

Introducción a la programación orientada a objetos

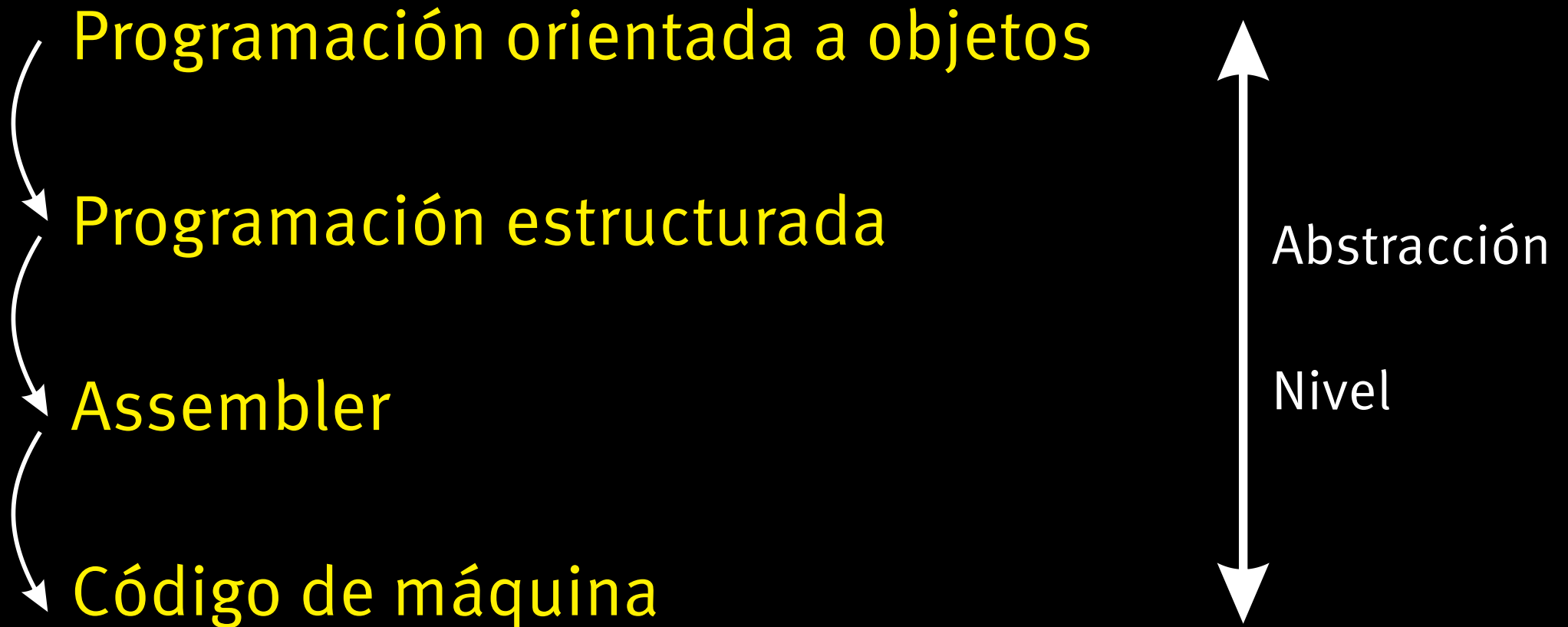
El lenguaje C++

Programación II

Lic. Mauro Gullino maurogullino@gmail.com

UTN FRH

Primero, una aclaración



Caso de estudio

Un cliente solicita un programa para realizar, cada fin de mes, la liquidación de sueldos de sus empleados.

En principio, los empleados son permanentes y cobran un sueldo básico y un 10% adicional por cada año trabajado (antigüedad).

Solución 1

```
struct empleado {
    char nombre[50]; int antiguedad; float basico;
}

int main() {
    struct empleado * todos;
    int cuantos, i;
    float sueldo;
    leer_empleados_archivo("empleados.txt", &todos, &cuantos);

    for(i=0; i<cuantos; i++) {
        sueldo = todos[i].basico * (1+(todos[i].antiguedad * 0.1));
        printf("%s debe cobrar: $ %f", todos[i].nombre, sueldo);
    }
}
```

Modificación 1

"Surgió mucho trabajo. Vamos a necesitar contratar gente temporal, que trabajará un tiempo con un contrato y luego se irá".

Cobrarán un sueldo básico por mes y nada más.

Solución 2

```
struct empleado {
    char nombre[50]; int antiguedad; float basico;
    char es_permanente;
}

int main() {
    struct empleado * todos;
    int cuantos, i;
    float sueldo;
    leer_empleados_archivo("empleados.txt", &todos, &cuantos);

    for(i=0; i<cuantos; i++) {
        sueldo = todos[i].basico;
        if(todos[i].es_permanente)
            sueldo = sueldo * (1+(todos[i].antiguedad * 0.1));
        printf("%s debe cobrar: $ %f", todos[i].nombre, sueldo);
    }
}
```

Modificación 2

"Tuvimos una inspección de AFIP. Necesitamos empezar a retener impuesto a las ganancias y esto tiene que verse en el recibo de sueldo"

Solución 3

```
struct empleado {
    char nombre[50]; int antiguedad; float basico;
    char es_permanente; }

#define PISOGAN 40000
#define IMPGAN 0.2

for(i=0; i<cuantos; i++) {
    sueldo = todos[i].basico;
    if(todos[i].es_permanente)
        sueldo = sueldo * (1+(todos[i].antiguedad * 0.1));
    if(sueldo>PISOGAN) retencion = sueldo * IMPGAN;
    else retencion = 0;

    sueldo = sueldo - retencion;
    printf("%s debe cobrar: $ %f", todos[i].nombre, sueldo);
    printf("se le han retenido $ %f", retencion);
}
```


Modificación 3

(se produce un paro en la fábrica)

"¿Qué hiciste?

¡A los empleados temporales no hay que hacerles retención ni poner eso en el recibo, sólo a los empleados permanentes!"

Solución 4

```
for(i=0; i<cuantos; i++) {  
    sueldo = todos[i].basico;  
    if(todos[i].es_permanente) {  
        sueldo = sueldo * (1+(todos[i].antiguedad * 0.1));  
  
        if(sueldo>PISOGAN) retencion = sueldo * IMPGAN;  
        else retencion = 0;  
        sueldo = sueldo - retencion;  
  
        printf("%s debe cobrar: $ %f", todos[i].nombre, sueldo);  
        printf("se le han retenido $ %f", retencion);  
    }  
    else {  
        printf("%s debe cobrar: $ %f", todos[i].nombre, sueldo);  
    }  
}
```

Modificación 4

(se levanta el paro en la fábrica)

"Ya tenemos muchos empleados, se nos complica saber quién está llegando a horario o no. Por lo tanto queremos registrar el horario de entrada y al liquidar el sueldo dar un premio por presentismo.

Cada empleado puede tener un premio distinto porque eso lo arreglamos con cada uno. ¡Esto es sólo para los permanentes!"

Solución 5

```
struct hor_entrada {  
    char dia;  
    char hora;  
    char minuto;  
}
```

```
struct empleado {  
    char nombre[50]; int antiguedad;  
    float basico;  
    char es_permanente;  
    struct hor_entrada entro[31]; //espacio en disco!!  
}
```

Observaciones

- ¿dividimos en dos estructuras, una para empleados temporales y otra para permanentes? ¿quedan en archivos distintos?
- ¿cuántas modificaciones más se le pueden agregar a la lógica del cálculo de sueldo hasta llegar a un momento que sea inmanejable/spaghetti?
- ¿cómo haríamos si fuera necesario modificar algún aspecto de la estructura que afecte a todos los tipos de empleado? Por ejemplo, el básico.
- ¿podríamos contratar a un programador para que nos arme la liquidación de, por ejemplo, "empleados que trabajan a distancia por hora" sin que tenga que tocar (o sea, *entender* y seguro *romper*) el código anterior?

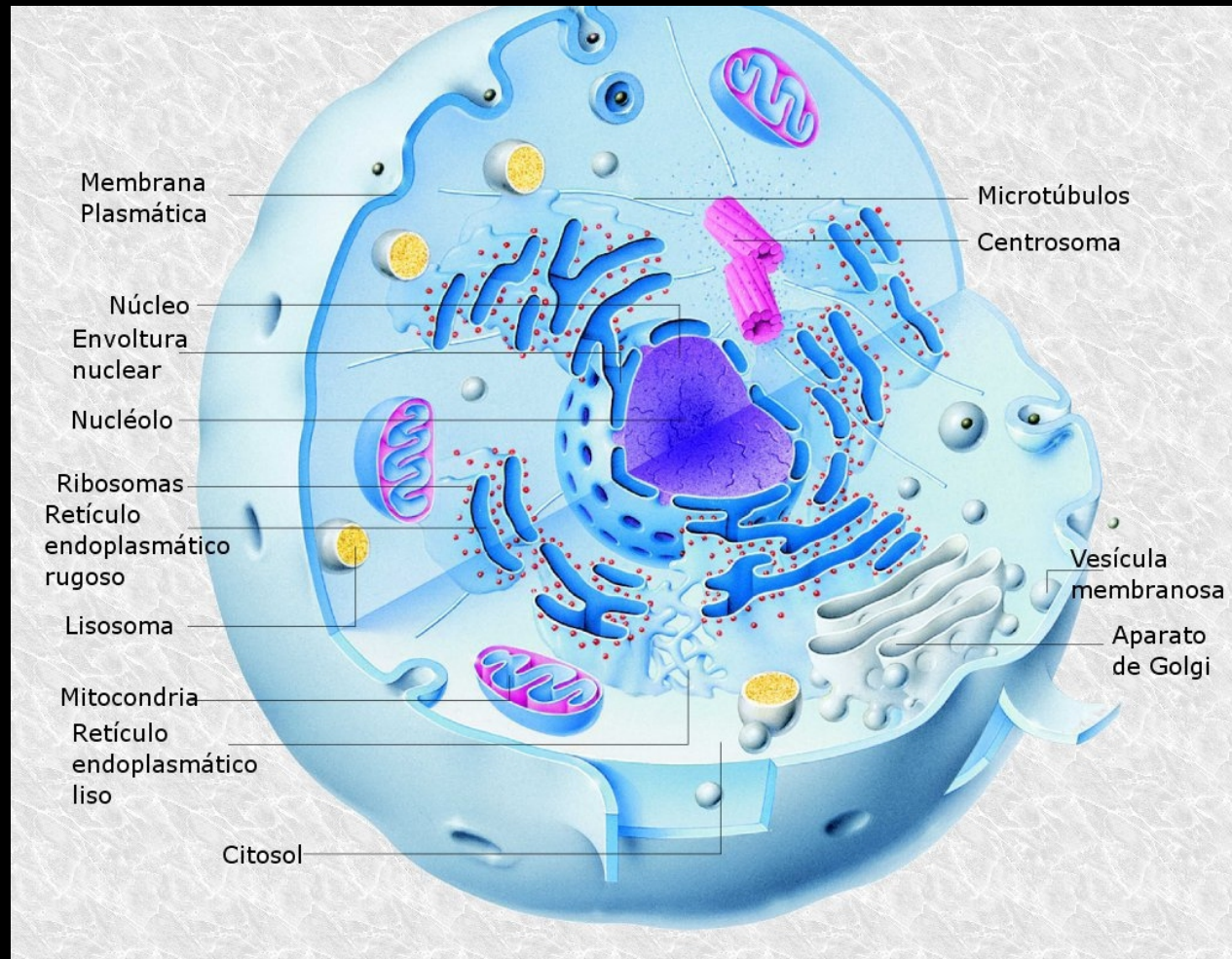
Orientación a objetos

La **P.O.O.** propone pensar el sistema en términos de OBJETOS

Cada OBJETO es una entidad que contiene:
datos Y comportamiento

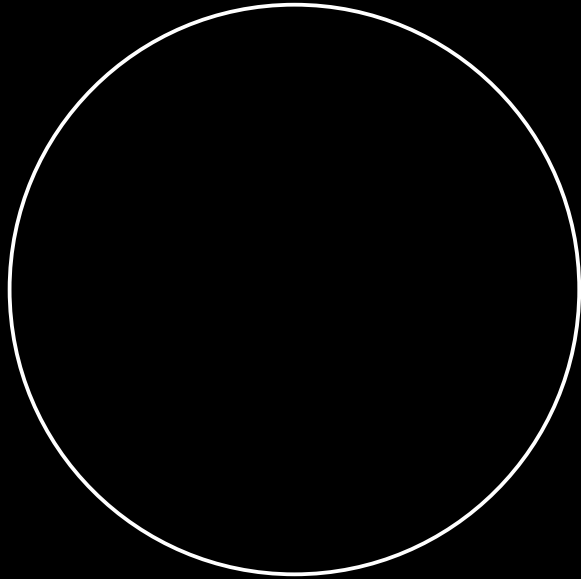
Los OBJETOS nos presentan una INTERFAZ:
- podemos enviarles ciertos mensajes

Los OBJETOS son programados con cierta IMPLEMENTACIÓN:
- la forma interna en cómo funcionan, *invisible* desde afuera

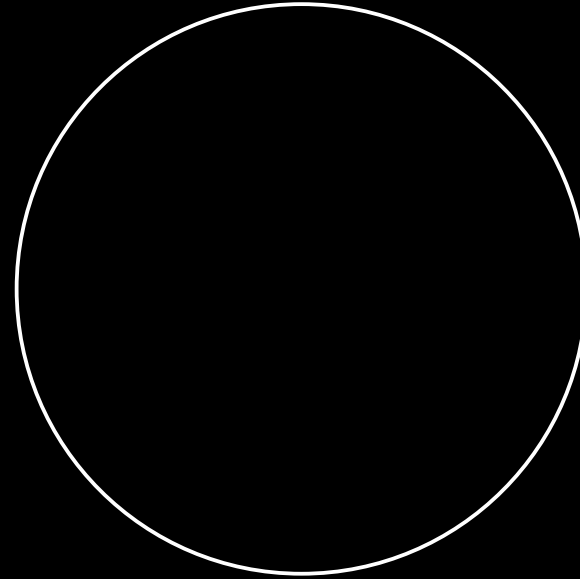


Orientación a objetos

empleado1

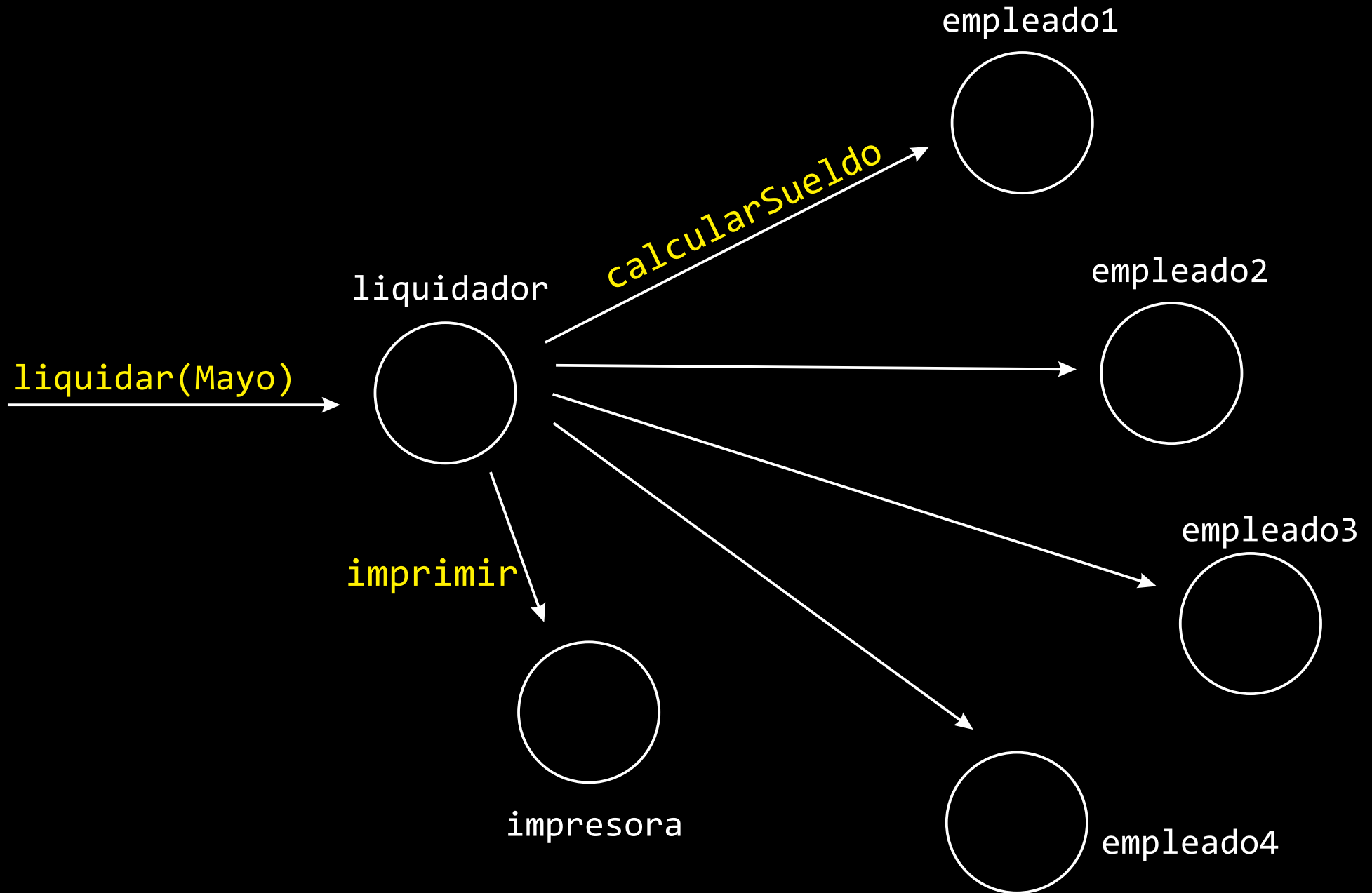


empleado2



decime tu sueldo
para octubre

Sistema diseñado con POO



Responsabilidades

Principio de la POO:

cada objeto debe tener una responsabilidad única
(*Single responsibility principle*)

empleado: saber responder al mensaje `calcularSuelo`

liquidador: saber responder al mensaje `liquidar`

impresora: saber responder al mensaje `imprimir`

Interfaces

Principio de la POO:

programa para una interfaz, no para una implementación

- liquidador no sabe cómo cada empleado calcula su sueldo, sólo sabe pedirlo (conoce la interfaz)
- empleado no sabe para qué le están pidiendo el sueldo, solo cumple su interfaz (responde al mensaje)
- impresora no sabe de dónde vienen los datos

Polimorfismo

Imaginemos una INTERFAZ como un "traje" que se ponen los objetos.

Dada una interfaz llamada "LIQUIDABLE", que consta de este mensaje: `calcularSueldo`

Si tenemos dos tipos de empleados (permanentes y temporales) decimos que son POLIMORFICOS si ambos cumplen la interfaz LIQUIDABLE.

Los dos tipos de objetos tendrán distinta IMPLEMENTACION (porque el cálculo es diferente) pero presentarán la misma INTERFAZ.

Polimorfismo

Podemos decir que todo objeto que cumpla la interfaz LIQUIDABLE (o sea, que sepa responder al mensaje) podrá utilizarse en la liquidación de sueldos.

Lo que es lo mismo: el objeto Liquidador envía mensajes a objetos que cumplen la interfaz LIQUIDABLE y no necesita conocer cómo se realiza el cálculo (o sea, la implementación concreta).

Esa es la ganancia de la P.O.O : la división de *responsabilidades*

Polimorfismo

Otro ejemplo: los AUTOS

Tienen una interfaz: todos se manejan igual (volante, pedales)

Tienen una implementación: lo veo abriendo el capot

Para un conductor, los autos son POLIMORFICOS
respecto del manejo: puedo manejar todo lo "manejeable".

Para un mecánico cada uno tiene mil detalles (el programador)

Encapsulamiento

Decimos que un objeto ENCAPSULA su comportamiento y estado, es decir, desde afuera sólo podemos ver la INTERFAZ.

No se puede modificar el estado interno del objeto, sólo podemos enviarle mensajes para solicitarle eso.

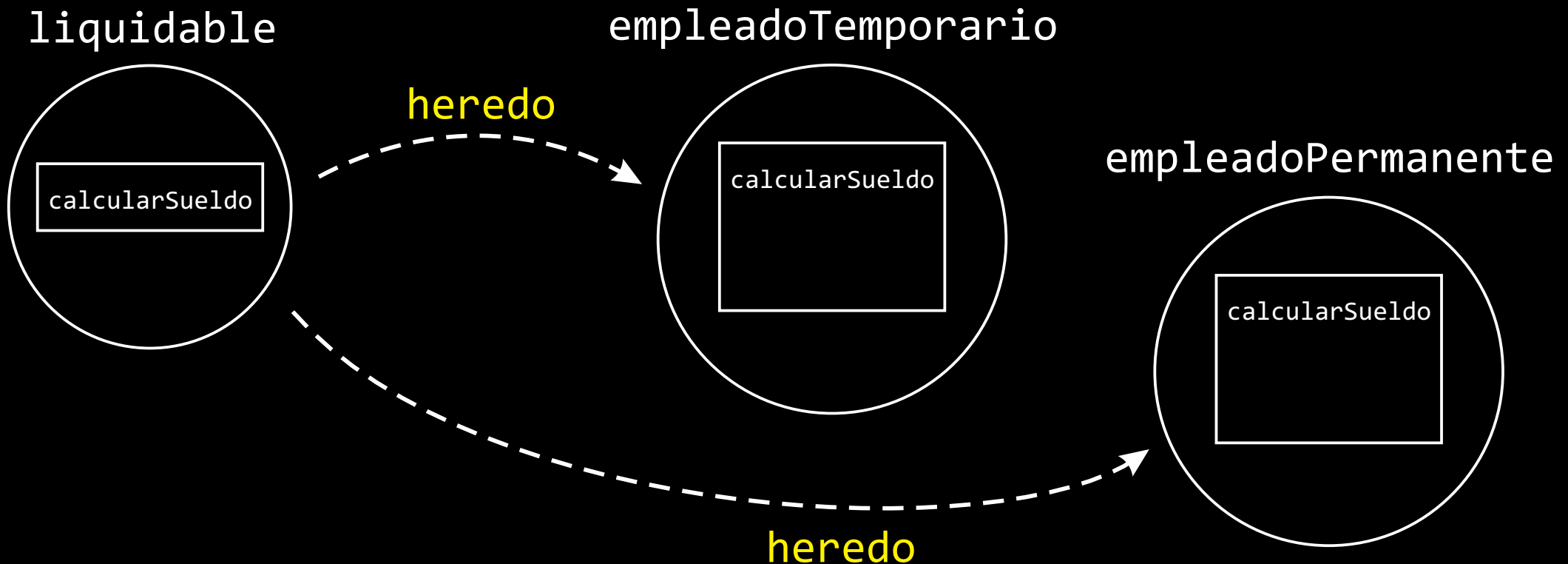
Si mantenemos este ocultamiento podremos disfrutar de los beneficios de la POO.

Herencia

¿Cómo forzar a que los objetos respondan a los MISMOS mensajes?

Los objetos se pueden crear a partir de otros objetos

Los objetos "hijos" HEREDAN la interfaz. Están "obligados" a responder.



Diseño P.O.O.

Forma de pensar el diseño de un sistema de información en P.O.O.:

- el `Liquidador` enviará mensajes a objetos que cumplan la interfaz `Liquidable` (polimorfismo)
- los empleados deben ser objetos `Liquidable` (herencia)
- se pueden agregar luego nuevos tipos de empleado, siempre que cumplan la interfaz, y no es necesario modificar al `Liquidador`
- los [datos + comportamiento] están ahora más DESACOPLADOS entre sí

Clases

Para reducir el trabajo se inventaron las CLASES.

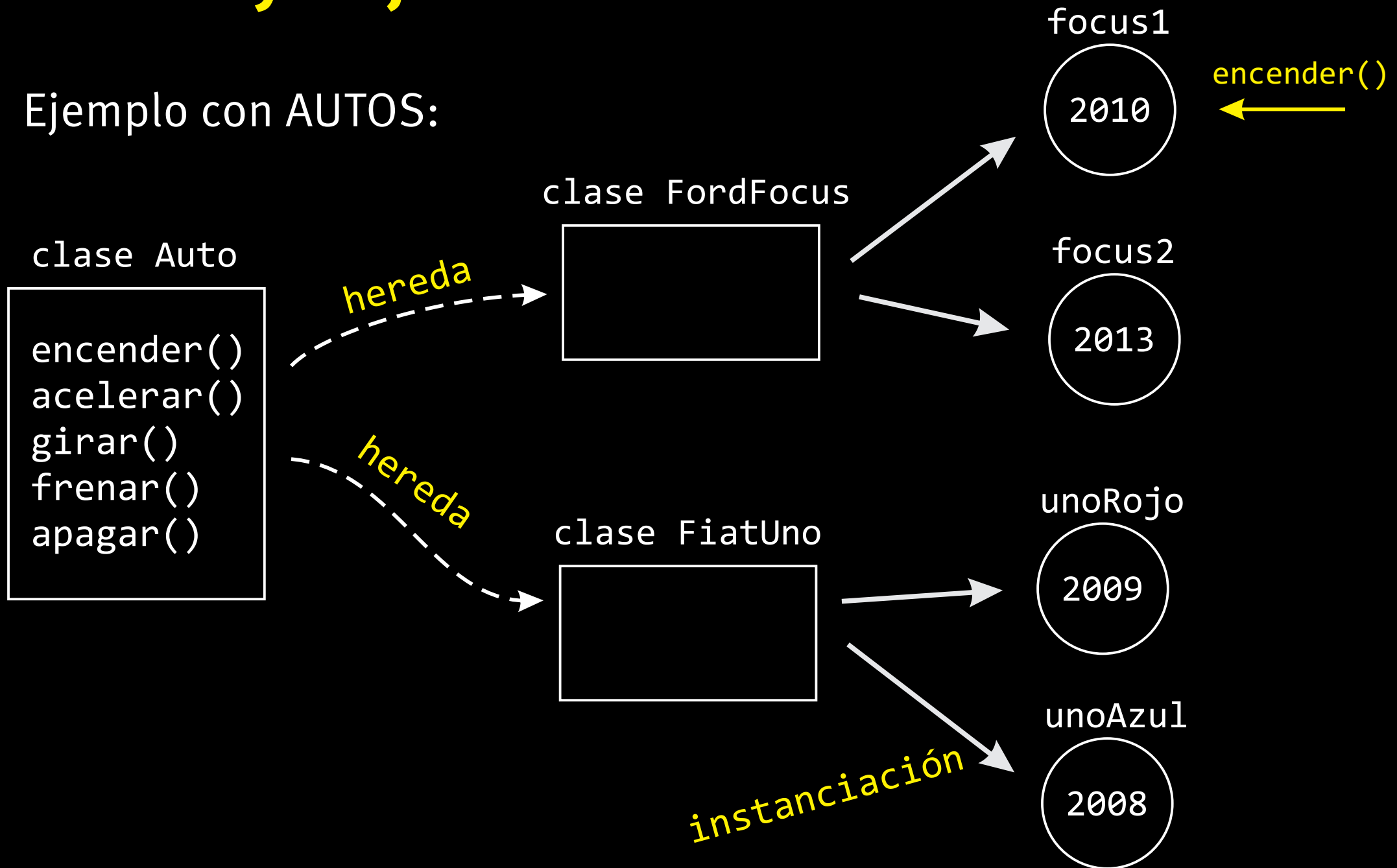
Las clases son como MOLDES con los cuales fabricamos OBJETOS.
Serán todos iguales y nos ahorrarán tiempo de construcción.
(similar a las struct)

Los OBJETOS de la misma CLASE tienen la misma INTERFAZ, ya que fueron instanciados desde el mismo "molde".

Podemos construir moldes a partir de otros moldes (herencia).
El sistema trabaja con los OBJETOS fabricados (instancias).

Clases y objetos

Ejemplo con AUTOS:



¡Bienvenidos a C++ !

C# Smalltalk PHP Javascript Python Ruby Java

Simula Delphi Actionscript Eiffel VB.net Objective-C



Empirical studies indicate that 20% of the people drink 80% of the beer. With C++ developers, the rule is that 80% of the developers understand at most 20% of the language. It is not the same 20% for different people, so don't count on them to understand each other's code.

```
class Auto {  
    private:  
        int velocidad;
```

miembros privados (atributos)

```
public:  
    void inicializar() {  
        velocidad = 0;  
    }  
  
    void acelerar(int cuanto) {  
        velocidad = velocidad + cuanto;  
    }  
  
    void frenar(int cuanto){  
        velocidad = velocidad - cuanto;  
        if(velocidad<0) velocidad=0;  
    }  
  
    int obtenerVelocidad() {  
        return velocidad;  
    }  
  
};
```

miembros públicos (métodos)

Accesibilidad

Miembros private

Generalmente variables

No son visibles en la interfaz (desde afuera)

No se heredan

Miembros public

Generalmente funciones (métodos)

Forman parte de la interfaz (los puedo llamar desde afuera)

Se heredan y siguen siendo públicos

```
//Bienvenidos a C++
```

```
int main(void) {
```

```
    Auto * a1 = new Auto;
```

```
    a1->inicializar();
```

```
    a1->acelerar(10);
```

```
    a1->acelerar(3);
```

```
    a1->velocidad = 100;    // no!!
```

```
    printf("velocidad auto: %d", a1->obtenerVelocidad() );
```

```
    return 0;
```

```
}
```


Conceptos clave

Objeto

Clase

Miembro

Mensaje

Interfaz

Implementación

Encapsulamiento

Accesibilidad

Responsabilidad

Herencia



Ejemplo de herencia (avanzado)



Mecanismo de herencia

```
class EmpleadoPermanente {  
    private:  
        int basico;  
        int antiguedad;  
  
    public:  
        float calcularSueldo() {  
            return basico * (1+(antiguedad*0.1));  
        }  
  
        void cargar(float bas, int antig) {  
            basico=bas;  
            antiguedad=antig;  
        }  
};
```

Mecanismo de herencia

```
class EmpleadoTemporal {  
    private:  
        int basico;  
  
    public:  
        float calcularSueldo() {  
            return basico;  
        }  
  
        void cargar(float bas) {  
            basico=bas;  
        }  
};
```

```
class Empleado {  
    protected:   
        float basico;  
  
    public:  
        virtual float calcularSueldo()=0;   
};
```

```
class EmpleadoPermanente : public Empleado {   
    private:  
        int antiguedad;  
  
    public:  
        float calcularSueldo() {   
            return basico * (1 + (antiguedad * 0.1) );  
        }  
  
        void cargar(float bas, int antig) {  
            basico=bas;  
            antiguedad=antig;  
        }  
};
```

Clases abstractas

- A toda clase con al menos 1 método virtual le decimos CLASE ABSTRACTA
- No se pueden crear INSTANCIAS de clases abstractas
- Las clases abstractas sirven sólo para heredar de ellas (para hacer las clases concretas)
- Las subclases serán las encargadas de IMPLEMENTAR el método virtual

PUBLIC

PRIVATE

PROTECTED

Visibles desde
afuera (interfaz)

sí

no

no

Visibles en
las subclases
(herencia)

sí

no

sí

```
class cA {
    int a;

    public:
        int b;

    protected:
        int c;
};

class cB : public cA {
    int d;

    protected:
        int f;

    public:
        int e;

        void pru() { c=10; /*si*/ }

        void pru2() { a=10; /*no*/ }
};
```

```
cA * obj1 = new cA;
cB * obj2 = new cB;
```

```
obj1->a = 5; //no
obj1->b = 2; //si
obj1->c = 3; //no
```

```
obj2->d = 4; //no
obj2->e = 8; //si
obj2->f = 1; //no
```

```
obj2->b = 6; //si!!
```


Ejercicios

Dada la clase virtual Empleado y la clase EmpleadoPermanente que hereda de la anterior (página 34):

1) Cree la clase EmpleadoTemporario

2) Implemente sus métodos públicos:

```
float calcularSueldo(void)
void cargar(float)
```

3) Realice un programa main que instancie y cargue dos empleados de cada clase. Muestre los sueldos que cobran.