

# Tema 3 – Herencia en Java – Parte 1

---

Programación Orientada a Objetos

Grado en Ingeniería Informática

# Contenido

---

- Introducción.
- Definición y tipos.
- Constructores.
- Redefinición.
- Visibilidad protegida
- Polimorfismo.
- Herencia y sistema de tipos.
- Ligadura dinámica.
- Casting
- Operador `instanceof`

# Introducción

---

- Entre las clases pueden existir **relaciones conceptuales**:  
**Extensión**, **Especialización**, **Combinación**.
- Ejemplos:
  - “Una pila puede definirse a partir de una cola o viceversa”
  - “Un rectángulo es una especialización de polígono”
  - “Libros y Revistas tienen propiedades comunes”
- **Herencia**:
  - Mecanismo para definir y utilizar estas relaciones.
  - Permite la **definición de una clase a partir de otra**.

# Introducción

---

- ❑ La herencia organiza las clases en una estructura jerárquica → **Jerarquía de clases**
- ❑ No es sólo un mecanismo de reutilización de código.
- ❑ Es **consistente con el sistema de tipos**.
- ❑ Ejemplos:



# Herencia

---

- Si la clase **B hereda de A**, entonces B incorpora la estructura (**atributos**) y comportamiento (**métodos**) de la clase A, pero puede incluir **adaptaciones**:
  - B puede añadir **nuevos atributos**.
  - B puede añadir **nuevos métodos**.
  - B puede **redefinir métodos heredados** (refinar o reemplazar).
  
- En general, las adaptaciones dependen del lenguaje OO.

# Herencia – Terminología

---

A



B

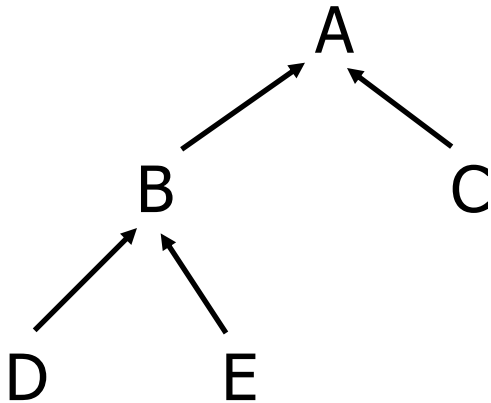


C

- ❑ B hereda de A (A es la superclase y B la subclase)
- ❑ A es la clase padre o clase base de B
- ❑ C hereda de B y A
- ❑ B y C son subclases de A
- ❑ B es un descendiente directo de A
- ❑ C es un descendiente indirecto de A
- ❑ A y B son ascendientes de C

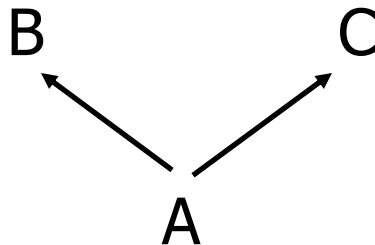
# Tipos de herencia

---



## □ Herencia simple:

- Una clase puede heredar de una única clase.
- Java, C#



## □ Herencia múltiple:

- Una clase puede heredar de varias clases.
- C++

# Reconocer la herencia

---

## □ **Especialización:**

- Se detecta que una clase es un **caso especial** de otra.
- Ejemplo: Rectángulo es un tipo de Polígono.

## □ **Generalización** (factorización):

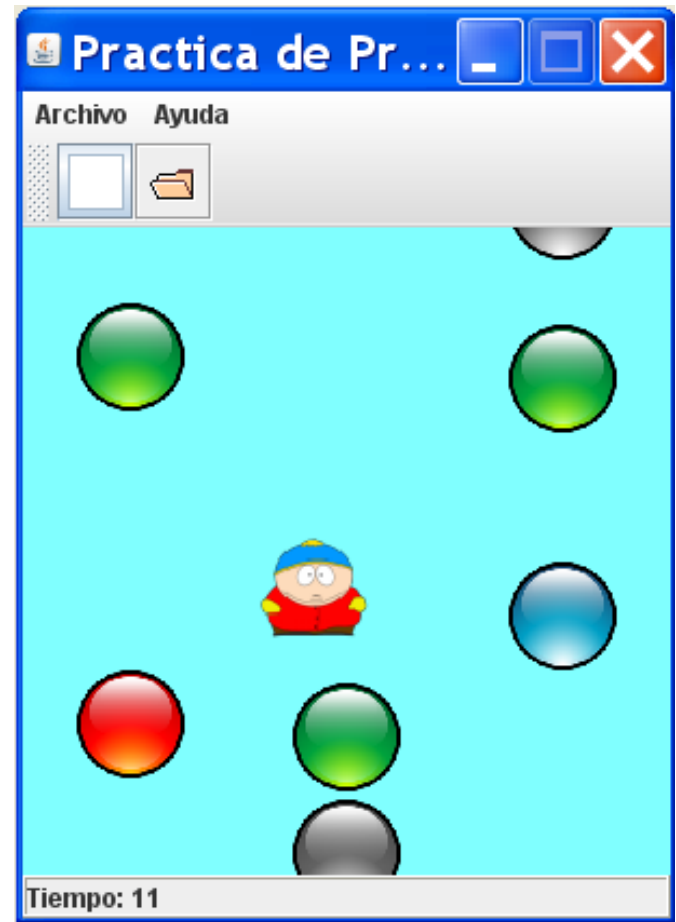
- Se detectan dos clases con **características en común** y se crea una clase padre con esas características.
- Ejemplo: Libro, Revista → Publicación

- No hay receta mágica para crear buenas jerarquías de herencia.



# Caso de estudio

- ❑ Implementación de un sencillo videojuego 2D.
- ❑ El juego está formado por un tablero con burbujas y un jugador que puede rebotar sobre ellas o hacerlas explotar.
- ❑ El componente básico del juego es la **Burbuja** que se mueve siempre en sentido vertical ascendente.



# Caso de estudio

---

- Las características que definen a una **Burbuja** son:
  - Propiedades:
    - *región* que ocupa en el espacio (círculo), *velocidad máxima*, *velocidad actual* y si está *explotada*.
  - Comportamiento:
    - Permite *explotar* una burbuja.
    - Permite mover una burbuja en sentido ascendente aumentando progresivamente su velocidad (*ascender*).
    - Permite *chocar* con una burbuja poniendo su velocidad actual a 0.
    - Permite *situar* una burbuja en un punto del tablero.

# Clase Burbuja 1/3

```
public class Burbuja {  
    public static final int VELOCIDAD_MAX = 50;  
    private Circulo region;  
    private final int velocidadMax;  
    private int velocidadActual;  
    private boolean explotada;  
  
    public Burbuja(Circulo region, int velocidadMax) {  
        this.region = new Circulo(region);  
        this.velocidadMax = velocidadMax;  
        this.explotada = false;  
        this.velocidadActual = 0;  
    }  
    public Burbuja(Circulo region) {  
        this(region, VELOCIDAD_MAX);  
    }  
    ...  
}
```

# Clase Burbuja 2/3

---

```
public class Burbuja {  
    ...  
    // Métodos de consulta y establecimiento  
  
    public Circulo getRegion() {  
        return new Circulo(region);  
    }  
    public int getVelocidadActual() {...}  
    public int getVelocidadMaxima() {...}  
    public boolean isExplotada() {...}  
    ...  
}
```

# Clase Burbuja 3/3

```
public class Burbuja {  
    ...  
    public void explotar() {  
        explotada = true;  
    }  
    public void situar(Punto posicion) {  
        int incX = posicion.getX() - region.getCentro().getX();  
        int incY = posicion.getY() - region.getCentro().getY();  
        region.desplazar(incX, incY);  
    }  
    public void ascender() {  
        region.desplazar(0, velocidadActual);  
        if (velocidadActual < velocidadMax)  
            ++velocidadActual;  
    }  
    public void chocar(){  
        velocidadActual = 0;  
    }  
}
```

# Caso de Estudio

---

- ❑ En el videojuego aparecen diferentes tipos de burbujas. Identificamos una burbuja que se para cuando se ha desplazado una determinada cantidad → **Burbuja Limitada**
- ❑ ¿Una Burbuja Limitada **es una** Burbuja?
  - Comparte todas las características de Burbuja.
  - Añade nuevas características.
  - Se diferencia en parte del comportamiento.
- ❑ → La clase **BurbujaLimitada hereda de Burbuja**:
  - **Nuevas características**: límite de desplazamiento, la cantidad que se ha desplazado y la consulta de si está parada.

# Clase BurbujaLimitada

---

```
public class BurbujaLimitada extends Burbuja {  
    //Nuevas propiedades  
    private final int limiteDesplazamiento;  
    private int cantidadDesplazamiento;  
  
    public BurbujaLimitada(...) {...}  
  
    public int getLimiteDesplazamiento() {...}  
    public boolean isParada(){  
        return (getVelocidadActual() == 0  
            || cantidadDesplazamiento >= limiteDesplazamiento);  
    }  
}
```

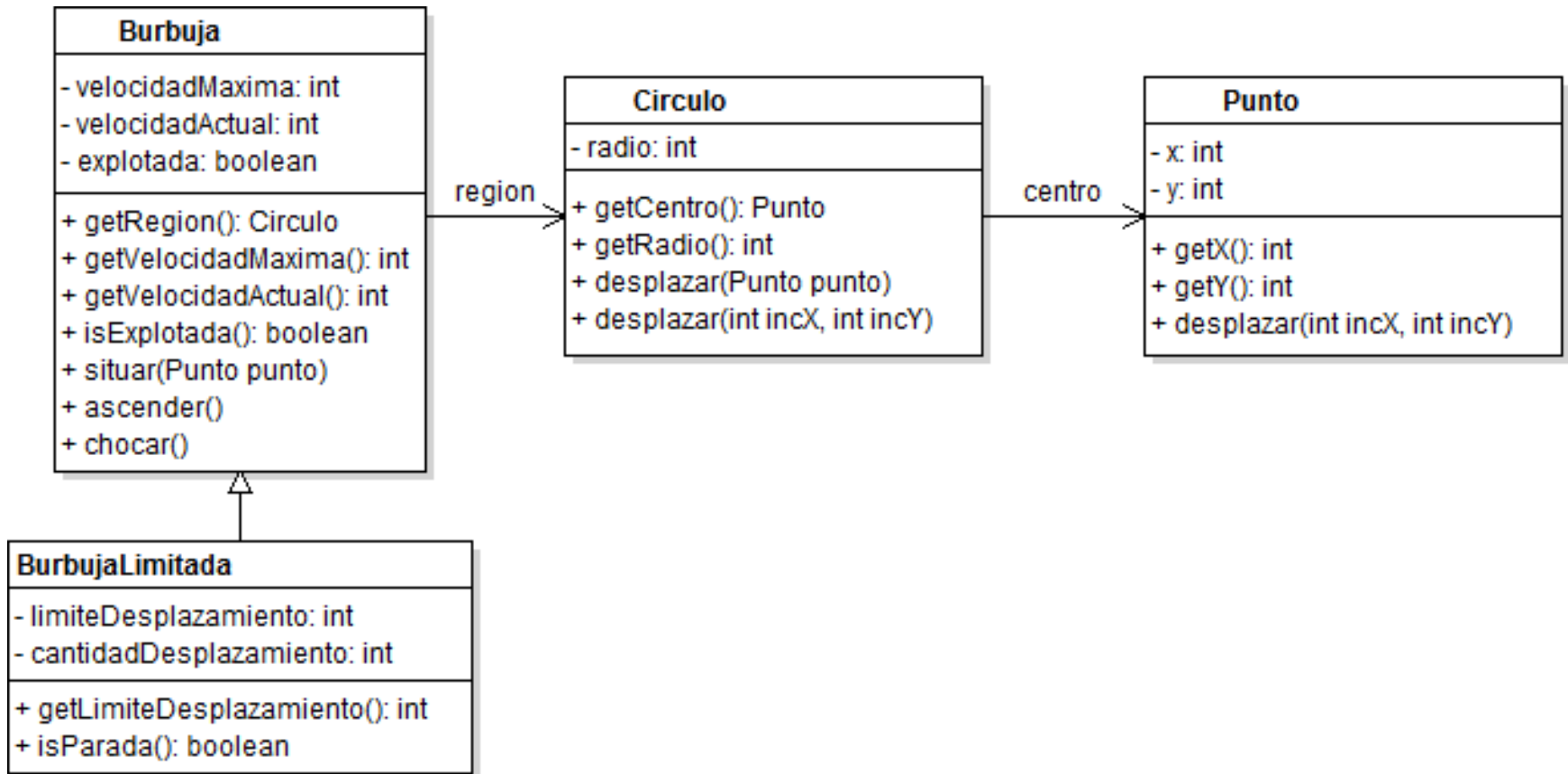
# Clase BurbujaLimitada

---

- ❑ La subclase incluye las **características específicas**:
  - Propiedad: *limite de desplazamiento*.
  - Atributo de implementación: *cantidad desplazamiento*.
  - Método para saber si una burbuja limitada está parada.
- ❑ La clase hija **hereda todos los atributos de la clase padre**, aunque no los vea debido a la ocultación de la información.
- ❑ La clase dispone de los métodos heredados como si fueran propios (ejemplo, `getVelocidadActual()`).



# Jerarquía de herencia



# Herencia y constructores

---

- ❑ En Java **los constructores no se heredan**.
- ❑ El **constructor** de una `BurbujaLimitada` se declara con los mismos parámetros que una `Burbuja` y añade la cantidad del límite de desplazamiento.
- ❑ La clase `BurbujaLimitada` **no tiene visibilidad sobre los atributos privados** de `Burbuja`.
- ❑ Java permite **invocar a los constructores de la clase padre** dentro de un constructor utilizando la llamada **`super(...)`**
- ❑ En las subclases también se puede reutilizar el código de los constructores con **`this(...)`**

# Clase BurbujaLimitada

---

```
public class BurbujaLimitada extends Burbuja {  
    ...  
    public BurbujaLimitada(Circulo region, int velocidadMax,  
                           int limiteDesplazamiento){  
        super(region, velocidadMax);  
  
        this.limiteDesplazamiento = limiteDesplazamiento;  
        this.cantidadDesplazamiento = 0;  
    }  
    ...  
}
```

# Clase BurbujaLimitada

---

```
public class BurbujaLimitada extends Burbuja {  
  
    ...  
    public BurbujaLimitada(Circulo region,  
                           int limiteDesplazamiento){  
        this(region, VELOCIDAD_MAXIMA, limiteDesplazamiento);  
    }  
    ...  
}
```

- BurbujaLimitada mantiene el contrato de construcción de la clase Burbuja.

# Herencia y constructores

---

- ❑ Cuando se aplica herencia, **la llamada a un constructor de la clase padre es obligatoria.**
- ❑ Debe ser la **primera sentencia** del código del constructor.
- ❑ **Si se omite la llamada**, el compilador asume que la primera llamada es `super()`
  - ➔ Llama al constructor sin argumentos de la clase padre.
  - ➔ Si no existe ese constructor, la clase no compila.

# Adaptación del código heredado

---

- ❑ La clase **BurbujaLimitada** añade nuevas propiedades: límite de desplazamiento.
- ❑ Sin embargo, hay que **adaptar** uno de los métodos heredados:
  - `ascender()`: una burbuja limitada cuando se ha desplazado el límite establecido se para y no sigue ascendiendo.
- ❑ La herencia permite la **redefinición de métodos** para adaptarlos a la semántica de la clase.

# Redefinición de métodos

---

- Un método es una **redefinición** si tiene la misma signatura (nombre, parámetros y tipo de retorno) que un método de la clase padre:
  - **Nota:** si cambia el tipo de algún parámetro o se añaden nuevos parámetros, entonces se está *sobrecargando* el método heredado.
  
- La redefinición reconcilia la **reutilización** con la **extensibilidad**:
  - Es habitual hacer cambios cuando se reutiliza un código.

# Redefinición de métodos

---

- La anotación **@Override**
  - Se escribe sobre la implementación de los métodos redefinidos.
  - Es opcional.
  - Se utiliza para indicar al compilador que un método es una redefinición.
  - Resulta de utilidad para detectar errores en la signatura de los métodos redefinidos en tiempo de compilación.



# Redefinición de métodos

---

- La redefinición de un método heredado puede ser de **dos tipos**:
  - **Refinamiento**: se añade nueva funcionalidad al comportamiento heredado.
  - **Reemplazo**: se sustituye completamente la implementación del método heredado.
  
- En el **refinamiento** de un método resulta útil **invocar a la versión heredada** del método.

# Redefinición de métodos y `super`

---

- ❑ El lenguaje proporciona la palabra reservada **`super`** que permite **llamar a la versión del padre** de un método que se redefine en la clase.
  - ❑ En general, el uso de *super* se recomienda sólo para el refinamiento de métodos.
- ➔ **No hay que utilizar *super* para llamar a métodos que se heredan.**

# Clase BurbujaLimitada

---

```
public class BurbujaLimitada extends Burbuja {  
    ...  
    @Override  
    public void ascender(){  
        if (cantidadDesplazamiento < limiteDesplazamiento){  
            int posicionYIni = getRegion().getCentro().getY();  
            super.ascender();  
            int posicionYFin = getRegion().getCentro().getY();  
            cantidadDesplazamiento += (posicionYFin - posicionYIni);  
        }  
    }  
    ...  
}
```

# Caso de estudio

---

- ❑ Identificamos un nuevo tipo de burbuja que se caracteriza por explotar cuando el jugador rebota un número determinado de veces (**Burbuja Débil**).
- ❑ Esta burbuja posee todas las características de una burbuja. Añade una nueva propiedad y cambia el comportamiento del método *chocar* (regla **es un**).
- ❑ La clase **BurbujaDebil** hereda de **Burbuja**.

# Clase BurbujaDebil

---

```
public class BurbujaDebil extends Burbuja {  
    private int botesRestantes;  
  
    public BurbujaDebil(Circulo region, int velocidadMax,  
                        int botesRestantes){  
        super(region, velocidadMax);  
  
        this.botesRestantes = botesRestantes;  
    }  
  
    public int getBotesRestantes() {  
        return botesRestantes;  
    }  
    ...  
}
```

# Redefinición de métodos

---

- ❑ La clase debe controlar el número de choques para explotar la burbuja.
- ❑ Para ello, **redefine** (refina) el método `chocar()`.

```
@Override
public void chocar(){
    --botesRestantes;

    super.chocar();

    if (botesRestantes == 0){
        this.explotar();
    }
}
```

# Caso de estudio

---

- Una **burbuja creciente** **es una** **burbuja débil** que crece cada vez que soporta un bote.
- La nueva burbuja tiene todas las características de una burbuja débil (regla **es un**).

➔ Clase **BurbujaCreciente** hereda de **BurbujaDebil**

# Caso de estudio. Burbuja Creciente

---

- El porcentaje de crecimiento es fijo (constante).
- Como burbuja débil que es soporta un número fijo de botes. Toma este valor del radio de su región.
  - El constructor de una burbuja creciente tiene menos parámetros que el constructor de la clase padre.
- **Adapta** el comportamiento del choque para gestionar el crecimiento de la burbuja (**redefinición**).
- Crecer significa **modificar su región**.
  - ¿Tiene acceso la burbuja creciente al atributo region?
  - ¿Tiene sentido un método público en la clase Burbuja que modifique la región?



# Visibilidad para la herencia

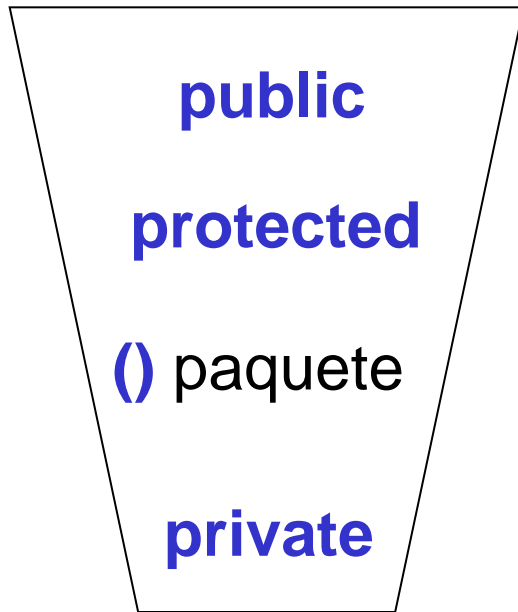
---

- ❑ Puede tener sentido que algunos **atributos** y **métodos** de una clase, sin ser públicos, puedan ser **accesibles a las subclases**.
- ❑ Ejemplo: el atributo **region** de **Burbuja**. La burbuja creciente tiene que modificar la región.
- ❑ **Nivel de visibilidad protegido** (*protected*): visibilidad para las subclases y las clases del mismo paquete.
- ❑ Es **discutible** el uso de visibilidad protegida **para los atributos**
  - En contra de la **ocultación de la información**.
- ❑ **Para los métodos, la visibilidad protegida es útil.**

# Niveles de visibilidad

---

- En Java, los **niveles de visibilidad son incrementales**



public → todo el código

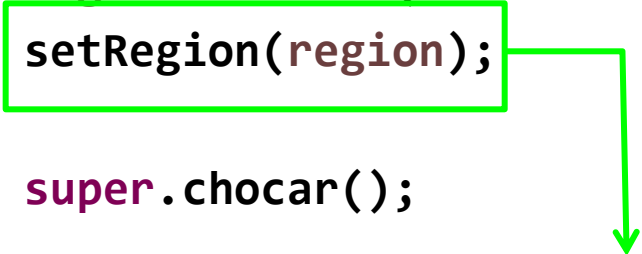
protected → clase + paquete + subclasses

(*nada*) → clase + paquete

private → clase

# Clase BurbujaCreciente

```
public class BurbujaCreciente extends BurbujaDebil {  
    private static final int PORCENTAJE_CRECIMIENTO = 125;  
  
    public BurbujaCreciente(Circulo region, int velocidadMax) {  
        super(region, velocidadMax, region.getRadio());  
    }  
    @Override  
    public void chocar() {  
        Circulo region = getRegion();  
        region.escalar(PORCENTAJE_CRECIMIENTO);  
        setRegion(region);  
        super.chocar();  
    }  
}
```



**Método con visibilidad protected en Burbuja**

# Caso de estudio

---

- ❑ Una **burbuja sensible** **es una burbuja** que se caracteriza por quedar detenida un *cierto tiempo* cada vez que recibe un choque.
- ❑ La nueva burbuja tiene todas las características de una burbuja (regla **es un**).

➔ Clase **BurbujaSensible** hereda de **Burbuja**

- ❑ Añade una **nueva funcionalidad** para comprobar si *está en espera*.
- ❑ Es necesario establecer una *marca de tiempo* en el choque (**redefinición**).
- ❑ **Adapta** el comportamiento del ascenso para comprobar si se ha superado el tiempo de espera (**redefinición**).

# Clase BurbujaSensible (1/2)

---

```
public class BurbujaSensible extends Burbuja {
    private static final int TIEMPO_ESPERA = 2000; // milisegundos
    private long instanteChoque; // atributo implementación

    public BurbujaSensible(Circulo region, int velocidadMaxima) {
        super(region, velocidadMaxima);
        instanteChoque = 0;
    }
    @Override
    public void chocar() {
        instanteChoque = System.currentTimeMillis();
        super.chocar();
    }
    public boolean isEnEspera(){
        return instanteChoque > 0;
    } ...
}
```

# Clase BurbujaSensible (2/2)

---

```
public class BurbujaSensible extends Burbuja {
    ...
    @Override
    public void ascender() {
        if (!isEnEspera()){

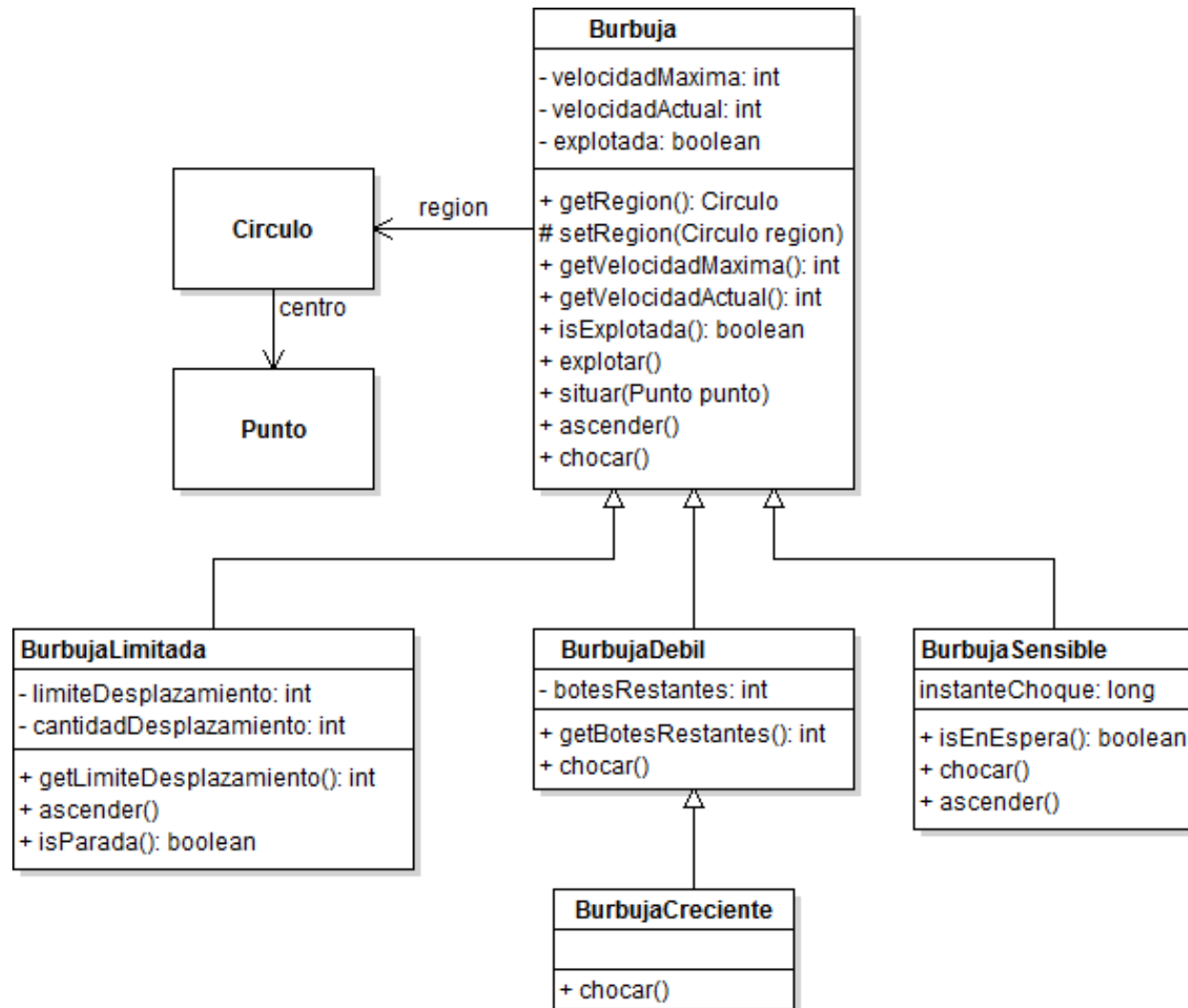
            super.ascender();

        } else {

            long ahora = System.currentTimeMillis();

            if (ahora - instanteChoque >= TIEMPO_ESPERA){
                super.ascender();
                instanteChoque = 0;
            }
        }
    }
}
```

# Jerarquía de herencia



# Polimorfismo

---

- ❑ Capacidad de una entidad (atributo, variable, parámetro) de **referenciar en tiempo de ejecución a objetos de diferentes clases**.
- ❑ Es **restringido por herencia**.
- ❑ El polimorfismo implica que una entidad tiene un **tipo estático** (declarado) y otro **dinámico** (al que referencia la entidad).

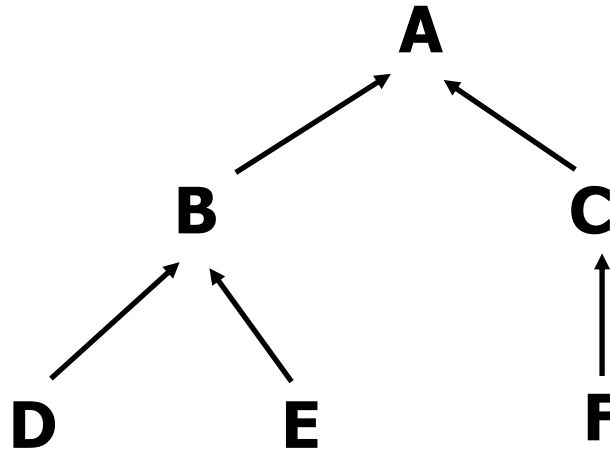


# Tipos estático y dinámico

---

- **Tipo estático** (*te*):
  - Tipo asociado a la declaración de una entidad.
  
- **Tipo dinámico**:
  - Tipo correspondiente a la clase del objeto conectado a la entidad en tiempo de ejecución.
  
- **Conjunto de tipos dinámicos** (*ctd*):
  - Conjunto de posibles tipos dinámicos de una entidad.

# Tipos estático y dinámico. Ejemplo



**A oa; B ob; C oc;**

**te(oa) = A**

**ctd(oa) = {A, B, C, D, E, F}**

**te(ob) = B**

**ctd(ob) = {B, D, E}**

**te(oc) = C**

**ctd(oc) = {C, F}**

# Polimorfismo

---

## ❑ Asignación polimórfica:

```
1. Burbuja burbuja = new Burbuja(...);  
2. BurbujaLimitada limitada = new BurbujaLimitada(...);  
  
// Asignación polimórfica  
3. burbuja = limitada;
```

- ❑ La variable `burbuja` tiene como **tipo estático** `Burbuja`.
- ❑ El **tipo dinámico** de la variable cambia:
  - Línea 1: clase `Burbuja`.
  - Línea 3: clase `BurbujaLimitada`.

# Herencia y sistema de tipos

---

- ¿Serían posibles las siguientes asignaciones?
  - Objeto `BurbujaLimitada` a variable `Burbuja`.
  - Objeto `BurbujaLimitada` a variable `BurbujaDebil`.
  - Objeto `BurbujaCreciente` a variable `Burbuja`.

# Compatibilidad de tipos

---

- Una clase (tipo) B es **compatible** con otra clase A si B es descendiente de A:
  - **BurbujaLimitada es compatible con Burbuja.**
  - **BurbujaCreciente es compatible con BurbujaDebil y Burbuja.**
  - **BurbujaLimitada NO es compatible con BurbujaDebil ni con BurbujaSensible.**

# Compatibilidad de tipos

---

- Una **asignación polimórfica** es **válida** si el **tipo estático** de la parte derecha es compatible con el tipo estático de la parte izquierda:

```
Burbuja burbuja = new BurbujaDebil (...);  
BurbujaLimitada limitada = new BurbujaLimitada (...);  
  
// Asignación polimórfica válida  
burbuja = limitada;  
  
// Asignación polimórfica NO válida  
BurbujaSensible sensible = limitada;
```

# Compatibilidad de tipos

---

- Un **paso de parámetros** es válido si el tipo estático del parámetro real es compatible con el tipo estático del parámetro formal.

```
public double simular (Burbuja burbuja) {  
    ...// simula el comportamiento de la burbuja  
}
```

```
BurbujaLimitada limitada = new BurbujaLimitada( ... );  
simulador.simular(limitada);
```

# Estructuras de datos polimórficas

---

- Organizar las clases en una jerarquía de herencia resulta útil para definir **estructuras de datos polimórficas**:

```
Burbuja[] burbujas = new Burbuja[3];  
burbujas[0] = new BurbujaLimitada (...);  
burbujas[1] = new BurbujaDebil (...);  
burbujas[2] = new BurbujaSensible (...);
```

- En el ejemplo, cada componente del array puede ser de un tipo distinto de burbuja.



# Validez de mensajes

---

- ¿Son válidos los siguientes mensajes?
  - Mensaje `getBotesRestates()` sobre una variable de tipo estático `Burbuja`.
  - Mensaje `ascender()` sobre una variable de tipo estático `BurbujaSensible`.

# Validez de mensajes

---

- ❑ **La herencia es consistente con el sistema de tipos.**
- ❑ Sobre una variable cuyo tipo estático es `BurbujaLimitada` podemos aplicar:
  - **Métodos heredados y redefinidos:** `isExplotada()`, `ascender()`, etc.
  - **Métodos propios:** `getLimiteDesplazamiento()`
- ❑ Si el tipo estático es `Burbuja` **no podemos aplicar métodos de los subtipos:**
  - `burbuja.getLimiteDesplazamiento(); //Error`
  - `burbuja.getBotesRestantes(); // Error`

# Validez de mensajes

---

```
Burbuja burbuja = new Burbuja(...);  
BurbujaSensible sensible = new BurbujaSensible(...);  
  
sensible.explotar();           //OK HEREDADO DE BURBUJA  
sensible.ascender();           //OK METODO REDEFINIDO  
sensible.isEnEspera();         //OK MÉTODO PROPIO  
  
burbuja = sensible;  
  
burbuja.isExplotada();         //OK METODO DE BURBUJA  
burbuja.isEnEspera();         //ERROR COMPILACION!
```

# Política pesimista de tipos

---

- El compilador **rechazaría** los siguientes casos debido a la **comprobación estática de tipos**:

```
Burbuja burbuja;  
BurbujaLimitada limitada = new ...;  
burbuja = limitada;  
limitada = burbuja; // Error de compilación
```

```
Burbuja burbuja;  
BurbujaSensible sensible = new ...;  
burbuja = sensible;  
burbuja.isEnEspera(); // Error de compilación
```

# Ligadura dinámica

---

- ¿Qué versión del método **ascender()** se ejecutaría?

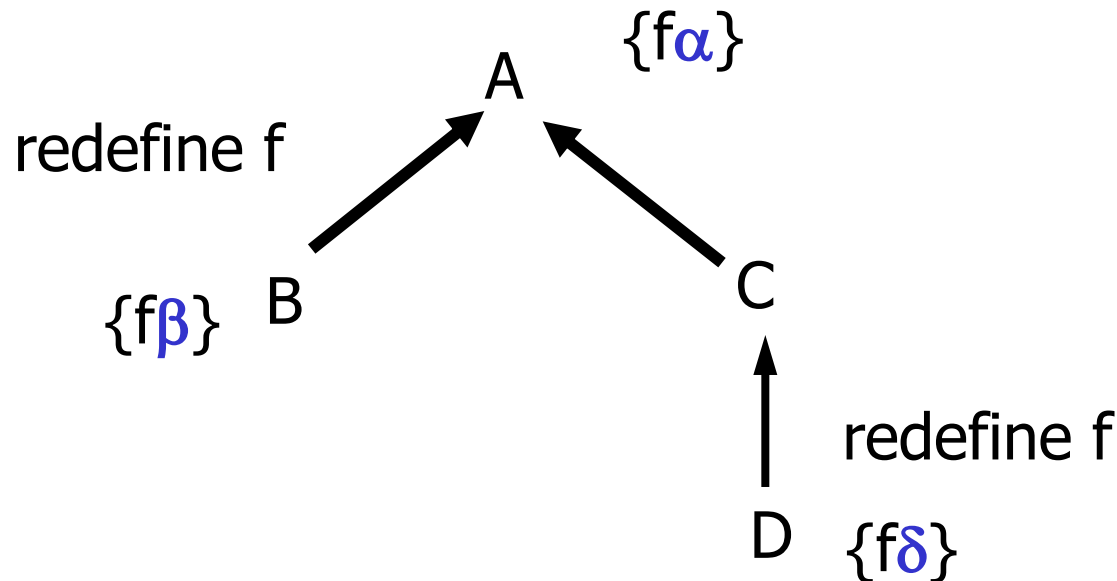
```
Burbuja burbuja;  
  
if (Math.random() > 0.5)  
    burbuja = new BurbujaLimitada(...);  
else  
    burbuja = new BurbujaDebil(...);  
  
burbuja.ascender();
```

# Ligadura dinámica

---

- ❑ La **versión de un método** en una clase es la introducida por la clase (redefinida) o la heredada.
- ❑ Versiones del método **ascender ( )** :
  - **Versión *Burbuja***: el método es definido en **Burbuja**
  - **Versión *BurbujaLimitada***: la clase **BurbujaLimitada** redefine el método.
  - La clase **BurbujaDebil** lo hereda.
  - La clase **BurbujaCreciente** hereda la versión de **BurbujaDebil**.
  - **Versión *BurbujaSensible***: la clase **BurbujaSensible** redefine el método.
- ❑ **Se ejecuta la versión asociada al tipo dinámico de la entidad sobre la que se aplica el método.**

# Ligadura dinámica y sistema de tipos



¿Qué versión se ejecuta?

```
A oa;  
B ob = new B();  
D od = new D();  
oa = ob;  
oa.f();  
  
oa = od;  
oa.f();
```

- Sea el mensaje  $x.f()$ , la **comprobación estática de tipos** garantiza que al menos existirá una versión aplicable para  $f$ , y la **ligadura dinámica** garantiza que se ejecutará la versión más apropiada.

# Ligadura dinámica. Caso de estudio

---

- ❑ Vamos a implementar un **simulador** para probar el comportamiento de las burbujas.

```
public static void simular (Burbuja burbuja) {...}
```

- ❑ ¿Qué métodos y versiones se ejecutan cuando se apliquen los métodos **ascender()** y **chocar()** sobre el parámetro burbuja?
- ❑ La ejecución de un método puede tener diferentes interpretaciones en tiempo de ejecución.
- ❑ En tiempo de ejecución, el código ejecutado varía según el tipo de burbuja al que sea aplicado (ligadura dinámica).



# Polimorfismo y ligadura dinámica

```
public class PruebaBurbujas {  
    private static final int LIMITE_ASCENSOS = 4;  
    private static final double PROBABILIDAD_EXPLOSION = 0.1;  
  
    private static void simular(Burbuja burbuja){ ... }  
  
    public static void main(String[] args) {  
        ... //Creamos las burbujas  
  
        //Estructura de datos polimórfica  
        Burbuja [] burbujas = {basica, limitada, debil,  
                                creciente, sensible};  
  
        for (Burbuja burbuja: burbujas)  
            simular(burbuja);  
    }  
}
```

¿Si aparece un nuevo tipo de burbuja?

# Polimorfismo y ligadura dinámica

```
public class PruebaBurbujas {  
    ...  
    private static void simular(Burbuja burbuja){  
        Random random = new Random();  
        int i = 0;  
        while (i < LIMITE_ASCENSOS && !burbuja.isExplotada()){  
            burbuja.ascender();  
  
            if (random.nextBoolean() == true)  
                burbuja.chocar();  
  
            if (random.nextDouble() < PROBABILIDAD_EXPLOSION)  
                burbuja.explotar();  
  
            System.out.println(burbuja.getInfo());  
            ++i;  
        }  
    }  
    ...  
}
```

# Método getInfo en Burbuja

---

```
public class Burbuja {  
  
    public String getInfo() {  
  
        return String.format("v. actual: %d, v. límite: %d, "  
            + "explotada: %b",  
            getVelocidadActual(),  
            getVelocidadMaxima(),  
            isExplotada());  
    }  
}
```

- El método se debe redefinir convenientemente en cada tipo de burbuja para mostrar sus propiedades (refinamiento).

# Redefinición getInfo

---

```
public class BurbujaDebil extends Burbuja {  
    ...  
    @Override  
    public String getInfo() {  
        return super.getInfo() +  
            " - botesRestantes: " + botesRestantes;  
    }  
}
```

```
public class BurbujaSensible extends Burbuja {  
    ...  
    @Override  
    public String getInfo() {  
        return super.getInfo() + " - EnEspera? " + isEnEspera();  
    }  
}
```

# Caso de estudio

---

- ❑ **Motivación:** ¿Cómo podemos consultar el número de botes restantes de todas las burbujas débiles?
- ❑ En la aplicación almacenamos todas las burbujas en un array:
  - `Burbuja[] burbujas;`
- ❑ Una posición del array puede apuntar a cualquier tipo de burbuja (**estructura de datos polimórfica**).
- ❑ Sobre una referencia de tipo estático **Burbuja** sólo puedo aplicar métodos de esa clase.
- ❑ **Solución:** casting (*narrowing*).

# Casting

---

- ❑ El compilador permite hacer un *casting* de una variable polimórfica **a uno de los posibles tipos dinámicos de la variable**: descendientes del tipo estático de la variable.
- ❑ Sería posible hacer un casting al tipo **BurbujaDebil** que tiene el método para consultar los botes restantes:

```
public static void main(String[] args) {  
    ... //Creamos las burbujas  
    for (Burbuja burbuja: burbujas){  
        BurbujaDebil debil = (BurbujaDebil)burbuja;  
        System.out.println(debil.getBotesRestantes());  
    }  
}
```

# Casting

---

- ❑ El casting de una variable de tipo objeto **NO realiza una conversión de objetos**
- ❑ Un casting de objetos debe entenderse como “el tipo dinámico de la variable es compatible con el tipo indicado en el casting”.
- ❑ Un casting **no compila** si el tipo indicado en el casting no es compatible (descendiente) con el tipo estático de la variable.
- ❑ Que un casting compile no implica que sea correcto. **Puede fallar en tiempo de ejecución.** ¿Por qué? En el ejemplo anterior falla.

# Operador `instanceof`

---

## ❑ Problema:

- Todas las burbujas no son débiles.
- El casting se resuelve en tiempo de ejecución y si es incorrecto aborta el programa.

## ❑ Solución: operador `instanceof`

- Permite consultar en tiempo de ejecución si el tipo dinámico de una variable **es compatible con** un tipo.
- Se entiende por tipo compatible **el tipo de la consulta o cualquiera de los subtipos.**



# Operador instanceof

---

```
public static void main(String[] args) {  
    ... //Creamos las burbujas  
    for (Burbuja burbuja: burbujas){  
        if (burbuja instanceof BurbujaDebil){  
            BurbujaDebil debil = (BurbujaDebil) burbuja;  
            System.out.println(debil.getBotesRestantes());  
        }  
    }  
}
```

# Operador instanceof

---

- A partir de la versión 14 es posible especificar la variable sobre la que se va a hacer un casting implícito.

```
public static void main(String[] args) {  
    ... //Creamos las burbujas  
    for (Burbuja burbuja: burbujas){  
        if (burbuja instanceof BurbujaDebil debil){  
            System.out.println(debil.getBotesRestantes());  
        }  
    }  
}
```

# Operador `instanceof`

---

- ❑ El operador `instanceof` también se puede utilizar con los registros (`Record`) para evaluar si un objeto es instancia de un tipo de clase registro.
- ❑ Además, permite extraer los valores de los atributos directamente, siguiendo el patrón del registro.

```
import static java.lang.Math.*;
public record Point(int x, int y) {
    ...
    static void printAngleFromXAxis(Object obj) {
        if (obj instanceof Point(int x, int y)) {
            System.out.println(toDegrees(atan2(y, x)));
        }
    }
}
```

# Consejos uso de la herencia

---

- ❑ Los atributos y métodos comunes deben situarse en clases altas de la jerarquía (**generalización**).
- ❑ Aplica herencia si entre dos clases existe la relación **es-un**.
- ❑ No debe utilizarse herencia salvo que todos los métodos heredados tengan sentido en la clase hija.
- ❑ En la **redefinición** de un método **no hay que cambiar la semántica** que tiene definida.
- ❑ Aplica **polimorfismo** y **ligadura dinámica** para evitar análisis de casos.