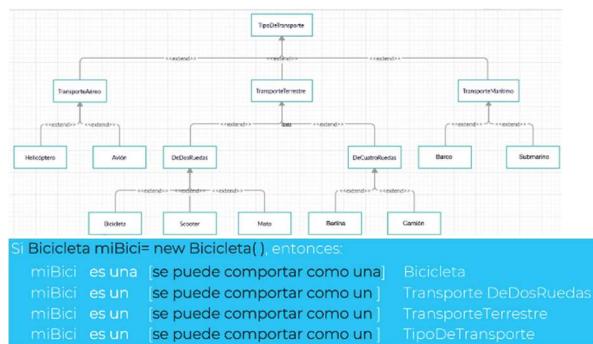


TEMA 5: POLIMORFISMO

POLIMORFISMO EN HERENCIA

- El POLIMORFISMO en herencia es el mecanismo por el cual un objeto se puede comportar de múltiples formas en cada momento, en función del contexto en el que aparece en el programa.
- Un objeto se puede comportar como una instancia de:
 - La clase a la que pertenece, es decir, aquella cuyo constructor se invocó a través de `new()`.
 - Una de las clases que se encuentran en un nivel superior a la clase a la que pertenece.
 - Esto incluye **clases abstractas**, pues la restricción impuesta sobre ellas es que no se pueden instanciar (usar `new()` para invocar sus constructores), pero en este caso el objeto se estaría inicializando usando constructores de otras clases.
 - Que un objeto se comporte como una de sus clases base abstracta sirve para restringir los métodos que puede invocar.



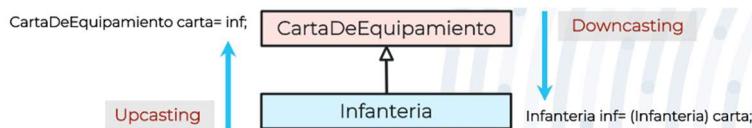
- 1 `InfanteriaEuropa infEuropa= new InfanteriaEuropa("AMARILLO");`
 - 2 `Infanteria infan= infEuropa;`
 - 3 `CartaDeEquipamiento cartaEquip= infEuropa;`
- En la línea 1 se crea `infEuropa`, un objeto de la clase `InfanteriaEuropa`. En la línea 2 se fuerza a que `infEuropa` se comporte como un objeto de la clase `Infanteria`, que es la clase base de `InfanteriaEuropa`. En la línea 3 se fuerza a que `infEuropa` se comporte como un objeto de la clase `CartaDeEquipamiento`, aún cuando esa clase es tipo abstracta.
- `CartaDeEquipamiento cartaEquip= infEuropa;`
- `CartaDeEquipamiento cartaEquip= new CartaDeEquipamiento();`

UPCASTING Y DOWNCASTING

- Cuando se hace que un objeto se comporte como una clase distinta a la que pertenece, se está **forzando un CASTING**, un cambio de tdd, en él.
- El casting no implica una nueva reserva de memoria: el objeto sigue siendo el mismo, independientemente de los castings que se hagan sobre él.
- El casting afecta a la visibilidad de los atributos y métodos que se pueden invocar desde el objeto.

Los únicos métodos y atributos accesibles por el objeto al que se le ha hecho casting son:

- Aquellos definidos explícitamente en la clase a la cual se ha realizado el cast.
- Aquellos que hereda (y tiene acceso) la clase a la que se hizo el cast.



UPCASTING

- En **UPCASTING** un objeto de una clase derivada se comporta como su clase base en la jerarquía.
 - When se realiza un upcasting no es necesario indicar de forma explícita a qué clase base se realiza el cast, pues sólo hay una.
- Es el mecanismo de polimorfismo más habitual, ya que es natural pensar que un objeto de una clase derivada se comporte como su clase base.
- Es un **concepto clave en POO**, pues facilita la toma de decisiones respecto al diseño del programa: si se necesita usar upcasting, entonces se deberá usar herencia.

`CartaDeEquipamiento carta= inf;` No es necesario incluir el cast (`CartaDeEquipamiento`) inf

- Después de hacer upcasting, el objeto ya no puede acceder a los métodos y atributos propios de la clase a la que pertenece, pues una clase base no tiene acceso a sus derivadas.
- Sin embargo, si la clase a la que pertenece el objeto tenía algún método sobrescrito, el objeto usará la implementación de la clase a la que pertenece y no la de su clase base.
 - Si se usase la implementación de la clase base, no existiría el polimorfismo con clases abstractas.
 - Por tanto, para aprovechar el potencial del upcasting se debe tener mucho cuidado escogiendo cuáles métodos se deben sobrescribir en la clase derivada y cuáles son propios.
 - Por eso las clases abstractas son tan útiles, pues permiten definir que métodos tienen que sobrescribir sus derivadas, facilitando el uso del upcasting para simplificar el código.
- El upcasting nunca producirá un error, pues con la herencia se garantiza que las clases derivadas tienen, por lo menos, los mismos métodos que las clases base.

```

public class Directivo extends Empleado {
    private ArrayList<Proyecto> proyectosDirigidos;

    @Override
    public float calcularSueldo() {
        float sueldo= super.calcularSueldo();
        return 1.1f*sueldo;
    }

    public ArrayList<Proyecto> proyectosConExito() {
        ArrayList<Proyecto> proyectosConExito= new ArrayList<>();
        for(Proyecto proyecto : this.proyectosDirigidos) {
            if(proyecto.getExito())
                proyectosConExito.add(proyecto);
        }
        return proyectosConExito;
    }
}
  
```

```

public static void main(String[] args) {
    Directivo directivo= new Directivo();
    Empleado empleadoDirectivo= directivo;
    empleadoDirectivo.
        *>calcularSueldo()
        <*>getAntiguedad()
        <*>getBase()
        <*>getClass()
        <*>getNombre()
        <*>getProyectosParticipado()
        <*>getSalario()
        <*>notify()
        <*>notifyAll()
        <*>getAntiguedad(float antiguedad)
        <*>setBase(float base)
        <*>setNombre(String nombre)
        <*>setProyectosParticipado(ArrayList<Proyecto> proyectos)
        <*>toString()
        <*>wait()
        <*>wait(long timeout)
  
```

La clase `Empresa` dispone de un atributo que es un `ArrayList` en el cual, a través de upcasting, se almacenan los empleados de la empresa, independientemente del tipo de empleado

El método `presupuesto()` obtiene el presupuesto de la empresa como suma del salario de los empleados. Para ello, se invoca el método `calcularSueldo()` que es común a todas las clases de la jerarquía

```

public static void main(String[] args) {
    Empresa empresa= new Empresa();
    ArrayList<Empleado> empleados= new ArrayList<>();
    empleados.add(new PuestoBase("EPB1", 12));
    empleados.add(new PuestoBase("EPB2", 6));
    empleados.add(new PuestoBase("EPB3", 4));
    empleados.add(new Directivo("ED1", 20));
    empleados.add(new Directivo("ED2", 14));
}
  
```

```

public class Directivo extends Empleado {
    @Override
    public float calcularSueldo() {
        return 1.1f*super.calcularSueldo();
    }
}
  
```

```

public class PuestoBase extends Empleado {
    @Override
    public float calcularSueldo() {
        return 1.025f*super.calcularSueldo();
    }
}
  
```

```

public class Directivo extends Empleado {
    private ArrayList<Proyecto> proyectosDirigidos;

    @Override
    public float calcularSueldo() {
        float sueldo= super.calcularSueldo();
        return 1.1f*sueldo;
    }
}
  
```

directivo es un objeto de la clase `Directivo`, que tiene implementado el método `proyectosConExito()`. Sin embargo, cuando se realiza upcasting y directivo se comporta como un `Empleado`, dicho método deja de ser visible para el objeto y no podrá ser invocado

```

public static void main(String[] args) {
    Directivo directivo= new Directivo();
    Empleado empleadoDirectivo= directivo;
    System.out.println(empleadoDirectivo.calcularSueldo());
}
  
```

UPCASTING

- En **UPCASTING** un objeto de una clase derivada se comporta como su clase base en la jerarquía.
 - ↳ Cuando se realiza un upcasting no es necesario indicar de forma explícita a qué clase base se realiza el cast, pues sólo hay una.
- El upcasting es el mecanismo de polimorfismo más **habitual**, ya que es **natural** pensar que un objeto de una clase derivada se comporta como su clase base.
- El upcasting **nunca producirá un error**, pues con la herencia se garantiza que las clases derivadas tienen, por lo menos, los mismos métodos que las clases base.

CartaDeEquipamiento carta= inf;

No es necesario incluir el cast
(CartaDeEquipamiento) inf

- Despues de hacer upcasting, el objeto ya **no puede acceder a los métodos y atributos propios** de la clase a la que pertenece, pues una clase base no tiene acceso a sus derivadas.
- Sin embargo, si la clase a la que pertenece el objeto tenía algún método **sobrescrito**, el objeto usará la implementación de la clase a la que pertenece y no la de su clase base.
 - ↳ Si se usase la implementación de la clase base, no existiría el polimorfismo con clases abstractas.
 - Por tanto, para aprovechar el potencial del upcasting se debe tener mucho cuidado escogiendo cuáles métodos se deben sobrescribir en la clase derivada y cuáles son propios.
 - Por eso las **clases abstractas** son tan útiles, pues permiten definir que métodos tienen que sobrescribir sus derivadas, facilitando el uso del upcasting para simplificar el código.
- El upcasting es un **concepto clave en POO**, pues facilita la toma de decisiones respecto al diseño del programa: si se necesita usar upcasting, entonces se deberá usar herencia.

DOWNCASTING

- En **DOWNCASTING** un objeto de una clase base se comporta como una de sus clases derivadas de la jerarquía.
 - ↳ Cuando se realiza un downcasting es necesario indicar de forma explícita a qué clase derivada se realiza el cast:
$$\text{Clase_derivada} < \text{nombre_obj_deriv} > = (\text{Clase_base}) < \text{nombre_obj_base} >$$
- Normalmente, sólo se usa cuando se necesita **deshacer el resultado de un upcasting anterior**, pues **fuerza el comportamiento** de los objetos, ya que normalmente no se puede asegurar que un objeto de una clase base se comporte como una de sus clases derivadas.
- Es una operación **no segura (unsafe)**, ya que puede generar una excepción en tiempo de ejecución (*ClassCastException*) pues no se puede garantizar que el objeto disponga de los métodos y atributos de la clase a la que se castea.
 - En tiempo de diseño, el compilador no comprueba si el cast se realiza correctamente, ni siquiera comprueba si ambas clases pertenecen a la misma jerarquía.
 - En tiempo de ejecución tiene lugar el casteo, pero sólo se comprueba si el objeto pertenece a la clase derivada a la que se castea.
 - ↳ Aunque la clase a la que pertenece el objeto soporte los mismos métodos y atributos que la clase derivada, si no lo es, se generará la excepción.
- Por tanto, se debe **comprobar en tiempo de ejecución** que la clase a la que pertenece el objeto es la clase a la que se realiza en casting usando el operador *instanceof*.
$$< \text{nombre_objeto} > \text{ instanceof } < \text{nombre_paquete} >. < \text{nombre_clase} >$$

```
public static void main(String[] args) {  
    Directivo directivo= new Directivo();  
    Empleado empleadoDirectivo= directivo;  
    PuestoBase puesto= (PuestoBase) empleadoDirectivo;  
}
```

La clase **Empresa** tiene un atributo que es un **ArrayList** en el que se almacenan los empleados de la empresa, independientemente del tipo de empleado

```
public static void main(String[] args) {  
    Empresa empresa= new Empresa();  
    ArrayList<Empleado> empleados= new ArrayList<>();  
    empleados.add(new PuestoBase("PFB1", 12));  
    empleados.add(new PuestoBase("PFB2", 6));  
    empleados.add(new PuestoBase("PFB3", 4));  
    empleados.add(new Directivo("ED1", 20));  
    empleados.add(new Directivo("ED2", 14));  
    ArrayList<Proyecto> proyectos= empresa.proyectosConExito();
```

proyectosConExito() es un método propio de la clase **Directivo** que obtiene los proyectos que han sido dirigidos por directivos y que han acabado en éxito

```
public ArrayList<Proyecto> proyectosConExito() {  
    ArrayList<Proyecto> proyectos= new ArrayList<>();  
    for(int i=0;i<this.empleados.size();i++) {  
        if(this.empleados.get(i) instanceof Directivo) {  
            Directivo directivo= (Directivo) this.empleados.get(i);  
            proyectos.addAll(directivo.proyectosConExito());  
        }  
    }  
    return proyectos;
```

directivo es un objeto de **Directivo** sobre el que se realiza **upcasting** con la clase **Empleado**, que es la clase base de **Directivo**, de modo que el objeto se renombra con otra referencia, **empleadoDirectivo**.

Posteriormente, se realiza **downcasting** del objeto con la clase **PuestoBase**, que también es una clase derivada de **Empleado**.

```
public Proyecto mayorPresupuesto(Empleado empleado) {  
    if(empleado instanceof empresa.Directivo) {  
        Directivo directivo= (Directivo) empleado;  
        Proyecto proyecto= directivo.mayorProyecto();  
        System.out.println(proyecto);  
        return proyecto;  
    }  
    return null;  
}
```

instanceof comprueba si el **empleadoDirectivo** tiene como tipo de dato **Directivo**. En caso de que el resultado sea cierto, se puede invocar al método **mayorProyecto**, que está definido en dicha clase

BENEFICIOS DEL POLIMORFISMO

- Facilita la **simplificación del código** ya que, como los objetos se pueden manejar como objetos de su clase base, la invocación de los métodos comunes a las clases derivadas se realiza desde una sola clase (la clase base).
- Facilita la **extensibilidad de los programas** ya que incluir nuevas clases que implementen los métodos comunes de las clases base no supondrá modificaciones en el código.
 - ↳ Eso no se cumple con **downcasting**, ya que incluir nuevas clases puede implicar cambios en el código, pero serían simplemente extensiones en los métodos en los que ya se manejan las distintas clases derivadas de las clases base.

- En la medida de lo posible se debe favorecer el uso del polimorfismo pues sus beneficios son muy grandes.
- En los conjuntos de datos (como **ArrayList** o **HashMap**) se deben usar las clases derivadas para almacenar los objetos.