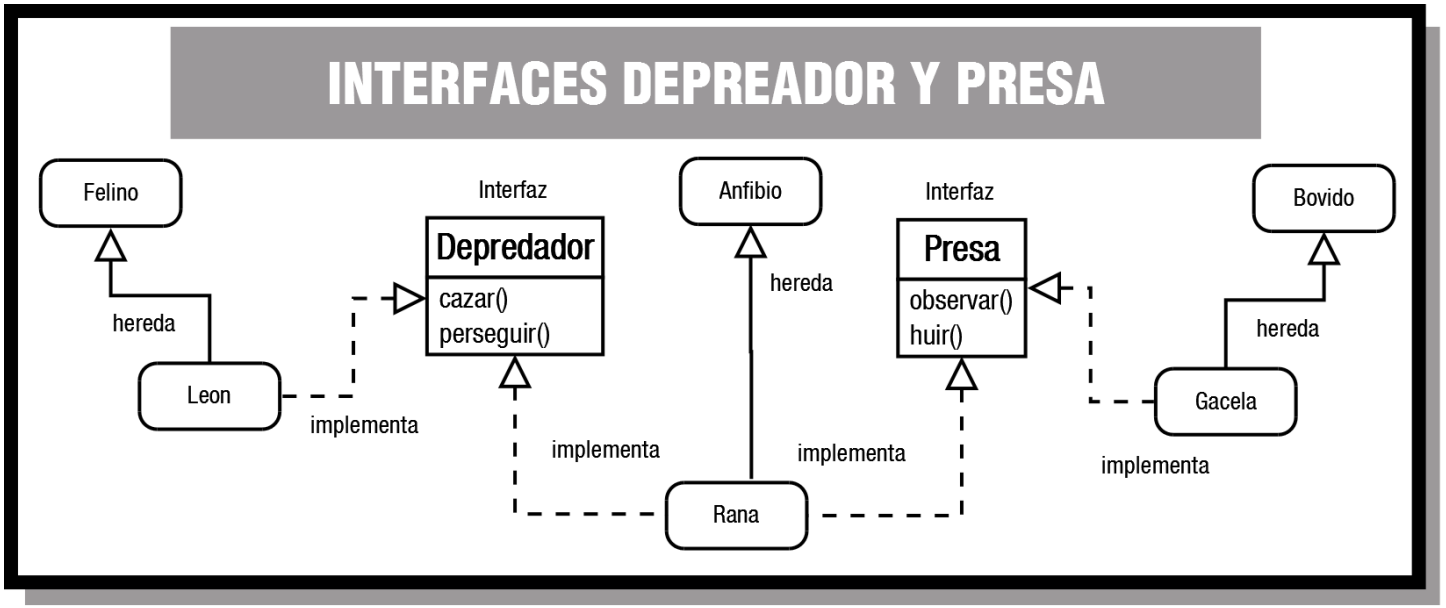
Unidad 6. POO Avanzada: Herencia, Interfaces y Polimorfismo



PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

José L. Berenguel

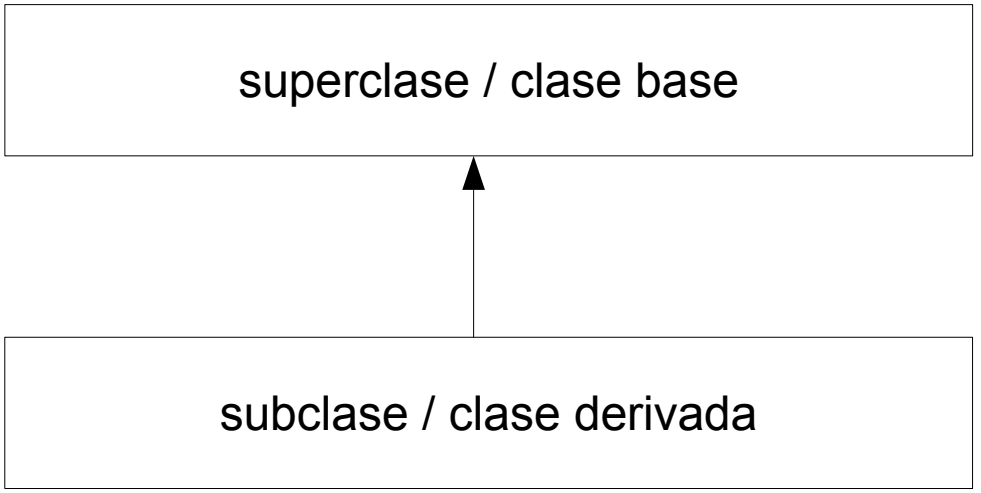
Tabla de Contenidos

Relaciones entre clases

- Se pueden distinguir diferentes formas de relación entre clases:
  - **Clientela**. Cuando una clase utiliza objetos de otra clase. Por ejemplo, cuando se crea un objeto en el interior de un método o se pasa por parámetro.
  - **Composición**. Cuando una clase contiene algún atributo que es una referencia a un objeto de otra clase. **Relación "tiene un"**.
  - **Anidamiento**. Cuando se definen clases en el interior de otra clase (***inner class***).
  - **Herencia**. Cuando una clase comparte determinadas características con otra (clase base), añadiéndole alguna funcionalidad específica (especialización). **Relación "es un"**.

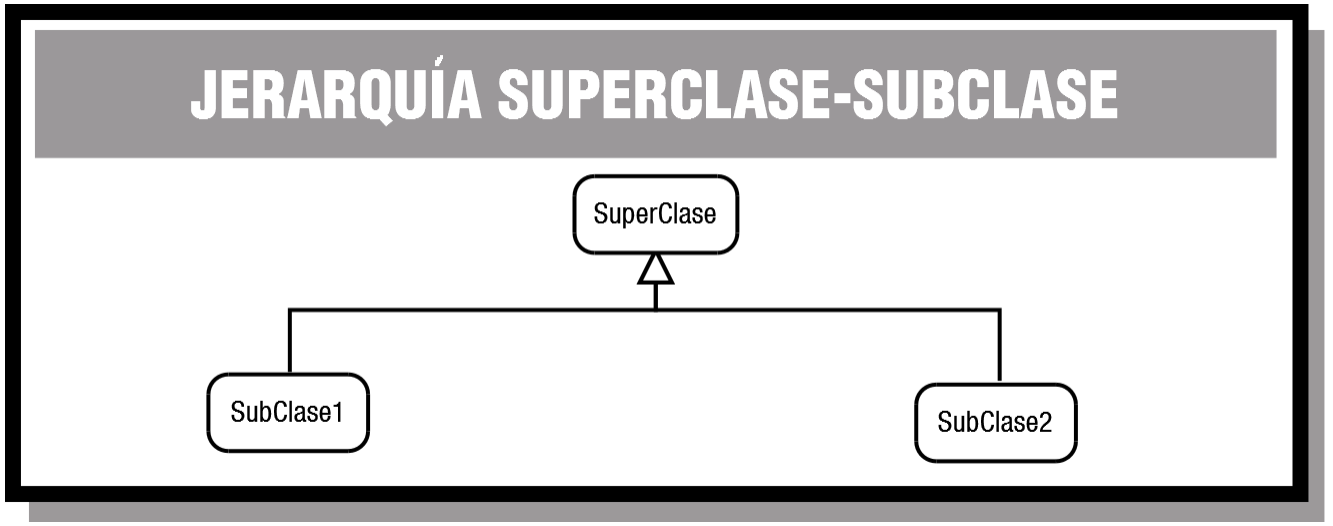
Herencia

- Concepto: Es la capacidad de crear clases que adquieran de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo añadir atributos y métodos propios.
- Facilita la reutilización y el mantenimiento de código.



Herencia

- En Java no está permitida la herencia múltiple: una subclase no puede heredar más de una clase.
- Una clase puede ser heredada por varias clases.
- No hay limitación en la cadena de herencia.
- Por defecto, toda clase Java hereda de ***Object***.



Herencia

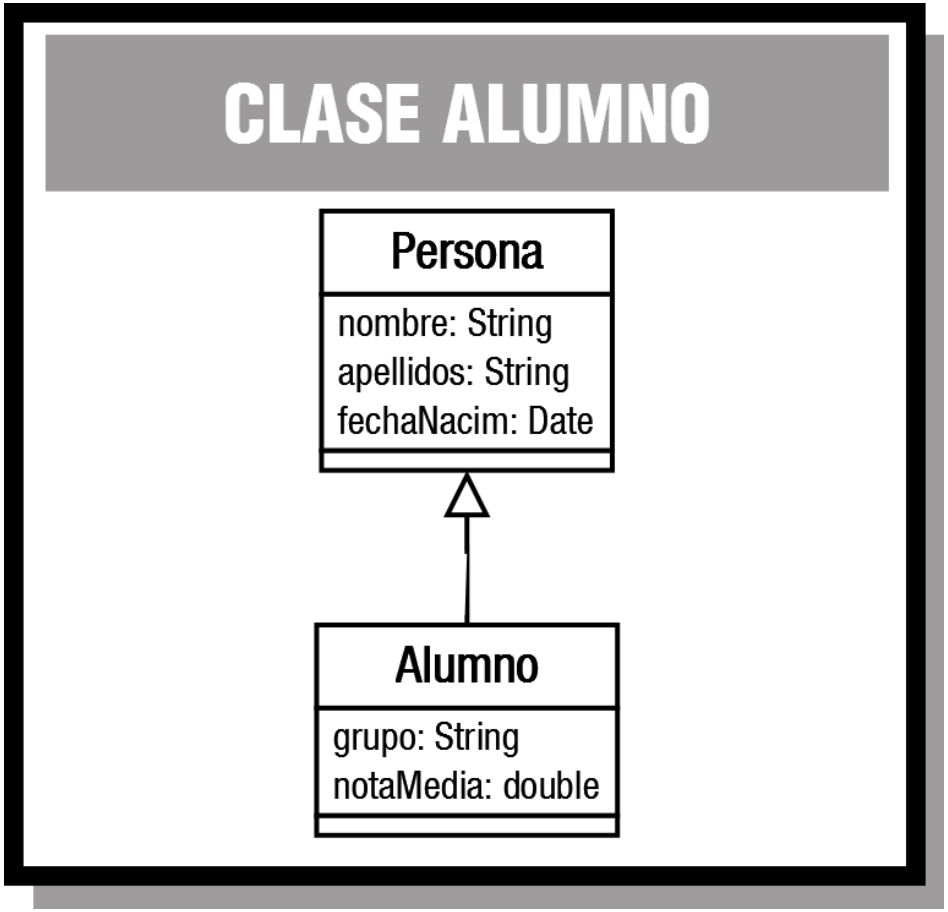
- Se utiliza la palabra reservada ***extends*** seguido del nombre de la superclase.
- Los miembros privados no son accesibles desde la clase derivada.
- Los constructores no se heredan.
- Relación "***es-un***": todo objeto de una subclase también es un objeto de la superclase.

```
public class PuntoColor extends Punto{
    private String color;
    //resto de la clase
}
```

Herencia

```
public class Persona {
     private String nombre;
     private String apellidos;
     private LocalDate fechaNacim;
     ...
}
```

```
public class Alumno extends Persona {
     private String grupo;
     private double notaMedia;
     ...
}
```



Herencia

- **Acceso a miembros heredados**.
  - Los atributos ***private*** no son accesibles por las subclases.
  - El modificador ***protected*** permite el acceso de las clases derivadas a estos elementos.
  - Los **miembros heredados** (atributos y métodos) pasan a formar parte de la clase derivada.

Cuadro de niveles accesibilidad a los atributos de una clase

	Misma clase	Otra clase del mismo paquete	Subclase (aunque sea de diferente paquete)	Otra clase (no subclase) de diferente paquete
public	X	X	X	X
protected	X	X	X	
Sin modificador (paquete)	X	X		
private	X			

Constructores en la herencia

- En la inicialización de objetos por medio de los constructores siempre se ejecuta antes el constructor de la superclase.
- ***super(argumentos)*** llama al constructor de la superclase. Debe ser la primera sentencia dentro del constructor.
- Si el programador no incluye la llamada a ***super***, el compilador añade la llamada al constructor por defecto ***super()***. Si este no está definido, se produce un error de compilación.

```
public class PuntoColor extends Punto{
    private String color;
    public PuntoColor(int x, int y, String c){
        super(x,y);
        this.color=c;
    }
}
```

Constructores en la herencia

- La llamada a ***super()*** se realiza ascendiendo en la cadena de herencia hasta que se alcanza la clase ***Object***. A partir de ahí, los atributos de cada clase se inicializan de manera descendiente.
- También se puede llamar a los constructores de la propia clase con ***this(argumentos)***:

```
public class PuntoColor extends Punto{
    private String color;
    public PuntoColor(int x, int y, String c){
        super(x,y);
        this.color=c;
    }
    public PuntoColor(int v, String c){
        this(v,v,c);
    }
}
```

La clase *Object*

- En Java, **por defecto cualquier clase hereda de *Object*** (aquellas clases donde no se indica *extends*).
- *Object* **define el estado y comportamiento básico** que debe tener cualquier clase.
  - La posibilidad de compararse con ***equals()***.
  - La capacidad de convertirse a cadena con ***toString()***.
  - La habilidad de devolver la clase del objeto.

Principales métodos de la clase `Object`

Método	Descripción
`Object ()`	Constructor.
`clone ()`	Método **clonador**: crea y devuelve una copia del objeto ("clona" el objeto).
`boolean equals (Object obj)`	Indica si el objeto pasado como parámetro es igual a este objeto.
`void finalize ()`	Método llamado por el **recolector de basura** cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
`int hashCode ()`	Devuelve un **código hash** para el objeto.
`toString ()`	Devuelve una representación del objeto en forma de `String`.

Sobrescritura de métodos

- La subclase puede **reescribir los métodos heredados** de la superclase, adaptando el comportamiento a las necesidades de la subclase:
  - El método **debe tener la misma signatura**: igual tipo y número de parámetros y tipo de retorno (si no estaríamos sobrecargando el método.
  - Puede tener un **modificador de acceso** menos restrictivo.
  - Para llamar al método original de la superclase se utiliza la expresión ***super.nombre_método(parámetros);***

Sobre la invariabilidad del tipo de retorno en el método sobrescrito, en Java 1.5 se introduce una excepción a este punto que se estudia más adelante en los tipos de retorno covariantes.

Sobrescritura de métodos

- Es recomendable usar la etiqueta *@override*, para indicar que el método sobrescribe uno de la superclase.
- Los **métodos estáticos** no se pueden sobrescribir.

```
public class Vehiculo{
    public void arrancar(){
        System.out.println("Arranca vehículo de forma genérica");
    }
}
```

```
public class Coche extends Vehiculo{
    //sobrescritura
    public void arrancar(){
        System.out.println("Arranca un coche");
    }
    //sobrecarga
    public void arrancar(String s){
        System.out.println("arranca un coche"+s);
    }
}
```

Sobrescritura de métodos

```
public class PuntoColor extends Punto{
    private String color;
    public PuntoColor(int x, int y, String c){
        super(x,y);
        this.color=c;
    }
    public PuntoColor(int v, String c){
        this(v,v,c);//invoca al otro constructor de la clase
    }
    public String getColor(){
        return this.color;
    }

}
```

Sobrescritura de métodos

```
public class PuntoColor extends Punto{
    private String color;
    public PuntoColor(int x, int y, String c){
        super(x,y);
        this.color=c;
    }
    public PuntoColor(int v, String c){
        this(v,v,c);//invoca al otro constructor de la clase
    }
    public String getColor(){
        return this.color;
    }
    //Sobrescritura del método toString()
    @Override
    public String toString(){
        String s=super.toString(); //invoca al método en Punto
        return s+" color="+this.color;
    }
}
```

Sobrescritura de métodos

```
/**
 * Método toString de la clase Alumno
 * Aprovecha el método toString de la clase Persona llamando a
 * super.toString().
*/

@Override
public String toString () {
    StringBuilder resultado;

    // Llamada al método “toString” de la superclase
    resultado= new StringBuilder (super.toString());

    // Añadimos la información “especializada” de la subclase
    resultado.append("\n");
    resultado.append ("Grupo: ").append(this.grupo).append("\n");
    resultado.append ("Nota media:
").append(String.format("%6.2f", this.notaMedia));

    return resultado.toString();
}
```

Sobrescribir el método equals()

- El método ***public boolean equals(Object o)*** está definido en la clase *Object*.
- Se utiliza para comprobar si dos objetos son iguales (comparando su estado).
- Se utilizan en numerosas clases del API, por ejemplo en las implementaciones de colecciones para poder utilizar el método ***contains()***, o para evitar duplicados en los conjuntos (*Set*), entre otras.
- Para sobrescribir *equals()* también se debe sobrescribir ***hashCode()***. Este método debe tener un código numérico idéntico para objetos iguales.

http://www.javapractices.com/topic/TopicAction.do?Id=17

Clases *final*

- Podemos evitar que una clase herede de otra declarándola como clase final.
- Si se intenta heredar de una clase final, se producirá un error de compilación.

```
public final class claseA{
    ...
}
```

```
//error de compilación
public claseB extends claseA{
    ...
}
```

Métodos *final*

- Si se desea que un método no pueda ser sobrescrito por las subclases debe declararse con el modificador ***final***.

```
[public|protected|private] final tipo_retorno metodo(argumentos){...}
```

Clases abstractas

- Son una pieza clave en la POO y juegan un papel importante en el concepto de ***polimorfismo***.
- No es posible crear objetos de una clase abstracta. Solo sirve de base para las subclases de ella.
  - ***Figura f=new Figura(); //error de compilación***
- Un **método abstracto** es aquel que **no tiene implementación**.
- Una clase abstracta puede no tener ningún método abstracto.

```
public abstract class Figura{
    public abstract double area();
    //otros métodos y constructores
}
```

Clases abstractas

- Las subclases de una clase abstracta están obligadas a implementar todos los métodos abstractos que heredan o a ser declaradas abstractas.
- Un **método abstracto no puede ser privado ni *static***, puesto que no podría redefinirse.

```
public abstract class Vehiculo{
    public abstract void arrancar();
}
```

```
public class Coche extends Vehiculo{
    public void arrancar(){
        System.out.println("Arranca un coche");
    }
}
```

Clases abstractas

- Una clase abstracta **puede tener constructores**, aunque no se permita crear objetos de ellas.
- Los constructores **son necesarios para inicializar los atributos** de la clase abstracta para los objetos de las subclases.

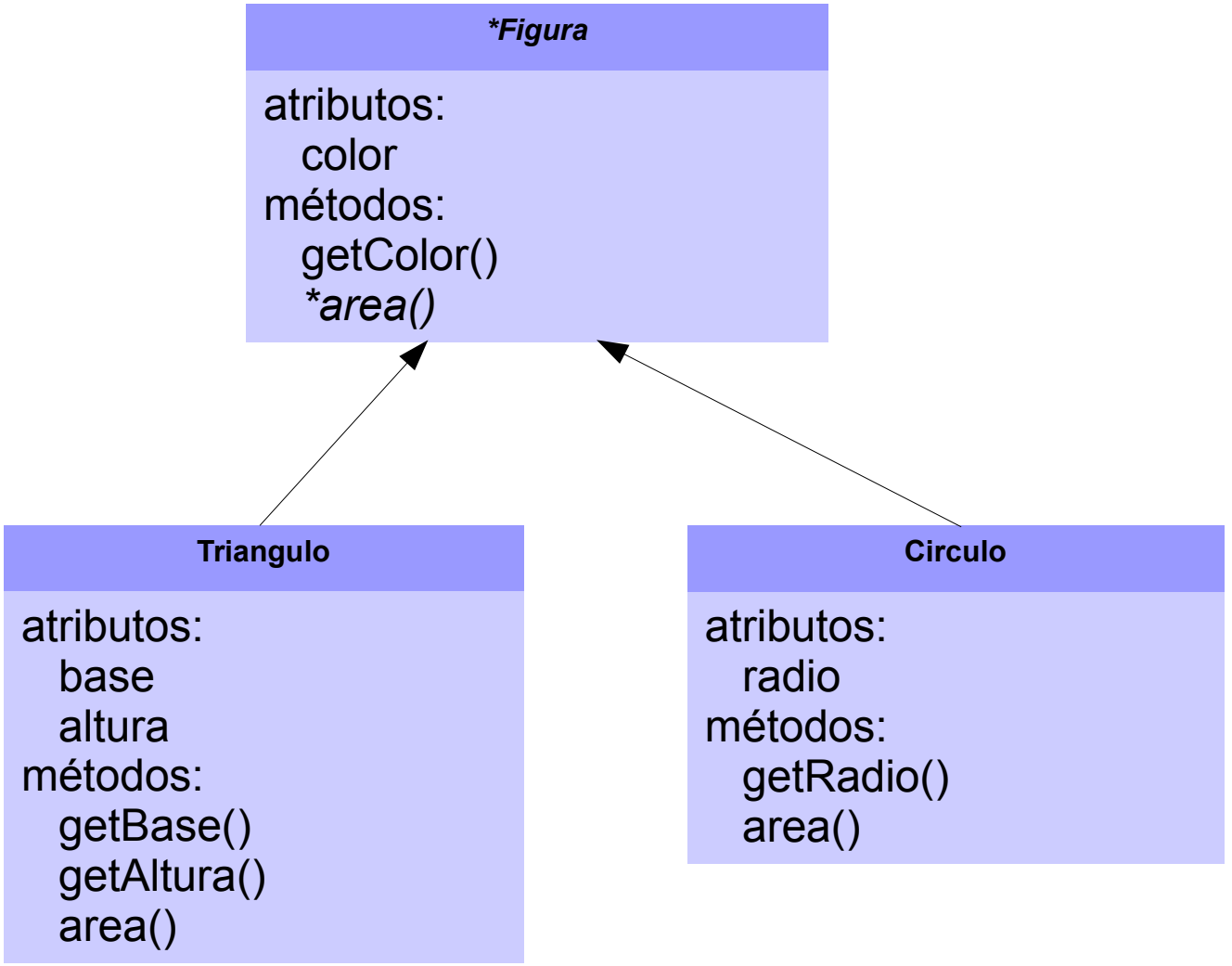
```
public abstract class Figura{
    private String color;
    public Figura(String c){
        this.color=c;
    }
    public String getColor(){
        return this.color;
    }
    public abstract double area();
}
```

Clases abstractas

```
public class Triangulo extends Figura{
    private double base, altura;
    public Triangulo(double base, double altura, String c){
        super(c);
        this.base=base;
        this.altura=altura;
    }
    public double area(){ return this.base*this.altura/2; }
    public double getBase(){ return this.base;}
    public double getAltura(){ return this.altura; }
}
```

```
public class Circulo extends Figura{
    private double radio;
    public Circulo(double radio, String c){
        super(c);
        this.radio=radio;
    }
    public double area(){ return Math.PI*radio*radio; }
    public double getRadio(){ return this.radio; }
}
```

Clases abstractas



Polimorfismo

- Recordemos que la herencia permitía establecer una relación ***"es-un"*** entre objetos.
- Esto permite que un objeto pueda asignarse a una referencia de su superclase:
  - ***Figura f = new Triangulo(...);***
- Pueden invocarse aquellos métodos del objeto que estén en la superclase (pero no aquellos que solo existan en la subclase). Para ello, habría que convertir la referencia a la subclase en cuestión (operador ***instanceof*** + ***casting***).

```
f.getColor(); //invoca a getColor() de Triangulo
f.area(); //invoca a area() de Triangulo
f.getBase(); //error de compilación
f.getAltura(); //error de compilación
```

Polimorfismo

- El polimorfismo permite utilizar una misma expresión para **invocar a diferentes versiones de un mismo método**.
- La versión del método a ejecutar se determina en tiempo de ejecución (método polimórfico). A esto se le conoce como **ligadura dinámica**.
- También se puede utilizar el **polimorfismo con interfaces**.

```
Figura f;
f=new Triangulo(..);
f.area(); //Método area() de Triangulo
f=new Circulo(..);
f.area(); //Método area() de Circulo
f=new Rectangulo(..)
f.area(); //Método area() de Rectangulo
...
```

Polimorfismo

```
public class GestionaFiguras{
    public static void main(String [] args){
        mostrar(new Triangulo(5,7,”verde”));
        mostrar(new Circulo(4,”azul”));
        mostrar(new Rectangulo(3,2,”naranja”));
    }

    public static void mostrar(Figura f){
        System.out.println(“El color de la figura es “+f.getColor());
        System.out.println(“El área de la figura es “+f.area());
    }
}
```

Tipos de retorno covariantes

- Al sobrescribir un método, es posible modificar el tipo de retorno siempre y cuando el nuevo tipo sea subclase del original.
- Supongamos el siguiente método en Figura:
  - ***abstract Figura getNewFigura();***
- Se podría sobrescribir en Circulo como:

```
public Figura getNewFigura(){
    return new Circulo(radio,getColor());
}
Circulo c2 = (Circulo)c1.getNewFigura(); //ejemplo de uso
```

```
public Circulo getNewFigura(){
    return new Circulo(radio,getColor());
}
Circulo c2 = c1.getNewFigura(); //ejemplo de uso
```

Interfaces

- Una interfaz establece un contrato común para todas las clases que se adhieran a esa interfaz.
- Una interfaz puede contener únicamente constantes y métodos abstractos.
- Todos los métodos son implícitamente ***public abstract*** y las constantes son implícitamente ***public static final***.
- Se define mediante la palabra reservada ***interface***:

```
[public] interface NombreInterfaz{
    tipo metodo1(argumentos);
    tipo metodo2(argumentos);
    ...
}
```

Interfaces

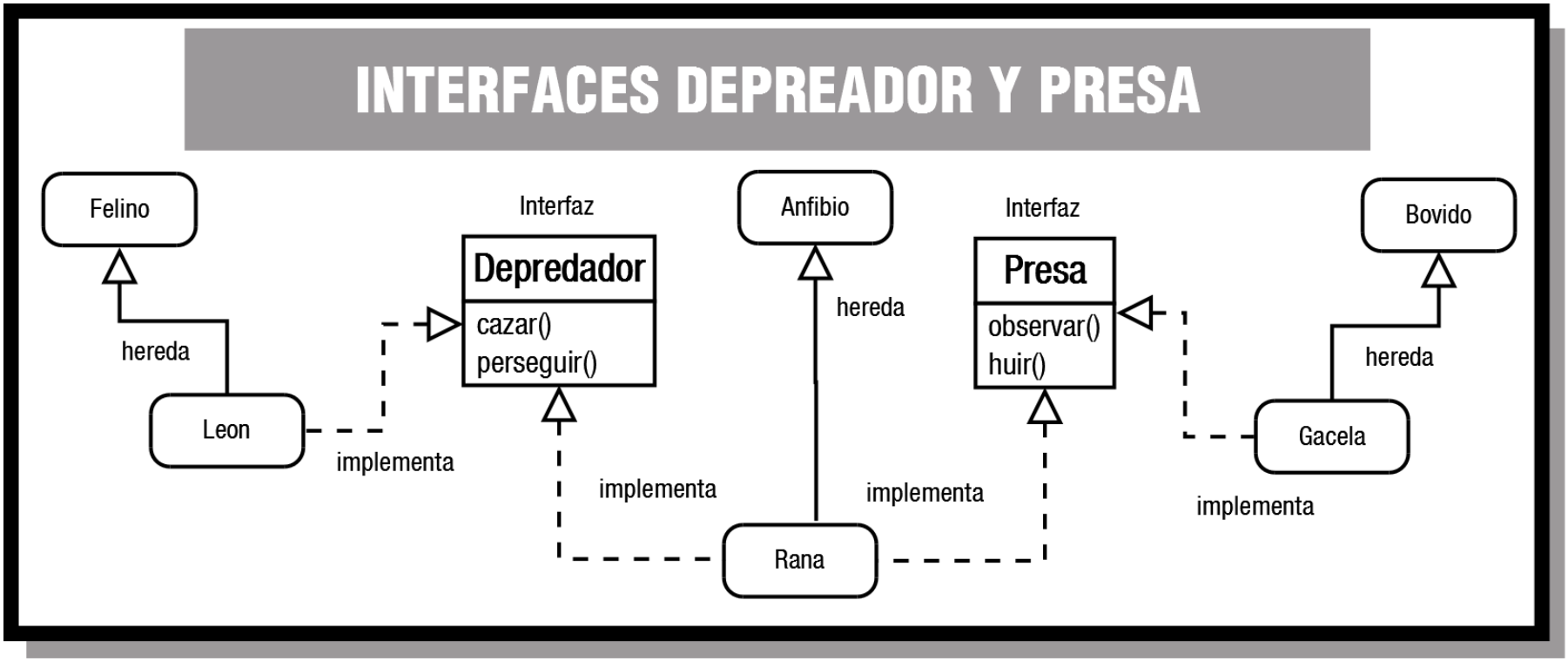
- En Java 8 se incorpora la posibilidad de escribir una implementación por defecto en los métodos de la interfaz utilizando la palabra reservada ***default***.
- En Java 8 se incorpora la posibilidad de definir métodos estáticos públicos en la interfaz.
- En Java 9 se incorpora la posibilidad de definir métodos privados y métodos estáticos privados en la interfaz.

Java 7	Java 8	Java 9
constante	constante	constante
Método abstracto	Método abstracto	Método abstracto
	Método predeterminado	Método predeterminado
	Método estático	Método estático
		Método privado
		Método estático privado

https://programmerclick.com/article/79661666399/

Interfaces

- Ejemplo en UML de diseño de clases con herencia e interfaces.



Implementación de una interfaz

- Se utiliza la palabra reservada ***implements***.
- Una clase puede implementar más de una interfaz.
- La clase debe implementar todos los métodos de la interfaz o declararse abstracta.

```
public class MiClase extends Superclase implements i1,i2,i3{
    ...
}
```

Implementación de una interfaz

- Al implementar más de una interfaz se pueden producir **colisiones de nombres si tienen un método con el mismo identificador**.
  - Si los métodos tienen **diferentes parámetros** no habría problema, ya que serían **métodos sobrecargados**.
  - Si lo métodos **solo se diferencian en su valor de retorno**, se produciría un **error de compilación**.
  - Si los dos métodos **son exactamente iguales**, se podría **implementar uno de los dos métodos** (en realidad al tener la misma signatura se considera el mismo método).

Comparable vs Comparator

- Interfaz ***Comparable*** y ***Comparable<T>***
  - Dispone de un método abstracto: ***compareTo()***.
  - Se utiliza para definir el orden natural entre los objetos de la misma clase.
  - Devuelve un número entero (*ob1.compareTo(ob2)*). Si es negativo ob1<ob2, si es positivo ob1>ob2, y si es 0 ob1==ob2.
  - Este interfaz es utilizado por numerosas clases y métodos del API Java. Por ejemplo el método ***Arrays.sort()*** ordenará un array de objetos utilizando su ordenación natural (llamando al método *compareTo*).

Comparable vs Comparator

- Interfaz ***Comparator*** y ***Comparator<T>***
  - Dispone de un método abstracto: ***compare(Object, Object)***.
  - Se utiliza para definir distintas ordenaciones al orden natural de los objetos. Es útil cuando queremos ordenar los objetos por diferentes criterios al del orden natural.
  - A diferencia del interfaz *Comparable*, este no se implementa en la misma clase del objeto, si no en una clase nueva.
  - Los métodos del API de Java están sobrecargados para poder indicar el tipo *Comparator* que se desea utilizar.

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

https://programmerclick.com/article/6790209380/

Comparable vs Comparator

```
import java.util.Comparator;

public class OrdenarPersonaPorAltura implements Comparator<Persona> {
    @Override
    public int compare(Persona o1, Persona o2) {
          // Devuelve un entero positivo si la altura de o1 es mayor que la de o2
          return o1.getAltura() - o2.getAltura();
    }
}
```

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

public class Programa {
    public static void main(String arg[]) {
        ArrayList<Persona> listaPersonas = new ArrayList<>();
        listaPersonas.add(new Persona(1,"Maria",185));
        listaPersonas.add(new Persona(2,"Carla",190));
        listaPersonas.add(new Persona(3,"Yovana",170));
        Collections.sort(listaPersonas, new OrdenarPersonaPorAltura());
        System.out.println("Personas Ordenadas por orden total: "+listaPersonas);
      }
}
```

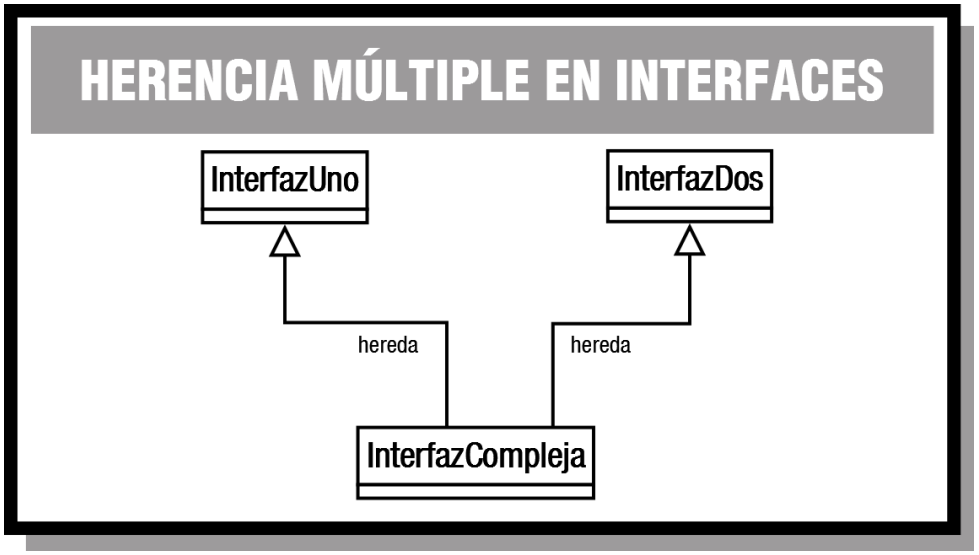
https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=599:interface-comparator-api-java-diferencias-con-comparable-clase-collections-codigo-ejemplo-cu00918c&catid=58&Itemid=180

Herencia de interfaces

- **Una interfaz puede heredar una o más interfaces**. La interfaz resultante es el conjunto de todos los métodos abstractos existentes en las superinterfaces.
- No se pueden heredar o implementar dos interfaces que causen un conflicto en alguno de sus miembros (constantes o métodos).

```
public interface InterfazCompleja extends InterfazUno, InterfazDos{

}
```



Interfaces y polimorfismo

- Una variable de tipo interfaz puede referenciar cualquier objeto de las clases que implementen dicho interfaz.
- Supongamos que se ha definido el interfaz ***Operaciones*** con los métodos ***rotar()*** y ***serializar()*** y que la clase ***Figura*** implementa dicho interfaz:

```
Operaciones op = new Triangulo();
op.rotar();
op.serializar();
```

Operador *instanceof*

- Resuelve si un objeto asignado a una variable polimórfica es una instancia de una determinada clase. Devuelve *true* o *false*.
- Sintaxis:
  - ***<variable_objeto> instanceOf <Nombre_Clase>***

```
public static void ejemplo(Figura f){
    if(f instanceOf Triangulo){
        Triangulo t=(Triangulo)f; //referencia a tipo Triangulo
        ..
    } else if(f instanceOf Circulo){
        Circulo c=(Circulo)f; //referencia a tipo Círculo
        ..
    }
}
```

Clases anidadas o internas

- Se distinguen diferentes tipos de clases internas:
  - **Clases internas estáticas o clases anidadas**. Se declaran con el modificador *static*.
  - **Clases internas miembro**. Conocidas como **clases internas** se declaran al máximo nivel de la clase contenedora y no estáticas.
  - **Clases internas locales**. Se declaran en el interior de un bloque de código, normalmente un método.
  - **Clases anónimas**. Similares a las internas locales, pero sin nombre. Solo existirá un objeto de ellas. Se suelen emplear en lo manejadores de eventos de las interfaces gráficas.

https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
https://www.w3schools.com/java/java_inner_classes.asp

Unidad 6. POO Avanzada: Herencia, Interfaces y Polimorfismo

Dudas y preguntas