5.
Polimorfismo

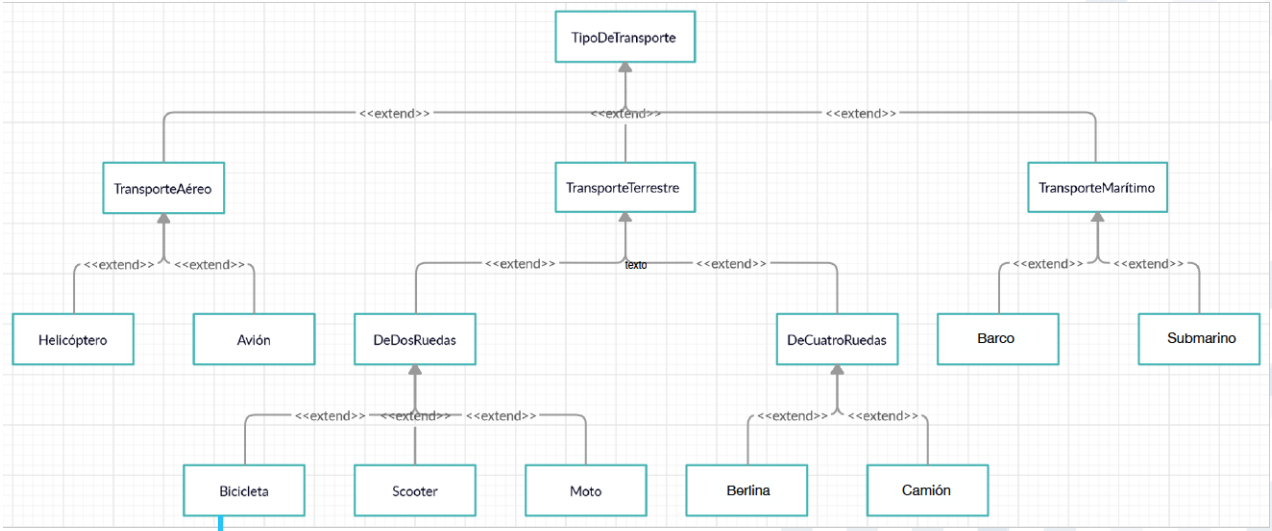
Herencia y simplificación de código

© Manuel Lama (manuel.lama@usc.es)

Polimorfismo en herencia

- El polimorfismo en herencia es el mecanismo mediante el cual un objeto se puede comportar de múltiples formas en cada momento, en función del contexto en el que parece dicho objeto dentro del programa
  - Un objeto se puede comportar como una instancia de la clase a la que pertenece, es decir, de la clase cuyo constructor se invoca a través de new
  - Un objeto se puede comportar como una instancia de alguna de las clases que se encuentran en un nivel superior de la jerarquía en relación a la clase a la cual pertenece realmente el objeto
- El polimorfismo está relacionado con la herencia y con el uso de los métodos de las clases que se heredan/sobrescriben

Polimorfismo en herencia



Si Bicicleta miBici= new Bicicleta( ), entonces:

miBici	es una	[se puede comportar como una]	Bicicleta
miBici	es un	[se puede comportar como un]	Transporte DeDosRuedas
miBici	es un	[se puede comportar como un]	TransporteTerrestre
miBici	es un	[se puede comportar como un]	TipoDeTransporte

Polimorfismo en herencia


```
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 Infantería, 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 de tipo abstracta

Polimorfismo en herencia

- ¿Existe una contradicción en el hecho de que una clase sea abstracta y que un objeto se comporte como dicha clase?
  - La restricción que se impone sobre una clase abstracta es que no se pueden crear objetos de dicha clase; en otras palabras, no se puede usar new para invocar a los constructores de las clases abstracta

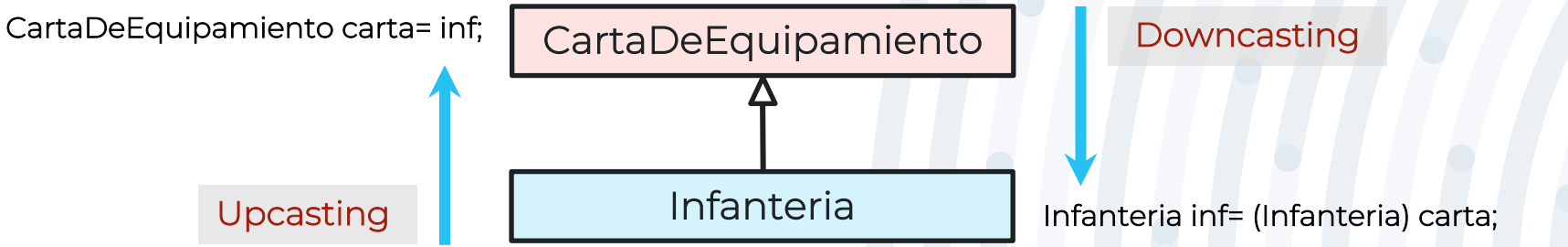
```
CartaDeEquipamiento cartaEquip= infEuropa;
```

```
CartaDeEquipamiento cartaEquip= new CartaDeEquipamiento();
```

  - Las clases abstractas no solamente se usan para estructurar las jerarquías de clases, sino que también para restringir los métodos que pueden invocar los objetos

Upcasting & Downcasting

- Cuando se indica que un objeto se comporta como una clase de la jerarquía diferente a la que pertenece, se está forzando un cambio en el tipo de dato del objeto
  - En upcasting un objeto de una clase derivada se comporta como una de las clases base de la jerarquía
  - En downcasting un objeto de una clase base se comporta como una de las clases derivadas de la jerarquía



Upcasting & Downcasting

- Este cambio de tipo de dato no implica una nueva reserva de memoria para el objeto, sino que el objeto seguirá siendo el mismo, independientemente de los cambios de tipo de dato que tengan lugar a lo largo del programa
- El casting de objetos no solamente conlleva un cambio en el tipo de dato, sino que también afecta a la accesibilidad de los atributos y de los métodos que se pueden invocar desde el objeto
  - Los únicos métodos que serán accesibles por el objeto son (1) los que están definidos en la clase a la cual se ha realizado el cast; y (2) los que hereda dicha clase de las clases base de la jerarquía

Upcasting

- Upcasting es el mecanismo más habitual con el que se facilita el polimorfismo, ya que parece natural pensar que un objeto de la clase derivada se comporta como una de sus clase base
  - Cuando se realiza un upcasting **no es necesario indicar de forma explícita un cast** entre la clase derivada y la clase base

```
CartaDeEquipamiento carta= inf;
```

No es necesario incluir el cast (CartaDeEquipamiento) inf

- Los métodos y atributos que son propios de la clase a la que pertenece el objeto no son accesibles, es decir, se pierde la visibilidad de esos métodos y atributos, puesto que el objeto deja de comportarse como esa clase

Upcasting

```java
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;
    }
```

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

```
calcularSueldo()                                   float
equals(Object obj)                                 boolean
getAntiguedad()                                    float
getBase()                                          float
getClass()                                         Class<?>
getNombre()                                        String
getProyectosParticipado()                          ArrayList<Proyecto>
hashCode()                                         int
notify()                                           void
notifyAll()                                        void
setAntiguedad(float antiguedad)                    void
setBase(float base)                                void
setNombre(String nombre)                           void
setProyectosParticipado(ArrayList<Proyecto> proyectos) void
toString()                                         String
wait()                                             void
wait(long timeout)                                 void
```

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

Upcasting

- Si un método está sobrescrito cuando se realiza upcasting, el objeto usará la implementación del método correspondiente a la clase a la que pertenece el objeto, de modo que aunque se comporta como otra clase, no utiliza las implementaciones de los métodos de esa otra clase
  - Si se usase la implementación de los métodos correspondientes a la clase sobre la que se realiza upcasting, no existiría polimorfismo cuando se tratase con clases abstractas

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

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

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

Upcasting

- Upcasting es un concepto clave en programación orientada a objetos, puesto que facilita la toma de decisiones en relación al diseño del programa
  - Si en un programa es necesario el uso de upcasting, entonces se deberá de aplicar el mecanismo de herencia

```
public class Jugador {
    private String nombre;
    private ArrayList<Pais> paises;
    private ArrayList<Infanteria> cartasInfanteria;
    private ArrayList<Caballeria> cartasCaballeria;
    private ArrayList<Artilleria> cartasArtilleria;
```

```
public class Jugador {
    private String nombre;
    private ArrayList<Pais> paises;
    private ArrayList<CartaDeEquipamiento> cartas;
```

Con upcasting no es necesario (1) usar un ArrayList para cada uno de los tipos de cartas de equipamiento; y (2) distinguir en los métodos qué lista se usa en cada momento. Con herencia + upcasting se simplifica de forma significativa el diseño del programa

Upcasting

- El mecanismo de upcasting nunca producirá un error, ya que con la herencia se garantiza que las clases derivadas tienen los mismos métodos que la clase base
  - Una vez se ha realizado upcasting sobre un objeto, no hay ningún modo de acceder a los métodos específicos de la clase a la cual pertenece dicho objeto
  - Para aprovechar todo el potencial del upcasting el diseño de los programas debe ser muy cuidadoso, eligiendo qué métodos se deben sobrescribir y cuáles son específicos de las clases
  - Las clases abstractas permiten definir qué métodos es necesario que tengan las clases que derivan de ellas, facilitando con ello el uso del upcasting para la simplificación de los programas

Upcasting

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

```
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));
```

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 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();
    }
```

Downcasting

- Con downcasting se está forzando el comportamiento de los objetos en las jerarquías de clases, puesto que por lo general **no se puede asegurar** que un objeto de una clase base se comporta como una clase derivada
  - Realizar downcastig no es una operación segura (unsafe), sino que es una operación en la cual se puede generar un error o una excepción en tiempo de ejecución

ClassCastException	Se genera cuando un cast de un objeto no está permitido al ser incompatible

  - Típicamente, se realiza downcasting cuando se necesita deshacer el resultado del upcasting, es decir, cuando se quiere recuperar la asignación del objeto a la clase a la cual pertenece

Downcasting

- Cuando se realiza downcasting es necesario indicar de forma explícita a qué clase derivada se "convertirá" el objeto de la clase base, es decir, en el código deberá de especificar el tipo de datos del objeto
  - Se usa un cast con el que se fuerza la conversión de los tipos de datos desde la clase base a la derivada, de modo que ese cast se especificará con el siguiente formato: *(<nombre_clase_derivada>)*

```
Infanteria inf= (Infanteria) carta;
```

  - El compilador no comprueba en tiempo de diseño si el cast tendrá lugar de forma correcta, ni tan siquiera comprueba que el objeto se encuentre en la misma jerarquía que la clase derivada a la que se "convierte"

Downcasting

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

En tiempo de ejecución se genera una excepción **ClassCastException**, ya que **se está tratando** un objeto de la clase **Directivo** como si fuese de la clase **PuestoBase**

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 **donwcasting** del objeto con la clase **PuestoBase**, que también es una clase derivada de **Empleado**

Downcasting

- El principal problema de la operación de downcasting es que no se puede garantizar que el objeto disponga de los mismos métodos y atributos que los de la clase derivada a la cual se realiza el casting
  - La operación de casting tiene lugar en tiempo de ejecución y en ella solo se comprueba si el tipo del objeto es la clase derivada
  - Aunque el objeto soporte los mismos métodos y atributos que los de la clase derivada, si el tipo del objeto no es la clase derivada, la operación de downcasting generará una excepción
  - Se debe comprobar en tiempo de ejecución que el tipo de dato del objeto sea la propia clase derivada a la que se realiza el casting

operador **instanceof**

Downcasting

- El operador instanceof determina en tiempo de ejecución si el tipo de dato de un objeto es una clase dada, de manera que si devuelve cierto se podrán invocar los métodos de dicha clase

<nombre_objeto> instanceof <nombre_paquete>.<nombre_clase>

```
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

Downcasting

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("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));
    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;
}
```

Beneficios del polimorfismo

- Facilita la simplificación del código del programa, puesto que los objetos se pueden manejar como objetos de las clases base, de modo que la invocación de los métodos comunes a todas las clases se realiza desde una misma clase (la clase base)
- Facilita la extensibilidad de los programas, ya que la inclusión de nuevas clases en la jerarquía que implementan los métodos comunes de las clases base no supondrá modificaciones en el código del programa
  - Con downcasting la introducción de nuevas clases podría implicar cambios en el código del programa, pero serían extensiones a los métodos en los que ya se manejan las distintas clases derivadas de las clases base

Polimorfismo en herencia

- Buenas prácticas de programación (XXVIII)

  En los conjuntos de datos, como ArrayList o HashMap, se deben de usar las clases derivadas para almacenar los objetos de las clases de la jerarquía

- Buenas prácticas de programación (XXIX)

  En el ula medida de lo posible se debe favorecer el uso de polimorfismo, puesto que promueve la simplificación de código y la mejora en el diseño de los programas