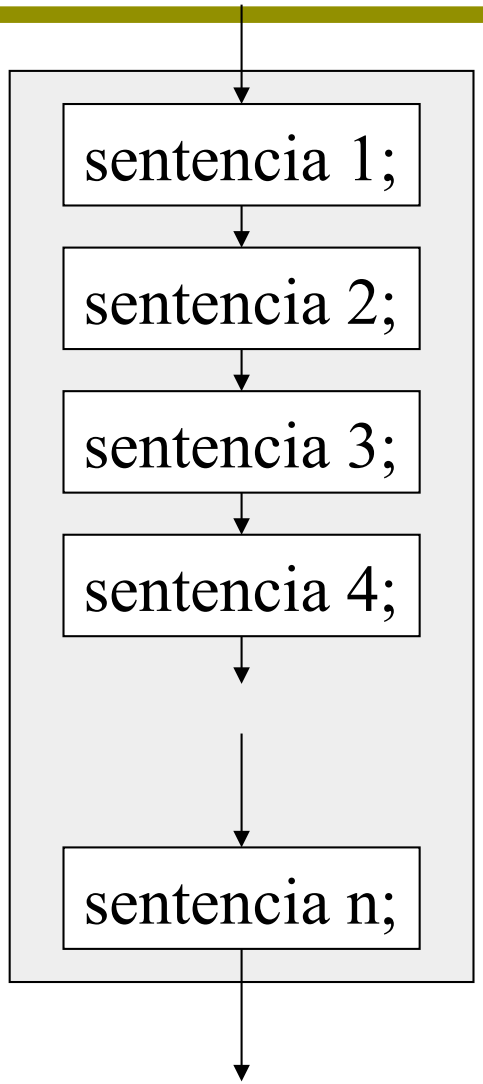
Programación Estructurada

- Una forma de estructurar los programas para que **no sean un laberinto**, sino algo fácil de entender por uno mismo y por los demás
- La idea fundamental es que los programas son un conjunto de bloques que encajan entre sí sobre la premisa de que cada bloque tiene **un sólo punto de entrada** y **un sólo punto de salida**, y <u>la salida de uno se enchufa a la entrada del siguiente</u>
  - o sea, un lego

Secuencias y bloques



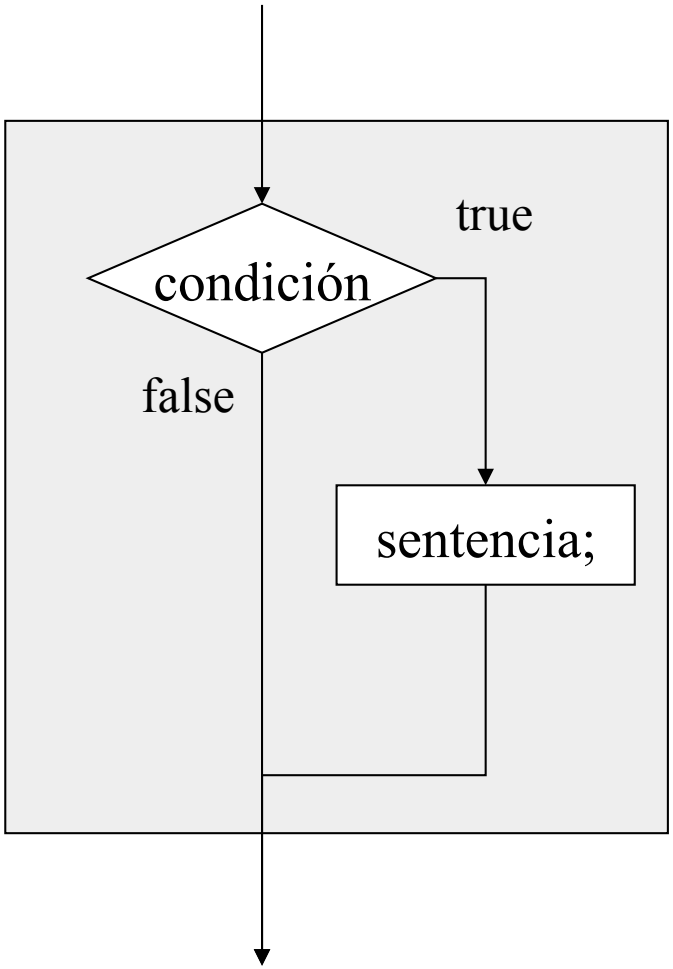

```
{   // principio de un bloque
  sentencia 1;
  sentencia 2;
  sentencia 3;
  sentencia 4;
  ... ... ...
  sentencia n;
}  // fin de un bloque
```

- el “;” es terminador

Contenidos

- **1. Sentencias condicionales**
- 2. Sentencias iterativas
- 3. Recorridos de arrays
- 4. Saltos y excepciones
- 5. Métodos recursivos

Sentencias Condicionales




```
temperatura = medidaTermometro();
if (temperatura < 8.0) {
    ropa=ropaDeInvierno();
}
```

```
private int monedas (int aDevolver, int valor)
{
        int cuantas= aDevolver / valor;
        if (cuantas > 0)
            System.out.println (cuantas + " de ”
                        + valor);
        return aDevolver % valor;
 }
```

Errores típicos

- Caso de duda, póngase { }

```
if (condición)
   sentencia1;
   sentencia2;
   sentencia3;
```

se entiende como

```
if (condición)
   sentencia1;
sentencia2;
sentencia3;
```

que no es lo mismo que

```
if (condición) {
   sentencia1;
   sentencia2;
   sentencia3;
}
```

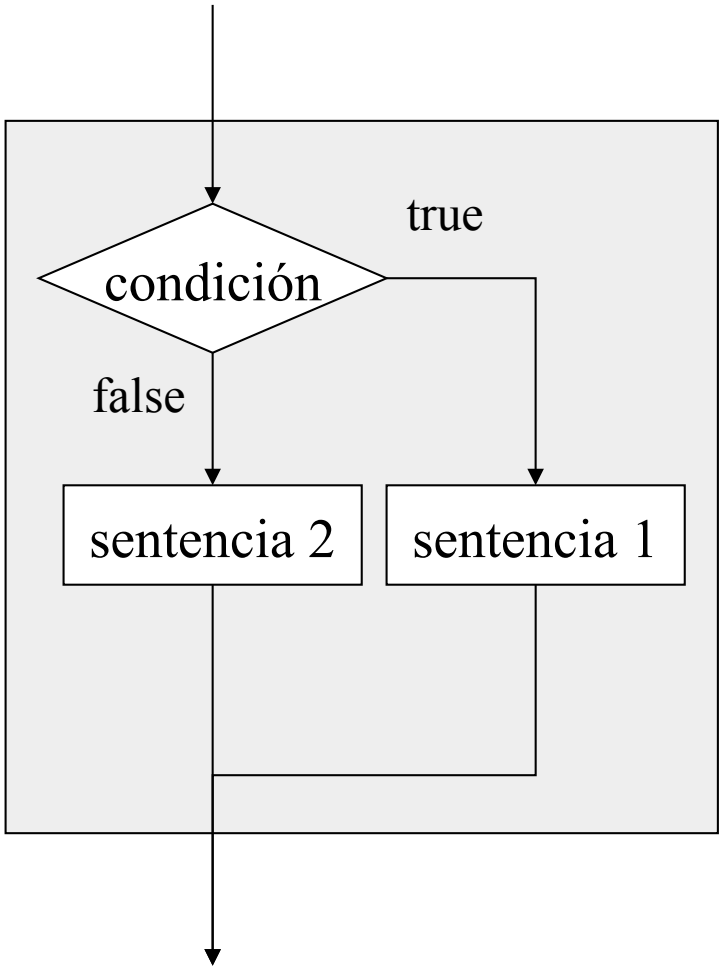
Regla:
En caso de duda o ambigüedad, ponga llaves

Ejecución condicional

Iniciar una máquina de cambio

```
public static void main (String[] args) {
  if (args.length != 2) {
      System.out.println ("uso: java MaquinaCambio vuelta tipoMoneda");
      System.exit(1);
   }
   MaquinaCambio panaderia= new MaquinaCambio();
   panaderia.daCambio(Integer.parseInt(args[0]),
                              Integer.parseInt(args[1]));
}
```

Ejecución Condicional




```
if (hora < 14)
   System.out.println ("buenos dias");
else
   System.out.println ("buenas tardes");
```

```
if (condicion) sentencia1;
else sentencia2;
```

```
if (condicion) {
        sentencia1;
        sentencia2;
} else {
        sentencia3;
        sentencia4;
}
```

Ambigüedad

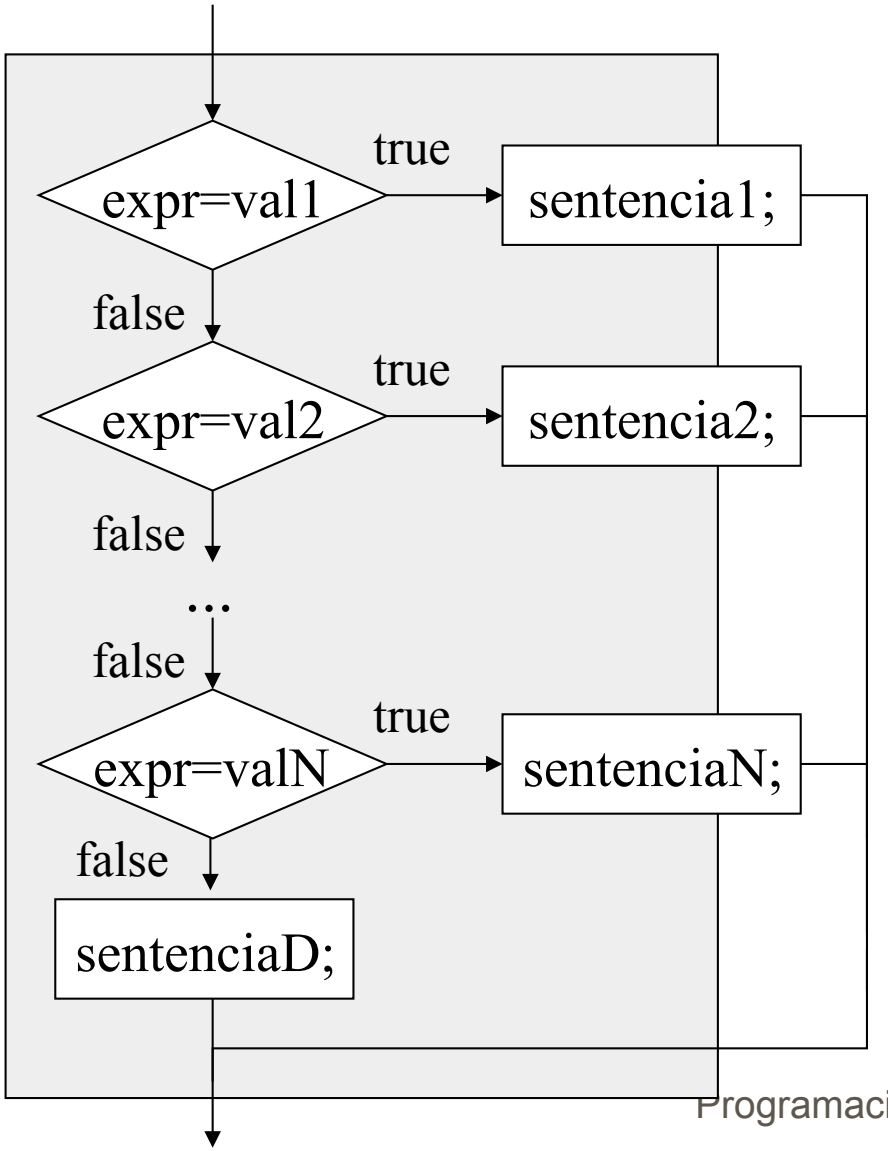
if (a) if (b) S1; else S2;

```
if (a) {
        if (b)
                S1;
        else
                S2;
} //fin condicion a
```

~~
```
if (a) {
        if (b)
                S1;
} else {
        S2;
} //fin condicion a
```
~~

if (a) { if (b) S1;} else S2;

Selección múltiple




```
if (mes == 1)
   ...print("enero");
else if (mes == 2)
   ...print("febrero");
else if (mes == 3)
   ...print("marzo");
...
...
else
   ...print("no se");
```

- La expresión se evalúa N veces

Selección múltiple

```
if (mes == 1)
    ...print("enero");
else if (mes == 2)
    ...print("febrero");
else if (mes == 3)
    ...print("marzo");
...
...
else
    ...print("no se");
```

```
switch (mes) {
  case 1:
      ...print("enero");
      break;
  case 2:
      ...print("febrero");
      break;
...
  default:
      ...print("no se");
}
```

expresión

valor

el break me saca fuera

Selección múltiple

```
switch (mes) {
case 1: case 3: case 5: case 7: case 8: case 10: case 12:
   dias =31;
    break;
case 4: case 6: case 9: case 11:
   dias = 30;
    break;
case 2:
   if (bisiesto)
            dias =29;
   else
            dias =28;
   break;
default:
   dias = 0;
}
```

expresión constante

Selección múltiple

```
switch (verbosidad) {
case EXTREMA:
    System.out.println(todosLosDetalles);
case NOTIFICACIONES:
    System.out.println(lasNotificaciones);
case ERRORES:
     System.out.println(losErrores);
    break;
default:
    System.out.println("nivel de verbosidad desconocida");
}
```

Sentencia **switch**

- Funcionan preferentemente sobre enteros, booleanos o caracteres, literales de enum
  - conocidos antes de comenzar la ejecución del switch
- Chequean que no haya duplicados
  - las condiciones tienen que ser excluyentes
- Mejor legibilidad
  - En caso de múltiples resultados de una condición
- Sin **break**, el programa no saltaría al final
  - Si hubiera **return** no haría falta **break**
- El **default** es opcional

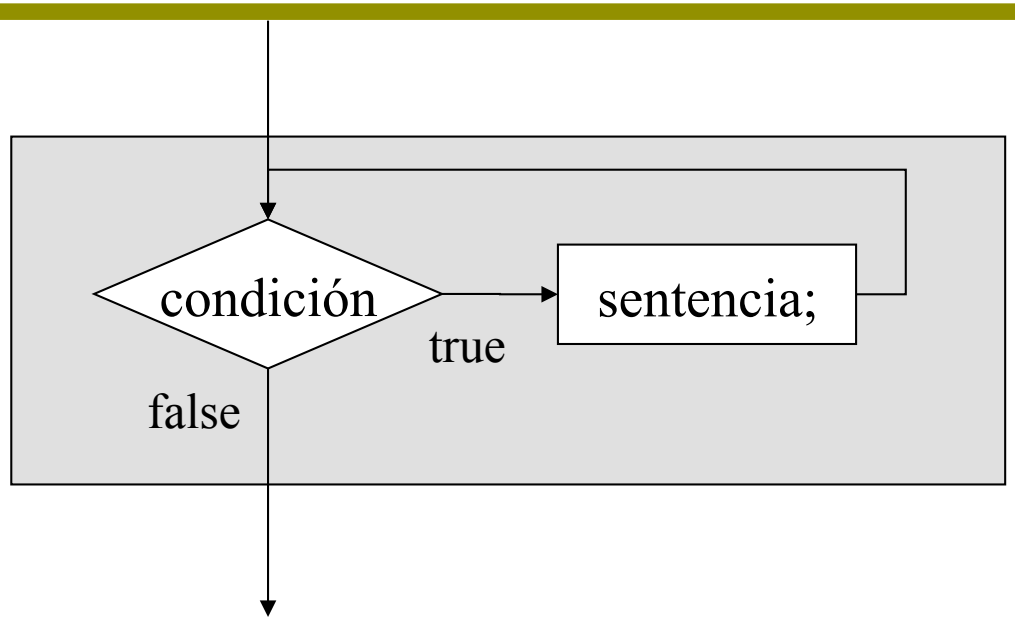
Contenidos

- 1. Sentencias condicionales
- **2. Sentencias iterativas**
- 3. Recorridos de arrays
- 4. Saltos y excepciones
- 5. Métodos recursivos

Bucles

- Se repite una acción N veces
- modelo **while (…){…}**
  - se repite **0 o más** veces
- modelo hasta, **do {…} while (…)**
  - se repite **1 o más** veces
- modelo **for (…) {…}**
  - se repite **un número determinado** de veces

while



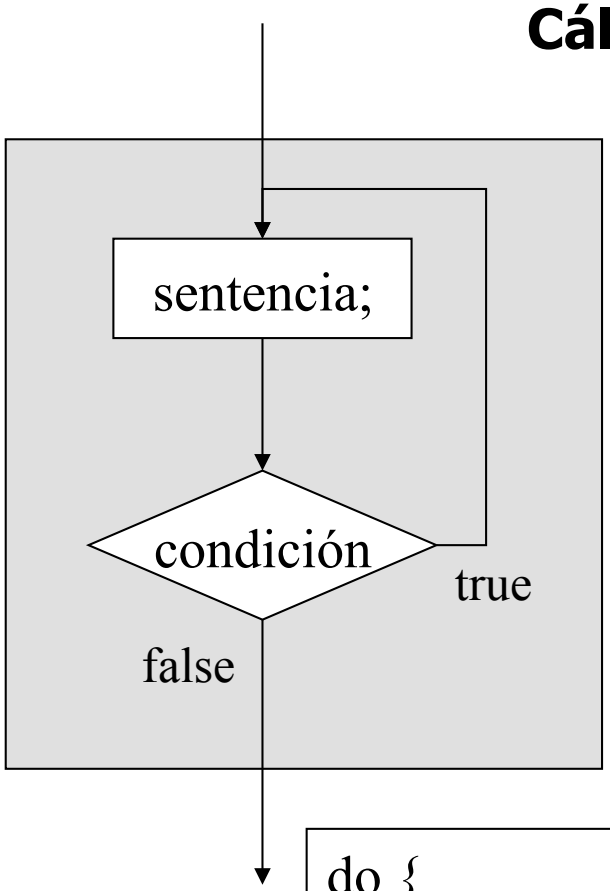

```
int factorial (int n){
    int fact= 1;
    while (n > 0) {
        fact*= n;
        n--;
    }
    return fact;
}
```

```
while (!aprobado)
    me.examino();
```

```
while (llueve())
    uso.paraguas();
```

do { … } while

Cálculo numérico de *e* con una precisión dada



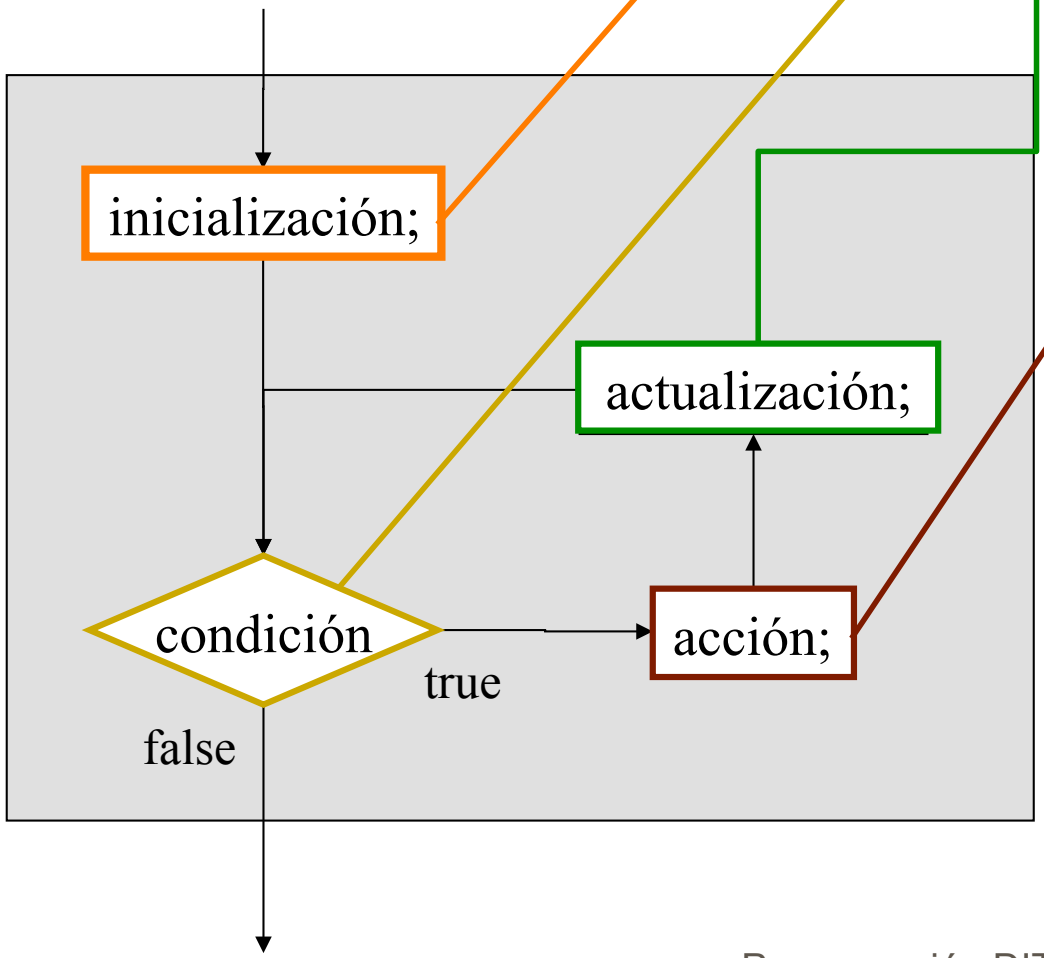
$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

```
int i = 0;
double e, ei;

ei= 1;
e= ei;

do {
    i++;
    ei= ei/i;
    e= e+ei;
    System.out.println ("e= " + e);
} while (ei > 1e-10);
```

```
do {
    nota = examen();
}while (nota < 5.0);
```

for

```
for (int i= 0; i < 10; i++){ …; print(i); }
```




```
for (int i= 10; i > 0; i--)
  ...print(i);
```

```
int factorial (int m) {
    int fact = 1;
    for (int n= m; n > 0; n--)
      fact*= n;
    return fact;
}
```

Ejemplo de for

- Imprimir argumentos de línea de llamada
  - recorrido de un array

Recorro la lista de argumentos

```
class ImprimeArgumentos {
    public static void main (String args[]) {
        for (int i=0; i < args.length; i++) {
            System.out.print ("argumento " + i);
            System.out.println(": " + args[i]);
        }
    }
}
```

Uso el argumento

Variables de control

- Se puede inicializar cualquier número de variables del ***mismo*** tipo

```
for (int i= 0, j= 10; i < 9 && j > 0; i++, j--)
  acción ;
```

separador

- Si estaba declarada antes, al acabar el `for` queda con el valor final
- Si se declara en el `for`, sólo existe en el bucle, por lo tanto su valor no podrá ser usado fuera
- La inicialización, condición y actualización son opcionales (¡ojo! **bucles infinitos**)

Bucles sin fin

- Bucles evidentes para el programador:

- **for (;;) {** // todos los campos vacíos
  se repite sin fin ;
  }
- **while (true) {** // La condición siempre “true”
  se repite sin fin;
  }
- **do {**
  se repite sin fin;
  }while (true); // La condición siempre “true”

- Erratas:
- **for (int i=0; j<10; i++) {** // la opción de salida nunca se cumple
  se repite sin fin ;
  }

Elección de estructura

1. Si nº fijo de veces, elegir `for`
2. Si 1 ó más veces, elegir `do { } while ();`
3. Si 0 ó más veces, elegir `while () {}`

- se trata de elegir la que mejor se ajuste al concepto para mejorar la **legibilidad**

Contenidos

- 1. Sentencias condicionales
- 2. Sentencias iterativas
- **3. Recorridos de arrays**
- 4. Saltos y excepciones
- 5. Métodos recursivos

Arrays: declaración

- Colectivo enumerado de valores del mismo tipo
  - `int[] vector;` // vector de enteros
  - `double[][] matriz;` // matriz de reales
  - `Tipo[]...[] variable;` // N dimensiones

- Número de elementos **<= Valor no modificable**
  - `vector.length`
  - `matriz.length`
  - `matriz[i].length`
  - `Variable.length`
  - `Variable[i].length`
  - `Variable[i]..[n].length`

Recorridos de arrays

- Acceso por elemento

```
int i= vector[0];
```

- Recorrido por posición: 0… length-1

```
for (int i= 0; i < vector.length; i++)
  vector[i] *= k;
```

- Recorrido de arrays multidimensionales por posición

```
void imprime (double[][] m) {
    for (int i= 0; i < m.length; i++) {
        for (int j= 0; j < m[i].length; j++)
            System.out.print(m[i][j] + " ");
        System.out.println();
    }
}
```

Recorridos de arrays

- Recorrido por tipo de elemento - sumatorio

```
int suma = 0;
for (int elemento : vector)
  suma += elemento;
```

- Recorrido de arrays multidimensionales por elemento

```
void imprime (double[][] m) {
    for ( double[] fila : m ) {
        for ( double elemento : fila )
            System.out.print( elemento + " ");
        System.out.println();
    }
}
```

Operaciones frecuentes I

- Inicialización

```
for (int i = 0; i < a.length; i ++)
   a[i] = 0;
```

- Recorrido –para imprimir

```
for (int i = 0; i < a.length; i ++)
   System.out.println(a[i]);
for (int n : a)
   System.out.println(n);
```

- Buscar un valor

```
int buscado;
boolean encontrado = false;
for (int i = 0; i < a.length; i ++) {
   if (a[i] == buscado) {
      encontrado = true; break;
}
```

Operaciones frecuentes II

- Buscar un valor-encontrar su índice (el primero)

```
int buscado;
int casilla = -1;
for (int i = 0; i < a.length; i ++) {
   if (a[i] == buscado) {
      casilla = i;
      break;
}
```

- Ejercicio: cuántas veces aparece un valor
- Buscar el máximo (o el mínimo)

```
int max = a[0];
for (int i : a)
   if (a[i] > max)
      max = a[i];
```

Operaciones frecuentes III

- Sumatorio

```
int suma = 0;
for (int i = 0; i < a.length; i ++) {
   suma += a[i];
}
```

- Contar elementos que cumplan un criterio

```
int n = 0;
for (int i : a)
   if (a[i] > 0)
      n++;
```

- Buscar diferencias máximas

```
double max = Math.abs( Math.abs (a[0]) - Math.abs(a[1]));
for (int i = 0; i < a.length; i++)
 for (int j = i + 1; j < a.length; j++) {
   double d = Math.abs(Math.abs (a[i]) - Math.abs(a[j]));
   if (d < max)
     max = d;
 }
```

Operaciones frecuentes IV

- Copiar a otro array elementos que cumplen un criterio

```
int n = 0;
for (int i = 0; i < a1.length && n < a2.length; i ++) {
  if (vale (a1[i])) {
    a2[n] = a1[i];
    n++;
  }
}
```

- Insertar elemento al final (llevando la cuenta)

```
If (n < a.length) {
  a[n] = i;
  n++;
}
```

- Borrar elemento en la casilla e-sima sin dejar hueco (llevando la cuenta)

```
for (int i = e; i < a.length-1; i ++)
  a[i] = a[i+1];
n--;
```

Operaciones frecuentes V

- Buscar un elemento en un array de referencias

```
Punto buscado = …;
Int indice = -1;
for (int i = 0; i < ps.length; i ++) {
  if (ps[i] != null && buscado.equals(ps[i])) {
    indice = i;
  }
}
```

- Insertar en casilla vacía

```
For (int i = 0; i < ps.length; i ++)
  if (ps[i] == null)
    ps[i] = p;
```

- Borrar elemento en la casilla e-sima sin dejar hueco

```
For (int i = e; i < a.length-1; i ++)
  a[i] = a[i+1];
```

Bucles: errores frecuentes

- Hacer una vez de más o de menos
- Equivocarse en el **caso inicial o final**
  - Hay que probar casos extremos: condición que no se satisface, sólo se satisface una vez, …
- No comprobar que las variables están correctamente inicializadas y donde se usan
- Poner **condiciones exactas**: (mejor, por exceso)
  - while (i != 10)…
    - while (i < 10) … // por si acaso
  - Poner **condiciones == o != sobre reales**

Recorrido de Matrices de 2 dimensiones

Buscar el menor entero en una Matriz de 2 dimensiones:

```
 private static int minimo (int [][]datos){
   int min = Integer.MAX_VALUE; // fijamos un valor que no puede ser superado

   for (int i=0; i< datos.length; i++){
     for (int j=0; j < datos[i].length; j++){
       if (datos[i][j] < min) min = datos[i][j]; // este pasa a ser el menor
     }
   }
   return min; // devolvemos en valor mínimo de todos
}

public static void main (String [] args){
      int [][]datos = {{0,2,3,4,5},{2,5,7,-1},{},
                       {0,2,-10,-15},{20,30,40}
                      };

      int min = minimo (datos);
      System.out.println ("El valor mínimo es: "+ min);
}
```

Contenidos

- 1. Sentencias condicionales
- 2. Sentencias iterativas
- 3. Recorridos de arrays
- **4. Saltos** y excepciones
- 5. Métodos recursivos

Ruptura de secuencia

- La secuencia de control de un programa se puede romper de tres maneras:
  - Saltos (**break**)
    - alteran el flujo de control **en bucles y switch**
  - Bucle parcial (**continue**)
    - no se ejecuta todo el cuerpo del bucle **en esa iteración**
  - Fin de método (**return**)
    - acaban la ejecución de un método prematuramente
  - Excepciones (**Exception**)
    - alteran el flujo de control cuando se produce un error

- Conviene evitar estas rupturas de programa, pues dificultan su legibilidad

break

- Vámonos **directamente a la salida**
  - Rompe la lógica del bucle, saliéndonos fuera
  - Útil para bucles que se hacen n + 1/2 veces
- Utilizable en {}, while, do/while, for y switch

```
int n= 0;
while (true) {
    ...print(n);
    if (n == 5) break;
    n++;
}
….. ;
```

```
for (int n= 0; ; n++)
    if (n == 5)
        break;
…. ;
```

Bucles n + 1/2

Leer la entrada estándar hasta que se escriba una línea con sólo un punto

```
public static void main (String[] args) {
   System.out.println ("Lee líneas hasta una línea punto .");
   Scanner teclado =new Scanner (System.in);
   int cuenta = 0;
   while (true) {
        String linea = teclado.next();
        if (linea.equals(".")) {
                    break;
        }
        cuenta = cuenta + 1;
        System.out.println (cuenta + ": " +linea);
   }
}
```

Se ejecuta siempre

No se ejecuta la última vez

continue

- La secuencia **continue**, dentro de un bucle, comienza inmediatamente la **siguiente iteración del ciclo**

```
for (int n= 0; n < 100; n++) {
    if (n % 2 == 0)
        continue;
    System.out.print(n);
}
```

Saltos etiquetados

- **break** causa la salida del bucle más interno (cercano), el que lo encierra directamente.
  - Para salirnos de una serie de estructuras anidadas de una sola vez, podemos usar la sentencia **break** etiquetada

```
bucleExterno: for (int x = 0; x < 100; x++)
  for (int y = 0; y < 100; y++)
    for (int z = 0; z < 100; z++)
      if (z*z == (x*x) + (y*y)) {
        System.out.println (z + " " + x + " " y);
        break bucleExterno;
      }
```

- **continue** sigue con la iteración del bucle que lo encierra directamente

return

- Si dentro de un bucle aparece un **return** el método termina sin contemplaciones

```
int buscaPunto (String s) {
    for (int p=0; p<s.length(); p++) {
            char c = s.charAt(p);
            if (c == ‘ . ‘) return p;
    }
    return -1;
} // fin método
```

Contenidos

- 1. Sentencias condicionales
- 2. Sentencias iterativas
- 3. Recorridos de arrays
- 4. Saltos y **excepciones**
- 5. Métodos recursivos

Errores de ejecución

```
public static void divisionPorCero() {
        int n= 2 / 0;
}
```

```
java.lang.ArithmeticException: / by zero
        at Errores.divisionPorCero(Errores.java:10)
```

Errores de ejecución

```
public static void objetoInexistente() {
        String s;
        boolean esManuel = s.equals("Manuel");
}
```

```
java.lang.NullPointerException
        at Errores.objetoInexistente(Errores.java:16)
```

Errores de ejecución

```
public static void arraySinCrear() {
        int[] vector;
        int n= vector[0];
}
```

```
java.lang.NullPointerException
        at Errores.arraySinCrear(Errores.java:21)
```

Errores de ejecución

```
public static void indiceDesbordado() {
        int[] vector= new int[10];
        int n= vector[10];
}
```

```
java.lang.ArrayIndexOutOfBoundsException: -1
        at Errores.indiceDesbordado(Errores.java:30)
```

Errores de ejecución

```
public static void arrayImposible() {
        int[] vector= new int[-1];
}
```

```
java.lang.NegativeArraySizeException
        at Errores.arrayImposible(Errores.java:25)
```

Excepciones (I)

- Las excepciones ofrecen salidas de emergencia
  - de bucles
  - de llamadas a métodos
- Se pretende que el programador se centre en el comportamiento “normal” (si todo va bien)
  - y que la situación excepcional aborte
    - Condición no deseada
    - Condición no tratable
    - Condición no considerada

Excepciones (II)

- Hay unos **objetos especiales**
  - Se clasifican como throwable (arrojables)
  - Primero se crean
    - con o sin mensaje de error como parámetro del constructor

```
new Exception();
new Exception("me ha pasado esto");
```

  - Luego se lanzan

```
throw new Exception("me rindo");
```

  - Después se pueden atrapar al vuelo

```
catch (Exception e) {
   System.err.println(e);
}
```

Excepciones (III)

- Pueden ser lanzadas por un programa
  - Con la sentencia **throw**

```
if ( chequeo_problema ) throw new Exception ("mensaje");
```

    - errores detectados por el programador
    - Errores detectados por el intérprete (división por 0 , null,…)
- Su activación interrumpe la ejecución secuencial
- Se propagan hasta que son capturadas
  - interrumpe toda sentencia que no la capture
- La ejecución continúa si existe cláusula **catch** que acepte la excepción activada

Ejemplo

```
public static double fraccion (int num) throws Exception
{
        return 1.0/num;
}
```

Usa el parámetro del método como dividendo;
si un dividendo fuera 0: explota.

La declaración del método indica que **puede ocurrir** un error y que debería tratarse – o morir!

Ejemplo

```
public static double fraccion (int num) throws Exception {
        if (num == 0) throw new
                Exception ("no se puede dividir por 0");
        return 1.0/num;
}
```

Detecta la división por 0, avisa al que invoca el método mediante una excepción que no puede ser ignorada por el código que ha invocado el método fraccion

El método fraccion **debe declarar que lanza Exception**

Ejemplo

```
public static double fraccionSinExcep (int num) {
        try {
                return fraccion(num);
        } catch ( Exception e ) {
                return 0.0;
        }
}
```

Detecta la Exception y la desactiva. **Ya no se necesita declarar que lanza excepciones.**

Ejemplo

```
public static void imprimeFraccion (int[] num) {
    for ( int i=0; i< num.length; i++) {
        try {
            if (num[i] == 0) throw new
                Exception ("No se puede dividir por 0");
            System.out.println( 1.0/num[i] );
        } catch ( Exception e ) {
            System.err.println(e);
        }
    }
}
```

Detecta la división por 0, avisa y sigue; **NO se declara que lance excepciones**

Try { … } catch (…) { … }

- Dentro del try se ***puede*** lanzar (**throw**) una excepción
  - **Explícitamente o al invocar un método que lance excepciones**
  - Se olvida el resto del código
  - Se ejecuta lo que diga el catch
- Si no se lanza nada, el catch como si no existiera

```
try {
    ...
    throw new Exception();
    ...
} catch (Exception e) {
    …
}
```

Sentencias con posibles errores
Código peligroso!

Notificación y recuperación de los posibles errores

Excepciones

- Error
  - CoderMalfunctionError, FactoryConfigurationError, LinkageError, VirtualMachineError, ...
- Exception
  - ClassNotFoundException, DataFormatException, IOException, NoSuchFieldException, NoSuchMethodException, RuntimeException, UnsupportedAudioFileException, ...
- RuntimeException
  - ArithmeticException, ClassCastException, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, NegativeArraySizeException, NoSuchElementException, NullPointerException, UnsupportedOperationException, ...

Obligación de declarar

- Las excepciones de clase Exception
  - **deben declararse en la cabecera de los métodos** que las puedan lanzar
  - **o ser capturadas** (catch) internamente
- Error
  - no suelen usarse en los programas
- RuntimeError
  - no hay que declararlas en las cabeceras
  - pueden capturarse en un catch
  - se usan para detectar errores de programación

Múltiples catch

```
try {
  ... ... ...
} catch (claseA ida) {
  ... ... ...
} catch (claseB idb) {
  ... ... ...
} catch (classC idc) {
  ... ... ...
}
```

Tipos de excepciones

- Los diferentes ***catch*** se intentan en el orden en que aparecen hasta que uno de ellos casa; después de casar con uno, los demás se olvidan
- Casar significa que la **excepción a cazar es de la clase indicada** o de una clase extensión de ella (aplicándose un *upcasting,* subtipo de…)

try {…}catch(…){… }finally{…}

- Se puede añadir un trozo **finally** que se ejecuta
  - bien cuando acaba el código normal (**try**)
  - bien cuando acaba el código excepcional (**catch**)
  - ... es decir, **SIEMPRE se ejecuta**; incluso si **try** lanza una excepción que no captura ningún **catch**

finally

```
try {
    ... ... ...
    throw new Exception();
    ... ... ...
} catch (Exception e) {
    ... ... ...
} finally {
    ... ... ...
}
```

```
try {
    ... ... ...
    throw new Exception();
    ... ... ...
} catch (Exception e) {
    ... ... ...
} finally {
    ... ... ...
}
```

try {…}catch(…){… }finally{…}

- Puede no haber ningún catch
- try puede estar vacío

```
MailMessage email = new MailMessage(Origen, Destino, Asunto, Mensaje);
SmtpClient smtp = new SmtpClient();

try {
    smtp.Send(email);
    System.out.println ("Mensaje enviado");
}  catch (Exception ex)     {
    System.out.println ("No se ha podido enviar el mensaje);
}  finally     {
    System.out.println ("Fin del proceso de envío de mensajes");
}
```

Intento enviar el mensaje

Informo del envío correcto del mensaje

Informo de que no he podido enviar el mensaje

Siempre informo del final del proceso

Excepciones anidadas

- Sólo puede haber una excepción "**en el aire**" en un momento dado
- Si un **catch** atrapa una excepción, ésta deja de propagarse
- El **catch** puede a su vez lanzar una excepción
  - La vieja es capturada

    catch (Excepcion e) { ...; throw e; }
  - Una nueva se lanza

    catch (Excepcion e) { ...; throw new Excepcion("..."); }

Métodos y excepciones

- Un método puede, en su cuerpo, lanzar una excepción sin saber o poder resolverla
- **Debe indicar en la cabecera que puede lanzar una excepción**
  - nótese que puede lanzarla o no
  - es una **opción**, no una obligación

```
acceso tipo metodo (argumentos) throws TipoDeException
{
    ...
    throw new TipoDeException("mensaje de error");
    ...
}
```

Métodos y excepciones

```
metodo_3 () throws Exception {
    throw new Exception();
}
```

```
metodo_2 () throws Exception {
    metodo_3();
}
```

```
metodo_1 () {
    try {metodo_2();
    } catch (Exception e) { e…; }

}
```

```
main (…) { metodo_1(); }
```

Traza:

llamada al método

Métodos y excepciones

```
metodo_3 () throws Exception {
    throw new Exception();
}
```

```
metodo_2 () throws Exception {
    metodo_3();
}
```

```
metodo_1 () {
    try {metodo_2();
    } catch (Exception e) { e…; }

}
```

```
main (…) { metodo_1(); }
```

Trasiego de la excepción

¿Cuándo usar excepciones?

- Aunque es un mecanismo de control de flujo de ejecución
  - conviene NO usarlo para programar lo que es normal
- Conviene usarlo cuando hay que programar una situación anómala
  - el que lo lea se pondrá en situación mental de analizar una situación excepcional

¿Cuándo usar excepciones?

- Un método lanzará una Exception, que queda documentada en la cabecera, cuando
  - no es responsabilidad del que llama al método, saber si los argumentos son correctos o dan pie a un error
  - es responsabilidad del que llama gestionar la excepción cuando se produzca
  - Se documenta con **@throws**
  - Por ejemplo:
    - Chequeo de argumentos válidos
    - Evitar la creación inconsistente de un objeto

PROGRAMACIÓN INFORMATIVA

¿ … RuntimeExceptionts?

- Un método lanzará una RuntimeException, que **NO** queda documentada en la cabecera, cuando
  - es responsabilidad del que llama al método, saber si los argumentos son correctos o darían pie a un error
  - el que llama no prevé ninguna excepción, ni tratamiento explícito alguno
  - el llamado se protege de fallos del programador lanzando una excepción de ejecución

PROGRAMACIÓN DEFENSIVA

Objetos exception

http://java.sun.com/javase/6/docs/api/java/lang/Exception.html

- Constructores
  - Exception()
  - Exception(String message)
- Métodos
  - String toString()
    - Devuelve texto explicativo de la excepción indicando la clase que la generó
  - String getMessage()
    - Devuelve el mensaje detallado de la excepción
  - void PrintStackTrace()
    - Devuelve la traza del stack en el momento de la excepción

Contenidos

- 1. Sentencias condicionales
- 2. Sentencias iterativas
- 3. Recorridos de arrays
- 4. Saltos y excepciones
- **5. Métodos recursivos**

Recursividad

- Métodos **que se llaman a sí mismos**
  - equivalente a las demostraciones por inducción
  - El resultado para N se calcula en función del resultado para N-1 y así, recursivamente, hasta llegar a un caso 0

$$f(n) = \begin{cases} 1 & n = 0, n = 1 \\ n * f(n-1) & n > 1 \end{cases}$$

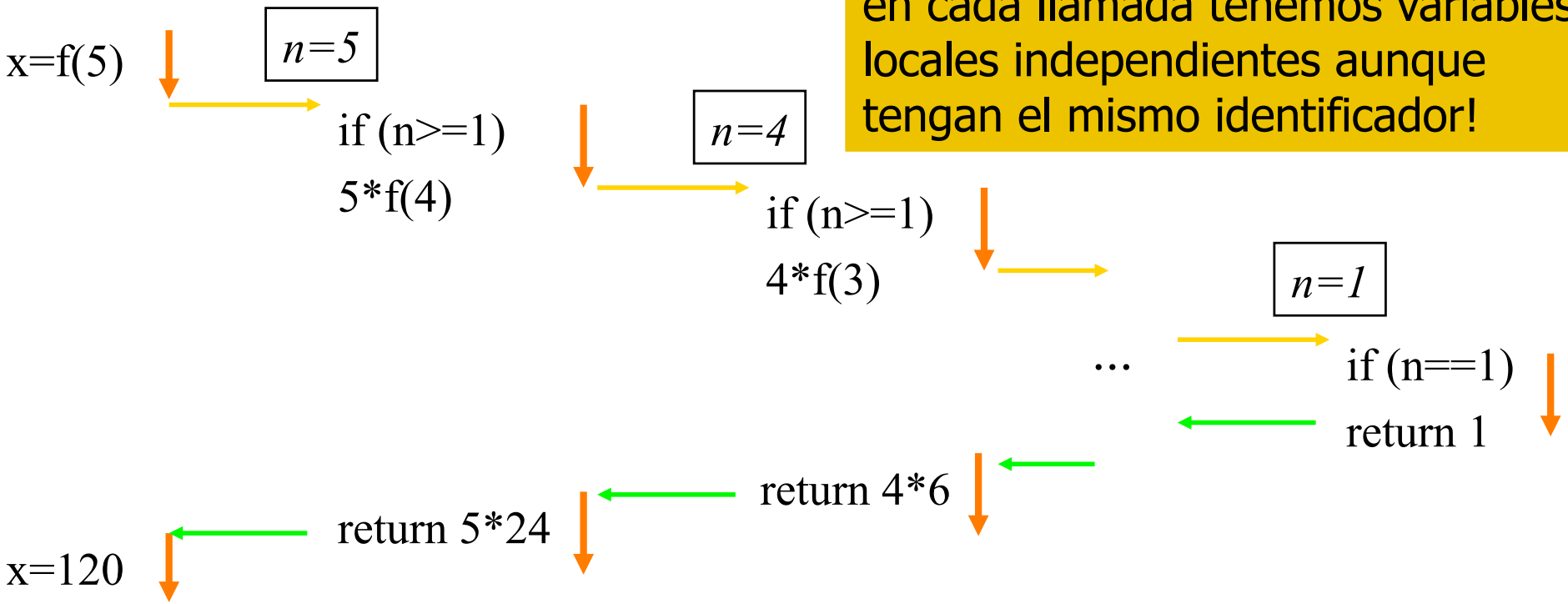
```
int fact (int n) throws Exception {
    if (n < 0)
        throw new Exception
            ("argumento no válido");
    if ( (n == 0) || (n == 1) )
        return 1;
    else
        return n * fact (n-1);
}
```

Recursividad

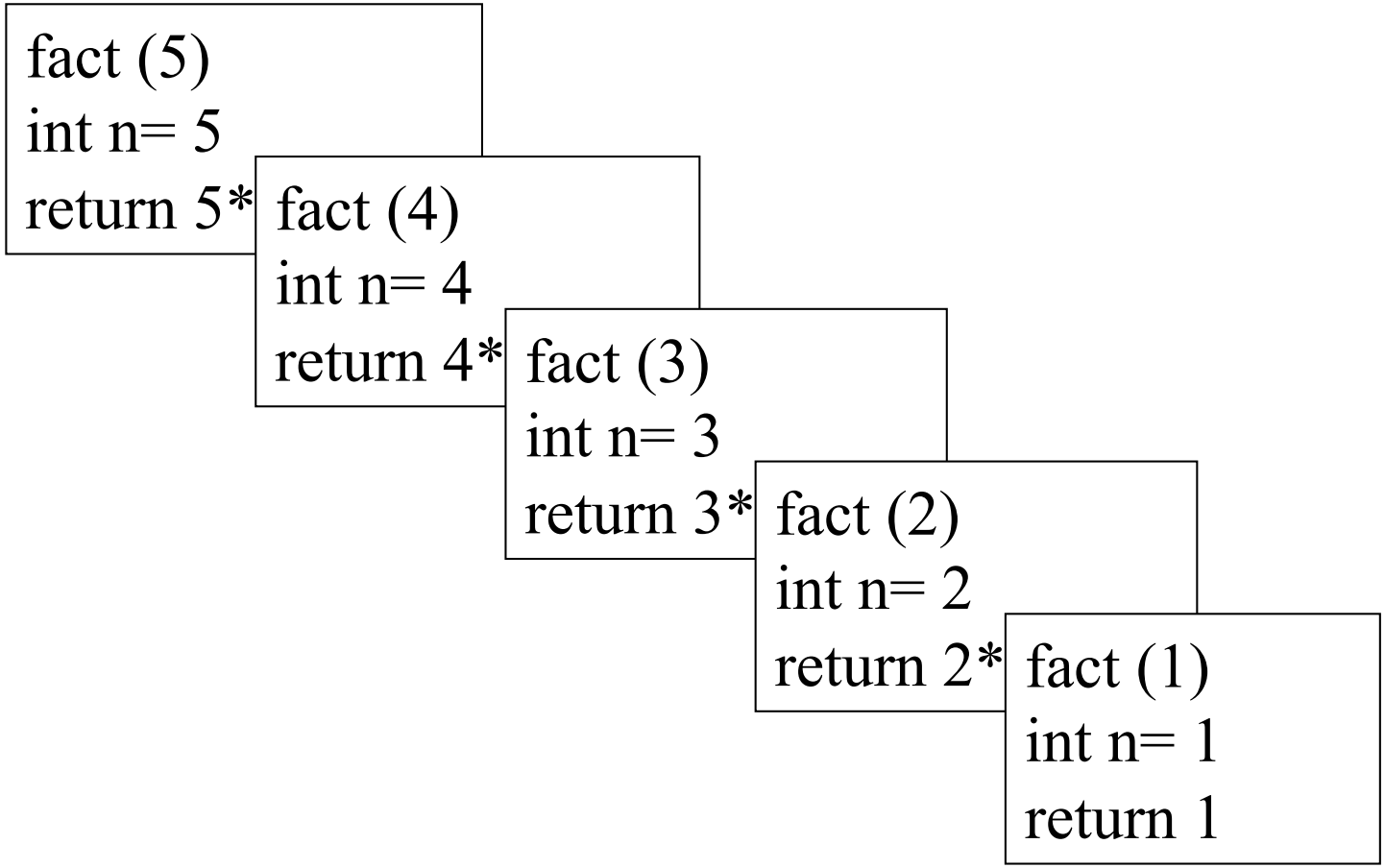
- Algunos problemas admiten una solución recursiva muy sencilla de comprender y de demostrar que es correcta matemáticamente
  - más legibles
  - caso básico + recursividad
- Necesitan
  - Una **condición de parada**
  - **Converger en cada paso** hacia la parada
  - Hay que cerciorarse de que no es un círculo vicioso

Ámbito dinámico (recursivo)

en cada llamada tenemos variables locales independientes aunque tengan el mismo identificador!



Contextos anidados



Limitaciones

- Si se crean muchísimos contextos, llega un momento que no caben en memoria y se produce desbordamiento
  - stack overflow
- Ocurre
  - cuando el problema es demasiado grande para la memoria del sistema
    - número de contextos * número de variables
  - cuando no converge al caso 0
    - error de diseño de la solución