

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.handle.send(dat)
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

# DAM/DAW

# PROGRAMACIÓN

## 03.1

# Introducción a la OOP

*(Object Oriented Programming)*

# Indice

- Abstracción
- Todo es un objeto
- Clases y objetos
- Principios de la OOP
- Creando objetos
- (des)Ventajas de la OOP

```

import threading, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sock.connect((addr, port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
            return
    def listen(self, host, port, func):
        try:
            self.sock.bind((host, port))
            self.sock.listen(2)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
            return
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sock.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            if not dat:
                self.handle.close()
                self.sock.close()
                self.flag=0
                return
            self.close(self.data)
            self.handle.send(data)
            self.close(self)

```

### Abstracción (I)

---

- El desarrollo de cualquier programa informático supone un cierto grado de abstracción. En primer lugar, del problema a resolver (“*espacio del problema*”) y, en segundo lugar, de las particularidades *hardware/software* de la implementación (“*espacio de la solución*”)
- Con el tiempo, las metodologías y los lenguajes de programación han ido cambiando para “manejar” el incremento progresivo de la **complejidad** de los programas. Paralelamente, esa evolución ha reducido de forma considerable el esfuerzo de “mapeado” de la solución computacional desde el “espacio del problema” al “espacio de solución”
- Por ejemplo, la aparición del lenguaje ensamblador, supuso una pequeña abstracción del hardware subyacente. Ya no era necesario introducir programas escritos directamente en código máquina, sino que se utilizan **representaciones simbólicas** de dichas instrucciones máquina.

### Abstracción (II)

---

- La aparición de lenguajes “**imperativos**” (FORTRAN, BASIC,...) supusieron, a su vez, una abstracción sobre el lenguaje ensamblador. Aún así, siguen requiriendo pensar más en base a la estructura de la computadora que en base a la estructura del problema
- En los años 60 aparece la **programación estructurada**. Lenguajes como C o Pascal, con soporte para subrutinas y ricas estructuras de control, facilitan el desarrollo de programas complejos. Aún así, las dimensiones de los proyectos software actuales, desde sistemas operativos a complejos sistemas de gestión empresarial, hacen que sean difíciles de escribir y mantener. Esto da lugar a la aparición de nuevas metodologías que faciliten el desarrollo de estos proyectos de gran escala.
- En los años 70 aparece **Smalltalk**. Basado en **Simula**, que introdujo los conceptos de clase, objeto, herencia,... suponen los inicios de la OOP

### Abstracción (y III)

- Los lenguajes orientados a objetos proporcionan al programador herramientas para describir la solución desde el propio “espacio del problema”, lo cual supone un nuevo y mayor grado de abstracción del lenguaje. Básicamente, nos permiten describir el problema en términos del propio problema, en lugar de en términos de la computadora donde finalmente se ejecutará la solución.
- Nos referiremos a los elementos del espacio del problema y su representación en el espacio de la solución como “objetos”
- A diferencia de la programación procedimental, organizada en torno al código, la OOP se organiza en torno a los datos. Son los propios datos los que “controlan el acceso al código”. En estos lenguajes, definiremos “estructuras” que no sólo nos permitirán representar los datos, sino definir las rutinas que actuarán sobre esos datos.

# Todo es un objeto (I)

- Alan Kay, creador del lenguaje Smalltalk, estableció las siguientes características básicas de su lenguaje y, por extensión, de lo que sería una aproximación orientada a objetos a la programación:
  - **Todo es un objeto.** Piensa en el objeto como una variable un tanto “especial”. Almacena datos pero también puedes “pedirle” a ese objeto que realice operaciones sobre sí mismo, sobre los datos que almacena. En teoría, cualquier componente conceptual del problema a resolver (clientes, servicios, aviones, espectáculos, entradas,... ), puede ser representado como un objeto en el programa
  - **Un programa no es más que un puñado de objetos diciéndole unos a otros qué hacer mediante el envío de mensajes.** Para hacer una petición a un objeto, le “envíamos un mensaje”. En la práctica, equivale a invocar un método que pertenece a un objeto particular.

# Todo es un objeto (II)

- **Cada objeto está formado, a su vez, por otros objetos (composición).**  
O, visto de otro modo, creamos nuevos objetos empaquetando objetos ya existentes (y sus funcionalidades). De este modo, podemos abordar problemas complejos al mismo tiempo que dicha complejidad queda oculta bajo la simplicidad del empleo de los objetos.
- **Cada objeto tiene un tipo.** Usando la terminología de la OOP con propiedad, diremos que **un objeto es una instancia de una clase**, donde “clase” es un sinónimo de “tipo”. Piensa en la “clase” como, p.e., los planos de un barco, diseñados por un ingeniero y que sólo existe sobre papel. Los “objetos” serán los barcos reales contruidos a partir de dichos planos (**relación uno-a-muchos**), cada uno “*independiente*” de los otros y que, aunque pueden compartir “*valores*” comunes (manga, eslora, ...), pueden variar otros (nombre, matrícula, color, bandera,...)



## Todo es un objeto (y III)

- Todos los objetos de un tipo particular pueden recibir los mismo mensajes. Al estar creados en base a la misma plantilla (o “clase”), todos los objetos del mismo tipo (o “clase”) responden a los mismo mensajes. Esto es así ya que es en la propia “clase” donde se define qué mensajes pueden recibir “sus” objetos. De igual modo, debido a las relaciones de “herencia”, un objeto podrá recibir los mismo mensajes que podría recibir su “padre”. Por ejemplo, un objeto de tipo “círculo” es también un objeto del tipo “forma geométrica”, por lo que podría aceptar también sus mismos mensajes.
- En resumen, *un objeto tiene un estado, un comportamiento y una identidad*

Es decir, un objeto tiene datos internos (que definen su estado), métodos (que definen su comportamiento), y cada objeto es distinguible de los otros objetos (por ejemplo, mediante una dirección de memoria única)



# Clases y Objetos (I)

- Fue **Aristóteles**, en el siglo IV A.C., el primero en introducir formalmente los conceptos de “tipo” o “**clase**” en su “*Historia de los Animales*” al hablar de “clases de peces” o “clases de animales” en la creencia de que, aún siendo seres únicos, cada animal es a su vez miembro de una “clase” que le confiere **características** y **comportamientos** similares a los de sus congéneres.
- Esa idea fundamental fue empleada en el primer lenguaje orientado a objetos, **Simula-67**, al introducir la palabra-clave **class**, para definir nuevos tipos abstractos en un programa.
- La clase, al describir un conjunto de objetos con idénticas características y comportamientos, actúa de forma equivalente a como lo hacen los tipos primitivos. Con la salvedad de que podemos definir nuestros propios “tipos” para “modelar” la naturaleza del problema

## Clases y Objetos (II)

- Una vez definida una clase, podremos crear tantos objetos de ella como queramos. Estos objetos se denominan *instancias* de la clase en cuestión.
- A la hora de definir una clase, deberemos indicar:
  - **Atributos** o “variables miembro”. Nos permiten establecer las características comunes de los objetos de la clase. El conjunto de los **valores** de dichos atributos de un objeto, es lo que conforma su **estado** en un **instante** determinado
  - **Métodos** o “funciones miembro”. Definen el “comportamiento” de los objetos de la clase. A través de las llamadas a dichos **métodos**, “interactuamos” con el objeto y actuamos sobre los datos que almacenan. Definen lo que se denomina *interfaz* de la clase, es decir, el conjunto de funciones que “expone” para poder “interactuar” con sus objetos.

# Clases y Objetos (y III)

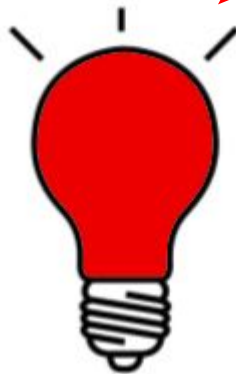
## CLASE

Light
-color -power -brightness
on() off() brighten() dim()

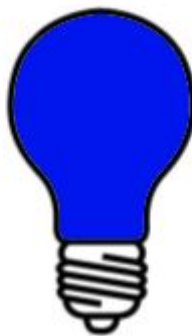


```
class Light {  
    ...  
    (definición de la clase)  
    ...  
}
```

## OBJETOS



color: red  
power: 80w



color: blue  
power: 60w

(creación de objetos)

```
Light red_light;  
Light blue_light;  
  
red_light = new Light('red', 80);  
blue_light = new Light('blue', 60);  
  
red_light.on();  
blue_light.off();
```

los **objetos** `red_light` y `blue_light`  
son **instancias** de la **clase** `Light`

# Principios de la OOP (I)

Para soportar los principios de la OOP, todos sus lenguajes tienen tres “rasgos” en común: **encapsulación**, **polimorfismo** y **herencia**

- **Encapsulación**

- Mecanismo por el cual **agrupamos** tanto los datos como el código que lo manipula, regulando el **acceso** a dichos datos a través de los “canales” adecuados (**interfaz**).
- En OOP, esta encapsulación tiene su expresión en los **objetos**. Construidos a partir de una **clase** a modo de “plantilla”, actúan como **cajas negras** que agrupan parte de los datos del problema junto con el código que actúa sobre dichos datos.
- La encapsulación nos permite **ocultar** tanto la **estructura interna** como la **implementación** (código) de los objetos

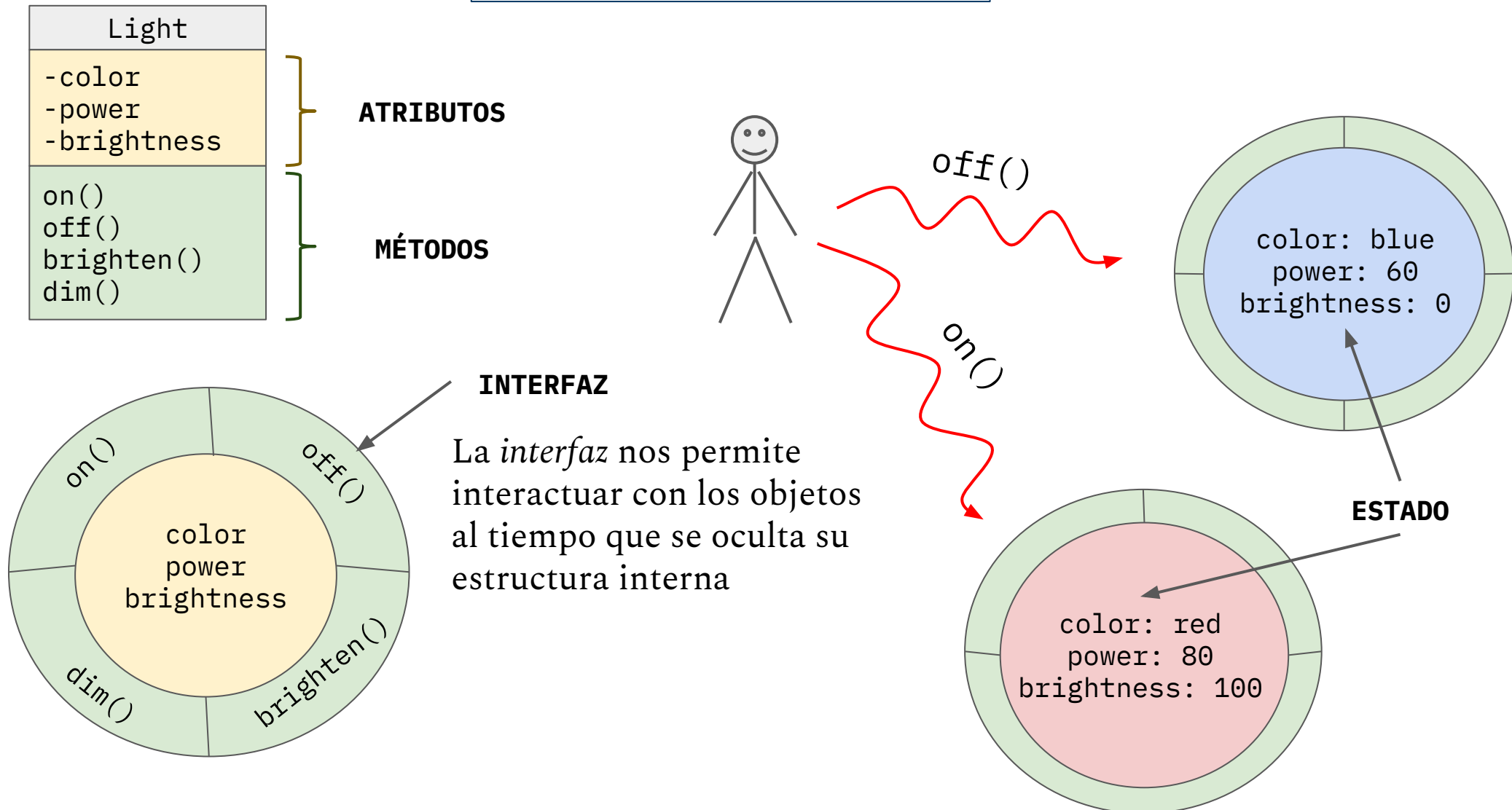
# Principios de la OOP (II)

---

- Idealmente, los atributos de un objeto deben encontrarse **ocultos** al resto de los objetos, es decir, no se va a poder acceder directamente a los atributos de un objeto para modificar su estado o consultarlo.
- Para acceder a los atributos de un objeto se deben utilizar **métodos**. Es decir, los métodos exponen toda la funcionalidad del objeto, mientras que los detalles del estado interno del objeto permanecen ocultos. Incluso algunos métodos auxiliares, relevantes **sólo** para el “funcionamiento interno” del objeto, deben permanecer también ocultos.
- Denominamos **interfaz** al conjunto de métodos públicos de la clase a través de los cuales sus objetos exponen su funcionalidad

# Principios de la OOP (III)

## ENCAPSULACIÓN



# Principios de la OOP (IV)

---

### ● Herencia

- La **herencia** es el proceso por el cual un objeto puede adquirir (“heredar”) las propiedades de otro objeto
- La herencia se implementa mediante una **clasificación jerárquica**, es decir, de **padre-a-hijo**. De este modo, las clases “hijas” heredan todas las características y comportamientos de su/s “padre/s”
- Las clases hijas pueden añadir sus propios métodos y atributos e, incluso, modificar los métodos heredados de sus clases padre. De este modo, podemos entender la herencia como una **especialización** sucesiva de las clases
- La herencia proporciona un mecanismo conveniente para la **reutilización** de código. Las clases padre agruparán aquellos atributos y comportamiento genéricos de sus subclases.

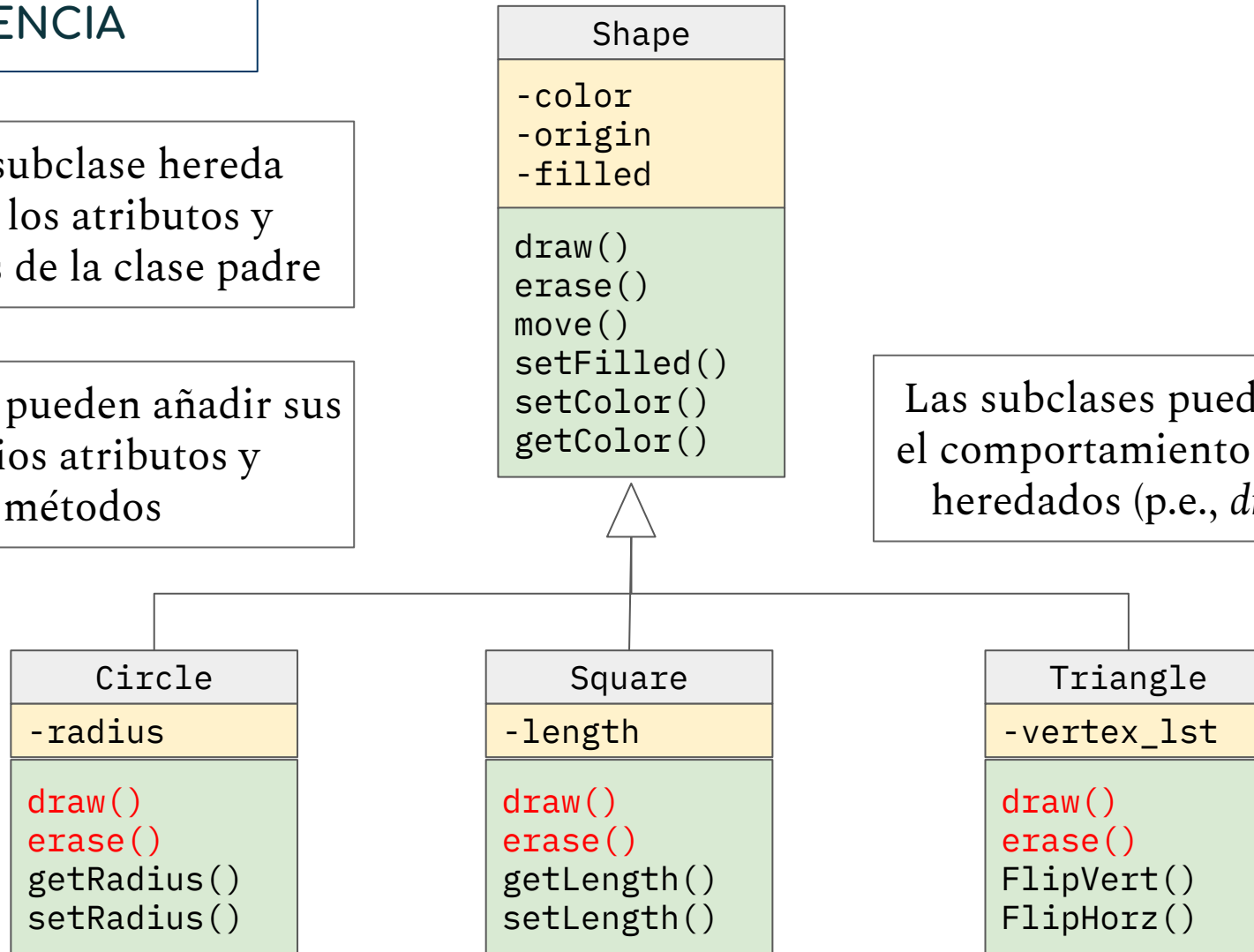


# Principios de la OOP (VI)

## HERENCIA

Cada subclase hereda todos los atributos y métodos de la clase padre

Además, pueden añadir sus propios atributos y métodos



Las subclases pueden “reescribir” el comportamiento de los métodos heredados (p.e., `draw()`, `erase()`)

# Principios de la OOP (VII)







---

## ● Polimorfismo

- En ocasiones, al trabajar con jerarquías, necesitamos tratar a un objeto de acuerdo a su *tipo base* en lugar de hacerlo de acuerdo a su *tipo específico*. Esto nos permite escribir código que no depende de las subclases específicas.
- Siguiendo con el ejemplo anterior de formas geométricas, podríamos diseñar un código que dibujara las figuras contenidas en una colección, con independencia de la subclase específica, invocando el método *draw()* de la superclase *Shape()*. Esa llamada será “redirigida” en cada caso, al método del objeto correspondiente (**polimorfismo**)
- Esto implica que no tenemos que preocuparnos de las particularidades de cada subclase. Incluso podrían añadirse nuevas subclases y nuestro código no se vería afectado.

## Principios de la OOP (VIII)

### POLIMORFISMO

```
lista_de_figuras = [  ,  ,  ,  ,  ,  ]
```

para cada `[objeto:Shape]` de `lista_de_figuras`:

hacer `[objeto:Shape].draw()`

Aunque invocamos el método ***draw()*** de la clase base **Shape**, en cada caso se ejecutará, de forma transparente para nosotros, el método correspondiente de cada objeto (**Circle.draw()**, **Square.draw()**, ...)

Aunque con posteridad se crearan y se añadieran a la colección nuevas subclases de figuras aún no contempladas, nuestro código seguiría funcionando sin problema

# Principios de la OOP (y IX)

---

- El hecho de que de antemano no sepamos el método de **qué objeto** se va a ejecutar, implica que el compilador no puede realizar las llamadas a funciones de modo tradicional.
- Los compiladores no-OOP resuelven las llamadas a las funciones mediante lo que se conoce como **early-binding** (enlace temprano). Ya que se conoce la función que se quiere ejecutar, el sistema “enlaza” la llamada con la dirección del código de dicha función
- Para dar soporte al polimorfismo, los lenguajes OOP implementan lo que se denomina **late-binding** o **dynamic-binding**. Cuando se envía un mensaje a un objeto (*function call*), el código que se va a ejecutar no se determina hasta tiempo de ejecución. El compilador puede hacer el análisis de tipado de los argumentos y retorno, pero desconoce el código exacto que se va a ejecutar. Dependiendo del lenguaje, debemos indicarlo o no explícitamente (C++ **virtual**)

## Creando objetos (I)

- Un aspecto crítico del trabajo con objetos es el modo en que son creados y destruidos. Cada objeto creado precisa de **recursos**, principalmente **memoria**. Una vez que ese objeto deja ser necesitado, esos recursos deben ser liberados para poder ser reutilizados.
- En lenguajes como C++, donde se trata de maximizar la **eficiencia**, los objetos pueden ser creados en la propia pila del programa o en la sección *static* de igual modo a cualquier otra variable, agilizando su acceso. Sin embargo, para objetos dinámicos, el programador debe realizar la liberación de recursos una vez ya no se necesiten.
- En lenguajes como Java, donde la gestión de la memoria es **dinámica**, la JVM asigna un espacio de memoria predeterminado a la aplicación (**heap**) donde se crean los objetos y proporciona un **Garbage Collector** que se encarga de liberar los recursos de objetos en desuso. Penaliza la eficiencia pero simplifica el desarrollo y previene errores.

### Creando objetos (y II)

- Una vez definida la clase, es decir, su estructura interna y su interfaz, podremos crear objetos de la misma. Los lenguajes OOP nos proporcionarán operadores para invocar la **construcción** y **destrucción** de los nuevos objetos (**new** en C++/Java, **delete** en C++, **del** Python)
- En el momento de crear el objeto, se invocará un método “especial” de la clase, el **constructor**. Este método es el encargado de inicializar la estructura interna del nuevo objeto. Habitualmente, nuestras clases dispondrán de varios constructores, con diferentes listas de argumentos (**sobrecarga** del constructor) para permitir diferentes inicializaciones.
- De igual modo, dispondremos de otro método especial, el **destructor**, que se invocará al destruir el objeto y nos permitirá la liberación de ciertos recursos (ficheros, red,...). En lenguajes dinámicos (Java, Python) que usan GC, no podemos saber de antemano cuándo se ejecutará.

# (des)Ventajas de la OOP

---

- Algunas **ventajas** serían:
  - **Reutilización** de código (herencia, encapsulación,...)
  - Facilita la **portabilidad** (abstracción, capas de *middleware*)
  - Facilita el **mantenimiento** (interfaces, encapsulación)
- Como **desventajas**, se podría señalar:
  - Nueva metodología, larga curva de **aprendizaje**
  - Un objeto no se diseña aisladamente, sino que depende de las relaciones con otros. Nuevas relaciones pueden provocar cambios en la definición. Necesidad de un proceso iterativo. La **experiencia** es un factor determinante en la realización de buenos diseños.