

# Capítulo 11

## Clases abstractas e Interfaces

# Objetivos

- Diseñar y usar clases abstractas.
- Generalizar las clases contenedoras numéricas, BigInteger y BigDecimal usando la clase numérica abstracta.
- Procesar un calendario usando las clases Calendar y GregorianCalendar.
- Especificar el comportamiento común de los objetos que utilizan interfaces.
- Definir interfaces y definir clases que implementan interfaces.
- Explorar las similitudes y diferencias entre clases concretas, clases abstractas e interfaces.
- Definir un orden natural usando la interfaz Comparable.
- Hacer objetos clonables usando la interfaz Cloneable.

# Clases abstractas

Un método definido en una clase, pero cuya implementación se delega en las subclases, se conoce como abstracto.

Es normal encontrarnos con un método que sabemos que deben tener todas las clases y su subclases, pero que solo podemos implementar conociendo el tipo específico, es decir, la subclase a la que pertenece.

Para declarar un método abstracto se antepone el modificador *abstract* y se declara el prototipo, sin escribir el cuerpo de la función.

Ejemplo

```
abstract void mostrarDatos();
```

Las subclases deberán implementar el método cada una con sus particularidades específicas de la clase.

# Clases abstractas

Toda clase que tiene un método abstracto debe ser declarada *abstract*.

Las clases abstractas no son instanciables, no se pueden crear objetos de esa clase. Existen solo para ser heredadas por otras.

Si una clase que hereda de una abstracta deja algún método abstractos sin implementar será abstracta también. Una clase abstracta puede tener algún método implementado y algunos atributos definidos, que serán heredados.

También podemos definir una clase abstracta que no contenga ningún método abstracto, se usa como clase base para definir otra subclase.

# Ejemplo Clases abstractas

Definimos una clase abstracta A, con un atributo a, un método implementado metodo1 y un método abstracto metodo2.

```
//clase abstracta
abstract class A {
    int a = 1;

    void metodo1() { //método implementado
        System.out.println("método1 definido en A");
    }

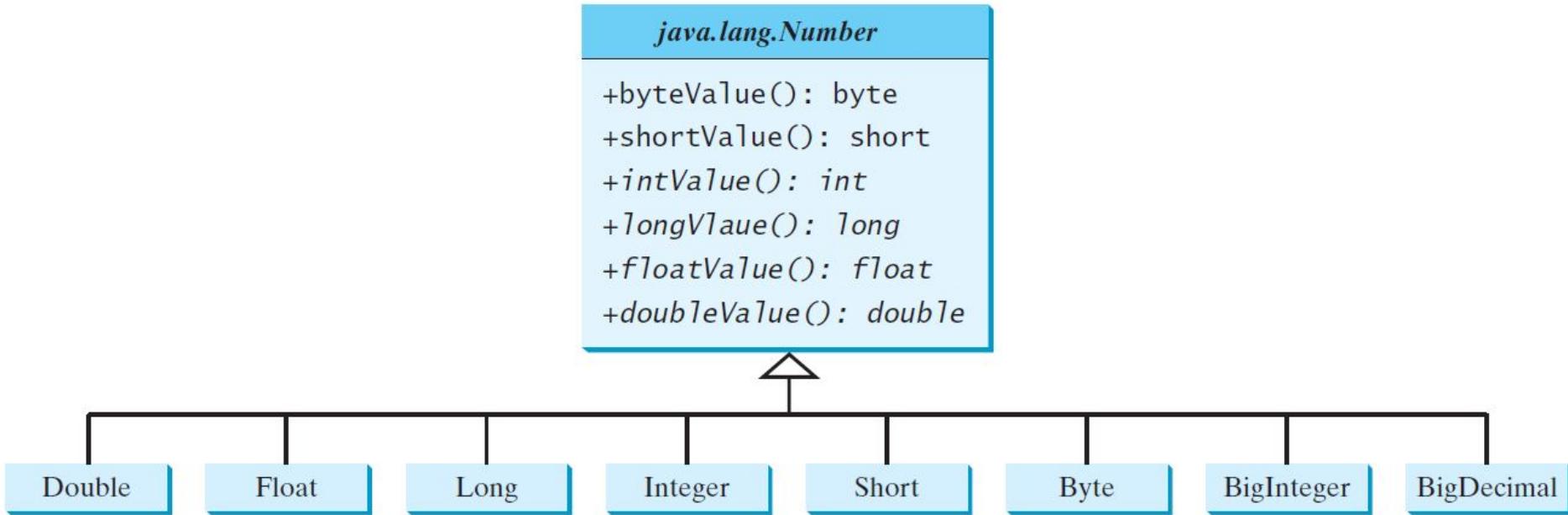
    abstract void metodo2(); //método abstracto para ser implementado
                            //por las subclases
}
```

Definimos las subclases B y C, en cada una de las cuales implementaremos el metodo2. Ya podemos instanciar las clases B y C y ejecutar ambos métodos metodo1 y metodo2 de cada una de ellas.

# Ejemplo Clases abstractas

```
class B extends A {  
    void metodo2() {  
        System.out.println("método2 definido en B");  
    }  
}  
  
class C extends A {  
    void metodo2() {  
        System.out.println("método2 definido en C");  
    }  
}  
  
B b = new B();  
C c = new C();  
System.out.println("Valor de a en la clase B: " + b.a); //heredado de A  
b.metodo1();  
b.metodo2();  
c.metodo1();  
c.metodo2();
```

# La clase abstracta Number



Los métodos `intValue()`, `longValue()`, `floatValue()` y `doubleValue()` se definen como métodos abstractos en la clase `Number`. La clase de `Number` es por lo tanto una clase abstracta.

# La clase abstracta Calendar y su subclase GregorianCalendar

## java.util.Calendar

```
#Calendar()  
+get(field: int): int  
+set(field: int, value: int): void  
+set(year: int, month: int,  
     dayOfMonth: int): void  
+getActualMaximum(field: int): int  
+add(field: int, amount: int): void  
+getTime(): java.util.Date  
  
+setTime(date: java.util.Date): void
```

Constructs a default calendar.

Returns the value of the given calendar field.

Sets the given calendar to the specified value.

Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.

Returns the maximum value that the specified calendar field could have.

Adds or subtracts the specified amount of time to the given calendar field.

Returns a **Date** object representing this calendar's time value (million second offset from the UNIX epoch).

Sets this calendar's time with the given **Date** object.



## java.util.GregorianCalendar

```
+GregorianCalendar()  
+GregorianCalendar(year: int,  
                   month: int, dayOfMonth: int)  
+GregorianCalendar(year: int,  
                   month: int, dayOfMonth: int,  
                   hour:int, minute: int, second: int)
```

Constructs a **GregorianCalendar** for the current time.

Constructs a **GregorianCalendar** for the specified year, month, and date.

Constructs a **GregorianCalendar** for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

# La clase GregorianCalendar

java.util.Calendar es una clase base abstracta para extraer información detallada, como año, mes, fecha, hora, minuto y segundo desde un objeto Date. Las subclases de Calendar pueden implementar sistemas de calendario específicos, como el calendario gregoriano.

Podemos usar new GregorianCalendar() para construir un objeto GregorianCalendar predeterminado con la hora actual y usar new GregorianCalendar (año, mes, fecha) para construir un objeto GregorianCalendar con el año, mes y fecha especificados.

El método get(int field) definido en la clase Calendar es útil para extraer la información de fecha y hora de un objeto Calendar.

# La clase GregorianCalendar

<i>Constant</i>	<i>Description</i>
<code>YEAR</code>	The year of the calendar.
<code>MONTH</code>	The month of the calendar, with 0 for January.
<code>DATE</code>	The day of the calendar.
<code>HOUR</code>	The hour of the calendar (12-hour notation).
<code>HOUR_OF_DAY</code>	The hour of the calendar (24-hour notation).
<code>MINUTE</code>	The minute of the calendar.
<code>SECOND</code>	The second of the calendar.
<code>DAY_OF_WEEK</code>	The day number within the week, with 1 for Sunday.
<code>DAY_OF_MONTH</code>	Same as DATE.
<code>DAY_OF_YEAR</code>	The day number in the year, with 1 for the first day of the year.
<code>WEEK_OF_MONTH</code>	The week number within the month, with 1 for the first week.
<code>WEEK_OF_YEAR</code>	The week number within the year, with 1 for the first week.
<code>AM_PM</code>	Indicator for AM or PM (0 for AM and 1 for PM).

# Ejercicios clases abstractas

## **Ejemplo 1: Sistema de Vehículos**

Diseña un programa que modele un sistema de vehículos. Cada vehículo tiene un atributo marca y un método para calcular el consumo de combustible dada una distancia, pero el cálculo depende del tipo de vehículo (automóvil 8l/100km, motocicleta 5l/100km). Usa clases abstractas para implementar esta funcionalidad.

## **Ejemplo 2: Empleados en una Empresa**

Diseña un sistema para calcular el salario de diferentes tipos de empleados. Los empleados tienen un atributo nombre y pueden ser de tipo a tiempo completo o parcial por horas. El cálculo del salario depende del tipo de empleado:  
t. completo (ej salario=1500€) y por horas(ejemplo tarifa=10€, nHora=176h).  
Usa clases abstractas para implementar esta funcionalidad.

# Interfaces

¿Qué es una interface?

¿Por qué es útil una interface?

¿Cómo defines una interface?

¿Cómo se usa una interface?

# ¿Qué es una interface?

Una interfaz es una construcción de clase que contiene sólo constantes y métodos abstractos. En muchos sentidos, una interfaz es similar a una clase abstracta, pero la función de una interfaz es especificar un comportamiento común para los objetos.

Por ejemplo, podríamos especificar que los objetos sean comparables, comestibles y clonables utilizando las interfaces apropiadas.

Una interfaz contiene únicamente la cabecera de una serie de métodos (opcionalmente también puede contener constantes).

La interfaz no especifica el “cómo” ya que no contiene el cuerpo de los métodos, sólo el “qué”.

Cada interfaz puede tener varias implementaciones asociadas.

# Definir una interface

Java usa la siguiente sintaxis general para definir una interfaz:

```
tipoDeAcceso interface NombreInterfaz {  
    //atributos, son public, static y final por defecto:  
    tipo atributo1 = valor1;  
    ... //otros atributos  
  
    //métodos abstractos, sin implementar:  
    tipo metodo1(listaPar1);  
    ... //otros métodos abstractos  
  
    //métodos static, implementados:  
    static tipo metodoEstatico1(listaPar1) {  
        //cuerpo del método estático 1  
    }  
    ... //otros métodos estáticos  
  
    //métodos por defecto, implementados:  
    default tipo metodoDefault1(listaPar1) {  
        //cuerpo de metodoDefault1  
    }  
    ... //otros métodos por defecto  
}
```

# Definir una interface

Si se omite tipoDeAcceso, el acceso de la interfaz está restringido al paquete en el que está incluida. Si es public, la interfaz podrá ser importada desde otro paquete mediante import.

Para que una clase implemente una interfaz, debe declararla en el encabezamiento, usando la palabra clave *implements*, e implementar todos los métodos abstractos de la interfaz en el cuerpo de la clase.

```
tipoDeAcceso class NombreClase implements nombreInterfaz {  
    ...  
    public tipo metodoAbstracto1(listaPar1) {  
        ... //cuerpo del método 1  
    }  
    ... //otros métodos abstractos  
}
```

# Definir una interface

Una clase puede implementar más de una interfaz, escribiéndolas separadas por comas, en el encabezamiento, e implementar todos los métodos abstractos de cada una de ellas.

```
tipoDeAcceso class nombreClase implements Interfaz1, Interfaz2, ... {  
    ...  
}
```

Una clase implementa una interfaz si tiene implementados todos sus métodos abstractos. Si se deja alguno sin implementar, la clase debe ser abstracta.

Una interfaz también puede heredar de otra u otras, se define:

```
interface nombreInterfaz extends superIntefaz1, superInterfaz2, ... {  
    ...  
}
```

# Interface es una clase especial

Una interfaz se trata como una clase especial en Java. Cada interfaz se compila en un archivo bycode separado, al igual que una clase normal. Desde Java 8, puede también contener métodos por defecto (default) y métodos estáticos (static).

Como con una clase abstracta, no podemos crear una instancia desde una interfaz utilizando el operador new, pero en la mayoría de los casos podemos usar una interfaz más o menos de la misma manera que usamos una clase abstracta.

Por ejemplo, podemos usar una interfaz como un tipo de datos para una variable, como resultado de un casting, y así sucesivamente

# Omitir modificadores en interfaces

Todos los campos de datos son public final static y todos los métodos son public abstract en una interfaz. Por esta razón, estos modificadores se pueden omitir, como se muestra a continuación:

```
public interface T1 {  
    public static final int K = 1;  
  
    public abstract void p();  
}
```

Equivalent

```
public interface T1 {  
    int K = 1;  
  
    void p();  
}
```

Se puede acceder a una constante definida en una interfaz usando la sintaxis:

InterfaceName.CONSTANT\_NAME (ej., T1.K).

# Interfaces vs. Clases Abstractas

La diferencia es que una clase sólo puede heredar de una clase abstracta, mientras que puede implementar más de una interfaz. Las clases abstractas están para ser heredadas por otras clases, mientras que las interfaces son implementadas por las clases.

Al igual que una clase, una interfaz también define un tipo. Una variable de un tipo de interfaz puede hacer referencia a cualquier instancia de la clase que implementa la interfaz. Si una clase extiende una interfaz, esta interfaz juega el mismo papel que una superclase.

Puede usar una interfaz como un tipo de datos y convertir una variable de un tipo de interfaz en su subclase, y viceversa.

# Interfaces vs. Clases Abstractas

En una interfaz, los datos deben ser constantes; una clase abstracta puede tener todo tipo de datos.

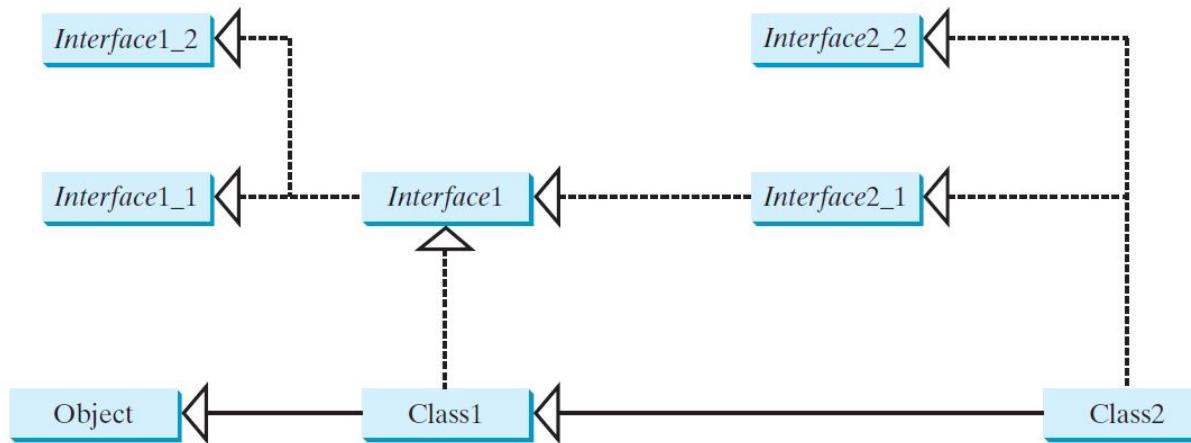
Cada método en una interfaz tiene solo una firma sin implementación; una clase abstracta puede tener métodos concretos.

	Variables	Constructores	Métodos
Clases Abstractas	Sin Restricciones	Los constructores son invocados por subclases mediante el encadenamiento de constructores. No se puede crear una instancia de una clase abstracta con el operador new.	Sin restricciones
Interface	Todas las variables deben ser <b>public static final</b>	Sin constructores. No se puede crear una instancia de una interfaz utilizando el operador new.	Todos los métodos deben ser <b>public abstract</b>

Todas las clases comparten una única raíz, la clase Object, pero no hay una única raíz para las interfaces.

# Interfaces vs. Clases Abstractas

Al igual que una clase, una interfaz también define un tipo. Una variable de un tipo de interfaz puede hacer referencia a cualquier instancia de la clase que implementa la interfaz. Si una clase extiende una interfaz, esta interfaz juega el mismo papel que una superclase. Puede usar una interfaz como un tipo de datos y convertir una variable de un tipo de interfaz en su subclase, y viceversa.



# Cuando usar una interfaz o una clase

Las clases e interfaces abstractas se pueden usar para modelar características comunes. ¿Cómo decides usar una interfaz o una clase? Por su sencillez se recomienda utilizar interfaces siempre que sea posible. Si la clase debe incorporar atributos, o resulta interesante la implementación de alguna de sus operaciones, entonces declararla como abstracta.

En general, una relación fuerte es una relación que describe claramente una relación padre-hijo, debe modelarse utilizando clases. Por ejemplo, un miembro del personal es una persona.

Una relación débil, es un tipo que indica que un objeto posee una cierta propiedad. Una relación débil es una relación que se puede modelar utilizando interfaces.

# Conflicto de interfaces

En raras ocasiones, una clase puede implementar dos interfaces con información que provoquen conflicto entre sí (por ejemplo, dos mismas constantes con diferentes valores o dos métodos con la misma firma pero diferente tipo de devolución). Este tipo de errores serán detectados por el compilador.

# Ejercicios de clases abstractas e interfaces

## Ejercicio 3: Sistema de Transporte

Crea un sistema que modele diferentes medios de transporte (autos y bicicletas). Define una clase abstracta Transporte que contenga atributos comunes como velocidad y capacidad, y métodos para moverse. Los autos tiene el atributo tipoCombustible y bicicleta el atributo tipo. Usa una interfaz Motorizado para vehículos que usen un motor, con los métodos encenderMotor() y apagarMotor(). Crea distintos objetos y prueba cada uno de los métodos.

## Ejercicio 4: Sistema de Dispositivos Electrónicos

Crea un sistema que modele diferentes dispositivos electrónicos. Define una clase abstracta Dispositivo con atributos comunes como marca y modelo, y métodos para encender. Usa una interfaz Conectividad para dispositivos que puedan conectarse a Internet, con los métodos conectarInternet() y desconectarInternet(). Crea distintos objetos y prueba cada uno de los métodos.

# Tipos de interfaces de la API

Hay algunas operaciones muy frecuentes y necesarias, que forman parte de las interfaces de la API. Algunos ejemplos:

- Para que un objeto sea ordenable, su clase debe implementar ***Comparable*** o ***Comparator***, que nos van a facilitar las operaciones de búsqueda u ordenación de objetos mediante la comparación de valores.
- Para que un objeto sea duplicable, su clase debe implementar ***Cloneable***.
- Para que un objeto pueda ser guardado en un fichero la clase debe implementar la interfaz ***Serializable***.

# Tipo Interface Comparable. Orden natural

La *interfaz Comparable* sirve para definir el orden natural de los objetos de una determinada clase y funciona de la siguiente manera:

La clase cuyo orden natural debe implementar la interfaz *Comparable*, y el único método abstracto que define la interfaz es:

```
int compareTo(Object o);
```

Realiza la comparación entre this (el objeto actual) y el objeto o, de forma que devuelva un número negativo si this es menor (anterior) a o, positivo si this es mayor (posterior) a o, y cero si son iguales.

Ejemplo: Si queremos que los objetos de la clase Persona se ordenen por su edad (atributo entero). Al ser el id numérico, la implementación del método se simplifica:

```
int compareTo(Object o) {  
    return edad - ((Persona)o).edad;  
}
```

# Tipo Interface Comparable. Orden natural

Ejemplo: Si queremos que los objetos de la clase *Persona* se ordenen por su campo apellidos (atributo String).

```
class Persona implements Comparable{
    String nombre;
    String apellidos;
    double estatura;
    Alumno(String nombre, String apellidos, double estatura){
        this.nombre=nombre;
        this.apellidos=apellidos;
        this.estatura=estatura;
    }
    public int compareTo(Object o) {
        return this.apellidos.compareTo(((Alumno)o).apellidos);
    }
}
```

# Tipo Interface Comparable. Orden natural

Muchas de las clases definidas en la API Java disponen de un orden natural a través de este mecanismo. Los números son comparables, las cadenas son comparables, y también lo son las fechas. Podemos usar el método compareTo para comparar dos números, dos cadenas y dos fechas. Por ejemplo:

```
1 System.out.println(new Integer(3).compareTo(new Integer(5)));
2 System.out.println("ABC".compareTo("ABE"));
3 java.util.Date date1 = new java.util.Date(2013, 1, 1);
4 java.util.Date date2 = new java.util.Date(2012, 1, 1);
5 System.out.println(date1.compareTo(date2));
```

Existen métodos de ordenación en distintas clases de la API estándar Java que usan el método compareTo para ordenar los objetos. Es el caso de Arrays.sort() y Collections.sort().

Por ejemplo, si tenemos un array de alumno, se ordenarán por su orden natural  
Arrays.sort(personas);

# Tipo Interface Comparator. Orden alternativo

La *interfaz Comparator* sirve para definir un orden alternativo de los objetos de una determinada clase, por ejemplo un orden alfabético por apellidos, etc. Dicha interfaz debe contener obligatoriamente el método compare:

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

El método compare() toma los dos objetos a comparar y debe devolver un número negativo si el primero va antes que el segundo en nuestro orden alternativo, positivo si va después y cero si son iguales.

Ejemplo: Ordenar personas alfabéticamente por apellidos y nombre:

```
int compare(Object o1, Object o2) {  
    Persona p1 = (Persona)o1;  
    Persona p2 = (Persona)o2;  
    return (p1.getApellidos() + p1.getNombre()).compareTo(p2.getApellidos() +  
        p2.getNombre());  
}
```

# Tipo Interface Comparator. Orden alternativo

Igual que ocurría con el orden natural, podemos usar métodos de la API estándar para ordenar los objetos de un array o colección usando un orden alternativo. Para ello existen versiones sobrecargadas de los métodos sort() que toman un parámetro extra de tipo Comparator:

```
Arrays.sort(personas, new ComparadorApellidosNombre());
```

# Tipo Interface Cloneable

A menudo es deseable crear una copia de un objeto. Para hacer esto, necesita usar el método de clonación y comprender la interfaz Cloneable.

Una interfaz contiene constantes y métodos abstractos, pero la interfaz Cloneable es un caso especial. La interfaz Cloneable en el paquete java.lang se define de la siguiente manera:

```
package java.lang;  
public interface Cloneable {  
}
```

Esta interfaz está vacía. Una interfaz con un cuerpo vacío se conoce como una **interfaz de marcador**. Una interfaz de marcador no contiene constantes o métodos. Se usa para denotar que una clase posee ciertas propiedades deseables. Una clase que implementa la interfaz Clonable se marca como clonable y sus objetos se pueden clonar utilizando el método clone() definido en la clase Object.

# Ejemplo Interface Cloneable

Muchas clases en la biblioteca Java implementan Cloneable (ej., Date and Calendar). Por lo tanto, las instancias de estas clases se pueden clonar. Por ejemplo, el siguiente código:

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();
System.out.println("calendar == calendarCopy is " +
(calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
calendar.equals(calendarCopy));
```

el resultado que nos muestra es:

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

# Ejemplo Implementación Interface Cloneable

Pero podemos hacer que nuestra clase implemente Cloneable:

```
class Persona implements Cloneable{  
    ...  
    @Override // Sobrescribir el método clone  
    protected Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

Por lo tanto, las instancias de esta clase ya se pueden clonar, en el Main incluimos el código para probarlo:

```
try {  
    // Crear un objeto Persona  
    Persona personal1 = new Persona("Juan", "Perez", 1.80);  
    System.out.println("Original: " + personal1);
```

# Ejemplo Implementación Interface Cloneable

```
// Crear un objeto Persona  
Persona personal = new Persona("Juan", "Perez", 1.80);  
System.out.println("Original: " + personal);  
// Clonar el objeto Persona  
Persona personaClonada = (Persona) personal.clone();  
System.out.println("Clonada: " + personaClonada);  
// Modificar el objeto clonado  
personaClonada.setNombre("Pedro");  
personaClonada.setEstatura(1.75);  
// Mostrar los cambios  
System.out.println("Clonada: " + personaClonada);  
}catch (CloneNotSupportedException e) {  
    System.err.println("Error al clonar: " + e.getMessage());  
}
```

# Tipos parametrizados o genéricos

El uso de tipos genéricos permite disponer de clases, interfaces o métodos que se puedan usar con muchos tipos de datos distintos. Ejemplos importantes son los métodos de comparación, `compareTo()` y `compare()`, de las interfaces `Comparable` y `Comparator` respectivamente. Permitiendo comparar objetos de cualquier clase.

```
class Persona implements Comparable<Persona>{  
    ...  
    public int compareTo(Persona o){  
        return apellidos.compareTo(o.apellidos);  
    }  
}
```

```
class ComparaNombres implements Comparator<Persona>{  
    public int compare(Persona o1, Persona o2){  
        return o1.nombre.compareTo(o2.nombre);  
    }  
}
```

# Ejercicios Tipos de interfaces de la API

## **Ejercicio 5: Uso de la Interfaz Comparable**

Crea un sistema que modele un producto con un precio y un nombre.

Implementa la interfaz Comparable para comparar dos productos por su precio.

## **Ejercicio 6: Uso de la Interfaz Comparator**

Crea un sistema que modele una lista de objetos Empleado, cada uno con un nombre y un salario. Implementa la interfaz Comparator para ordenar empleados de diferentes maneras:

1. Por salario de menor a mayor.
2. Por nombre en orden alfabético.

# Ejercicios Tipos de interfaces de la API

## **Ejercicio 7: Implementación y Uso de Interfaz Comparable**

Haz la implementación de un programa en Java haciendo que la clase Alumno, con los atributos nif(String), nombre(String), y nota(double), se ordene de forma natural por su campo nif (atributo String).

Pruébalo mostrando los datos de los alumnos, previamente cargado en un array de tamaño 3, antes y después de ordenarlos.

## **Ejercicio 8: Implementación y Uso de Comparable y Comparator**

Modifica el ejercicio anterior para incluir las ordenaciones de los alumnos por nombre y por nota descendente.

Pruébalo mostrando los datos de los alumnos de las distintas ordenaciones.

# Ejercicios Tipos de interfaces de la API

## **Ejercicio 9: Implementación y Uso de la Interfaz Cloneable**

Modifica el ejercicio anterior sobre la clase Alumno para que implemente la interfaz Cloneable para crear copias exactas y pruébalo cambiando el tamaño del array a 4, clonando por ejemplo el tercer alumno en la última posición, cámbiale los datos al clon y muestralos a posteriori.

## **Ejercicio 10: Implementación y Uso de la Interfaz Cloneable**

Crea un sistema que permita clonar objetos de tipo Empleado, que tienen un nombre y un salario. Implementa la interfaz Cloneable para crear copias exactas.