

Jerarquías de clases

Clases: subclases y superclases

Jerarquía de clases: herencia, polimorfismo

Sobrescritura de métodos heredados

Ocultación de atributos heredados

Clases abstractas y finales

Control de acceso a clases y sus componentes

Paquetes

Definición básica de subclases

```
public class Persona {  
    public String nombre;  
    public int edad;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}
```

*Usamos atributos **public** solamente por facilitar los ejemplos que vienen a continuación.*

```
public class Empleado extends Persona {  
    public long sueldoBruto;  
    public Directivo jefe;  
    ...  
}
```

*Empleado es una **subclase** de Persona;
Persona es la **superclase** de Empleado*

```
public class Directivo extends Empleado {  
    public long incentivo;  
    public ArrayList<Empleado> equipo;  
    public void fijarIncentivo(long c) { incentivo = c; }  
}
```

Clase raíz de la jerarquía

¿Cuál es la clase raíz de la jerarquía?

La clase predefinida **Object**

La anterior definición de clase **Persona** (sin superclase explícita) es equivalente a la siguiente subclase de la clase raíz **Object**

```
public class Persona extends Object {  
    public String nombre;  
    public int edad;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}
```

¿Qué contiene Object?

- protected Object clone() throws CloneNotSupportedException
Crea y devuelve una copia del objeto
- public boolean equals(Object obj)
Indica si obj es "igual" al objeto this
- protected void finalize() throws Throwable
Llamado por el recolector de basura sobre un objeto cuando determina que no hay más referencias al objeto
- public final Class getClass()
Devuelve la clase en tiempo de ejecución
- public int hashCode()
Devuelve un Código hash para el object
- public String toString()
Devuelve una representación del objeto como string
- Methods to synchronize threads (notify, ...)

*Los analizaremos
más adelante*

Jerarquía de tipos: relación “es_un”, conversión

```
Persona p1, p2 = new Persona();  
Empleado e, emp = new Empleado();  
Directivo d, dir = new Directivo();
```

■ Conversión automática implícita (generalización)

```
p1 = emp; // Empleado → Persona  
p2 = dir; // Directivo → Persona  
e = dir; // Directivo → Empleado
```

*Un Directivo puede
hacer automáticamente
el papel de Empleado
y de Persona*

■ Conversión explícita (especialización), responsabilidad del programador

```
e = p2; // Error de compilación  
d = (Directivo) p2; // Persona → Directivo  
d = (Directivo) p1; // Error de ejecución:  
// p1 no es un Directivo  
d = (Perro) p1; // Error de compilación (Perro no es  
// subclase ni superclase de Persona)
```

*Una Persona puede
hacer el papel de
Directivo si realmente
es un Directivo*

Jerarquía de tipos: compatibilidad y conversión de argumentos

```
class Corporacion {  
    void f(Empleado empleado) { ... }  
    void g(Directivo directivo) { ... }  
}
```

```
Directivo dir      = new Directivo();  
Empleado e        = dir,  
                  emp      = new Empleado();  
Corporacion c      = new Corporacion();
```

- Conversión implícita (*hacia mayor “generalidad”*)

```
c.f(dir); // Directivo → Empleado
```

- Conversión explícita

```
c.g(e); // Error de compilación  
c.g((Directivo) e); // Empleado → Directivo  
c.g((Directivo) emp); // Error de ejecución: emp no es  
                      // un Directivo
```

Herencia de métodos y atributos

Se heredan atributos y métodos de la superclase (no sólo los propios de la superclase sino también los heredados por ella).

```
Empleado emp = new Empleado();  
Directivo dir = new Directivo();
```

*Aquí aprovechamos que antes
declaramos atributos **public**.*

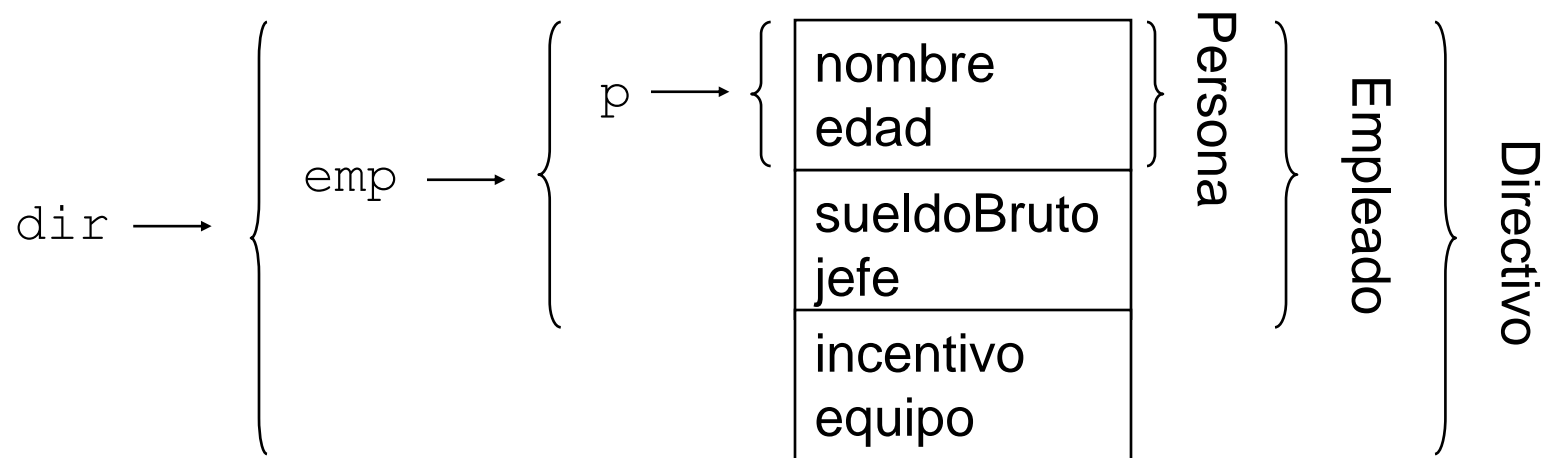
```
emp.nombre = "Pedro";      // campo de emp como Persona  
emp.edad = 28;  
emp.sueldoBruto = 2000;    // campo de emp como Empleado  
emp.jefe = dir;  
System.out.println(emp);  // llamada implícita a emp.toString()  
                           // método de Persona  
  
dir.nombre = "Maria";  
dir.edad = 45;  
dir.sueldoBruto = 5000;  
dir.jefe = null;  
dir.incentivo = 1500;  
System.out.println(dir.toString());
```

Herencia y jerarquía de tipos

```
Directivo dir = new Directivo();  
Empleado emp = dir;  
Persona p = dir;
```

Tipo estático vs.
tipo dinámico

```
p.sueldoBruto = 1000; // Error: sueldoBruto no  
                      // definido para Persona  
emp.fijarIncentivo(0); // Error: fijarIncentivo no definido  
                      // para Empleado
```



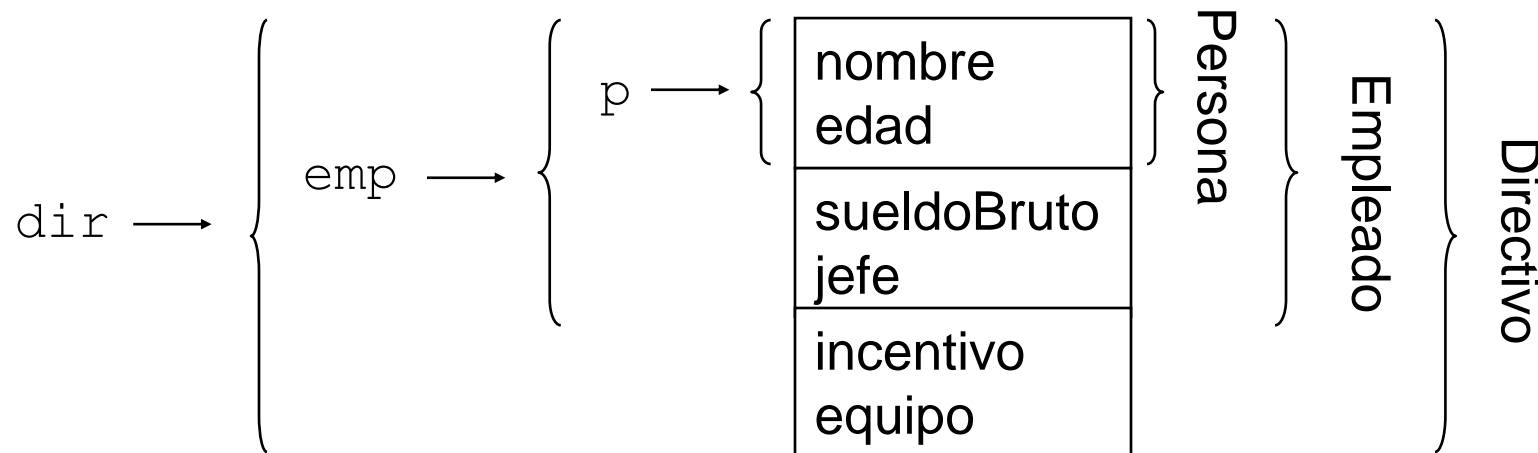
Herencia y jerarquía de tipos

```
Directivo dir = new Directivo();  
Empleado emp = dir;  
Persona p = dir;
```

Tipo estático vs.
tipo dinámico

```
((Empleado)p).sueldoBruto = 1000; // OK  
((Directivo)p).sueldoBruto = 1000; // OK  
((Directivo)emp).fijarIncentivo(0); // OK
```

Los castings han de evitarse siempre que sea posible



La herencia se produce también con `private`

```
public class Persona {  
    private String nombre;  
    private int edad;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}  
  
public class Empleado extends Persona {  
    private long sueldoBruto;  
    private Directivo jefe;  
    ... // metodos propios de Empleado... necesarios ahora ?!  
}  
  
public class Directivo extends Empleado {  
    private long incentivo;  
    private ArrayList<Empleado> equipo;  
    public void fijarIncentivo(long c) { incentivo = c; }  
}
```

Protected: Para acceder a los atributos desde la clase hija.

Sobrescritura (*overriding*) de métodos

- Sobrescritura de métodos heredados en la subclase
- La definición de la subclase toma precedencia sobre la de su superclase
- Superclase es accesible desde subclase con variable **super**
- Sobrescritura de métodos (especializarlos para la subclase)
 - Se **sobrescribe** con idénticos tipos de argumentos y retorno covariante,
 - Retorno covariante: el método que sobrescribe, devuelve **el mismo tipo**, o un **subtipo**.
 - Si no coinciden los tipos de los argumentos, se trata de una *sobrecarga* (p.ej.: “A” + “2” vs. 3 + 2)
 - No se puede aumentar la privacidad al sobrescribirlo (pero podemos ir de protected a public, o de package a protected o public)
- Facilita la extensión y la reutilización, aprovechando la ligadura dinámica
- Se reduce la proliferación de identificadores de métodos

Sobrescritura (*overriding*) de métodos

```
public class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad +  
            "\nSueldo: " + sueldoBruto + "\nJefe: " +  
            ((jefe == null)? nombre : jefe.nombre);  
    }  
}  
  
// Bloque main  
Empleado emp = new Empleado();  
Persona p = emp;  
emp.toString(); // toString de Empleado  
p.toString();   // toString de Empleado (ligadura dinamica)
```

Nota. `toString()` de `Persona` es una *nueva implementación* del método heredado de la clase raíz de la jerarquía de herencia `Object`

Sobrescritura (*overriding*) de métodos

```
public class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    public String toString() { // Mejor sin repetir código  
        return super.toString() + // de toString de Persona  
            "\nSueldo: " + sueldoBruto + "\nJefe: " +  
            ((jefe == null)? nombre : jefe.nombre);  
    }  
}  
  
// Bloque main  
Empleado emp = new Empleado();  
Persona p = emp;  
emp.toString(); // toString de Empleado  
p.toString();   // toString de Empleado (ligadura dinamica)
```

Nota. `toString()` de `Persona` es una *nueva implementación* del método heredado de la clase raíz de la jerarquía de herencia `Object`

Sobrescritura (*overriding*) de métodos

```
public class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    @Override  
    public String toString() {  
        return super.toString() +  
            "\nSueldo: " + sueldoBruto + "\nJefe: " +  
            ((jefe == null)? nombre : jefe.nombre);  
    }  
}
```

Los métodos que sobrescriben a otros se pueden anotar explícitamente con **@Override**.

Ayuda a detectar fallos en la declaración de un método sobrescrito.

Sobrescritura (*overriding*) de métodos

```
public class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    long sueldoLiquido() {  
        long resultado;  
        resultado =... //calculos complejos de sueldo liquido  
        return resultado;  
    }  
}
```

Es mejor hacerlo usando el método de la superclase

```
public class Directivo extends Empleado {  
    long incentivo;  
    ...  
    long sueldoLiquido() {  
        return super.sueldoLiquido() + incentivo;  
    }  
}
```

Sobrescritura (*overriding*) de métodos:

Retorno Co-variante

```
class C {  
    public void print() { System.out.println("C"); }  
}  
class D extends C {  
    public void print() { System.out.println("D"); }  
}
```

```
class A {  
    public C f() { return new C(); }  
}  
class B extends A {  
    @Override public D f() { return new D(); }  
}
```

Retorno co-variante



```
public class Test1 {  
    public static void main(String[] args) {  
        new B().f().print();    // escribe D  
        new A().f().print();    // escribe C  
        A a = new B();  
        a.f().print();          // ¿Qué escribe?  
    }  
}
```


Retorno Co-variante

¿Por qué funciona?

```
public class Test1 {  
    public static void main(String[] args) {  
        A aes[] = {new B(), new A()};  
        // Podemos asignar de manera segura el retorno de f()...  
        // ... a una variable del tipo de retorno del método f()  
        // de la clase padre.  
        for (A a : aes) {  
            C c = a.f();    // Esta signación es segura  
            c.print();  
        } // imprime D C  
    }  
}
```

¿Qué problema habría en permitir también el retorno del modo opuesto (contra-variante)?

Ejercicio (de un examen parcial)

Necesitamos construir una aplicación para la gestión de un almacén de artículos de segunda mano.

El almacén gestiona productos, que tienen un nombre, un precio en euros, y un descuento, que puede ser un porcentaje configurable del precio total, o una cantidad fija de euros. Los descuentos tienen un texto que describe la promoción. Una vez creado un producto, su precio y su descuento no pueden modificarse.

Utilizando los principios de la orientación a objetos, codificar en Java las clases necesarias para que el siguiente programa produzca lo que se muestra a continuación.

```
package products;
```

```
public class Warehouse {  
    public static void main(String[] args) {  
        // p1 is a Product with price 150.0€, and 15.0% discount due to promotion "no VAT"  
        Product p1 = new Product("Floor Lamp", 150.0,  
                                   new PercentageDiscount("No VAT", 15.0));  
        // p2 is a Product with price 90.0€, and 10.0€ discount by "clearance"  
        Product p2 = new Product("Cutlery 50 items", 90.0,  
                                   new FixedDiscount("Clearance", 10.0));  
        System.out.println("Products in warehouse:\n "+p1+"\n "+p2);  
        System.out.println("Higher price: "+Product.higherPrice());  
    }  
}
```

Output:

Products in warehouse:

Floor lamp price: 127.5 with discount: No VAT

Cutlery 50 items price: 80.0 with discount: Clearance

Higher price: 127.5

Intento erróneo de sobrescritura

```
public class Point {  
    private int x = 0, y = 0, color;  
  
    int getX() { return x; }  
    int getY() { return y; }  
}
```

```
class RealPoint extends Point {  
    double dx = 0.0, dy = 0.0;
```

Error de compilación:

El retorno covariante no aplica a tipos primitivos (en cualquier caso además aquí no tendríamos retorno covariante).

¿Error de diseño?

```
double getX() { return dx; }  
double getY() { return dy; }  
}
```


Corrección de la sobrescritura

```
public class Point {  
    private int x = 0, y = 0, color;
```

```
    int getX() { return x; }  
    int getY() { return y; }  
}
```

```
class RealPoint extends Point {  
    double dx = 0.0, dy = 0.0;
```

```
    int getX() { return (int) Math.floor(dx); }  
    int getY() { return (int) Math.floor(dy); }  
}
```



El error de compilación se puede evitar así: la sobrescritura de métodos ahora es correcta, pero...

¿Persiste el error de diseño?



Esto no es sobrescritura sino sobrecarga

```
public class Point {  
    private int x = 0, y = 0, color;  
  
    void move(int mx, int my) { x += mx; y += my; }  
  
}
```

```
class RealPoint extends Point {  
    double dx = 0.0, dy = 0.0;
```

El método move(int,int) se hereda y coexiste con el nuevo método move(double,double) en RealPoint

No hay sobrescritura sino **sobrecarga** de métodos en la clase RealPoint

¿Sentido de move(int,int) en RealPoint?

```
    void move(double mx, double my) { dx += mx; dy += my; }  
  
}
```

Sobrescritura y sobrecarga pueden coexistir

```
public class Point {  
    private int x = 0, y = 0, color;
```

```
    void move(int mx, int my) { x += mx; y += my; }
```

```
}
```

```
class RealPoint extends Point {
```

```
    double dx = 0.0, dy = 0.0;
```

```
    void move(int mx, int my) {  
        move((double)mx, (double)my);
```

```
    }
```

```
    void move(double mx, double my) { dx += mx; dy += my; }
```

```
}
```

El método `move(int,int)` se sobrescribe y se añade el método `move(double,double)` con sobrecarga en `RealPoint`

El ejemplo completo: ¿bien diseñado?

```
public class Point {  
    private int x = 0, y = 0, color;  
  
    public void move(int mx, int my) { x += mx; y += my; }  
    public int getX() { return x; }  
    public int getY() { return y; }  
}
```

¿De qué sirve que RealPoint herede las variables x e y de tipo int, si se añaden dx y dy de tipo double?

```
class RealPoint extends Point {  
    double dx = 0.0, dy = 0.0;  
    public void move(int mx, int my) {  
        move((double)mx, (double)my);  
    }  
    public void move(double mx, double my) { dx += mx; dy += my; }  
  
    public int getX() { return (int)Math.floor(dx); }  
    public int getY() { return (int)Math.floor(dy); }  
}
```


Ejemplo con encubrimiento de atributos

```
public class Point {  
    private int x = 0, y = 0, color;  
  
    public void move(int mx, int my) { x += mx; y += my; }  
    public int getX() { return x; }  
    public int getY() { return y; }  
}
```

Se puede *encubrir* la herencia de las variables `x` e `y` de tipo `int`, añadiendo variables con el mismo nombre de tipo `double` en `RealPoint`

```
class RealPoint extends Point {  
    double x = 0.0, y = 0.0;  
    public void move(int mx, int my) {  
        move((double)mx, (double)my);  
    }  
    public void move(double mx, double my) { x += mx; y += my; }  
  
    public int getX() { return (int)Math.floor(x); }  
    public int getY() { return (int)Math.floor(y); }  
}
```

Ejemplo con encubrimiento de atributos

```
public class Point {  
    private int x = 0, y = 0, color;  
  
    public void move(int mx, int my) { x += mx; y += my; }  
    public int getX() { return x; }  
    public int getY() { return y; }  
}
```

Se puede *encubrir* la herencia de las variables `x` e `y` de tipo `int`, añadiendo variables con el mismo nombre de tipo `double` en `RealPoint`

```
class RealPoint extends Point {  
    double x = 0.0, y = 0.0;  
    public void move(int mx, int my) {  
        move((double)mx, (double)my);  
    }  
    public void move(double mx, double my) { x += mx; y += my; }  
  
    public int getX() { return (int)Math.floor(x); }  
    public int getY() { return (int)Math.floor(y); }  
}
```

// ¿faltaría añadir algún(os) constructor(es)?

Encubrimiento (*hiding*) de atributos

- Encubrimiento de atributos heredados
- La definición de la subclase oculta a la de la superclase
- Superclase accesible desde la subclase con **super**
- Encubrimiento de atributos
 - Los atributos de la superclase y de la subclase coexisten
 - El tipo de ambos atributos no tiene por qué coincidir
 - Se reserva un espacio de memoria para ambas definiciones
- En este caso se utiliza ligadura estática
- En general es preferible evitar el encubrimiento de variables heredadas (no resulta tan útil como la sobrescritura de métodos)

Encubrimiento (*hiding*) de atributos

```
public class Musico extends Persona {  
    String nombre; // oculta a la variable nombre de Persona  
    public void mostrarNombres() {  
        System.out.println("Musico:  " + nombre);  
        System.out.println("Persona: " + super.nombre);  
    }  
}
```

```
// Bloque main
```

```
Musico m = new Musico();
```

```
Persona p = m;
```

```
                // acceso con ligadura estática
```

```
m.nombre = "Stevie Wonder"; // nombre de Musico
```

```
p.nombre = "Stevland Morris"; // nombre de Persona
```

```
((Musico)p).nombre = "Stevie Wonder"; //nombre de Musico
```

```
((Persona)m).nombre = "Stevland Morris"; //nombre de Persona
```

Ejercicio: ¿Qué imprime este programa?

```
class AClass{
    public int a = 3;
}

class BClass extends AClass{
    public double a = 4.5;
}

public class Test2 {
    public static void main(String[] args) {
        BClass aa = new BClass();
        AClass ac = new AClass();
        AClass ab = new BClass();
        System.out.println(aa.a+" "+ac.a+" "+ab.a);
    }
}
```

Ejercicio: ¿Qué imprime este programa?

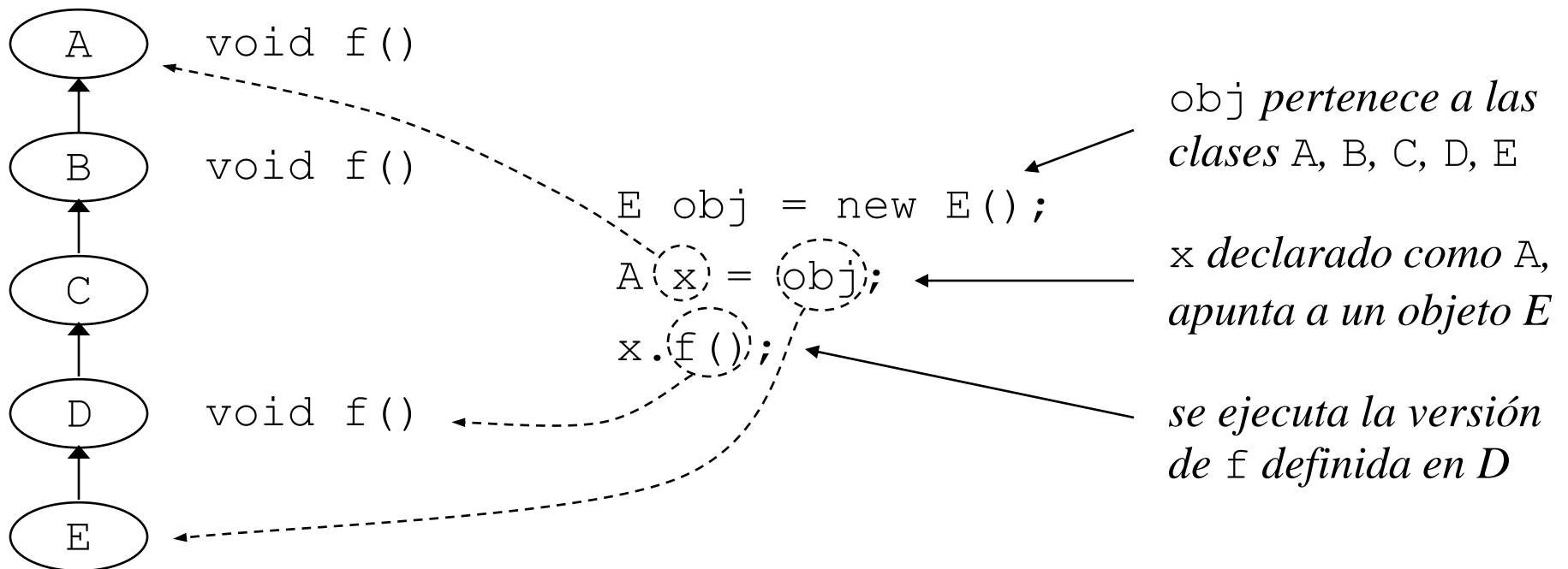
```
class AClass{
    public int a = 3;
}

class BClass extends AClass{
    public double a = 4.5;
}

public class Test2 {
    public static void main(String[] args) {
        BClass aa = new BClass();
        AClass ac = new AClass();
        AClass ab = new BClass();
        System.out.println(aa.a+" "+ac.a+" "+((BClass)ab).a);
    }
}
```

Ligadura dinámica

- La ejecución de métodos sobreescritos se resuelve por ligadura dinámica en tiempo de ejecución
- Se ejecuta la definición del método de la clase más específica del objeto, independientemente de cómo se ha declarado la referencia al objeto



- Los métodos estáticos (métodos de clase) tienen ligadura estática

Ligadura dinámica: ejemplo

```
public class Persona {  
    String nombre;  
    int edad;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}  
  
public class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    public String toString() {  
        return super.toString() +  
            "\nSueldo: " + sueldoBruto + "\nJefe: " +  
            ((jefe == null)? nombre : jefe.nombre);  
    }  
}
```


Ligadura dinámica: ejemplo (cont.)

```
public class Directivo extends Empleado {  
    long incentivo;  
    ArrayList<Empleado> equipo = new ArrayList<Empleado>();  
    public String toString() {  
        return super.toString() + // toString() de Empleado  
            "\nIncentivo: " + incentivo;  
    }  
    public void fijarIncentivo(long c) { incentivo = c; }  
}
```

Ligadura dinámica: ejemplo (cont.)

```
// Bloque main
Directivo dir = new Directivo();
Empleado emp = new Empleado();
Empleado e = dir;
Persona p = new Persona();
Persona x = emp;
Persona y = e;
String s;
s = p.toString();      // toString de Persona
s = emp.toString();    // toString de Empleado
s = dir.toString();    // toString de Directivo
s = x.toString();      // toString de Empleado
s = y.toString();      // toString de Directivo
s = e.toString();      // toString de Directivo
y.fijarIncentivo(1500); // ERROR
```

La ligadura de los argumentos es estática

No se trata de sobrescritura sino de sobrecarga de método

```
public class ClaseA {  
    public void f(Persona per) {  
        System.out.println("Clase Persona");  
    }  
    public void f(Empleado emp) {  
        System.out.println("Clase Empleado");  
    }  
}
```

*Se ejecuta la definición **compatible** más específica*

```
// Bloque main  
ClaseA a = new ClaseA();  
Directivo dir = new Directivo();  
Persona p = dir;  
a.f(dir);  
a.f(p); // (*)  
OtraSuperClaseDePersona x = p;  
a.f(x); // ERROR
```

La ligadura de los argumentos es estática ¿Por qué?

```
// Bloque main  
ClaseA a = new ClaseA();  
Directivo dir = new Directivo();  
Persona p = dir;
```

a.f(p); // (*)

// (*)
// metodo f (Persona per)
Persona per = p;
System.out.println("Clase Persona");

// metodo f (Empleado emp)
Empleado emp = p;
System.out.println("Clase Empleado");

*Incorrecto: haría
falta un casting*

*En el paso de parámetros hay una asignación de los
parámetros actuales a los formales*

Jerarquía de clases y constructores

Los constructores no se heredan ni se sobrescriben

Cada (sub/super)clase tiene su propio constructor

- Al crear un `Empleado`, hay que invocar al constructor de `Persona`
- Invocación automática implícita
 - Se invoca al constructor de la superclase sin argumentos
 - Si no está definido se produce error
- Invocación explícita
 - `super (...)` en la primera línea del constructor de `Empleado`
- Invocación a otros constructores de la misma clase: `this (...)`

Subclases y constructores: ejemplo

```
public class Persona {  
    String nombre;  
    int edad;  
    public Persona(String str, int i) {  
        nombre = str;  
        edad = i;  
    }  
    public String toString() { ... }  
}
```

Error al crear un Empleado:

el constructor por defecto **Empleado()** invoca al constructor por defecto **Persona()** que **ahora ya no está definido**

Antes sí lo estaba porque no había ningún constructor en **Persona**


Podemos añadir explícitamente

```
public Persona() { nombre = ""; edad = 0; }
```

O mejor, añadir un constructor nuevo a **Empleado**

Subclases y constructores: ejemplo cont.

```
class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    public Empleado(String str, int i, long sueldo,  
                      Directivo dir) {  
        nombre = str; }  
        edad = i; }  
        sueldoBruto = sueldo;  
        jefe = dir;  
    }  
    String toString() { ... }  
}
```



Error: Aunque asignemos valor a `nombre` y `edad`, se sigue invocando automáticamente a `Persona()` constructor que no está definido

Error al crear un directivo: el constructor por defecto `Directivo()` invoca al constructor por defecto `Empleado()` que ya no está definido

Subclases y constructores: ejemplo cont.

```
class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    public Empleado (String str, int i, long sueldo,  
                     Directivo dir) {  
        super (str, i);  
        sueldoBruto = sueldo;  
        jefe = dir;  
    }  
    String toString () {  
        ...  
    }  
}
```

*Siempre en la
primera línea*

Subclases y constructores: ejemplo cont.

```
class Directivo extends Empleado {
    long incentivo;
    ArrayList<Empleado> equipo
        = new ArrayList<Empleado>();
    public Directivo(String nombre, int edad,
        long sueldo, long incentivo) {
        this(nombre, edad, sueldo, null, incentivo);
    }
    public Directivo(String nombre, int edad,
        long sueldo, Directivo jefe,
        long incentivo) {
        super(nombre, edad, sueldo, jefe);
        this.incentivo = incentivo;
    }
    String toString() {
        ...
    }
    void fijarIncentivo(long c) { incentivo = c; }
}
```

*Siempre en la
primera línea*

Clases abstractas, métodos abstractos

- Clase abstracta
 - No completamente definida (no del todo concretada)
 - No permite que se creen objetos de ese tipo
 - `new Persona()` → Error si clase Persona es **abstract**
 - Útil para definir subclases, que heredan atributos y métodos de la clase abstracta
 - Puede contener métodos abstractos (y no abstractos)
- Métodos abstractos
 - Métodos sin código, se declaran pero no se definen
 - Deben sobreescribirse en cada subclase concreta
- Toda clase con un método abstracto ha de declararse abstracta
- Toda subclase que no implemente un método abstracto también

Clase abstracta y subclases: ejemplo

```
public abstract class Figura {  
    public abstract double perimetro();  
}  
  
class Circulo extends Figura {  
    Punto2D centro;  
    double radio;  
    public double perimetro() { return 2 * Math.PI * radio; }  
}  
  
class Triangulo extends Figura {  
    Punto2D a, b, c;  
    public double perimetro() {  
        return a.distancia(b) + b.distancia(c) + c.distancia(a);  
    }  
}
```

Aquí `perimetro()` es un método abstracto, sin `{ }` sin implementación, solo `;`

Estas subclases dan su propia implementación de `perimetro()` heredado y ya no es método abstracto

Clase y subclases abstractas: ejemplo

```
public abstract class Figura {  
    public abstract double perimetro();  
    public abstract void resaltar();  
}  
  
abstract class Circulo extends Figura {  
    Punto2D centro;  
    double radio;  
    public double perimetro() { return 2 * Math.PI * radio; }  
}  
  
abstract class Triangulo extends Figura {  
    Punto2D a, b, c;  
    public double perimetro() {  
        return a.distancia(b) + b.distancia(c) + c.distancia(a);  
    }  
}
```

Aquí perimetro() y resaltar() son métodos abstractos

Se da implementación de perimetro() pero no de resaltar(). Son clases abstractas

Clases abstractas: ejemplo ampliado

```
public abstract class Figura {  
    public abstract double perimetro();  
    public abstract void resaltar();  
    // NO sería erróneo public abstract String toString();  
}
```

```
abstract class FiguraColor extends Figura { //Error sin abstract  
    Color colorLinea, colorFondo;  
}
```

Esta subclase debe ser abstracta: aunque no declara métodos abstractos, los hereda y los deja sin implementar

```
class Circulo extends Figura {  
    Punto2D centro;  
    double radio;  
    public double perimetro() { return 2 * Math.PI * radio; }  
    public void resaltar() { return; } // o bien { } no hacer nada  
    public String toString() {  
        return "CIRC: centro en" + centro + " y radio" + radio;  
    }  
}
```

Aunque la implementación esté vacía resaltar() deja de ser abstracto

Clases abstractas: ejemplo ampliado

```
public abstract class Figura {  
    public abstract double perimetro();  
    public abstract void resaltar();  
}  
  
abstract class FiguraColor extends Figura { //Error sin abstract  
    Color colorLinea, colorFondo;  
}  
  
class TrianguloColor extends FiguraColor {  
    Punto2D a, b, c;  
    public double perimetro() {  
        return a.distancia(b) + b.distancia(c) + c.distancia(a);  
    }  
    public void resaltar() {  
        colorLinea.brillo();  
        colorFondo.brillo();  
    }  
}
```

Clases abstractas: ejemplo ampliado

```
public abstract class Figura {  
    public abstract double perimetro();  
    public abstract void resaltar();  
}  
  
abstract class FiguraColor extends Figura { //Error sin abstract  
    Color colorLinea, colorFondo;  
}  
  
class TrianguloColor extends FiguraColor {  
    Punto2D a, b, c;  
    public double perimetro() {  
        return a.distancia(b) + b.distancia(c) + c.distancia(a);  
    }  
    public void resaltar() {  
        colorLinea.brillo();  
        colorFondo.brillo();  
    }  
}
```

*¿Puede un
método privado
ser abstracto?*

Utilidad de los métodos abstractos

```
public class GrupoFiguras { // lista de objetos Figura
    private ArrayList<Figura> figuras
                                = new ArrayList<Figura>();
```

```
    public void agregarFigura(Figura fig) {
        figuras.add(fig);
    }
```

Instanciando la clase genérica ArrayList<T> con Figura como T se pueden añadir objetos de cualquier subclase de Figura a la lista

```
    public void resaltar() {
        Iterator<Figura> iterador = figuras.iterator();
        while (iterador.hasNext()) {
            iterador.next().resaltar();
        }
        return;
    }
}
```

Se invoca a resaltar() sobre cada Figura sin necesidad de saber (en compilación) de qué subclase de Figura se trata

Utilidad de los métodos abstractos

```
public class GrupoFiguras { // lista de objetos Figura  
    private List<Figura> figuras = new ArrayList<Figura>();
```

También válido: List es una interfaz, implementado por colecciones como ArrayList o Vector.

```
    public void agregarFigura(Figura fig) {  
        figuras.add(fig);  
    }
```

```
public void resaltar() {
```

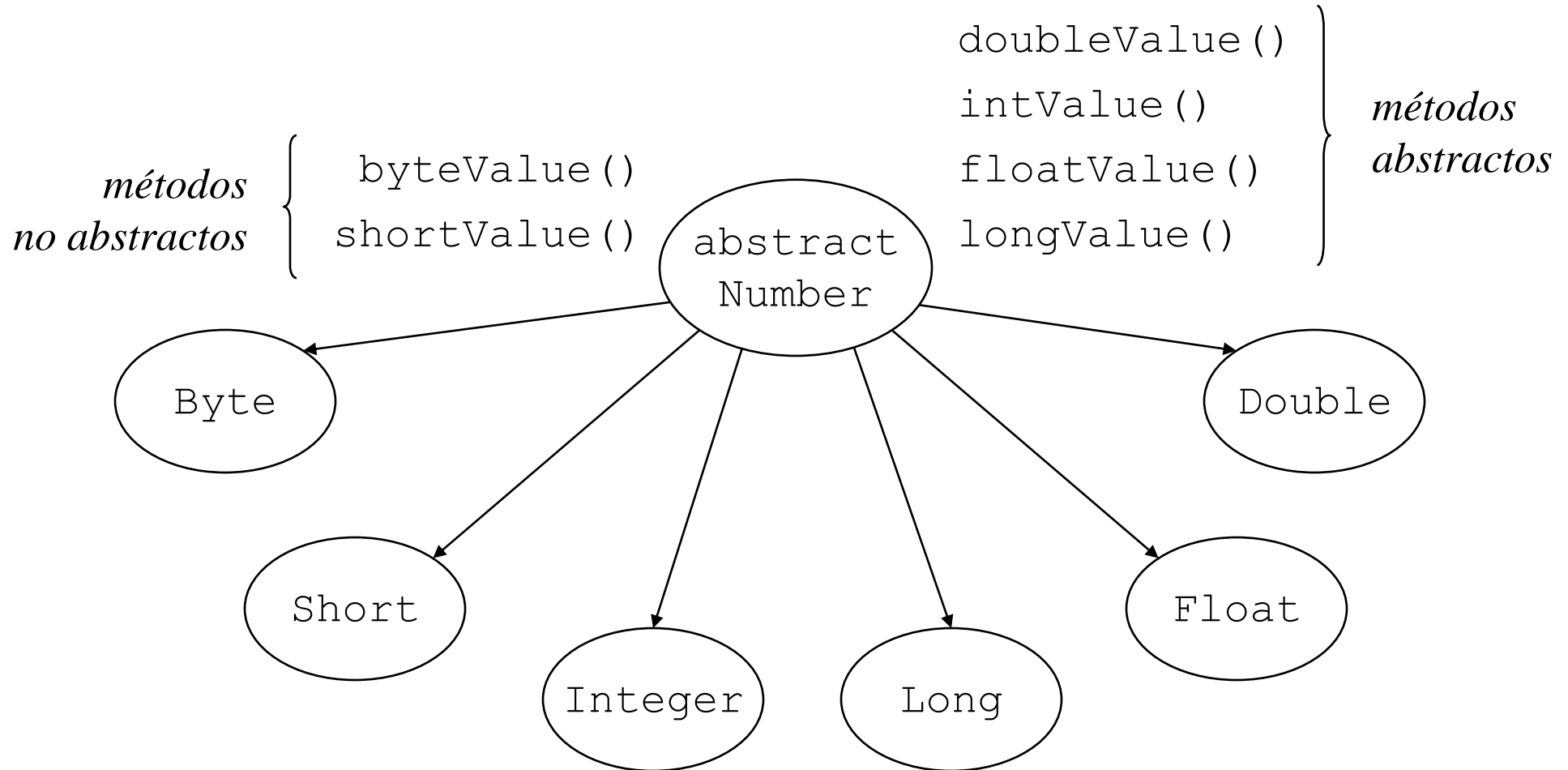
```
    for (Figura f: figuras) {  
        f.resaltar();  
    }
```

Podemos usar el for mejorado para iterar sobre colecciones.

```
    return;
```

```
}
```

java.lang.Number es una clase abstracta



Ejercicio

- Crear un programa que:
 - Emule un sistema de ficheros sencillo
 - Un fichero tiene nombre, tamaño en bytes, y un tipo (R, RW, W)
 - Una carpeta tiene un nombre y puede contener ficheros o carpetas. Su tamaño viene dado por el tamaño de los ficheros y carpetas que contiene.
- Mejorar el programa para evitar añadir un elemento a una carpeta si:

¿Cómo comprobarías que un elemento no está en dos carpetas?

Modificador **final** en atributo, método y clase

- Atributos con **final** (*parecidas a constantes*)
 - Su primer valor asignado no cambia después
 - Los atributos de instancia con final deben inicializarse en su declaración, o asignarse en cada constructor
 - Los atributos de clase con final deben inicializarse en su declaración, o asignarse en la *inicialización de la clase*
- Métodos con **final**
 - No pueden ser modificados (sobrescritos) en subclases que los hereden
- Clases con **final**
 - No pueden ser *extendidas* mediante subclases

Clase, método y atributo con `final`: ejemplo

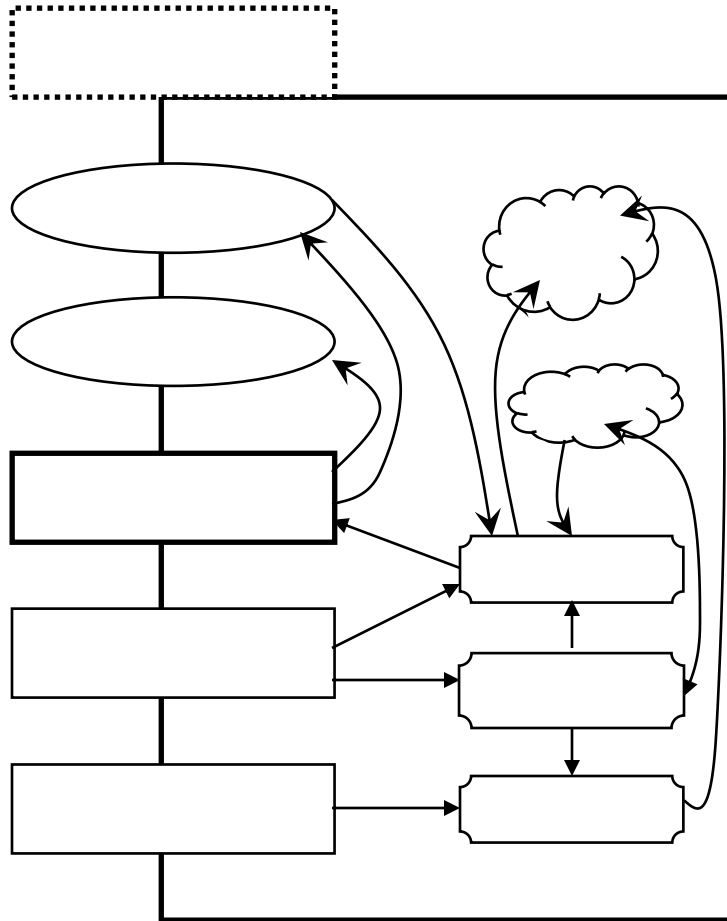
```
final class ClaseA {  
    ...  
}  
class ClaseB extends ClaseA { // Error: ClaseA no permite subclasses  
    private final int x;        // Error: no se asigna valor a x  
    private final int y = 0;    // OK: se inicializa y  
    private final int z;        // OK: no se inicializa z pero ...  
                                // se le da valor en todos los constructores  
  
    public ClaseB() { x = 0; z = 0; }  
    public ClaseB(int n) { z = n; }  
  
    public final double f(int x) { return (x-1)/(x+1); }  
}  
  
class ClaseC extends ClaseB {  
    public double f(int x) { //Error: f no se puede sobrescribir  
        return (x-2)/(x+2);  
    }  
}
```






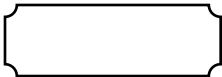
Noción de paquete en Java

- Conjunto de clases relacionadas entre sí, que se ofrecen al programador de aplicaciones como una unidad software cerrada
- Solo las clases públicas del paquete son accesibles desde fuera (usando `import` o con notación `paquete.clase`)
- Evita conflictos de símbolos entre clases de distintos paquetes
- Cada clase solo puede pertenecer a un paquete
- Los paquetes pueden dividirse jerárquicamente en subpaquetes
- Si no se define ningún paquete para una clase, queda incluida en el paquete por defecto (sin nombre, no importable, mejor no usar).
- Además de clases y enumerados, los paquetes pueden agrupar *interfaces*, que estudiaremos pronto

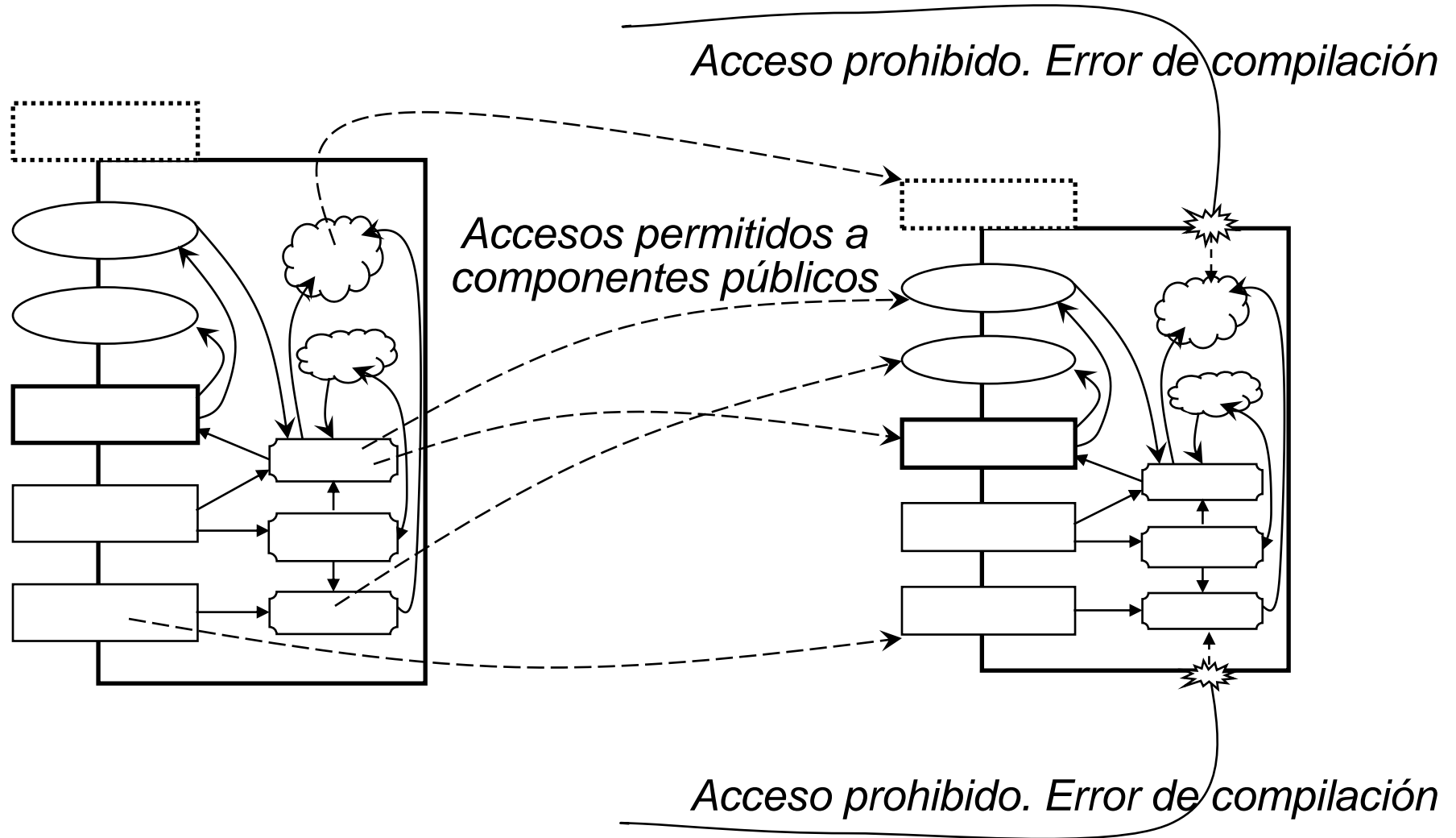
Visión esquemática de una clase

Leyenda:



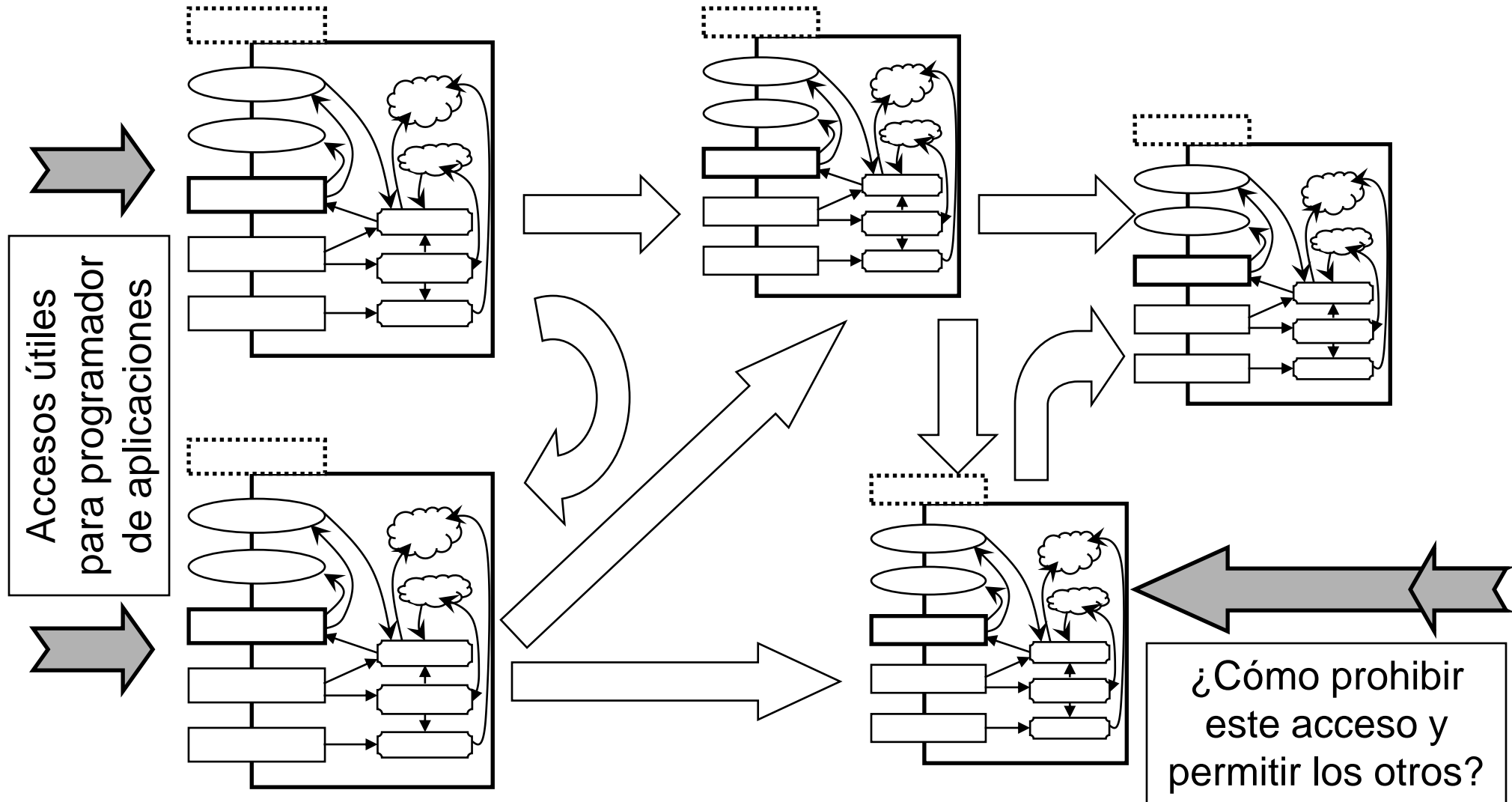
-  Nombre de clase
-  Variable pública
-  Constructor público
-  Método público
-  Variable privada
-  Método privado

Relaciones y accesos entre clase separadas



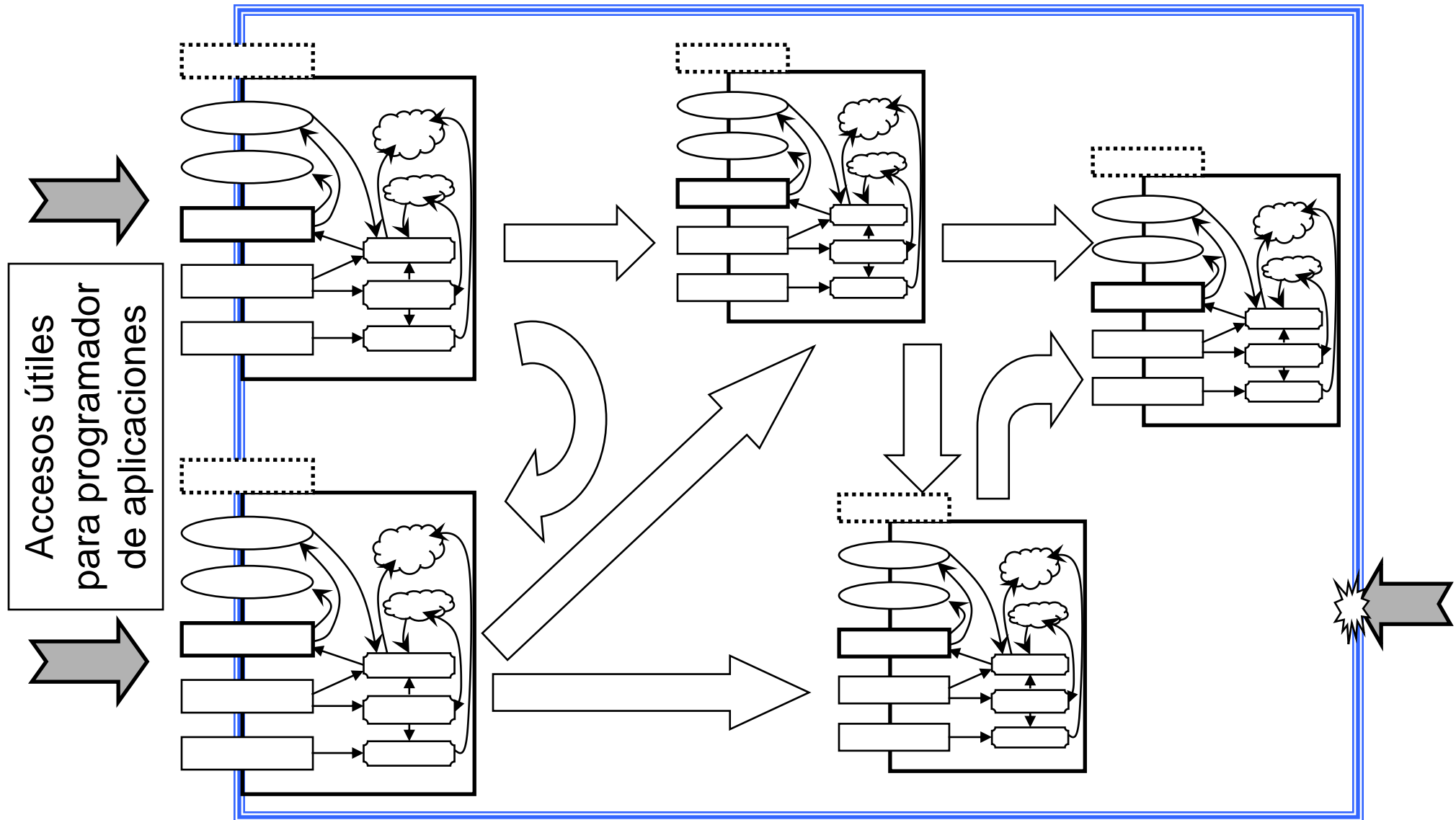
Clases separadas interrelacionadas

Si solo algunas de esas clases tienen sentido para el programador de aplicaciones, deberíamos empaquetarlas y definir un interfaz



Clases agrupadas en un paquete

Solo las clases públicas (más externas) del paquete son accesibles



Cómo definir paquetes

Comenzando cada unidad de compilación con declaración **package**

Almacenando todas las unidades de compilación de un mismo paquete en un directorio cuyo nombre coincida con el del paquete

graphics/Circle.java

```
package graphics;  
  
public class Circle {  
    public void paint() {  
        ...  
    }  
    ...  
}
```

graphics/Rectangle.java

```
package graphics;  
  
public class Rectangle {  
    public void paint() {  
        ...  
    }  
    ...  
}
```

Cómo utilizar clases de otro paquete

Uso directo con notación `paquete.clase`

```
...  
graphics.Circle c = new graphics.Circle();  
c.paint ();  
...
```

Importar una clase

```
import graphics.Circle;  
...  
Circle c = new Circle();  
c.paint();  
...
```

Importar todas la clases del paquete

```
import graphics.*;  
...  
Circle c = new Circle();  
Rectangle r = new Rectangle();  
c.paint(); r.paint();  
...
```

Import static

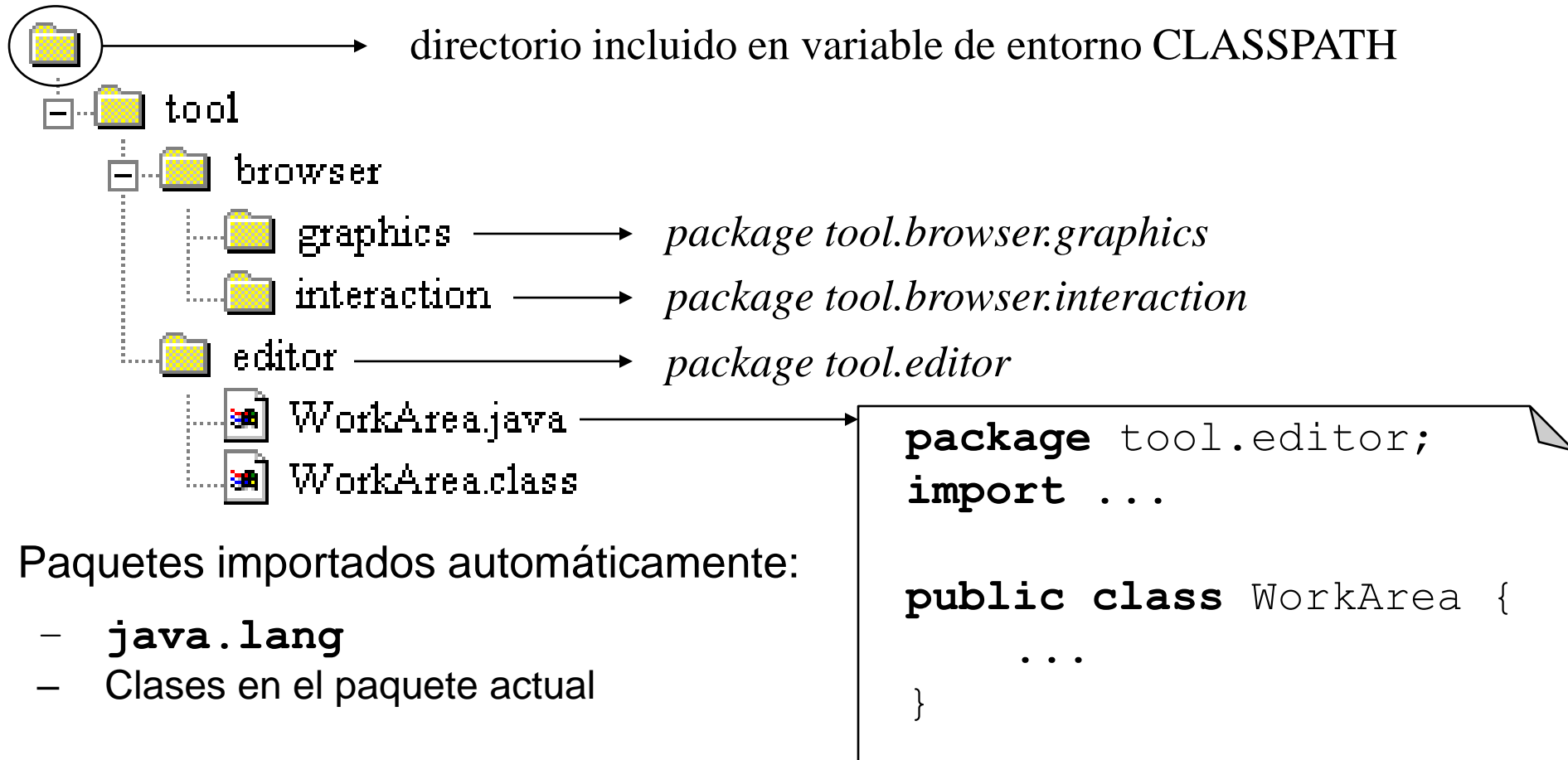
- Importar un atributo o método estático declarado en otra clase
- Evita tener que usar `<name-class>.<name-method>()`, y permite `<name-method>()` en su lugar
- Ejemplo:

```
import static java.lang.Math.PI;  
import static java.lang.Math.abs;
```

```
public class Main {  
    public static void main(String ...args) {  
        System.out.println("PI="+PI);  
        System.out.println("abs(-45)="+abs(-45));  
    }  
}
```

Ubicación de paquetes en directorios

- Nombre del paquete → estructura de directorios
- Variable de entorno **CLASSPATH** en el sistema operativo: contiene la lista de directorios donde Java busca paquetes



- Paquetes importados automáticamente:
 - **java.lang**
 - Clases en el paquete actual

Cómo definir subpaquetes

Comenzar cada unidad de compilación con declaración **package**

Ubicar unidades de compilación de un mismo subpaquete en un subdirectorio cuyo nombre coincida con el del subpaquete y respetar la jerarquía de (sub)directorios y (sub)paquetes

graphics/color/Circle.java

```
package graphics.color;  
  
public class Circle {  
    public void paint() {  
        ...  
    }  
    ...  
}
```

graphics/blackwhite/Circle.java

```
package graphics.blackwhite;  
  
public class Circle {  
    public void paint() {  
        ...  
    }  
    ...  
}
```

Paquetes predefinidos en Java (API)

java.applet

java.awt

java.awt.datatransfer

java.awt.event

java.awt.image

java.beans

java.io

java.lang

java.lang.reflect

java.math

java.net

java.rmi

java.rmi.dgc

java.rmi.registry

java.rmi.server

java.security

java.security.acl

java.security.interfaces

java.sql

java.text

java.util

java.util.zip

...

<http://docs.oracle.com/javase/19/docs/api/>

(actualmente cerca de 300)

Módulos Java

- Desde la versión 9, Java permite declarar módulos
- Mecanismo adicional de modularidad al que propocionan los paquetes
 - Útil para construir aplicaciones muy grandes, con alto número de paquetes
 - Mejor encapsulación
 - Similar a la noción de “*plugin*” en sistemas de componentes como OSGi o Eclipse.
- Declara una agrupación de paquetes (y recursos), con restricciones de acceso a los mismos, dependencias, etc.
- No lo veremos ni utilizaremos en la asignatura (se recomienda no usarlo en las prácticas)

Control de acceso o visibilidad

Ya hemos usado **public** y **private**

```
class ClaseA {  
    public int x;  
    private int y;  
    public void f() { ... }  
    private void g() { ... }  
    void h()  
        x = 2;  
        y = 6;  
        f();  
        g();  
        ClaseA a = new ClaseA();  
        a.x = 2;  
        a.y = 6;  
        a.f();  
        a.g();  
}
```

```
class ClaseB {  
    void m() {  
        ClaseA a = new ClaseA();  
        a.x = 2;  
        a.y = 6; //Error private  
        a.f();  
        a.g(); //Error private  
        a.h();  
    }  
}
```

¿Qué visibilidad tiene `h()` ?

Control de acceso: otras modalidades

Modalidades de visibilidad de variables, métodos y constructores de una clase

	<u>Clase</u>	<u>Package</u>	<u>Subclase</u>	<u>Cualquiera</u>
private	X			
(por defecto: <i>package</i>)	X	X		
protected	X	X	X	
public	X	X	X	X

Modalidades de control de acceso para clases:

- *top-level* clases, solamente **public** o *package* (por defecto)
- *clases internas* a otras clases, también **protected** o **private**

Nota: *protected* y *package* son equivalentes si la superclase y la subclase están en el mismo paquete

Control de acceso de variables y métodos dentro de clases

`archivo.java` \longleftrightarrow *Unidad de compilación única*

```
class ClaseA {  
    int w;    // package  
    private int x;  
    protected int y;  
    public int z;  
    private void g() { ... }  
    void h() {  
        w = 2;  
        x = 6;  
        y = 8;  
        z = 3;  
        g();  
    }  
}
```

```
class ClaseB {  
    void m() {  
        ClaseA a = new ClaseA();  
        a.w = 2;  
        a.x = 6; //Error private  
        a.y = 8;  
        a.z = 3;  
        a.g();    //Error private  
        a.h();  
    }  
}
```

Control de acceso dentro de clases

Dos unidades de compilación: paquete único (por defecto)

A.java

B.java

```
class ClaseA {  
    int w;    // package  
    private int x;  
    protected int y;  
    public int z;  
    private void g() { ... }  
    void h() {  
        w = 2;  
        x = 6;  
        y = 8;  
        z = 3;  
        g();  
    }  
}
```

```
class ClaseB {  
    void m() {  
        ClaseA a = new ClaseA();  
        a.w = 2;  
        a.x = 6; //Error private  
        a.y = 8;  
        a.z = 3;  
        a.g();    //Error private  
        a.h();  
    }  
}
```

Control de acceso a clases en paquetes

Dos unidades de compilación y dos paquetes: p1 y paquete por defecto

A.java

```
package p1;  
class ClaseA { // package  
    int w;    // package  
    private int x;  
    protected int y;  
    public int z;  
    private void g() { ... }  
    void h() {  
        w = 2;  
        x = 6;  
        y = 8;  
        z = 3;  
        g ();  
    }  
}
```

B.java

```
class ClaseB {  
    void m() {  
        ClaseA a = new ClaseA();  
        // Error:  
        // ClaseA no encontrada  
        // importamos p1, o bien  
        // probamos p1.ClaseA()  
        // Error:  
        // ClaseA no pública en p1  
    }  
}
```

Control de acceso a clases en paquetes

Dos unidades de compilación y dos paquetes

p1/ClaseA.java

```
package p1;  
public class ClaseA {  
    int w;    // package  
    private int x;  
    protected int y;  
    public int z;  
    private void g() { ... }  
    void h() {  
        w = 2;  
        x = 6;  
        y = 8;  
        z = 3;  
        g ();  
    }  
}
```

B.java

```
import p1.ClaseA;  
class ClaseB {  
    void m() {  
        ClaseA a = new ClaseA();  
        a.w = 2; //Error package  
        a.x = 6; //Error private  
        a.y = 8; //Error protected  
        a.z = 3;  
        a.g();    //Error private  
        a.h();    //Error package  
    }  
}
```

También valdría con `import p1.*;`
O bien, sin import con `p1.ClaseA()`

Agrupación de clases en un paquete

*Dos unidades de compilación y **un solo paquete**: p1*

p1/ClaseA.java

```
package p1;  
public class ClaseA {  
    int w;    // package  
    private int x;  
    protected int y;  
    public int z;  
    private void g() { ... }  
    void h() {  
        w = 2;  
        x = 6;  
        y = 8;  
        z = 3;  
        g ();  
    }  
}
```

p1/B.java

```
package p1;  
class ClaseB {  
    void m() {  
        ClaseA a = new ClaseA();  
        a.w = 2;    // ok package  
        a.x = 6;    //Error private  
        a.y = 8;    // ok protected  
        a.z = 3;  
        a.g();      //Error private  
        a.h();      // ok package  
    }  
}
```

También valdría con **import p1.*;**
O bien, sin import con **p1.ClaseA()**

Clases en paquetes distintos

Dos unidades de compilación y dos paquetes: p1 y p2

p1/ClaseA.java

```
package p1;
public class ClaseA {
    int w;    // package
    private int x;
    protected int y;
    public int z;
    private void g() { ... }
    void h() {
        w = 2;
        x = 6;
        y = 8;
        z = 3;
        g ();
    }
}
```

p2/ClaseB.java

```
package p2;
import p1.*;
public class ClaseB {
    void m() {
        ClaseA a = new ClaseA();
        a.w = 2; //Error package
        a.x = 6; //Error private
        a.y = 8; //Error protected
        a.z = 3;
        a.g();    //Error private
        a.h();    //Error package
    }
}
```

Agrupación de clases dentro de paquetes

Las clases públicas de un paquete son importables

p1/ClaseA.java

```
package p1;
public class ClaseA {
    ...
}

// clase no publica del
// paquete p1
class Aux {
    ...
}

}
```

p2/ClaseB.java

```
package p2;
import p1.*;
public class ClaseB {
    void m() {
        ClaseA a = new ClaseA();
        ...
        Aux = new Aux(); // Error
        ...
    }
}
```

Incluso con `p1.Aux` hay error

Acceso permitido a subclasses: **protected**

Pero solo a través de expresiones del tipo de las subclasses

p1/ClaseA.java

```
public class ClaseA {  
    int w;    // package  
    private int x;  
    protected int y;  
    public int z;  
    private void g() { ... }  
    void h() {  
        w = 2;  
        x = 6;  
        y = 8;  
        z = 3;  
        f ();  
    }  
}
```

p2/ClaseB.java

```
package p2;  
import p1.*;  
public class ClaseB  
    extends ClaseA {  
    void m() {  
        ClaseA a = new ClaseA();  
        ClaseB b = new ClaseB();  
  
        a.y = 8; //Error protected  
        b.y = 7; // ok protected  
        a = new ClaseB();  
        a.y = 6; //Error protected  
        y = 5;    // ok protected  
    }  
}
```

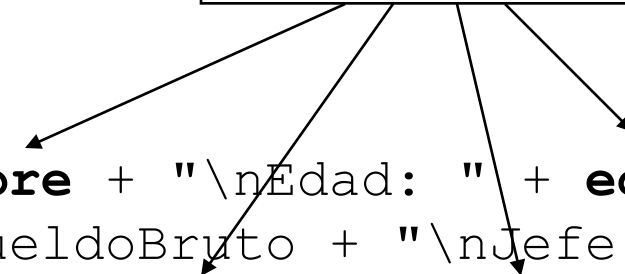
Nota: Sólo es problemático si ClaseA y ClaseB están en distintos paquetes

Ejemplo control de acceso: public, private, package

```
class Persona {  
    private String nombre;  
    private int edad;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}
```

```
class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad +  
            "\nSueldo: " + sueldoBruto + "\nJefe: " +  
            ((jefe == null)? nombre : jefe.nombre);  
    }  
}
```

*Error: nombre y
edad son privados*



Ejemplo de control de acceso: protected (I)

```
class Persona {  
    protected String nombre;  
    protected int edad;  
    public String toString() {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}
```

Error: sólo puede ser public

```
class Empleado extends Persona {  
    long sueldoBruto;  
    Directivo jefe;  
    String toString() {  
        return super.toString () +  
            "\nSueldo: " + sueldoBruto + "\nJefe: " +  
            ((jefe == null)? nombre : jefe.nombre);  
    }  
}
```

*Correcto incluso si
Empleado y Persona
en distinto paquete*

*En distinto paquete habría error si jefe
fuera Persona, pero no si fuese Empleado*

Ejemplo de control de acceso: protected (II)

```
package personal;
```

```
public class Persona {  
    protected String nombre;  
    protected int edad;  
    protected String idString () {  
        return "Nombre: " + nombre + "\nEdad: " + edad;  
    }  
}
```

*Cuidado: toString
se hereda con public*

```
package personal;
```

```
...  
// En cualquier clase  
Persona p = new Persona ();  
p.idString ();  
...
```

```
package otro;
```

```
...  
// En cualquier clase  
personal.Persona p =  
    new personal.Persona();  
p.idString (); // Error  
...
```

Ejercicio de control de acceso: protected (III)

```
package p1;

public class A {
    int w; // package
    private int x;
    protected int y;
    public int z;
}
```

```
package p2;

class C {
    void h() {
        p1.A a = new p1.A();
        a.w = 2;
        a.x = 6;
        a.y = 8;
        a.z = 3;
    }
}

class D extends p1.A {
    void h() {
        p1.A a = new p1.A();
        w = 2; a.w = 2;
        x = 2; a.x = 6;
        z = 3; a.z = 3;
        a.y = 8;

        y = 8;
        D d = new D ();
        d.y = 8;
    }
}
```