

TEMA 1: Desarrollo de Software

Módulo

Entornos de Desarrollo

para los ciclos

Desarrollo de Aplicaciones Multiplataforma

Desarrollo de Aplicaciones Web



Entornos-GS; Tema 1 :DesarrolloSoftware

© Gerardo Martín Esquivel, Septiembre de 2022

Algunos derechos reservados.

Este trabajo se distribuye bajo la Licencia "Reconocimiento-No comercial-Compartir igual 3.0 Unported" de Creative Commons disponible en <http://creativecommons.org/licenses/by-nc-sa/3.0/>

1.1 Clasificación del Software.....	3
1.1.1 Según el tipo de tarea que realiza.....	3
1.1.2 Según el método de distribución.....	3
1.1.3 Según la licencia.....	4
1.2 Fases del ciclo de vida del software.....	5
1.2.1 Análisis.....	5
Técnicas que se utilizan durante el análisis.....	5
Resultados de la fase de análisis.....	6
Documentos finales en la fase de análisis.....	6
1.2.2 Diseño.....	9
Estructuras del diseño estructurado: secuencial.....	9
Estructuras del diseño estructurado: condicional.....	10
Estructuras del diseño estructurado: iterativa.....	12
Observaciones sobre los bucles.....	15
Las condiciones complejas.....	16
Contadores y acumuladores.....	17
Diagramas que usamos en la fase de diseño.....	20
Errores habituales.....	25
Diagrama de flujo para un programa completo.....	26
1.2.3 Codificación.....	27
Compilación.....	29
1.2.4 Pruebas.....	29
1.2.5 Documentación.....	30
1.2.6 Mantenimiento.....	30
1.3 Concepto de programa.....	31
1.3.1 Elementos de la CPU.....	31
1.4 Clasificación de los lenguajes de programación.....	32
1.4.1 Según el nivel de abstracción.....	32
1.4.2 Según la forma de ejecución.....	32
La máquina virtual de Java.....	33
1.4.3 Según el paradigma de programación.....	33
1.5 Instalación de la máquina virtual de Java y el Kit de desarrollo de Java.....	34
1.5.1 Instalación del JRE en Ubuntu.....	35
Comprobamos <i>si</i> ya tenemos el JRE.....	35
Instalación del JRE de OPENJDK.....	35
Instalación del JRE de Oracle.....	35
1.5.2 Instalación del <i>JDK</i> en Ubuntu.....	36
Comprobamos <i>si</i> ya tenemos el JDK.....	36
Instalación del JDK de OPENJDK.....	36
Instalación del JDK de Oracle.....	36
1.5.3 Instalación del JRE y JDK de OpenJDK en Windows.....	37
1.5.4 Instalación del JRE y JDK de <i>Oracle</i> en Windows.....	38

1.1 Clasificación del Software

Software es el "conjunto de programas y datos necesarios para ejecutar una tarea en un ordenador". En castellano se podría traducir como **Soporte Lógico**, pero lo habitual es usar el término en inglés.

Frente al término software está el término **Hardware** que hace referencia a todos los elementos físicos que componen el sistema informático (teclado, pantalla, CPU, impresora, ...)

A continuación clasificamos el software según varios criterios.

1.1.1 Según el tipo de tarea que realiza

- **Software de sistema:** es el que controla y hace funcionar el hardware. Son los sistemas operativos (como **Windows**, **Linux**, **Mac OS** o **Android**) y los controladores de dispositivos (más conocidos como **drivers**), etc.
- **Software de aplicación:** son los programas que nos ayudan a hacer tareas específicas. Ejemplos de este tipo de software son los programas ofimáticos, aplicaciones de contabilidad, diseño asistido por ordenador, etc.
- **Software de desarrollo:** son los programas que ayudan a los programadores en su tarea de escribir programas. Entre ellos se encuentran los **entornos de desarrollo integrados (IDE** - Integrated Development Environment) que agrupan todas las herramientas útiles con una avanzada interfaz gráfica de usuario y que están preparados para usarlos con distintos lenguajes de programación.

1.1.2 Según el método de distribución

- **Shareware:** es el software que se distribuye de forma gratuita con alguna limitación (en cuanto a tiempo o funcionalidad) para que el usuario lo evalúe y, si es de su agrado, y quiere disfrutar de la versión completa o sin límite de tiempo, deberá hacer un pago, generalmente pequeño. Ejemplos: **WinZip**, **WinRAR**, **PC-tools** o **Virus Scan**.
- **Freeware:** Gratuito. Suele incluir una licencia de uso que hace algunas restricciones, como la obligatoriedad de informar de quién es el autor y la prohibición de modificarlo o hacer negocio con él. La traducción correcta de **Freeware** es "**software gratuito**" y no debe confundirse con el "software libre" que veremos a continuación. Esta suele ser una confusión habitual derivada del hecho de que la palabra inglesa **free** se puede traducir como **libre** o como **gratuito**.
- **Adware:** que podría traducirse como software adjunto o añadido, es publicidad (en forma de anuncios o de software promocional) que acompaña a programas gratuitos. En ocasiones es posible evitarlos durante el proceso de instalación si estamos atentos. El usuario también tiene la posibilidad de adquirir las versiones de pago que estarán libres de **Adware**.
- **Software de uso específico:** es el software que se desarrolla a petición del cliente para un problema determinado. Estos programas están muy personalizados con los logos y los protocolos de funcionamiento de la empresa para la que va destinada.

1.1.3 Según la licencia

La licencia de un programa es el contrato en el que se establece el uso que el usuario final puede hacer de ese programa.

- **Software libre:** permite al usuario utilizarlo y distribuirlo libremente. La licencia más habitual en software libre es la **GPL** (General Public License - Licencia Pública General).
- **Software libre de código abierto:** Es software libre que se distribuye con el código fuente, de modo que el usuario también podrá estudiar su funcionamiento interno y modificarlo o añadirle funcionalidad.
- **Software propietario:** Es el software que se distribuye con rígidas condiciones de uso. Generalmente se prohíbe la redistribución, modificación, copia, ejecución simultánea en varias máquinas. Este software normalmente se distribuye en formato binario (es decir, sin acceso al código fuente) para hacer efectiva la prohibición de modificarlo. Sin embargo el software propietario también puede ser gratuito o de pago.
- **Software de dominio público:** es el que no tiene ningún tipo de licencia y puede ser usado por cualquiera en la forma que desee. El motivo puede ser porque el autor sea desconocido o porque ha pasado el tiempo suficiente para que expiren los derechos de autor.

1.2 Fases del ciclo de vida del software

Se define el **ciclo de vida del software** como el conjunto de procesos y tareas que se realizan desde que surge la necesidad de una aplicación hasta que, una vez que ha quedado obsoleto, finaliza su uso.

El ciclo de vida del software tiene etapas diferenciadas:

- **Análisis:** En esta etapa se estudian las necesidades que tiene que cubrir la aplicación que vamos a desarrollar. Será necesario tener en cuenta todas las exigencias. El resultado será un documento que recoge toda la información.
- **Diseño:** Durante esta etapa se decide cómo se va a resolver: estructuras de datos, interfaz de usuario, procedimientos. También se selecciona el lenguaje de programación y el sistema gestor de bases de datos, entre otros. El resultado incluirá diagramas que serán fundamentales para los desarrolladores.
- **Codificación:** En esta etapa se usa un lenguaje de programación para escribir, de forma comprensible para la máquina, todo lo diseñado. El resultado de esta etapa será código ejecutable.
- **Pruebas:** Se comprueba que el programa hace todo lo que tiene que hacer y que además, lo hace con la calidad esperada.
- **Mantenimiento:** Esta etapa se desarrolla después de la entrega del producto al cliente. En ella habrá que hacer cambios. Estos cambios se pueden deber a que se detectan errores, a que se producen cambios en el entorno de trabajo (por ejemplo, se cambia el sistema operativo) o a que el cliente desee añadir nuevas funcionalidades.

1.2.1 Análisis

Esta fase es fundamental para el éxito del proyecto, sería imposible hacer un buen trabajo si no se entienden previamente cuales son las necesidades y las condiciones. El principal problema que nos encontramos es la comunicación. A menudo surgen problemas derivados de que el cliente no tiene conocimientos informáticos y el informático no conoce la el campo de la actividad del cliente.

TÉCNICAS QUE SE UTILIZAN DURANTE EL ANÁLISIS

- **Entrevistas:** Comunicación entre los informáticos y clientes.
- **Brainstorming:** Son reuniones en grupo que buscan generar ideas diferentes desde distintos puntos de vista. Es muy adecuado al principio del proyecto.
- **Prototipos:** Son versiones iniciales para clarificar algún concepto. Después puede desecharse o usarse como base para añadir más cosas.
- **Casos de uso:** Es una técnica definida con **UML** (Unified Modeling Language), se basa en escenarios que describen como se usa el software en una determinada situación.

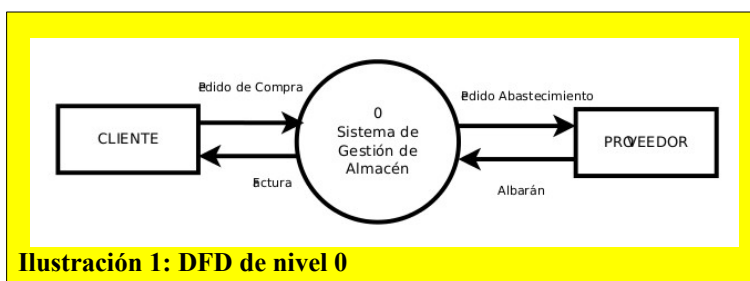
RESULTADOS DE LA FASE DE ANÁLISIS

- **Requisitos funcionales:** Detallan las funciones que realiza el sistema, como debe responder ante determinadas entradas, como se comporta en cada situación, etc. Ejemplos: "El usuario puede agregar un nuevo contacto", "El usuario puede imprimir la lista de contactos".
- **Requisitos no funcionales:** Sobre las características del sistema: fiabilidad, mantenibilidad, sistema operativo, hardware sobre el que se ejecutará, etc. Ejemplos: "La aplicación debe funcionar tanto bajo Windows como Linux", "La respuesta a consultas debe ser inferior a 5 segundos".

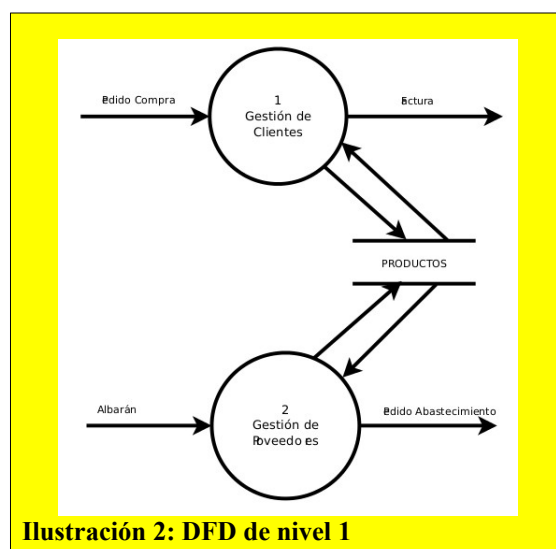
DOCUMENTOS FINALES EN LA FASE DE ANÁLISIS

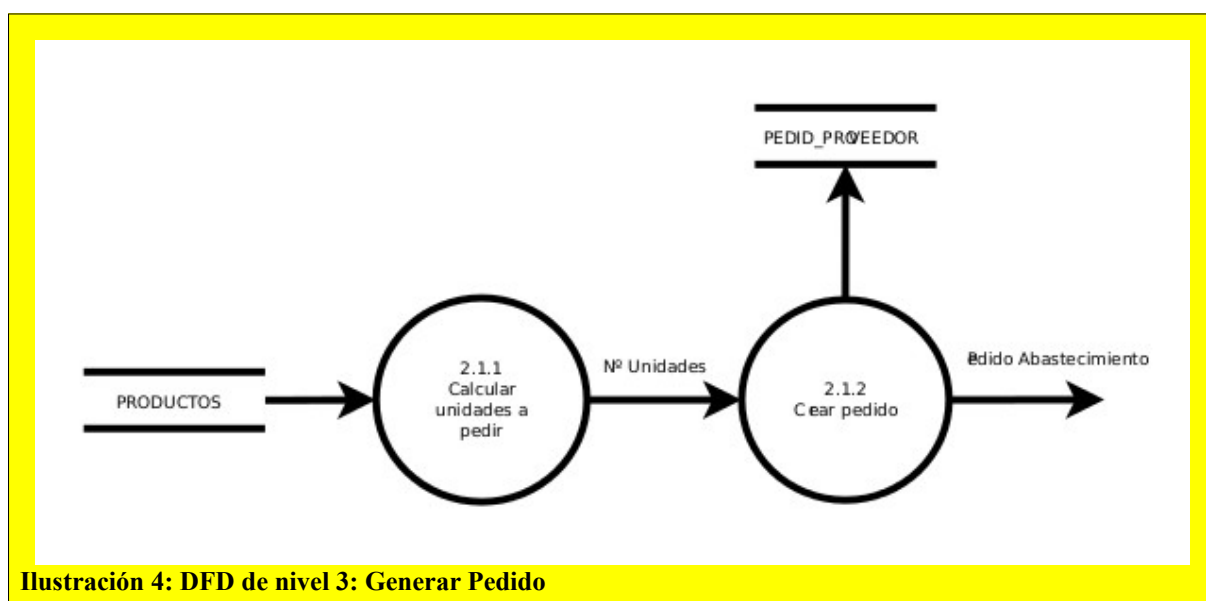
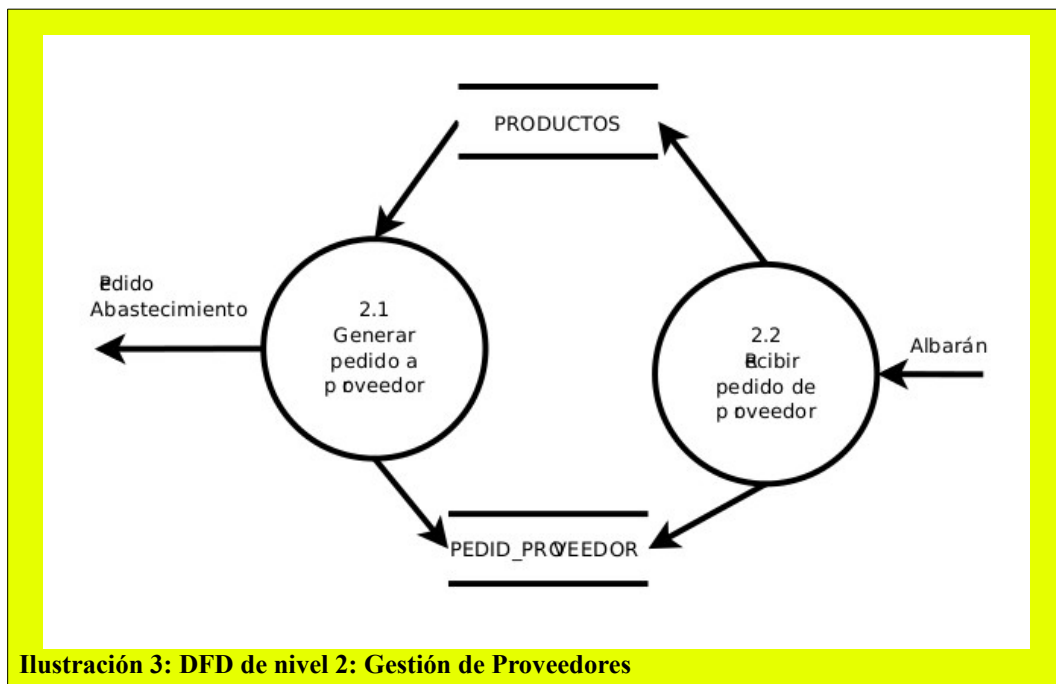
- **Diagramas de Flujo de Datos (DFD):** Representa el flujo de los datos entre los distintos procesos, entidades externas y almacenes del sistema.

Se utilizan rectángulos para representar las entidades externas (componentes que no forman parte del sistema, pero que interaccionan con él, como un proveedor o un cliente).



- Se usan elipses para representar los procesos internos.
- Se usan dos líneas horizontales paralelas para mostrar los almacenes de datos.
- Se usan flechas para indicar el flujo (el camino) que siguen los datos dentro del sistema.





En este ejemplo faltaría un **DFD** de nivel 2 y todos los de nivel 3 hasta completar el conjunto de procesos.

Nota: En la elaboración de los **DFD** se empieza considerando un único problema global que poco a poco se va dividiendo en problemas más pequeños hasta llegar a procesos fácilmente resolubles. Esta es una técnica muy habitual en informática llamada "**Divide y vencerás**".

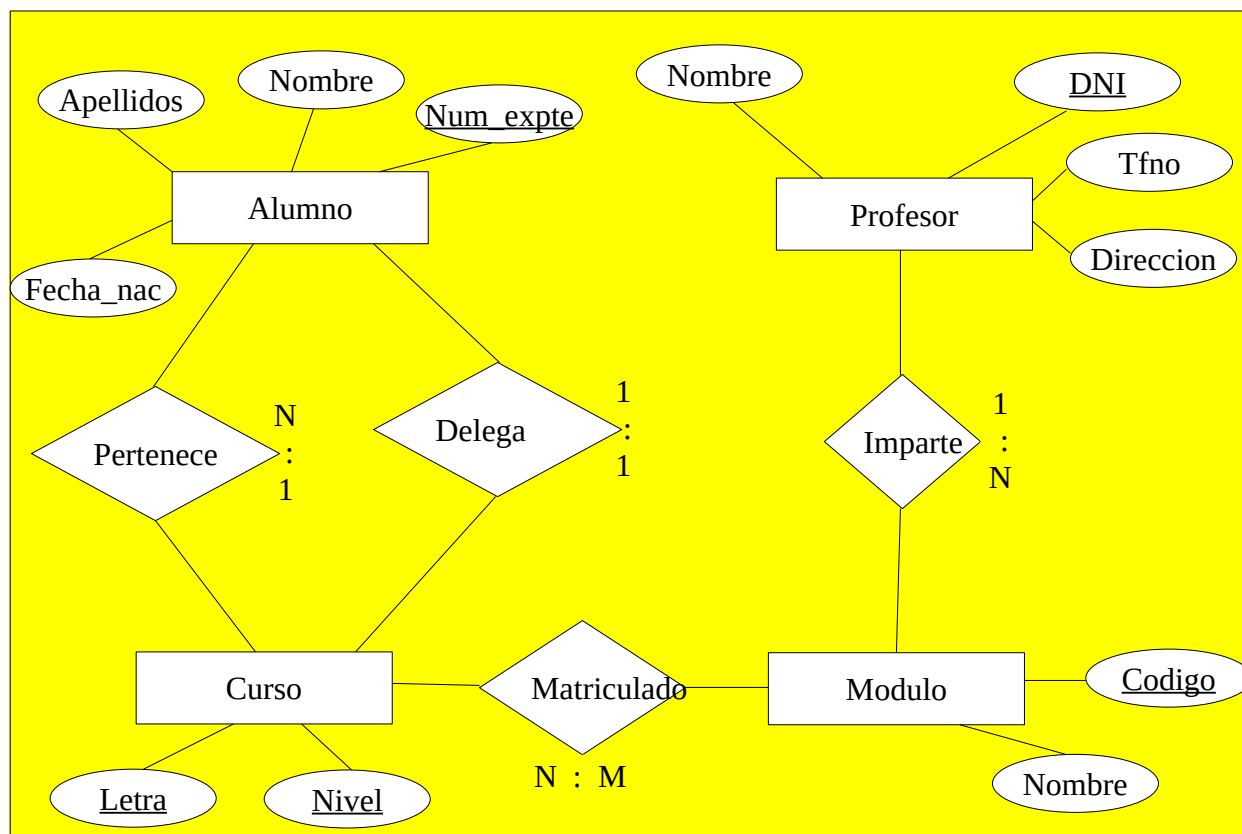
- **Diccionarios de Datos (DD):** Es un documento escrito que acompaña al **DFD** describiendo detalladamente los datos que allí se utilizan. Siguiendo con el ejemplo anterior, en el diccionario de datos deberá de especificarse con detalle qué datos componen un "Pedido Abastecimiento", un "Albarán", etc.

- **Diagramas Entidad/Relación (DER):** Representan los datos y la forma en que se relacionan. Los elementos relevantes de nuestro sistema (entidades) se representan mediante rectángulos. Las características de cada uno de ellos (atributos) se representan mediante elipses. Y las relaciones entre entidades (relaciones) se representan mediante rombos.

Nota: La confección de **diagramas E/R** forma parte del contenido del módulo **Bases de Datos** de primer curso de **DAM/DAW**.

Ejemplo 1

Se desea diseñar la base de datos de un Instituto. En la base de datos se quiere guardar los datos de los profesores del Instituto (DNI, nombre, dirección y teléfono). Los profesores imparten módulos, y cada módulo tiene un código y un nombre. Cada alumno está matriculado en uno o varios módulos. De cada alumno se desea guardar el nº de expediente, nombre, apellidos y fecha de nacimiento. Los profesores pueden impartir varios módulos, pero un módulo sólo puede ser impartido por un profesor. Cada curso tiene un grupo de alumnos, uno de los cuales es el delegado del grupo.



1.2.2 Diseño

En esta etapa partimos de los **Diagramas de Flujo de Datos** y **Diagramas Entidad/Relación** que se han generado en la etapa anterior y que definen los requisitos funcionales y no funcionales.

Ahora, a partir de esos documentos, vamos a crear una representación del software, en forma de algoritmos. Un **algoritmo** es una **secuencia finita de pasos ordenados sin ambigüedad que llevan a la resolución de un problema**.

Hay principalmente dos tipos de diseño: el estructurado y el orientado a objetos.

- **Diseño estructurado:** Nace en los años 60 y define cada programa como un guión que indica todas las instrucciones que se tienen que ejecutar y en qué orden, combinando tres estructuras básicas. Se basa en unas ideas muy sencillas:

- Todos los programas tienen **un único inicio** y **un único final**.
- Cualquier programa puede construirse con la combinación de tres estructuras lógicas: secuencial, condicional y repetitiva. Se prohíbe explícitamente el uso de instrucciones que generan saltos en el flujo del programa (del tipo **goto** o **gosub**).
- Se permite reutilizar software mediante la creación de **subprogramas** (también llamados rutinas, subrutinas, procedimientos, funciones, métodos, etc.) que tienen inicio y fin y un nombre a través del cual se pueden hacer llamadas al subprograma desde otros programas.

El diseño de un programa estructurado se hace mediante los llamados **diagramas de flujo** que veremos con detalle en el apartado siguiente.

- **Diseño orientado a objetos:** El diseño orientado a objetos no es completamente ajeno al diseño estructurado, sino un enfoque más moderno y más complejo que incorpora también el diseño estructurado. En la orientación a objetos, el sistema se entiende como un conjunto de **objetos** que tienen características (**propiedades**) y comportamiento (**métodos**) y que interactúan entre sí.

Para representar los objetos y sus relaciones usaremos los **diagramas de clases** que se crean usando un lenguaje gráfico llamado **UML** (Unified Modeling Language - Lenguaje de Modelado Unificado) y para detallar el comportamiento de los métodos usaremos los mismos **diagramas de flujo** que usamos en el diseño estructurado.

ESTRUCTURAS DEL DISEÑO ESTRUCTURADO: SECUENCIAL

En una estructura secuencial las instrucciones se siguen una tras otra con el orden establecido.

```
instrucción 1
instrucción 2
instrucción 3
```

Ejemplo 2: algoritmo para sumar dos números:

```
Inicio
  Preguntar el primer número
  Preguntar el segundo número
  Calcular la suma de primer número y segundo número
  Comunicar el resultado
Fin
```

Nota: Observa que el orden es muy importante. No podríamos calcular la suma antes de conocer los sumandos, ni podríamos comunicar el resultado antes de calcularlo.

ESTRUCTURAS DEL DISEÑO ESTRUCTURADO: CONDICIONAL

La estructura condicional permite seleccionar un camino (conjunto de instrucciones) u otro a partir de la evaluación de una condición lógica.

```
Si se cumple <condición>  
    instrucción A  
Caso contrario  
    instrucción B  
Fin_Si
```

Ejemplo 3: algoritmo para determinar la mayoría de edad en función de la edad

```
Inicio  
    Preguntar la edad al usuario  
    Si se cumple edad<18  
        Mostrar mensaje "Es usted menor de edad"  
    Caso contrario  
        Mostrar mensaje "Es usted mayor de edad"  
    Fin_Si  
Fin
```

A veces solamente hay un camino. Si se cumple la condición se recorre ese camino. Si no se cumple, pasamos directamente al final del condicional:

```
Si se cumple <condición>  
    instrucción A  
Fin_Si
```

Ejemplo 3b: algoritmo para determinar la minoría de edad en función de la edad

```
Inicio  
    Preguntar la edad al usuario  
    Si se cumple edad<18  
        Mostrar mensaje "Es usted menor de edad"  
    Fin_Si  
Fin
```

La **condicional múltiple** aparece cuando encadenamos varias condiciones:

```
Si se cumple <condición1>  
    instrucción A  
Caso contrario  
    Si se cumple <condición2>  
        instrucción B  
    Caso contrario  
        Si se cumple <condición3>  
            instrucción C  
        Caso contrario  
            instrucción D  
        Fin_si  
    Fin_si  
Fin_si
```

Nota: La *indentación* (separación de cada línea desde el margen izquierdo) es muy importante para clarificar la lectura del algoritmo.

Nota: También resulta clarificador indicar donde termina una estructura condicional con su correspondiente *Fin_si*. Observa que para cada *Si* existe un *Fin_si* y que ambos están en la misma columna de texto.

Ejemplo 4: algoritmo para determinar la situación en función de la edad

```
Inicio
  Preguntar la edad al usuario
  Si se cumple edad<18
    Mostrar mensaje "Es usted menor de edad"
  Caso contrario
    Si se cumple edad<65
      Mostrar mensaje "Está usted en edad de trabajar"
    Caso contrario
      Mostrar mensaje "Usted puede jubilarse"
    Fin_si
  Fin_si
Fin
```

La selección múltiple se puede expresar también así:

```
Según el valor de Dato
  si valor1: instrucción A
  si valor2: instrucción B
  si valor3: instrucción C
  en otro caso: instrucción D
Fin_según
```

En esta estructura solamente se sigue uno de los caminos. Nunca más de un camino.

Ejemplo 5: algoritmo para determinar la provincia a partir del código postal

```
Inicio
  Preguntar el codigo al usuario
  Según el valor de codigo
    si 18: mostrar "Granada"
    si 04: mostrar "Almería"
    si 23: mostrar "Jaén"
    en otro caso: mostrar "provincia desconocida"
  Fin_según
Fin
```

A veces no hay un camino final para "el resto de valores". Si el valor coincide con alguno de los señalados se recorre ese camino. Si no coincide con ninguno, pasamos directamente al final del condicional múltiple:

```
Según el valor de Dato
  si valor1: instrucción A
  si valor2: instrucción B
  si valor3: instrucción C
Fin_según
```

Ejemplo 5b: algoritmo para determinar la provincia a partir del código postal

```
Inicio
  Preguntar el código al usuario
  Según el valor de código
    si 18: mostrar "Granada"
    si 04: mostrar "Almería"
    si 23: mostrar "Jaén"
  Fin_según
Fin
```

Muy Importante: La estructura condicional **SI / FIN_SI** utiliza **condiciones** para bifurcar los caminos. La estructura **SEGUN / FIN_SEGUN** utiliza **valores discretos** para bifurcar los caminos.

Ejemplo 6: algoritmo para dividir

```
Inicio
  Preguntar el numerador
  Preguntar el divisor
  Si divisor=0
    mostrar mensaje: "no se puede dividir"
  Caso contrario
    calcular resultado = numerador : divisor
    comunicar el resultado
  Fin_si
Fin
```

Nota: En el ejemplo anterior se combinan instrucciones **secuenciales** y **condicionales**.

Nota: La expresión **resultado = numerador : divisor** es una **asignación**. Primero se calcula la expresión de la derecha **numerador : divisor** y el valor obtenido se asigna a la variable de la izquierda **resultado**. La **asignación** es una operación destructiva: se pierde el valor previo. La **asignación** se suele indicar en todos los lenguajes de programación con el símbolo igual (=) o algo similar, pero no se debe confundir con una igualdad como la que hay en este mismo ejemplo: **divisor=0**.

ESTRUCTURAS DEL DISEÑO ESTRUCTURADO: ITERATIVA

En la estructura iterativa se repite una instrucción o (conjunto de instrucciones) mientras se cumpla una condición. Cuando la condición no se cumpla se continúa el resto del programa.

Habitualmente se distinguen tres tipos de estructuras repetitivas: **Bucle de condición al principio**, **Bucle de condición al final** y la **Bucle de rango**:

- **Bucle de condición al principio:** en esta estructura la condición se evalúa al principio del bloque repetitivo. Como consecuencia, el bloque se repetirá 0 o más veces (si la primera vez que se evalúa la condición no se cumple, no se ejecutará el bloque ninguna vez y el programa continuará con las instrucciones que hay tras el bloque repetitivo).

```
Mientras <condición>
  instrucción 1
  instrucción 2
Fin_Mientras
```

Nota: Observa que en los bucles también usamos la indentación y marcamos el final para que quede muy claro cual es el bloque de instrucciones que se repite.

Ejemplo 7: Contar hasta 3

```
Inicio
  Establecer contador=1
  Mientras contador<=3
    Mostrar contador
    Incrementar el contador en 1
  Fin_Mientras
Fin
```

- **Bucle de condición al final:** en esta estructura la condición se evalúa al final del bloque repetitivo. El bloque se ejecutará 1 o más veces porque la primera vez entrará en el bucle (siempre) antes de evaluar la condición.

```
Repetir
  instrucción 1
  instrucción 2
Hasta <condición>
```

Nota: Tal y como hemos expresado este bucle (*Repetir-hasta*) el bloque se repetirá siempre que la condición sea falsa, es decir *hasta que la condición sea verdadera*.

En algunos lenguajes de programación este bucle se expresa con los términos (*Repetir-mientras*) y por tanto el bloque se repite siempre que la condición sea verdadera. No tiene mayor importancia, solamente debemos fijarnos si funciona de uno u otro modo.

Ejemplo 8: Esperar que el usuario introduzca un número par

```
Inicio
  Repetir
    Mostrar mensaje: "Introduzca un número par"
    Esperar numero
  Hasta numero es par
Fin
```

En este programa, el bucle se repite cada vez que el usuario introduzca un número impar. Cuando introduzca un número par, se cumplirá la condición y se llega al final del bucle y del programa.

- **Bucle de rango:** en esta estructura se establece de forma predeterminada cuantas veces debe ejecutarse el bloque repetitivo.

```
Para valores de dato desde A hasta B
  instrucción 1
  instrucción 2
Fin_para
```

Ejemplo 9: Mostrar los primeros 7 múltiplos de 5

```
Inicio
  Para valores de numero desde 1 hasta 7
    Calcular multiplo = numero * 5
    Mostrar multiplo
  Fin_para
Fin
```

En este programa el bloque se repetirá 7 veces. En cada una de las repeticiones, la variable **numero** tomará los valores 1, 2, 3, 4, 5, 6 y 7 respectivamente.

Ejemplo 10: algoritmo para mostrar la tabla de multiplicar de un número dado

```
Inicio
  Mostrar mensaje: Introduzca un número
  Esperar el número: N
  Para valores de contador desde 1 hasta 10
    Calcular resultado = N x contador
    Mostrar mensaje: N por contador = resultado
  Fin_para
Fin
```

Nota: Las distintas estructuras iterativas son intercambiables entre sí. Puedes intentar escribir el algoritmo de la tabla de multiplicar usando las otras estructuras iterativas (mira el **ejemplo 10b**, a continuación). No obstante, en cada algoritmo una de las estructuras será más adecuada que las otras. Un buen programador elegirá siempre el bucle más adecuado:

Bucle de condición al principio: se usa cuando el bucle se tiene que ejecutar 0 o más veces; la condición está al principio del bucle.

Bucle de condición al final: se usa cuando el bucle se tiene que ejecutar 1 o más veces; la condición está al final del bucle.

Bucle de rango: se usa cuando sabemos el número de iteraciones de antemano. Quizá no sabemos el número exacto, pero podemos conocerlo de forma abstracta (por ejemplo, una iteración para cada carácter de una cadena de caracteres significa que el número de iteraciones será igual a la longitud de la cadena).

Ejemplo 10b: tabla de multiplicar (con bucle Bucle de condición al final)

```
Inicio
  Mostrar mensaje: Introduzca un número
  Esperar el número: N
  Establecer contador=1
  Repetir
    Calcular resultado = N x contador
    Mostrar mensaje: N por contador = resultado
    Incrementar el contador en una unidad
  Hasta contador>10
Fin
```

OBSERVACIONES SOBRE LOS BUCLES

Las estructuras iterativas (también llamadas bucles) son las más complejas y requieren que les prestemos un poco más de atención. Todos los bucles, ya sean de condición al principio, de condición al final o de rango, tienen estos elementos:

- **Variable de control:** Es una variable que juega un papel decisivo en el bucle, pues todos sus elementos dependen de esta variable. Pueden ser varias pero de momento nuestros bucles serán sencillos y tendrán una única variable de control.

En los *ejemplos 7 y 10* la variable de control es *contador*; en los *ejemplos 8 y 9* la variable de control es *numero*.

- **Condición de salida:** Es la condición de cuya veracidad depende seguir ejecutando el cuerpo del bucle o salir de esta estructura. Naturalmente, la variable de control estará implicada en la condición de salida.

En el *ejemplo 7* la condición de salida es *contador<=3*, en el *ejemplo 8* la condición de salida es *numero es par*, en el *ejemplo 9* la condición de salida no está tan visible pero sería algo así como *cuando numero esté fuera del rango 1-7*.

- **Iniciación previa (de la variable de control):** Antes de que se evalúe la condición de salida habrá que iniciar la variable de control (es decir, asignarle un valor), porque de no ser así no podrá evaluar la condición.

En el *ejemplo 7* la iniciación es *Establecer contador=1*, en el *ejemplo 8* la iniciación es *Esperar numero*. En este último caso la iniciación está dentro del bucle pero antes de que se evalúe por primera vez la condición de salida. En el *ejemplo 9* la iniciación está implícita en la cabecera asignando a la variable *numero* el valor *1* para la primera iteración.

- **Modificación en el cuerpo del bucle (de la variable de control):** Dentro del cuerpo del bucle, que se ejecuta una y otra vez, será necesario que se modifique la variable de control. Si eso no ocurre, cuando la ejecución entra dentro del bucle ya no saldrá nunca (*bucle infinito*) porque si no cambia la variable de control, no cambiará el resultado de la condición de salida. Si la primera vez es verdadera, siempre será verdadera y si la primera vez es falsa, siempre será falsa.

LAS CONDICIONES COMPLEJAS

A veces las condiciones no son tan simples como las que hemos utilizado en los ejemplos anteriores. Podemos hacerlas más complejas usando el **Y** lógico y el **O** lógico.

- El **Y lógico**: Decimos que la condición compuesta **condicion1 Y condicion2** es verdadera si y solo si las dos condiciones lo son.

Ejemplo 11: algoritmo para distinguir entre jubilados y jubiladas

```

Inicio
  Preguntar edad
  Preguntar genero
  Si edad >=65 Y genero=varon
    Mostrar "usted es jubilado"
  Fin_si
  Si edad >=65 Y genero=mujer
    Mostrar "usted es jubilada"
  Fin_si
Fin

```

Observa que para que se muestre el rótulo "**Jubilado**" es necesario que se cumplan dos condiciones: **la edad tiene que ser mayor o igual que 65** y además **el género tiene que ser varón**. Si alguna (o ambas) es falsa, el conjunto se evalúa como falso.

- El **O lógico**: Decimos que la condición compuesta **condicion1 O condicion2** es verdadera si al menos una de las dos condiciones lo es.

Ejemplo 12: algoritmo para calcular el descuento de la entrada al estadio de fútbol

```

Inicio
  Preguntar la edad
  Preguntar si el usuario es socio
  Si edad<18 O el usuario es socio
    descuento = 50
  caso contrario
    descuento = 0
  Fin sin
Fin

```

Fijate que, en este caso, para que se aplique el descuento, basta con que se cumpla una de las condiciones: **la edad es menor de 18** (aunque no sea socio) o que **el usuario es socio** (aunque no sea menor de edad). Si se cumplen ambas, el conjunto también se evalúa como verdadero. Solamente cuando las dos sean falsas, se evaluará como falso el conjunto

Condición 1	Condición 2	Condición1 Y Condición 2	Condición1 O Condición 2
Verdadero	Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso	Verdadero
Falso	Verdadero	Falso	Verdadero
Falso	Falso	Falso	Falso

CONTADORES Y ACUMULADORES

En las estructuras iterativas, a veces, aparecen unas variables con una labor muy concreta, como los *contadores* y los *acumuladores*.

Los *contadores* son variables que cuentan las iteraciones que hace el bucle (u otras cosas). Se suelen inicializar a *0* o a *1* antes del bucle e incrementarse en una unidad en cada iteración. Como ya sabes, los bucles *de rango* usan un contador de forma implícita. En los otros bucles tendremos que hacerlo de forma manual. El *ejemplo 8* pedía al usuario un número par. Ahora modificamos ese algoritmo para saber cuantos intentos hace el usuario hasta que introduce ese número par.:

Ejemplo 13: Esperar un número par... contabilizando los intentos

```
Inicio
    contador = 0
Repetir
    Mostrar mensaje: "Introduzca un número par"
    Esperar numero
    contador = contador + 1
Hasta numero es par
    Mostrar mensaje: "Has necesitado " contador " intentos"
Fin
```

Observa que *contador* contiene el "número de veces que el usuario ha introducido un número". Al principio lo inicializamos a cero (el usuario aún no lo ha introducido ninguna vez) y después, cada vez que recogemos un número del usuario, incrementamos el contador en una unidad.

Nota: La expresión *contador = contador + 1* es una expresión que en matemáticas no tiene sentido. No es una igualdad, sino una asignación. En la asignación se evalúa la expresión de la derecha y el resultado se asigna a la variable de la izquierda. Por tanto, esta expresión significa que tomamos el valor de *contador*, le sumamos *1* y el resultado se asigna a la variable *contador*.

La asignación es una operación destructiva: eso significa que cuando a la variable se le asigna un valor desaparece cualquier valor anterior.

En un mismo bucle podemos necesitar más de un contador. En el siguiente ejemplo vamos a pedir al usuario 10 números y contaremos cuántos son positivos, cuántos negativos y cuántos cero.

Ejemplo 14: Contando positivos, negativos y ceros

```
Inicio
    positivos = 0
    negativos = 0
    ceros = 0
    Para indice desde 1 hasta 10
        Mostrar mensaje: "Introduzca un número"
        Esperar numero
        Si numero > 0
            positivos = positivos + 1
        Caso_contrario
            Si numero < 0
                negativos = negativos + 1
            Caso_contrario
                ceros = ceros + 1
        Fin_si
    Fin_para
    Mostrar mensaje: "Hay " positivos " positivos, " negativos " negativos y" ceros " ceros"
Fin
```

Nota: Observa que las variables que usamos de contadores no tienen que llamarse *contador*.

Por otra parte, los *acumuladores* son variables que permiten ir almacenando parcialmente un resultado que se calcula a lo largo de cada una de las iteraciones. Supongamos que se le van a pedir al usuario 1000 números para calcular la suma de todos ellos. La variable que vamos a usar para solicitar todos los números es la misma. Eso quiere decir que cada cantidad desaparece cada vez que se introduce la siguiente. El acumulador nos permite ir sumando cada cantidad antes de pedir la siguiente.

Ejemplo 15: Sumar 1000 números que introduce el usuario

```
Inicio
    suma = 0
    Para indice desde 1 hasta 1000
        Pedir numero
        suma = suma + numero
    Fin_para
    Mostrar suma
Fin
```

Observa que en este ejemplo el acumulador se llama *suma*, porque alojará, al final, el resultado de toda la suma. Es necesario inicializar esta variable antes de entrar en el bucle porque, de no ser así, cuando intentemos sumarle su valor al primer número habría un error: no se podría sumar el valor de *suma* y *numero* si no sabemos que valor tiene *suma*.

Además, el valor inicial de suma tiene que ser cero, porque si fuese otro valor alteraría el valor de la suma final.

Ejemplo 16: Multiplicar 10 números que introduce el usuario**Inicio****producto = 1****Para indice desde 1 hasta 10****Pedir numero****producto = producto * numero****Fin_para****Mostrar producto****Fin**

En este otro ejemplo, el acumulador (**producto**) se inicializa con el valor **1**. Puesto que ahora estamos multiplicando los números, ese valor es el que no altera el resultado.

DIAGRAMAS QUE USAMOS EN LA FASE DE DISEÑO

De la misma forma que en la fase de análisis usamos diagramas para dejar detallado todo el trabajo realizado, también en la fase de diseño disponemos de otro conjunto de diagramas.

Si hemos optado por un diseño orientado a objetos, los diagramas los realizaremos con **UML** (Unified Modeling Language). Dejamos esta herramienta apartada de momento porque la estudiaremos en detalle a lo largo del curso.

Si hemos optado por un diseño estructurado podremos ayudarnos con los diagramas de flujo, los diagramas de cajas y el pseudocódigo (entre otras muchas alternativas):

- **Diagramas de flujo:** Utilizamos paralelogramos para las instrucciones de entrada y salida, rombos para las condiciones lógicas, flechas para el flujo de control y rectángulos para el resto de instrucciones, entre otros elementos.
- **Diagramas de cajas:** Cada instrucción se representa como una caja. Las instrucciones se agrupan en cajas más grandes que contienen las estructuras y el programa es la caja más grande que contiene todo lo demás.
- **Pseudocódigo:** Es un texto descriptivo para diseñar el algoritmo. Mezcla palabras del lenguaje natural con condiciones sintácticas y palabras clave. No hay un estándar definido y dependerá mucho de la persona que lo escriba. Los cuadros anteriores con los ejemplos de las estructuras básicas de un diseño estructurado son pseudocódigo.

Figura 1: Estructura secuencial

Diagrama de flujo	Diagrama de cajas	Pseudocódigo
		Instrucción 1 Instrucción 2 Instrucción 3

Figura 2: Estructura condicional

Diagrama de flujo	Diagrama de cajas	Pseudocódigo
		Si <condición> instr 1 Caso contrario instr 2 Fin_si

Figura 3: Selección múltiple

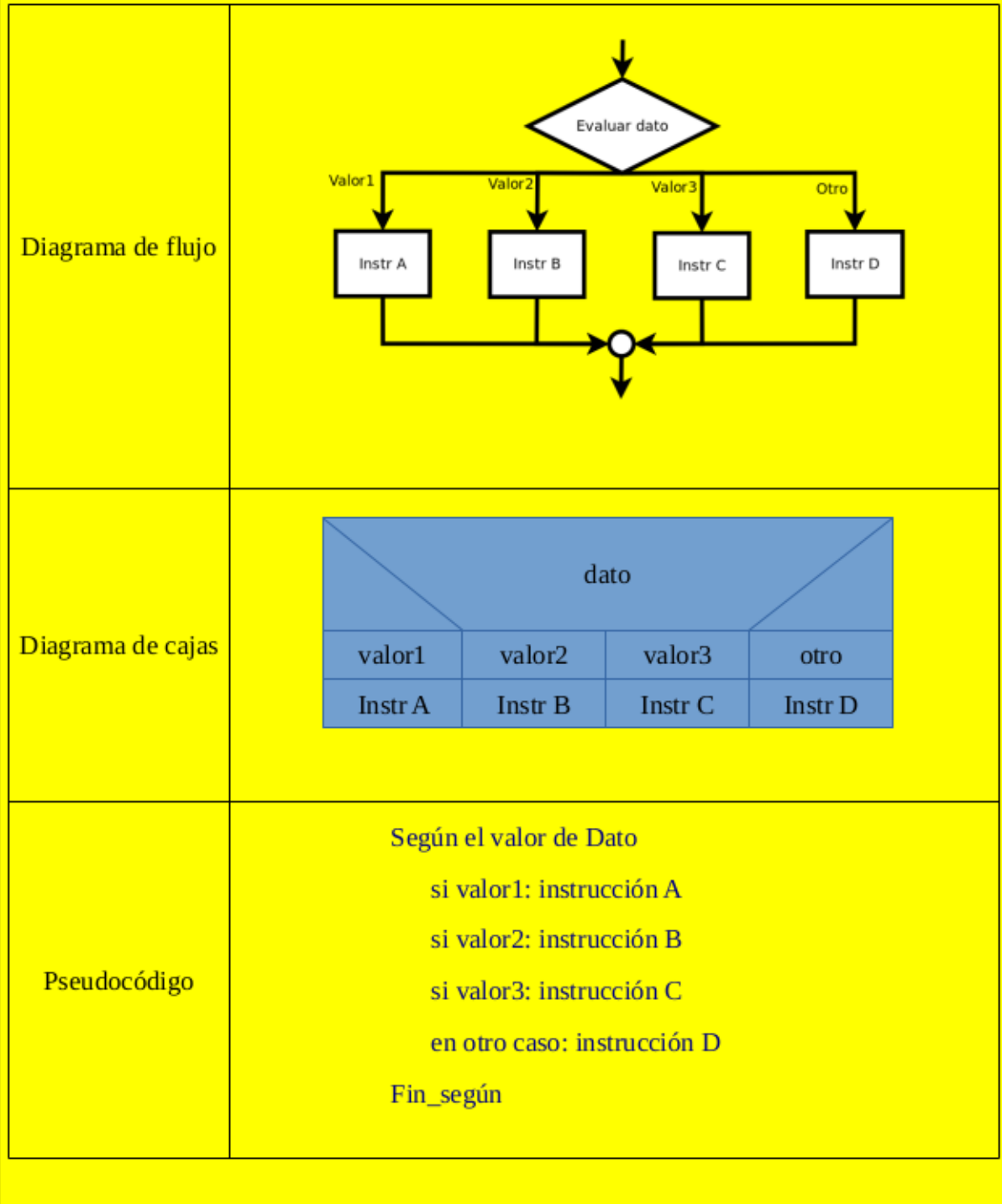


Figura 4: Estructura repetitiva: Bucle de condición al principio

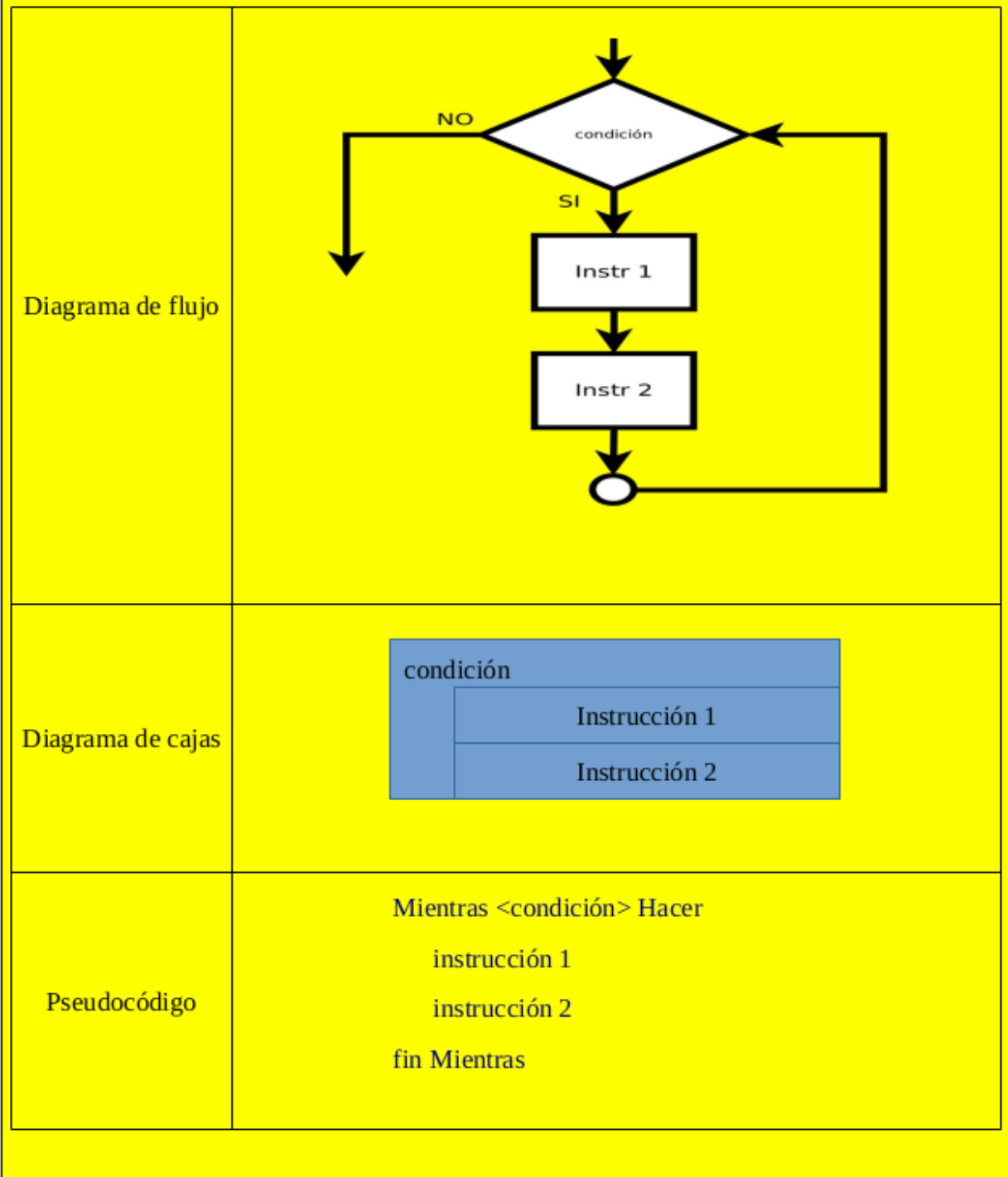


Figura 5: Estructura repetitiva: Bucle de condición al final

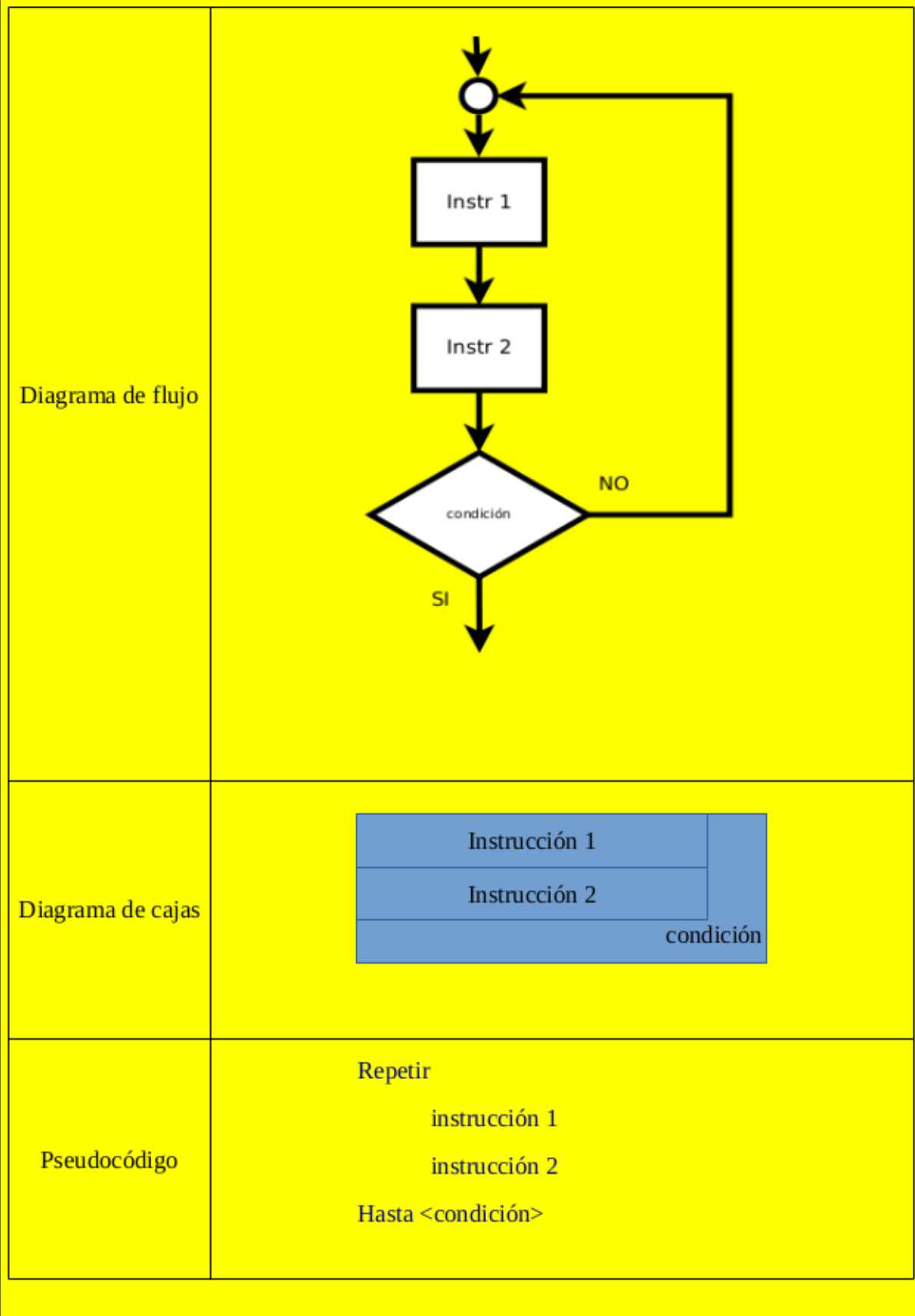
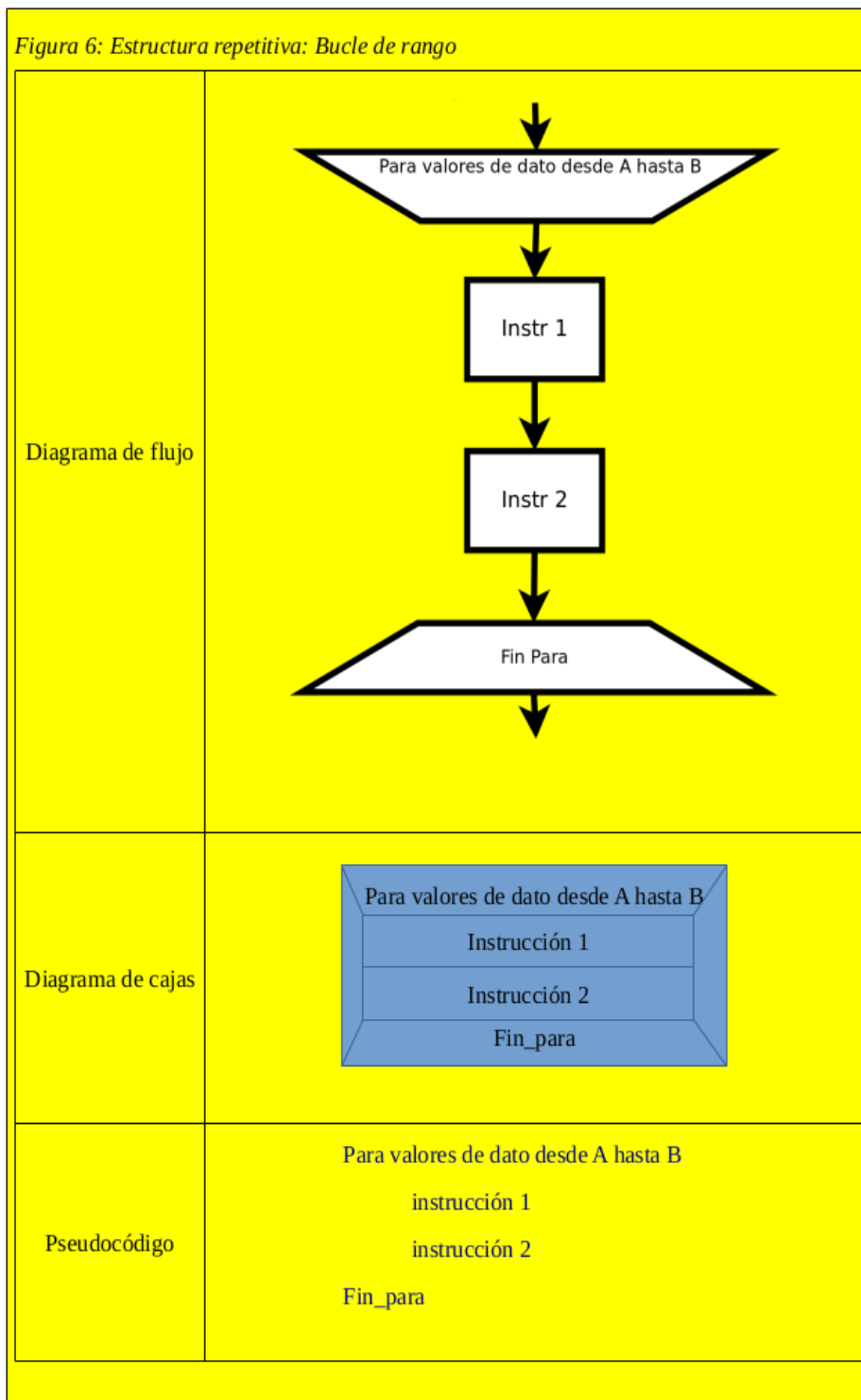


Figura 6: Estructura repetitiva: Bucle de rango



ERRORES HABITUALES

Las distintas estructuras que hemos visto se pueden mezclar y anidar tanto como queramos, pero sin modificar su esencia. Así, dentro de uno de los caminos de una estructura condicional será posible incluir una estructura iterativa y dentro del cuerpo de esta podrá haber otras condiciones o incluso otros bucles.

Pero para respetar la programación estructurada los bucles no podrán tener salidas extra ni vueltas a la condición antes de terminar todo el bloque de instrucciones.

Los siguientes son **EJEMPLOS PROHIBIDOS** en programación estructurada y en este módulo.

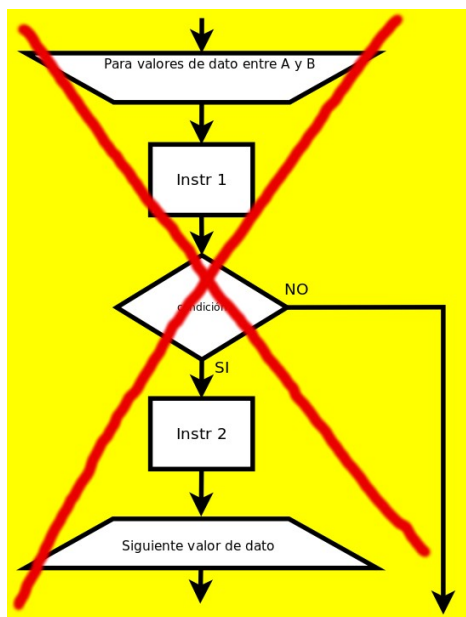
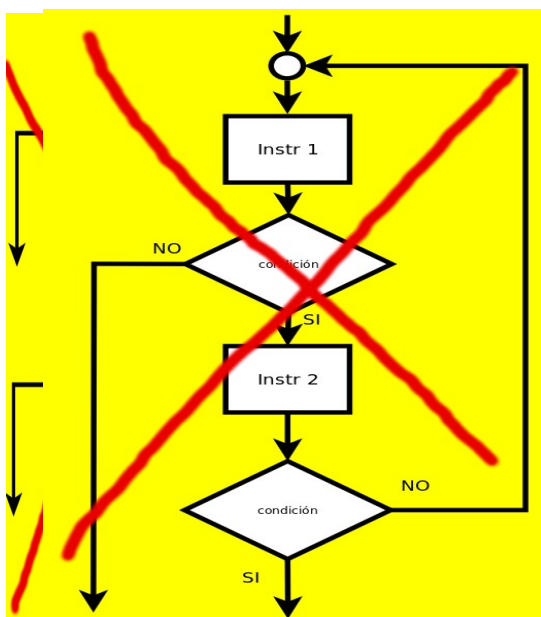
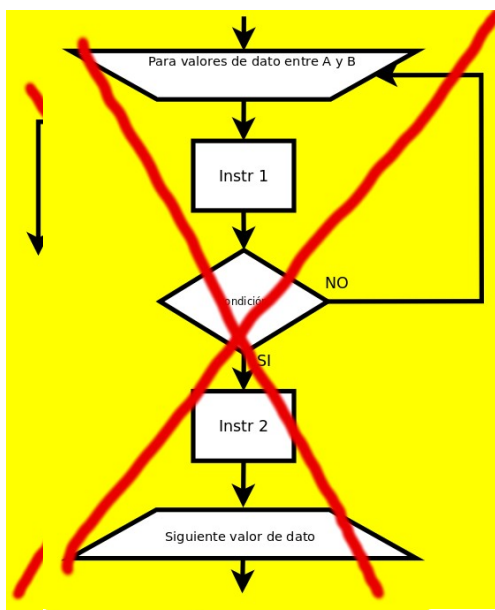








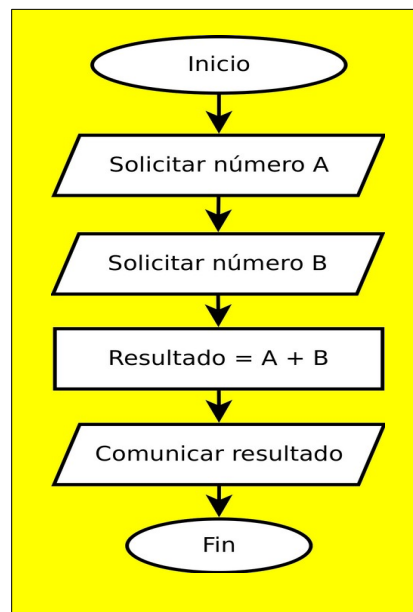
DIAGRAMA DE FLUJO PARA UN PROGRAMA COMPLETO

En los ejemplos anteriores los diagramas de flujo representan parte de un programa. Cuando creamos el diagrama de flujo para un programa completo tenemos que tener en cuenta que todos los programas tienen **un único principio** y **un único fin** que representamos con **elipses**.

Además distinguimos dibujos distintos para instrucciones distintas.

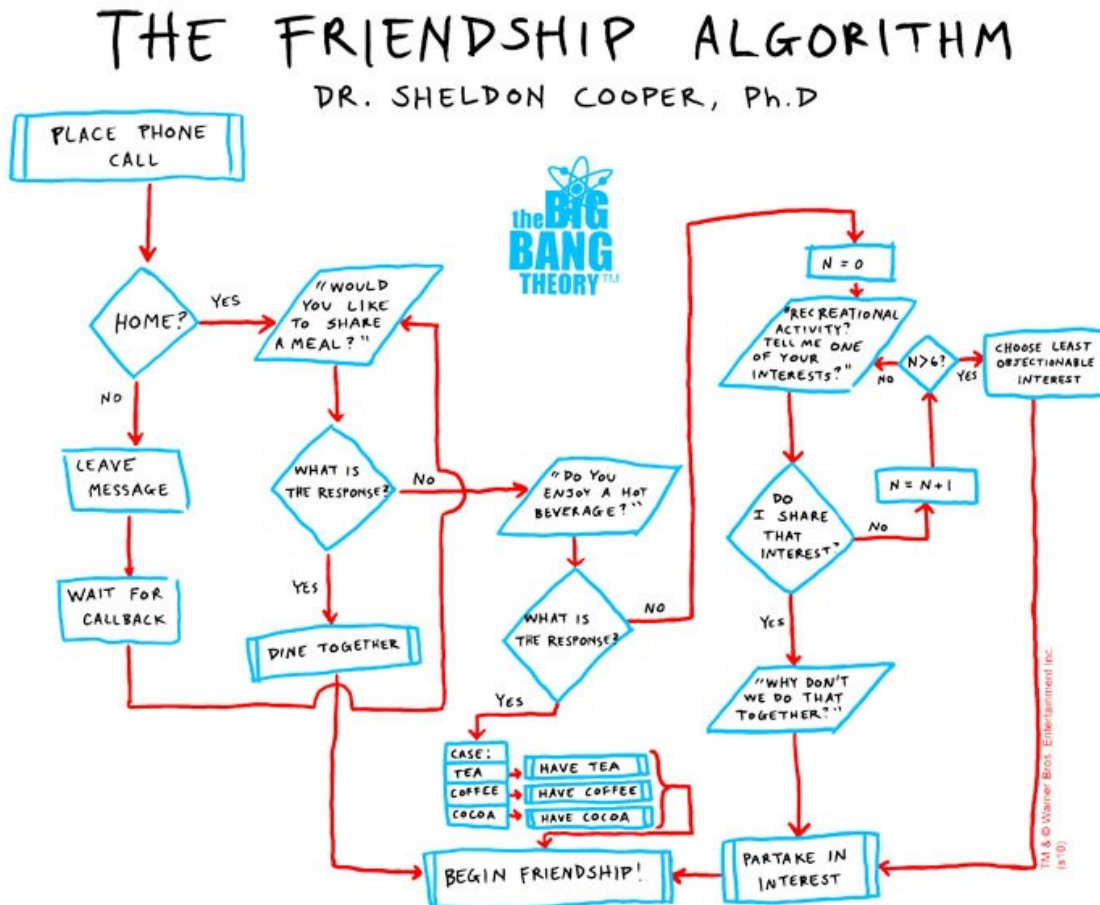
Símbolo	Significado
	Inicio de programa
	Instrucción de proceso.
	Instrucción de entrada o salida, como leer un dato del teclado o comunicar un dato por pantalla.
	Decisión.
	Subprograma. Hace referencia a un programa completo ya definido.
	Final de programa

Ejemplo 1: Diagrama de flujo para el programa completo "**Sumar dos números**"



Ejemplo 2: Diagrama de flujo de la amistad

En <https://www.youtube.com/watch?v=uFUboyAX1b8> puedes ver una divertida escena de la serie **Big Bang Theory** en la que **Sheldon Cooper** crea el algoritmo para hacer amigos. Su algoritmo tiene un error y cae en un bucle infinito. Por suerte, su compañero **Howard Wolowitz** añade una condición de salida para que, usando un contador, no pase de 6 iteraciones.



1.2.3 Codificación

Una vez terminado el diseño tendremos los diagramas de flujo de todos los procesos que constituyen nuestra aplicación. En la fase de codificación tendremos que volver a escribir todos los algoritmos, pero ahora en un lenguaje de programación. Eso es lo que se llama **código fuente**.

Un **programa** es **un algoritmo detallado de forma que pueda ser ejecutado por un ordenador**.

Para escribir el **código fuente** tendremos que conocer el lenguaje de programación con su sintaxis correcta. Ahora no existe la libertad que teníamos en pseudocódigo para expresar una instrucción, en el lenguaje de programación cada palabra que se escribe tiene una combinación exacta de mayúsculas/minúsculas, paréntesis, espacios, etc.

Es cierto que aún queda cierto margen de libertad que permite que cada programador tenga su propio estilo, pero cada vez más se tiende hacia un estilo común que permita hacer más legible el código. Tener un estilo común es muy importante en un equipo de programadores, donde cada uno de ellos se tendrá que enfrentar a las modificaciones del código que ha escrito otro compañero, pero

incluso en los pocos casos en los que sólo tú vayas a ver ese código, tener un buen estilo será más importante de lo que, de entrada, puedas pensar.

Herramientas como **Eclipse** o **Netbeans** ofrecen buenas ayudas para homogeneizar el estilo.

```
/* Ejemplo de un programa escrito en C para sumar dos números */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
void main()  
{  
    int a;  
    int b;  
    int resultado;  
  
    printf("Programa para sumar dos números:\n");  
  
    /* Preguntamos los números */  
    printf("Introduzca el primer número: ");  
    scanf("%i", &a);  
    printf("Introduzca el segundo número: ");  
    scanf("%i", &b);  
  
    /* Calculamos la suma */  
    resultado = a + b;  
  
    /*Comunicamos el resultado */  
    printf("La suma de %d y %d es: %d", a, b, resultado);  
    printf("\nFin de programa...\n");  
}
```

Dentro del código debemos incluir también **comentarios**. Los comentarios son líneas que no aportan nada al programa, de hecho, cuando se esté ejecutando, se ignorarán esas líneas. Los comentarios van dirigidos al programador y a los compañeros que deban revisarlo posteriormente.

```
//Ejemplo de programa escrito en Java para sumar dos números  
import java.util.Scanner;  
  
public class Suma {  
    public static void main(String[] args){  
        int a;  
        int b;  
        int resultado;  
  
        Scanner teclado=new Scanner(System.in);  
        System.out.println("Programa para sumar dos números:");  
        System.out.println("Introduzca el primer número: ");  
        a=teclado.nextInt();  
        System.out.println("Introduzca el segundo número: ");  
        b=teclado.nextInt();  
        resultado = a + b;  
        System.out.println("La suma de "+a+" y "+b+" es: "+resultado);  
        System.out.println("Fin del programa.");  
    }  
}
```

Para generar el **código fuente** sólo necesitamos un **editor de textos** (Ojo!!! no confundir con un procesador de textos) con el que escribimos el programa según las reglas de un lenguaje de programación. El nombre del fichero que creamos tiene que tener la extensión que indica el lenguaje que hemos usado.

Por ejemplo:

<code>suma.c</code>	(si hemos escrito el programa con lenguaje C)
<code>Suma.java</code>	(si hemos escrito el programa con lenguaje Java)

COMPILACIÓN

Una vez que terminamos de escribir el **código fuente**, necesitamos traducirlo a **código objeto**. La traducción de **código fuente** a **código objeto** se hace mediante unos programas llamados **compiladores** o **intérpretes**, según el tipo de traducción que hagan.

Ejemplo: compilación y ejecución de un programa en C

Con esta instrucción en el terminal **Linux** se genera un nuevo fichero, **suma**, que es ejecutable:

```
prompt$ gcc suma.c -o suma
```

Para ejecutar el programa usamos la siguiente instrucción:

```
prompt$ ./suma
```

Ejemplo: compilación y ejecución de un programa en Java

Con esta instrucción en **Linux** se genera un nuevo fichero, **Suma.class**, que es ejecutable:

```
prompt$ javac Suma.java
```

Para ejecutar el programa usamos la siguiente instrucción:

```
prompt$ java Suma
```

Durante el proceso de compilación se hace un análisis del código para ver si es correcto. Cuando se detectan errores, se aborta el proceso de compilación y se informa del problema encontrado del modo más detallado posible, para que se pueda solucionar antes de volver a compilar.

Para probarlo puedes introducir errores intencionados en los ejemplos anteriores (borrar un paréntesis, etc.) y volver a compilar.

1.2.4 Pruebas

Durante esta etapa vamos a detectar errores, no sólo de codificación, sino también los posibles errores de etapas anteriores. Tendremos que diseñar pruebas que saquen a la luz esos errores con el menor tiempo y esfuerzo posible. **Una prueba tiene éxito cuando detecta un error nuevo.**

Un **caso de prueba** es un documento que especifica los valores de entrada, los valores de salida esperados y las condiciones previas a la ejecución de la prueba. Entre los valores de entrada, se deben incluir no sólo los valores válidos y esperados, también valores inválidos e inesperados. Por ejemplo, se trata de detectar cómo reacciona el programa cuando pide al usuario que introduzca su edad y éste introduce una letra.

Nota: Lo más conveniente es que las pruebas las haga un programador distinto al que ha realizado el programa.

Después de ejecutar las pruebas, se comparan los resultados obtenidos con los esperados y en caso de contradicción se identifican los errores. Se hará un informe incluyendo tanto las pruebas con resultados satisfactorios como las que han localizado algún error.

Si existe un error que no se localiza, habrá que diseñar nuevas pruebas para obtener más información. Cuando se localiza un error, se depura y se vuelven a hacer pruebas.

A la hora de diseñar los casos de pruebas se usan dos técnicas:

- **pruebas de caja blanca:** Se centran en validar la estructura interna del programa, analizando los detalles del código.
- **pruebas de caja negra:** Sólo prestan atención a la entrada y la salida para comprobar que la funcionalidad del programa es correcta.

Ejemplo: Pruebas para el programa Suma

Ejecuta el programa anterior varias veces y comprueba que funciona correctamente para una muestra amplia de entradas. Analiza que pasa cuando introduces:

- números enteros positivos.
- números negativos.
- números con parte decimal.
- Letras.

1.2.5 Documentación

La documentación del programa se va elaborando en cada una de las etapas. Ahora de lo que se trata es de reunir y clasificar toda la documentación generada.

Hay que distinguir dos partes en la documentación:

- **La documentación del proceso:** Es una documentación técnica que usarán los miembros del equipo de desarrollo, los de mantenimiento y los gestores de la empresa. Será un instrumento útil para:
 - ◆ Comunicar eficazmente a los miembros del equipo de desarrollo.
 - ◆ Ayudar al equipo de mantenimiento.
 - ◆ Ayudar a gestionar el presupuesto.
- **La documentación del producto:** Son los manuales de usuario. Deben de incluir los manuales para el usuario final y también para los administradores del sistema.

1.2.6 Mantenimiento

El mantenimiento es la modificación del software después de la entrega al usuario. Hay varios tipos de mantenimiento:

- **Mantenimiento adaptativo:** Para adaptar el programa a los cambios que se producen en el entorno original: cambio de equipos, cambio de sistemas operativos, etc.
- **Mantenimiento correctivo:** Para corregir los errores que se detecten.
- **Mantenimiento perfectivo:** Para añadir funciones adicionales que el cliente pida.
- **Mantenimiento preventivo:** Son cambios internos que deben ser transparentes al usuario. Son para mejorar el rendimiento de alguna función, mejorar la legibilidad del código, etc.

1.3 Concepto de programa

Un programa informático es un conjunto de instrucciones que, siguiendo un orden establecido, resuelven un problema o parte de él. Tal y como hemos visto, el programador escribe ese programa en un lenguaje de programación, sin ambigüedades, que pueden entender los humanos (**código fuente**). Posteriormente y mediante un compilador, este programa es traducido a un lenguaje que puede entender la máquina: el **lenguaje máquina (código objeto)**. El lenguaje máquina está formado por un número de instrucciones muy básicas que son específicas del microprocesador concreto que lo va a ejecutar.

El programador no necesita conocer el lenguaje máquina, ya que es el compilador el que hace la traducción y tiene en cuenta qué tipo de microprocesador lo va a ejecutar.

Las instrucciones del lenguaje máquina son tan simples como "recuperar un dato de una posición de memoria", "sumar el contenido de dos registros de la **CPU**", "incrementar en uno el valor de un registro", etc. Ya te puedes imaginar que cada instrucción del lenguaje que usa el programador se traduce en varias de estas instrucciones máquina tan simples.

1.3.1 Elementos de la CPU

La **CPU** tiene básicamente tres elementos:

- La **Unidad Aritmético-Lógica (ALU)**: Hace operaciones matemáticas simples y operaciones lógicas con los datos que recibe.
- La **Unidad de Control (UC)**: Interpreta las instrucciones máquina y genera señales de control para ejecutarlas. Está formado por un decodificador, un reloj y los registros internos:
 - ➡ **Decodificador**: Analiza la instrucción en curso y genera las señales de control para ejecutarla.
 - ➡ **Reloj**: Proporciona impulsos eléctricos a intervalos constantes, marcando el ritmo de funcionamiento del decodificador.
 - ➡ Los **registros internos**:
 - ◆ **Contador de Programa**: Contiene la dirección de memoria donde se encuentra la siguiente instrucción del programa.
 - ◆ **Registro de Instrucción**: Contiene la instrucción que hay que ejecutar. Tiene dos partes: el código de instrucción que indica la operación que hay que realizar y la dirección de memoria donde está el dato que tenemos que usar.
 - ◆ **Registro de dirección de memoria**: Contiene una dirección de memoria que usaremos para la lectura o escritura de un dato.
 - ◆ **Registro de intercambio de memoria**: Contiene el dato que tenemos que escribir en memoria o que ya hemos leído de memoria.

Nota: Junto a este tema puedes ver la presentación **FuncionamientoCPU.pps** que muestra como se va ejecutando un programa simple a nivel interno.

1.4 Clasificación de los lenguajes de programación

1.4.1 Según el nivel de abstracción

- Lenguajes de **bajo nivel**: Son más cercanos a la máquina que al hombre. El lenguaje de más bajo nivel es el **lenguaje máquina** formado exclusivamente por ceros y unos, que es lo que entiende la máquina. Por encima del lenguaje máquina está el **lenguaje ensamblador**, difícil de aprender y específico de cada procesador, porque en realidad se trata de una traducción directa del lenguaje máquina a unos códigos nemotécnicos.

Ejemplo: Programa en lenguaje ensamblador para Intel 8088

```
MOV DX, 010B
MOV AH, 09
INT 21
MOV AH, 00
INT 21
```

Ejemplo: El programa anterior en lenguaje máquina (hexadecimal y binario)

BA	10111010
0B	00001011
01	00000001
B4	10110100
09	00001001
CD	11001101
21	00100001
B4	10110100
00	00000000
CD	11001101
21	00100001

- Lenguajes de **nivel medio**: Son lenguajes intermedios que permiten tratar elementos físicos del equipo directamente, pero con palabras del lenguaje natural (inglés). Un ejemplo de este tipo de lenguajes es el **C**.
- Lenguajes de **alto nivel**: Son más cercanos al hombre que a la máquina y para poderse ejecutar necesitan ser traducidos al lenguaje máquina (igual que ocurre con el ensamblador o los de nivel medio). El programador que usa estos lenguajes no tiene que conocer las particularidades internas de la máquina. Ejemplos de lenguajes de alto nivel: **BASIC**, **C++**, **C#**, **Java**, **Pascal**, **PHP**, **PL/SQL**, etc.

1.4.2 Según la forma de ejecución

- **Lenguajes compilados**: en estos lenguajes la traducción (del código fuente al objeto) se hace mediante un programa llamado **compilador**. El compilador hace primero un análisis sintáctico y, si todo va bien, genera el programa completo en un formato ejecutable. Ejemplos: **C**.
- **Lenguajes interpretados**: en estos lenguajes la traducción la hace un **intérprete**. En la interpretación no se genera un código objeto, sino que se traducen y se ejecutan una a una las instrucciones. Cuando termina la ejecución de una instrucción, se traduce la siguiente. Ejemplos: **BASIC**, **PHP**, **JavaScript**, ...

LA MÁQUINA VIRTUAL DE JAVA

Con el lenguaje **Java** se combinan la compilación y la interpretación. Primero se hace una compilación que genera el programa en un formato intermedio llamado **bytecodes** y después es interpretado por una máquina virtual.

La **máquina virtual Java** es una máquina virtual de proceso (no confundir con una máquina virtual de sistema como las que proporcionan virtualizadores como **VMware** o **VirtualBox**). Se trata de una máquina intermedia entre el hardware y el Sistema Operativo que permite que un programa funcione de la misma manera independientemente del Sistema Operativo con el que trabajemos.

Para poder ejecutar programas escritos en **Java**, necesitamos instalar el entorno de ejecución de la **máquina virtual Java (JRE)**, que puedes descargar gratuitamente, eligiendo el fichero adecuado a tu SO, desde www.oracle.com/technetwork/java/javase/downloads/index.html.

Para escribir un programa en **Java** podemos usar un editor de textos y crear un fichero con la extensión **.java** (por ejemplo, **MiPrograma.java**). A continuación lo compilamos y obtenemos un fichero con la extensión **.class** (por ejemplo, **MiPrograma.class**). Este último fichero está en el formato intermedio de **Java (bytecodes)** y estará listo para ejecutarse sobre cualquier **máquina virtual Java**, independientemente del hardware y Sistema Operativo que estemos usando.

Nota: Tienes un ejemplo de las instrucciones necesarias para compilar y ejecutar un programa en **Java** en el apartado de **Codificación**, subapartado **Compilación**, de este mismo tema.

1.4.3 Según el paradigma de programación

- **Lenguajes estructurados:** Los lenguajes estructurados son los que utilizan las tres estructuras mencionadas anteriormente en este tema (**secuencial**, **condicional** y **repetitiva**). Un programa escrito con un lenguaje estructurado permite leerlo de principio a fin sin perder la idea de lo que va haciendo. Sin embargo cuando se trata de un programa complejo, puede obtenerse un código demasiado largo, por eso, la evolución de la programación estructurada llevó a la **programación modular**, donde un problema complejo se descompone en múltiples problemas simples, todos ellos resueltos de forma estructurada con una sola entrada y una sola salida. La programación modular aporta otros beneficios: varios programadores pueden trabajar simultáneamente sobre módulos distintos y los módulos podrán reutilizarse en otros programas (reusabilidad del software).
- **Lenguajes orientados a objetos:** Con esta filosofía, un programa no será un conjunto de instrucciones, sino un conjunto de objetos. Cada objeto tiene unas características, llamadas **atributos** y es capaz de hacer determinadas operaciones, llamadas **métodos**. Los objetos se comunican unos con otros mediante mensajes. Cada objeto pertenece a una **clase**, que es una plantilla de ese tipo de objeto.

La Programación Orientada a Objetos (**POO**) facilita aún más el trabajo en equipo y la reutilización del software, pero es menos intuitiva y necesita mayor capacidad de abstracción por parte del programador.

1.5 Instalación de la máquina virtual de Java y el Kit de desarrollo de Java

Ya hemos visto que para poder ejecutar un programa **Java** necesitamos tener instalada la **máquina virtual Java**, conocida por **JRE** (Java Runtime Environment - Entorno de ejecución Java).

Si además queremos compilar un programa **Java** (traducir el código fuente de un fichero **.java** al código objeto en un fichero **.class**) necesitamos el **Kit de Desarrollo Java**, conocido como **JDK** (Java Development Kit).

Tienes que tener en cuenta que hay distintas máquinas de **Java**, desarrolladas por equipos diferentes que van ofreciendo las versiones con una numeración similar. Una de ellas es el proyecto abierto **OpenJDK**, la otra es de Oracle (**Oracle JDK**). En general puede valer cualquiera de ellas, pero a partir de la **versión 11** Oracle ha cambiado sustancialmente la licencia¹, de modo que es necesario pagar (no por el desarrollo de una aplicación con su **JDK**, pero si por el posterior uso de esa aplicación que hayas desarrollado).

Además, también desde la **versión 11**, **OpenJDK** ha mejorado su compatibilidad con el software de Oracle, de manera que lo normal será que usemos **OpenJDK** quedando el **JDK de Oracle** para situaciones profesionales donde se requiera un soporte más extenso.

En el momento de escribir este documento (septiembre de 2022) las últimas versiones son las que se indican en la siguiente tabla. Observa que no siempre se recomienda usar la última porque algunas versiones son más estables que otras o tienen un mantenimiento más prolongado (**LTS** - Long Time Support). En este anexo vamos a ver como puedes instalarlos en un sistema **Ubuntu** y en un sistema **Windows**.

The screenshot shows the Oracle Java website with a blue header containing 'Java', 'Descargar', 'Developer Resources', and 'Ayuda'. Below the header, there are sections for 'Recursos de ayuda' (Help Resources) and 'Descargas de Java para Linux' (Java Downloads for Linux). The 'Descargas de Java para Linux' section includes a 'Recomendación Version 8 Update 341' and a date of publication. A prominent yellow box contains 'Información importante sobre la licencia de Oracle Java', stating that the license has changed for versions published from April 16, 2019, onwards. Below this, there is a table for Linux downloads with columns for the download type, file size, and instructions. The table lists four options: Linux RPM (60.43 MB), Linux (90.22 MB), Linux x64 (90.58 MB), and Linux x64 RPM (60.12 MB), each with a corresponding 'Instrucciones' link. A note at the bottom of the table states: 'Tras instalar Java, tendrá que activarlo en su explorador.'

	OpenJDK		Oracle JDK	
	JRE	JDK	JRE	JDK
última versión	18	18	19	19
versión recomendada	11	11	8 ²	8

Nota: La versión que instales del **JRE** debe ser igual o superior a la del **JDK**. El comando **java** no ejecuta ningún programa que haya sido compilado con una versión superior.

Nota importante: Se recomienda usar la **versión 11 de OpenJDK**.

1 Amplía esta información en: <https://www.campusmvp.es/recursos/post/java-11-ya-esta-aqui-te-toca-pagar-a-oracle-o-cambiarte-a-otras-opciones.aspx>

2 Recomendación de la propia página de descarga de Oracle.

1.5.1 Instalación del JRE en Ubuntu

COMPROBAMOS SI YA TENEMOS EL JRE

Antes de nada vamos a comprobar si ya estuviera instalado, tecleando, desde un terminal:

```
java -version
```

Esta orden nos mostrará la versión del **JRE** instalado (eso indica que ya lo tenemos) o nos dirá que es un comando desconocido, lo que significa que no lo tenemos instalado.

INSTALACIÓN DEL JRE DE OPENJDK

Si no está, debemos descargar e instalar el paquete **openjdk-11-jre** (este paquete corresponde al proyecto **OpenJDK**) desde los repositorios de **Ubuntu**. Esto se puede hacer por dos caminos:

- Desde el terminal con la orden:

```
sudo apt-get install openjdk-11-jre
```

Nota: Los nombres de los paquetes usados en este documento pueden cambiar en posteriores versiones.

- Desde el gestor de paquetes **Synaptic**. Si no tenemos el **Synaptic** lo tendremos que instalar previamente desde el **centro de software de Ubuntu**.

INSTALACIÓN DEL JRE DE ORACLE

Si se prefiere la versión **Oracle JDK** tendrás que descargar el paquete correspondiente desde https://www.java.com/es/download/linux_manual.jsp.

Elige la descarga en el formato que necesites, por ejemplo, si usas **Ubuntu** elige **Linux x64**.

Es un fichero **.tar.gz** que se desempaqueta en la carpeta que desees, por ejemplo, **/opt/java/** con el siguiente comando:

```
sudo mkdir /opt/java
cd /opt/java
sudo tar xzvf nombreFichero.tar.gz
```

Nota: El texto amarillo hay que sustituirlo por el que corresponda en tu caso. No obstante, se aconseja hacer la instalación en la carpeta **/opt** que es el lugar adecuado para aplicaciones externas.

Después, podemos borrar el fichero **.tar.gz** para ahorrar espacio, porque ya no será necesario.

Nota: Recuerda que el ejecutable de **java** se localiza en **/opt/java/jre1.8.0_341/bin** y para que el sistema operativo pueda encontrarlo tendrás que añadir esa ruta a la variable de entorno **PATH**. Para hacerlo, edita el fichero **/etc/environment** y añade la ruta usando dos puntos (:) como separador. Para que el cambio sea efectivo tendrás que reiniciar la sesión.

Una vez hecha la instalación, ya sea **OpenJDK** u **Oracle JDK**, confirmamos con:

```
java -version
```

Y ahora si que debe mostrar la versión instalada.



1.5.2 Instalación del JDK en Ubuntu

COMPROBAMOS SI YA TENEMOS EL JDK

Antes de nada vamos a comprobar si ya estuviera instalado, tecleando, desde un terminal:

```
javac -version
```

Esta orden nos mostrará la versión del compilador (eso indica que ya lo tenemos) o, en caso de no estar, nos dirá que es un comando desconocido.

INSTALACIÓN DEL JDK DE OPENJDK

Si no lo tenemos, debemos descargar e instalar el paquete **openjdk-11-jdk** (este paquete corresponde al proyecto **OpenJDK**) desde los repositorios de **Ubuntu**, bien usando el **Synaptic** o bien desde el terminal con la orden:

```
sudo apt-get install openjdk-11-jdk
```

INSTALACIÓN DEL JDK DE ORACLE

Si se prefiere la versión **Oracle JDK** tendrás que descargar el paquete correspondiente desde <https://www.oracle.com/java/technologies/downloads/#java8>.

Elige la descarga en el formato que necesites, por ejemplo, si usas **Ubuntu** elige **x64 Compressed Archive**. Es un fichero **.tar.gz** que se desempaqueta en la carpeta que desees, por ejemplo, **/opt/java/** con el siguiente comando:

```
cd /opt/java  
sudo tar zxvf nombreFichero.tar.gz
```

Nota: El texto amarillo hay que sustituirlo por el que corresponda en tu caso. No obstante, se aconseja hacer la instalación en la carpeta **/opt** que es el lugar adecuado para aplicaciones externas.

Product/file description	File size	Download
ARM 64 RPM Package	59.15 MB	jdk-8u301-linux-aarch64.rpm
ARM 64 Compressed Archive	70.84 MB	jdk-8u301-linux-aarch64.tar.gz
ARM 32 Hard Float ABI	73.55 MB	jdk-8u301-linux-arm32-vfp-hflt.tar.gz
x86 RPM Package	109.49 MB	jdk-8u301-linux-i586.rpm
x86 Compressed Archive	138.48 MB	jdk-8u301-linux-i586.tar.gz
x64 RPM Package	109.24 MB	jdk-8u301-linux-x64.rpm
x64 Compressed Archive	138.78 MB	jdk-8u301-linux-x64.tar.gz

Después, podemos borrar el fichero **.tar.gz** para ahorrar espacio, porque ya no será necesario.

Nota: Recuerda que el ejecutable de **javac** se localiza en **/opt/java/jdk1.8.0_341/bin** y para que el sistema operativo pueda encontrarlo tendrás que añadir esa ruta a la variable de entorno **PATH**. Para hacerlo, edita el fichero **/etc/environment** y añade la ruta usando dos puntos (:) como separador. Para que el cambio sea efectivo tendrás que reiniciar la sesión.

Una vez hecha la instalación, ya sea **OpenJDK** u **Oracle JDK**, confirmamos con

```
javac -version
```

Y ahora si que debe mostrar la versión del compilador.

1.5.3 Instalación del JRE y JDK de OpenJDK en Windows

En la página <https://developers.redhat.com/products/openjdk/download> puedes encontrar todas las descargas del proyecto **OpenJDK** ordenados desde las más recientes. Como puedes ver en la imagen, hemos señalado los ficheros **MSI** para el **JRE** y el **JDK** de la **versión 11**. El instalador **MSI** hará la instalación completa, incluidos los cambios en la variable de entorno **PATH**, de forma automática.

jre-8u265-x86 ZIP	JRE 8 Windows 32-bit	Release date July 22, 2020	Download
jdk-8u265 Sources	OpenJDK 8 Source Code	Release date July 22, 2020	Download
jdk-11.0.8-x64 ZIP	OpenJDK 11 Windows 64-bit	Release date July 22, 2020	Download (240.66 MB)
jdk-11.0.8-x64 MSI	OpenJDK 11 Windows 64-bit	Release date July 22, 2020	Download (242.14 MB)
jre-11.0.8-x64 ZIP	JRE 11 Windows 64-bit	Release date July 22, 2020	Download (47.33 MB)
jre-11.0.8-x64 MSI	JRE 11 Windows 64-bit	Release date July 22, 2020	Download (48.61 MB)
jdk-11.0.8 Sources	OpenJDK 11 Source Code	Release date July 22, 2020	Download (106.62 MB)
jdk-14.0.2-x64 ZIP	OpenJDK 14 Windows 64-bit	Release date July 22, 2020	Download (251.67 MB)

Otra opción es descargar las versiones en **ZIP** (En nuestro último acceso el **JRE** solamente estaba disponible en formato **ZIP**). En ese caso, las descomprimos en la carpeta adecuada (por ejemplo, **C:\Program Files\openjdk\jre** y **C:\Program Files\openjdk\jdk**) y añadimos las rutas a la variable **PATH** (**C:\Program Files\openjdk\jre\bin** y **C:\Program Files\openjdk\jdk\bin**).

También en **Windows** puedes comprobar si ya dispones del **JRE** y del **JDK** con las instrucciones

```
java -version
```

y

```
javac -version
```

respectivamente.

1.5.4 Instalación del JRE y JDK de Oracle en Windows

Cuando instalamos el **JDK** también queda instalado el **JRE**. Si queremos instalar solamente el **JRE**, desde https://www.java.com/es/download/windows_manual.jsp bajamos la opción **Windows fuera de línea 64 bits**. Una vez ejecutado y terminada la instalación, volvemos a comprobar:

```
java -version
```

La instalación es completa e incluso modifica la variable **PATH**. La carpeta por defecto donde queda instalado el **JRE** es **C:\Program Files\Java\jre1.8.0_301**, aunque durante la instalación puede seleccionarse otra.

Con el mismo ejecutable podemos hacer la desinstalación.

Para el **JDK**, (se instalará también el **JRE**) desde <https://www.oracle.com/java/technologies/javase-downloads.html>, seleccionamos para descarga la última versión disponible de **java 8** para **Windows x64**, que será un archivo **.exe**.

Linux x64 Compressed Archive	138.22 MB	jdk-8u291-linux-x64.tar.gz
macOS x64	207.42 MB	jdk-8u291-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	133.69 MB	jdk-8u291-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	94.74 MB	jdk-8u291-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	134.48 MB	jdk-8u291-solaris-x64.tar.Z
Solaris x64	92.56 MB	jdk-8u291-solaris-x64.tar.gz
Windows x86	155.67 MB	jdk-8u291-windows-i586.exe
Windows x64	168.67 MB	jdk-8u291-windows-x64.exe

Tras la descarga, lo ejecutamos y se hará una instalación completa, incluida la modificación de la variable **PATH**, para acceder al **JRE** (el acceso al **JDK** desde la variable **PATH** tendremos que hacerlo manualmente). Las carpetas donde quedarán alojados los **JRE** y **JDK** serán respectivamente **C:\Program Files\Java\jre1.8.0_291** y **C:\Program Files\Java\jdk1.8.0_291**, aunque la podemos modificar durante la instalación.

Una vez terminada la instalación, volvemos a comprobarlo con:

```
javac -version
```

La desinstalación, en este caso, se debe hacer desde la utilidad de **Windows** para **Quitar programas**.

