Herencia, polimorfismo y genericidad.

1. Relaciones de herencia

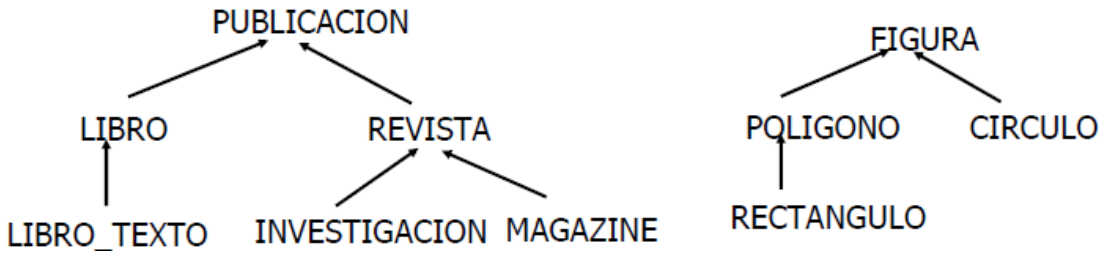
La herencia es un mecanismo de la Programación Orientada a Objetos que permite definir unas clases a partir de otras ya existentes. Por ejemplo, permite definir una clase Alumno a partir de una clase, ya existente, Persona.

```
public class Alumno extends Persona {…}
```

Entre las clases Alumno y Persona se da una **relación ES_UN** (IS_A): *Alumno ES_UN Persona*. Cuando una clase (Alumno) hereda (extends) de otra (Persona):

- La clase hija (Alumno), hereda todos los atributos y todos los métodos de su clase padre (Persona).
- La clase hija puede tener atributos nuevos que no tiene la clase padre.
- La clase hija puede tener métodos nuevos, que no tiene la clase padre.
- La clase hija puede reescribir (*override*) métodos que ya existían en la clase padre para darles un nuevo comportamiento.

La herencia permite crear jerarquías de clases que facilitan la reutilización de código y que los diseños sean más consistentes



Las clases se especializan, e incorporan nuevos atributos y métodos que no estaban en sus clases antecesoras. En el ejemplo, diríamos que:

- LIBRO es la **clase padre** (o **superclase**) de LIBRO_TEXTO.
- LIBRO_TEXTO es una **clase hija** (o **subclase**) de LIBRO
- LIBRO_TEXTO es una subclase de PUBLICACION. Las relaciones de herencia son **transitivas**: si C hereda de B y B hereda de A, entonces C también hereda de A
- PUBLICACION es una superclase de LIBRO_TEXTO.
- Libros y Revistas son una **especialización** de Publicación, un tipo especial de Publicación.
- Publicación es la **generalización** de Libros y Revistas.

1.1. Tipos de herencia

La herencia puede ser **simple** o **múltiple:**

En la **herencia simple** las clases tienen un único padre, es decir, una clase solo hereda las características de otra clase.

En la **herencia múltiple,** se contempla la posibilidad de que una clase pueda heredar atributos y métodos de varias clases.

Java no permite herencia múltiple

1.2. Acceso a los atributos y métodos de la clase padre.

Como hemos dicho, una clase hereda todos los atributos y métodos de la clase de la que se extiende. Sin embargo, eso no implica necesariamente que esos atributos y métodos heredados sean accesibles desde la clase hija. Ello dependerá del **modificador de ámbito** con el que estén definidos estos atributos y métodos en la clase padre (*private*, *public* , …).

Hasta ahora hemos venido utilizando dos modificadores: private y public, pero existen dos ámbitos más que cobran valor cuando hay relaciones de herencia: *protegido* y "*frendly*".

- El **ámbito protegido** se aplica a un atributo o método utilizando la palabra reservada ***protected*** en su declaración.
- El **ámbito "friendly"** se aplica no poniendo ningún modificador de ámbito.

En la siguiente tabla resumen se indica desde qué clases son accesibles los atributos y métodos definidos con cada modificador

Modificador	**Accesible desde …**
private	Solo desde la clase en que está definido
"friendly"	Desde la clase en que está definido Desde otras clases que se encuentran en el mismo paquete
protected	Desde la clase en que está definido Desde las subclases. Desde otras clases que se encuentran en el mismo paquete
public	Desde cualquier clase

1.3. La clase Object

Cuando se crea una clase sin indicar que hereda de otra, ésta heredará de la clase Object del paquete java.lang.

```
public class Persona { …. }
```

equivale a

```
public class Persona extends Object {...}
```

Por tanto, todas las clases tienen como superclase a la clase Object y heredarán de ella sus características. La clase Object tiene los siguientes métodos, algunos de ellos ya conocidos y otros que se estudiarán más adelante. Daremos especial importancia a equals, toString y hashCode

- `public boolean equals (Object o):` Compara dos objetos y dice si son iguales. En su comportamiento por defecto, devuelve true si los dos objetos que se comparan tienen la misma dirección de memoria, es decir, si son el mismo objeto.
- `public String` **`toString`** `():` Devuelve una representación del objeto en forma de cadena de texto. Por defecto, el String que devuelve hace referencia al tipo del objeto y a su dirección de memoria.
- `public Class` **`getClass`** `():` Devuelve un objeto de la clase Class, que contiene información sobre la clase a la que pertenece el objeto.
- `public int` **`hashCode`** `():` Devuelve un número entero, calculado a partir del objeto. Por defecto, el número que devuelve se calcula a partir de la dirección de memoria del objeto. Estudiaremos para qué sirve este método en el siguiente tema, dedicado al API Collections de Java.
- `public void finalize():` Es un método al que llama el Garbage Collector (Recolector de basura) para eliminar los objetos de memoria cuando éstos ya no se utilizan.
- `public Object clone():` Método que se utiliza para devolver una copia del objeto

1.4. Constructores

Con respecto a los métodos Constructores, hay que tener en cuenta las siguientes consideraciones:

- Los constructores no se heredan. Por tanto, es necesario que cualquier clase incorpore su propio constructor.
- Si una clase no tiene constructor de forma explícita, entonces Java incorpora un constructor implícito (es decir, que no aparece escrito en ningún sitio): public NombreClase() { }. A este constructor se llama *constructor por defecto*

En las subclases:

- La primera instrucción del constructor de una subclase tiene que ser una llamada al constructor de la clase padre. Para llamar al constructor de la clase padre, se usa el método **super(…),** con aquellos parámetros que el constructor de la clase padre requiera.
    - Si no se llama a super(…) explícitamente, Java llamará de forma implícita a super() (sin parámetros). En tal caso, si la clase padre no tiene un constructor sin parámetros se producirá error de compilación.
- Si no creamos un constructor en la subclase, Java crea uno implícito por defecto que llama a super() (sin parámetros). Si la clase padre no tiene un constructor sin parámetros se producirá error de compilación.
- El constructor de una subclase no tiene por qué recibir los mismos parámetros que el de su clase padre

1.5. Reescritura de métodos (overriding)

Se dice que estamos reescribiendo un método cuando una subclase implementa un método que ya estaba presente en alguna de sus clases antecesoras, es decir, cuando se vuelve a implementar un método heredado.

Se reescribe un método para cambiar el comportamiento que tiene en sus clases antecesoras.

Para reescribir un método hay que respetar la declaración del método:

- El nombre ha de ser el mismo.
- Los parámetros y tipo de retorno han de ser los mismos.
- El modificador de acceso no puede ser más restrictivo. Así, un método que en la clase padre era protected, podría ser public en la subclase, pero no al contrario.

La reescritura puede ser **total** o **parcial**:

- Es **total** cuando no se aprovecha la implementación del método en la clase padre. El método se reescribe completamente.
- Es **parcial** cuando desde en el método de la subclase se llama al mismo método de la superclase y el resultado obtenido se completa de alguna manera para producir un nuevo resultado. Para llamar al mismo método en la clase padre, se usa la palabra reservada **super**.

1.6. La palabra reserada “super”

super se utiliza para acceder desde una clase métodos de la clase padre:

- ***super.metodo(…),*** para acceder a un método definido de la clase padre que se está reescribiendo en la clase hija.
- ***super(…)*** para llamar al constructor de la clase padre desde el constructor de la clase hija.

1.7. Clases abstractas y métodos abstractos

Clases abstractas.

Definimos una clase como abstracta cuando no tiene sentido instanciarla. No tiene sentido crear un objeto de dicha clase.

Una clase se define abstracta añadiendo la palabra reservada abstract en su declaración:

```
public abstract class Figura {…}
```

Son clases que sirven para que otras clases hereden de ella, pero que no tiene sentido instanciar (crear objetos de ella). Por ejemplo, puede tener sentido crear objetos de tipo Circulo pero no de tipo Figura. La clase Figura sirve para modelizar aquello que tienen en común distintos tipos de figuras (Circulo, Rectangulo, etc), como pueden ser su color o su posición.

No tendrá sentido crear una clase abstracta si no es para heredar de ella.

Las clases abstractas se usan para

- Agrupar bajo un mismo tipo a otras clases
- Contener código reutilizable, común a sus subclases.
- Forzar un API a sus subclases, es decir, obligar a que las subclases tengan una serie de métodos con nombres, parámetros y tipos de retorno comunes

Declaración

```
modificador_acceso abstract class nom_clase {…}
```

Ejemplo:

```
public abstract class Figura {…}
```

Métodos abstractos.

Un método abstracto es un método que no tiene implementación en la clase en que se define abstracto. Se define abstracto porque a ese nivel de la jerarquía de clases no se conoce cómo implementarlo. Lo habitual es que sea posible implementarlo en subclases posteriores.

Un método se define abstracto utilizando la palabra reservada abstract. En un método abstracto únicamente se define la cabecera del método.

```
public class Figura {
        . . .
        public abstract double area();
        public abstract double perimetro();
        . . .
}
```

Consideraciones:

- Si una clase tiene algún método abstracto, ENTONCES la clase tiene que ser abstracta. Sin embargo, una clase puede ser abstracta y no tener métodos abstractos.
- Si una clase tiene métodos abstractos, ENTONCES sus subclases TIENEN QUE:
    - Implementar todos los métodos abstractos de sus superclases, en cuyo caso la subclase puede no ser abstracta, o
    - Continuar siendo abstracta, como lo es su clase padre
- En definitiva, una clase que tiene métodos por implementar tiene que ser abstracta.
- Aunque sea abstracto en la clase, se le puede llamar.
- Definir un método abstracto en una clase, implicará que todos los objetos pertenecientes a sus subclases dispondrán de ese método implementado.

Por ejemplo, en la clase Figura, los métodos área() y perímetro() son abstractos:

- En la clase Figura no disponemos de la suficiente información para poder dar contenido (implementar) esos dos métodos.
- Estos métodos podrán tener una implementación en las subclases. Si, por ejemplo, la clase Circulo implementa esos dos métodos, ya no tiene métodos pendientes de implementar y no tiene por qué ser abstracta.
- Definiendo estos dos método en la clase Figura, nos aseguramos de que todas las subclases (no abstractas) de Figura tengan implementados estos métodos.
- Los métodos área y perímetro se pueden usar en otros métodos de la clase Figura, aunque aún no esté implementado en la clase. Por ejemplo, el método toString de la clase Figura podría devolver un texto con el área y el perímetro de la figura.

1.8. Polimorfismo. Tipo estático y dinámico

Cuando se ha definido una jerarquía de clases, es posible usar variables del tipo de una superclase para referenciar a objetos de alguna de sus subclases. Cuando esto sucede, se dice que hay **polimorfismo.** Por ejemplo:

```
Object o = “Hola”;
```

En esta expresión, la variable o ha sido definida de tipo Object. El objeto al que referencia, sin embargo, pertenece a la clase String (subclase de Object).

En el siguiente ejemplo también habrá polimorfismo:

```
public static double sumaAreas(Figura[] v){
        double suma = 0;
        for(int i = 0; i < v.length; i++){
                suma = suma + v[i].area();
        }
        return suma;
}
```

El tipo de los elementos del array es Figura, mientras que cada uno de los elementos del array será un objeto perteneciente a alguna de sus subclases (Circulo, Rectángulo, …).

Las asignaciones polimórficas serán posible siempre que el objeto a la derecha de la asignación **sea del mismo tipo o de un subtipo** que el de la variable a la izquierda de la asignación:

Serían posibles asignaciones como:

```
Figura f = new Circulo(…);
Persona p = new Alumno(…);
Publiacion p = new Revista(…)
```

Pero serían incorrectas otras como:

```
Circulo c = new Rectangulo(…);
Circulo c = new Alumno(…);
Revista r = new Publicacion(…);
```

Cuando hay polimorfismo, tenemos **dos tipos:**

- **Tipo estático**
  - Es el tipo con el que se declara la variable.
  - No cambia durante la ejecución del programa.
  - Java lo utiliza en tiempo de compilación, para determinar si el programa tiene errores o no.
    - Por ejemplo, si la variable f se inicializa como Figura f = new Circulo(…), la instrucción f.getRadio() producirá error de compilación, porque la clase Figura no tiene método getRadio().

- **Tipo dinámico**
  - Es el tipo del objeto al que referencia la variable
  - Puede cambiar durante la ejecución, puesto que una misma variable puede referenciar a diversos objetos a lo largo de la ejecución del programa.
  - Java lo utiliza en tiempo de ejecución para determinar qué método ejecutar entre varios posibles.
    - Por ejemplo, si la variable f se inicializa como Figura f = new Circulo(…), la instrucción f.toString() llamará al método toString() de la clase Círculo, y no al de la clase Figura.

El polimorfismo permite implementar métodos genéricos, es decir métodos que no tengan que ser cambiados aunque se incorporen nuevas subclases a una jerarquía. En el siguiente ejemplo (que ya ha aparecido anteriormente), la variable v es polimórfica. Gracias a esta característica, el método sumaAreas seguirá siendo válido tal y como está aunque se añadan nuevas clases a la jerarquía de figuras (Triangulo, Pentagono, …)

```
public static double sumaAreas(Figura[] v){
        double suma = 0;
        for(int i = 0; i < v.length; i++){
                suma = suma + v[i].area();
        }
        return suma;
}
```

1.9. Interfaces

¿Qué son?

Los interfaces son elementos del lenguaje Java que únicamente contienen definiciones de constantes y cabeceras de métodos, sin su implementación. *(Además también permiten definir métodos estáticos y métodos por defecto, pero éstas son características avanzadas que se usan para añadir funcionalidad a los interfaces sin que se produzcan problemas de compatibilidad con código que se creó anteriormente)*

Un interface es similar a una clase que tuviera todos sus métodos abstractos.

¿Cómo se definen?

Los interfaces se definen en archivos .java cuyo nombre tiene que ser el mismo que el del interfaz. Por ejemplo, el interface *"Cola"* se definiría en el archivo *"Cola.java"*.

Para definir un interface se utiliza la palabra reservada ***interface***. Entre las llaves, se ponen las cabeceras de los métodos que define el interface, sin usar la palabra *public* ni la palabra *abstract*. Aunque todos los métodos se consideran públicos y abstractos, no es necesario indicarlo.

```
public interface Cola {
	//Solo se definen las cabeceras de los metodos.
	//Todos los metodos se suponen public y abstract
	// (no hace falta ponerlo)

	void encolar(Object o);
	Object desencolar();
	int getTamanyo();
}
```

¿Cómo se usan?

Los interfaces se definen para que las clases los implementen. Cuando una clase implementa un interface, está obligada a implementar todos los métodos que define el interface. Si una clase implementase un interface y no tuviera todos sus métodos, la clase tendrá que ser abstracta. Además de los métodos que define el interface, la clase puede tener otros métodos.

Para indicar que una clase implementa un interface se utiliza la palabra reservada **implements.** Una clase puede más de un interface. En ese caso el nombre de los interfaces se pone tras la palabra implements separados por comas.

```
public class ListaDeEspera implements Cola {
	// Cuando una clase implementa un interface,
	// adquiere la obligacion de implementar todos
	// los metodos que el interface define

	// Si no se implementan todos, la clase tendra
	// que ser abstracta.

	private ArrayList<Persona> lista;

	//Constructor
	public void ListaDeEspera () {
		lista = new ArrayList<>();
	}
```

```
	@Override
	public void encolar(Object o) {
		//AÃ±adimos al final de la lista
		lista.add((Persona) o);
	}

	@Override
	public Object desencolar() {
		//Quitamos el primer elemento de la lista y lo devolvemos
		Object o = lista.remove(0);
		return o;
	}

	@Override
	public int getTamanyo() {
		//Devolvemos el numero de personas en la lista
		return lista.size();
	}

}
```

Podríamos decir que el interface ***Cola*** establece **"el qué"**, mientras que la clase ***ListaDeEspera,*** que implementa el interface, dice **"el como".**

Al implementar un interface, una clase ***adquiere el compromiso*** de tener implementados todos los métodos que el interface define. Este compromiso, se traduce luego en dar una mayor potencia y flexibilidad al lenguaje, ya que los interfaces se pueden utilizar como un tipo para declarar variables (tipo estático) y asignar a esas variables objetos de cualquier clase que implemente el interface. De esa forma, el polimorfismo se extiende también a los interfaces.

Interface a partir de otro interface.

Es posible definir un interfaz a partir de otro utilizando, de nuevo, la palabra ***extends***

```
public interface ColaExtendida extends Cola {
	boolean esVacia();
}
```

El interface ColaExtendida del ejemplo tendrá cuatro métodos: los tres que define el interface *Cola* y el método *esVacia()* que añade ColaExtendida. Una clase que implementase este último interface tendría que tener implementados los cuatro métodos.

Esta característica facilita la tarea de añadir métodos a los interfaces sin que afecten a clases ya desarrolladas: Supongamos por ejemplo que tenemos 5 clases que implementan *Cola* y necesitamos añadir el método *esVacia()* al interface para una sexta clase que vamos a diseñar. Si añadiéramos el método *esVacia()* al interface *Cola,* las 5 clases ya desarrolladas dejarían de compilar, ya que no tendrían implementado el nuevo método. En lugar de eso, crearíamos el nuevo interface *ColaExtendida* a partir de *Cola* (como en el ejemplo) y la nueva clase implementaría el nuevo interface.

¿Para qué sirven?

Hemos dicho que los interface sirven para ser implementados por las clases. Pero, ¿de qué sirve que las clases implementen los interfaces?

Hay varias utilidades de los interfaces:

- **Definir un API (Application Programming Interface).** Dicho de otra forma, obligar a que un conjunto de clases tengan determinados métodos. Un ejemplo claro de ello es el API JDBC de Java, que es el API que permite acceder a Bases de Datos desde Java.

La dificultad de implementar un conjunto de clases que permitan acceder a bases de datos está en la cantidad de Sistemas Gestores de Bases de Datos distintos que existen (mySql, MariaDB, SQLite, PostgreSQL, Microsoft Sql Sever, Oracle, ….). Cada uno almacena de forma distinta su información

Sería inabordable para Java implementar clases que fuesen capaces de acceder a cada uno de los tipos de BD existentes. En lugar de ello el API de Java define únicamente un conjunto de interfaces:

- Interface Connection: con los métodos connect y disconnect.
- Interface Statemment con métodos que permiten ejecutar una sentencia Select, recorrer sus resultados, etc.
- Etc …

Por otra parte, los distintos fabricantes de bases de datos proporcionan conjuntos de clases que implementan los interfaces definidos en el API JDBC de Java. Es decir, proporcionan clases con los métodos que el API de Java ha definido y que son capaces de hacer la tarea definida con su propia base de datos.

La estructura anterior permite que podamos escribir programas en Java que acceden a Base de Datos sin que tengamos que preocuparnos de a qué base de datos se va a acceder (MySql, Oracle, …) puesto que el programa Java no cambiará aunque cambie la base de datos.

- **Hacer que clases no relacionadas por herencia tengan métodos comunes** o, dicho de otra manera, **Simular herencia múltiple.**

En este sentido, trabajamos con la jerarquía de clases de animales y vimos que si queríamos que Perro y Gato tuvieran un método común (llevarALaPeluquería()), la solución correcta pasaba por definir un interface Mascota con ese método y hacer que Perro y Gato lo implementasen. Ello permitía recorrer un array con Perros y Gatos y aplicar el método, haciendo el correspondiente “casting”, como se ve en el ejemplo:

```
Animal[] misAnimales = . . .
. . .
. . .
for (int i = 0; i < misAnimales.length; i++) {
      ((Mascota) misAnimales[i]).llevarALaPelu();
}
```

- **Implementar métodos genéricos,** que son capaces de hacer su tarea con cualquier clase que implemente determinado interface. Un ejemplo de ello lo tendríamos en el interface Comparable y un método genérico de ordenación. El método que aparece a continuación es capaz de ordenar cualquier array de objetos siempre que la clase a la que pertenecen los objetos implementen el interface

```
public static void ordenar(Object[] v){
    for (int i = 0; i < v.length; i++) {
        //Buscamos el menor elemento de array
        //a partir de la posicion i
        int posMin = i;
        for (int j = i; j < v.length; j++) {
           if(((Comparable)v[j]).compareTo(v[posMin]) < 0) {
                  posMin = j;
           }
        }
        //Intercambiamos el menor con el i
        Object aux = v[i];
        v[i] = v[posMin];
        v[posMin] = aux;

    }
}
```

1.10. Sintaxis de genericidad (Notación de diamante.)

El Polimorfismo hace posible diseñar métodos y clases genéricas. Por ejemplo, el método ***ordenar*** visto anteriormente es un método genérico, ya que permite ordenar un array de cualquier tipo de objetos, siempre y cuando éstos implementen Comparable, como por ejemplo un array de Strings

```
public static void main(String[] args) {
       String[] v1 = {"b","a","d","c"};
       ordenar(v1);
}
```

En el siguiente ejemplo, se llama al método ordenar con un array de Object que contiene Strings, Integers y Figuras

```
public static void main(String[] args) {
       String[] v1 = {new Integer(8),"a","d","c", new Figura(10,10,"Rojo"};
       ordenar(v1);
}
```

Este último programa tiene dos **problemas**:

- Al ejecutarlo **se producirá error de ejecución** porque, aunque todos los objetos que hay en el array implementan el interface Comparable, ninguno de los métodos compareTo de sus clases son capaces de comparar con un objeto que no sea de su propio tipo, es decir, el compareTo de String es capaz de comparar con otro String, pero no con un Integer o una Figura.
- **No se produce error de compilación**, ya que la llamada es correcta. El método ordenar espera recibir un array de Object y está recibiendo un array de Object.

Cuando un programa crece en complejidad, es fácil que éstos contengan errores. Lo ideal sería que el compilador nos avisara de todos los errores que contiene el programa, pero esto no es posible y hay errores que se producen al ejecutar los programas. Los errores de ejecución son más difíciles de detectar y corregir porque a menudo se producen cuando el programa lleva ya mucho tiempo usándose y generalmente no está claro cuál es la parte del código que origina el error.

La incorporación de la sintaxis de genericidad (notación de diamante) permite que errores que, como en ejemplo anterior, se detectan al ejecutar, sean detectados por el compilador y el programador los pueda corregir más rápidamente.

Clases genéricas

Una clase genérica es una clase que trabaja con uno o varios tipos de datos, sin tener que especificar cuáles son esos tipos de datos cuando se diseña la clase.

En el siguiente ejemplo, la clase Equipo permite representar un Equipo de elementos de determinado tipo, por ejemplo un Equipo<Futbolista> o un Equipo<Baloncestista>. Un Equipo tiene un nombre, una serie de miembros y un capitán. Los miembros y el capitán del equipo serán objetos del mismo tipo, pero ese tipo está "abierto". Para ello, se indica que el tipo T al que pertenecen los miembros y el capitán son un tipo genérico que está por determinar **<T>**

```
public class Equipo <T> {
	private String nombre;
	private ArrayList<T> miembros;
	private T capitan;

	public Equipo(String nombre){
		this.nombre = nombre;
		miembros = new ArrayList<>();
	}
	public void anyadir(T f){
		if(!miembros.contains(f)){
			miembros.add(f);
		}
	}
	public void quitar(T f){
		if(capitan.equals(f)){
			capitan = null;
		}
		miembros.remove(f);
	}
	public String toString(){
		return "Equipo: " + nombre + "\n" +
				"Capitan: " + capitan + "\n" +
				"Miembros: " + miembros;
	}

	public void setCapitan(T f){
		capitan = f;
	}
}
```

El tipo que sustituye a T se indicará cada vez que se cree un objeto de la clase Equipo

```
Equipo <Futbolista> eq1 = new Equipo("Valencia CF");
```

Además, el tipo que sustituye a T se puede restringir, usando la palabra reservada ***extends***

```
public class Equipo <T extends Comparable<T>> {
	...
}
```

En este caso, no cualquier tipo puede sustituir a T. Si se intentase crear un Equipo de alguna clase que no implemente Comparable, se producirá error de compilación.

Interfaces genéricos

De la misma forma, se pueden definir interfaces genéricos, como por ejemplo el Interface Comparable<T> que ya se encuentra definido en Java

```
public interface Comparable <T> {
	int compareTo(T o);
}
```

De esta forma, si decimos que la clase Figura implementa Comparable<Figura>, estamos indicando que el tipo T que aparece en el interface Comparable será ***Figura*** y por tanto, la clase Figura tendrá que tener un método compareTo como el que aparece a continuación:

```
public class Figura implements Comparable <Figura> {

	public int compareTo(Figura o){
		. . .
		. . .
	}
}
```

Si se intenta comparar una Figura con un objeto que no sea una figura, se producirá error de compilación y esto es gracias a la sintaxis de genericidad.

Métodos genéricos

También es posible usar la sintaxis de genericidad en métodos. El método ***ordenar*** visto anteriormente es genérico, pero no usaba la notación de diamante. Incorporando ésta quedaría de la siguiente manera:

```
public static <T extends Comparable<T>> void ordenar(T[] v){
    for (int i = 0; i < v.length; i++) {
        //Buscamos el menor elemento de array
        //a partir de la posicion i
        int posMin = i;
        for (int j = i; j < v.length; j++) {
           if((v[j]).compareTo(v[posMin]) < 0) {
                  posMin = j;
           }
        }
        //Intercambiamos el menor con el i
        Object aux = v[i];
        v[i] = v[posMin];
        v[posMin] = aux;

    }
}
```

Observa como las referencias a ***Object*** han desaparecido y, además, ya no es necesario hacer un “casting” a Comparable.

Java deducirá cual es el tipo T de la llamada que se haga: si se llama al método con un array de String, deducirá que T es String; si se llama al método con un array de Integer, deducirá que T es Integer, etc…

Con este nuevo método, la llamada que se hace a continuación, produciría error de . compilación, ya que al llamar con un array que contiene Strings, Integers y Figura, Java deduce que T es Object, y Object no implementa Comparable. Java detecta el error.

```
public static void main(String[] args) {
        String[] v1 = {new Integer(8),"a","d","c", new Figura(10,10,"Rojo"};
        ordenar(v1);
}
```

1.11. El modificador final.

El modificador ***final*** tiene un significado distinto dependiendo de a qué se aplique:

- Un **atributo final**, es una constante, es decir, un atributo cuyo valor no se puede modificar:
- Un **método final** es un método que no se podrá reescribir en las subclases.
- Una **clase final** es una clase de la que no se puede heredar. Es decir, no se pueden hacer subclases de una clase final.