Grado Superior
Programación

Herencia y polimorfismo

COLEGIO litterator

Curso 2023/24

Profesor
Jonathan Carrero

Java

Herencia

Polimorfismo, vinculación dinámica y casting

Concepto

Permite **reutilizar código**, definiendo nuevas clases a partir de otras ya existentes.

Si una clase hereda de otra diremos que la *especializa*.

Si una clase hereda de otra, en particular hereda todos sus métodos y atributos **públicos o protegidos**, pudiendo además declarar y definir más atributos y métodos, y **redefinir** algunos de los métodos heredados.

Jerarquía de clases

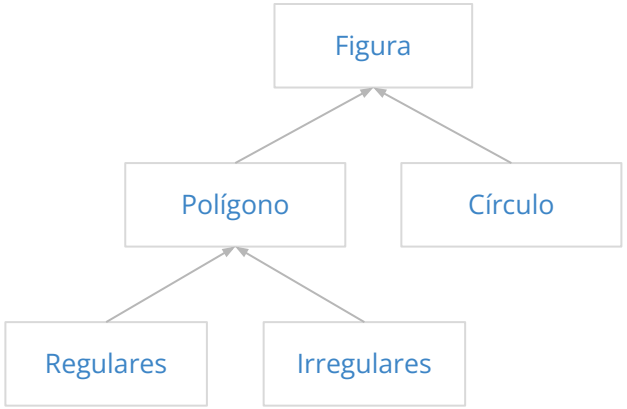
Por lo tanto, si una clase **B** hereda de otra clase **A** entonces:

- **B** hereda los atributos y métodos de la clase **A**.
- **B** puede, además:
  - Definir nuevos atributos.
  - Implementar nuevos métodos.
  - Redefinir métodos que ya existen.

Jerarquía de clases

Por lo tanto, si una clase **B** hereda de otra clase **A** entonces:

- **B** hereda los atributos y métodos de la clase **A**.
- **B** puede, además:
  - Definir nuevos atributos.
  - Implementar nuevos métodos.
  - Redefinir métodos que ya existen.



Sintaxis

Utilizamos la palabra reservada `extends`:

Se indica que la clase SubClase **hereda** de la clase SuperClase o que la SubClase **extiende** a la clase SuperClase.

Sintaxis

Utilizamos la palabra reservada `extends`:

```
public class SubClase extends SuperClase {
        // Cuerpo de la subclase
}
```

Se indica que la clase SubClase **hereda** de la clase SuperClase o que la SubClase **extiende** a la clase SuperClase.

Herencia, ejemplo 1

```
public class Persona {
	protected long dni;
	protected String nombre;

	public Persona() {
		dni = -1;
		nombre = “”;
	}
	public Persona(long dni, String n){
		this.dni = dni;
		this.nombre = n;
	}
	public void setDNI(long dni) {
		this.dni = dni;
	}
	public void setNombre(String n){
		this.nombre = n;
	}
}
```

```
public class Alumno extends Persona {
	private long idMatricula;
	private int creditosAprobados;

	public Alumno() {
		this.idMatricula = -1;
		this.creditosAprobados = 0;
	}
	public void setIdMatritula(long id) {
		this.idMatricula = id;
	}
	public void setCreditosAP(int ca) {
		this.creditosAprobados = ca;
	}
}
```

Herencia, ejemplo 1

```
public class Persona {
	protected long dni;
	protected String nombre;

	public Persona() {
		dni = -1;
		nombre = “”;
	}
	public Persona(long dni, String n){
		this.dni = dni;
		this.nombre = n;
	}
	public void setDNI(long dni) {
		this.dni = dni;
	}
	public void setNombre(String n){
		this.nombre = n;
	}
}
```

```
public class Alumno extends Persona {
	private long idMatricula;
	private int creditosAprobados;

	public Alumno() {
		this.idMatricula = -1;
		this.creditosAprobados = 0;
	}
	public void setIdMatritula(long id) {
		this.idMatricula = id;
	}
	public void setCreditosAP(int ca) {
		this.creditosAprobados = ca;
	}
}
```

Si instancio un objeto de tipo `Alumno`, ¿qué métodos podré utilizar?

Herencia, ejemplo 1

Como hemos visto, puedo usar los métodos **heredados** de su super clase y, además, los nuevos métodos que hayamos definido en la propia clase `Alumno`.

```
// Creo una instancia de la subclase Alumno
Alumno a = new Alumno();
// Uso un método de su super clase Persona
a.setNombre(“Pedro Gómez”);
// Uso un método de su super clase Persona
a.setDNI(“123456J”);
// Uso un método de su propia clase
a.setCreditosAP(9);
```

Niveles de protección

En la implementación de la clase `SubClase` podemos utilizar todos los atributos públicos y protegidos de la clase `SuperClase`, pero **nunca** los privados.

El identificador `protected` para métodos y atributos/campos indica que estos se comportan como **públicos** en presencia de herencia.

En caso de que no haya herencia, `protected` se comporta como atributo público dentro del paquete que contiene a la clase.

Niveles de protección

	Público	Protegido	Por defecto	Privado
Clase en el mismo paquete	✓	✓	✓	✗
Subclase en el mismo paquete	✓	✓	✓	✗
Clase en otro paquete	✓	✗	✗	✗
Subclase en otro paquete	✓	✓	✗	✗

Constructoras

Para inicializar a un objeto de una subclase es necesario invocar a **alguna** constructora de su superclase.

En caso de no poner la llamada explícita, se invocará por **defecto** a la constructora sin argumentos de la superclase.

La invocación a la constructora de la superclase debe ser **la primera** instrucción de cualquier constructora de la subclase.

Constructoras

Para invocar a la constructora sin argumentos de la superclase:

```
super();
```

Para invocar a una constructora con argumentos:

```
super(a1, …, an)
```

Constructoras

```
public class A1 {
	public A1(){
		System.out.println("A1: Por defecto");
	}
	public A1(int x){
		System.out.println("A1: Un argumento");
	}
}
```

```
public class A2 extends A1 {
	public A2(){
		System.out.println("A2: Por defecto");
	}
	public A2(int pp){
		super(pp); // ¿Y si se quita?
		System.out.println("A2: Un argumento");
	}
}
```

Constructoras

```
public class A1 {
	public A1(){
		System.out.println("A1: Por defecto");
	}
	public A1(int x){
		System.out.println("A1: Un argumento");
	}
}
```

```
public class A2 extends A1 {
	public A2(){
		System.out.println("A2: Por defecto");
	}
	public A2(int pp){
		super(pp); // ¿Y si se quita?
		System.out.println("A2: Un argumento");
	}
}
```

```
A2 a = new A2();
A2 b = new A2(2);
```

¿Qué muestra por pantalla?

Constructoras

```
public class A3 extends A2 {
	public A3(){
		System.out.println("A3: Por defecto");
	}
	public A3(int p1, int p2){
		super(p1);
		System.out.println("A3: Dos argumentos");
	}
}
```

```
public class A4 extends A3 {
	public A4(int x){
		super(x,x);
		System.out.println("A4: Un argumento");
	}
}
```

Constructoras

```
public class A3 extends A2 {
	public A3(){
		System.out.println("A3: Por defecto");
	}
	public A3(int p1, int p2){
		super(p1);
		System.out.println("A3: Dos argumentos");
	}
}
```

```
public class A4 extends A3 {
	public A4(int x){
		super(x,x);
		System.out.println("A4: Un argumento");
	}
}
```

```
A3 a = new A3();
A3 b = new A3(2,3);
```

¿Qué muestra por pantalla?

Constructoras

```
public class A3 extends A2 {
	public A3(){
		System.out.println("A3: Por defecto");
	}
	public A3(int p1, int p2){
		super(p1);
		System.out.println("A3: Dos argumentos");
	}
}
```

```
public class A4 extends A3 {
	public A4(int x){
		super(x,x);
		System.out.println("A4: Un argumento");
	}
}
```

```
A3 a = new A3();
A3 b = new A3(2,3);
```

¿Qué muestra por pantalla?

```
A4 a = new A4(2);
```

¿Qué muestra por pantalla?

Colisión de nombres con herencia

Si una subclase contiene un atributo con el mismo nombre que su superclase, entonces **toda** referencia a ese atributo en la implementación de la subclase **se referirá** al de la subclase.

Para hacer referencia al de la superclase hay que utilizar la palabra `super`.

Colisión de nombres con herencia

Si una subclase contiene un atributo con el mismo nombre que su superclase, entonces **toda** referencia a ese atributo en la implementación de la subclase **se referirá** al de la subclase.

Para hacer referencia al de la superclase hay que utilizar la palabra `super`.

```
class Padre {
	protected int c;
	public Padre() {
		this.c = 15;
	}
}
```

```
class Hija extends Padre {
	protected int c;
	public Hija() {
		super();
		this.c = 12;
	}
	public int suma(){
		return this.c + super.c;
	}
}
```

Métodos

Al igual que los campos, los métodos públicos y protegidos **también** se heredan.

Cualquier método heredado se puede **redefinir**. Se denomina **sobreescritura** de un método.

Cuando se invoca a un método, primero se busca si dicho método existe en la clase del objeto (es decir, en la clase asociada a su `new`). Si existe, se ejecuta. En otro caso, se sube por la jerarquía de clases hasta encontrarle. Y una vez que se encuentra, se ejecuta.

También es posible redefinir un método e invocar dentro de su cuerpo al equivalente en su superclase.

Métodos, ejemplo 1

```
public class A1 {
      public void m(){
            System.out.println("A1: metodo m");
      }
}

public class A2 extends A1 {}

public class A3 extends A2 {}
```

Métodos, ejemplo 1

```
public class A1 {
      public void m(){
            System.out.println("A1: metodo m");
      }
}

public class A2 extends A1 {}

public class A3 extends A2 {}
```

```
A3 a = new A3();
     a.m();
```

¿Qué método se ejecuta?

Métodos, ejemplo 2

```
public class A1 {
	public void m(){
		System.out.println("A1: metodo m");
	}
}

public class A2 extends A1 {
	public void m(){
		System.out.println("A2: metodo m");
	}
}

public class A3 extends A2 {}
```

Métodos, ejemplo 2

```
public class A1 {
	public void m(){
		System.out.println("A1: metodo m");
	}
}

public class A2 extends A1 {
	public void m(){
		System.out.println("A2: metodo m");
	}
}

public class A3 extends A2 {}
```

```
A3 a = new A3();
a.m();
```

¿Qué método se ejecuta?

Métodos, ejemplo 3

```
public class A1 {
	public void m(){
		System.out.println("A1: metodo m");
	}
}

public class A2 extends A1 {
	public void m(){
		System.out.println("A2: metodo m");
	}
}

public class A3 extends A2 {}
```

Métodos, ejemplo 3

```
public class A1 {
	public void m(){
		System.out.println("A1: metodo m");
	}
}

public class A2 extends A1 {
	public void m(){
		System.out.println("A2: metodo m");
	}
}

public class A3 extends A2 {}
```

```
A2 a = new A2();
a.m();
```

¿Qué método se ejecuta?

Métodos

Cuando se redefine un método es posible invocar al de la superclase utilizando el objeto `super`.

```
super.m();
```

Utilizar `super` supone subir por la jerarquía de clases **hasta encontrar** el método con ese nombre y argumentos del mismo tipo.

Métodos, ejemplo 4

```
public class A1 {
	public void m(){
		System.out.println("A1: metodo m");
	}
}

public class A2 extends A1 {}

public class A3 extends A2 {
	public void m(){
		System.out.println("A3: metodo m");
		super.m();
	}
}

public class A4 extends A3 {}
```

Métodos, ejemplo 4

```
public class A1 {
	public void m(){
		System.out.println("A1: metodo m");
	}
}

public class A2 extends A1 {}

public class A3 extends A2 {
	public void m(){
		System.out.println("A3: metodo m");
		super.m();
	}
}

public class A4 extends A3 {}
```

```
A4 a = new A4();
a.m();
```

¿Qué sale por pantalla?

Modificador `final`

El modificador `final` sobre un método indica que dicho método **no puede** ser **redefinido**.

El modificador `final` sobre una clase indica que **no es posible heredar** de dicha clase.

Sobrecarga

Dentro del mismo ámbito podemos tener métodos con el mismo nombre, tipo de retorno, mismo número de argumentos pero de **distinto tipo**.

```
class A {
      public void show(float d){…}
      public void show(String cad){…}
      public void show(int i){…}
}
```

Sobrecarga

Dentro del mismo ámbito podemos tener métodos con el mismo nombre, tipo de retorno, mismo número de argumentos pero de **distinto tipo**.

```
class A {
      public void show(float d){…}
      public void show(String cad){…}
      public void show(int i){…}
}
```

```
A a = new A();
a.show(3f);
a.show(3);
a.show(“hola”);
```

Sobreescritura

Distintas clases, relacionadas a través de la herencia, implementan el mismo método.

La sobrecarga se resuelve en **tiempo de compilación**.

La sobreescritura se resuelve en **tiempo de ejecución** debido a la existencia del polimorfismo.

Polimorfismo

De momento hemos visto:

- Sobrecarga: dos métodos tienen el mismo nombre, mismo tipo de retorno pero difieren en los parámetros (número y/o tipo).

- Sobreescritura: Una clase hija redefine un método de su superclase.

Con el polimorfismo entran en juego las variables polimórficas. Una variable polimórfica no es más que una variable que se declara de un tipo (clase) pero en **tiempo de ejecución** puede contener valores de distinto tipo (subclase).

Polimorfismo

El término polimorfismo significa que hay un nombre (variable, método o clase) y muchos significados diferentes (distintas definiciones).

El polimorfismo implica que toda variable tiene un tipo **estático** y otro **dinámico**.

El *tipo estático* es el tipo asociado en la **declaración**.

El *tipo dinámico* es el tipo del objeto **contenido** dentro de una variable.

Polimorfismo

El término polimorfismo significa que hay un nombre (variable, método o clase) y muchos significados diferentes (distintas definiciones).

El polimorfismo implica que toda variable tiene un tipo **estático** y otro **dinámico**.

El *tipo estático* es el tipo asociado en la **declaración**.

El *tipo dinámico* es el tipo del objeto **contenido** dentro de una variable.


```
A a = new B();

Tipo estático: A
Tipo dinámico: B
```

Polimorfismo

El término polimorfismo significa que hay un nombre (variable, método o clase) y muchos significados diferentes (distintas definiciones).

El polimorfismo implica que toda variable tiene un tipo **estático** y otro **dinámico**.

El *tipo estático* es el tipo asociado en la **declaración**.

Nada que ver con `static`

El *tipo dinámico* es el tipo del objeto **contenido** dentro de una variable.

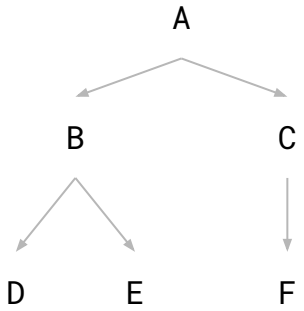

```
A a = new B();

Tipo estático: A
Tipo dinámico: B
```

Polimorfismo

TE (x): Tipo Estático de la variable x.

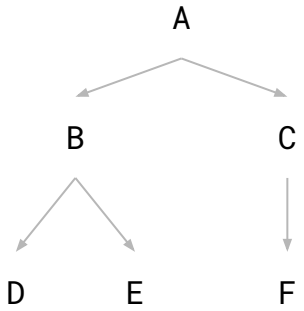
CTD (x): Conjunto de posible Tipos Dinámicos de x.



Polimorfismo

TE (x): Tipo Estático de la variable x.

CTD (x): Conjunto de posible Tipos Dinámicos de x.

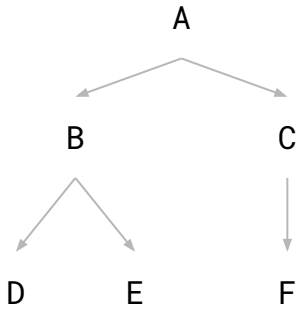



```
A a;
B b;
C c;
```

Polimorfismo

TE (x): Tipo Estático de la variable x.

CTD (x): Conjunto de posible Tipos Dinámicos de x.




```
A a;
B b;
C c;
```

```
TE (a) = A
TE (b) = B
TE (c) = C
```

```
CTD (a) = A, B, C, D, E, F
CTD (b) = B, D, E
CTD (c) = C, F
```

Polimorfismo

Dada una jerarquía de clases:

```
public class A {…}
public class B extends A {…}
public class C extends B {…}
```

Podemos asignar a un elemento de una superclase cualquiera de sus subclases:

```
A a1 = new B();
A a2 = new C();
 B b = new C();
```

Polimorfismo

Cuando una clase B extiende a una clase A hay que entender que la clase A está **contenida** en la clase B (por eso B la extiende).

Por ejemplo, si A = animales y B = perros, está claro que un perro siempre es un animal, pero un animal no siempre es un perro. Por lo tanto tiene sentido:

```
A a = new B(); // Un perro es un animal
```

Pero no tiene sentido:

```
A a = new B(); // Un perro es un animal
```

Vinculación dinámica

Al establecer una jerarquía de clases, algunas pueden sobreescribir métodos.

En **tiempo de ejecución** se decide qué método se debe aplicar (vinculación dinámica) de acuerdo a las siguientes reglas:

```
A a = new B(…);
a.p(…);
```

1. Se comprueba que `p(…)` está definido en A (tipo estático) o en alguna superclase de A. En otro caso, error.
2. Desde B subiendo por sus superclases, buscamos la primera implementación de `p(….)` y la aplicamos.

Vinculación dinámica, tarea

Es importante saber cómo funciona la vinculación dinámica ya que, potencialmente, aparecerá siempre que estemos en presencia de herencia.

A continuación puedes ver algunos lenguajes que soportan herencia simple y múltiple, por lo que entender su funcionamiento es esencial a la hora de programar.

Herencia múltiple:	Herencia simple:
`C++, Python y Eiffel`	`Java, Ada, C#`

Tarea: se proponen cinco casos de uso. Para cada uno de ellos, haz un esquema de herencia y piensa el resultado antes de ejecutarlos. Finalmente, ejecuta el código y razona el resultado obtenido.

Vinculación dinámica, ejemplo 1

```
class A {
      public void p(int x){
            System.out.println(“p: A”);
      }
}
class B extends A {}
class C extends B {}
class D extends C {}
```

```
B b = new C(); // Tipo estático B. Tipo dinámico C.
b.p(2); // ¿Correcto?
```

Vinculación dinámica, ejemplo 2

```
class A {
      public void p(int x){
            System.out.println(“p: A”);
      }
}
class B extends A {
      public void p(int x){
            System.out.println(“p: B”);
      }
}
class C extends B {}
class D extends C {}
```

```
B b = new C(); // Tipo estático B. Tipo dinámico C.
b.p(2); // ¿Correcto? ¿Qué escribe?
```

Vinculación dinámica, ejemplo 3

```
class A {}
class B extends A {
      public void p(int x){
            System.out.println("p: B");
      }
}
class C extends B {
      public void p(int x){
            System.out.println("p: C");
      }
}
class D extends C {}
```

```
B b = new C(); // Tipo estático B. Tipo dinámico C.
b.p(2); // ¿Correcto? ¿Qué escribe?
```

Vinculación dinámica, ejemplo 4

```
class A {}
class B extends A {}
class C extends B {
      public void p(int x){
            System.out.println(“p: C”);
      }
}
class D extends C {}
```

```
B b = new C(); // Tipo estático B. Tipo dinámico C.
b.p(2); // ¿Correcto? ¿Qué escribe?
```

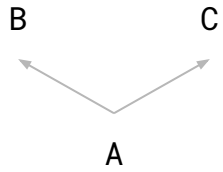
Vinculación dinámica, ejemplo 5

```
class A {}
class B extends A {}
class C extends B {}
class D extends C {
      public void p(int x){
      System.out.println(“p: D”);
      }
}
```

```
B b = new C(); // Tipo estático B. Tipo dinámico C.
b.p(2); // ¿Correcto? ¿Qué escribe?
```

Herencia múltiple

Consiste en que una clase tiene más de una superclase.

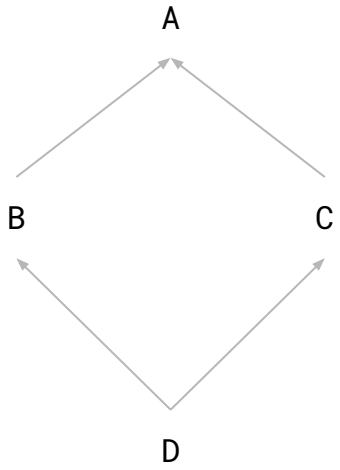


En Java la herencia múltiple está **prohibida** aunque puede obtenerse un comportamiento similar utilizando interfaces.

El principal problema de la herencia múltiple radica en la herencia en forma de diamante.

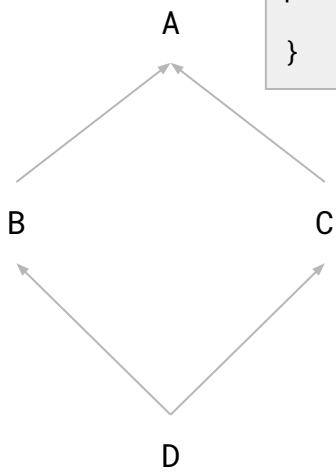
Herencia múltiple

Herencia en forma de diamante:



Herencia múltiple

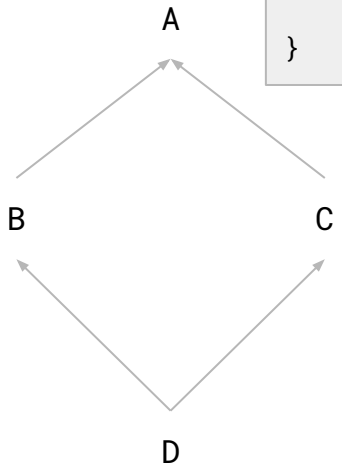
Herencia en forma de diamante:




```
public void p(){
      System.out.println(“p: A”);
}
```

Herencia múltiple

Herencia en forma de diamante:

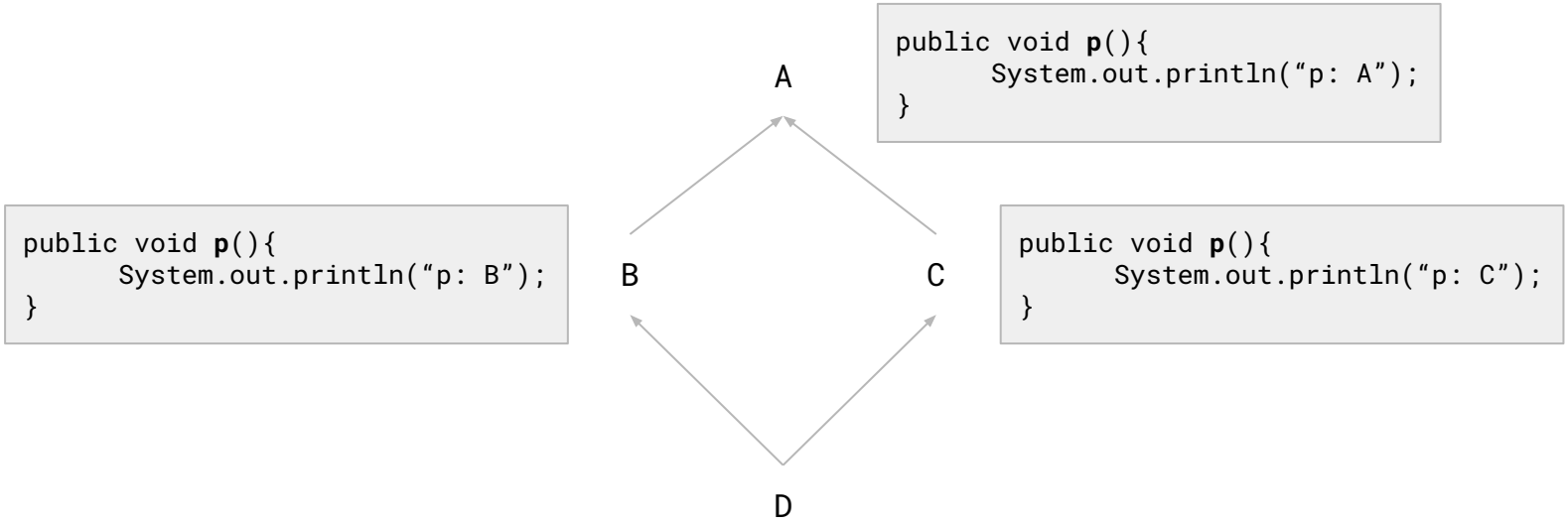



```
public void p(){
      System.out.println(“p: A”);
}
```

```
public void p(){
      System.out.println(“p: C”);
}
```

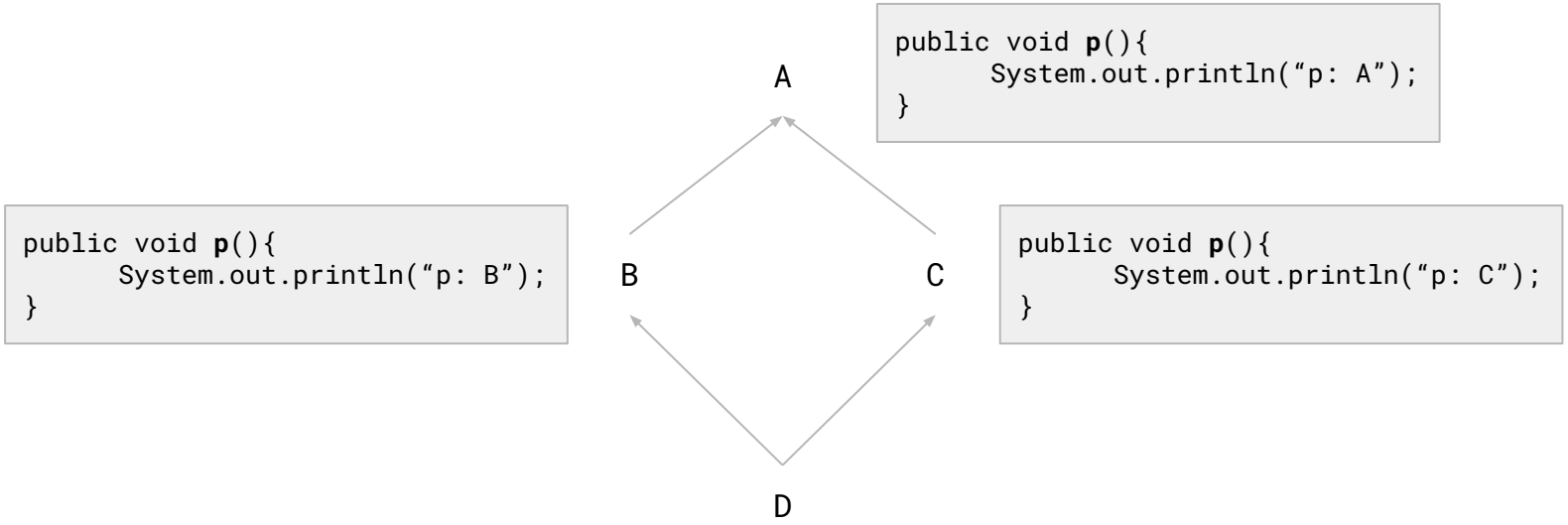
Herencia múltiple

Herencia en forma de diamante:



Herencia múltiple

Herencia en forma de diamante:




```
A a = new D();
a.p(); // ¿Qué p() ejecutamos?
```

Casting

Si B contiene algún método no declarado en A, ni en ninguna de sus posibles superclases, entonces hay que hacer casting. Por ejemplo, si B implementa un método `m` que no está implementado en B, y tenemos la asignación:

```
A a = new B();
```

Entonces para llamar con `a` al método `m` debemos usar la siguiente sintaxis:

```
((B) a).m(...);
```

`((B) a)` hace un casting para que el tipo del objeto A pase a ser B.

Tipo de un objeto en ejecución

Antes de hacer un casting se puede preguntar por el tipo del objeto en tiempo de ejecución. Podemos hacerlo de dos formas:

- Utilizando `getClass()`, heredado de la clase `Object`. Esto devolverá la clase exacta de un objeto en tiempo de ejecución.
- Utilizando `instanceof`. Esto comprobará si el objeto tiene un tipo compatible con la clase correspondiente.

Tipo de un objeto en ejecución

Antes de hacer un casting se puede preguntar por el tipo del objeto en tiempo de ejecución. Podemos hacerlo de dos formas:

- Utilizando `getClass()`, heredado de la clase `Object`. Esto devolverá la clase exacta de un objeto en tiempo de ejecución.
- Utilizando `instanceof`. Esto comprobará si el objeto tiene un tipo compatible con la clase correspondiente.

```
class Simple1 {
	public static void main(String args[]){
		Simple1 s = new Simple1();
		System.out.println(s instanceof Simple1); // true
	}
}
```

Grado Superior
Programación

Herencia y polimorfismo

Curso 2023/24

Profesor
Jonathan Carrero