



Ministerio de Producción
Presidencia de la Nación

Ministerio de Educación y Deportes

Subsecretaría de Servicios Tecnológicos y Productivos



**PROGRAMACIÓN ORIENTADA
A OBJETOS**



Programación Orientada a Objetos

Temas:

Casting de tipos primitivos

Casting de objetos

Downcasting

Upcasting

Polimorfismo

Dynamic Binding

Casting

El **casting** significa **convertir tipos de datos**. Es posible aplicar **casting** a variables de tipo primitivo para obtener un valor convertido a otro tipo primitivo (por ej. de un número decimal a un entero) o para convertir objetos de una clase a otra clase.

El **operador de casting** tiene un solo operando a la derecha que es la expresión que se desea convertir y se representa encerrando entre paréntesis un tipo de datos. Ej: (int), (float), (String)

Casting sobre tipos primitivos:

¿Qué imprime?

```
public class Prueba {  
  
    public static void main(String[] args) {  
        int num1 = 100; //ocupa 4 bytes  
        short num2 = (short) num1; // num1 no entra en num2 ya que short tiene 2 bytes.  
                                //Se debe realizar un casteo explicito  
        System.out.println(num1);  
        System.out.println(num2);  
    }  
}
```

CASTING

Casting

El **casting** significa **convertir tipos de datos**. Es posible aplicar **casting** a variables de tipo primitivo para obtener un valor convertido a otro tipo primitivo (por ej. de un número decimal a un entero) o para convertir objetos de una clase a otra clase.

El **operador de casting** tiene un solo operando a la derecha que es la expresión que se desea convertir y se representa encerrando entre paréntesis un tipo de datos. Ej: (int), (float), (String)

Casting sobre tipos primitivos:

```
public class Prueba {  
  
    public static void main(String[] args) {  
        int num1 = 100; //ocupa 4 bytes  
        short num2 = (short) num1; // num1 no entra en num2 ya que short tiene 2 bytes.  
                                //Se debe realizar un casteo explicito  
        System.out.println(num1);  
        System.out.println(num2);  
    }  
}
```

CASTING

¿Qué imprime?

100
100

Casting

El **casting** significa **convertir tipos de datos**. Es posible aplicar **casting** a variables de tipo primitivo para obtener un valor convertido a otro tipo primitivo (por ej. de un número decimal a un entero) o para convertir objetos de una clase a otra clase.

El **operador de casting** tiene un solo operando a la derecha que es la expresión que se desea convertir y se representa encerrando entre paréntesis un tipo de datos. Ej: (int), (float), (String)

Casting sobre tipos primitivos:

```
public class Prueba {  
  
    public static void main(String[] args) {  
        int num1=1000000;    // 4 bytes  
        short num2 = (short) num1; // num1 no entra en num2 ya que short tiene 2 bytes.  
                                //Se debe realizar un casteo explicito  
        System.out.println(num1);  
        System.out.println(num2);  
    }  
}
```

CASTING

¿Qué imprime?

100
100

¿Qué sucede si se sustituyera la primera línea `int num1=100` por `int num1=1000000`?

Casting

El **casting** significa **convertir tipos de datos**. Es posible aplicar **casting** a variables de tipo primitivo para obtener un valor convertido a otro tipo primitivo (por ej. de un número decimal a un entero) o para convertir objetos de una clase a otra clase.

El **operador de casting** tiene un solo operando a la derecha que es la expresión que se desea convertir y se representa encerrando entre paréntesis un tipo de datos. Ej: (int), (float), (String)

Casting sobre tipos primitivos:

```
public class Prueba {  
  
    public static void main(String[] args) {  
        int num1=1000000;    // 4 bytes  
        short num2 = (short) num1; // num1 no entra en num2 ya que short tiene 2 bytes.  
                                //Se debe realizar un casteo explicito  
        System.out.println(num1);  
        System.out.println(num2);  
    }  
}
```

CASTING

¿Qué imprime?

100
100

¿Qué sucede si se sustituyera la primera línea `int num1=100` por `int num1=1000000`?

1000000
16960

Casting

El **casting** significa **convertir tipos de datos**. Es posible aplicar **casting** a variables de tipo primitivo para obtener un valor convertido a otro tipo primitivo (por ej. de un número decimal a un entero) o para convertir objetos de una clase a otra clase.

El **operador de casting** tiene un solo operando a la derecha que es la expresión que se desea convertir y se representa encerrando entre paréntesis un tipo de datos. Ej: (int), (float), (String)

Casting sobre tipos primitivos:

```
public class Prueba {  
  
    public static void main(String[] args) {  
        int num1=1000000;    // 4 bytes  
        short num2 = (short) num1; // num1 no entra en num2 ya que short tiene 2 bytes.  
                                //Se debe realizar un casteo explicito  
        System.out.println(num1);  
        System.out.println(num2);  
    }  
}
```

CASTING

¿Qué imprime?

100
100

¿Qué sucede si se sustituyera la primera línea `int num1=100` por `int num1=1000000`?

1000000
16960

El código compila pero **hay pérdida de datos**, pues el 1000000 se escapa del rango del short [-32768, 32767]. El resultado es incoherente

Casting de Tipos Primitivos

De int a short:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ int i= 1000000000;
        short s= i;
        System.out.println(i);
        System.out.println(s);
    }
}
```

El literal 1000000000 es considerado de tipo **int**, es por eso que la compilación falla, requiere *casting*

Casting de Tipos Primitivos

De int a short:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ int i= 100000000;
        short s= i;
        System.out.println(i);
        System.out.println(s);
    }
}
```

El literal 100000000 es considerado de tipo **int**, es por eso que la compilación falla, requiere *casting*

La forma de solucionarlo es haciendo un *casting* explícito a short

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        int i= 100000000;
        short s= (short) i;
        System.out.println(i);
        System.out.println(s);
    }
}
```

Casting de Tipos Primitivos

De int a short:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ int i= 100000000;
        short s= i;
        System.out.println(i);
        System.out.println(s);
    }
}
```

El literal 100000000 es considerado de tipo **int**, es por eso que la compilación falla, requiere *casting*

La forma de solucionarlo es haciendo un *casting* explícito a short

¿Qué imprime?

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        int i= 100000000;
        short s= (short) i;
        System.out.println(i);
        System.out.println(s);
    }
}
```

Casting de Tipos Primitivos

De int a short:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        ✗ int i= 100000000;  
        short s= i;  
        System.out.println(i);  
        System.out.println(s);  
    }  
}
```

El literal 100000000 es considerado de tipo **int**, es por eso que la compilación falla, requiere *casting*

La forma de solucionarlo es haciendo un *casting* explícito a short

¿Qué imprime?

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int i= 100000000;  
        short s= (short) i;  
        System.out.println(i);  
        System.out.println(s);  
    }  
}
```

100000000
-7936

Casting de Tipos Primitivos

De int a short:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        ✗ int i= 100000000;  
        short s= i;  
        System.out.println(i);  
        System.out.println(s);  
    }  
}
```

El literal 100000000 es considerado de tipo **int**, es por eso que la compilación falla, requiere *casting*

La forma de solucionarlo es haciendo un *casting* explícito a short

¿Qué imprime?

100000000
-7936

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int i= 100000000;  
        short s= (short) i;  
        System.out.println(i);  
        System.out.println(s);  
    }  
}
```

El código compila, en este ejemplo **hay pérdida de datos**, pues el 100000000 se escapa del rango del short [-32768, 32767]. El resultado no refleja el valor original.

Casting de Tipos Primitivos

De int a byte:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 35;  
        System.out.println(b);  
    }  
}
```

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 350;  
        System.out.println(b);  
    }  
}
```

Teniendo en cuenta que el literal 35 es considerado de tipo **int**, si no *casteamos* a **byte** el programa NO compilaría

Casting de Tipos Primitivos

De int a byte:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 35;  
        System.out.println(b);  
    }  
}
```

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 350;  
        System.out.println(b);  
    }  
}
```

Teniendo en cuenta que el literal 35 es considerado de tipo **int**, si no *casteamos* a **byte** el programa NO compilaría

¿Qué imprime?

Casting de Tipos Primitivos

De int a byte:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 35;  
        System.out.println(b);  
    }  
}
```

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 350;  
        System.out.println(b);  
    }  
}
```

Teniendo en cuenta que el literal 35 es considerado de tipo **int**, si no *casteamos* a **byte** el programa NO compilaría

¿Qué imprime?

35

Casting de Tipos Primitivos

De int a byte:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 35;  
        System.out.println(b);  
    }  
}
```

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 350;  
        System.out.println(b);  
    }  
}
```

Teniendo en cuenta que el literal 35 es considerado de tipo **int**, si no *casteamos* a **byte** el programa NO compilaría

¿Qué imprime?

35

¿Qué imprime?

Casting de Tipos Primitivos

De int a byte:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 35;  
        System.out.println(b);  
    }  
}
```

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 350;  
        System.out.println(b);  
    }  
}
```

Teniendo en cuenta que el literal 35 es considerado de tipo **int**, si no *casteamos* a **byte** el programa NO compilaría

¿Qué imprime?

35

¿Qué imprime?

94

Casting de Tipos Primitivos

De int a byte:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 35;  
        System.out.println(b);  
    }  
}
```

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        byte b = (byte) 350;  
        System.out.println(b);  
    }  
}
```

Teniendo en cuenta que el literal 35 es considerado de tipo **int**, si no *casteamos* a **byte** el programa NO compilaría

¿Qué imprime?

35

¿Qué imprime?

94

El código compila, pero **hay pérdida de datos**, pues el 350 se escapa del rango del byte [-128, 127]. El resultado no refleja el valor original.

Casting de Tipos Primitivos

De int a char:

Es posible convertir variables de tipo entero en tipo char pues el tipo char está soportado por enteros.

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int a=35, b=12,d=13;  
        char c;  
        c = (char) (a + b + d);  
        System.out.println(c);  
    }  
}
```

Casting de Tipos Primitivos

De int a char:

Es posible convertir variables de tipo entero en tipo char pues el tipo char está soportado por enteros.

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int a=35, b=12,d=13;  
        char c;  
        c = (char) (a + b + d);  
        System.out.println(c);  
    }  
}
```

¿Qué imprime?

Casting de Tipos Primitivos

De int a char:

Es posible convertir variables de tipo entero en tipo char pues el tipo char está soportado por enteros.

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int a=35, b=12,d=13;  
        char c;  
        c = (char) (a + b + d);  
        System.out.println(c);  
    }  
}
```

¿Qué imprime?



Casting de Tipos Primitivos

De int a char:

Es posible convertir variables de tipo entero en tipo char pues el tipo char está soportado por enteros.

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int a=35, b=12,d=13;  
        char c;  
        c = (char) (a + b + d);  
        System.out.println(c);  
    }  
}
```

¿Qué imprime?

<

El resultado que se asignará a **c** es el valor 60 que representa al 61er caracter en la representación Unicode que es el caracter '<'.



Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        ✗ float num1=25.5;  
        System.out.println(num1);  
    }  
}
```



Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        ✗ float num1=25.5;  
        System.out.println(num1);  
    }  
}
```

El literal 25.5 es considerado de tipo **double**, es por eso que la compilación falla, requiere *casting*.

Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ float num1=25.5;
        System.out.println(num1);
    }
}
```

El literal 25.5 es considerado de tipo **double**, es por eso que la compilación falla, requiere *casting*.

La forma de solucionarlo es haciendo un *casting* explícito a float o agregarle la F al final del literal:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        float num1=(float)25.5; //float num1=25.5F;
        System.out.println(num1);
    }
}
```

Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ float num1=25.5;
        System.out.println(num1);
    }
}
```

El literal 25.5 es considerado de tipo **double**, es por eso que la compilación falla, requiere *casting*.

La forma de solucionarlo es haciendo un *casting* explícito a float o agregarle la F al final del literal:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        float num1=(float)25.5; //float num1=25.5F;
        System.out.println(num1);
    }
}
```

```
public class Prueba {
    public static void main(String[] args) {
        double num1=25.5;//ocupa 8 bytes
        float num2 = (float)num1;//num1 no entra en num2
        //float es de 4 bytes. Necesita casting explicito.
        float num3 = 25.5F;
        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
    }
}
```

Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ float num1=25.5;
        System.out.println(num1);
    }
}
```

El literal 25.5 es considerado de tipo **double**, es por eso que la compilación falla, requiere *casting*.

La forma de solucionarlo es haciendo un *casting* explícito a float o agregarle la F al final del literal:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        float num1=(float)25.5; //float num1=25.5F;
        System.out.println(num1);
    }
}
```

Se realiza un *casteo* explícito al tipo float

```
public class Prueba {
    public static void main(String[] args) {
        double num1=25.5;//ocupa 8 bytes
        float num2 = (float)num1;//num1 no entra en num2
        //float es de 4 bytes. Necesita casting explícito.
        float num3 = 25.5F;
        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
    }
}
```

Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ float num1=25.5;
        System.out.println(num1);
    }
}
```

El literal 25.5 es considerado de tipo **double**, es por eso que la compilación falla, requiere *casting*.

La forma de solucionarlo es haciendo un *casting* explícito a float o agregarle la F al final del literal:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        float num1=(float)25.5; //float num1=25.5F;
        System.out.println(num1);
    }
}
```

Se realiza un *casteo* explícito al tipo float

```
public class Prueba {
    public static void main(String[] args) {
        double num1=25.5;//ocupa 8 bytes
        float num2 = (float)num1;//num1 no entra en num2
        //float es de 4 bytes. Necesita casting explicito.
        float num3 = 25.5F;
        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
    }
}
```

De esta manera se indica que estoy asignando un **float** y **NO** es necesario hacer *casting*

Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ float num1=25.5;
        System.out.println(num1);
    }
}
```

El literal 25.5 es considerado de tipo **double**, es por eso que la compilación falla, requiere *casting*.

La forma de solucionarlo es haciendo un *casting* explícito a float o agregarle la F al final del literal:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        float num1=(float)25.5; //float num1=25.5F;
        System.out.println(num1);
    }
}
```

Se realiza un *casteo* explícito al tipo float

```
public class Prueba {
    public static void main(String[] args) {
        double num1=25.5;//ocupa 8 bytes
        float num2 = (float)num1;//num1 no entra en num2
        //float es de 4 bytes. Necesita casting explícito.
        float num3 = 25.5F;
        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
    }
}
```

De esta manera se indica que estoy asignando un **float** y **NO** es necesario hacer *casting*

¿Qué imprime?

Casting de Tipos Primitivos

De double a float:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        ✗ float num1=25.5;
        System.out.println(num1);
    }
}
```

El literal 25.5 es considerado de tipo **double**, es por eso que la compilación falla, requiere *casting*.

La forma de solucionarlo es haciendo un *casting* explícito a float o agregarle la F al final del literal:

```
package clase16.casting;
public class Prueba {
    public static void main(String[] args) {
        float num1=(float)25.5; //float num1=25.5F;
        System.out.println(num1);
    }
}
```

Se realiza un *casteo* explícito al tipo float

```
public class Prueba {
    public static void main(String[] args) {
        double num1=25.5;//ocupa 8 bytes
        float num2 = (float)num1;//num1 no entra en num2
        //float es de 4 bytes. Necesita casting explícito.
        float num3 = 25.5F;
        System.out.println(num1);
        System.out.println(num2);
        System.out.println(num3);
    }
}
```

De esta manera se indica que estoy asignando un **float** y **NO** es necesario hacer *casting*

¿Qué imprime?

25.5
25.5
25.5

Casting de Tipos Primitivos

De double a int:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        double doubVal;  
        int intVal;  
        doubVal = 2.8;  
        intVal = (int) doubVal; // casting an int  
        System.out.println(doubVal);  
        System.out.println(intVal);  
    }  
}
```

Casting de Tipos Primitivos

De double a int:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        double doubVal;  
        int intVal;  
        doubVal = 2.8;  
        intVal = (int) doubVal; // casting an int  
        System.out.println(doubVal);  
        System.out.println(intVal);  
    }  
}
```

Hay pérdida de datos, se pierde la parte fraccionaria del número decimal.

Casting de Tipos Primitivos

De double a int:

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        double doubVal;  
        int intVal;  
        doubVal = 2.8;  
        intVal = (int) doubVal; // casting an int  
        System.out.println(doubVal);  
        System.out.println(intVal);  
    }  
}
```

Hay pérdida de datos, se pierde la parte fraccionaria del número decimal.

Si se intenta castear a un valor más grande que el que puede ser almacenado en un **int**, el valor será transformado antes de ser guardado y el resultado NO refleja el valor original.



Casting de Tipos Primitivos

De int a long: no requiere casting

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int i= 100000000;  
        long j=i;  
        long k= 1000000000000000000L;  
        System.out.println(i);  
        System.out.println(j);  
        System.out.println(k);  
    }  
}
```

Casting de Tipos Primitivos

De int a long: no requiere casting

```
package clase16.casting;  
public class Prueba {  
    public static void main(String[] args) {  
        int i= 100000000;  
        long j=i;  
        long k= 100000000000000000L;  
        System.out.println(i);  
        System.out.println(j);  
        System.out.println(k);  
    }  
}
```

La promoción a un tipo más amplio NO requiere casting.

Como la variable **k** está declarada de **long** se produce un error en compilación si no le agregamos la **L** al final del número.

Casting de Objetos - Downcasting

```
public class Fecha extends Object {  
    private int dia = 1;  
    private int mes = 1;  
    private int año = 2006;  
  
    public boolean equals(Object o) {  
        boolean result=false;  
        if ((o!=null) && (o instanceof Fecha)) {  
            Fecha f=(Fecha) o;  
            if ((f.dia==this.dia) && (f.mes==this.mes) && (f.año==this.año))  
                result=true;  
        }  
        return result;  
    }  
}
```

El operador **instanceof** permite chequear el tipo real del objeto **o**

DOWNCASTING
casting descendente, desde
Object a una subclase

El método **equals()** **sobreescrito** tiene un argumento de tipo **Object**. Si sobre dicho objeto quisiéramos acceder a las variables de instancia y métodos disponibles del tipo **Fecha**, **NO** podríamos hacerlo.

Para poder comparar 2 objetos **Fecha** es necesario convertir el objeto **o** en un objeto de tipo **Fecha** mediante *casting* explícito al tipo **Fecha**.

Luego, es posible acceder a las variables de instancia y métodos disponibles de los objetos **Fecha**.

El DOWNCASTING es explícito

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

El mecanismo de **herencia** asegura que **TODOS** los métodos de la superclase están disponibles en las subclases: cualquier mensaje que le enviemos a un objeto de la superclase podrá también ser enviado a un objeto de la subclase.

En nuestro ejemplo, si los objetos **CuentaBancaria** tienen los métodos **extraer()**, **depositar()** y **getSaldo()**, **CajaDeAhorro** también los tendrá definido. **Esto nos asegura que un objeto CajaDeAhorro es de tipo CuentaBancaria.**

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

UPCASTING
casting ascendente, desde
la subclase a la superclase

Están visibles únicamente los métodos definidos en
CuentaBancaria

El **UPCASTING** es automático, no hay necesidad de hacerlo explícito

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

¿Qué métodos se ejecutan?

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

¿Qué métodos se ejecutan?

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

¿Qué métodos se ejecutan?

El heredado de CuentaBancaria

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

¿Qué métodos se ejecutan?

El heredado de CuentaBancaria

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

¿Qué métodos se ejecutan?

El heredado de CuentaBancaria

El sobreescrito en CajaDeAhorro

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

¿Qué métodos se ejecutan?

El heredado de CuentaBancaria

El sobreescrito en CajaDeAhorro

Casting de Objetos - Upcasting

```
public class CuentaBancaria {  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public double extraer(double monto) {...}  
    public double depositar(double monto) {...}  
}
```

```
public class CajaDeAhorro extends CuentaBancaria {  
    public double extraer(double monto) {  
        this.monto= this.monto - monto;  
        return this.monto;  
    }  
    public double depositar(double monto) {  
        this.monto= this.monto + monto;  
        return this.monto;  
    }  
    // Otros métodos de CajaDeAhorro  
}
```

UPCASTING

```
public class Banco {  
    public static void main(String[] args) {  
        CajaDeAhorro caja = new CajaDeAhorro();  
        CuentaBancaria cta=caja;  
        cta.getSaldo();  
        cta.depositar(1200);  
        cta.extraer(150);  
    }  
}
```

¿Qué métodos se ejecutan?

El heredado de CuentaBancaria

El sobreescrito en CajaDeAhorro

El sobreescrito en CajaDeAhorro

Casting de Objetos - Upcasting

El upcasting es seguro: pasamos de un tipo específico a uno más general.

Esto es, la subclase es un super conjunto de la superclase, podría contener más métodos que la superclase, pero al menos contendrá los métodos definidos en la superclase.

Lo único que puede ocurrir con el **upcasting** es que se reduzca la interface disponible de la clase, ahora tendrá menos métodos.

Es por esta razón que el compilador permite hacer **upcasting** sin un *casting* explícito.

UPCASTING

```
public class Banco {  
    public static void bonificar(CuentaBancaria c) {  
        c.depositar(100.00);  
    }  
    public static void main(String[] args) {  
        CajaAhorro caja = new CajaDeAhorro();  
        bonificar(caja);  
    }  
}
```

En el pasaje de parámetros se
está haciendo la conversión al
tipo de la superclase

El argumento del método **bonificar()** es de tipo **CuentaBancaria** y
el objeto con el que se invoca es de tipo **CajaDeAhorro**

¿Podemos enviarle como parámetro al **bonificar()** un objeto de tipo **CajaDeAhorro**?

Casting de Objetos - Upcasting

El upcasting es seguro: pasamos de un tipo específico a uno más general.

Esto es, la subclase es un super conjunto de la superclase, podría contener más métodos que la superclase, pero al menos contendrá los métodos definidos en la superclase.

Lo único que puede ocurrir con el **upcasting** es que se reduzca la interface disponible de la clase, ahora tendrá menos métodos.

Es por esta razón que el compilador permite hacer **upcasting** sin un *casting* explícito.

UPCASTING

```
public class Banco {  
    public static void bonificar(CuentaBancaria c) {  
        c.depositar(100.00);  
    }  
    public static void main(String[] args) {  
        CajaAhorro caja = new CajaDeAhorro();  
        bonificar(caja);  
    }  
}
```

En el pasaje de parámetros se está haciendo la conversión al tipo de la superclase

El argumento del método **bonificar()** es de tipo **CuentaBancaria** y el objeto con el que se invoca es de tipo **CajaDeAhorro**

¿Podemos enviarle como parámetro al **bonificar()** un objeto de tipo **CajaDeAhorro**? **SI!!**

Casting de Objetos - Upcasting

El upcasting es seguro: pasamos de un tipo específico a uno más general.

Esto es, la subclase es un super conjunto de la superclase, podría contener más métodos que la superclase, pero al menos contendrá los métodos definidos en la superclase.

Lo único que puede ocurrir con el **upcasting** es que se reduzca la interface disponible de la clase, ahora tendrá menos métodos.

Es por esta razón que el compilador permite hacer **upcasting** sin un *casting* explícito.

UPCASTING

```
public class Banco {  
    public static void bonificar(CuentaBancaria c) {  
        c.depositar(100.00);  
    }  
    public static void main(String[] args) {  
        CajaAhorro caja = new CajaDeAhorro();  
        bonificar(caja);  
    }  
}
```

En el pasaje de parámetros se está haciendo la conversión al tipo de la superclase

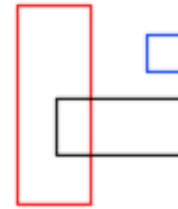
El argumento del método **bonificar()** es de tipo **CuentaBancaria** y el objeto con el que se invoca es de tipo **CajaDeAhorro**

¿Podemos enviarle como parámetro al **bonificar()** un objeto de tipo **CajaDeAhorro**? **SI!!**

Un objeto **CajaDeAhorro** es también de tipo **CuentaBancaria** y por lo tanto todos los métodos que se invoquen adentro de **bonificar()** están definidos en la **CajaDeAhorro**. El método **bonificar()** funciona bien para todos los objetos **CuentaBancaria** y de sus clases derivadas.

Polimorfismo

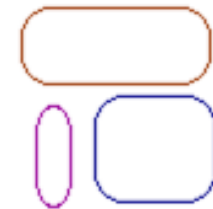
Ejemplo: un programa que dibuja diferentes formas geométricas en la pantalla.



Rectángulos



Óvalos



Rectángulos Redondeados

```
import java.awt.Color;
public class Figura {
// Color de la figura
    private Color color;
    public void setColor(Color newColor) {
        // Cambia el color de la figura
        color = newColor;
        dibujar();
    }
    public void dibujar() {
        System.out.println("Figura genérica-no"
            + "-se-dibujarme");
    }
}
```

Las clases **Rectangulo**, **Ovalo** y **RectanguloRedondeado**, representan los tres tipos de figuras que nos interesan. La clase **Figura** representa las características comunes de los tres tipos de figuras, es la superclase.

La clase **Figura** podría incluir variables de instancia como **color**, **posición** y **tamaño** de la figura.

La clase **Figura** podría incluir métodos de instancia para **cambiar el color**, **la posición** y **el tamaño** de una figura.

El cambio de color involucra cambiar el valor de la variable de instancia y luego, dibujarla con el nuevo color.

Sin embargo hay un problema con el método **dibujar()**: cada figura se dibuja de manera diferente.

Polimorfismo

Cada tipo de figura tiene su propia implementación del método **dibujar()**

```
public class Rectangulo extends Figura {  
  
    public void dibujar() {  
        System.out.println("Rectangulo.dibujar()");  
    }  
    // más métodos de instancia de Rectangulo  
}
```

```
public class RectRedondeado extends Figura {  
  
    public void dibujar() {  
        System.out.println("RecRedondeado.dibujar()");  
    }  
    // más métodos de instancia RectRedondeado  
}
```

```
public class Ovalo extends Figura {  
  
    public void dibujar() {  
        System.out.println("Ovalo.dibujar()");  
    }  
    // más métodos de instancia Ovalo  
}
```

dibujar() es un método
sobreescrito: un mismo
nombre de método con
diferente comportamiento

Polimorfismo

```
public class Dibujante {  
    public static void graficador(Figura[] figuras){  
        for (Figura f : figuras)  
            f.dibujar();  
    }  
}
```

`figuras` almacena diferentes tipos de figuras: 1 rectángulo, 1 óvalo y 1 rectángulo redondeado

```
public static void main(String args[]) {  
    Figura [] figuras=new Figura[3];  
    figuras[0]=new Rectangulo();  
    figuras[1]=new Ovalo();  
    figuras[2]=new RectRedondeado();  
    Dibujante.graficador(figuras);  
}
```

`figuras` es un arreglo de tipo **Figura**.

Cada vez que se ejecuta la sentencia **f.dibujar();**

el método que se ejecuta es el **sobreescrito** por las subclases de **Figura**: **Rectangulo**, **Ovalo** o **RectRedondeado** y **NO** la versión definida en **Figura**.

No hay manera de saber mirando el texto del programa qué figura se dibujará, pues depende del valor que tome la variable **f** en ejecución.

El método **dibujar()** tiene “diferentes formas”.

¿Cómo sabe el programa qué figura dibujar cada vez que se invoca el método **dibujar()**?

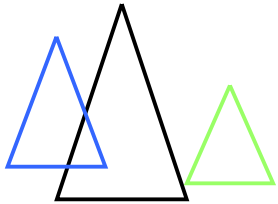
Observemos que aunque el tipo de la variable **f** es **Figura**, el programa tiene la habilidad de ejecutar el método correcto **resolviendo en ejecución el tipo real del objeto al que hace referencia la variable f**. Esta técnica de vincular un método con el código correcto se llama *binding* dinámico (lo veremos más adelante en esta clase).



Polimorfismo

- El **POLIMORFISMO** es uno de los pilares de la Programación Orientada a Objetos.
- **POLIMORFISMO** significa: “**un mismo nombre representa diferentes formas**”
- Los objetos son entidades activas que envían y reciben mensajes y el **POLIMORFISMO** es el **mecanismo que permite que diferentes objetos respondan al mismo mensaje de manera diferente.**
- El **POLIMORFISMO** es la capacidad de un método (mensaje) de tener diferente comportamiento dependiendo del objeto sobre el que está actuando (receptor del mensaje).
- En Java el **Polimorfismo** se manifiesta de la forma: **múltiples métodos con igual nombre y diferente comportamiento.**
- En algunos casos, múltiples métodos tienen el mismo nombre pero diferente lista de argumentos, es el caso de **métodos sobrecargados** y en otros casos múltiples métodos tienen el mismo nombre, tipo de retorno y lista de argumentos, se trata de **métodos sobreescritos.**
- En términos de programación, el polimorfismo en **Java** se manifiesta de 3 maneras:
 - **Métodos sobrecargados**
 - **Métodos sobreescritos a través de la herencia**
 - **Métodos sobreescritos a través de interfaces (aún no lo hemos visto)**
- Una de las principales ventajas del **POLIMORFISMO** es que facilita **la extensibilidad.**

Polimorfismo y Extensibilidad



Triángulos

En nuestra aplicación **Graficador** que dibuja figuras geométricas en pantalla, decidimos dibujar **triángulos**. Para ellos agregamos la clase **Triangulo** como subclase de **Figura**, y **sobreescribimos** el método **dibujar()**.

```
public class Dibujante {  
    public static void graficador(Figura[] figuras){  
        for (Figura f : figuras)  
            f.dibujar();  
    }  
  
    public static void main(String args[]) {  
        Figura figuras=new Figura[3];  
        figuras[0]=new Rectangulo();  
        figuras[1]=new Ovalo();  
        figuras[2]=new Triangulo();  
        figuras[3]=new RectRedondeado();  
        Dibujante.graficador(figuras);  
    }  
}
```

EXTENSIBILIDAD

El código que escribimos en nuestra aplicación dibujará triángulos, a pesar que la clase **Triangulo** no estaba definida cuando escribimos nuestra primera versión del **Graficador**.

```
public class Triangulo extends Figura {  
  
    public void dibujar() {  
        System.out.println("Triangulo.dibujar()");  
    }  
    // más métodos de instancia de Triangulo  
}
```

Polimorfismo y Binding Dinámico

¿Puede el compilador determinar cuál es el objeto real que invoca al método **dibujar()**?

```
public class Dibujante {  
    public static void graficador(Figura[] figuras){  
        for (Figura f : figuras)  
            f.dibujar();  
    }  
  
    public static void main(String args[]) {  
        Figura figuras=new Figura[3];  
        figuras[0]=new Rectangulo();  
        figuras[1]=new Ovalo();  
        figuras[2]=new Triangulo();  
        figuras[3]=new RectRedondeado();  
        Dibujante.graficador(figura);  
    }  
}
```

Polimorfismo y Binding Dinámico

¿Puede el compilador determinar cuál es el objeto real que invoca al método **dibujar()**?

NO!!

```
public class Dibujante {  
    public static void graficador(Figura[] figuras){  
        for (Figura f : figuras)  
            f.dibujar();  
    }  
  
    public static void main(String args[]) {  
        Figura figuras=new Figura[3];  
        figuras[0]=new Rectangulo();  
        figuras[1]=new Ovalo();  
        figuras[2]=new Triangulo();  
        figuras[3]=new RectRedondeado();  
        Dibujante.graficador(figura);  
    }  
}
```

Polimorfismo y Binding Dinámico

¿Puede el compilador determinar cuál es el objeto real que invoca al método **dibujar()**?

NO!!

Analizando el código es **imposible determinar qué tipo de figura geométrica se dibujará**. En compilación solamente se conoce el tipo de la variable que contiene una referencia a una figura real, pero NO el tipo real del objeto al que apunta la variable, eso depende **del valor que tome la variable en ejecución**.

```
public class Dibujante {  
    public static void graficador(Figura[] figuras){  
        for (Figura f : figuras)  
            f.dibujar();  
    }  
  
    public static void main(String args[]) {  
        Figura figuras=new Figura[3];  
        figuras[0]=new Rectangulo();  
        figuras[1]=new Ovalo();  
        figuras[2]=new Triangulo();  
        figuras[3]=new RectRedondeado();  
        Dibujante.graficador(figura);  
    }  
}
```


Polimorfismo y Binding Dinámico

¿Puede el compilador determinar cuál es el objeto real que invoca al método **dibujar()**?

NO!!

Analizando el código es **imposible determinar qué tipo de figura geométrica se dibujará**. En compilación solamente se conoce el tipo de la variable que contiene una referencia a una figura real, pero NO el tipo real del objeto al que apunta la variable, eso depende **del valor que tome la variable en ejecución**.

```
public class Dibujante {  
    public static void graficador(Figura[] figuras) {  
        for (Figura f : figuras)  
            f.dibujar();  
    }  
  
    public static void main(String args[]) {  
        Figura figuras=new Figura[3];  
        figuras[0]=new Rectangulo();  
        figuras[1]=new Ovalo();  
        figuras[2]=new Triangulo();  
        figuras[3]=new RectRedondeado();  
        Dibujante.graficador(figura);  
    }  
}
```

f es de tipo **Figura** pero apunta a diferentes figuras geométricas.

Con **polimorfismo basado en sobreescritura**, la versión del método **dibujar()** que se ejecutará está determinada por el tipo real del objeto apuntado por **f** y NO por el tipo de la variable **f** con la que se invoca a **dibujar()**.

La decisión de qué versión del **dibujar()** invocar NO se puede hacer en compilación, se difiere a ejecución. Esta técnica se conoce como **dynamic binding** o **late binding**.

Polimorfismo y Binding

Binding es el mecanismo que resuelve la invocación de un método con el cuerpo del método.

Si el **binding** lo puede resolver el compilador, se llama **binding temprano (early binding)**, este es el caso del **polimorfismo basado en sobrecarga**: el compilador puede determinar a qué método se invocará basándose en la cantidad, tipo y orden de los argumentos.

fuelle

```
m1(){  
m2(){  
m3(){  
main(){  
  m1();  
  m2();  
  .....  
}
```

compilador

compilado

```
.....  
.....  
Llamado a m1()  
Llamado a m2()  
.....  
.....
```

m1 ()

```
.....  
.....  
.....
```

m2 ()

```
.....  
.....  
.....
```

compilados

Early Binding

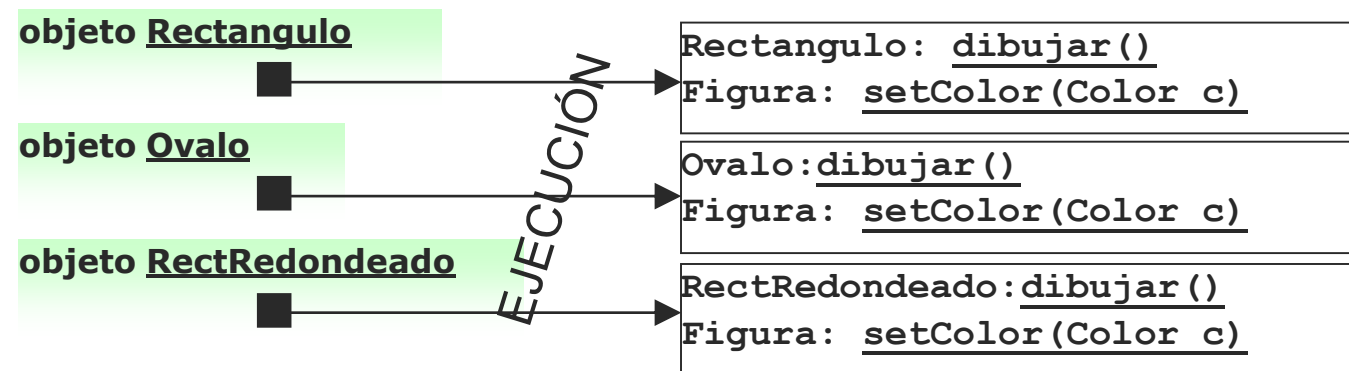
En compilación se resuelven todas las invocaciones a métodos

Se aplica a polimorfismo basado en sobrecarga

Polimorfismo y Binding

Binding es el mecanismo que resuelve la invocación de un método con el cuerpo del método.

Si el **binding** se hace en ejecución se denomina **binding tardío (late binding)** o **binding dinámico (dynamic binding)**. Es el caso del **polimorfismo basado en sobreescritura**.



Dynamic Binding



Resuelve el polimorfismo
basado en sobreescritura

Java soporta **Dynamic Binding** automáticamente.



Ejemplo: Héroes y Villanos

¿Cómo modelarían el siguiente enunciado?

Se desea modelar un juego compuesto por héroes y villanos.

Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza.

Adicionalmente, el juego debe proveer un mecanismo de agrupamiento de los personajes en ligas para realizar enfrentamientos entre grupos de personajes.

Cada liga puede estar compuesta tanto de personajes como de otras ligas.

Cada liga tiene su propio nombre de liga y un valor de fuerza.

La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.

Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje como de una liga.



Ejemplo: Héroes y Villanos

Empecemos a modelar...

“Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. [...] Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje [...].”

Ejemplo: Héroes y Villanos

Empecemos a modelar...

“Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. [...] Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje [...].”

Heroe
-nombre : String -fuerza : float
+Heroe(nombre : String, fuerza : float) +getFuerza() : float

Villano
-nombre : String -fuerza : float
+Villano(nombre : String, fuerza : float) +getFuerza() : float

Ejemplo: Héroes y Villanos

Empecemos a modelar...

“Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. [...] Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje [...].”

Heroe
-nombre : String -fuerza : float
+Heroe(nombre : String, fuerza : float) +getFuerza() : float

Villano
-nombre : String -fuerza : float
+Villano(nombre : String, fuerza : float) +getFuerza() : float

¿Son ambas clases necesarias?

Ejemplo: Héroes y Villanos

Empecemos a modelar...

“Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. [...] Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje [...].”

Heroe
-nombre : String -fuerza : float
+Heroe(nombre : String, fuerza : float) +getFuerza() : float

Villano
-nombre : String -fuerza : float
+Villano(nombre : String, fuerza : float) +getFuerza() : float

¿Son ambas clases necesarias?

NO!!!

Ejemplo: Héroes y Villanos

Empecemos a modelar...

“Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. [...] Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje [...].”

Heroe	Villano	
-nombre : String -fuerza : float	-nombre : String -fuerza : float	Mismos atributos!
+Heroe(nombre : String, fuerza : float) +getFuerza() : float	+Villano(nombre : String, fuerza : float) +getFuerza() : float	

¿Son ambas clases necesarias?

NO!!!

Ejemplo: Héroes y Villanos

Empecemos a modelar...

“Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. [...] Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje [...].”

Personaje
-nombre : String -fuerza : float
+Personaje(nombre : String, fuerza : float) +getFuerza() : float

Ambas clases pueden ser reemplazadas por la clase Personaje.

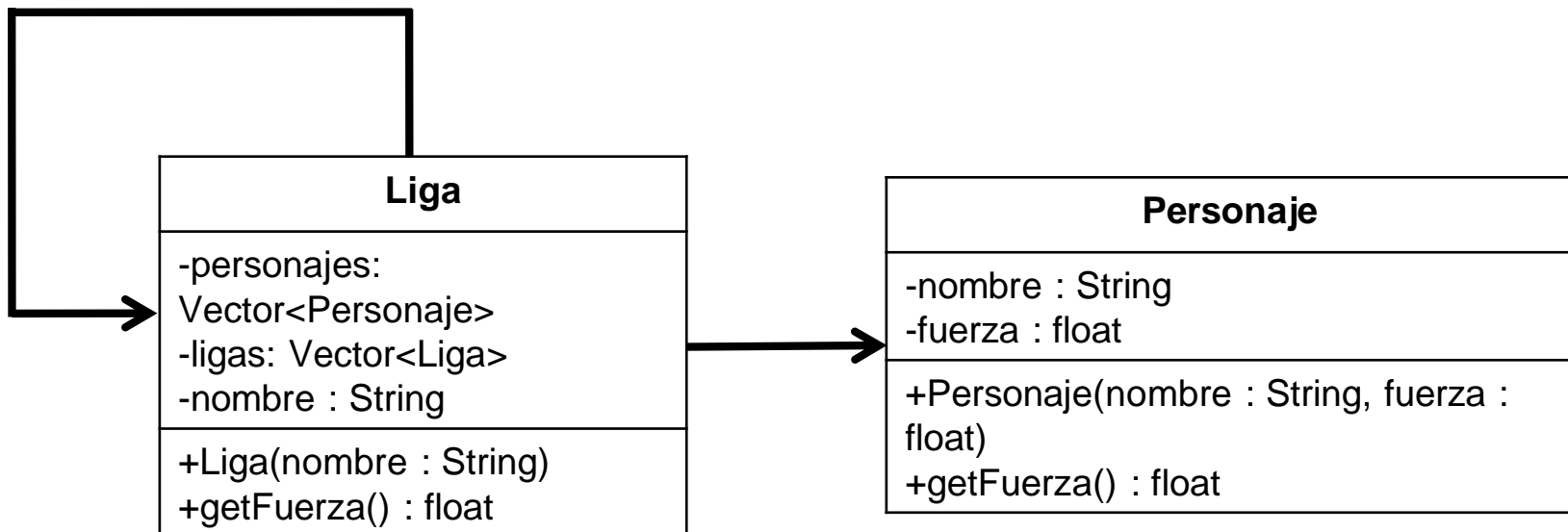


Ejemplo: Héroes y Villanos

“Adicionalmente, el juego debe proveer un mecanismo de agrupamiento de los personajes en ligas para realizar enfrentamientos entre grupos de personajes. Cada liga puede estar compuesta tanto de personajes como de otras ligas. Cada liga tiene su propio nombre de liga y un valor de fuerza.”

Ejemplo: Héroes y Villanos

“Adicionalmente, el juego debe proveer un mecanismo de agrupamiento de los personajes en ligas para realizar enfrentamientos entre grupos de personajes. Cada liga puede estar compuesta tanto de personajes como de otras ligas. Cada liga tiene su propio nombre de liga y un valor de fuerza.”



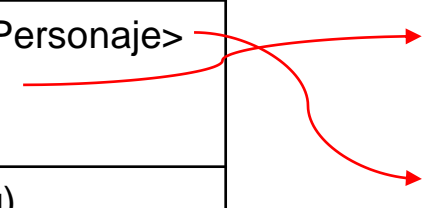
Ejemplo: Héroes y Villanos

“Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje como de una liga.”

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

Liga
-personajes: Vector<Personaje> -ligas: Vector<Liga> -nombre : String
+Liga(nombre : String) +getFuerza() : float

```
public float getFuerza(){  
    float sum = 0;  
  
    for(Liga l : ligas)  
        sum += l.getFuerza();  
  
    for(Personaje p : personajes)  
        sum += p.getFuerza();  
  
    return sum/(float)(ligas.size()+personajes.size());  
}
```



Debido a la existencia de dos atributos específicos para las ligas y los personajes surge la necesidad de diferenciar en el código entre la obtención de la fuerza tanto por parte de los personajes como de las ligas.

¿Es posible mejorar el diseño?

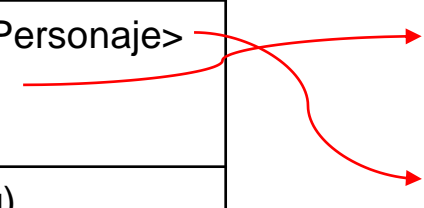
Ejemplo: Héroes y Villanos

“Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje como de una liga.”

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

Liga
-personajes: Vector<Personaje> -ligas: Vector<Liga> -nombre : String
+Liga(nombre : String) +getFuerza() : float

```
public float getFuerza(){  
    float sum = 0;  
  
    for(Liga l : ligas)  
        sum += l.getFuerza();  
  
    for(Personaje p : personajes)  
        sum += p.getFuerza();  
  
    return sum/(float)(ligas.size()+personajes.size());  
}
```



Debido a la existencia de dos atributos específicos para las ligas y los personajes surge la necesidad de diferenciar en el código entre la obtención de la fuerza tanto por parte de los personajes como de las ligas.

¿Es posible mejorar el diseño? SI !!

Ejemplo: Héroes y Villanos

En lugar de hacer que Liga distinga entre ligas y personajes, se puede crear una super-clase (“Enfrentable”) de la que hereden tanto Liga como Personaje. El atributo común pasa al padre de la jerarquía, quien también define la interfaz de los métodos comunes.

<i>Enfrentable</i>
-nombre : String
+Enfrentable(nombre : String) +getFuerza() : float

Ejemplo: Héroes y Villanos

En lugar de hacer que Liga distinga entre ligas y personajes, se puede crear una super-clase (“Enfrentable”) de la que hereden tanto Liga como Personaje. El atributo común pasa al padre de la jerarquía, quien también define la interfaz de los métodos comunes.

<i>Enfrentable</i>
-nombre : String
+Enfrentable(nombre : String) +getFuerza() : float

¿Cómo se conforma la clase Liga?

De Liga y Personaje o se puede conformar de Enfrentable?

Ejemplo: Héroes y Villanos

En lugar de hacer que Liga distinga entre ligas y personajes, se puede crear una super-clase (“Enfrentable”) de la que hereden tanto Liga como Personaje. El atributo común pasa al padre de la jerarquía, quien también define la interfaz de los métodos comunes.

<i>Enfrentable</i>
-nombre : String
+Enfrentable(nombre : String) +getFuerza() : float

¿Cómo se conforma la clase Liga?

De Liga y Personaje o se puede conformar de Enfrentable?

Con este nuevo diseño, la clase Liga puede conformarse con objetos de tipo Enfrentable.

Liga
-enfrentables : Vector<Enfrentable>
+Liga(nombre : String) +getFuerza() : float +addEnfrentable(String : enfrentable) : void

Ejemplo: Héroes y Villanos

En lugar de hacer que Liga distinga entre ligas y personajes, se puede crear una super-clase (“Enfrentable”) de la que hereden tanto Liga como Personaje. El atributo común pasa al padre de la jerarquía, quien también define la interfaz de los métodos comunes.

<i>Enfrentable</i>
-nombre : String
+Enfrentable(nombre : String) +getFuerza() : float

¿Cómo se conforma la clase Liga?

De Liga y Personaje o se puede conformar de Enfrentable?

Con este nuevo diseño, la clase Liga puede conformarse con objetos de tipo Enfrentable.

Y Enfrentable, ¿Qué puede ser?

Liga
-enfrentables : Vector<Enfrentable>
+Liga(nombre : String) +getFuerza() : float +addEnfrentable(String : enfrentable) : void

Ejemplo: Héroes y Villanos

En lugar de hacer que Liga distinga entre ligas y personajes, se puede crear una super-clase (“Enfrentable”) de la que hereden tanto Liga como Personaje. El atributo común pasa al padre de la jerarquía, quien también define la interfaz de los métodos comunes.

<i>Enfrentable</i>
-nombre : String
+Enfrentable(nombre : String) +getFuerza() : float

¿Cómo se conforma la clase Liga?

De Liga y Personaje o se puede conformar de Enfrentable?

Con este nuevo diseño, la clase Liga puede conformarse con objetos de tipo Enfrentable.

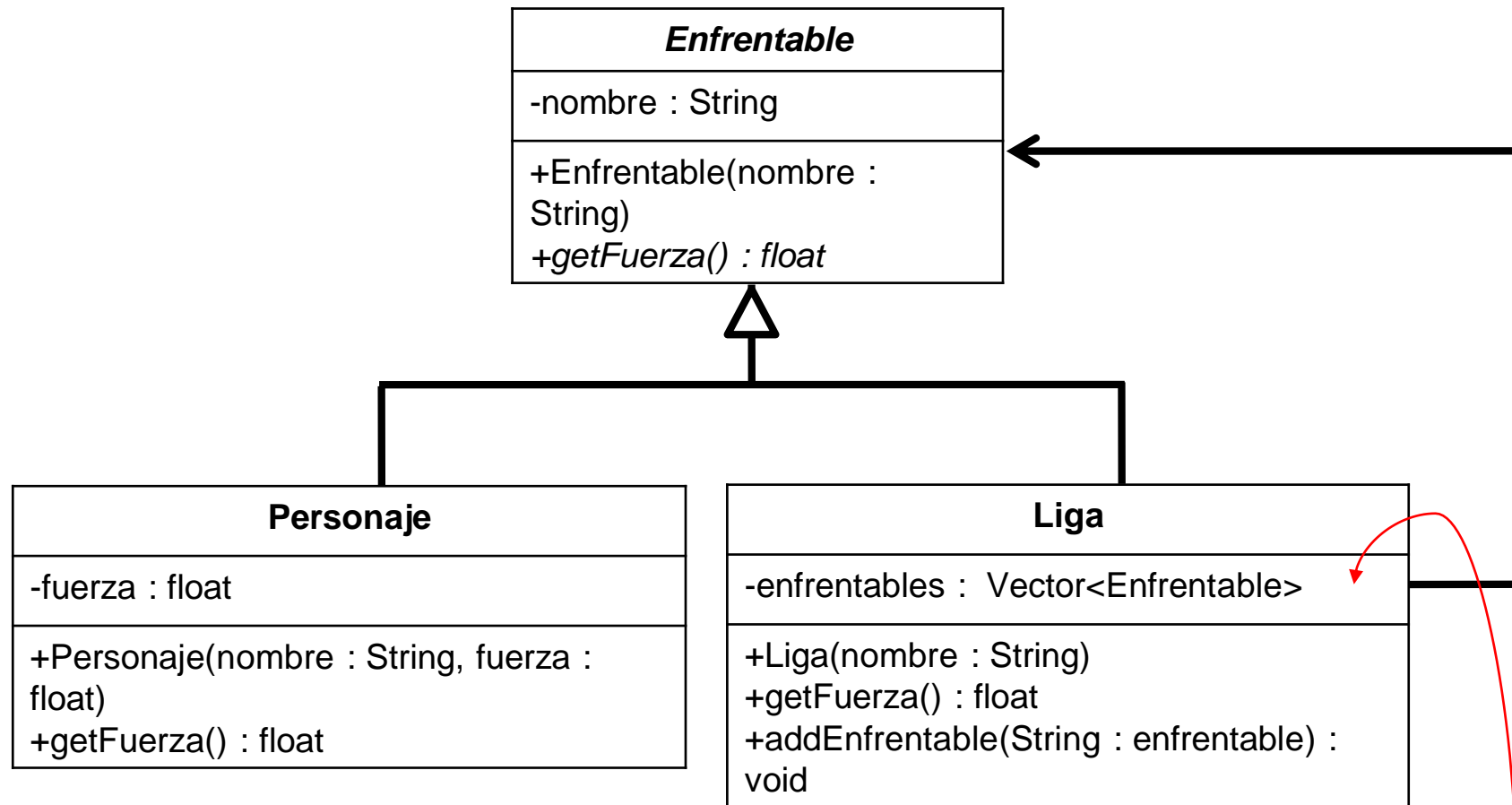
Y Enfrentable, ¿Qué puede ser?

Puede ser tanto una Liga como un Personaje. Recuérdese que declarando la variable de tipo padre, puede instanciarse dicha variable en cualquiera de los tipos (concretos) de sus hijos.

Liga
-enfrentables : Vector<Enfrentable>
+Liga(nombre : String) +getFuerza() : float +addEnfrentable(String : enfrentable) : void

Ejemplo: Héroes y Villanos

¿Cómo queda el diseño?



Ahora se tiene un único atributo de tipo *Enfrentable* y un único método que permite agregar los enfrentables.



Ejemplo: Héroes y Villanos

¿Qué mecanismo es el que permite que una variable pueda cambiar su tipo instanciado en tiempo de ejecución?

- **Polimorfismo.**
 - Un único nombre puede denotar objetos de distintas clases que se encuentran relacionadas por una super-clase común.
 - Cualquier objeto denotado por este nombre es capaz de responder a un conjunto común de operaciones.

Ejemplo: Héroes y Villanos

¿Qué mecanismo es el que permite que una variable pueda cambiar su tipo instanciado en tiempo de ejecución?

- **Polimorfismo.**
 - Un único nombre puede denotar objetos de distintas clases que se encuentran relacionadas por una super-clase común.
 - Cualquier objeto denotado por este nombre es capaz de responder a un conjunto común de operaciones.

En este caso, al declarar una variable de tipo Enfrentable, se estaría creando una referencia polimórfica, es decir, que puede referenciar a instancias de más de una clase.

```
Enfrentable ligaJusticia = new Liga("Liga de la Justicia");
```

```
Enfrentable batman = new Personaje("Batman",10);
```



Ejemplo: Héroes y Villanos

¿Qué pasa con la implementación del método `getFuerza()`? ¿Es la misma para ambas clases? ¿Puede dicha implementación encontrarse en el padre de la jerarquía? De acuerdo al enunciado:

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

Ejemplo: Héroes y Villanos

¿Qué pasa con la implementación del método `getFuerza()`? ¿Es la misma para ambas clases? ¿Puede dicha implementación encontrarse en el padre de la jerarquía? De acuerdo al enunciado:

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

```
public float getFuerza(){  
  
    if(this instanceof Personaje)  
        return this.fuerza;  
    else  
        if(this instanceof Liga){  
            float sum = 0;  
            for(Enfrentable e : enfrentables)  
                sum += e.getFuerza();  
            return sum/(float)enfrentables.size();  
        }  
    return 0;  
}
```


Ejemplo: Héroes y Villanos

¿Qué pasa con la implementación del método `getFuerza()`? ¿Es la misma para ambas clases? ¿Puede dicha implementación encontrarse en el padre de la jerarquía? De acuerdo al enunciado:

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

```
public float getFuerza(){  
  
    if(this instanceof Personaje)  
        return this.fuerza;  
    else  
        if(this instanceof Liga){  
            float sum = 0;  
            for(Enfrentable e : enfrentables)  
                sum += e.getFuerza();  
            return sum/(float)enfrentables.size();  
        }  
    return 0;  
}
```

¿Pero sería correcto tener este código?

Ejemplo: Héroes y Villanos

¿Qué pasa con la implementación del método `getFuerza()`? ¿Es la misma para ambas clases? ¿Puede dicha implementación encontrarse en el padre de la jerarquía? De acuerdo al enunciado:

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

```
public float getFuerza(){  
  
    if(this instanceof Personaje)  
        return this.fuerza;  
    else  
        if(this instanceof Liga){  
            float sum = 0;  
            for(Enfrentable e : enfrentables)  
                sum += e.getFuerza();  
            return sum/(float)enfrentables.size();  
        }  
    return 0;  
}
```

¿Pero sería correcto tener este código?

No! Se estarían desaprovechando todas las ventajas antes mencionadas tanto del polimorfismo como del binding dinámico. Por otra parte, se estaría reduciendo tanto la extensibilidad como la modificabilidad del diseño.

Ejemplo: Héroes y Villanos

¿Qué pasa con la implementación del método `getFuerza()`? ¿Es la misma para ambas clases? ¿Puede dicha implementación encontrarse en el padre de la jerarquía? De acuerdo al enunciado:

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

```
public float getFuerza(){  
  
    if(this instanceof Personaje)  
        return this.fuerza;  
    else  
        if(this instanceof Liga){  
            float sum = 0;  
            for(Enfrentable e : enfrentables)  
                sum += e.getFuerza();  
            return sum/(float)enfrentables.size();  
        }  
    return 0;  
}
```

¿Pero sería correcto tener este código?

No! Se estarían desaprovechando todas las ventajas antes mencionadas tanto del polimorfismo como del binding dinámico. Por otra parte, se estaría reduciendo tanto la extensibilidad como la modificabilidad del diseño.

¿Es posible mejorar el diseño?

Ejemplo: Héroes y Villanos

¿Qué pasa con la implementación del método `getFuerza()`? ¿Es la misma para ambas clases? ¿Puede dicha implementación encontrarse en el padre de la jerarquía? De acuerdo al enunciado:

“La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman.”

```
public float getFuerza(){  
  
    if(this instanceof Personaje)  
        return this.fuerza;  
    else  
        if(this instanceof Liga){  
            float sum = 0;  
            for(Enfrentable e : enfrentables)  
                sum += e.getFuerza();  
            return sum/(float)enfrentables.size();  
        }  
    return 0;  
}
```

¿Pero sería correcto tener este código?

No! Se estarían desaprovechando todas las ventajas antes mencionadas tanto del polimorfismo como del binding dinámico. Por otra parte, se estaría reduciendo tanto la extensibilidad como la modificabilidad del diseño.

¿Es posible mejorar el diseño?

Si!



Ministerio de
Educación y Deportes
Presidencia de la Nación



Ministerio de Producción
Presidencia de la Nación

Ejemplo: Héroes y Villanos

Sin saber en qué tipo está instanciado el objeto se puede invocar el método `getFuerza()`
¿Cuál es el mecanismo que permite que la implementación del método cambie en tiempo de ejecución?



Ejemplo: Héroes y Villanos

Sin saber en qué tipo está instanciado el objeto se puede invocar el método `getFuerza()`
¿Cuál es el mecanismo que permite que la implementación del método cambie en tiempo de ejecución?

Binding dinámico! El binding dinámico es el mecanismo que permite que el código de los métodos sea asociado a una instancia recién en el momento en el cuál esta recibe el mensaje para ejecutar dicho método.



Ejemplo: Héroes y Villanos

Sin saber en qué tipo está instanciado el objeto se puede invocar el método `getFuerza()`
¿Cuál es el mecanismo que permite que la implementación del método cambie en tiempo de ejecución?

Binding dinámico! El binding dinámico es el mecanismo que permite que el código de los métodos sea asociado a una instancia recién en el momento en el cuál esta recibe el mensaje para ejecutar dicho método.

¿Cómo quedaría?

Ejemplo: Héroes y Villanos

Sin saber en qué tipo está instanciado el objeto se puede invocar el método `getFuerza()`
¿Cuál es el mecanismo que permite que la implementación del método cambie en tiempo de ejecución?

Binding dinámico! El binding dinámico es el mecanismo que permite que el código de los métodos sea asociado a una instancia recién en el momento en el cuál esta recibe el mensaje para ejecutar dicho método.

¿Cómo quedaría?

Enfrentable

```
public abstract float getFuerza();
```

Personaje

```
@Override  
public float getFuerza(){  
    return this.fuerza();  
}
```

Liga

```
@Override  
public float getFuerza(){  
    float sum = 0;  
    for(Enfrentable e : enfrentables)  
        sum += e.getFuerza();  
    return  
    sum/(float)enfrentables.size();  
}
```


Ejemplo: Héroes y Villanos

Supóngase que ligaJusticia se encuentra compuesta de la siguiente forma:

```
Enfrentable dosFantásticos = new Liga("Los Dos Fantásticos");

Enfrentable mole = new Personaje("Mole",4);

Enfrentable hombreInvisible = new Personaje("El Hombre Invisible",3);

dosFantásticos.addEnfrentable(mole);
dosFantásticos.addEnfrentable(hombreInvisible);

Enfrentable ligaJusticia = new Liga("Liga de la Justicia");

Enfrentable batman = new Personaje("Batman",10);

ligaJusticia.addEnfrentable(dosFantásticos);
ligaJusticia.addEnfrentable(batman);
```

ligaJusticia

dosFantásticos

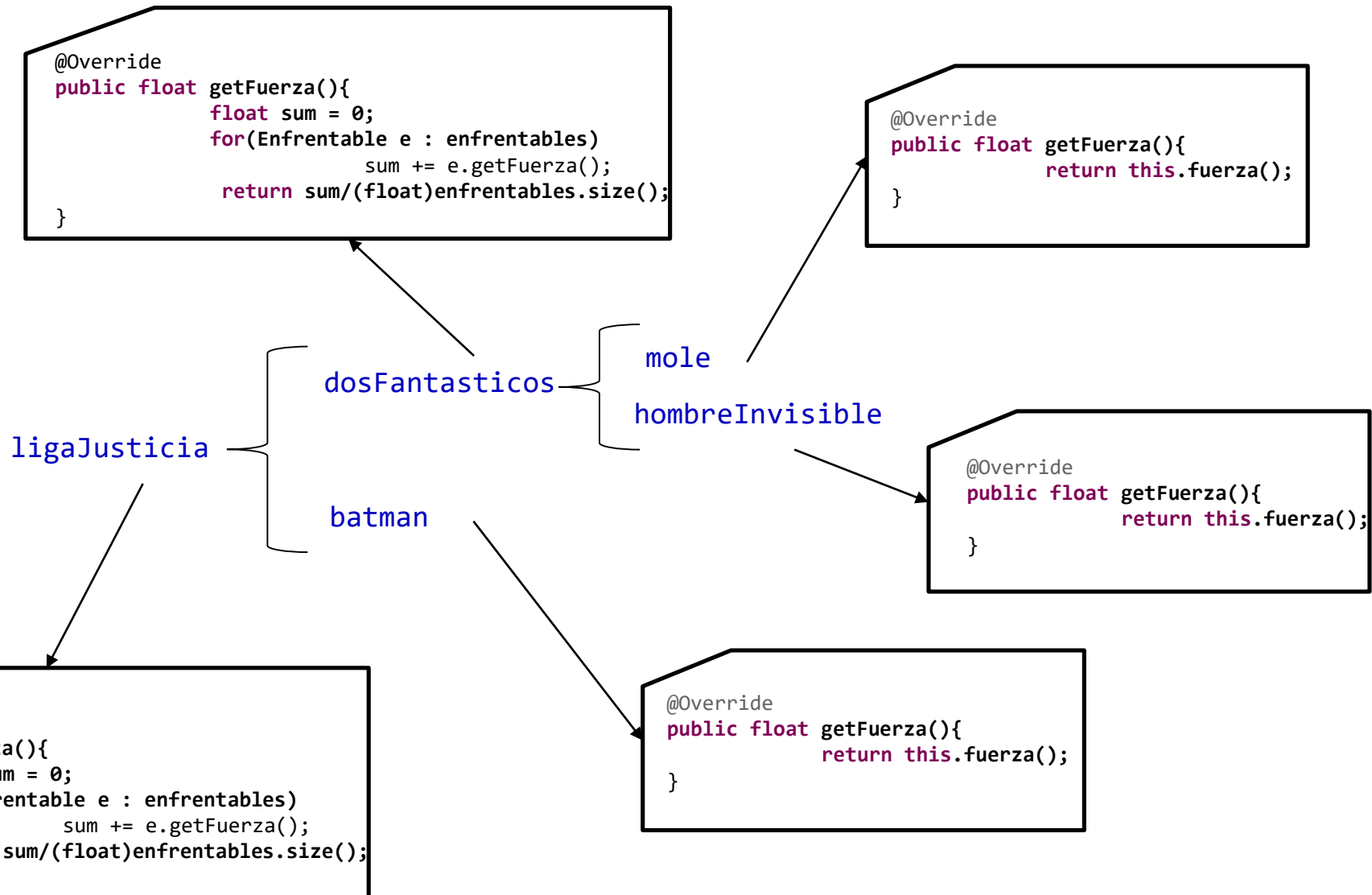
batman

mole

hombreInvisible

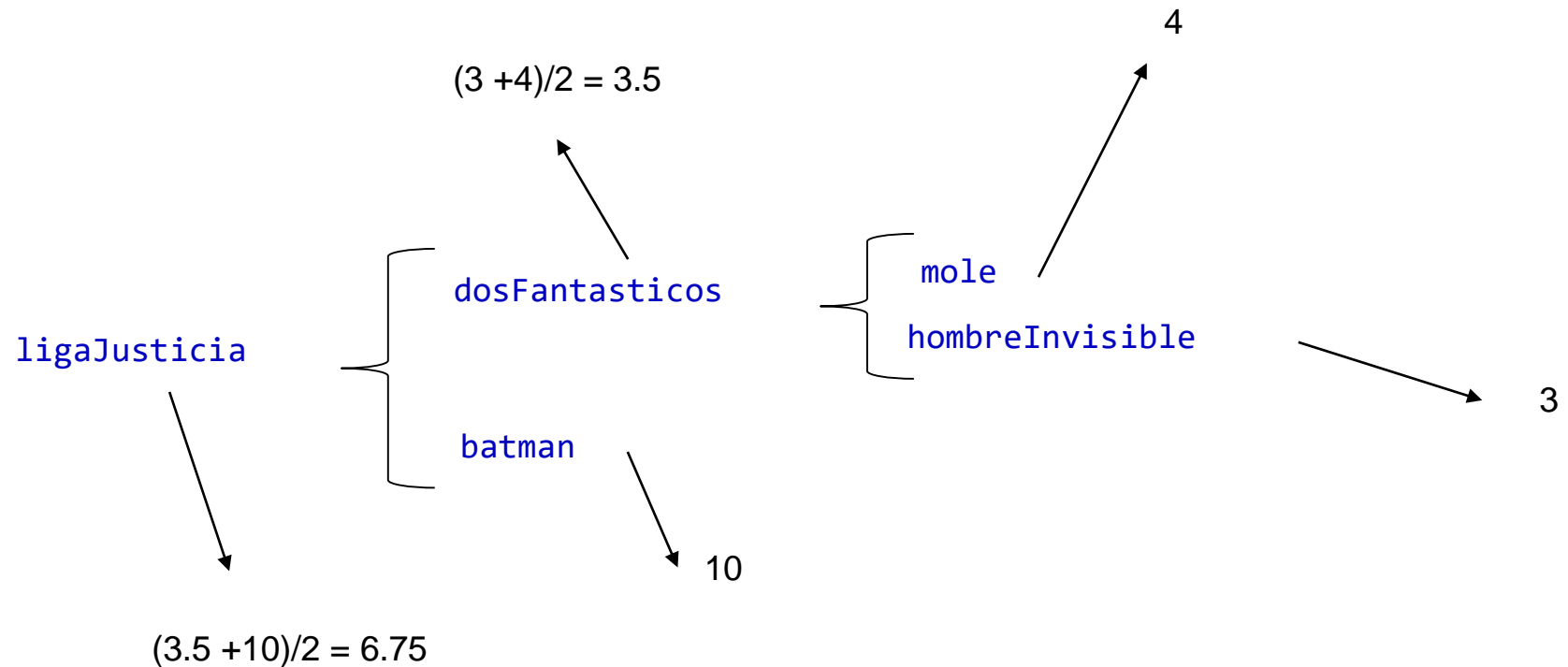
¿Cómo se haría el cálculo de la fuerza?

Ejemplo: Héroes y Villanos



Ejemplo: Héroes y Villanos

Cuando los llamados se resuelven...





Referencias

- **Introducción a Java (polimorfismo)**
 - <http://www.programacion.com/java/tutorial/intjava/9/>
- **Pensando en Java por Bruce Eckel, 3ra Edición**
 - Capítulo 7, Polimorfismo
- **Fundamentos del Lenguaje Java**
 - <http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/herencia/herencia1.htm>
- **Definición de polimorfismo de Wikipedia**
 - [http://es.wikipedia.org/wiki/Polimorfismo_\(programaci%C3%B3n_orientada_a_objetos\)](http://es.wikipedia.org/wiki/Polimorfismo_(programaci%C3%B3n_orientada_a_objetos))



Ministerio de Producción
Presidencia de la Nación

Ministerio de Educación y Deportes

Subsecretaría de Servicios Tecnológicos y Productivos



**PROGRAMACIÓN ORIENTADA
A OBJETOS**