Tema 4: Programación Estructurada

Introducción a la Programación
Grado en Ingeniería Informática, EPI Gijón

{jdiez,oluaces,juanjo}@uniovi.es

Departamento de Informática - Universidad de Oviedo en Gijón

Objetivos

- Comprender los principios de la Programación Estructurada como forma de producir software eficiente y fácil de entender y mantener.
- Conocer el funcionamiento de los esquemas de composición de acciones secuenciales, condicionales (`if-else` o `switch`) e iterativas (`while`, `for` o `do-while`).
- Ser capaz de utilizarlos para hacer programas sencillos.
- Dominar la escritura de expresiones condicionales complejas usando los operadores $\&\&$, $||$ y !.
- Entender los esquemas iterativos básicos y su combinación mediante bucles anidados.
- Ser capaz de identificar el ámbito de una variable y conocer cómo se produce su creación y la liberación de la memoria que ocupa.

Contenidos

Programación Estructurada

¿Qué es y por qué surge?

- En la construcción de los programas se empleaban saltos incondicionales (sentencias **goto**).
- Esto hacía que pudiera ser complicado seguir el flujo de un programa y entenderlo.
- Los programas eran difíciles de depurar y mantener.
- En los años 70 se determinó que había que seguir un forma más *estructurada* de escribir los programas.
- La idea principal es hacer programas imperativos que tengan una estructura lógica, clara y secuencial, sin saltos incondicionales.

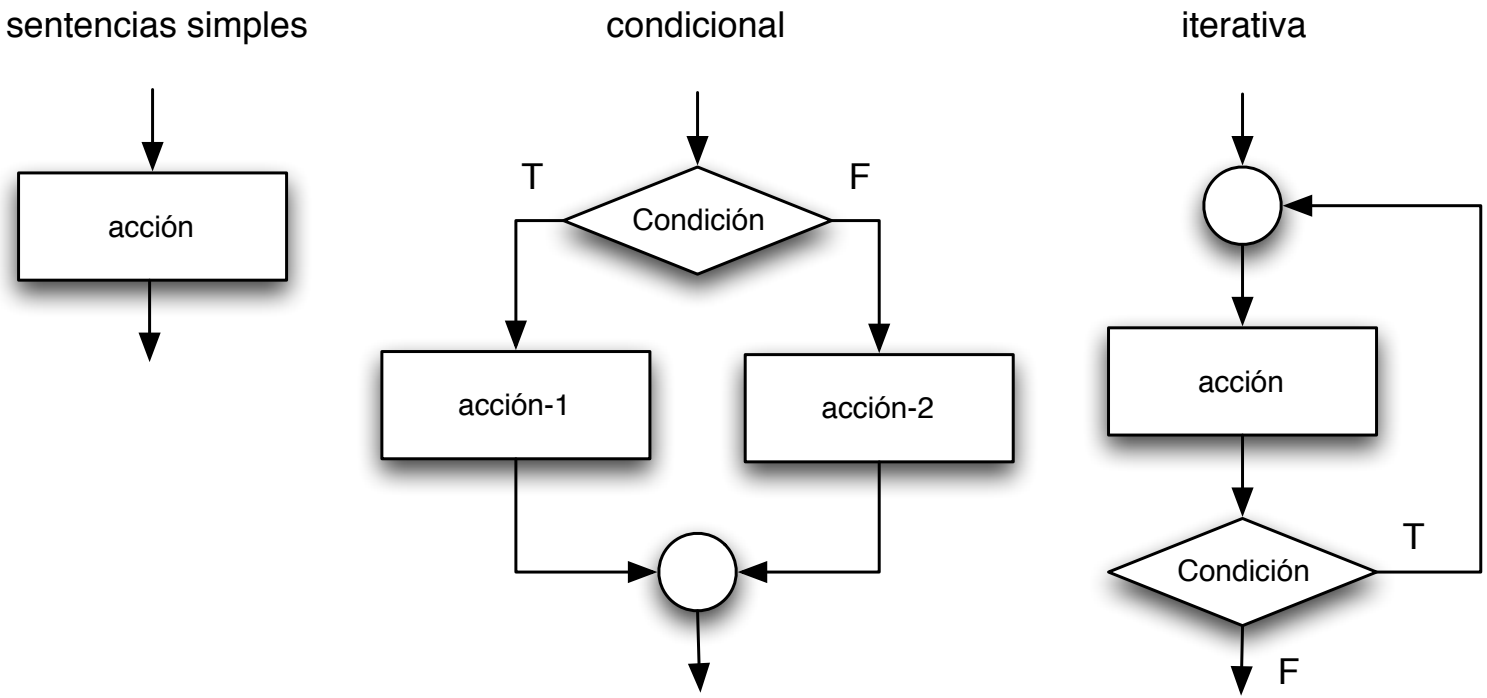
Programación Estructurada

Elementos

- No todos los programas pueden resolverse con un secuencia de acciones simples como hemos hecho hasta ahora.
- Hay veces que queremos que determinadas instrucciones solamente se ejecuten cuando se cumple una cierta condición.
- Del mismo modo, una situación muy común en cualquier programa es que ciertas instrucciones se deban repetir varias veces.
- De ambas situaciones surge la necesidad de emplear sentencias condicionales e iterativas.
- Junto con las sentencias simples, constituyen los tres elementos que se combinan para diseñar cualquier algoritmo.
- Programar es sencillo, ¡solamente tenemos tres tipos de instrucciones!

Tipos de Sentencias

Sentencias simple, condicional e iterativa

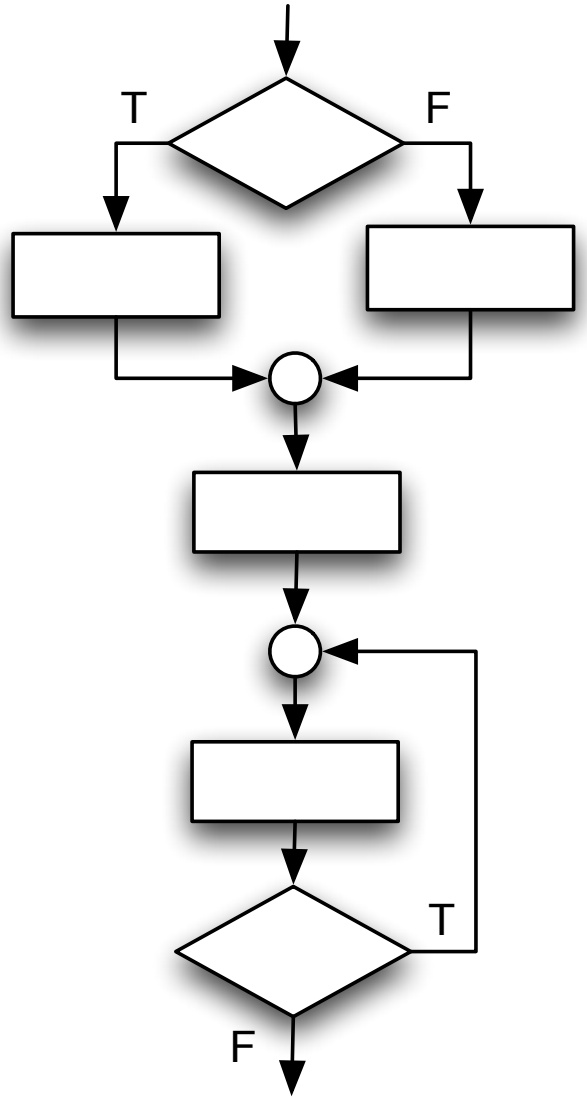


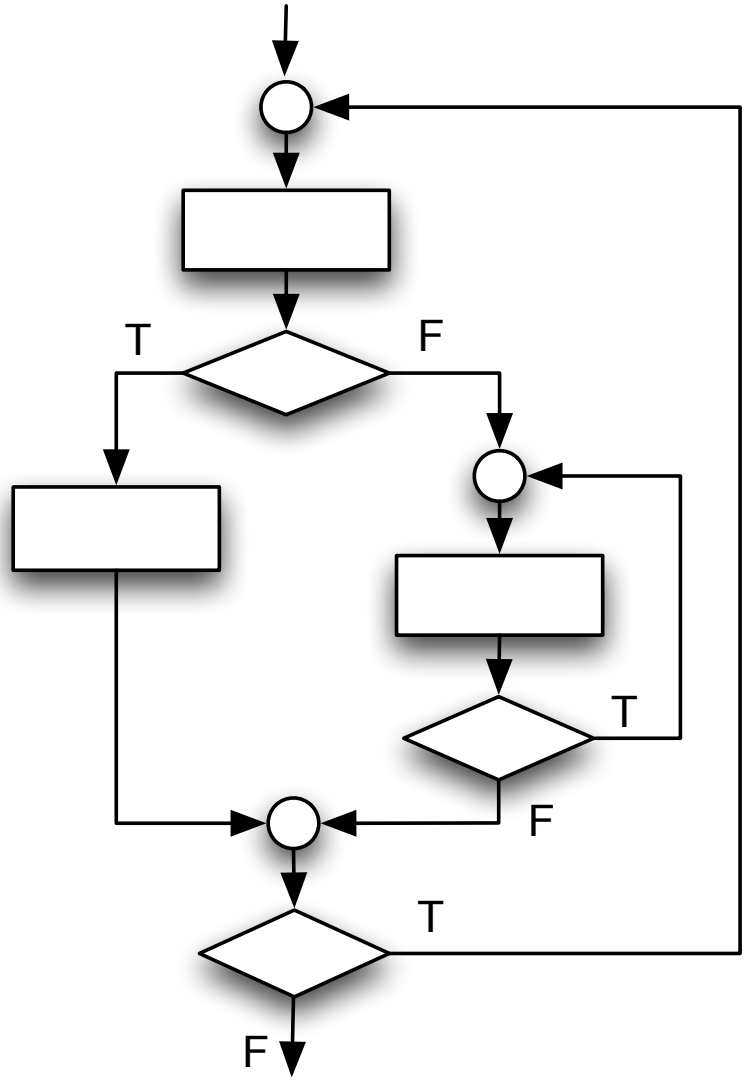
Importante

El teorema del programa estructurado (Böhm-Jacopini) demuestra que todo programa puede escribirse utilizando únicamente sentencias simples, condicionales e iterativas

Formas de combinar sentencias

Secuencias y anidamientos





Programa: Mejorar la clase Círculo

Enunciado y algoritmo

Enunciado

Modificar la clase `Círculo` *de forma que se garantice que el* `radio` *de un objeto de la clase no pueda tomar un valor negativo.*

Algoritmo 1 Método `setRadio()`

Función `setRadio(r: real) : nada`
si `r` ≥ 0 **entonces**
    `radio` $=$ `r`
fin si

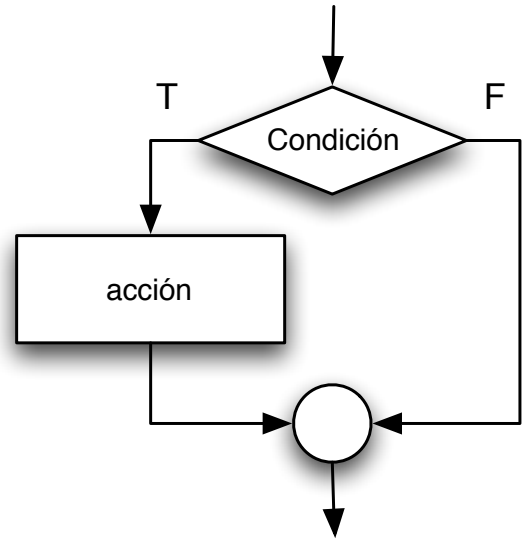
- Aprender lo qué es una sentencia condicional.
- Conocer las sentencias condicionales que tiene Java.
- Saber lo que es una condición.
- Escribir la condición adecuada, dominando los operadores relacionales y lógicos.

Sentencia **if** simple

Se ejecutan acciones cuando se cumple una cierta condición

Sentencia `if` simple

`if` (*condición*) *acción*



- La semántica de la sentencia es que la *acción* solamente se realiza cuando la *condición* es cierta.
- Cuando la *condición* es falsa, la sentencia no hará nada.
- Si la *acción* está formada por una secuencia de acciones, entonces se deben encerrar entre llaves.
- En cambio, cuando solamente tiene una instrucción se debe finalizar con punto y coma (;)

Clase Círculo: método setRadio() seguro

Atributo radio no negativo

```
19      public void setRadio(double r) {
20          if ( r >= 0 )   radio=r;
21      }
```

- La asignación `radio=r` solamente se ejecuta cuando la condición (`r` $>= 0$) es cierta.
- Si la condición es falsa, es decir si `r` es menor que 0, la asignación no se hace.
- El método `setRadio()` garantiza ahora que, en todos los objetos `Círculo`, el atributo `radio` siempre será no negativo.

Importante

Si lo que se quiere lograr es que se ejecute una acción cuando una condición sea falsa, se debe escribir la condición complementaria o negarla con el operador de negación (!)

Condiciones

Expresiones lógicas

Condición

Es una expresión lógica, es decir, una expresión que, o bien produce el valor cierto (`true`), o bien produce el valor falso (`false`).

- Son expresiones que producen valores lógicos (tipo `boolean`).
- Pueden ser:
  1. Expresiones simples, con un único término lógico.
  2. Expresiones compuestas, que combinan expresiones simples usando los operadores lógicos: Y (&&), O (||) y NO (!).
- Hay cuatro clases de expresiones simples:
  1. variables de tipo `boolean`,
  2. expresiones con operadores relacionales,
  3. expresiones con operadores de comparación (o el método `equals()` con objetos), y
  4. métodos que devuelvan un valor lógico.

El tipo boolean

Dos valores: **true** y **false**

- Posibles valores: las constantes `true` y `false`.
- Operadores: los lógicos y los de comparación.
- No permite ninguna clase de conversiones:
  1. No se convierten de forma automática a ningún otro tipo básico.
  2. Del mismo modo, ningún tipo básico se convierte automáticamente a `boolean`.
  3. Es más, tampoco se permiten las conversiones usando el operador de conversión en ninguno de los dos sentidos.
- Este hecho es lo que obliga a que las condiciones sean siempre expresiones lógicas.

Importante

El hecho de que el tipo `boolean` no permita ningún tipo de conversiones obliga a que las condiciones siempre tengan que ser expresiones lógicas

Operadores Lógicos

Combinación de expresiones lógicas

- Permiten combinar varias expresiones lógicas simples.
- Sus operandos deben ser siempre expresiones lógicas, es decir, de tipo `boolean`.
- Los tres más típicos son los operadores Y, O y NO.

Op.	Uso	Devuelve **true** si...	Ejemplo	
&&	op1 && op2	op1 y op2 son ciertos	a && b	**false**
||	op1 || op2	op1 o op2 son ciertos	a || b	**true**
!	! op	op es falso	! a	**false**

p	q	p && q	p || q	! p
F	F	F	F	T
F	T	F	T	T
T	F	F	T	F
T	T	T	T	F

Operaciones en cortocircuito

Eficiencia en las operaciones lógicas

- En una operación lógica Y ú O a veces es posible determinar el resultado sin necesidad de evaluar toda la expresión.
- Por ejemplo:
  1. Si en una operación Y el primer operando es falso, es evidente que el resultado de la operación será falso.
  2. En un operación O, cuando el primer operando es cierto el resultado será cierto también.
- Los operadores && y || hacen uso de esta propiedad para hacer su evaluación más rápida.
- En muchos casos no será necesario evaluar el segundo operador.
- El código generado es más eficiente y el programa se ejecutará más rápido.
- Regla (si podemos determinarlo de antemano): poner primero el operando que con mayor probabilidad sea cierto en una operación || y falso en una operación &&.

Operadores Relacionales

Comparación de orden entre dos operandos

- Son binarios y permiten comparar la relación de orden que se da entre sus dos operandos.
- Se aplican sobre todos los tipos básicos, exceptuando el propio tipo `boolean`.
- Todos esos tipos presentan una relación de orden entre sus valores.
- Si se comparan operandos de distintos tipos básicos se realizarán las conversiones automáticas oportunas.
- Sea cual sea el tipo de los operandos, el valor de la expresión siempre será `boolean`.

Op.	Uso	Devuelve **true** si...	Ejemplo	
$<$	op1 < op2	op1 es menor que op2	$7 < 4$	**false**
$<=$	op1 <= op2	op1 es menor o igual que op2	$4 <= 7$	**true**
$>$	op1 > op2	op1 es mayor que op2	$7 > 4$	**true**
$>=$	op1 >= op2	op1 es mayor o igual que op2	$4 >= 7$	**false**

Un error muy típico

Estamos en Programación, no en Matemáticas

Ejemplo: comprobar si una variable entera `i` está entre 0 y 10:

- Matemáticas: $0 <= \texttt{i} <= 10$
- Programación: $(0 <= \texttt{i})\ \&\&\ (\texttt{i} <= 10)$

¿Por qué no se puede escribir como en Matemáticas?

- Es una expresión y como en todas se aplican las reglas de precedencia y asociatividad conocidas.
- Si utilizáramos la expresión habitual típica en Matemáticas:
  1. Dado que los dos operadores son del mismo grupo y tienen asociatividad ID, primero se hace la comparación $0 <= \texttt{i}$.
  2. Esa expresión producirá un valor `boolean`.
  3. Con lo que en la segunda comparación estaríamos comparando un valor `boolean` con la constante `int` 10.
  4. En Java la expresión produce un error y el programa no se podría ejecutar.

Operadores de Comparación

Comprobar si dos valores son iguales o distintos

Op.	Uso	Devuelve **true** si...	Ejemplo	
==	op1 == op2	op1 y op2 son iguales	$7 == 4$	**false**
!=	op1 != op2	op1 y op2 son distintos	$4 != 7$	**true**

- Comparan la igualdad o desigualdad entre sus dos operandos.
- Producen un valor `boolean`.
- Se pueden aplicar sobre todos los tipos básicos
- Pero:
  1. Hay que ser cautos al hacer comparaciones de igualdad entre valores reales. Es muy difícil que dos valores reales sean exactamente iguales.
  2. Tampoco se suelen emplear con el tipo `boolean`, porque es mejor usar directamente sus valores lógicos.

Se puede escribir:

```
boolean cond= ... ;
if ( cond == true )  ...
```

pero es mejor:

```
boolean cond= ... ;
if ( cond )  ...
```

Comparando objetos

Método equals()

- Se puede usar el operador == con objetos, y en general con cualquier variable de un tipo referenciado.
- Solamente devuelve `true` cuando las dos variables referencian el mismo objeto.
- Pero, dos objetos pueden ser iguales sin ser exactamente el mismo objeto.
- Ejemplo: dos Círculos con el mismo `radio` son iguales.
- Para poder evaluar la igualdad de objetos de una nueva clase se debe escribir un método llamado `equals()`, que recibe un objeto y devuelve un valor `boolean`.

```
29      /**Devuelve cierto si dos objetos Círculo son iguales
30       * @param c el objeto con el que se va a comparar
31       * @return true si el radio de los dos objetos es igual*/
32      public boolean equals(Círculo c) {
33          return getRadio() == c.getRadio();
34      }
```

Programa: Máximo de dos Números

Enunciado y algoritmo

Enunciado

Dados dos números reales leídos de teclado, realizar un programa que imprima el máximo valor de ambos.

Algoritmo 2 Método `Máximo2()`

```
Función Máximo2 (a: real, b: real) : real
si a > b entonces
    retorna a
sino
    retorna b
fin si
```

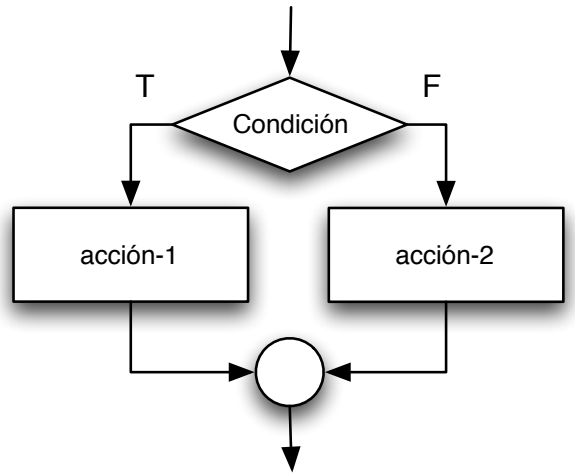
- En este caso, necesitamos una sentencia condicional que haga también algo cuando la condición sea falsa.
- Es un `if` en su forma más general, con parte *entonces* y *sino*.

Sentencia **if-else**

Se ejecutan distintas sentencias en función del valor de la condición

Sentencia `if-else`

`if` (*condición*) *acción-1*
`else` *acción-2*



- La semántica de la sentencia es que se ejecuta la *acción-1* cuando la *condición* es cierta y la *acción-2* cuando es falsa.
- Nunca se ejecutarán las dos acciones, pero siempre se ejecutará una de las dos.
- Como en los `if` simples, si la acción de cada parte es única se finaliza en punto y coma (;) y si es una secuencia se pone entre llaves.

Máximo de dos Números

Código fuente

Método:

```
 7     /**Calcula el valor máximo de dos números reales
 8      * @param a primer número real
 9      * @param b segundo número real
10      * @return el valor máximo de a y b */
11     public static double Máximo2(double a, double b) {
12         if ( a > b )  return a;
13         else          return b;
14     }
```

Programa principal:

```
26         double num1, num2;

28         Scanner teclado= new Scanner(System.in);

31         num1=teclado.nextDouble();
32         num2=teclado.nextDouble();

34         System.out.printf("Máximo: %f\n",Máximo2(num1,num2));
```

Operador (? :)

No es un **if**, es un operador

Operador (? :)

(*condición* ? *expresión-1* : *expresión-2*)

- Es un operador ternario (el único del lenguaje).
- La semántica del operador es que la expresión total produce el valor de la *expresión-1* cuando la *condición* es cierta y el de la *expresión-2* cuando es falsa.
- Importante: NO está pensado para escribir sentencias condicionales con acciones.
- En las dos expresiones no se ponen acciones, sino expresiones que producen un valor.
- Ambas expresiones suelen ser del mismo tipo.

```
20    public static double Máximo2bis(double a, double b) {
21        return ( a > b ? a : b );
22    }
```

Programa: Máximo de tres Números

Enunciado y algoritmo

Enunciado

Dados tres números reales leídos de teclado, realizar un programa que imprima el máximo valor de ellos.

Algoritmo 3 Método `Máximo3()`

```
Función Máximo3 (a: real, b: real, c:real) : real
si a > b entonces
    si a > c entonces  retorna a
    sino  retorna c
    fin si
sino
    si b > c entonces  retorna b
    sino  retorna c
    fin si
fin si
```

Sentencias condicionales anidadas

Se van descartando casos

- La idea es ir descartando casos.
- Los `if` anidados se van aprovechando de los anteriores y necesitan condiciones más simples.

```
21    public static double Máximo3(double a, double b, double c) {
22        if ( a > b )  {
23            //b no es el mayor, no hace falta comparar b y c
24            if ( a > c )  return a;
25            else          return c;
26        }
27        else {
28            //a no es el mayor, no hace falta comparar a y c
29            if ( b > c )  return b;
30            else          return c;
31        }
32    }
```

Reutilizar

Un método se puede escribir usando uno más simple

- En cada paso lo que se hace es comparar entre dos de los números cuál es el mayor.
- Lo hacemos dos veces: una en el primer `if-else` y otra en el `if-else` anidado.
- Mirado de una forma más abstracta, el código anterior podría resumirse en:
  1. Primero miramos entre dos de ellos cuál es el mayor, y
  2. luego, comparamos ese mayor con el tercero.
- Lo mismo podría lograse con dos llamadas encadenadas al método `Máximo2()`.

```
39      public static double Máximo3bis(double a, double b, double c) {
40          return ( Máximo2( Máximo2(a,b) , c) );
41      }
```

Sentencias **if-else** anidadas

¿Con qué **if** va cada **else**?

Importante

Un `else` *siempre va con el* `if` *más próximo de su bloque de código que no tenga otro* `else` *asociado*

```
1 if (a <= b)
2     if (c < d) a = b-2;
3     else if (d < e) b = b+3;
4          else if (e > f) c = a+5;
```

```
1 if (a <= b)
2     if (c < d) a = b-2;
3     else if (d < e) b = b+3;
4 else if (e > f) c = a+5;
```

¿Con qué if va el último else?

La forma de sangrar el programa no influye, se aplica la regla anterior y en ambos casos va con el `if` de la penúltima línea.

```
1 if (a <= b)
2     if (c < d) a = b-2;
3     else { if (d < e) b = b+3; }
4 else if (e > f) c = a+5;
```

```
1 if (a <= b) {
2     if (c < d) a = b-2;
3     else if (d < e) b = b+3;
4 } else if (e > f) c = a+5;
```

Programa: Calcular una tarifa de autobus

Enunciado

Enunciado

Una compañía de autobuses cubre los trayectos entre Madrid y las principales ciudades del norte de España. Sus tarifas varían según el destino y la edad del viajero, ofreciendo precios más reducidos para jóvenes y jubilados. El precio de cada billete se rige por la tabla que aparece a continuación. Diseñar un programa que dada la edad del viajero y el destino (leído como un carácter, la inicial de la ciudad: C / G / S / B), imprima la tarifa a aplicar.

	menor de 18	entre 18 y 64	mayor de 64
Coruña	30	35	30
Gijón	25	30	25
Santander	25	30	25
Bilbao	30	35	30

Tipo **char**

Cómo leer valores de tipo **char** de teclado

- La clase `Scanner` NO tiene un método `nextChar()` que lea el siguiente carácter.
- Para leerlo debemos usar dos métodos consecutivamente:
  1. el método `next()` que lee el siguiente *token* o elemento.
  2. el método `charAt()` que permite obtener un carácter de ese elemento. El primer carácter tiene como índice 0 y no 1.

```
33      public static void main(String[ ] args) {
34          char destino; //letra indicando el destino
35          int edad;     //edad del viajero
36          //Objeto Scanner asociado con el teclado
37          Scanner teclado= new Scanner(System.in);
38          //Leemos los datos del viaje
39          System.out.print("Introduce el destino (C/G/S/B): ");
40          destino=teclado.next().charAt(0);
41          System.out.print("Introduce la edad: ");
42          edad=teclado.nextInt();
43          //Mostramos la tarifa en pantalla
44          System.out.printf("Tarifa: %f\n",Tarifa(destino,edad));
45      }
```

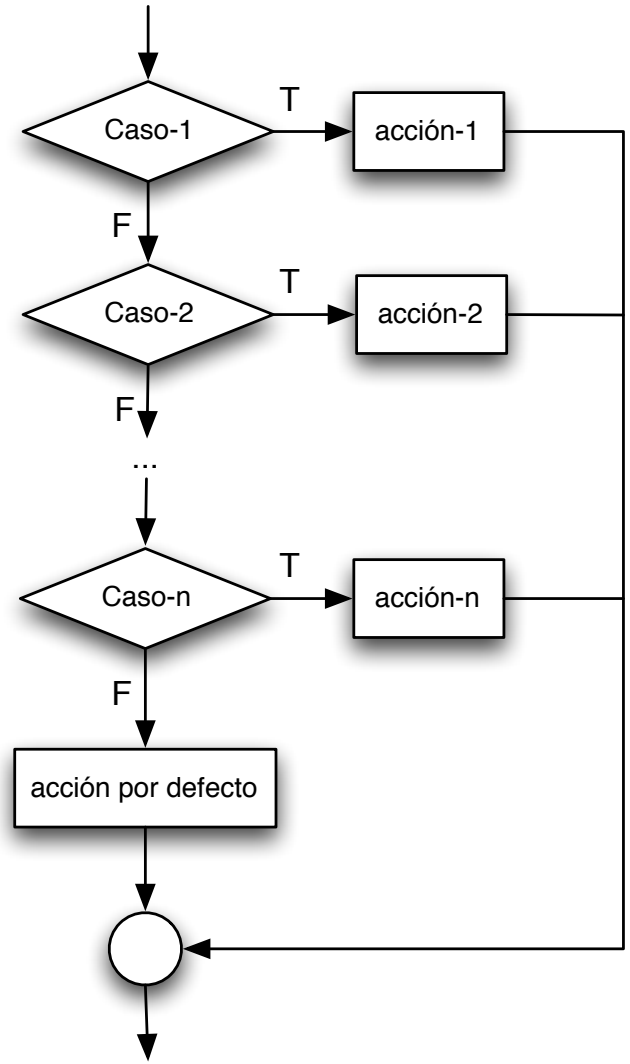
Programa: Calcular la tarifa de un viaje en bus

Método Tarifa()

```
15      /**Calcula la tarifa
16       * @param d letra indicando el destino del viaje
17       * @param e edad del viajero
18       * @return la tarifa aplicable para ese destino/viajero */
19      public static double Tarifa(char d, int e) {
20          if ( d=='G' || d=='S' ) {
21              //Destino Gijón o Santander
22              if  ( e >= 18 && e <= 64 )  return 30;
23              else                        return 25;
24          }
25          else {
26              //El destino es Coruña o Bilbao
27              if  ( e < 18 || e > 64 ) return 30;
28              else                     return 35;
29          }
30      }
```

Sentencia **switch**

Otra forma de escribir un conjunto de **if-else** anidados



- Hay un conjunto más o menos numeroso de casos distintos, y
- cada caso requiere hacer acciones diferentes.
- Los casos se comprueban en orden.
- Si todos los casos fallan, al final puede hacerse una acción por defecto:
  1. dar un error, o
  2. tratar el caso más numeroso.

Programa: Mejorar la Clase Fecha

Enunciado

Enunciado

Modificar la clase Fecha de forma que se garantice que un objeto de la clase siempre tenga una fecha válida. Se entenderá por una fecha válida: un año positivo, un mes entre 1 y 12 y un número de día correcto de acuerdo con el resto de atributos.

- Garantizar que los tres atributos de un objeto de la clase `Fecha` sean correctos.
- Los métodos `set()` deben comprobar que el atributo correspondiente se está actualizando con un valor válido.
- Para el atributo `día` hay distintos valores límite en función del valor del `mes` y del `año`.
- ¿Cómo programar el método `setFecha()`, teniendo en cuenta que para determinar si una fecha es correcta unos campos dependen de otros?

Clase Fecha: métodos set() seguros

Código fuente

```
38      public void setFecha(short d, short m, short a) {
39          //Fijamos las fechas en este orden, para que al fijar el
40          //día, sepamos si es correcto de acuerdo al nuevo año y mes
41          setAño(a);
42          setMes(m);
43          setDía(d);
44      }
```

```
86      public void setMes(short m) {
87          if ( m>=1 && m<=12 ) mes=m;
88      }

99      public void setAño(short a) {
100         if ( a > 0 ) año=a;
101     }
```

Método setDía()

Uso de una sentencia **switch**

```
55      public void setDía(short d) {
56          if ( ( d >= 1 ) && ( d <=31) ) {
57              //Sólo cambiamos el día si d está entre 1 y 31
58              short días_mes; //variable para calcular los días que tiene el mes
59              switch ( getMes() ) {
60                 case FEBRERO:    //28 ó 29 días
61                                  días_mes = (short) (esBisiesto() ? 29 : 28);
62                                  break;
63                 case ABRIL:
64                 case JUNIO:
65                 case SEPTIEMBRE:
66                 case NOVIEMBRE:  //meses de 30 días
67                                    días_mes=30;
68                                    break;
69                 default: //Es un mes de 31 días
70                           días_mes=31;
71              }
72              //cambiamos el día si es menor o igual que el día máximo del mes
73              if ( d <= días_mes ) día=d;
74          }
75      }
```

Sentencia **switch**

Acciones para un conjunto de casos múltiple

Sentencia `switch`

```
switch ( expresión ) {
    case valor-1: acción-1; break;
    case valor-2: acción-2; break;
    ...
    case valor-n: acción-n; break;
    default: acción-por-defecto;
}
```

1. La *expresión* debe ser de un tipo básico *discreto* (tipos enteros y carácter), de la clase `String` (Tema 5) o de un tipo enumerado (Tema 6).
2. Los *valores* de cada caso deben ser expresiones del mismo tipo que la *expresión* del `switch`.
3. No hace falta usar llaves aunque la *acción* asociada con un caso sea una secuencia de instrucciones.

Semántica de la sentencia **switch**

¿Cómo se ejecuta una sentencia **switch**?

- Se comprueba por orden cuál de los *valores* de los distintos casos coincide con el valor de la *expresión*.
- Cuando un caso es igual, se ejecutan todas la sentencias que estén escritas a partir de ese punto.
- Por ello se suele poner una sentencia `break` al final de las acciones de cada caso.
- Al final puede incluirse un caso por defecto, que se aplicaría cuando ninguno de los otros casos son iguales a la *expresión*.

```
switch (n) {
    case 3: System.out.print("*");
    case 2: System.out.print("*");
    case 1: System.out.print("*");
}
```

¿Cuántos asteriscos se imprimen en cada caso?

Método setDía()

Otras consideraciones sobre la sentencia **switch**

- Habitualmente se usan constantes para indicar los casos.

```
13      public static final short ENERO=1;
14      public static final short FEBRERO=2;
        ...
24      public static final short DICIEMBRE=12;
```

- Empleamos el caso por defecto para agrupar el caso más numeroso.
- No usamos llaves para delimitar las acciones de cada caso, a pesar de que los dos primeros tienen dos acciones.
- La sentencia `switch` puede parecer una sentencia *no estructurada*: el flujo *salta* al caso igual y después *salta* a la instrucción siguiente al `switch` con la sentencia `break`.

Importante

Cualquier `switch` *podría reescribirse usando sentencias* `if-else`

Sentencias iterativas o bucles

Necesidad y definiciones

- Ejemplos:
  1. Calcular la suma de 10 números leídos de teclado.
  2. Abonar los intereses en todas las cuentas de un banco.
- En ambos casos necesitamos una sentencia que nos permita repetir varias veces ciertas acciones.

Bucle

Sentencia que permite repetir un conjunto de acciones un cierto número de veces.

También se puede denominar **Iteración**, aunque este término se suele emplear para referirse a cada una de las veces que se repiten las acciones que contiene un bucle.

Sentencias iterativas o bucles

Características

- Permiten hacer varias veces las mismas acciones.
- Las acciones internas se repiten mientras la condición del bucle sea cierta.
- Las acciones del bucle deben cambiar la/s variable/s que intervenga/n en la condición de forma que ésta sea falsa en algún momento.
- Si la condición nunca es falsa, el bucle es infinito.

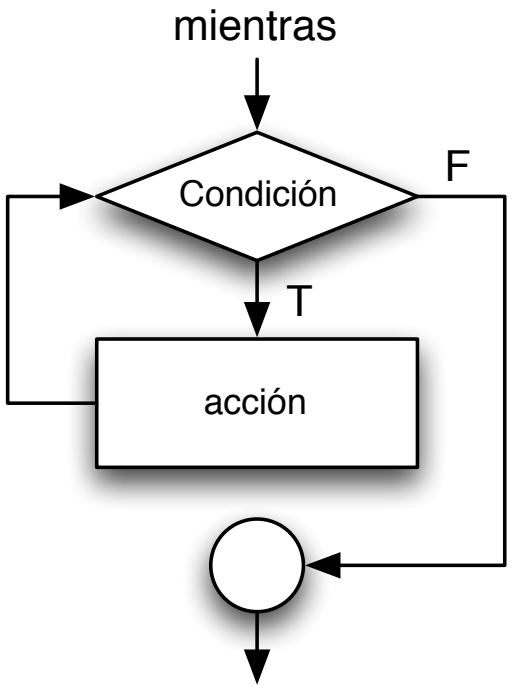
Importante

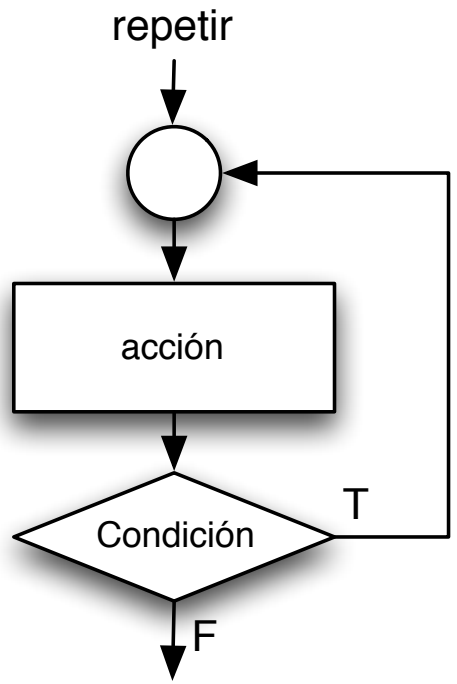
Cuando se escribe un bucle se debe garantizar siempre que en algún momento la condición será falsa, si no el bucle será infinito

Sentencias iterativas o bucles

Tipos de bucles

1. **Mientras** (`while` y `for`): la acción se repite 0 o más veces.
2. **Repetir** (`do-while`): la acción se repite 1 o más veces





Programa: Sumar una secuencia de nº pares

Enunciado y algoritmo

Enunciado

Hacer un programa que dado un número entero positivo calcule la suma de todos los números pares comprendidos entre 2 y ese número (ambos incluidos).

Algoritmo 4 Suma de números pares

```
Leer número de teclado
par=2;
suma = 0
mientras par<=número hacer
    suma = suma + par
    par = par+2
fin mientras
Imprimir suma
```

Programa: Sumar una secuencia de nº pares

Código fuente

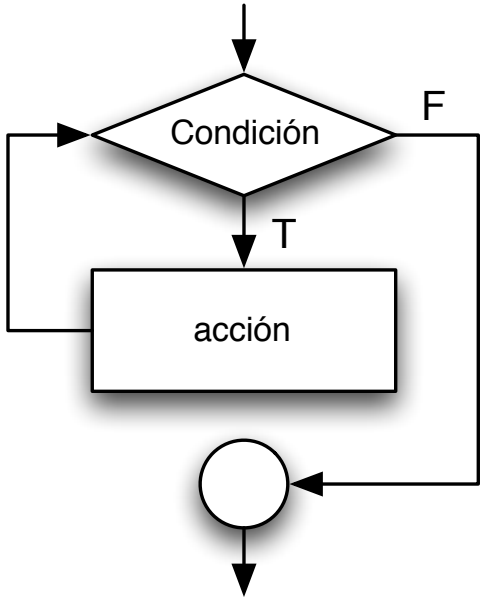
```
 6 public class SumaNúmerosPares {

 8     public static void main(String[ ] args) {
 9         //Objeto Scanner asociado con el teclado
10         Scanner teclado= new Scanner(System.in);
11         //Declaramos una variable entera para leer el número
12         int número;
13         //Leemos el nº entero
14         System.out.print("Introduce un entero:");
15         número=teclado.nextInt();
16         //Una variable
17         int par=2;  //para ir recorriendo los pares
18         int suma=0; //para calcular la suma
19         while ( par <= número ) {
20             suma+=par;
21             par+=2;
22         }
23         //Mostramos la suma en la pantalla
24         System.out.printf("La suma es %d\n",suma);
25     }
26 }
```

Sentencia **while**

Acciones que se repiten mientras una condición sea cierta

Sentencia `while`

`while` (*condición*)
acción



- Semántica: la *acción* se repite *mientras* la *condición* sea cierta.
- Número de ejecuciones: 0 o más veces.
- No es seguro que la *acción* se haga alguna vez, ya que la *condición* puede ser falsa la primera vez.
- Como siempre, si la *acción* está compuesta de varias instrucciones se deben poner llaves.

Operadores de asignación aritméticos

Operación aritmética + una asignación

- Permiten hacer dos operaciones en una, primero una operación aritmética y después una asignación.
- La variable sobre la que se realiza la asignación es el primer operando de la operación aritmética.
- Como en todas las asignaciones, el valor que devuelve la expresión es el valor que se asigna a la variable de la izquierda (left-value).

Op.	Uso	Equivalencia	Ejemplo (a= 5)	
$+=$	op1 $+=$ op2	op1 $=$ op1 $+$ op2	a$+=7$	12
$-=$	op1 $-=$ op2	op1 $=$ op1 $-$ op2	a$-=7$	-2
$*=$	op1 $*=$ op2	op1 $=$ op1 $*$ op2	a$*=7$	35
$/=$	op1 $/=$ op2	op1 $=$ op1 $/$ op2	a$/=7$	0
$\%=$	op1 $\%=$ op2	op1 $=$ op1 $\%$ op2	a$\%=7$	5

La sentencia **break**

Saltos incondicional a la siguiente sentencia. Prohibido usarla

Sentencia `break`

```
break;
```

- Finaliza el bucle y pasa a la siguiente instrucción.
- Tras ejecutar un `break` no se harán:
  1. ni las acciones entre el `break` y el final del bucle,
  2. ni se volverá a evaluar la condición.
- Se suele poner dentro de una sentencia condicional, de forma que se sale del bucle bajo una condición.
- Sería equivalente a incluir esa condición negada en la condición del bucle (es preferible).

Importante

Si una instrucción `break` no se pone dentro de una sentencia condicional, el bucle haría una sola iteración

La sentencia **continue**

Salto incondicional a la condición del bucle. Prohibido usarla

Sentencia `continue`

```
continue;
```

- Provoca un salto incondicional a la condición del bucle.
- Es decir, tras un `continue` lo siguiente sería evaluar la condición del bucle de nuevo.
- No se haría ninguna acción que esté entre `continue` y el final del bucle.
- Se suele poner también dentro de una sentencia condicional.

Importante

Si una instrucción `continue` no se pone dentro de una sentencia condicional, las acciones entre el `continue` y el final del bucle nunca se ejecutarían

break y continue: prohibidos

Dos ejemplos de como escribir bucles alternativos

- No emplear `break`: añadir la condición negada del `break` a la condición del bucle.

```
while ( n >= 0 ) {
    ...
    if ( n%2 == 0 ) break;
}
```

```
while (( n >= 0 ) && ( n%2 != 0 )) {
    ...
}
```

- No emplear `continue`: añadir una sentencia condicional que incluya las acciones posteriores a `continue`.

```
while ( n >= 0 ) {
    ...
    if ( n%2 == 0 ) continue;
    ... //acciones posteriores
}
```

```
while ( n >= 0 ) {
    ...
    if ( n%2 != 0 ) {
        ... //acciones posteriores
    }
}
```

Programa: Media de 10 números reales

Enunciado y algoritmo

Enunciado

Hacer un programa que lea 10 números reales de teclado y calcule su media.

1. El bucle debe hacer exactamente 10 iteraciones, y
2. en cada iteración debemos leer un nº y sumarlo

Algoritmo 5 Media de 10 números reales

```
suma = 0
contador = 1
mientras contador ≤ 10 hacer
    Leer número de teclado
    suma = suma + número
    contador = contador + 1
fin mientras
Imprimir suma/10
```

Programa: Media de 10 números reales

Código fuente

```
 5 public class Media10Reales {

 7     public static void main(String[ ] args) {
 8         //Objeto Scanner asociado con el teclado
 9         Scanner teclado= new Scanner(System.in);
10         //Declaramos una variable real para leer los números
11         double número;
12         //y otra para ir calculando su suma
13         double suma=0;
14         System.out.print("Introduce 10 números reales:");
15         for (int i=1; i<=10; i++) {
16            número=teclado.nextDouble();  //leemos el siguiente
17            suma += número;               //sumamos
18         }
19         //Mostramos la media en la pantalla
20         System.out.printf("Su media es %f\n",suma/10);
21     }
22 }
```

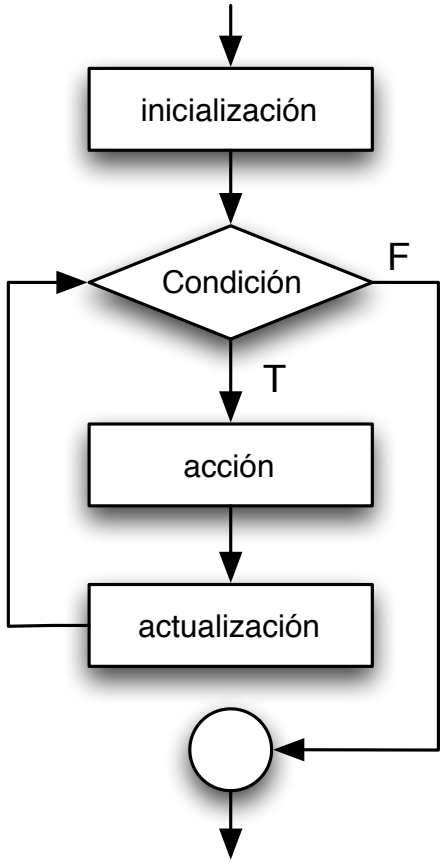
Sentencia **for**

Otra forma de escribir un bucle **while**

Sentencia `for`

```
for ( inicialización; condición; actualización )
        acción
```

- Está pensado para bucles en los que se "cuenta" el número de repeticiones o se recorren secuencias de números.
- Tiene 3 partes:
  1. *inicialización*: de la variable con la que se controla el número de iteraciones.
  2. *condición*: para controlar si se ha alcanzado el número de iteraciones.
  3. *actualización*: acciones para actualizar la variable de control.
- Ninguna es obligatoria.



Pre- y pos-, incremento y decremento

Incrementar o decrementar en una unidad una variable

Se pueden usar con enteros y reales y producen dos cosas:

1. Un incremento o decremento en una unidad de la variable que aparezca en la expresión.
2. Un valor de la expresión en su conjunto. Dependerá de la posición del operador:
   - Si el operador va antes (pre-), devolverá el valor de la variable incrementada o decrementada.
   - Si va después (pos-), devolverá el valor que tenía la variable antes de que se haga el incremento o decremento.

Op.	Uso	Descripción	Ejemplo (a$=5$)	
$++$	$++$op	Preincremento	$++$a	6 (a$=6$)
$++$	op$++$	Posincremento	a$++$	5 (a$=6$)
$--$	$--$op	Predecremento	$--$a	4 (a$=4$)
$--$	op$--$	Posdecremento	a$--$	5 (a$=4$)

Otras consideraciones sobre for

Detalles sintácticos

- Tanto la parte de *inicialización* como la de *actualización* permiten más de una acción. Se separan por comas.

```
for (int i=1, j=10; i < j; i++, j--)    ...
```

- Ninguna de las partes es obligatoria.

```
int i=1;
for ( ; i < j ; i++) {
    ...
}  //i se puede usar tras el for
```

```
for ( ;  ; ) {
    ...
    if ( ... ) break;
}
```

- Una sentencia for se podría escribir con while y viceversa.

```
//Suma de números pares con for
int suma=0;
for (int par=2; par<=número; par+=2 )
   suma+=par;
```

```
//Media de 10 reales con while
int i=1, suma=0;
while ( i <= 10 ) {
   número=teclado.nextDouble();
   suma+=número;
   i++;
}
System.out.printf("Media %f",suma/10);
```

Programa: Contar los dígitos de un nº entero

Enunciado y algoritmo

Enunciado

Hacer un programa que dado un número entero leído de teclado, imprima el número de dígitos que tiene.

1. Un bucle que al menos se repite una vez, y
2. que hace un número de iteraciones que desconocemos.

Algoritmo 6 Contar los dígitos de un nº entero

```
Leer número de teclado
dígitos = 0
hacer
    dígitos = dígitos + 1
    número = número / 10
mientras número > 0
Imprimir dígitos
```

Programa: Contar los dígitos de un nº entero

Código fuente

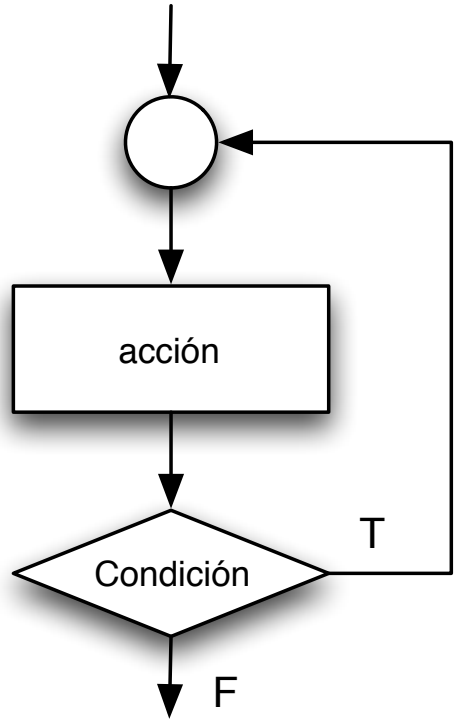
```
 5 public class ContarDígitos {

 7     public static void main(String[ ] args) {
 8         //Objeto Scanner asociado con el teclado
 9         Scanner teclado= new Scanner(System.in);
10         //Declaramos una variable entera para leer el número
11         int número;
12         //Leemos el nº entero
13         System.out.print("Introduce un entero:");
14         número=teclado.nextInt();
15         //Una variable para ir contando sus dígitos
16         int dígitos=0;
17         do {
18             dígitos++;
19             número/=10;
20         }
21         while ( número != 0 );
22         //Mostramos el nº de dígitos la pantalla
23         System.out.printf("Tiene %d dígitos\n",dígitos);
24     }
25 }
```

Sentencia **do while**

Acciones que se repiten una o más veces

Sentencia `do-while`

```
do
    acción
while ( condición );
```



- Semántica: la *acción* se repite mientras la *condición* sea cierta.
- Número de ejecuciones: 1 ó más veces.
- Si la *acción* está compuesta de varias instrucciones se deben poner llaves.

Esquemas iterativos típicos

Esquemas con bucles

1. **Tratamientos de secuencias de elementos**
   - Se caracterizan por hacer el mismo tratamiento con todos los elementos de una secuencia.
   - Hay que identificar la **secuencia**:
     - Secuencias descritas por enumeración.
     - Secuencias de longitud conocida.
     - Secuencias delimitadas por un valor centinela.
2. **Búsquedas asociativas**
   - También tenemos una secuencia de elementos,
   - pero no tratamos los elementos, sino que
   - **se busca el primer elemento** de la secuencia que cumpla una **propiedad**.

Secuencias

Debes identificar la secuencia

Secuencia

Sucesión ordenada de elementos que un programa procesará en ese mismo orden.

Identificar la secuencia:

1. saber qué elementos componen la secuencia,
2. en qué orden están, y
3. cuándo se acaba la secuencia.

Diseñar el bucle (tres operaciones):

1. inicializar la secuencia (obtener el primer elemento),
2. pasar de un elemento al siguiente (obtener sgte. elemento), y
3. detectar el final de la secuencia (condición de parada).

Suma de números pares

Secuencia y diseño del bucle

Ejemplo #1: Sumar los números pares hasta `número` (incluido):

- Secuencia: 2, 4, . . . , `número`.
- Tipo: secuencia descrita por enumeración.
- Operaciones:
  1. Obtener primer elemento: `i`=2.
  2. Obtener siguiente elemento: `i`=`i`+2.
  3. Fin de la secuencia: `i`>`número`.

Media de 10 números reales

Secuencia y diseño del bucle

Ejemplo #2: Media de 10 números reales leídos de teclado:

- Secuencia original: $valor1$, $valor2$, ..., $valor10$.
- Tipo: secuencia de longitud conocida (10 elementos)
- Secuencia alternativa: la secuencia original de 10 números la podemos tratar recorriendo la secuencia 1, 2, ..., 10 y en cada iteración leer de teclado el $valor$-i y sumarlo.
- Operaciones:
  1. Obtener primer elemento: `i`=1.
  2. Obtener siguiente elemento: `i`=`i`+1.
  3. Fin de la secuencia: `i`>10.

Contar los dígitos de un nº entero

Secuencia y diseño del bucle

Ejemplo #3: Contar los dígitos de un número entero leído de teclado:

- Secuencia: `número`, `número`/10, `número`/100, . . . , 0.
  Ejemplo: 147539, 14753, 1475, 147, 14, 1, 0.
- Tipo: secuencia delimitada por un valor centinela.
- Operaciones:
  1. Obtener primer elemento: `número`.
  2. Obtener siguiente elemento: `número`=`número`/10.
  3. Fin de la secuencia: `número`==0.

Tratamientos de secuencias de elementos

Algoritmos genéricos: enumeración y valor centinela

- Identificar la secuencia.
- Definir sus 3 operaciones: *primer-elemento*, *sgte-elemento*, *fin-secuencia*.
- Escribir el algoritmo eligiendo el tipo de bucle:
  1. `for`: secuencias de longitud conocida u otras en las que la operación *sgte-elemento* es simple,
  2. `while`, `do-while`: resto de casos, especialmente cuando *sgte-elemento* es más compleja.

Algoritmo 7 Tratar secuencias (mientras)

```
elemento=primer-elemento
mientras NO fin-secuencia hacer
    Tratar elemento
    elemento=sgte-elemento
fin mientras
```

Algoritmo 8 Tratar secuencias (repetir)

```
elemento=primer-elemento
hacer
    Tratar elemento
    elemento=sgte-elemento
mientras NO fin-secuencia
```

Tratamientos de secuencias de elementos

Algoritmos genéricos: secuencias de longitud conocida

- No hace falta comprobar si cada elemento de la secuencia pertenece o no.
- Comprobamos el número de iteraciones.
- En algunos programas es más natural obtener primero el elemento y luego tratarlo. Elimina la necesidad de obtener el primer elemento antes del bucle.

Algoritmo 9 Tratar secuencias (longitud conocida N)

```
para i desde 1 hasta N hacer
    Obtener elemento
    Tratar elemento i-ésimo
fin para
```

Programa: Sumar una secuencia de n° positivos

Enunciado y algoritmo

Enunciado

Hacer un programa que dada una secuencia, posiblemente vacía, de números enteros positivos introducidos por teclado y que finaliza cuando se introduzca un número negativo, calcule su suma.

- Secuencia: valor centinela (< 0)
  Ejemplos: -23
  4 3 2 1 -1

1. *primer-elemento*: leer `número`.
2. *sgte-elemento*: leer `número`.
3. *fin-secuencia*: `número`< 0.

Algoritmo 10 Suma de números positivos

suma $=$ 0
Leer número de teclado
mientras número ≥ 0 **hacer**
    suma $=$ suma $+$ número
    Leer número de teclado
fin mientras
Imprimir suma

Programa: Sumar una secuencia de nº positivos

Código fuente

```
 6 public class SumaNúmerosPositivos {

 8     public static void main(String[ ] args) {
 9         //Objeto Scanner asociado con el teclado
10         Scanner teclado= new Scanner(System.in);
11         //Declaramos una variable entera para leer los números
12         int número;
13         //y otra para ir calculando su suma
14         int suma=0;
15         System.out.print("Secuencia de enteros positivos:");
16         //Secuencia: números positivos... número negativo
17         //Leemos el primer entero
18         número=teclado.nextInt();
19         while ( número >= 0 ) {
20             suma += número;              //sumamos
21             número=teclado.nextInt();  //leemos el siguiente
22         }
23         //Mostramos la suma en la pantalla
24         System.out.printf("Su suma es %d\n",suma);
25     }
26 }
```

Programa: Imprimir los divisores de un nº entero

Enunciado y algoritmo

Enunciado

Hacer un programa que dado un número entero positivo leído de teclado, imprima todos los divisores de ese número.

- Secuencia: 1, 2, . . . , `número` (enumeración de elementos).
  1. *primer-elemento*: `i`=1.
  2. *sgte-elemento*: `i`=`i`+1.
  3. *fin-secuencia*: `i`>`número`.

Algoritmo 11 Imprimir los divisores de un nº entero

```
Leer número de teclado
para i desde 1 hasta número hacer
    si número % i == 0 entonces
        Imprimir i
    fin si
fin para
```

Programa: Imprimir los divisores de un nº entero

Código fuente

```
5  public class Divisores {
6
7      public static void main(String[ ] args) {
8          //Objeto Scanner asociado con el teclado
9          Scanner teclado= new Scanner(System.in);
10         //Declaramos una variable entera para leer el número
11         int número;
12         //Leemos el nº entero
13         System.out.print("Introduce un entero:");
14         número=teclado.nextInt();
15         //Imprimimos sus divisores
16         //Secuencia: i: 1..número
17         for (int i=1; i<=número; i++)
18             //Es divisor si el resto da cero!
19             if ( número % i == 0)
20                 System.out.printf("%d ",i);
21     }
22 }
```

Búsquedas asociativas

Buscar el primer elemento que cumple una propiedad

- No se pretende hacer una cierta acción sobre todos los elementos de una secuencia.
- Los objetivos de las búsquedas asociativas son:
  1. Determinar si en la secuencia **hay o no** algún elemento que cumpla una cierta propiedad.
  2. En caso de existir, encontrar **el primer elemento** que cumple dicha propiedad.
- No siempre se recorren todos los elementos de la secuencia. El bucle puede finalizar por dos motivos:
  1. se ha recorrido toda la secuencia y ninguno de sus elementos cumple la propiedad buscada, o
  2. se ha encontrado el primer elemento que la cumple
- Esto hace que el bucle tenga una condición formada por dos términos.
- Además, obliga a que después del bucle comprobemos por cuál de los dos términos ha acabado el bucle.

Búsquedas asociativas

Algoritmo genérico

Deben definirse dos cosas:

1. La **secuencia** y sus 3 operaciones: *primer-elemento*, *sgte-elemento*, *fin-secuencia*.
2. La **propiedad** que se quiere encontrar entre los elementos de la secuencia: *elemento-encontrado*

Algoritmo 12 Búsquedas asociativas

```
elemento=primer-elemento
mientras NO fin-secuencia Y NO elemento-encontrado hacer
    elemento=sgte-elemento
fin mientras
si NO fin-secuencia entonces
    se ha encontrado el elemento
sino
    NO se ha encontrado el elemento
fin si
```

Programa: Número primo

Enunciado y algoritmo

Enunciado

Hacer un programa que lea un número entero positivo de teclado e imprima si el número es primo o no.

- Secuencia: 2, . . . , `número`/2 (enumeración de elementos).
- Propiedad: `número` % `i`==0

Algoritmo 13 Número primo

```
Leer número de teclado
i=2
mientras (i<=número/2) Y (número % i!=0) hacer
    i=i+1
fin mientras
si (i<=número/2) entonces Imprimir NO ES primo
sino Imprimir ES primo
fin si
```

Programa: Número primo

Código fuente

```
 5 public class Primo {

 7     public static void main(String[ ] args) {
 8         //Objeto Scanner asociado con el teclado
 9         Scanner teclado= new Scanner(System.in);
10         //Declaramos una variable entera para leer el número
11         int número;
12         //Leemos el nº entero
13         System.out.print("Introduce un entero:");
14         número=teclado.nextInt();
15         //Hacemos una búsqueda asociativa
16         //Secuencia: i: 2..número/2
17         //Propiedad: i divisor de número
18         int i=2;  //para recorrer la secuencia
19         while ( ( i <= número/2 ) && ( número % i !=0 ) )
20             i++;
21         //Mostramos si es primo o no
22         if ( i <= número/2 )
23               System.out.printf("%d NO es primo\n",número);
24         else  System.out.printf("%d SÍ es primo\n",número);
25     }
26 }
```

Programas: Números primos entre 1 y 1000

Enunciado y análisis

Enunciado

Hacer un programa que imprima los números primos entre 1 y 1000.

- Esquema iterativo #1 (tratamiento)
  1. Secuencia. `j`: 1 .. 1000
  2. Acción: saber si el número es primo o no.
- Esquema iterativo #2 (búsqueda)
  - Secuencia. `i`: 2 .. `j`/2
  - Propiedad: `j` % `i` == 0.

Programa: Números primos entre 1 y 1000

Algoritmo

Algoritmo 14 Números primos entre 1 y 1000

```
j=1;
mientras j<=1000 hacer
    i=2
    mientras (i<=j/2) Y (j % i!=0) hacer
        i=i+1
    fin mientras
    si (i>j/2) entonces Imprimir j
    fin si
    j=j+1
fin mientras
```

Programa: Números primos entre 1 y 1000

Código fuente

```
3  public class NúmerosPrimos {

5      public static void main(String[ ] args) {
6          //Hacemos un tratamiento de una secuencia
7          //Secuencia: 1..1000
8          for (int j=1; j<=1000; j++) {
9             //Hacemos una búsqueda asociativa
10            //Secuencia: i: 2..j/2
11            //Propiedad: i divisor de j
12            int i=2;  //para recorrer la secuencia
13            while ( ( i <= j/2 ) && ( j % i !=0 ) )
14               i++;
15            //Mostramos si es primo o no
16            if ( i > j/2 )  System.out.printf(" %d ",j);
17         }
18     }
19 }
```

Bucles anidados

Un par de consideraciones

- Se deben abstraer ciertas operaciones complejas (serán los bucles internos).
- Se deben inicializar siempre las variables de control justo antes.

```
i = ...
while ( i ... ) {
    j = ...
    while ( j ... ) {
        ...
    }
}
```

```
i = ...
j = ...
while ( i ... ) {
    //OJO! j NO se inicializa cada
        vez
    while ( j ... ) {
        ...
    }
}
```

Importante

Las variables de control de un bucle se deben inicializar justo antes de su inicio. Es especialmente crucial en los bucles anidados

Pruebas funcionales

Caja blanca y caja negra

Objetivo: Realizar pruebas funcionales de pequeños programas que tengan sentencias condicionales y bucles.

Enfoques:

1. **Caja blanca**: consiste en examinar el propio código, comprobando los caminos lógicos del programa, los bucles y las condiciones, y analizando el estado del programa en diversos puntos.
2. **Caja negra**: consiste en tratar el código como una caja negra que recibe una entradas y produce una salida. Típicamente se diseñan una batería de casos de prueba y el programa debe producir en todos ellos la salida correcta.

Pruebas de caja blanca

Concepto de cobertura

Importante

Hay que tratar de probar TODAS las sentencias del programa

Cobertura

Es una medida porcentual que indica la cantidad de código que ha sido cubierto (o probado) durante las pruebas realizadas.

- Cobertura de segmentos. Ejecutar todos los segmentos del programa.
- Cobertura de ramas. Ejecutar todos los caminos posibles.
- Cobertura de condiciones. Cuando tenemos condiciones complejas, hay que probar todas las combinaciones posibles.
- Cobertura de bucles. Hay que probar casos en los que no se entre en el bucle (si eso es posible), casos en los que se ejecute varias veces y comprobar que no sea infinito.

Depuradores

Herramientas para encontrar y corregir errores

Depurador

Programa integrado en los entornos de desarrollo que permite localizar y corregir los errores que contienen los programas.

El depurador permite:

- Ejecutar paso a paso las sentencias del programa.
- Ejecutar el programa hasta una cierta sentencia o hasta que se cumpla una cierta condición. Estos puntos donde se detiene la ejecución se denominan puntos de ruptura o *breakpoints*.
- Ir viendo la evolución de la variables del programa a medida que sus sentencias se ejecutan.

Importante

La ejecución paso a paso de los bucles ayuda a comprender mejor cómo funcionan realmente y cómo cambian sus variables

Pruebas de caja negra

Comprobar que el programa produce las salidas correctas

- El probador introduce unos datos de entrada y espera que la salida sea la correcta (caso de prueba).
- Objetivo: encontrar casos en los que el programa no funciona.
- Para que el programa sea correcto debe hacer todos los casos de prueba correctamente.
- Problema: el nº de entradas posibles es enorme o infinito.

Importante

No hace falta probar todas las posible entradas, probando ciertas entradas (clases de equivalencia) puede bastar para verificar que un programa funciona

Clases de equivalencia

Determinar los valores que debemos probar

- Idea: dividir el conjunto posible de valores de entrada en varios subconjuntos (**clases de equivalencia**). Si el programa funciona para algunos de los valores de un subconjunto, entonces también funciona para todos los demás.
- Solamente hace falta probar ciertos valores de cada subconjunto

Ejemplo: Tenemos una entrada que representa el valor del mes

Hay tres clases de equivalencia:

1. Menos de 1 (valores incorrectos menores).
2. Entre 1 y 12 (valores correctos).
3. Más de 12 (valores incorrectos mayores).

Pruebas: un valor aleatorio menor que 1, otro valor entre 1 y 12, otro mayor que 12, y los valores límite: 0, 1, 12 y 13.

Casos de prueba: sentencias condicionales

Tarifas de autobús: tabla de valores

	menor de 18	entre 18 y 64	mayor de 64
Coruña	30	35	30
Gijón	25	30	25
Santander	25	30	25
Bilbao	30	35	30

Aplicando el concepto de la clases de equivalencia y los valores límite, habría que:

- Probar al menos una vez cada uno de los casos de la tabla. Por ejemplo, para Coruña habría que probar: C 12, C 37 y C 80.
- Probar además, para cada caso, los valores límite de la edad (18 y 64). Siguiendo con Coruña: C 17, C 18, C 64 y C 65.
- Probar valores incorrectos para la ciudad destino (una letra que no sea ni C, ni G, ni B, ni S).

Casos de prueba: tratamiento de secuencias

Suma de pares y contar dígitos

Ejemplo: Suma de los números pares entre 2 y un entero dado

- `número`=1 : el bucle no hace iteraciones, resultado 0
- `número`=2 : el bucle hace una iteración, resultado 2
- `número`=8 : el bucle hace varias iteraciones, resultado 20
- Sería conveniente probar con valores impares. Por ejemplo, con 9 debe dar lo mismo que con 8 porque se suman los mismos pares (2+4+6+8).

Ejemplo: Contar el número de dígitos de un entero

- `número`=7 : el bucle hace una iteración, resultado: 1 dígito
- `número`=65 : el bucle hace dos iteraciones, 2 dígitos
- `número`=5341291 : el bucle hace varias iteraciones, 7 dígitos
- Además, probar ciertos valores límite: 0, 9, 10, 99, 100, etc.

Casos de prueba: búsquedas asociativas

Determinar si un número es primo o no

En las búsquedas asociativas hay que probar dos tipos de entradas:

- Datos de entrada que hacen que NO haya ningún elemento que cumpla la propiedad buscada.
- Datos para los que SÍ se encuentra lo que se busca.

Ejemplo: Determinar si un nº es primo no:

- Divisor NO encontrado (número primo). Por ejemplo, bastaría con probar los primeros números primos (2, 3, 5, 7) y luego algún otro mayor (p.e. 29, 83).
- Divisor encontrado (número no primo). Probar alguno de los primeros números no primos (4, 6, 14) y luego algunos otros mayores, especialmente impares (p.e. 15, 27, 33).

Ámbito de una variable

Dónde puede ser utilizada

Ámbito

Conjunto de instrucciones en las que una variable puede ser usada.

- Una variable solamente puede ser usada en el bloque de código en el que se declara, desde el punto en el que se declara.
- Por ejemplo, la variable de control que se declara en un bucle `for` no se puede usar fuera del `for`.
- Naturalmente, una variable declarada dentro de un bloque que tiene otros bloques anidados, sí se puede usar en esos bloques anidados.
- Por ejemplo, si se declara una variable al principio de un método, se puede usar en todos sus bloques anidados, tanto en los de los `if`'s o en los de los bucles que contenga.

Ámbito de una variable

Un ejemplo

```
1  public static void main(String [ ] args ) {
2     ...
3     int a;
4     if ( ... ) {
5         double b;
6         for (int i=...;...;...) {
7             for (int j=...;...;...) {
8                 ...
9             } //fin for#2
10            ...
11        } //fin for#1
12    } //fin parte then
13    else {
14         ...
15    }
16 }
```

1. `args`: accesible en todo el método,
2. `a`: en todo el método, menos las sentencias anteriores a su declaración.
3. `b`: en la parte *entonces* del `if`.
4. `i`: en el primer `for` (que incluye al segundo).
5. `j`: solamente en el segundo `for`.

Tiempo de vida de una variable

Cómo y cuándo se crea y se destruye una variable

Tiempo de vida de una variable

Período de tiempo en el que la variable existe, desde que se crea y se reserva espacio en memoria para almacenarla, hasta que se destruye y se libera el espacio que ocupa.

Las reglas son simples:

1. Creación: cuando se declara. Se reserva espacio en la Pila de forma que la variable se guarda en la cima de la Pila, apilada sobre todas las variables creadas anteriormente.
2. Liberación: cuando se acaba su ámbito. Se libera el espacio de la Pila que ocupaba la variable.

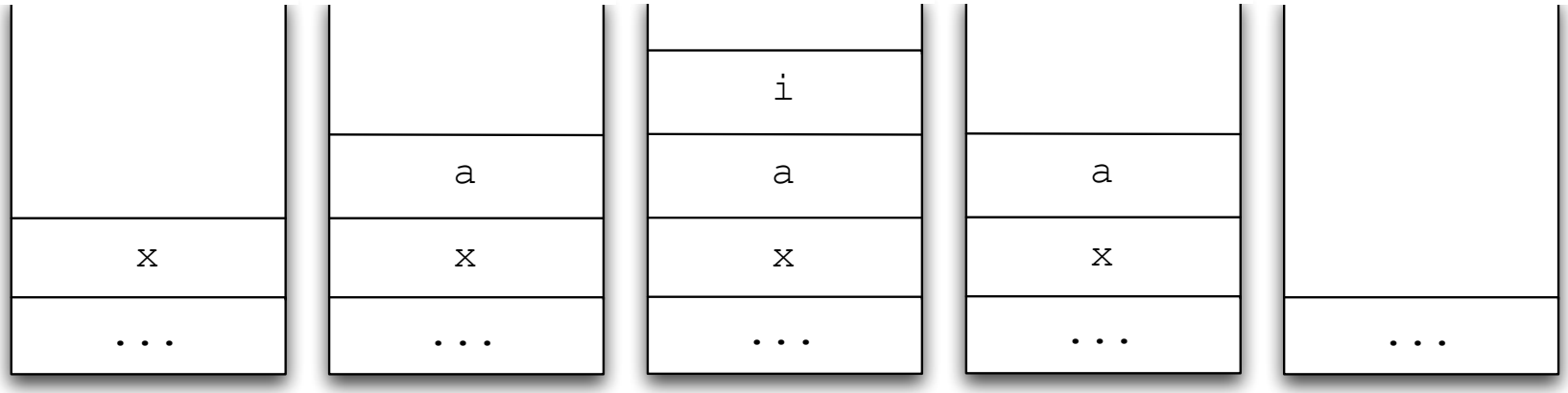
Para el programador es un **proceso transparente**:

- La gestión de la Pila NO la realiza el programador.
- Sólo decide la creación de variables mediante su declaración.
- Pero no se tiene que ocupar de la liberación, es automática.

Tiempo de vida

Ejemplo de creación y liberación en la Pila

```
1 public int f (int x) {  //Reservamos memoria para el parámetro x
2       int a;       //Reservamos memoria para a
3       ...
4       for (int i=0;i<10;i++) { //Reservamos memoria para i
5       ...
6       }   //i deja de tener validez, se libera el espacio que
            ocupa
7       ...
8 }     //a y x dejan de tener validez, se libera su espacio
```



¿Y qué pasa con los objetos?

También es un proceso transparente para el programador

- Las variables de tipos referenciados (p.e. objetos) se liberan usando el mismo mecanismo, se eliminan de la Pila al acabar su ámbito.
- ¿Pero qué pasa con el objeto al que apuntan? Esa memoria dinámica es liberada por el **Recolector de basura**.
- Se encarga de liberar la memoria del Montón de todos aquellos objetos que ya no están referenciados por ninguna variable.