

```

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.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

06.2

OOP: Interfaces en Java

Indice

- Introducción
- Declaración
- Implementación
- Interfaces y polimorfismo
- Interfaces y herencia
- Interfaces vs Clases abstractas

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket.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"
    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"
    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 (self.handle.send(data)):
                self.handle.send(data)
        self.close(self)
        self.sock.close()
```

Introducción (I)

- En OOP es habitual el tener que establecer “qué” debe hacer una clase pero no “cómo” hacerlo
- De hecho, acabamos de ver un ejemplo de ello: los *métodos abstractos*.
- En esencia, estos métodos abstractos especifican un *interfaz* del método, pero no su implementación. Son las *subclases* de la clase abstracta las que se encargaran de proporcionar, cada una, su propia implementación
- Pero ¿que pasa cuando identificamos un conjunto de comportamientos comunes en grupos de clases pertenecientes a diferentes jerarquías?
- Supongamos, por ejemplo, que recibimos información en tiempo real de una serie de estaciones meteorológicas. A partir de dicha información deseamos: *mostrarla* (app, web,...), *generar estadísticas* relativas al día en curso (max, mín, promedio,...), *realizar predicciones* para las próximas horas o días... Clases funcionalmente diferentes pero con algo *común*, deben *actualizarse* con cada nueva medida realizada (patrón *Observer*)

Introducción (y II)

- Java, proporciona la palabra clave *interface* con objeto de poder definir grupos de métodos que podrán ser implementados por diferentes clases
- Un *interfaz* es sintácticamente similar a una clase abstracta, en el sentido de que definirá uno o más métodos sin cuerpo (desde JDK8, el *interface* puede definir una *implementación por defecto* de los métodos)
- Una vez que un *interface* ha sido definido, puede ser implementado por una o más clases. Para ello, incluirán la cláusula *implements* en la declaración de la clase, indicando el nombre del *interfaz* que implementa.
- Una clase podrá implementar simultáneamente más de un *interfaz*. A continuación de *implements*, se indicarán separados por comas, todos los *interfaces* implementados por la clase. Es una forma de soportar cierto grado de *herencia múltiple*
- Una clase está obligada a implementar *todos* los métodos de *sus interfaces*

Declaración (I)

- Declaremos un *interface* empleando la siguiente sintaxis general:

```
[public] interface NombreInterfaz [extends InterfazPadre] {  
    tipo VAR1 = valor1;  
    tipo VAR2 = valor2;  
    ...  
    [private*] tipo nombreMetodo1(lista_de_params);  
    [private*] tipo nombreMetodo2(lista_de_params);  
}
```

- Un *interface* podrá especificar acceso *public* o *por defecto* (si se omite). Si se declara *public*, debe estar en un fichero con el mismo nombre
- Puede declarar variables, que serán *public*, *final* y *static* de forma implícita (es decir, *constantes*) y deben ser *inicializadas*
- Todos los métodos serán *public* de forma implícita (* se pueden declarar métodos *private* desde JDK9). Desde JDK8 se pueden incluir métodos con cuerpo *por defecto* y métodos *estáticos* (que no pueden ser sobrescritos)

Declaración (y II)

❖ Ejemplo:

```
public interface Producto {
```

```
    double IVA_G = 0.21;  
    double IVA_R = 0.10;  
    double IVA_SR = 0.04;
```

Cualquier variable del *interface* será implícitamente *public*, *static*, *final*. Por tanto, **constante**. Debe ser inicializada en la declaración

```
    double getPrecio();  
    String getNombre();
```

Métodos que deben ser **sobreescribir** por las clases que implementen este *interface*. Además, deben ser declarados en la clase como **public**

```
    public static double getTotal(Producto[] lista) {  
        double sum = 0.0;  
        for(Producto p: lista) sum += p.getPrecio();  
        return sum;  
    }
```

Método estático
Se invoca directamente desde el *interface*

```
}
```

Implementación (I)

- Una vez el interface ha sido declarado, podrá ser implementado por una o más clases. Para ello, la clase incluirá la cláusula **implements** seguida por el nombre del *interface* (pueden ser varios separados por comas):

```
class Nombre [extends Superclase] implements Interface1[, Interface2,...] {  
    // Cuerpo de la clase  
}
```

- La clase **deberá implementar todos** los **métodos abstractos** del *interface* (sin cuerpo) y **deberá establecer** para ellos el modo de acceso **public**
- No está obligada a sobrescribir** los **métodos *por defecto*** (con cuerpo) del *interface*. En dicho caso, utilizará la implementación del propio *interface*
- No tiene acceso** a los **métodos *private*** del *interface*. Son de uso interno de los métodos con implementaciones por defecto o *private* del *interface*
- No “hereda” ni puede sobrescribir** los **métodos *static*** del *interface*. Estos sólo se pueden invocar haciendo uso del propio *interface*

Implementación (II)

❖ Ejemplo (clase Libro):

```
public class Libro implements Producto {  
    private final double precio;  
    private final String titulo;  
    private final int numpag;  
  
    public Libro(String titulo, double precio, int numpag) {  
        this.titulo = titulo;  
        this.precio = precio;  
        this.numpag = numpag;  
    }  
  
    @Override  
    public double getPrecio() { return this.precio; }  
    @Override  
    public String getNombre() { return this.titulo; }  
    public int getNumpag() { return this.numpag; }  
}
```

La clase Libro implementa el *interface* Producto

Métodos del *interface* que deben ser implementados por la clase

Implementación (III)

❖ Ejemplo (clase Pelicula):

```
public class Pelicula implements Producto {
```

La clase Libro implementa el *interface* Producto

```
    private final double precio;
```

```
    private final String titulo;
```

```
    private final int durac;
```

```
    public Pelicula(String titulo, double precio, int durac) {
```

```
        this.titulo = titulo;
```

```
        this.precio = precio;
```

```
        this.durac = durac;
```

```
    }
```

```
    @Override
```

```
    public double getPrecio() { return this.precio; }
```

```
    @Override
```

```
    public String getNombre() { return this.titulo; }
```

```
    public int getDuracion() { return this.durac; }
```

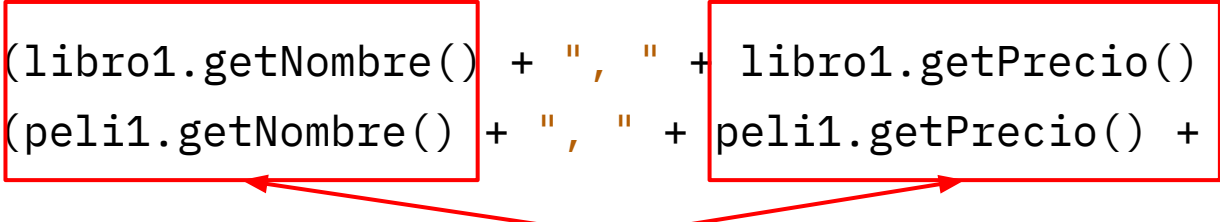
```
}
```

Métodos del *interface* que deben ser implementados por la clase

Implementación (y IV)

❖ Ejemplo (clase de prueba Tienda):

```
public class Tienda {  
    public static void main(String[] args) {  
        Libro libro1 = new Libro("Crimen y Castigo", 10.40, 752);  
        Pelicula peli1 = new Pelicula("Matrix", 9.99, 150);  
  
        System.out.println(libro1.getNombre() + ", " + libro1.getPrecio() + "€");  
        System.out.println(peli1.getNombre() + ", " + peli1.getPrecio() + "€");  
    }  
}
```



Métodos declarados en el *interface* e implementados por las clases

❖ Salida:

```
Crimen y Castigo, 10.4€  
Matrix, 9.99€
```

Interfaces y Polimorfismo (I)

- Java nos permite crear **variables** referencia cuyo **tipo** sea un *interface*
- Una variable de este tipo podrá referenciar a **cualquier** objeto que **implemente** dicho *interface*
- Al invocar un método de un objeto a través de una referencia de tipo *interface*, se ejecutará la versión implementada por el objeto referenciado
- Este mecanismo consistente en enlazar el código el nombre del método con una determinada implementación en tiempo de ejecución se conoce como *late binding* o *dynamic binding*
- Este uso de los *interfaces* es otra de las formas que tiene Java de dotar al lenguaje de **polimorfismo**. El proceso es similar al empleo de variables del tipo de la *superclase* para “mapear” el acceso a las implementaciones sobreescritas de sus métodos a través de los objetos de sus *subclases*

Interfaces y Polimorfismo (II)

❖ Ejemplo (clase de prueba Tienda):

```
public class Tienda {  
    public static void main(String[] args) {  
        Libro libro1 = new Libro("Crimen y Castigo", 10.40, 752);  
        Pelicula peli1 = new Pelicula("Matrix", 9.99, 150);  
  
        Producto p;  
        p = libro1;  
        System.out.println(p.getNombre() + ", " + p.getPrecio() + "€");  
        p = peli1;  
        System.out.println(p.getNombre() + ", " + p.getPrecio() + "€");  
    }  
}
```

Se invocará de forma dinámica la implementación del método correspondiente al objeto referenciado

❖ Salida:

```
Crimen y Castigo, 10.4€  
Matrix, 9.99€
```

Interfaces y Polimorfismo (III)

- Podemos ver otro ejemplo de polimorfismo en la implementación del método estático *getTotal()* de nuestro *interface Producto*
- Recordemos...

```
public static double getTotal(Producto[] lista) {  
    double sum = 0.0;  
    for(Producto p: lista) sum += p.getPrecio();  
    return sum;  
}
```

- El método anterior define como parámetro un *array* de referencias al *interface* *Producto*. Por tanto, podría contener referencias a objetos de cualquier clase que implemente dicho *interface*
- Vamos a verlo con un ejemplo. Recuerda que, al tratarse de un método estático, sólo puede invocarse empleando el nombre del *interface*

Interfaces y Polimorfismo (y IV)

❖ Ejemplo (clase de prueba Tienda):

```
public class Tienda {  
    public static void main(String[] args) {  
        Libro libro1 = new Libro("Crimen y Castigo", 10.40, 752);  
        Pelicula peli1 = new Pelicula("Matrix", 9.99, 150);  
  
        System.out.println(libro1.getNombre() + ", " + libro1.getPrecio() + "€");  
        System.out.println(peli1.getNombre() + ", " + peli1.getPrecio() + "€");  
  
        System.out.println("Total: " + Producto.getTotal(  
            new Producto[] { libro1, peli1 } ) + "€");  
    }  
}
```

❖ Salida:

Nuevo array de referencias *Producto* incializado con instancias de las clases *Libro* y *Pelicula*

```
Crimen y Castigo, 10.4€  
Matrix, 9.99€  
Total: 20.30€
```

Interfaces y Herencia

- Un *interface* puede derivar (*extender*) de otro *interface*
- Como en el caso de la herencia de clases, se empleará la cláusula *extends*
- Un *interface* sólo podrá tener un padre
- Una clase que implemente un *interface* que, a su vez, herede de otro *interface*, deberá implementar *todos* los métodos de dicha cadena de herencia (que no tengan un cuerpo *por defecto*, sean *static* o *private*)
- Una clase podrá implementar cuantos *interfaces* desee, añadiéndolos a la cláusula *implements*. Aunque conceptualmente no podemos hablar de herencia múltiple, este mecanismo proporciona a las clases Java cierta capacidad para “incorporar” diferentes comportamientos
- La implementación múltiple de *interfaces* provocará errores cuando esos interfaces incluyan métodos con el mismo nombre y cuerpo *por defecto*, si la clase no sobrescribe dichos métodos

Interfaces vs Clases abstractas (I)

- Tanto clases abstractas como *interfaces* dan soporte a los principios de **abstracción** (métodos abstractos) y **polimorfismo** de la OOP
- Si bien existen ciertas similitudes (imposibilidad de instanciar objetos y declaración de métodos abstractos), existen notables diferencias:

Clase abstracta	Interface
Puede tener método abstractos y no-abstractos	Sólo métodos abstractos (desde Java 8 puede tener métodos no-abstractos <i>por defecto</i> y estáticos)
No soporta herencia múltiple	Soporta implementación múltiple de <i>interfaces</i>
Puede tener variables <i>final</i> , no- <i>final</i> , <i>static</i> y no- <i>static</i>	Sólo variables <i>public</i> , <i>final</i> y <i>static</i>
Puede tener miembros con diferentes modos de acceso	Los métodos son <i>public</i> (desde Java 9 puede incluir métodos <i>private</i> para uso interno del <i>interface</i>)
Proporciona un nivel de abstracción parcial	Proporciona un nivel de abstracción total

Interfaces vs Clases abstractas (y II)

- Así que la pregunta sería, ¿cuándo usar una u otra?
- Aunque dependerá del caso concreto, podemos tomar como punto de partida las siguientes recomendaciones:
- Optaremos por una clase abstracta si:
 - Existen clases **relacionadas entre sí** que necesitan compartir o reutilizar parte del código
 - Existen clases **relacionadas** que presentan una estructura común
- Optaremos por un *interface* si:
 - Deseamos especificar un comportamiento común para clases no necesariamente relacionadas
 - Necesitamos que las clases puedan incorporar múltiples comportamientos independientes (herencia múltiple)