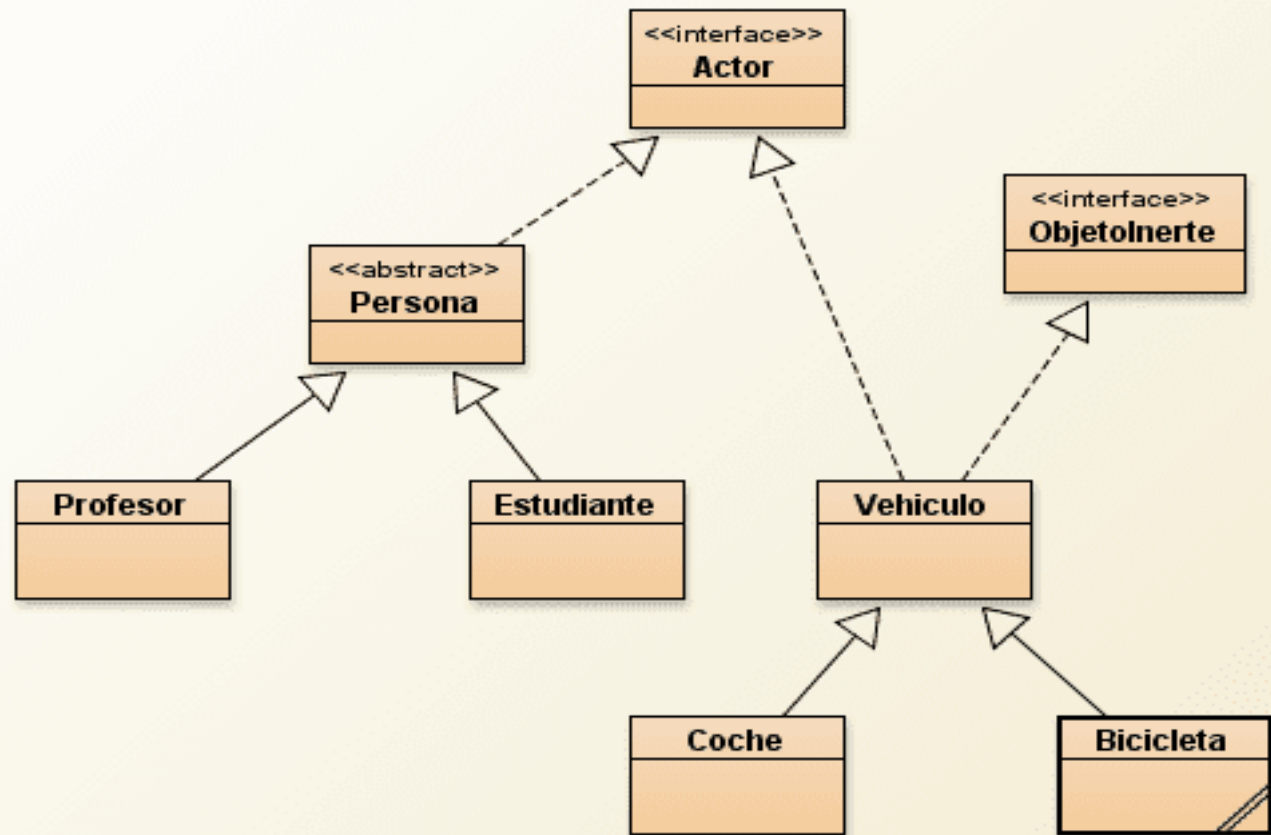


UT07. INTERFACES Y CLASES INTERNAS

Índice

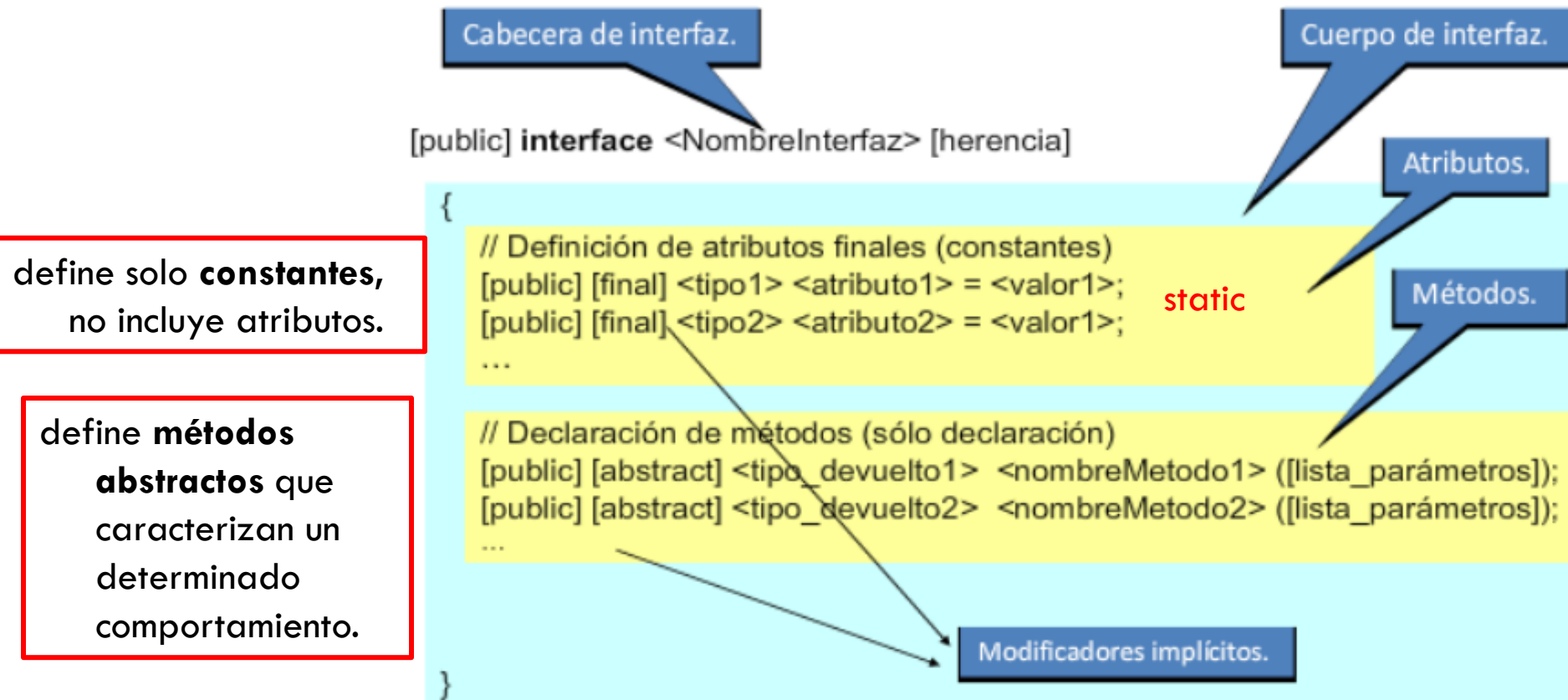
- 1.- Interfaces
- 2.- Interfaces vs Clases abstractas
- 3.- Métodos por defecto y privados
- 4.- Polimorfismo con interfaces
- 5.- Ordenación y comparación de objetos
- 6.- Clases anidadas
 - 6.1.- Estáticas
 - 6.2.- Internas
 - 6.3.- Locales
 - 6.4.- Anónimas



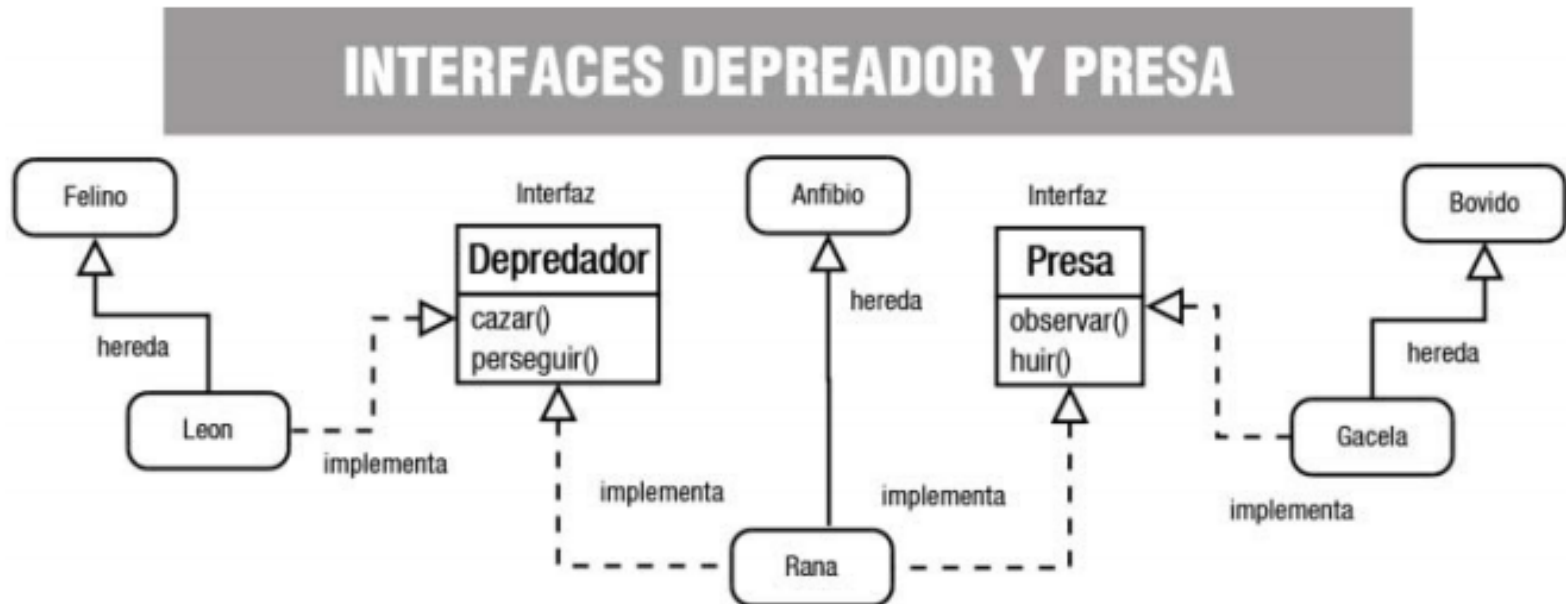
1. Interfaces en Java

Interfaces en Java

- Es una colección de métodos abstractos y atributos constantes en las que se especifica **qué se debe hacer pero no cómo**.



Interfaces en Java



- Una **Rana** **es un Anfibio** y por tanto **heredará** de dicha clase pero algunas acciones que queremos sea capaz de hacer tienen que ver con que **se comporta como un Depredador** (observar una presa, perseguirla, comérsela, ...) y **se comporta como una Presa**, es decir **Rana** implementa comportamientos de la interface **Depredador** y **Presa**.

Interfaces en Java

- Mecanismo para definir **tipos de datos** sin implementación.
- Las interfaces serán **implementadas** por clases:
 - Usaremos **implements** en lugar de **extends**.
 - Una clase debe **implementar todos los métodos de la interface** o la clase será declarada como **abstracta**.
- Una **interface** establece unos **comportamientos** o capacidades (métodos) **pero no cómo deben ser** (implementación) ni lo que el objeto es. Es por ello que las interfaces suelen llevar el sufijo **-able, -or, -ente ...**

Interfaces en Java

- Una interface puede ser declarada **public** → el fichero .java se llamará como la interface.
- Si no se especifica modificador de acceso la visibilidad de la interface será “**package**”.
- Se genera un fichero .class para cada interface.
- Todos los miembros de la interface son **public** de manera **implícita**:
 - Los atributos son **public static final** y hay que darles valor inicial (¡No hay variables de instancia!)
 - Los métodos son **public abstract**.
- La clase que implemente una interface deberá declarar todos los métodos public o el compilador se quejará.

Interfaces en Java

```
public interface Atrapable {  
    int TIEMPO_RETENCION = 1000;  
  
    void atrapar();  
    void liberar();  
  
}
```

La declaración de la constante
puede ser simplificada
Así como la de los métodos

```
public class Burbuja extends Juego implements Atrapable{  
  
    @Override  
    public void atrapar() { ... }  
    @Override  
    public void liberar() { ... }  
  
}
```

Los métodos deben ser
redefinidos como públicos o el
compilador se quejará

Interfaces vs Clases

- Tienen en **común**:
 - Pueden contener métodos.
 - Se escriben en un fichero .java con el nombre de la clase o interface.
 - Cada fichero .java genera un fichero .class al compilar.
- Se **diferencian** en:
 - No se pueden instanciar.
 - No tienen constructores.
 - Solamente pueden contener atributos **public static final**
 - Una **clase solo hereda de otra clase** (**herencia simple**) mientras que una **interface puede heredar de más de una interface** (**herencia múltiple**). **Ojo!!** Las interfaces solo extienden interfaces.
- Una clase **puede implementar varias interfaces** además de heredar de una clase.

Interfaces en Java


- Una interface puede hacer que **dos clases tengan un mismo comportamiento** independientemente de la jerarquía de clases a la que pertenezcan.
- Una **interface puede ser implementada por cualquier clase**, permitiendo que clases sin relación alguna puedan compartir un determinado comportamiento sin tener que forzar una relación de herencia.

Interfaces en Java

- Las interfaces tienen su propia jerarquía, diferente e independiente de la jerarquía de clases.

```
public interface Atrapable {  
    public abstract void atrapar();  
}  
  
public interface Puntuable {  
    public abstract void puntuar();  
}  
  
public interface Trazable extends Atrapable, Puntuable {  
    public abstract void trazar();  
}
```

Ojo!! Una interfaz puede
“extender” cualquier
número de interfaces!!!



```
public class Juego implements Trazable {  
  
    public void atrapar() { ... }  
    public void puntuar() { ... }  
    public void trazar() { ... }  
}
```

Interfaces en Java

- Las interfaces son útiles porque resuelven:
 - El problema de la **herencia simple**: una clase puede implementar varias interfaces.
 - La **comprobación estricta de tipos de Java**: un objeto puede ser convertido ascendentemente (upcast) a varios tipos base, uno por cada interface implementada → uso de **instanceof** para comprobar el tipo de una variable de referencia.

Interface en Java. Ejemplo

File DeDos.java

```
public class DeDos implements Series{
    int inicio;
    int valor;
    public DeDos () {
        inicio = 0;
        valor = 0;
    }
    @Override
    public int siguiente() {
        return valor+=2;
    }
    @Override
    public void reiniciar() {
        valor = inicio;
    }
    @Override
    public void comenzar(int ini) {
        valor = ini;
        inicio=ini;
    }
}
```

File Series.java

```
public interface Series {
    int siguiente();
    void reiniciar();
    void comenzar(int ini);
}

public static void main(String[] args) {
    DeDos obj = new DeDos();
    for(int i=0;i<5;i++)
        System.out.print(obj.siguiente()+" ");
    obj.reiniciar();
    System.out.println("");
    for(int i=0;i<5;i++)
        System.out.print(obj.siguiente()+" ");
    obj.comenzar(10);
    System.out.println("");
    for(int i=0;i<5;i++)
        System.out.print(obj.siguiente()+" ");
}
```

Interface en Java. Ejercicio resuelto

- **Ejercicio:** define la interface **Imprimible** que proporcione varios métodos que permitan mostrar el contenido de una clase:
 - **devolverContenidoString**, que devuelve un String con todo el contenido de la clase. El formato será una lista de pares “nombre=valor” para cada atributo, separados por comas y todo entre llaves.
 - **devolverContenidoArrayList** que cree y devuelva un arraylist de String con todo el contenido del objeto. El formato será una lista de pares “nombre=valor” para cada atributo, separados por comas y todo entre llaves

Interface en Java. Ejercicio resuelto

- La definición de la interface es la siguiente:

```
public interface Imprimible {  
    String devolverContenidoString();  
    ArrayList devolverContenidoArrayList();  
}
```

- Ahora cualquier clase que necesite emplear estos métodos puede implementar esta interface.

Interface en Java. Ejercicio resuelto

- Supongamos la clase **Persona** que implementa **Imprimible**:

```
public abstract class Persona implements Imprimible{  
    private String nombre;  
    private int edad;  
    private LocalDate fechaNac;  
    public Persona(String nombre, LocalDate fecha){  
        this.nombre = nombre;  
        this.fechaNac=fecha;  
        this.edad=Period.between(fechaNac,  
LocalDate.now()).getYears();  
    }  
}
```

```
@Override  
public String devolverContenidoString(){  
    return "nombre="+nombre+", fecha de  
nacimiento="+fechaNac +", edad "+edad;  
}
```

```
@Override  
public ArrayList devolverContenidoArrayList(){  
    ArrayList contenido = new ArrayList();  
    contenido.add("nombre="+this.nombre);  
    contenido.add("fechaNac="+fechaNac);  
    contenido.add("edad="+this.edad);  
    return contenido;  
}
```


Interface en Java. Ejercicio resuelto

```
public class Alumno extends Persona{
    private String curso;
    private final String nre;

    public Alumno(String nombre, LocalDate fechaNac,
String nre, String curso){
        super(nombre,fechaNac);
        this.nre=nre;
        this.curso=curso;
    }
    public String getCurso() {
        return curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
    public String getNre() {
        return nre;
    }
}
```

```
ArrayList devolverContenidoArrayList(){
    ArrayList contenido =
super.devolverContenidoArrayList();
    contenido.add("NRE="+this.nre);
    contenido.add("curso="+this.curso);
    return contenido;
}
}
```

Alumno al heredar de **Persona** implementa a la interface implícitamente y puede sobrescribir cualquiera de los métodos adaptándolos a sus necesidades.

Interface en Java. Ejercicio resuelto

```
public class App {  
    public static void main(String[] args) {  
        Alumno a1 = new Alumno("Juan Ruiz",LocalDate.of(2000, 12, 3),"nre434","1º");  
        System.out.println(a1.devolverContenidoArrayList());  
    }  
}
```

Interfaces en Java

- **¿Cuándo definir una interface?**
 - **Cuando** solo vas a proporcionar una lista de métodos abstractos, es decir, **no hay atributos**, es recomendable definir una interface en lugar de una clase abstracta.
 - Es más, cuando necesites definir una clase base, puedes comenzar por declararla como interface y solo si estás obligado a definir métodos o variables miembro, puedes convertirla en clase abstracta (no instanciable) o clase instanciable.

Interfaces en Java

- A partir de **Java 8** el concepto de Interface se amplía y las interfaces pueden contener también métodos de extensión:
 - **Métodos por defecto** (públicos)
 - **Métodos estáticos**
 - **Métodos privados**, a partir de **Java 9**.

Métodos por defecto

- Los métodos por defecto se declaran utilizando la palabra **default** y son métodos **públicos** implícitamente:

```
public interface Atrapable {  
    int TIEMPO_RETENCION = 1000;  
  
    void atrapar();  
    void liberar();  
  
    default void retener() {  
        atrapar();  
        Alarma.dormir(TIEMPO_RETENCION);  
        Liberar();  
    }  
}
```

- Estos métodos pueden ser redefinidos en las clases que implementan las interfaces donde se declaran.

Métodos estáticos

- Los métodos estáticos se especifican mediante la palabra reservada `static` y son públicos por defecto:

```
public interface Atrapable {  
    ...  
    static void esconder() { //public por defecto  
        System.out.println("Se esconde");  
    }  
}
```

- Los métodos estáticos pertenecen a la interface y se invocan:

```
Atrapable.esconder();
```

Métodos privados

- Los métodos privados se especifican mediante la palabra reservada `private`:

```
public interface Atrapable {  
    ...  
    private void mostrar() {  
        System.out.println("Se muestra");  
    }  
}
```

- Los métodos `private` pueden ser también `static` y no son accesibles fuera del código de la interface. Solo pueden ser invocados por otros métodos no abstractos de la interface.

Recuerda: Un método sobrescrito debe ser `public`

Interface en Java – Ejercicio

- Define las interfaces:
 - **Pintable**: define un método por defecto *colorear* que no devuelve nada y recibe un Color que imprime por pantalla.
 - **Borrable**: define un método abstracto *borrar*.
 - **Dibujable**: hereda de Pintable y Borrable y define un método abstracto *dibujar*.
- Define la clase abstracta **Figura** que implementa Dibujable: define dos métodos abstractos *perímetro* y *área*, ambos devuelven un double. Define también un método no abstracto.
- Define la clase **Rectángulo** que herede de Figura e implementa todos los métodos que esté obligado a implementar y redefine el resto.

Interface en Java – Ejercicio - Solución

```
public interface Pintable{
    default void colorear(Color color){
        System.out.println(color);
    }
}

public interface Borrable{
    void borrar();
}

public interface Dibujable extends Borrable, Pintable{
    void dibujar();
    static void probar(){
        System.out.println("Probamos a dibujar");
    }
}

public abstract class Figura implements Dibujable{
    public void mostrar(){
        System.out.println("La figura se muestra");
    }
    public abstract double perimetro();
    public abstract double area();
}
```

```
public class Rectangulo extends Figura{
    private int a, b;
    @Override //estoy obligado a redefinir
    public double perimetro() { return 2 * (a+b); }
    @Override //estoy obligado a redefinir
    public double area() { return a * b; }
    @Override //estoy obligado a redefinir
    public void dibujar() {
        System.out.println("Dibujo un rectangulo de lados "+a+" "+b);
    }
    @Override //estoy obligado a redefinir
    public void borrar() {
        System.out.println("Borro el rectángulo");
    }
    @Override //No estoy obligado a redefinir
    public void colorear(Color color){
        System.out.println("Rectángulo ");
        super.colorear(color);
    }
    @Override //No estoy obligado a redefinir
    public void mostrar(){
        System.out.println("El rectángulo se muestra");
    }
}
```

Problemas de las interfaces

- ❑ Derivados de la herencia múltiple: al implementar varias interfaces se puede producir:
 - ❑ Colisión de métodos.
 - ❑ Colisión de métodos por defecto → error en tiempo de compilación
- ❑ En tal caso,
 - ❑ si los métodos colisionados tienen **diferentes parámetros** no habrá problema → sobrecarga.
 - ❑ Si los métodos colisionados tienen **diferente tipo de valor de retorno**, se dará un error de compilación
 - ❑ Si los métodos colisionados **son exactamente iguales** en identificador, parámetros y tipo de retorno, solamente se podrá implementar uno de los métodos.


Interfaces en Java y Polimorfismo

- El **polimorfismo** también se puede dar con interfaces.
- Es posible declarar variables cuyo tipo sea una **interface**.
 - Dicha variable únicamente podrá referenciar objetos de una clase que **implemente la interface** (bien por sí misma o porque lo haga su superclase).
 - El objeto referenciado por una variable cuyo tipo sea una interface **solo podrá invocar métodos definidos en la interface**, es decir, **no podrían usarse los métodos y atributos definidos en su clase**.
- Las referencias cuyo tipo es una interface permiten hacer referencia a objetos que no tienen ninguna relación jerárquica entre sí utilizando la misma variable.
- El operador **instanceof** permite comprobar si un objeto implementa o no una interface.

Interfaces en Java – Comparar u ordenar objetos

- ❑ Las interfaces se utilizan habitualmente para comparar objetos entre sí, permitiendo ordenar un conjunto de objetos.
- ❑ Recordemos los métodos de la clase Arrays:

```
public class EjemploOrdenar {  
    public static void main(String[] args) {  
        int[] numeros = new int[10];  
        for(int i=0;i<numeros.length;i++){  
            numeros[i]=numeros.length-i;  
        }  
        //Muestro el array tal y como ha sido inicializado  
        for(int i=0;i<numeros.length;i++){  
            System.out.println(numeros[i]+" ");  
        }  
        //Ordenamos el array  
        Arrays.sort(numeros);  
        System.out.println(Arrays.toString(numeros));  
    }  
}
```



static void sort (int[] a)

Interfaces en Java – Comparar u ordenar objetos

- Veamos el mismo ejemplo pero con objetos como elementos:

```
public class EjemploOrdenar {  
    public static void main(String[] args) {  
        Integer[] numeros = new Integer[10];  
        for(int i=0;i<numeros.length;i++){  
            numeros[i]=new Integer(numeros.length-i);  
        }  
    }  
}
```

static void sort (Object[] a)

Para poder usar el método sort los objetos deben implementar la interface Comparable. Se aplica el orden natural de los objetos

```
System.out.println(Arrays.toString(numeros));  
→ Arrays.sort(numeros);  
System.out.println(Arrays.toString(numeros));
```

```
}
```

```
}
```

Interface Comparable

- ❑ La interface **Comparable** del paquete `java.lang` permite establecer un criterio de comparación natural o por defecto (**orden natural**) entre objetos de una clase.
- ❑ La interface Comparable tiene un solo método **compareTo** que permite comparar objetos en base a un criterio:

`int compareTo (T ob)` donde T es el tipo de objetos a
comparar

Este método devuelve:

- 1 si el objeto this es menor que ob
- 1 si el objeto this es mayor que ob
- 0 si son iguales

Interface Comparable

- ❑ **Comparable** se utiliza cuando existe **un solo orden natural** o criterio de ordenación para los objetos, que es el utilizado por los métodos del API de Java, como **sort**.
- ❑ El método **sort** ordena ascendentemente un array de Object, de acuerdo con el orden natural de sus elementos, el cual dependerá de la naturaleza de dichos elementos.
 - ❑ Si consultamos el API de Java, veremos que para usar el método **sort**, la clase de los objetos a ordenar debe implementar la interface Comparable.
- ❑ **Comparable debe ser implementada por las clases cuyos objetos queremos comparar u ordenar.**
 - ❑ Esta interface es implementada por las clases Integer, Double, Character, Byte, String, LocalDate, ...

Interface Comparable

Ejemplo: Dado un array de objetos Alumno, ordenémoslo alfabéticamente por apellido (supongamos un solo apellido).

Añadimos a la clase Persona anterior:

```
public abstract class Persona implements Comparable {  
    ...  
    @Override  
    public int compareTo (Object o) {  
        Persona p = (Persona) o; //Debemos hacer el casting  
        //La clase String implementa Comparable  
        if(this.apellidos.compareTo(p.apellidos)<0)  
            return -1;  
        else if(this.apellidos.compareTo(p.apellidos)>0)  
            return 1;  
        else return 0;  
    }  
}
```


Interface Comparable

Si la clase Alumno hereda de Persona, también implementa la interface **Comparable** de forma implícita.

```
public static void main(String[] args) {  
    Alumno [] alumnos = new Alumno[4];  
    alumnos[0] = new Alumno("Antonio","Rodriguez",LocalDate.of(2002, Month.MARCH, 23),1236);  
    alumnos[1] = new Alumno("Pedro","García",LocalDate.of(2003, Month.MARCH, 23),345);  
    alumnos[2] = new Alumno("Sonia","Alvarez",LocalDate.of(2004, Month.MARCH, 23),125);  
    alumnos[3] = new Alumno("Jaime","Urrea",LocalDate.of(2000, Month.MARCH, 23),8987);  
    System.out.println(Arrays.toString(alumnos));  
    Arrays.sort(alumnos); //Alumno implementa Comparable  
    System.out.println(Arrays.toString(alumnos));  
}
```

Interface Comparator

- **Comparator** (**java.util**) es una interface que utilizamos cuando necesitamos distintos criterios de ordenación entre objetos.

- Implica implementar el método:

- int compare(T ob1, T ob2)** -1 si ob1 es menor que ob2
1 si ob1 es mayor que ob2
0 si son iguales

- Llamamos **comparador** a cualquier objeto de una clase que implemente esta interface.

- Necesitaremos una clase de comparadores distinta para cada criterio de comparación que necesitemos.

- A diferencia de la interface Comparable, la interface Comparator no es implementada por la clase cuyos objetos queremos ordenar sino por la clase del comparador.

Interface Comparator

Para aplicar un orden diferente al implementado ya por `Persona`, definimos una nueva clase que implemente la interface **Comparator** para la clase `Persona`. El nuevo criterio para ordenar en este ejemplo va a ser por **edad**.

```
public class OrdenPorEdad implements Comparator<Persona>{  
    @Override  
    public int compare(Persona o1, Persona o2) {  
        if(o1.getEdad() < o2.getEdad()){  
            return -1;  
        } if(o1.getEdad() > o2.getEdad()){  
            return 1;  
        }else  
            return 0;  
    }  
}
```

```
public static void main(String[] args) {  
    Alumno [] alumnos = new Alumno[4];  
    alumnos[0] = new Alumno("Antonio","Rodriguez",LocalDate.of(2002, Month.MARCH, 23),1236);  
    alumnos[1] = new Alumno("Pedro","García",LocalDate.of(2003, Month.MARCH, 23),345);  
    alumnos[2] = new Alumno("Sonia","Alvarez",LocalDate.of(2004, Month.MARCH, 23),125);  
    alumnos[3] = new Alumno("Jaime","Urrea",LocalDate.of(2000, Month.MARCH, 23),8987);  
    System.out.println(Arrays.toString(alumnos));  
    Arrays.sort(alumnos, new OrdenPorEdad());  
    System.out.println(Arrays.toString(alumnos));  
}
```

Interfaces en Java – Ejercicio resuelto

Ejercicio.

- a) Recupera la 1ª clase Cuenta de ejercicios anteriores e implementa un orden natural basado en el nombre del titular.
- b) Implementa otro orden basado en el saldo.

Interfaces en Java – Solución a)

```
public class Cuenta implements Comparable{
    private double saldo;
    private String titular;
    private static String tipo;
    ...
    @Override
    public int compareTo(Object o) {
        Cuenta c1 = (Cuenta) o;
        return this.titular.compareTo(c1.titular);
    }
}
```

Interfaces en Java – Solución b)

```
public class OrdenarPorSaldo implements Comparator<Cuenta> {  
    @Override  
    public int compare(Cuenta o1, Cuenta o2) {  
        if(o1.getSaldo() < o2.getSaldo()) return -1;  
        else if(o1.getSaldo() > o2.getSaldo()) return 1;  
        else return 0;  
    }  
}
```

Interfaces en Java – Ejercicio resuelto

Ordenamos por nombre del titular:

```
public static void main(String[] args) {  
    Cuenta[] banco = new Cuenta[5];  
    banco[0]=new Cuenta("Susana Rosa",6000);  
    banco[1]=new Cuenta("Andrés Pérez",234567);  
    banco[2]=new Cuenta("Jaime Ruiz",6000);  
    banco[3]=new Cuenta("Andrés López",200);  
    banco[4]=new Cuenta("Susana Rosa",1000);  
    Arrays.sort(banco);  
    System.out.println(Arrays.toString(banco));  
}
```

Para usar sort debemos
implementar Comparable

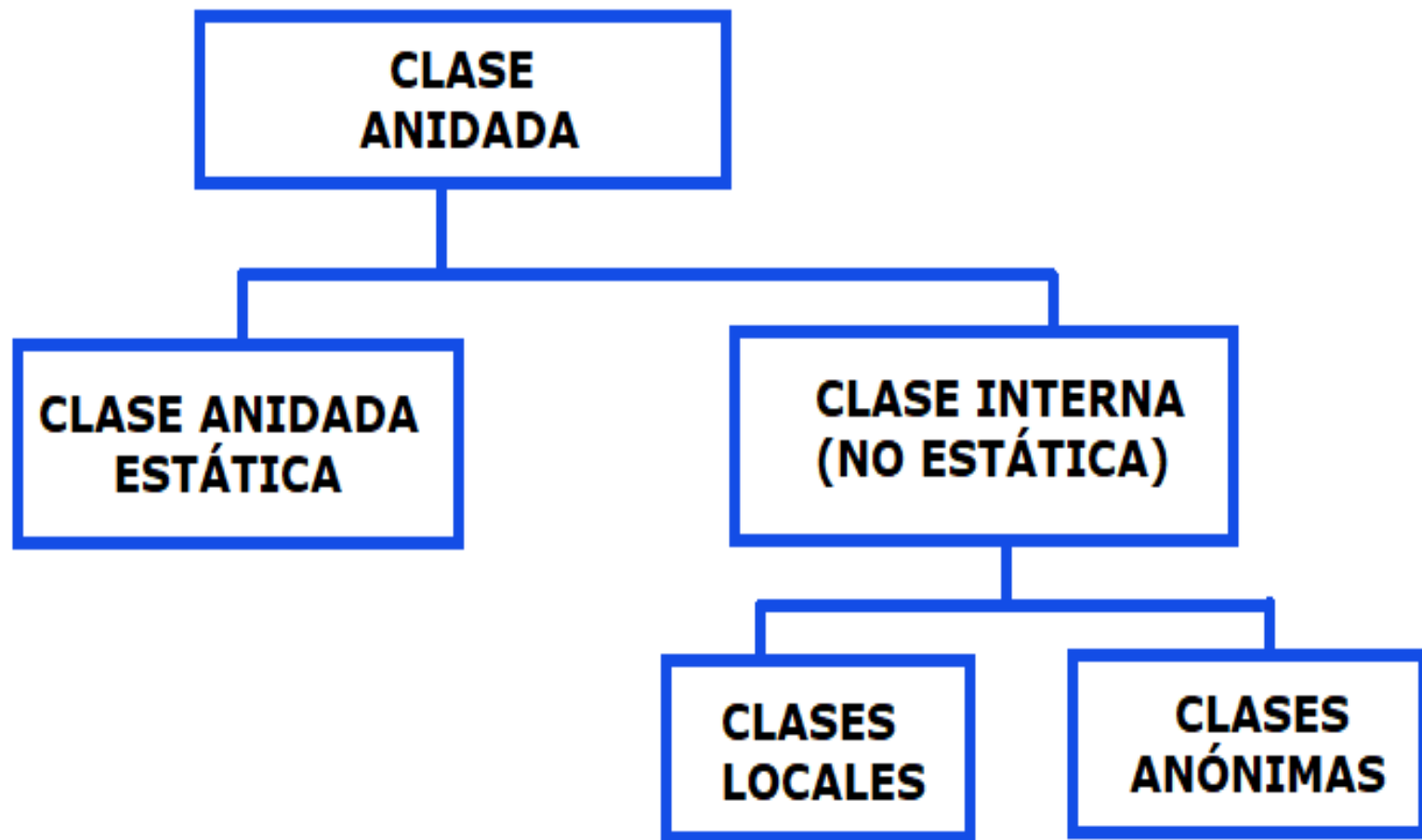
Para hacer esto debemos haber
redefinido toString en Cuenta

Interfaces en Java – Ejercicio resuelto

Para aplicar un nuevo orden, usamos la interface Comparator:

```
class OrdenarPorSaldo implements Comparator<Cuenta>{  
    @Override  
    public int compare (Cuenta o1, Cuenta o2) {  
        if(o1.getSaldo() < o2.getSaldo()) return -1;  
        else if(o1.getSaldo() > o2.getSaldo()) return 1;  
        else return 0;  
    }  
}
```

```
public static void main(String[] args) {  
    Cuenta[] banco = new Cuenta[5];  
    banco[0]=new Cuenta("Susana Rosa",6000);  
    banco[1]=new Cuenta("Andrés Saez",234567);  
    banco[2]=new Cuenta("Jaime Ruiz",6000);  
    banco[3]=new Cuenta("Andrés López",200);  
    banco[4]=new Cuenta("Susana Rosa",1000);  
    Arrays.sort(banco, new OrdenarPorSaldo());  
    System.out.println("Orden por saldo" + Arrays.toString(banco));  
}
```

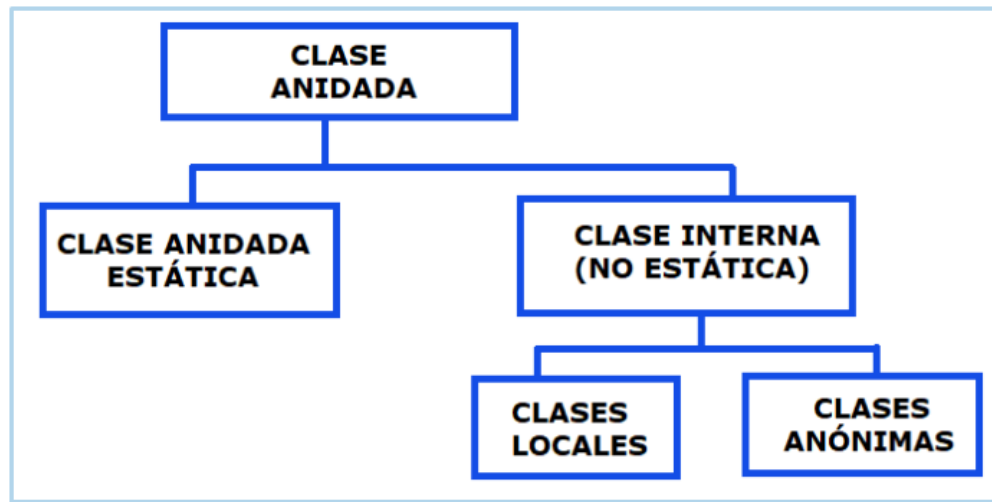
6. Clases anidadas en Java

Clases anidadas o Nested Classes

- Una **clase anidada** es una clase definida dentro de otra clase.
- Se utilizan cuando la lógica de las clases está muy relacionada o para **ocultar tipos de datos**.
- Las clases anidadas permiten agrupar las clases que solo se usan en un lugar, **favoreciendo el encapsulamiento**, creando código más fácil de leer y mantener.
 - Una clase anidada no existe independientemente de la clase contenedora.
 - No se trata de composición.
 - Al ser miembro de una clase, pueden declararse public, private, protected, package, static.
 - La clase anidada tiene acceso a los miembros de la clase contenedora.
 - La clase contenedora puede acceder a los miembros de la clase anidada pero **no directamente**, sino a través de un objeto de la misma.
 - Se puede anidar clases, interfaces, clases e interfaces ...
 - El compilador genera `ClaseContenedora$ClaseInterna.class`

Clases anidadas

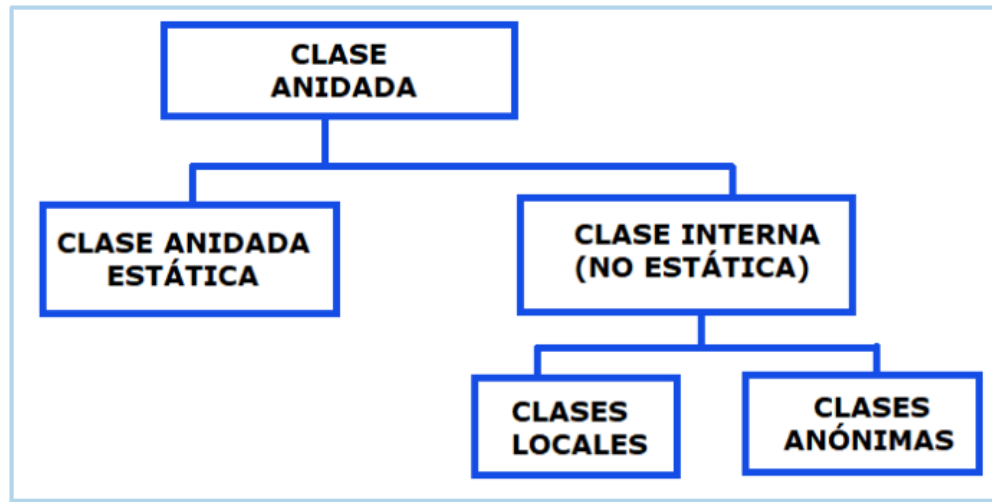
TIPOS



- Clases **anidada estáticas** (clases **anidadas o static nested class**) declaradas con el modificador `static`
- Clases **internas miembro** (clases **internas o inner class**). Aquellas que se declaran dentro de otra clase pero fuera de cualquier método. No son estáticas.
 - Clases **internas locales**, declaradas dentro de un bloque de código (un método normalmente, un bucle, una cláusula `if ...`)
 - Clases **internas anónimas**, similares a las internas locales, pero sin nombre (solo existirá un objeto de ellas y no tendrá constructor). Se usan en la gestión de eventos (GUI).

Clases anidadas

TIPOS



- Dependiendo de si la clase anidada debe acceder o no a los datos de la clase contenedora, se define como **estática** o **no estática** con la palabra reservada `static`.
- Las **clases estáticas** no necesitan una referencia a la clase que la contiene y por ello son más eficientes y es el método preferido para definirlos.
- Si se necesita acceso a los miembros de la clase contenedora hay que definirlos como **no estáticas**.

Clases anidadas estáticas

- Al igual que los métodos y variables de clase, una clase estática **no puede hacer referencia directamente a los miembros de instancia** (variables y métodos) definidos en la clase contenedora: solo puede usarlos a través de una referencia de objeto.
- Sintaxis para crear una instancia de la clase anidada static:

```
ClaseExterna.ClaseAnidadaStatic obj = new  
    ClaseExterna.ClaseAnidadaStatic ();
```

- Son útiles para reducir código fuente.
- Son comunes en el desarrollo de interfaces gráficas (GUI) y para construir comparadores de objetos.

Clases anidadas estáticas

```
// Programa que muestra el acceso a los campos de la clase Externa desde la clase Anidada
class Externa {
    //Miembro estático
    static int externo_x=10;
    //Miembro de instancia
    int externo_y=20;
    //Miembro privado
    private static int externo_privado=30;
//Clase anidada estática
    static class AnidadaStatic {
        void mostrar(){
            //Puede acceder al miembro estático de la clase externa
            System.out.println("externo_x: "+externo_x);
            //Puede acceder a mostrar un miembro estático privado de la clase externa
            System.out.println("externo_privado: "+externo_privado);
            // La siguiente declaración dará error de compilación ya que la clase anidada estática no
            //puede acceder directamente a un miembro no estático
            // System.out.println ("externo_y =" + externo_y);
            //Externa obj = new Externa(); obj.externo_y //Se podría acceder a través de un objeto
        }
    }
}
```

Clases anidadas estáticas

- Dada la clase Cuenta anterior, vamos a crear comparadores utilizando clases anidadas estáticas:

```
public class Cuenta implements Comparable {
    ....
    @Override
    public int compareTo(Object o) {
        Cuenta c1 = (Cuenta) o;
        return this.titular.compareTo(c1.titular);
    }
    //Clase anidada
    static class OrdenarPorSaldo implements Comparator<Cuenta>{
        @Override
        public int compare (Cuenta o1, Cuenta o2) {
            if(o1.getSaldo() < o2.getSaldo()) return -1;
            else if(o1.getSaldo() > o2.getSaldo()) return 1;
            else return 0;
        }
    }
    ...
}
```

Clases anidadas estáticas

Dada la clase Cuenta anterior, vamos a crear los comparadores utilizando clases anidadas estáticas:

```
public static void main(String[] args) {  
    Cuenta[] banco = new Cuenta[5];  
    banco[0] = new Cuenta("Andres Pérez", 1000000);  
    banco[1] = new Cuenta("Jaime Alvarez", 500);  
    banco[2] = new Cuenta("Jaime Ruiz", 100);  
    banco[3] = new Cuenta("Andres Pay", 234567);  
    banco[4] = new Cuenta("Susana Rosa", 6000);  
    Arrays.sort(banco); //ordena por titular (String)  
    System.out.println("Orden natural" + Arrays.toString(banco));  
    Arrays.sort(banco, new Cuenta.OrdenarPorSaldo());  
    System.out.println("Otro orden" + Arrays.toString(banco));  
}
```


Clases internas

- Tienen **acceso a todos los miembros** de la clase contenedora (también llamada adjunta) y puede referirse a ellos directamente → para que exista una clase interna debe existir un objeto de la clase contenedora (*)
- No pueden declarar atributos static

```
public class Externa {  
    class Interna {  
        // static int otroAtributo; //ERROR  
        public void imprimir() {  
            System.out.println("Clase interna");  
        }  
    }  
    public void escribir() {  
        System.out.println("Clase externa");  
    }  
}
```

```
public class PruebaClaseInterna {  
    public static void main(String[] args) {  
        Externa ext = new Externa(); (*)  
        Externa.Interna int = ext.new Interna(); //Ojo! Se permite porque Interna  
        int.imprimir(); //es friendly  
        ext.escribir();  
    }  
}
```

Clases internas

```
// Programa Java para demostrar el acceso a los atributos desde la clase interna
class ClaseExterna {
    //Miembro estático
    static int externo_x=10;
    //Miembro de instancia
    int externo_y=20;
    //Miembro privado
    private int externo_privado=30;
    class ClaseInterna{
        void mostrar(){
            //Puede acceder al miembro estático de la clase externa
            System.out.println("externo_x: "+externo_x);
            //Puede acceder a un miembro no estático de la clase externa
            System.out.println("externo_y: "+externo_y);
            //Puede acceder a mostrar un miembro privado de la clase externa
            System.out.println("externo_privado: "+externo_privado);
        }
    }
}
```

Clases internas. Ejemplo

```
public class Persona {  
    private String nombre;  
    private FechaNac fecha;  
  
    private class FechaNac{  
        private int dia, mes, año;  
        private FechaNac(int dia, int mes, int año){  
            this.dia = dia;  
            this.mes = mes;  
            this.año = año;  
        }  
    }  
  
    //Constructores  
    public Persona(){ }  
    public Persona(String nombre, int d, int m, int a){  
        this.nombre = nombre;  
        fecha = new FechaNac(d,m,a);  
    }  
}
```

```
    public String getNombre(){ return nombre;}  
    public String fechaNacimiento(){  
        return fecha.dia+"/"+  
            fecha.mes+"/"+  
            fecha.año;  
    }  
    public String toString(){  
        return "Nombre "+nombre+  
            fecha.dia+"/"+  
            fecha.mes+"/"+  
            fecha.año;  
    }  
}
```

Clases internas locales

- Son aquellas que se definen dentro de un bloque (y ese es su alcance):
 - Dentro de un método
 - Dentro de un bucle for
 - Dentro de una cláusula if
 - ...
- No son miembros de ninguna clase adjunta, pertenecen al bloque en el que están definidos y no pueden tener modificador de acceso asociado.
- Pueden ser final o abstract.
- Pueden extender una clase abstracta o implementar una interface.
- Tienen acceso a los campos de la clase que las contienen.
- Deben crear una instancia en el bloque en el que están definidas → **no pueden ser instanciadas fuera del bloque donde se han definido.**

Clases internas locales

```
public class Externa {
    int atributo1 = 10;
    public void imprimir(String parametro) {
        System.out.println("Comienzo del método imprimir de la clase Externa.");
        int variablelocal = 4;
        class Local { //solo podremos crear objetos Local dentro del método imprimir
            public void imprimir() {
                System.out.println("Método imprimir de la clase Local.");
                System.out.println(atributo1);
                System.out.println(parametro);
                System.out.println(variablelocal);
            } //Local tiene acceso a métodos, atributos, var locales y parámetros
            //del método donde se declara
        }
        Local local1 = new Local();
        local1.imprimir();
        System.out.println("Fin del método imprimir de la clase Externa.");
    }
}

public static void main(String[] ar) {
    Externa externa1 = new Externa();
    externa1.imprimir("Prueba");
}
}
```

Clases internas locales

❑ Errores:

```
public class ClaseExterna {  
    private int getValor(int dato) {  
        static class Interna {  
            private int getDato() {  
                System.out.println("Dentro de la clase interna.");  
                if(dato < 10) {  
                    return 5;  
                } else {  
                    return 15;  
                }  
            }  
        }  
        Interna interna = new Interna();  
        return interna.getDato();  
    } //Cierra método getValor  
    public static void main(String[] args){  
        ClaseExterna externa = new ClaseExterna();  
        System.out.println(externa.getValor(10));  
    }  
}
```

Error de compilación:
La clase interna local no se puede
declarar static




Clases internas locales

❑ Errores:

```
public class ClaseExterna {  
    private void miMetodo() {  
        class Interna {  
            private int metodoInterno() {  
                System.out.println("Dentro de la clase interna.");  
            }  
        }  
    }  
    public static void main(String[] args){  
        ClaseExterna externa = new ClaseExterna();  
        Interna interna = new Interna();  
        System.out.println(interna.metodoInterno());  
    }  
}
```

Error de compilación:

La clase interna no se puede instanciar fuera del bloque en el que ha sido definida.



Clases internas anónimas

- Son clases anidadas sin nombre.
- Por tanto no pueden tener un constructor.
- Son una solución rápida para implementar una clase que se va a utilizar una vez y de forma inmediata.
- Normalmente se declaran **como una subclase de otra clase** (abstracta o no) o **como implementación de una interface**.
- Se usan mucho en la programación de eventos. Lo veremos más adelante.

Clases internas anónimas

```
public class PruebaClaseAnonima {
    abstract class A {
        public abstract void imprimir();
    }
    interface B {
        void imprimir();
    }
    public void probar() {
        (new A() { //Una clase anónima que hereda de A e implementa el método heredado
            public void imprimir() {
                System.out.println("Clase");
            }
        }).imprimir();

        (new B() { //Otra clase anónima que implementa la Interface B y su método
            public void imprimir() {
                System.out.println("Interface");
            }
        }).imprimir();
    }

    public static void main(String[] ar) {
        PruebaClaseAnonima p = new PruebaClaseAnonima();
        p.probar();
    }
}
```

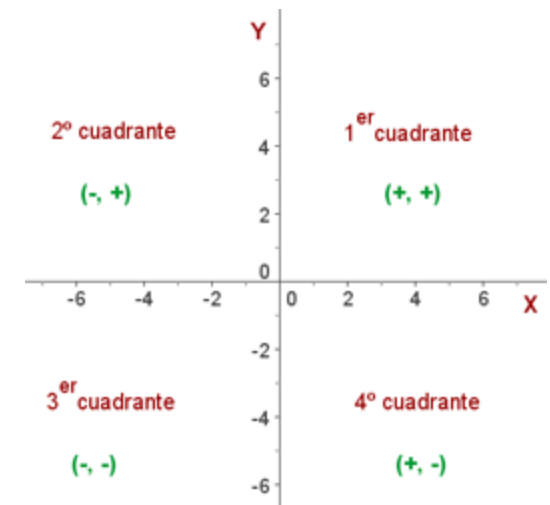
Clases internas anónimas

Dada la clase Cuenta anterior, vamos a crear un comparador utilizando clases anónimas:

```
public static void main(String[] args) {
    Cuenta[] banco = new Cuenta[5];
    banco[0] = new Cuenta("Andres Pérez",1000000);
    banco[1] = new Cuenta("Jaime Alvarez",500);
    banco[2] = new Cuenta("Jaime Ruiz",100);
    banco[3] = new Cuenta("Andres Pay",234567);
    banco[4] = new Cuenta("Susana Rosa",6000);
    Arrays.sort(banco);
    System.out.println("Usando CompareTo " + Arrays.toString(banco));
    Arrays.sort(banco, new Comparator<Cuenta>() {
        public int compare(Cuenta c1, Cuenta c2){
            if(c1.getSaldo() < c2.getSaldo()){ return -1;
            } else if(c1.getSaldo() > c2.getSaldo()) return 1;
            else return 0;
        }
    });
    System.out.println("Ordenar por saldo" + Arrays.toString(banco));
}
```

Ejercicio con clases anidadas

- Confeccionar una clase llamada **Coordenadas** que almacene una lista de objetos puntos en el plano cartesiano.
- ✓ Declara la **clase interna Punto** que represente un punto en el plano.
 - ✓ La clase Coordenadas debe almacenar un ArrayList de elementos de tipo Punto y proporcionar métodos
 - ✓ Además la clase Coordenadas debe poder calcular la cantidad de puntos almacenados en cada cuadrante (1,2,3,4).
- Crear una clase principal que cree diferentes puntos en el plano y muestre cuántos puntos hay en cada cuadrante.



Ejercicio con clases anidadas

```
//Clase contenedora  Coordenadas.java
public class Coordenadas {
    //Clase anidada
    private class Punto{
        private int x, y;
        private Punto(int x, int y){
            this.x=x;
            this.y=y;
        }
        //Devuelve el valor del cuadrante en el que
        // se sitúa el punto
        private int cuadrante(){
            if(x>0&&y>0) return 1;
            else if(x<0&&y>0) return 2;
            else if (x<0 &&y<0) return 3;
            else if(x>0&&y<0) return 4;
            else return -1;
        }
    }
}
```

```
//Atributos de la clase Coordenadas
private ArrayList<Punto> puntos;
public Coordenadas(){
    puntos = new ArrayList<Punto>();
}
//Métodos de Coordenadas
public void add(int x, int y){
    puntos.add(new Punto(x,y));
}
public int puntosCuadrante(int cuadrante){
    int cant = 0;
    for(Punto p: puntos){
        if(p.cuadrante() == cuadrante)
            cant++;
    }
    return cant;
}
```

Ejercicio con clases anidadas

```
//Clase ejecutable PruebaCoordenadas.java
public class PruebaCoordenadas {
    public static void main(String[] args) {
        Coordenadas coo = new Coordenadas();
        coo.add(30, 30);
        coo.add(2, 7);
        coo.add(-3, 2);
        coo.add(-5, -4);
        coo.add(-9, 10);
        System.out.println("Cantidad de puntos primer cuadrante "+coo.puntosCuadrante(1));
        System.out.println("Cantidad de puntos segundo cuadrante "+coo.puntosCuadrante(2));
        System.out.println("Cantidad de puntos tercer cuadrante "+coo.puntosCuadrante(3));
        System.out.println("Cantidad de puntos cuarto cuadrante "+coo.puntosCuadrante(4));
    }
}
```