoritmo 1.1 Ejemplos de uso de elementos recuperados de una Lista para tipos genéricos

```
public class TestPila {

    public static void main(String[] arg) {

        Pila miPila = new Pila();
        String cadena;
        int i;

        for (i = 1; i <= 10; i++) {
            // crea y carga 10 elementos en la pila
            // al insertar, la conversión de String a Object es automática (1)
            miPila.apilar("elem-" + i);
        }

        for (i = 1; i <= 10; i++) {
            // al usar un método propio de String
            // es necesario hacer casting (2)
            cadena = (String) miPila.obtenerTope();
            System.out.println(cadena.toUpperCase());
            miPila.desapilar();
        }

    }
}
```

1.1.2. Métodos heredados de Object

Además de ser la superclase de todas las clases Java, la clase Object define el comportamiento básico que todos los objetos Java deben tener. Para ello, esta clase define una serie de métodos que hereda a todas sus subclases y por lo tanto, están disponibles en todas las clases Java. De los métodos que se heredan de Object, en este momento nos interesan especialmente los siguientes:

- *toString*: es necesario para escribir el método toString de la estructura, independientemente del tipo almacenado

- *equals*: es necesario para preguntar si un elemento es el que se está buscando, por ejemplo para los métodos *localizar* de Lista, *pertenece* y *padre* de Árbol Binario o de Árbol Genérico, entre otros.

1.1.2.1. Redefiniendo el método toString() de Object en clases propias

El método *toString*() de Object está definido para devolver una cadena de texto (String) que representa el contenido del objeto. El método toString() se invoca de forma automática cuando se nuestra un objeto mediante la instrucción System.out.println o System.out.print. Esto quiere decir que las siguientes instrucciones son equivalentes:

- System.out.println(unObjeto.toString());
- System.out.println(unObjeto);

Ejemplo de reescritura del método toString

Sea el ejemplo de una clase llamada Punto, que tiene dos atributos que representan las coordenadas en los ejes X e Y, tal como se implementa en el Algoritmo 1.2:

Algoritmo 1.2 Ejemplo de clase Punto definida por el usuario

```
public class Punto {

    // atributos
    private double x;
    private double y;

    //constructor
    public Punto(double nX, double nY) {
        this.x = nX;
        this.y = nY;
    }

    // interfaz
    public double getX() {
        return x;
    }

    public void setX(double nX) {
        this.x = nX;
    }

    public double getY() {
        return y;
    }

    public void setY(double nY) {
        this.y = nY;
    }

}
```

En Algoritmo 1.3, se muestra una clase de prueba que crea una instancia de la clase Punto e intenta imprimir su contenido:

Algoritmo 1.3 Ejemplo de clase que utiliza a la clase Punto

```
public class PruebaPunto {

    public static void main(String[] arg) {

        Punto miPunto = new Punto(5.4, 6.78);

        System.out.println(miPunto.toString());

    }
}
```

Al ejecutarse el método main en el Algoritmo 1.3, como la clase Punto no tiene un método toString() propio, la instrucción System.out.println(miPunto.toString()) invocará al método toString() heredado desde Object, y luego mostrará el String devuelto por dicho método. Por ejemplo, se mostrará por pantalla algo similar a `miPaquete.Punto@de4588.` Esta cadena devuelta por el método toString() heredado de Object contiene el nombre del package (miPaquete) seguido del nombre de la Clase (Punto) y de un número que es un identificador de la referencia a la instancia miPunto.

Obviamente esto no es lo que se espera cuando se quiere mostrar un objeto, por lo tanto, en la clase Punto se debe redefinir o sobrescribir (override) el método toString() para representar nuestro objeto de forma más adecuada. Para sobrescribir un método heredado *hay que mantener la signatura o encabezado del método tal cual se hereda.* Si se escribe el método de forma distinta (por ejemplo, añadiendo un parámetro o cambiando el tipo de un parámetro o del resultado) se sobrecargará el método en lugar de redefinirlo.

A continuación, sobreescribiremos el método toString() de la clase Punto para mostrar el valor de las coordenadas del punto, con la notación "(x,y)" para que cuando se ejecute la instrucción System.out.println(miPunto) se muestre, por ejemplo, la cadena (5.4, 6.78). Para ello se agregará en la clase Punto el método toString() como en el Algoritmo 1.4:

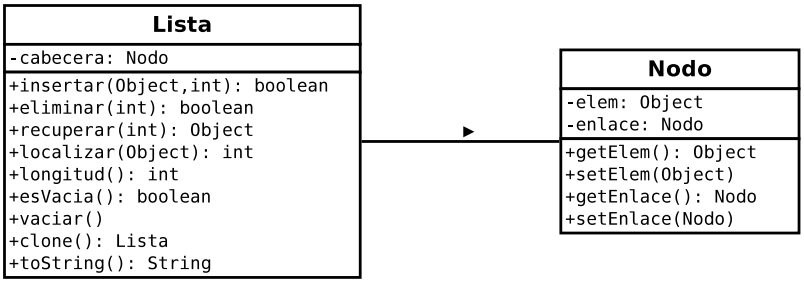
Algoritmo 1.4 Ejemplo de reescritura del método toString de Object

```
@Override
public String toString(){
    return "(" + x + ", " + y + ")";
}
```

La anotación @*Override* no es necesaria, aunque se recomienda agregarla antes de la definición del método para indicar que se está redefiniendo el método heredado de Object. Después de añadir este método a la clase, el método *main* anterior mostrará por pantalla la cadena "(5.4, 6.78)".

1.1.2.2. Redefiniendo el método equals() de Object en clases propias

Para ilustrar por qué es necesario redefinir el método equals, a continuación se presenta el código del método *localizar* de la clase Lista, implementada de acuerdo a la siguiente especificación UML, donde el TipoElemento se ha definido de tipo Object.



El método *localizar* presentado en el Algoritmo 1.5, es la implementación en Java para la especificación anterior. Este método, de acuerdo a la especificación del TDA Lista, comienza por el elemento de la cabecera y avanza de manera iterativa, comparando cada elemento que encuentra con el elemento buscado. Para ello, la sentencia repetitiva while primero pregunta si el puntero aux es distinto de nulo y, aprovechando el operador AND con cortocircuito de Java (&&), pregunta si el elemento apuntado por aux es el elemento buscado. Se debe prestar atención a la forma de redactar esta condición, donde se pregunta *aux.getElem().equals(buscado)* en lugar de utilizar el operador == , dado que el operador de igualdad == sirve sólo para los tipos simples de Java y el método equals está definido para todas las clases que heredan de Object (retornando false por defecto).

Algoritmo 1.5 Implementación del método *localizar* de Lista para tipo de elemento genérico

```
public int localizar(Object buscado) {
    // devuelve la posición del elemento buscado
    // si el elemento no existe devuelve -1
    int pos;

    // avanza buscando el elemento
    Nodo aux = this.cabecera;
    int i = 1;
    while (aux != null && !aux.getElem().equals(buscado)) {
        aux = aux.getEnlace();
        i++;
    }
    // sale porque aux = null (error) o pq encontro el elemento (pos i)
    if (aux == null) {
        pos = -1;
    } else {
        pos = i;
    }
    return pos;
}
```

> Importante
> *Comparar dos objetos con el método equals no es lo mismo que hacerlo con el operador ==*
>
> Al contrario: El operador == compara si las referencias a los objetos que se están comparando apuntan al mismo objeto, mientras que el método *equals* compara si su contenido es el mismo.

Como se dijo antes, todas las clases predefinidas de Java (String, Integer, Double, Character, etc). ya tienen implementado el método *equals* y podrán ser utilizados como elementos de una Lista o cualquier estructura que no requiera tener sus elementos ordenados, simplemente recordando escribir las comprobaciones de igualdad de elementos utilizando el método equals en lugar del operador ==.

Si bien las pruebas con las estructuras vistas en la materia se pueden realizar utilizando solamente objetos de las clases predefinidas, también es importante saber cómo trabajar con clases definidas por el usuario para modelar un dominio en especial, para lo cual el método *equals* debe ser redefinido de manera adecuada, como se explicará a continuación.

Ejemplo de reescritura del método equals

Como se explicó antes, el método *equals* de la clase Object se espera que compare dos objetos para ver si son o no iguales y devuelva un valor lógico (boolean) adecuado. Por defecto, para toda clase definida por el usuario, el método *equals* devuelve siempre false.

Como se hizo antes con el método *toString*(), se debe sobrescribir el método *equals*() para adaptarlo a lo que signifique la igualdad para la clase que se desea implementar. Una posible implementación para la clase Punto es la que se muestra en el Algoritmo 1.6. Para ello, el método equals primero verifica que el

objeto recibido por parámetro no es null, luego lo fuerza a convertirse en Punto (casting) y a continuación verifica que ambos tienen los mismos datos internos. Luego, devuelve *true* si son iguales o *false* en caso contrario.

Algoritmo 1.6 Ejemplo de reescritura del método equals de Object

```
@Override
public boolean equals(Object obj) {
    boolean res;
    if (obj == null) {
        res = false;
    } else {
        Punto otro = (Punto) obj;
        res = (this.x == otro.x && this.y == otro.y);
    }

    return res;
}
```

Es importante notar que el método *equals* que se sobreescribe recibe un parámetro de tipo Object, no Punto, ya que sino no estaría sobreescribiendo el método de Object sino creando uno nuevo con igual nombre y distinto tipo de parámetro (sobrecarga).

El método equals, así como el método hashCode, se pueden generar automáticamente utilizando la opción "Insertar Código" disponible en entornos de desarrollo tipo NetBeans o Eclipse. Por ejemplo, en Netbeans se debe acceder al menú Source -> Insert Code.

1.2. Implementando estructuras que requieren ordenar sus elementos

La clase Object de Java sólo permite preguntar si un objeto es igual o no a otro, pero no tiene métodos para preguntar si un objeto es mayor o menor que otro, por lo tanto, para implementar estructuras que necesitan mantener sus elementos ordenados, es necesario definirlas para objetos que, además de heredar de la clase Object (que ya sabemos que todas las clases en Java lo hacen), cumplan con la interfaz *Comparable* de Java .

A diferencia de Object, que es una clase concreta y que tiene los métodos *toString* y *equals* implementados de manera básica, una interfaz Java es una colección de métodos abstractos y propiedades constantes. En las interfaces se especifica *qué se debe hacer* pero no su implementación (cómo). Las clases que implementen dichas interfaces serán las que describan la lógica del comportamiento de los métodos.

Según su definición, la interfaz Comparable se emplea para indicar si el objeto llamador, representado por `this`, es menor, igual o mayor que el objeto que se recibe por parámetro.

Para ello, la interfaz Comparable define que toda clase que la implemente debe implementar el método:

```
public int compareTo(Object otro)
```

El resultado del método compareTo tendrá el siguiente significado según el valor retornado:

- **Mayor a cero (>0)** significa que el valor de this es mayor que el valor del otro, es decir valor(this) > valor(otro)
  Ejemplo: si estamos comparando elementos de la clase Integer y creamos un objeto num = 9 (que es equivalente a hacer num=new Integer(9)), y luego se ejecuta la sentencia num.compareTo(5), el resultado esperado es algo mayor a 0, ya que 9>5

- **Menor a cero (<0)** significa que el valor de this es menor que el valor del otro, es decir valor(this) < valor(otro)

Ejemplo: si estamos comparando el objeto num = 9 del ejemplo anterior y se ejecuta la sentencia num.compareTo(14), el resultado esperado es un entero menor a 0, ya que 9<14

- **Igual a cero (=0)** significa que el valor de this es igual que el valor de otro, es decir valor(this) = valor(otro)
  Ejemplo: sea el objeto num = 9 del ejemplo anterior, la sentencia num.compareTo(9) devuelve 0, ya que 9=9
  Nota: compareTo con resultado 0 es equivalente a la ejecución del método *equals* con los mismos objetos.

Por ejemplo, sea el siguiente fragmento de código utilizando el método compareTo de la clase String, que sabemos que implementa la interfaz Java Comparable:

```
String s1 = "HOLA";
String s2 = "AGUA";
String s3 = "HORA";
String s4 = "agua";
String s5 = "agua";

System.out.println(s1.compareTo(s2));
System.out.println(s2.compareTo(s1));
System.out.println(s1.compareTo(s3));
System.out.println(s4.compareTo(s5));
System.out.println(s2.compareTo(s5));
```

Los resultados obtenidos al ejecutarlo son los siguientes:

- s1.compareTo(s2) retorna 7. Como 7 > 0 indica que "HOLA" > "AGUA"
  Esto es verdad, ya que la letra H está después en el alfabeto que la letra A.

- s2.compareTo(s1) retorna -7. Como -7 < 0 indica que "AGUA" < "HOLA"
  Esto es verdad, por la misma razón que la anterior.

- s1.compareTo(s3) retorna -6. Como -6 < 0 indica que "HOLA" < "HORA"
  La comparación la realiza considerando que las letras H y O son iguales en ambas cadenas, y la tercera letra (L) es menor que la tercera letra (R) de la segunda cadena.

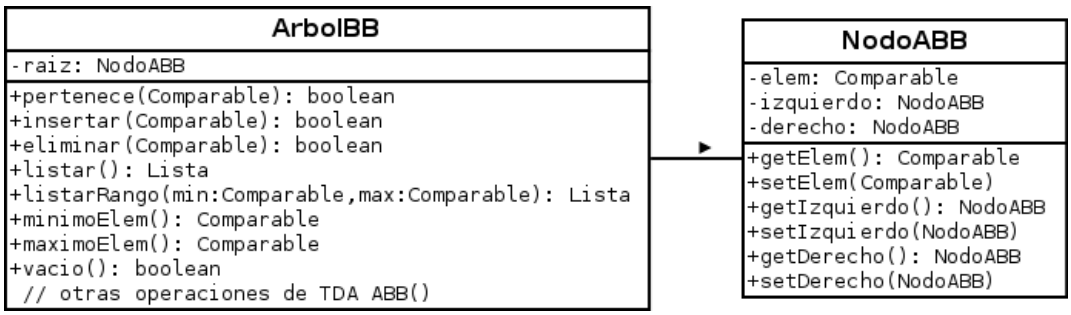
- s4.compareTo(s5) retorna 0. Como 0=0 indica que "agua"="agua"
  Obviamente, las cadenas "agua" en s4 y "agua" en s5 son exactamente iguales

- s2.compareTo(s5) compara las cadenas "AGUA" y "agua" y retorna -32, lo que indica que "AGUA" es menor que "agua". ¿Por qué ocurre esto? Java analiza el orden entre caracteres no de forma alfabética (como acostumbramos los humanos), sino de forma lexicográfica. Para ello utiliza el código ASCII de cada letra. Si nos fijamos en una tabla ASCII[2] veremos que el valor ASCII de la letra "A" mayúscula es 65 y de la letra "a" minúscula es 97. Por eso, si se desea que la comparación sea alfabética y no lexicográfica, es importante normalizar los caracteres de ambos String a mayúsculas o minúsculas antes de realizar la comparación.

De esta manera, aprovechando la interfaz Java Comparable es posible implementar estructuras que necesitan mantener sus elementos de acuerdo a un orden (parcial o total) preestablecido, como el árbol binario de búsqueda, el árbol heap, etc.

Para ilustrar cómo deben implementarse los métodos de una estructura que manipule elementos de tipo Comparable, a continuación se presentará el código del método *pertenece* de la clase ArbolBB, implementada de acuerdo a la siguiente especificación UML, donde el TipoElemento se ha definido de tipo Comparable.

[2] http://ascii.cl/es/



A continuación se presenta el método *pertenece* de dicha clase ArbolABB. Preste atención a la manera de hacer las consultas en las condiciones *if*, evaluando el resultado del método compareTo de la clase:

```
public class ArbolBB {

    private NodoABB raiz=null;


    public boolean pertenece(Comparable elemento) {
        return perteneceAux(this.raiz, elemento);
    }

    private boolean perteneceAux(NodoABB n, Comparable elemento) {
        boolean exito = false;
        if (n != null) {
            if ((elemento.compareTo(n.getElem()) == 0)) {
                // Elemento encontrado
                exito = true;
            } else if (elemento.compareTo(n.getElem()) < 0) {
                // elemento es menor que n.getElem()
                // busca a la izquierda de n
                exito = perteneceAux(n.getIzquierdo(), elemento);
            } else {
                // elemento es mayor que n.getElem()
                // busca a la derecha de n
                exito = perteneceAux(n.getDerecho(), elemento);
            }
        }
        return exito;
    }
```

Como ocurría con la clase Object, las clases de Java predefinidas (String, Integer, Double, Character, etc.) implementan la interfaz Comparable, por lo tanto podrán ser utilizados como elementos de ArbolABB o cualquier estructura que requiera tener sus elementos ordenados definida para elementos Comparable. Simplemente se debe recordar escribir las comprobaciones de órden menor y mayor utilizando el método compareTo definido para dicha interfaz.

Si bien las pruebas con las estructuras vistas en la materia se pueden realizar utilizando las clases predefinidas de Java, es muy importante saber cómo hacerlo con las clases de un dominio en especial, para lo cual se deben realizar los pasos que se indicarán a continuación.

Haciendo Comparable una clase propia

A continuación se muestra un ejemplo de cómo una clase puede implementar la interfaz Comparable para ser utilizado en una estructura definida de forma genérica. Para ello se ha tomado como ejemplo la misma clase Punto usada en la primera parte del artículo.

Se debe prestar atención que para hacer una clase Comparable, es imprescindible incluir la leyenda "implements Comparable" a continuación de la declaración "public class Punto", para indicar que la clase va a cumplir con los requisitos de dicha interfaz.

Además, se debe agregar el método compareTo con un parámetro de tipo Object y resultado de tipo int e implementarlo de manera que cumpla los requisitos de devolver algo > 0 si valor(this) $>$ valor(otro); algo < 0 si valor(this) $<$ valor(otro); y 0 si valor(this) $=$ valor(otro).

```
public class Punto implements Comparable {

    private double x;
    private double y;

    @Override
    public int compareTo(Object obj) {
        // se compara tomando en primer lugar x y luego y
        int resultado;
        Punto otro = (Punto) obj;
        if (this.x < otro.x) {
            resultado = -1;
        } else if (this.x > otro.x) {
            resultado = 1;
        } else {
            // x es igual, debo ver a y
            if (this.y < otro.y) {
                resultado = -2;
            } else if (this.y > otro.y) {
                resultado = 2;
            } else {
                resultado = 0;
            }
        }
        return resultado;
    }
```

Al tener la clase Punto dos atributos (x e y), para implementar el método compareTo es necesario definir si ambos pertenecerán a la clave o no. Como x e y son ambos necesarios para identificar un punto en el plano, se eligió primero comparar el valor de x y luego el valor de y.

Importante: La clave de un elemento es un atributo o un conjunto de atributos, que lo identifican de forma unívoca. Por ejemplo, la clave de un alumno en el sistema de la FAI es su legajo. En caso que la clave esté formada por un sólo atributo, comparar dos objetos requerirá comparar sólo ese atributo (ejemplo: legajo de un alumno). En caso que la clave sea compuesta, se deberá elegir a uno de ellos para compararlo en primer lugar. Sólo en caso que dicho atributo sea igual en ambos objetos, se considerará el segundo atributo. Y en caso que el primero y el segundo sean iguales en ambos objetos, se considerará el tercer atributo (si lo hubiera), etc.[3]

A modo de ejemplo, sean los siguientes puntos:

```
Punto punto1 = new Punto(2.7, 3.2);
Punto punto2 = new Punto(2.7, 3.5);
Punto punto3 = new Punto(3.8, 3.2);
Punto punto4 = new Punto(2.7, 3.1);
```

El resultado obtenido al ejecutar el método *compareTo* de la clase Punto, para compararlos entre sí, es el siguiente:

- punto1.compareTo(punto1): 0
- punto2.compareTo(punto3): -1
- punto3.compareTo(punto2): 1
- punto1.compareTo(punto2): -2

[3]Ejercicio: pensar cómo implementaría el método compareTo de la clase Fecha cuya clave está formada por los atributos día, mes y año.

- punto1.compareTo(punto4): 2