

```

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.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

04.2

Estructuras de control

II

Indice

- Control de acceso
- Más sobre métodos
 - Objetos como parámetros
 - Retornando objetos
 - Sobrecarga
 - Recursividad
- El tipo *enum*

```

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.data):
                self.handle.send(data)
            self.close(self)
            self.flag=0
            self.sock.close()

```

Control de acceso (I)

- Los principales beneficios de la **encapsulación** que proporcionan las clases, podemos resumirlos en:
 - **enlaza** los datos con el código que los manipula
 - proporciona mecanismos de **control de acceso** a los diferentes miembros de la clase (atributos y métodos)
- En esencia, existen dos tipos básicos de miembros de clase:
 - **público**, que puede ser accedido libremente desde código definido fuera de la propia clase
 - **privado**, que **sólo** puede ser accedido desde métodos definidos en la propia clase
- Restringir el acceso a los miembros de la clase es una parte fundamental de la OOP pues, al proporcionar un interfaz de acceso bien definido, previene el uso incorrecto de los objetos y sus datos

Control de acceso (II)

❑ Modificadores de acceso en Java

- En Java, para establecer el modo de acceso a un miembro de una clase, antepondremos en su declaración uno de los siguientes modificadores: *public*, *private* o *protected*
- Si no se especifica ningún modificador, se aplica el acceso *default*

	<i>default</i>	<i>private</i>	<i>protected</i>	<i>public</i>
misma clase	✓	✓	✓	✓
mismo paquete / subclase	✓	✗	✓	✓
mismo paquete / no subclase	✓	✗	✓	✓
otro paquete / subclase	✗	✗	✓	✓
otro paquete / no subclase	✗	✗	✗	✓

Control de acceso (III)

❖ Acceso *default*

- Es el que hemos estado utilizando principalmente hasta ahora, pues es el que se aplica implícitamente cuando no se indica ninguno

Las clases, métodos y atributos declarados sin especificar ningún modificador de acceso, usarán el modificador *default* y **sólo podrán ser accedidas desde el mismo paquete**

- En el siguiente ejemplo, trataremos de acceder desde una clase de un paquete a otra clase de otro paquete para la que se habrá establecido acceso por defecto:
 - p1/A.java (*default access*)
 - p2/B.java

Control de acceso (IV)

```
//: p1/A.java
```

```
package p1;
```

```
class A {
```

```
    int a1;
```

```
    void display() {
```

```
        System.out.println(  
            "Hello World!");  
    }
```

```
}
```

```
//: p2/B.java
```

```
package p2;
```

```
import p1.A;
```

```
public class B {
```

```
    public static void main(String[] args) {
```

```
        A obj = new A();
```

```
        obj.a1 = 5;
```

```
        obj.display();
```

```
    }
```

```
}
```

- Ninguno de los accesos señalados (clase, método y atributo) estaría permitido, produciéndose los correspondientes errores de compilación

Control de acceso (V)

❖ Acceso *private*

- Es el modificador de acceso más restrictivo

Los métodos y atributos (variables miembro) declarados con el modificador *private* sólo podrán ser accedidas desde la misma clase en que fueron declarados

- El modificador *private* se puede aplicar a clases *anidadas* (*nested class*) pero **no a la clase principal** (*top-level class*)
- En el siguiente ejemplo, crearemos dos clases, A y B, dentro del mismo paquete *p1*. Declararemos los miembros de A como *private* y trataremos de acceder a ellos desde la clase B:
 - *p1/A.java* (*private access*)
 - *p1/B.java*

Control de acceso (VI)

```
//: p1/A.java
package p1;

class A {
    private int a1;

    private void display() {
        System.out.println("Hello World!");
    }
}
```

```
//: p1/B.java
package p1;

public class B {
    public static void main(String[] args) {
        A obj = new A();
        obj.a1 = 5;
        obj.display();
    }
}
```

- Al compilar la clase B, obtendremos el siguiente error:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
    The field A.a1 is not visible
    The method display() from the type A is not visible

    at p1.B.main(B.java:7)
```


Control de acceso (VII)

❖ Acceso *protected*

- Amplía el acceso *default* a subclases de otros paquetes

Los métodos y atributos (variables miembro) declarados con el modificador *protected* podrán ser accedidos desde el mismo paquete o desde subclases de otros paquetes

- El modificador *protected* **no se puede aplicar a clases**
- Vamos a modificar el ejemplo anterior con dos clases, A y B, dentro del mismo paquete *p1*, pero declararemos los miembros de la clase A como *protected* y trataremos de acceder a ellos desde la clase B:
 - *p1/A.java* (*protected access*)
 - *p1/B.java*

Control de acceso (VIII)

```
//: p1/A.java
package p1;

class A
{
    protected int a1;

    protected void display() {
        System.out.println("Hello World!");
    }
}
```

```
//: p1/B.java
package p1;

public class B {
    public static void main(String[] args) {
        A obj = new A();
        obj.a1 = 5;
        obj.display();
    }
}
```

- Al ejecutar la clase B, obtendremos la salida sin errores esperada:

```
hal@bowman:~$ java p1.B
Hello World!
hal@bowman:~$
```

Control de acceso (IX)

❖ Acceso *public*

- El modo de acceso *public* no establece ningún tipo de restricción

Las clases, métodos y atributos (variables miembro) declaradas con el modificador ***public*** podrán ser accedidos sin ningún tipo de restricción desde cualquier punto del código del programa, perteneciente al mismo u otro paquete

- Vamos a modificar el primero de los ejemplo, con dos clases, *A* y *B*, pertenecientes a los paquetes *p1* y *p2*, de forma que el método *display()* sea accesible para *B*:
 - *p1/A.java* (*public access*)
 - *p1/B.java*

Control de acceso (X)

```
//: p1/A.java
package p1;

public class A
{
    int a1;

    public void display() {
        System.out.println(
            "Hello World!");
    }
}
```

```
//: p2/B.java
package p2;

import p1.A;

public class B {
    public static void main(String[] args) {
        A obj = new A();

        //obj.a1 = 5;

        obj.display();
    }
}
```

- El método se ejecutará sin problemas:

```
hal@bowman:~$ java p1.B
Hello World!
```

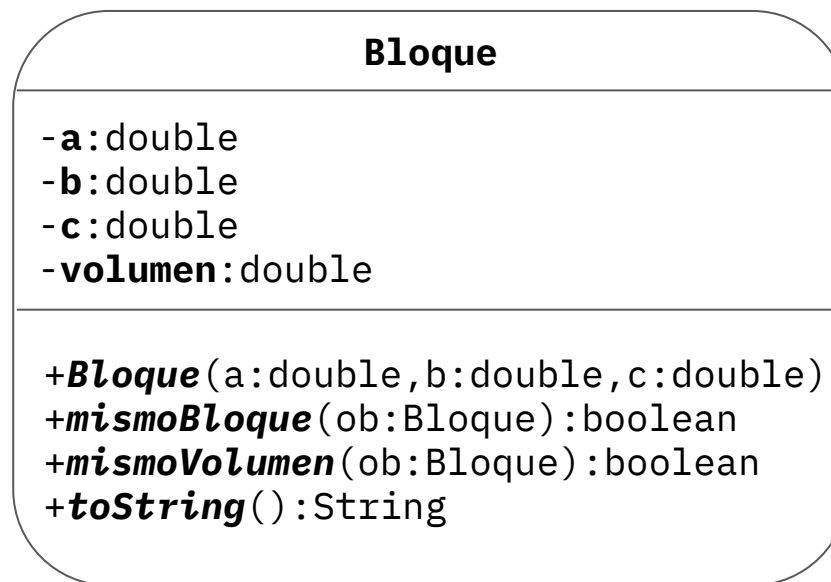
Control de acceso (y XI)

❑ Algunas consideraciones finales...

- El modo de acceso de la clase: *public*, *default* o *private* (sólo *nested-classes*) tiene **precedencia** sobre los modificadores de acceso de sus miembros. Es decir, un miembro *public* de una clase *default*, no podría ser accedido desde una clase de otro paquete
- Los **interfaces** Java pueden tener modificador de acceso *public* o no indicarse, en cuyo caso se aplicará el modificador *default* y sólo podrá ser *implementado* por clases del mismo paquete. Los métodos de un *interface* son **implícitamente public**
- En general, se deben establecer los permisos **más restrictivos** y que permitan la funcionalidad esperada, para cada uno de los miembros de la clase. Se usará el modificador *private* salvo que exista una razón para no usarlo.
- Los atributos de una clase **deben** ser *private* (salvo las constantes) y el acceso a ellos se realizará mediante métodos *get/set* públicos

Métodos: objetos como parámetros (I)

- Hasta ahora, en las llamadas a métodos, hemos usado simples argumentos de tipos primitivos. Sin embargo, es común declarar parámetros de tipo *referencia* (clases) y que, por tanto, reciban objetos como argumentos
- Para ilustrarlo, vamos a implementar la siguiente clase:



Métodos: objetos como parámetros (II)

```
//: Bloque.java
```

```
public class Bloque {  
    private double a;  
    private double b;  
    private double c;  
    private double volume;  
  
    public Bloque(double a, double b, double c) {  
        this.a = a;  
        this.b = b;  
        this.c = c;  
        volume = a*b*c;  
    }  
  
    public boolean mismoBloque(Bloque ob) {  
        return this.a == ob.a &&  
            this.b == ob.b &&  
            this.c == c;  
    }  
}
```

```
    public boolean mismoVolumen(Bloque ob) {  
        return this.volume == ob.volume;  
    }  
  
    public String toString() {  
        return "Bloque [a: " + a +  
            ", b: " + b +  
            ", c: " + c + "];"  
    }  
}  
  
//:~
```

Métodos: objetos como parámetros (III)

- La siguiente clase nos permitirá probar los métodos de la clase Bloque. Fíjate como, en este caso, los argumentos son objetos:

```
class BloqueDemo {  
    public static void main(String[] args) {  
        Bloque b1 = new Bloque(10, 2, 5);  
        Bloque b2 = new Bloque(10, 2, 5);  
        Bloque b3 = new Bloque(4, 5, 5);  
  
        System.out.println("b1 mismas dimensiones que b2: " + b1.mismoBloque(b2));  
        System.out.println("b1 mismas dimensiones que b3: " + b1.mismoBloque(b3));  
        System.out.println("b1 mismo volumen que b3: " + b1.mismoVolumen(b3));  
    }  
}
```

```
bowman@hal:~/work/src/java$ java BloqueDemo  
b1 mismas dimensiones que b2: true  
b1 mismas dimensiones que b3: false  
b1 mismo volumen que b3: true
```

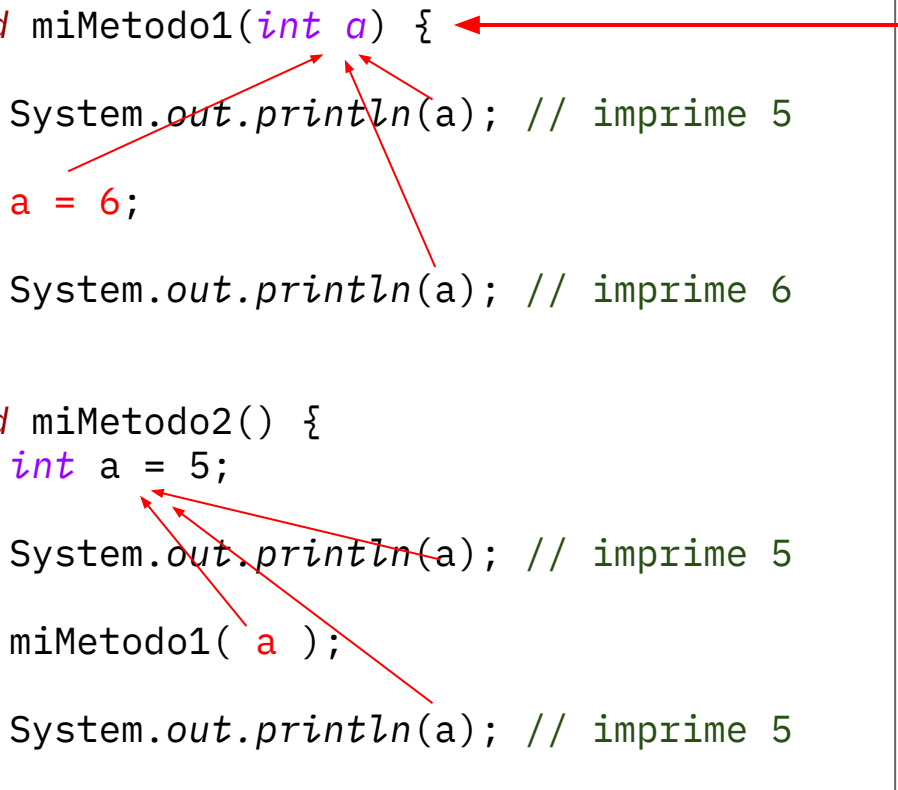

Métodos: objetos como parámetros (IV)

- Como vemos, pasar objetos como argumentos a nuestros métodos no tiene mayor complicación, pues el mecanismo empleado es el mismo que con cualquier tipo primitivo.
- Sin embargo, bajo el telón, existen ciertas diferencias que debemos tener en cuenta para evitar situaciones no deseadas.
- Cuando el parámetro es de un tipo primitivo, se crea en el método una variable local (*formal parameter*) para recoger el valor pasado como argumento. Es decir, el parámetro del método es absolutamente independiente de cualquier otra variable que se haya empleado como argumento. Sólo tiene una **copia** de su valor. Cualquier cambio que hagamos al valor del parámetro en el método no afectará a la variable empleada en la llamada
- Es lo que se conoce como **paso de parámetros por valor** (*call-by-value*)

Métodos: objetos como parámetros (V)

❖ Ejemplo:

```
void miMetodo1(int a) {  
    System.out.println(a); // imprime 5  
    a = 6;  
    System.out.println(a); // imprime 6  
}  
  
void miMetodo2() {  
    int a = 5;  
    System.out.println(a); // imprime 5  
    miMetodo1( a );  
    System.out.println(a); // imprime 5  
}
```



El **parámetro *a*** tiene una copia del valor de la **variable *a*** de *miMetodo2()*
Son variables **independientes**

Métodos: objetos como parámetros (VI)

- Sin embargo, cuando el argumento es un objeto, lo que obtiene el método es una *referencia* al objeto (no una copia del objeto). Es lo que se denomina *paso por referencia* (*call-by-reference*)
- Este hecho puede tener “dramáticas” consecuencias pero, para comprenderlas, debemos entender que una variable de tipo *referencia* lo que almacena en realidad no es el “propio objeto” (como pasa con las variables de tipos primitivos), lo que almacena es la *dirección* (referencia) del objeto en memoria.
- Cuando la JVM crea un objeto, le asigna cierta cantidad de espacio en el *heap* (espacio de memoria empleado por la JVM para “alojar” nuevos objetos y clases del JRE). La variable de tipo referencia almacena la dirección de comienzo de dicha estructura en memoria. Así, cada vez que usemos dicha variable, la JVM localizará el objeto en memoria a partir de la dirección almacenada

Métodos: objetos como parámetros (VII)

- Esto se ve claramente cuando asignamos el valor de una variable de tipo referencia a otra. No obtenemos una copia del objeto, si no que ambas variables “apuntarán” al mismo objeto

```
//: Test.java

class A { int a; }

public class Test {
    public static void main(String[] args) {
        A obj1, obj2;

        obj1 = new A();
        obj1.a = 5;
        System.out.println(obj1.a); // imprime 5

        obj2 = obj1; // copia la referencia, no el objeto
        obj2.a = 10; // modificamos obj1 a través de obj2
        System.out.println(obj1.a); // imprime 10 (no 5)
    }
}

/* Output:
5
10
*///:~
```

Métodos: objetos como parámetros (y VIII)

- Así, deberemos tener claro que, cuando usamos parámetros de tipo referencia, éste recibe la dirección del propio objeto. Por tanto, cualquier modificación realizada en el método a través del parámetro se estará realizando sobre el objeto externo

```
//: Test.java

class A { int a; }

public class Test {
    public static void main(String[] args) {
        A obj1 = new A();
        obj1.a = 5;
        System.out.println(obj1.a); // imprime 5

        change(obj1, 10); // modificamos obj1 a través del método
        System.out.println(obj1.a); // imprime 10 (no 5)
    }
    public static void change(A obj, int val) { obj.a = val; }
}

/* Output:
5
10
*///:~
```

Métodos: retornando objetos (I)

- Un método puede **retornar** cualquier tipo de dato, incluidos los tipos *referenciados*. Un ejemplo de esto lo vimos en nuestra clase *Bloque*, donde el método *toString()* devuelve un objeto de la clase *String*
- Por supuesto, podemos retornar objetos de **cualquier** clase creada por nosotros. En el siguiente ejemplo vamos a añadir un nuevo método a la clase *Bloque* para obtener un nuevo bloque escalado a partir del original

```
//: Bloque.java

//...

public Bloque escalaBloque(double factor) {
    return new Bloque(a*factor, b*factor, c*factor);
}
```

Métodos: retornando objetos (II)

- Probemos el nuevo método...

```
class BloqueDemo2 {  
    public static void main(String[] args) {  
        Bloque b1 = new Bloque(10, 2, 5);  
        Bloque b2 = b1.escalaBloque(2.0);  
  
        System.out.println("b1: " + b1);  
        System.out.println("b2: " + b2);  
    }  
}
```

```
bowman@hal:~/work/src/java$ java BloqueDemo2  
b1: Bloque [a: 10.0, b: 2.0, c: 5.0]  
b2: Bloque [a: 20.0, b: 4.0, c: 10.0]
```

- Cada vez que invoquemos el método *escalaBloque()* obtendremos un **nuevo** objeto de la clase Bloque
- El objeto retornado por un método permanecerá en existencia hasta que no haya más **referencias** a él, momento en que será eliminado por GC

Métodos: retornando objetos (II)

```
//: Bloque.java
```

```
public class Bloque {  
    private double a;  
    private double b;  
    private double c;  
    private double volume;  
  
    public Bloque(double a, double b, double c) {  
        this.a = a;  
        this.b = b;  
        this.c = c;  
        volume = a*b*c;  
    }  
  
    public boolean mismoBloque(Bloque ob) {  
        return this.a == ob.a &&  
            this.b == ob.b &&  
            this.c == c;  
    }  
}
```

```
public boolean mismoVolumen(Bloque ob) {  
    return this.volume == ob.volume;  
}
```

```
public String toString() {  
    return "Bloque [a: " + a +  
        ", b: " + b +  
        ", c: " + c + "];"  
}
```

```
//:~
```


Métodos: sobrecarga (I)

- En Java, dos o más métodos de la misma clase pueden compartir el nombre, siempre y cuando su declaración de parámetros sea distinta. En este caso, diremos que dichos métodos están **sobrecargados**
- Un caso típico de sobrecarga de métodos es el de disponer de diferentes constructores para una clase. Esto facilitará la creación de objetos de dicha clase a partir de diferentes conjuntos de argumentos. Las clases del API de Java son también un claro ejemplo del empleo de la sobrecarga de métodos. Por ej: [java.lang.Math.max\(\)](#)
- En general, para sobrecargar un método, simplemente declararemos una nueva versión de él y el compilador se encargará del resto. La única **restricción** será que el número y/o tipo de los parámetros de cada método sobrecargado sea diferente. **No es suficiente** con que dos métodos difieran únicamente en el tipo del **valor devuelto**.

Métodos: sobrecarga (II)

- Vamos a modificar nuestra clase Bloque para sobrecargar el constructor con una nueva versión que nos permita crear un cubo (las tres dimensiones serán iguales), y una nueva versión del método de escalado para tener factores de escalado independientes en cada dimensión:

```
//: Bloque.java
//...
// versión 2 del constructor
public Bloque(double d) {
    this(d, d, d);
}
//...
// versión 2 de escalaBloque
public Bloque escalaBloque(double fa, double fb, double fc) {
    return new Bloque(a*fa, b*fb, c*fc);
}
//...
```

Métodos: sobrecarga (y III)

- Probemos los nuevos métodos...

```
class BloqueDemo2 {  
    public static void main(String[] args) {  
        Bloque b1 = new Bloque(10, 20, 30);  
        Bloque b2 = new Bloque(10);  
        Bloque b3 = b2.escalaBloque(1.0, 2.0, 3.0);  
  
        System.out.println("b1: " + b1);  
        System.out.println("b2: " + b2);  
        System.out.println("b3: " + b3);  
  
        System.out.println("b1 igual a b3: " + b1.mismoBloque(b3));  
    }  
}
```

```
bowman@hal:~/work/src/java$ java BloqueDemo2  
b1: Bloque [a: 10.0, b: 20.0, c: 30.0]  
b2: Bloque [a: 10.0, b: 10.0, c: 10.0]  
b3: Bloque [a: 10.0, b: 20.0, c: 30.0]  
b1 igual a b3: true
```

Métodos: recursividad (I)

- Muchos lenguajes, Java entre ellos, soportan que un método pueda llamarse a si mismo. Es lo que se conoce como **recursividad**
- En cierto modo podemos verlo como una definición circular de un método, al incluir una sentencia en la que se invoca a si mismo
- El clásico ejemplo en programación de un método recursivo es la implementación del cálculo del *factorial* de un número. Esta función consiste en el producto de todos los números enteros entre 1 y el número del que queremos calcular su factorial:

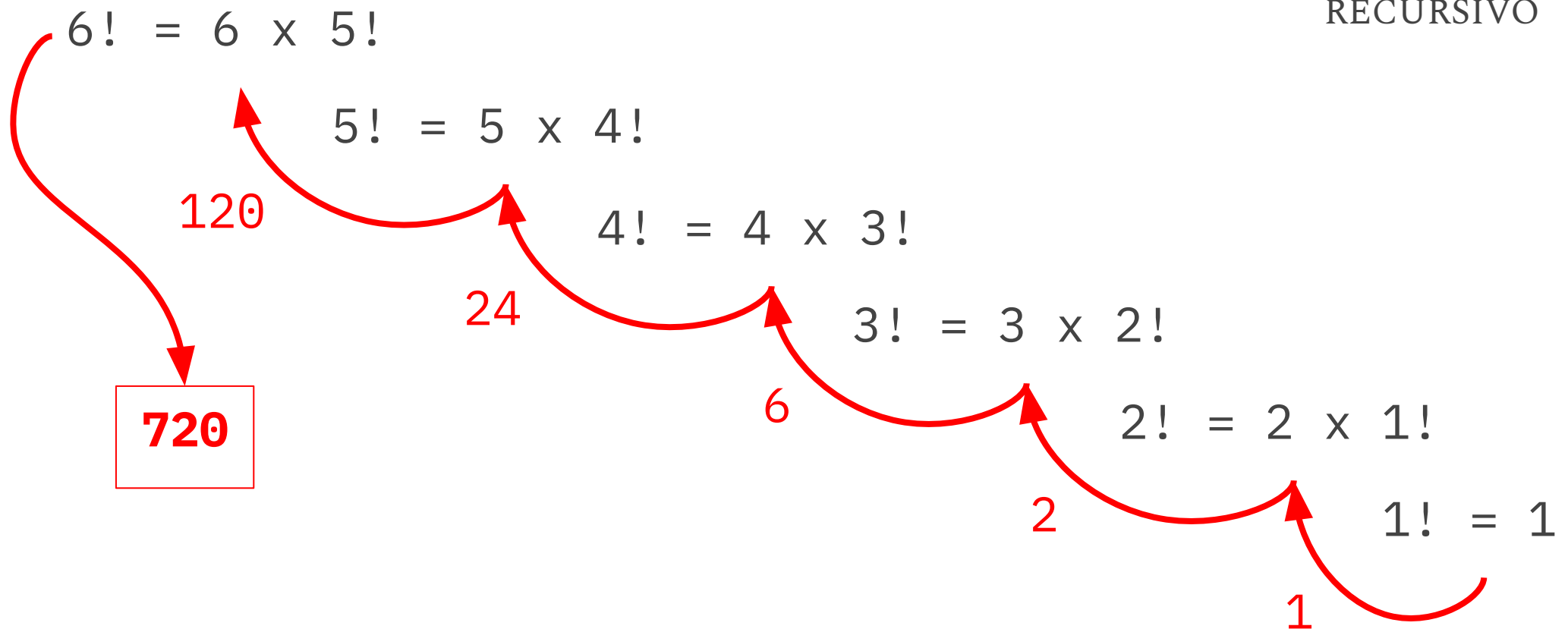
$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

- Si nos fijamos en el ejemplo, el factorial de 6 no es más que el producto de 6 por el factorial del número que le precede, 5 ($6! = 6 \times 5!$) y éste, a su vez, sería 5 por el factorial de 4 ($5! = 5 \times 4!$) y así... hasta llegar a 1

Métodos: recursividad (II)

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

NO-RECURSIVO



RECURSIVO

Métodos: Recursión (III)

- Así pues, podríamos describir la función *factorial* de un número N como:

$$fact(n) = \begin{cases} 1 & \text{si } n=1 \\ n \times fact(n-1) & \text{si } n>1 \end{cases}$$

$\forall n \in \mathbb{N}$ (para cualquier número n natural: 1,2,3,...)

- Implementemos entonces las dos versiones del método para calcular el *factorial* de un número natural recibido como argumento. La primera, *factLoop()*, empleará un bucle para el cálculo de los productos mientras que la segunda, *factRec()*, se basará en el empleo de la recursividad para la realización del cálculo

Métodos: Recursión (IV)

```
//: Factorial.java
class Factorial {
    public static long factLoop(int n) {
        long f = 1;
        for(int i=2; i<=n; i++) { f *= i; }
        return f;
    }

    public static long factRec(int n) {
        if(n==1) { return 1; }
        return n*factRec(n-1);
    }

    public static void main(String[] args) {
        System.out.println(factLoop(10));
        System.out.println(factRec(10));
    }
}

/* Output:
6! = 720
6! = 720
*///:~
```

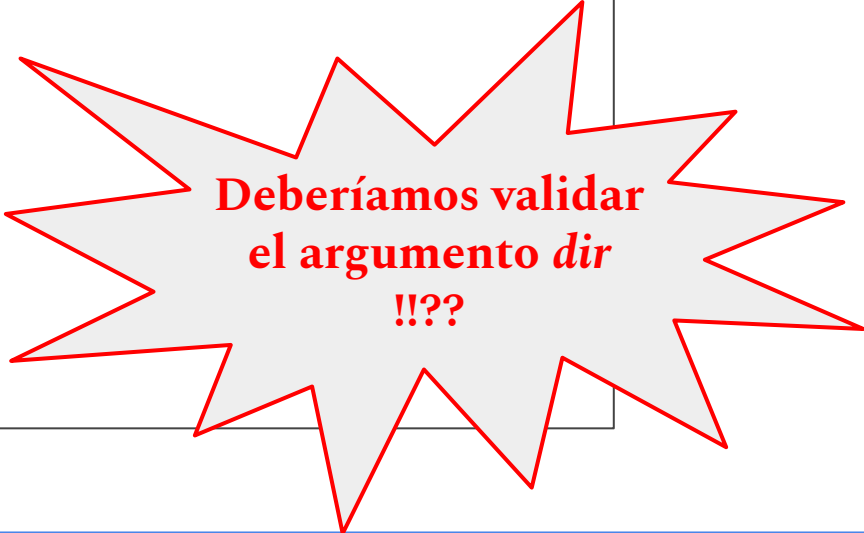
Métodos: Recursión (y V)

- En situaciones como la mostrada, la “versión” recursiva de un método puede ejecutarse de forma algo más lenta. Esto es debido a la “sobrecarga” de procesamiento que suponen las sucesivas llamadas al método: actualización de la pila (*stack*), creación de nuevas variables locales,... Es más, podría llegar a provocar la terminación anormal del programa al agotar los recursos existentes
- Sin embargo, en determinados problemas, el empleo de la recursividad supone una aproximación más **simple** y **elegante** a su solución. Un claro ejemplo de ello es el algoritmo de ordenación **QuickSort**, difícilmente implementable de manera iterativa. Problemas relacionados con la **IA**, por ejemplo, también tienden a soluciones recursivas
- Por último, es **fundamental** que nuestros métodos recursivos establezcan alguna **condición** que les permita **finalizar** sin una nueva llamada al método, o se ejecutarán por siempre (en el ejemplo: *if(n==1) return 1;*)

El tipo *enum* (I)

- En ocasiones, necesitamos crear *listas* de constantes para facilitar la implementación y legibilidad de nuestros programas. Por ejemplo, para crear variables que almacenen el día de la semana (“Lunes”, “Martes,...”) o los puntos cardinales (“N”, “S”, ...)

```
class Orientacion {  
    public static final NORTE = "N";  
    public static final SUR = "S";  
    public static final ESTE = "E";  
    public static final OESTE = "O";  
  
    String dir;  
  
    public Orientacion(String dir) {  
        this.dir = dir;  
    }  
}
```



**Deberíamos validar
el argumento *dir*
!!??**

El tipo *enum* (II)

- Para facilitar la creación de listas de constantes (**enumeración**), Java nos proporciona el tipo especial *enum*
- Al declarar una **variable** de un tipo *enum* específico, sólo podrá tomar uno de los valores especificados en dicha enumeración
- Para crear un tipo enumerado, usaremos la siguiente sintaxis:

```
enum nombre {  
    NOMBRE_CONSTANTE_1,  
    NOMBRE_CONSTANTE_2,  
    . . .  
    NOMBRE_CONSTANTE_n,  
}
```

- Los tipos enumerados son en realidad **clases** Java (limitadas), por lo que se aplican las mismas **restricciones** en cuanto a ficheros y modos de acceso

El tipo *enum* (III)

❖ Ejemplo

```
//: EnumDemo.java
enum Direcciones { NORTE, SUR, ESTE, OESTE }

public class EnumDemo {
    public static void main(String[] args) {
        Direcciones d;

        d = Direcciones.NORTE; // asignación de un valor

        // esta asignación generaría un error:
        // d = "NORTE";

        System.out.println("Voy hacia el " + d);
    }
}

/* Output:
Voy hacia el NORTE
*/:~
```

El tipo *enum* (IV)

- Vemos que la asignación de valores a las variables de tipo enumerado se hace indicando una de las constantes definidas en la enumeración
- No podemos asignar, por ejemplo, un *String* como “NORTE”. Sin embargo, podemos usar el método *valueOf()* para convertir dicho *String* al valor correspondiente de la enumeración:

```
d = Direcciones.valueOf("NORTE");
```

- Dada la propia naturaleza de clase de las enumeraciones, podremos definir métodos y atributos que tomen valores diferentes para cada constante de la enumeración. Podemos definir un constructor con parámetros para la enumeración que se invocará en el momento de crear cada una de las constantes (no lo podemos usar para crear objetos)
- Veamos un ejemplo:

El tipo *enum* (V)

```
//: EnumParamDemo.java
enum Direcciones {
    // cada constante lleva un valor que se pasa al constructor
    NORTE(1), SUR(2), ESTE(3), OESTE(4); // fin declaración constantes
                                     // tienen que ir al principio

    private final int value; // atributo
    Direcciones(int value) { this.value = value; } // constructor
    public int getValue() { return this.value; } // getter del atributo
}

public class EnumParamDemo {
    public static void main(String[] args) {
        Direcciones d = Direcciones.NORTE;
        System.out.println("Voy al " + d + " (value: " + d.getValue() + ")");
    }
}

/* Output:
Voy al NORTE (value: 1)
*/:~
```

El tipo *enum* (y VI)

- Por último, los tipos enumerados se usan habitualmente en el control de sentencias de selección *switch*:

```
//: DondeVas.java
enum Direcciones { NORTE, SUR, ESTE, OESTE }
public class DondeVas {
    public static void main(String[] args) {
        Direcciones d = Direcciones.SUR;
        switch(d) {
            case NORTE:    // sólo se pone el nombre de la constante
            case SUR:
                System.out.println("Voy al " + d);
                break;
            default:
                System.out.println("Voy al... Buff! no sé...");
        }
    }
}
/* Output:
Voy al SUR
*//:~
```