

PROGRAMACIÓN I

T.U.P.

U.T.N. F.R.P.

1

Programación I

Estrategia

2

Repasemos :

Si Vs Según:

Que tipos de condiciones se analizan?

Cuantos casos por defecto se pueden colocar?

Iterativos :

Para que sirven ???

Cuales diferenciamos???

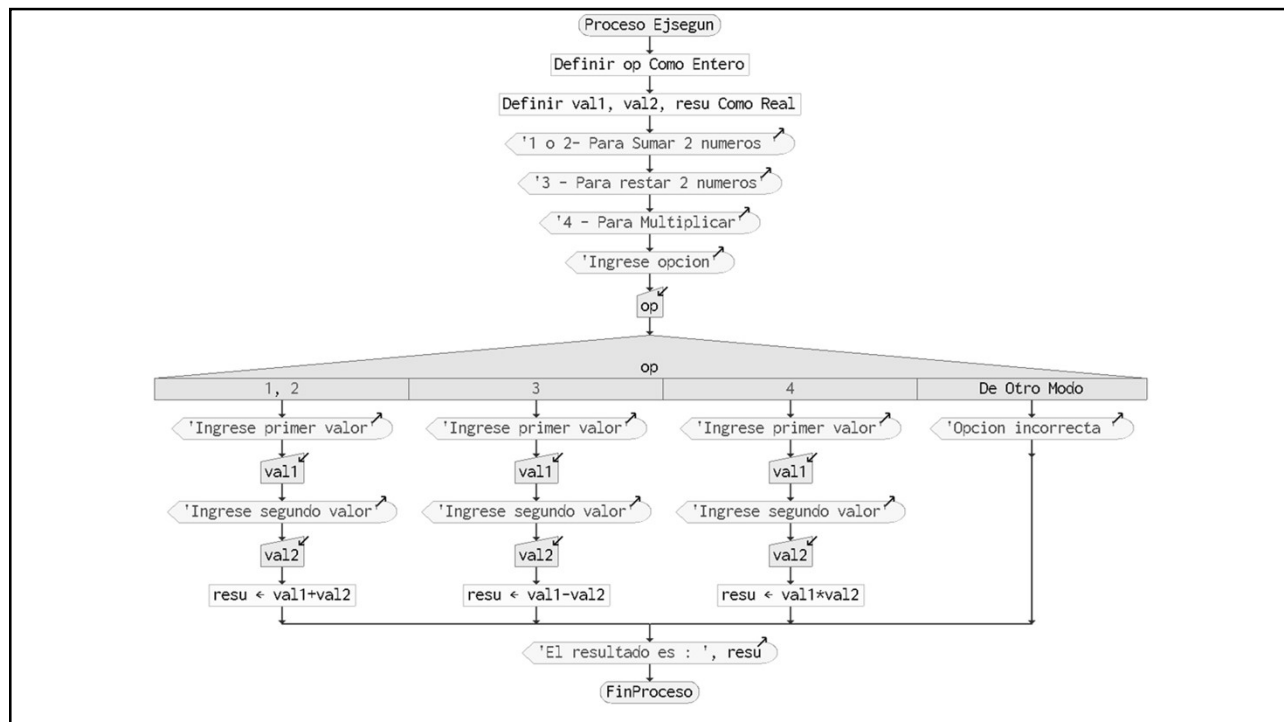
Que uso se le da a cada uno ????

PARA vs Mientras :

Como diferenciamos los
mecanismos de control de bucle?

Cuando uso cada una?

2



3

4

Estructura PARA

En general el uso de la estructura Mientras se da cuando no es conocida la cantidad de iteraciones requeridas en una aplicación.

Sabemos que hay situaciones en las cuales la cantidad de iteraciones es conocida de antemano:

“Se desea calcular el promedio de 4 valores ingresados por el usuario”

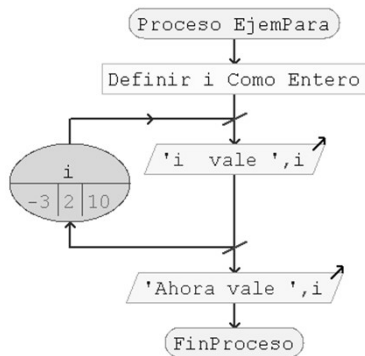
“Se necesita conocer cuantos valores se encuentran en el intervalo A y B siendo divisibles por 3”

4

Estructura PARA Representación

5

Diagrama



Pseudocodigo

```

Proceso EjemPara
Definir i Como Entero;
Para i<=-3 Hasta 10 Con Paso 2 Hacer
    Escribir 'i vale ',i;
FinPara
Escribir 'Ahora vale ',i;
FinProceso
  
```

5

6

Casos

6

Estrategias con repeticiones

7

Ejercicio :

Se requiere una aplicación que permita determinar el valor mayor y menor de una lista de números enteros ingresados por el usuario.

Considere que la cantidad de valores es desconocida, el rango de valores de los números es los enteros positivos, es decir son todos mayores a cero.

7

Estrategia Con repeticiones

8

Ejercicio - Análisis

Datos – Entradas

Valores ingresados por el usuario.

Incógnitas

Valor Mayor M

Valor Menor m

Relaciones

Sea V el conjunto de valores ingresados por el usuario

Valor mayo = $\{x/x \in V \text{ y } M \in V \text{ y } M \geq x\}$

Valor menor = $\{x/x \in V \text{ y } x \geq m \text{ y } m \in V\}$

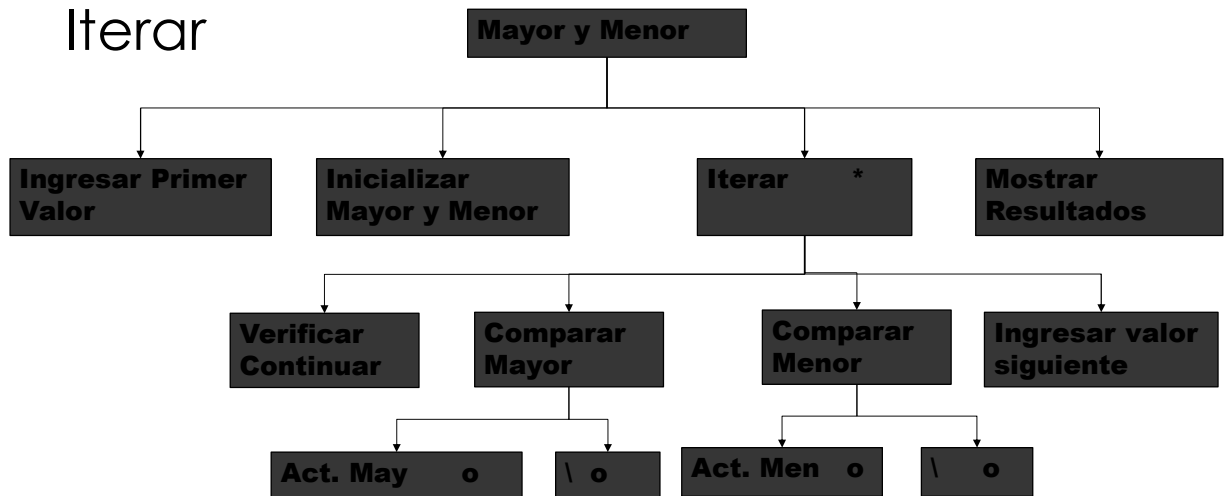
Como los valores son positivos consideramos que se ingresa cero para finalizar

8

Estrategias con Repeticiones

9

Iterar



9

Iteraciones

10

Ejercicio :

Se requiere una aplicación que permita determinar la nota promedio obtenida por un grupo de alumnos en un examen. La cantidad de alumnos es desconocida y es por esto que la aplicación deberá considerar que se termina la carga de los valores a considerar con el valor -1.

Considere que los valores de las calificaciones se encuentran en el rango de 0 a 10 .

Algunas consideraciones:

- El valor -1 no debe ser considerado a la hora de determinar el promedio.
- Si no se tienen notas válidas, no es posible calcular el promedio.

10

Análisis de iteraciones

11

Análisis

Datos – Entradas :

Notas de los alumnos

Incógnitas :

Nota Promedio del conjunto

Cantidad de notas

Relaciones :

Iniciar Suma de notas en cero y cantidad de notas en cero

Para cada nota ingresada

$\text{Suma de notas} = \text{Suma de notas} + \text{Nota}$

$\text{Cantidad de notas} = \text{Cantidad de notas} + 1$

Al finalizar

$\text{Promedio} = \text{Suma de notas} / \text{Cantidad de notas}$

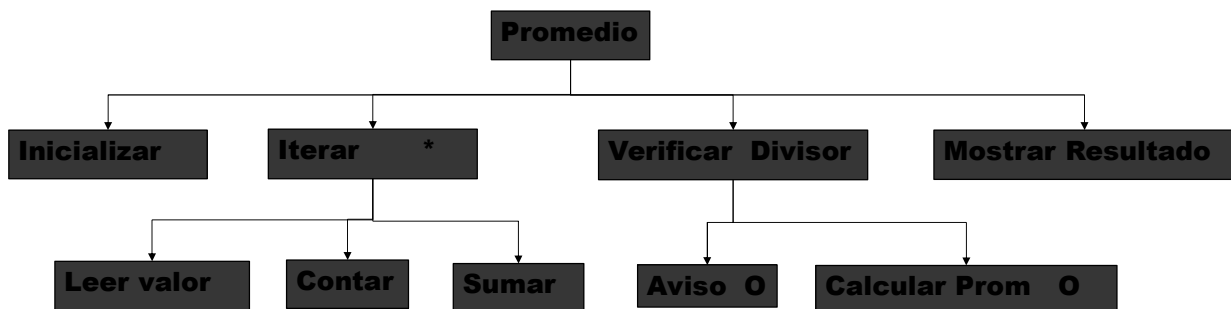
11

Estrategias Iterativas

12

Estrategias con iterativos

En los casos en que se realicen iteraciones el o los bloques que se asocian a las partes del problema se señalan con un *



12

Estrategia

13

Ej) Se necesita calcular el jornal de una lista de 20 operarios. Considerando que el valor a cobrar por hora y la cantidad de horas trabajadas varia de un operario a otro Indicar el valor del jornal a cobrar por cada operario considerando que las horas trabajadas después de la octava hora se consideran como extras y el valor a pagar por cada hora se incrementa en un 50%. Luego de Procesados los operarios indicar la suma total de jornales calculados

En este problema aparece un uso característico de los sistemas informáticos , relacionado con la repetición de actividades , automatización de procesos de cálculo o registración de múltiples eventos.

Una actividad muy común de estos casos es la de repetir cierta parte de la solución una cierta cantidad de veces que puede estar pre determinada o no.

En este tipo de problemas surge como parte de la estrategia la iteración , que relaciona todas las operaciones repetitivas que deben realizarse para resolver el problema. Los bloques destinados a reunir el conjunto de actividades que se repiten se marcan con un asterisco en la parte superior.
Este ejemplo particular combina también un análisis de condición excluyente, aunque no necesariamente forma parte de todas las actividades iterativas.

13

Análisis

14

Ejercicio 6) Se necesita calcular el jornal de una lista de 20 operarios. Considerando que el valor a cobrar por hora y la cantidad de horas trabajadas varia de un operario a otro Indicar el valor del jornal a cobrar por cada operario considerando que las horas trabajadas después de la octava hora se consideran como extras y el valor a pagar por cada hora se incrementa en un 50%. Luego de Procesados los operarios indicar la suma total de jornales calculados

Análisis : Para poder determinar el jornal de cada operario por cada uno de ellos debemos conocer la cantidad de horas trabajadas y el valor de la hora simple a cobrar . Además, el problema indica cuantos operarios son 20 en total.

Datos -Entradas : CantidadHoras , ValorHora y Cantidad de operarios (N).

Incógnitas : Jornal a cobrar por cada operario y SumaTotal

Relaciones : Primero debemos determinar si el operario supera ls 8 hs de trabajo o no ello nos indica que tipo de calculo realizar .

Si Cantidad de horas > 8 $\text{Jornal} = (\text{horas} - 8 * \text{ValorHora} * 1,5) + (8 * \text{ValorHora})$

SINO $\text{Jornal} = \text{CantidadHoras} * \text{ValorHora}$

Esto debe realizarse por cada operario , pero se deja explicito en la estrategia .

$\text{Suma total} = \text{Jornal operario1} + \text{Jornal operario2} + \dots + \dots + \text{Jornal OperarioN}$

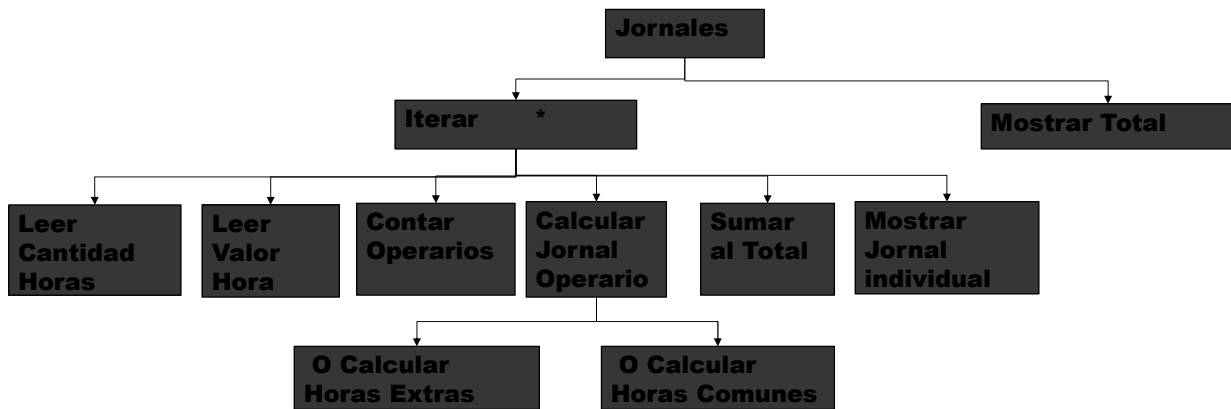
14

Estrategias Iterativas

15

Aquí es necesario hacer repetidamente el cálculo de cada operario , con los ingresos de datos de cada uno

Aquí veremos con * las actividades a realizar por cada operario y por separado el total que se indica al finalizar



15

Estrategias - Tipos de problemas

16

Estrategias según se trate de problemas de tipo:

Secuenciales

Aplanados

Entrada - Proceso - Salida

Condicionales

Sub problemas excluyentes

Entradas – Proceso - Decisión – Proceso – Salidas

Mayor cantidad de niveles que los secuenciales.

Iterativos

Múltiples niveles

Repeticiones

Sub problemas condicionales

Iteración (Entradas – Proceso Individual - Decisión) Proceso General – Salidas

16

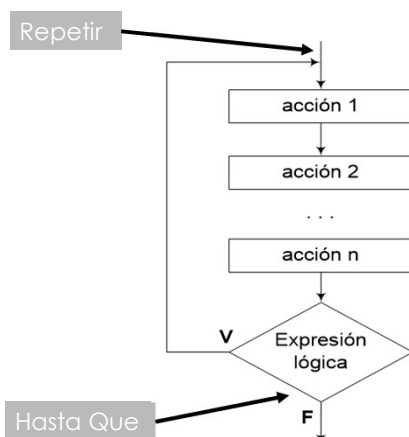
Muchas Gracias !!!!

17

Estructura Repetir

18

La representación de la estructura utilizando diagrama de flujo es:



- Siempre se ejecuta el conjunto de acciones asociado a la estructura, al menos la primera vez.
- Si la expresión lógica evaluada al final del bloque es verdadera, se inicia nuevamente la ejecución del bloque de acciones asociadas a la estructura.
- Cuando el resultado de evaluar la expresión lógica es falso, se finalizan las iteraciones, saliendo de la estructura.

18


Sintaxis Pseudocodigo - Repetir

19

La sintaxis de esta estructura es:

```

Repetir
  [acción 1];
  [acción 2];
  ...
  [acción n];
Hasta Que (exp lógica);
  
```



- El fragmento de código sobre el que la estructura se aplica es el delimitado por las palabras clave Repetir y Hasta Que.

- En una primera iteración todo el código es ejecutado.

- Finalizada la ejecución de la acción n, es evaluada la expresión lógica asociada a la palabra clave Hasta Que.

- Mientras la expresión lógica sea evaluada como verdadera, continuarán las iteraciones comenzando otra vez desde la primer acción después de la palabra **Repetir**.

- Cuando la evaluación de la expresión lógica dé como resultado falso, se finalizarán las iteraciones. Continuando con la siguiente instrucción después de la **(exp lógica)**;

- De este comportamiento se desprende que de la misma manera que con la estructura Mientras, el conjunto de acciones iterado debe modificar de alguna manera la expresión lógica evaluada.

19

Ejemplo Repetir

20

Se requiere una aplicación que calcule el promedio de 3 valores enteros positivos. La aplicación debe validar que los valores ingresados sean positivos y mostrar el valor del promedio calculado en pantalla.

20

Ejemplo Repetir

21

Una opción Proceso EjRepetir

Parte 1

Definir num, acum Como Entero;

Definir prom Como Real;

acum <- 0;

repetir

 escribir "Ingrese el primer valor valor positivo " ;

 leer num;

Hasta Que num > 0;

acum <- acum + num;

21

Ejemplo Repetir

22

Una opción

Parte 2

repetir

 escribir "Ingrese el segundo valor valor positivo " ;

 leer num;

Hasta Que num > 0;

 acum <- acum + num;

repetir

 escribir "Ingrese el tercer valor valor positivo " ;

 leer num;

Hasta Que num > 0;

 acum <- acum + num;

prom <- acum / 3;

Escribir "El promedio es " , prom;

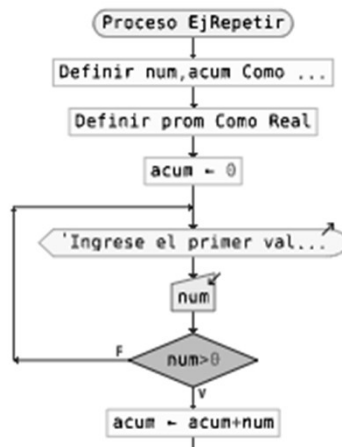
FinProceso

22

Ejemplo Repetir

23

Parte1

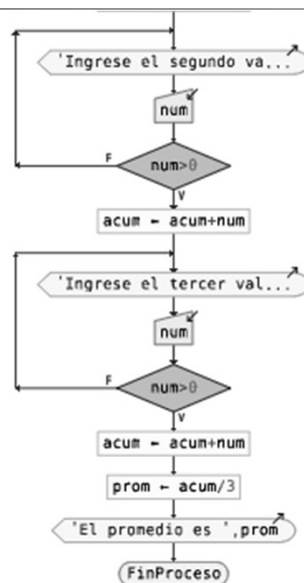


23

Ejemplo Repetir

24

Parte 2



24

Ejemplo Repetir y Para

25

Otra opción

Proceso EjRepetir

Definir num, i , acum Como Entero;

Definir prom Como Real;

acum <- 0;

Para i <- 1 hasta 3 Con Paso 1 Hacer

repetir

escribir "Ingrese el valor positivo ", i ;

leer num;

Hasta Que num > 0;

acum <- acum + num;

FinPara

prom <- acum / 3;

Escribir "El promedio es ", prom;

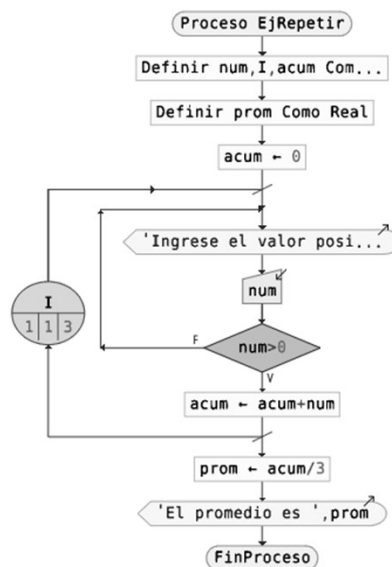
FinProceso

Uso de
repetir
dentro de
un PARA.
Validar un
valor
ingresado

25

Ejemplo Repetir y Para

26



26

27

Estructuras Iterativas

Usos recomendados

► Mientras (while)

- Se recomienda su uso cuando no se conoce o no se puede determinar dentro del algoritmo la cantidad de iteraciones necesarias. Usos de centinelas y banderas.

► Para (for)

- Se recomienda en los casos que se conoce, o el se puede determinar en el algoritmo, la cantidad de iteraciones necesarias.

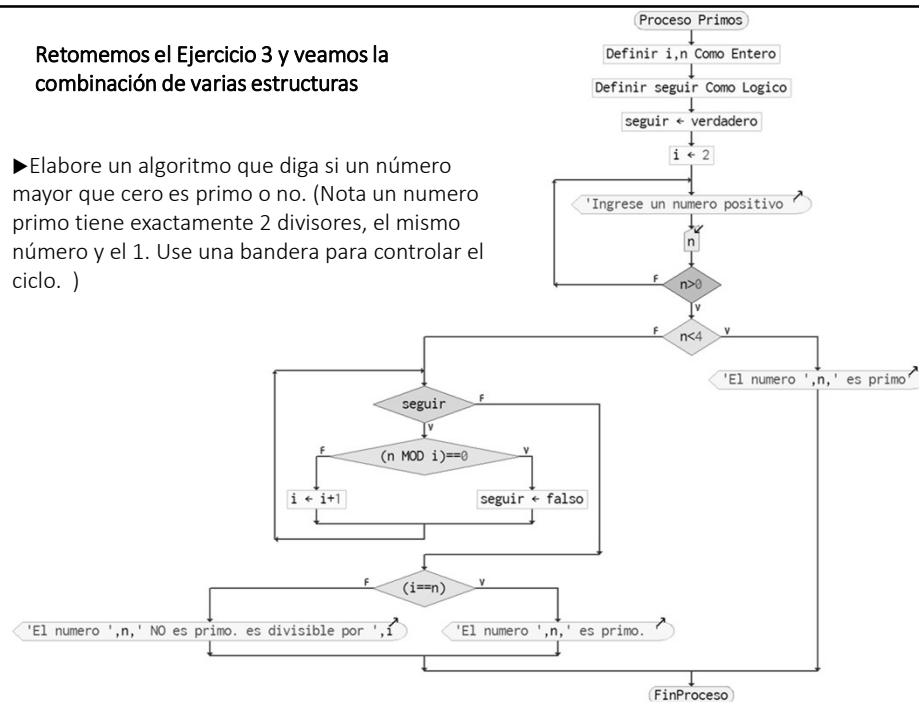
► Repetir (do while)

- Se recomienda su uso cuando no se conoce la cantidad de iteraciones pero se tiene certeza de que al menos una es necesaria.

27

Retomemos el Ejercicio 3 y veamos la combinación de varias estructuras

- Elabore un algoritmo que diga si un número mayor que cero es primo o no. (Nota un número primo tiene exactamente 2 divisores, el mismo número y el 1. Use una bandera para controlar el ciclo.)



28

Programación I

29

Repasemos :

Lenguajes de Programacion :

Para que sirve ???

En que casos la usamos???

Cuales conoce?

29

Programación I

- ▶ Formalmente un lenguaje tiene ciertas características .
 - ▶ Tienen reglas de semántica (Forma en que se escribe un programa Completo)
 - ▶ En algunos casos, la semántica no podrá determinarse en tiempo de compilación y, por lo tanto, deberá evaluarse en tiempo de ejecución o prueba de escritorio.
 - ▶ Tienen reglas de sintaxis (Reglas de construcción de las instrucciones) Que va primero y que va después en una instrucción o llamado a una subrutina.
 - ▶ Dichas características cumplen con ciertos criterios o reglas de diseño de aplicaciones.
 - ▶ El lenguaje de programación pone a disposición del programador ciertos recursos, técnicas y herramientas para su uso.
 - ▶ El enfoque de las herramientas técnicas y reglas de construcción derivan de un paradigma de programación.

30

30

Lenguaje

31

La mayor parte de definiciones que se dan sobre el lenguaje son discrepantes, aun así se dan ciertas regularidades y todas recogen de algún modo que:

- a) Es un sistema compuesto por unidades (los signos lingüísticos) cuya organización interna puede describirse estructural o funcionalmente.
- b) Su adquisición y uso posibilita formas específicas de relación y de acción sobre el medio (especialmente el medio social)
- c) El lenguaje se materializa en y da lugar a formas de conducta que permiten interpretarlo como un tipo de comportamiento.

El énfasis en un punto u otro es lo que identifica y diferencia a las disciplinas que estudian el lenguaje. Los componentes definitorios del lenguaje según su ámbito científico son la dimensión estructural o formal para conocer como es el sistema lingüístico en sí mismo, la dimensión funcional para conocer para qué les sirve a los usuarios el lenguaje y la dimensión comportamental para conocer cómo se usa el lenguaje cuando se producen y se comprenden los mensajes comunicativos.

31

Lenguaje de programación

32

El énfasis en un punto u otro es lo que identifica y diferencia a las disciplinas que estudian el lenguaje. Los componentes definitorios del lenguaje según su ámbito científico son la dimensión estructural o formal para conocer como es el sistema lingüístico en sí mismo, la dimensión funcional para conocer para qué les sirve a los usuarios el lenguaje y la dimensión comportamental para conocer cómo se usa el lenguaje cuando se producen y se comprenden los mensajes comunicativos.

La disciplina orientada a la programación de computadoras hace énfasis en la formalidad del lenguaje, son importantes los signos lingüísticos y definen comportamiento según su significado. Por ejemplo: un símbolo como $=$ o $>$ o $<$; tienen una interpretación particular en cierto lenguaje.

Los lenguajes de programación permiten la comunicación entre humanos y computadoras , con ellos los programadores indican que acciones deben realizar las computadoras.

32

Paradigma

33

Paradigma:

"Un paradigma es una forma de ver el mundo, una perspectiva general, una manera de fragmentar la complejidad del mundo real" (Patton, 1990).

"Conjunto de creencias y actitudes, como una visión del mundo compartida por un grupo de científicos que implica una metodología determinada" (Balina, s.f).

"En la ciencia un paradigma es un conjunto de realizaciones científicas universalmente reconocidas que, durante cierto tiempo proporcionan modelos de problemas y soluciones a una comunidad científica" (Kuhn,)

Que es un paradigma?

Algo que está constituido por los descubrimientos científicos universalmente reconocidos que, durante cierto tiempo, proporcionan a un grupo de investigadores problemas tipo y soluciones tipo (Marín, 2007). □ El conjunto de las creencias, valores reconocidos y técnicas que son comunes a los miembros de un grupo dado (Marín, 2007).

33

Paradigma según Khun

34

Definición de paradigma de Khun:

Estructura coherente constituida por una red de conceptos mediante la cual los científicos ven su campo.

Características fundamentales

- No está gobernado por reglas racionales algo histórico: el paradigma primero gobierna a los practicantes.
- Los paradigmas son inconmensurables: no existe una base común para compararlos.
- Son dogmáticos.

Frase de resumen: Las verdades de hoy serán los errores del mañana

34

Paradigmas de programación

35

Paradigmas de programación.

Paradigma de programación es una propuesta tecnológica adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que unívocamente trata de resolver uno o varios problemas claramente delimitados.

La resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de Software.

Los lenguajes de programación suelen implementar, a menudo de forma parcial, varios paradigmas

35

Paradigmas de programación

36

Paradigmas de programación.

Paradigma de programación es una propuesta tecnológica adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que unívocamente trata de resolver uno o varios problemas claramente delimitados.

Esto evidencia que los diferentes paradigmas cubren de manera parcial la complejidad de los problemas a ser representados. Que se han presentado varias alternativas a lo largo del tiempo, y que en las diferentes propuestas se han cubierto aspectos que en las anteriores habían alcanzado su límite.

Es importante acompañar el estudio del desarrollo de los diferentes paradigmas con los cambios producidos en el contexto histórico en el que fueron propuestos. De esta manera se tendrá una visión más objetiva y facilitara la comprensión de los principios asociados a cada uno de ellos.

36

Paradigmas de programación

37

Paradigmas de programación.

La resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de Software.

Los lenguajes de programación suelen implementar, a menudo de forma parcial, varios paradigmas

Si bien existe un cierto paralelismo entre lenguajes y paradigmas, es notable que no todos los lenguajes aplican la misma interpretación del paradigma, también los lenguajes aportan herramientas a los paradigmas que permiten mejorar su definición, completando la tecnología definida.

Por lo tanto no es lo mismo lenguaje que paradigma, siendo el paradigma una propuesta general de las técnicas y reglas de formación y los lenguajes instancias particulares de la implementación de dichas reglas.

El estudio del desarrollo de los paradigmas mostrara que el Paradigma de Orientación a Objetos toma como base el paradigma de la programación estructurada, que a su vez toma como base otros paradigmas anteriores. En cada definición de paradigma se incorporan modificaciones y reglas de construcción que permiten el cumplimiento del nuevo enfoque de pensamiento.

37

Paradigmas de programación

38

Fundamentación.

Paradigmas de programación El término Paradigma de Programación fue adoptado por Floyd[28], al recibir el Paradigma de Programación ACM Turing Award en 1978. Su idea es que varias de las observaciones de Kuhn sobre los paradigmas científicos, son pertinentes en el contexto particular de las Ciencias de la Computación,

Por ejemplo:

- Los libros de texto parecen implicar que el contenido de la ciencia está exclusivamente ejemplificado por las observaciones, leyes y teorías descritos en sus páginas. Esto suele ser el caso en los libros de computación que reducen esta ciencia a los algoritmos y lenguajes descritos en sus páginas.
- El estudio de los paradigmas constituye la fuente principal en la formación del estudiante, para preparar su integración a una comunidad científica particular donde llevará a cabo su práctica. En las Ciencias de la Computación es posible observar a diversas comunidades, hablando sus propios lenguajes y usando sus propios paradigmas. De hecho, los lenguajes de programación estimulan el uso de algunos paradigmas, mientras que inhiben notoriamente otros.

38

Paradigmas de programación

39

Justificación.

En opinión de Floyd, las dificultades típicas presentes en el desarrollo de programas de cómputo al considerar aspectos como confiabilidad, puntualidad, flexibilidad, eficiencia y costo; son reflejo de un repertorio inadecuado de paradigmas. Repertorio de paradigmas de programación, un conocimiento deficiente de los paradigmas existentes, la manera en que enseñamos esos paradigmas y en la manera en que los lenguajes de programación soportan, o no, los paradigmas de sus comunidades de usuarios.

Y concluye—Si el avance general del arte la programación requiere de la continua invención y elaboración de paradigmas; el avance individual requiere de la expansión del repertorio personal de paradigmas.

39

Paradigmas de programación

40

Historia de los paradigmas de Programación.

Al estudio de los lenguajes en cuanto al enfoque del proceso de programación se le denomina paradigmas de la programación, entendiéndose el término paradigma como la forma de ver y hacerlos programas.

Bajo este enfoque se tienen cuatro paradigmas los cuales son:

- paradigma por procedimientos o paradigma imperativo
- paradigma declarativo
- paradigma funcional
- paradigma orientado a objetos

▲

40

Paradigmas de programación

41

Historia de los paradigmas de Programación.

El paradigma por procedimientos, es tal vez el más conocido y utilizado en el proceso de programación, donde los programas se desarrollan a través de procedimientos. Pascal C y BASIC son tres de los lenguajes imperativos más importantes. La palabra latina imperare significa "dar instrucciones". El paradigma se inició al principio del año 1950 cuando los diseñadores reconocieron que las variables y los comandos o instrucciones de asignación constituían una simple pero útil abstracción del acceso a memoria y actualización del conjunto de instrucciones máquina. Debido a la estrecha relación con la arquitectura de la máquina, los lenguajes de programación imperativa pueden ser implementados muy eficientemente, al menos en principio.

El paradigma imperativo aún tiene cierto dominio en la actualidad. Una buena parte del software actual ha sido desarrollado y escrito en lenguajes imperativos. La gran mayoría de programadores profesionales son principalmente o exclusivamente programadores imperativos (Hay que añadir que los paradigmas de la programación concurrente y orientada al objeto son en realidad sub-paradigmas de la programación imperativa, así que sus adeptos también son programadores

41

Paradigmas de programación

42

Historia de los paradigmas de Programación.

Programación Lógica

El paradigma declarativo o paradigma de programación lógica se basa en el hecho que un programa implementa una relación antes que una correspondencia. Debido a que las relaciones son más generales que las correspondencias (identificador - dirección de memoria), la programación lógica es potencialmente de más alto nivel que la programación funcional o la imperativa.

El lenguaje más popular enmarcado dentro de este paradigma es el lenguaje PROLOG.

El auge del paradigma declarativo se debe a que el área de la lógica formal de las matemáticas ofrece un sencillo algoritmo de resolución de problemas adecuado para, usarse en un sistema de programación declarativo de propósito general.

42

Paradigmas de programación

43

Historia de los paradigmas de Programación.

Si la programación imperativa se caracteriza por el uso de variables, comandos y procedimientos, la programación funcional se caracteriza por el uso de expresiones y funciones. Un programa dentro del paradigma funcional, es una función o un grupo de funciones compuestas por funciones más simples estableciéndose que una función puede llamar a otra, o el resultado de una función puede ser usado como argumento de otra función. El lenguaje por excelencia ubicado dentro de este paradigma es el LISP. Por ejemplo si se desea obtener la nota promedio de un alumno podría construirse una función promedio la cual se obtendría a partir de otras funciones más simples: una (sumar) la cual obtiene la suma de las entradas de la lista, otra (contar) la cual cuenta el número de entradas de la lista y la tercera (dividir) que obtiene el cociente de los valores anteriores, su sintaxis será:

```
(dividir (sumar notas) (contar notas))
```

Obsérvese que la estructura anidada refleja el hecho de que la función dividir actúa sobre los resultados de suma y contar.

43

Paradigmas de programación

44

Historia de los paradigmas de Programación.

El paradigma orientado a objetos, se basa en los conceptos de objetos y clases de objetos. Un objeto es una variable equipada con un conjunto de operaciones que le pertenecen o están definidas para ellos. El paradigma orientado a objetos actualmente es el paradigma más popular y día a día los programadores, estudiantes y profesionales tratan de tomar algún curso que tenga que ver con este paradigma, podría decirse, que programar orientado a objetos está de moda.

Alrededor de 1970 David Parnas planteó el ocultamiento de la información como una solución al problema de gerenciar grandes proyectos software. Su idea fue encapsular cada variable global en un módulo con un grupo de operaciones (al igual que los procedimientos y las funciones) que permitan tener un acceso directo a la variable. Otros módulos pueden acceder a la variable sólo indirectamente, llamando a estas operaciones. Hoy se usa el término objeto para tales módulos o variables encapsuladas a sí mismas. Lenguajes imperativos como Pascal Y C han sido modificados (o añadidos) para que soporten el paradigma orientado a objetos para dar Delphi en el caso de Pascal y C++ en el caso de C.

Una de las bondades importantes de los lenguajes orientados a objetos es que las definiciones de los objetos pueden usarse una y otra vez para construir múltiples objetos con las mismas propiedades o modificarse para construir nuevos objetos con propiedades similares pero no exactamente iguales.

44

Paradigmas de programación

45

Historia de los paradigmas de Programación.

El lenguaje orientado a objetos por excelencia es Smaltalk desarrollado en Palo Alto Research Center durante los 1970's.

Pero que es exactamente un lenguaje orientado a objetos- Los siguientes conceptos señalan las características generalmente aceptadas acerca de los lenguajes orientados a objetos.

- Objetos y clases son obviamente los conceptos fundamentales. Una clase es un conjunto de objetos que comparten las mismas operaciones.
- Objetos (o al menos referencia a objetos) deben ser valores de la clase base. Así, cualquier operación puede tomar un objeto como un argumento y puede devolver un objeto como resultado. De esta manera el concepto de clase de objetos está relacionado con el concepto de tipo de dato.
- Herencia es también vista como un concepto clave dentro del mundo de los objetos. En este contexto, la herencia es la habilidad para organizar las clases de objetos en una jerarquía de subclases y superclases y las operaciones dadas para una clase se pueden aplicar a los objetos de la subclase.

45

Paradigmas de programación

46

Paradigmas de Programación y la Inteligencia Artificial.

Programación lógica.

Basado en la idea de que la deducción lógica puede constituirse en una forma de computación universal. Su lenguaje más representativo es Prolog[20], basado en el principio de resolución-SL de Kowalskiy Kuehner[49], una regla de inferencia que permite hacer demostrar que una cláusula lógica es consecuencia de un conjunto de ellas (el programa) y computar los valores de las variables en la cláusula, que hacen esto posible.

Otros lenguajes lógicos incluyen a Datalog[52], una eficiente y completa versión restringida de Prolog, sin funciones ni predicados extralógicos; Mercury[86] con tipos de datos explícitos; y el orientado a objetos Logtalk[60].

Programación funcional.

Basado en la idea de la computación en términos de funciones matemáticas y su transformación a través de formas funcionales (funciones de orden superior). Su concepción original se debe a Backus [2], aunque tiene raíces en el Cálculo- λ de Church[15]. El primer lenguaje con características funcionales fue Lisp de McCarthy[54] que es de nuestro interés porque además es una implementación de los sistemas simbólicos físicos. Otros lenguajes más próximos a la idea original de Backus incluyen ML[33], Ocaml[74] y Haskell[89].

46

Paradigmas de programación

47

Historia de los paradigmas de Programación.

CONCLUSIONES

La comprensión básica de los conceptos de los lenguajes de programación y los diferentes paradigmas son necesarios para todos los ingenieros de software, no tanto para los especialistas en un lenguaje de programación. Esto se debe a que los lenguajes de programación son una herramienta fundamental.

Los lenguajes de programación influyen notablemente la manera en que pensamos acerca del diseño y construcción del software y los algoritmos y estructuras de datos que utilizemos para desarrollar software.

47

Paradigmas de programación

48

Adecuación a paradigmas de Programación.

Resulta importante que los programadores entiendan la situación cambiante y de constante expansión de las ciencias de la computación y la información.

El procesamiento de datos de manera automática y la adecuación a las técnicas de los modelos actuales. Da soporte para los modelos futuros, que se definirán en relación con los presentes , como ajuste o extensiones de funcionalidad más allá de los límites actuales.

Los lenguajes de programación influyen notablemente la manera en que pensamos acerca del diseño y construcción del software y los algoritmos y estructuras de datos que utilizemos para desarrollar software.

Los paradigmas de programación condicionan no solo los lenguajes, sino todo el proceso de desarrollo de software y las metodologías usadas para ello.

48

Cronología

49

Secuencia cronológica con algunos de los paradigmas más importantes y sus años de aparición, junto con algunas de sus limitaciones y soluciones:

Paradigma imperativo (1950s): Es el paradigma más antiguo y básico, que consiste en dar instrucciones explícitas a la máquina para cambiar su estado. Algunos ejemplos de lenguajes imperativos son Fortran y Algol. Algunas limitaciones de este paradigma son: la dificultad para manejar la complejidad de los programas, la falta de modularidad y reusabilidad del código, y la dependencia del hardware.

Paradigma estructurado (1960s): Es una extensión del paradigma imperativo, que introduce el uso de estructuras de control como if, while, for, etc. para organizar el flujo del programa. Algunos ejemplos de lenguajes estructurados son C y Pascal. Algunas limitaciones de este paradigma son: la dificultad para expresar conceptos abstractos, la falta de encapsulamiento y ocultación de la información, y la posibilidad de generar errores por el uso de variables globales y saltos incondicionales (goto).

49

Cronología

50

Paradigma funcional (1960s): Es un paradigma inspirado en la matemática, que consiste en definir el resultado deseado como el valor de una serie de aplicaciones de funciones. Algunos ejemplos de lenguajes funcionales son LISP y Haskell. Algunas limitaciones de este paradigma son: la dificultad para representar el estado mutable, la falta de eficiencia en el uso de la memoria y el procesamiento, y la incompatibilidad con algunos paradigmas imperativos.

Paradigma orientado a objetos (1970s): Es un paradigma que se basa en el concepto de objeto, que es una entidad que contiene datos (atributos) y comportamientos (métodos). Algunos ejemplos de lenguajes orientados a objetos son Smalltalk y C++. Algunas limitaciones de este paradigma son: la dificultad para modelar problemas que no se ajustan al concepto de objeto, la falta de expresividad y flexibilidad en el diseño del código, y la posibilidad de generar acoplamiento e inconsistencia entre los objetos.

50

Cronología

51

Paradigma lógico (1970s): Es un paradigma que se basa en el concepto de lógica, que consiste en declarar el resultado deseado como la respuesta a una pregunta sobre un sistema de hechos y reglas. Algunos ejemplos de lenguajes lógicos son Prolog y Datalog. Algunas limitaciones de este paradigma son: la dificultad para manejar la incertidumbre, la falta de control sobre el orden de ejecución, y la ineficiencia en el uso de la memoria y el procesamiento.

Paradigma concurrente (1980s): Es un paradigma que se basa en el concepto de concurrencia, que consiste en ejecutar varias tareas de forma simultánea o paralela. Algunos ejemplos de lenguajes concurrentes son Erlang y Java. Algunas limitaciones de este paradigma son: la dificultad para sincronizar y comunicar las tareas, la posibilidad de generar condiciones de carrera y bloqueos, y la complejidad para depurar y probar el código.

51

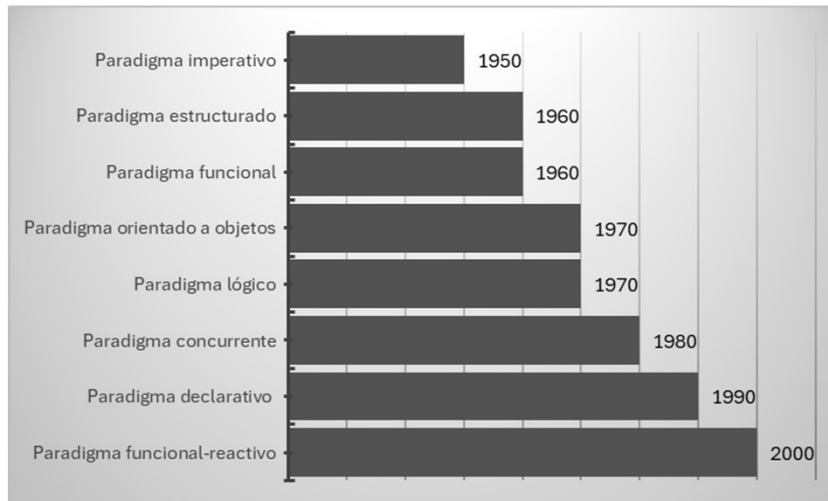
Cronología

52

Paradigma declarativo (1990s): Es un paradigma que se basa en el concepto de declaratividad, que consiste en especificar el qué y no el cómo del problema. Algunos ejemplos de lenguajes declarativos son SQL y HTML. Algunas limitaciones de este paradigma son: la dificultad para expresar algoritmos complejos, la falta de control sobre el rendimiento y la optimización, y la dependencia de la implementación del lenguaje.

Paradigma funcional-reactivo (2000s): Es un paradigma que combina los conceptos de programación funcional y programación reactiva, que consiste en declarar el resultado deseado con flujos de datos y la propagación del cambio. Algunos ejemplos de lenguajes funcionales-reactivos son Elm y RxJava. Algunas limitaciones de este paradigma son: la dificultad para manejar los efectos secundarios, la falta de soporte y documentación, y la curva de aprendizaje.

52



53

Hasta aquí....

54

- Hemos desarrollado algoritmos utilizando instrucciones simples y operadores aritméticos.
- Hemos visto el concepto de expresión condicional y los operadores que intervienen en las mismas.
- Hemos utilizado estructuras iterativas que nos han ayudado en los problemas repetitivos.
- Hemos realizado el seguimiento de los algoritmos y escrito sus pruebas de escritorio.

54

Pero....

55

- Aunque los algoritmos puedan ser verificados no hemos obtenido programas ejecutables.
- Necesitamos una herramienta que nos permita obtener ejecutables a partir de un lenguaje de programación .
- Los diagramas de flujo no tienen un lenguaje formal asociado.
- Adoptaremos C# como lenguaje de programación.

<https://visualstudio.microsoft.com/es/downloads/>

55

Programación en C#

56

Introducción :

El entorno de desarrollo de C# de Microsoft Visual Studio nos permite desarrollar aplicaciones de Consola (CLI) y también las utilizan interfaces gráficas (GUI).

Debido a la sencilla estructura que tienen las aplicaciones de consola son las primeras que comenzaremos a desarrollar.

En una aplicación de consola el usuario interactúa con la aplicación utilizando el teclado y la ventana de consola.

<https://visualstudio.microsoft.com/es/downloads/>

<https://visualstudio.microsoft.com/es/vs/older-downloads/>

```

file:///C:/TSP/Programacion1/Ejemplos/Clase4_ej1/Clase4_ej1/bin/Debug/Clase4_ej1.EXE
Ingrese primer sumando : 23
Ingrese otro sumando : 77
La suma de los valores ingresados es : 100

```

56

<https://visualstudio.microsoft.com/es/downloads/>

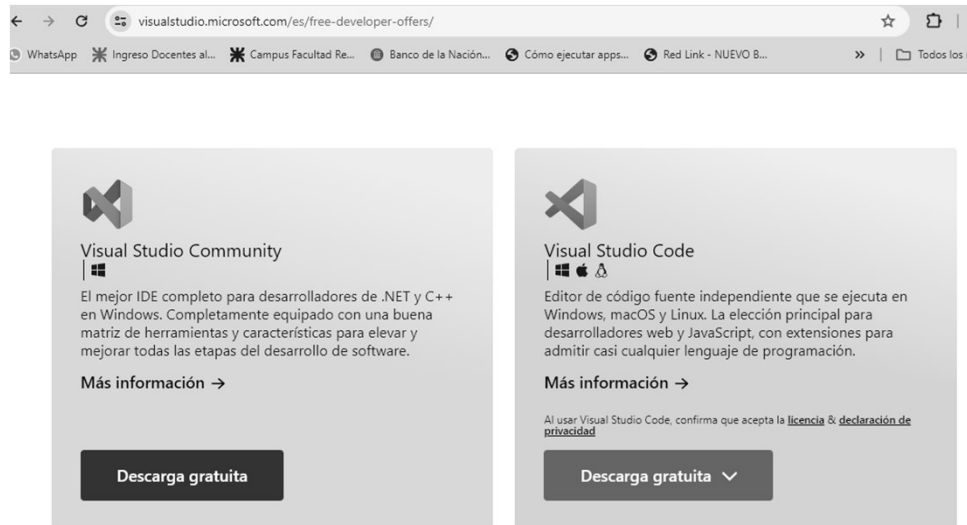


04/09/17

<https://visualstudio.microsoft.com/es/vs/older-downloads/>

57

<https://visualstudio.microsoft.com/es/free-developer-offers/>



58

POO - Algunas ideas....

59

- C# es un lenguaje Orientado a Objetos
- Un objeto representa una entidad con un rol bien definido en el dominio de un problema.
- Se caracterizan a partir de su **estado, comportamiento e identidad**.
- Los objetos pertenecen a alguna clase que los representa.
- Las clases definen las características de los objetos de un tipo determinado.

59

Introducción a Visual Studio

60

Pantalla
Inicial



Tareas iniciales



Clonar un repositorio

Obtiene código desde un repositorio en línea, como GitHub o Azure DevOps.



Abrir un proyecto o una solución

Abre un archivo .sln o proyecto de Visual Studio local.



Abrir una carpeta local

Navegar y editar el código en cualquier carpeta



Crear un proyecto

Elija una plantilla de proyecto con la técnica scaffolding de código para comenzar.

Continuar sin código →

60

Introducción a Visual Studio

61



61

Introducción a Visual Studio Express

62



62

Introducción a Visual Studio

63

Aplicación de
Consola

Configuración
del proyecto

Configure su nuevo proyecto

Aplicación de consola ☒ C# ☐ Linux ☐ macOS ☐ Windows ☐ Consola

Nombre del proyecto
ConsoleApp1

Ubicación
D:\Facultad_UTN\

Nombre de la solución ConsoleApp1

☒ Colocar la solución y el proyecto en el mismo directorio

63

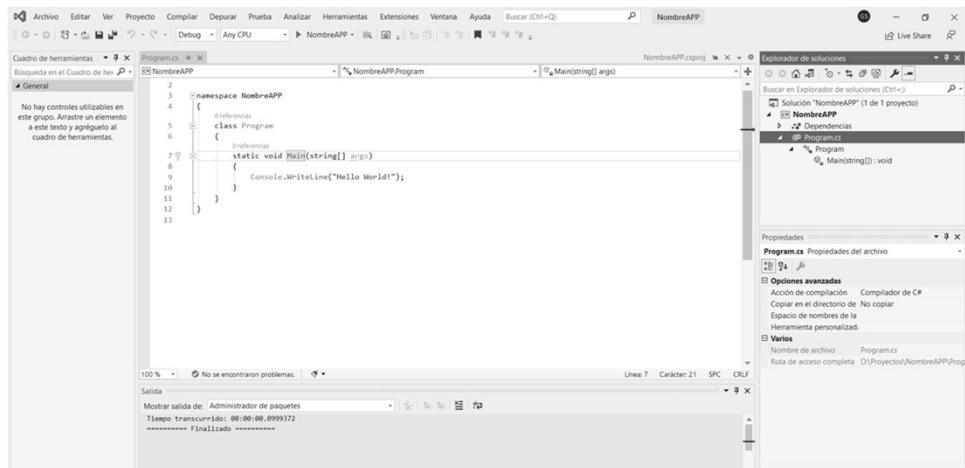
Introducción a Visual Studio

64

Aplicación
de Consola

Pantalla
Program.CS

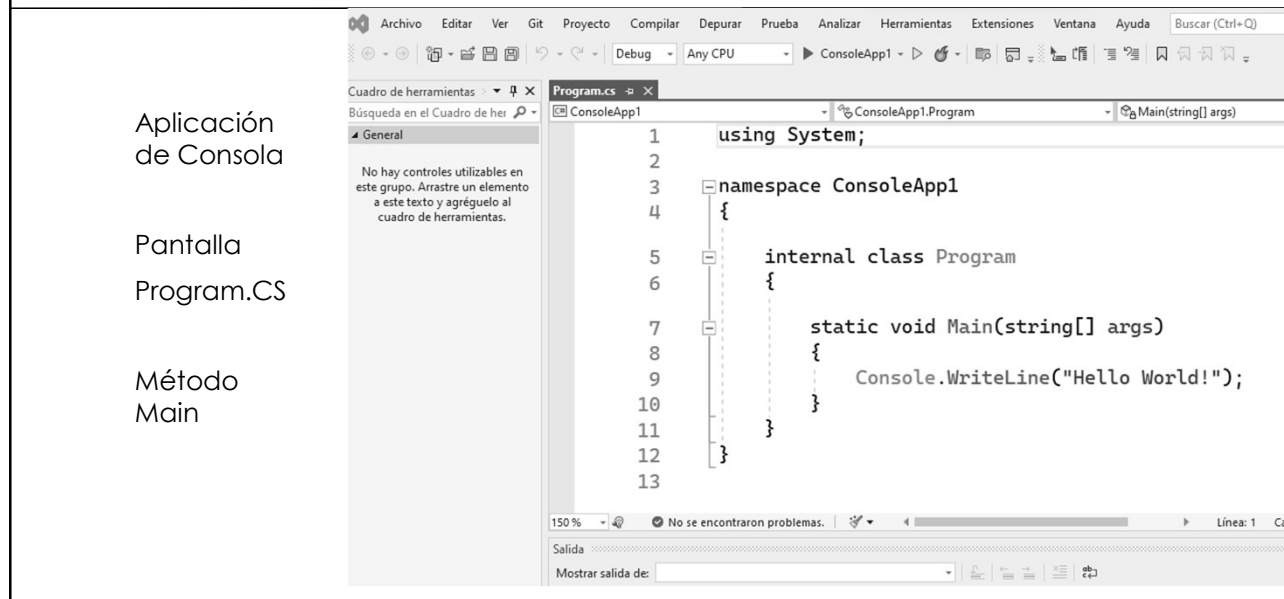
Método
Main



64

Introducción a Visual Studio

65



65

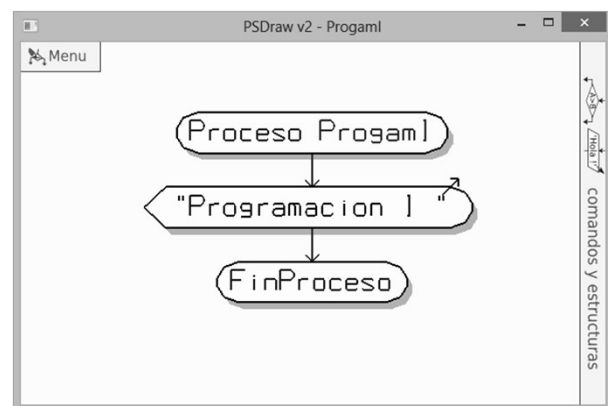
Una aplicación de consola

66

Con el objetivo de analizar la estructura de una aplicación consideremos la implementación de un algoritmo simple, un algoritmo que muestre u mensaje por pantalla.

Su diagrama de flujo sería:

Proceso Program1
Escribir "Programacion I";
FinProceso



66

Una aplicación de consola

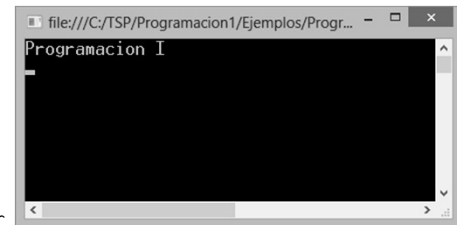
67

La implementación del algoritmo correspondería con la siguiente:

```
using System;
namespace ejemplo1{
    class Program{
        // Comienzo de la aplicación
        static void Main(string[] args){

            /* Conjunto de órdenes a ejecutar
             * por la aplicación.
             */
            Console.WriteLine("Programación I");

        }
    }
}
```



67

Una aplicación de consola

68

Comentarios: permiten documentar la aplicación y/o sus partes. Son desestimados por el compilador.

```
using System;

namespace ejemplo1{
    class Program{

        // Comienzo de la aplicación
        static void Main(string[] args){

            /* Conjunto de órdenes a ejecutar
             * por la aplicación.
             */
            Console.WriteLine("Programación I");

        }
    }
}
```

68

Comentarios

69

Existen dos tipos de comentario en C#:

Comentarios de una línea:

Para ellos se utiliza //. Todo el texto a la derecha de la doble barra es un comentario. Por ejemplo:

```
// Comentario de una línea
```

Comentarios delimitados:

Son comentarios donde se marca su inicio utilizando los caracteres /* y su finalización con */. Todo el texto encerrado entre ellos será considerado como comentario. Por ejemplo:

```
/* Comentario  
   delimitado */
```

Es importante utilizar comentarios para documentar la aplicación. Documente considerando que Usted mismo será quien utilice la documentación.

69

Una aplicación de consola

70

Directivas **using**: indica al compilador bibliotecas de clases que se utilizaran en la aplicación.

```
using System;  
namespace ejemplo1{  
    class Program{  
        // Comienzo de la aplicación  
        static void Main(string[] args){  
            /* Conjunto de órdenes a ejecutar  
             * por la aplicación.  
             */  
            Console.WriteLine("Programación I");  
        }  
    }  
}
```

70

Directivas using

71

En C# existe un conjunto de clases predefinidas que permiten desarrollar aplicaciones rápidamente a partir de su uso. Según las clases utilizadas en nuestras aplicaciones serán las directivas *using* a utilizar.

El conjunto de clases del que se dispone se conoce como Biblioteca de Clases de .NET Framework.

En nuestro ejemplo, debido a que utilizamos la clases **Console**, y a que la misma se encuentra definida en el espacio de nombre **System**, debemos utilizar una directiva **using** de la siguiente manera:

```
using System;
```

Las directivas **using** siempre se colocan al comienzo del archivo del cual forman parte.

71

C# aplicación de consola

72

Palabra reservada **namespace**: define un espacio de nombre.

```
using System;
```

```
namespace ejemplo1{
    class Program{

        // Comienzo de la aplicación
        static void Main(string[] args){

            /* Conjunto de órdenes a ejecutar
             * por la aplicación.
             */
            Console.WriteLine("Programación I");
        }
    }
}
```

72

Palabra reservada **namespace**

73

Los espacios de nombre permiten agrupar contenido de diferente tipo. Cuando utilizamos la directiva `using`, estábamos haciendo referencia a un espacio de nombre. Puntualmente, en el ejemplo, estábamos haciendo referencia al espacio de nombre **System** donde se ha definido la clases **Console**.

Cuando desarrollamos una aplicación, la misma define su propio espacio de nombre, en nuestro ejemplo, **ejemplo1**. La clase Program, está incluida en el espacio de nombre **ejemplo1**.

Si agregamos nuevas clases o componentes a nuestra aplicación, los mismos automáticamente, contendrán una definición de espacio de nombre que se corresponda con el espacio de nombre de la aplicación.

Al reutilizar una clase existente, desarrollada en otra aplicación se debe tener en cuenta la diferencia en los espacios de nombre.

73

Una aplicación de consola – C#

74

Palabra reservada **class**: indica que comienza la definición de una clase.

```
using System;
```

```
namespace ejemplo1{
```

```
    class Program{
```

```
        // Comienzo de la aplicación
```

```
        static void Main(string[] args){
```

```
            /* Conjunto de órdenes a ejecutar por la
               aplicación. */
```

```
            Console.WriteLine("Programación I");
```

```
        }
```

```
    }
```

```
}
```

74

Palabra reservada **class**

75

Las aplicaciones en C# se implementan a partir de una clase. Esta clase debe contener un método **estático** llamado **main** que contiene el conjunto de órdenes a ejecutar cuando la aplicación se inicia.

En caso de ser necesario, es posible cambiar la clase predeterminada que dará inicio a la aplicación. Esto se puede realizar desde el menú **Proyecto / Propiedades / Objeto de inicio**.

El nombre asignado a las clases debe ser un identificador válido, por lo que:

- Debe comenzar con una letra o el símbolo `_`
- No puede contener espacios en blanco ni caracteres especiales como: `" = * + ~ ; , . : # % & / () ? ¿ ! - ^ [] { } ' < >`
- No puede ser una palabra reservada.

Se suele utilizar una convención para nombrar las clases. Esta consiste en utilizar una letra en mayúscula para la primera letra del nombre y para cada letra que de comienzo a una nueva palabra que constituya el nombre. Por ejemplo: **Empleado EmpleadoTiempoCompleto EmpleadoMedioTiempo**

75

Una aplicación de consola

76

Método main: cumple con una declaración preestablecida. Es el método que define que órdenes se ejecutarán al comienzo de la aplicación.

```
using System;

namespace ejemplo1{
    class Program{

        // Comienzo de la aplicación
        static void Main(string[] args){

            /* Conjunto de órdenes a ejecutar por la aplicación.
            */
            Console.WriteLine("Programación I");

        }
    }
}
```

76

Método main

77

Todas las aplicaciones deben contener un método **main** en la clase que da origen al objeto de inicio. Este método tiene algunas características particulares:

Cláusula **static**: los métodos definidos con este atributo corresponden con métodos que no requieren de un objeto para ser invocados.

Tipo de retorno **void**: los métodos tienen un tipo y valor de retorno. En el caso particular de main, se indica como tipo de retorno void con el fin de indicar que este método no tendrá ningún valor de retorno.

Argumento de la función: Permite acceder a los argumentos con los cuales es llamada la aplicación. Todos ellos son cadenas de caracteres.

El código a ejecutar por el método corresponde con el fragmento delimitado por las llaves ({ }) que se encuentran a continuación.

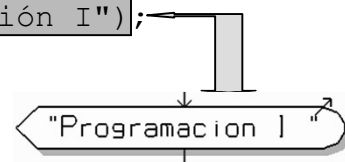
77

Una aplicación de consola

78

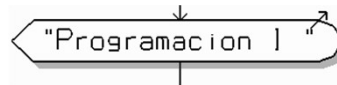
Operación de escritura: La clase **Console** implementa métodos que permiten mostrar valores en la pantalla y obtener valores ingresados por el usuario.

```
using System;
namespace ejemplo1{
    class Program{
        // Comienzo de la aplicación
        static void Main(string[] args){
            /* Conjunto de órdenes a ejecutar
            * por la aplicación.
            */
            Console.WriteLine("Programación I");
        }
    }
}
```



78

Clase **console**



79

Las clases definen las características de los objetos, y son estos objetos los que utilizamos. Sin embargo, existen clases que por sus características se utilizan directamente sin crear un objeto de su tipo. Uno de estos casos es la clase **Console**.

Entre los métodos que implementa esta clase se encuentran:

Write y **WriteLine**: permiten mostrar texto en la pantalla. Toman como argumento una cadena con el texto y el formato a utilizar. Por ejemplo:

```
Console.Write("{0} {1}", "Hola", "Mundo");
```

Produciría una salida cómo: **Hola Mundo**

La diferencia entre estas funciones es que **WriteLine** agrega un retorno de carro (o salto de línea) luego de mostrar la cadena de caracteres mientras que **Write** no. Permitiendo de esta manera concatenar diferentes salidas.

79

Secuencias de escape

80

Las secuencias de escape son caracteres que al agregarse a la cadena de formato de utilizadas permiten lograr diferentes efectos.

A continuación se indican algunas de ellas:

Secuencia	Descripción
<code>\n</code>	Nueva línea
<code>\t</code>	Tabulación horizontal
<code>\r</code>	Retorno de línea
<code>\\</code>	Barra diagonal
<code>\"</code>	Comillas

Por ejemplo:

```
console.WriteLine("Hola \n \"Mundo\"");
```

Produciría una salida como:

Hola
"Mundo"

80

Clase **console**

81

Todos los valores que ingresemos en una aplicación de consola, en C#, utilizando el método `ReadLine` nos retornará un valor de tipo texto.

Si quisiéramos almacenar un valor ingresado por el usuario en una variable de tipo entero, deberemos convertir el valor ingresado a entero.

Por ejemplo:

```
static void Main(string[] args)
{
    int num;
    Console.WriteLine("Ingrese un número:");
    num = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Usted ingresó el valor: {0}",
num);
}
```

Definimos una variable de tipo entero.

Cargamos en la variable un valor ingresado por el usuario convirtiéndolo a entero.

Mostramos el valor ingresado por el usuario en pantalla.

81

Clase **console**

82

Otro de los métodos relevantes de esta clase es el método `ReadLine`. Este método permite cargar un valor ingresado por el usuario. Para esto es necesario definir previamente un espacio donde almacenarlo.

Las palabras reservadas ***int*** y ***string*** definen respectivamente variables capaces de almacenar valores de tipo numérico entero y de tipo cadena.

En el siguiente ejemplo observamos:

```
static void Main(string[] args)
{
    string valor;
    Console.WriteLine("Ingrese un texto:");
    valor = Console.ReadLine();
    Console.WriteLine("Usted ingresó el texto: {0}", valor);
}
```

La definición de una variable de tipo cadena.

Cargamos en la variable un valor ingresado por el usuario.

Mostramos el valor ingresado por el usuario en pantalla.

82

Tipos Enteros

83

La siguiente tabla muestra los tamaños e intervalos de los tipos enteros, que constituyen un subconjunto de los tipos simples.

Si el valor representado por un literal entero supera el intervalo de valores del tipo `ulong`, se producirá un error de compilación.

Tipo	Intervalo	Size
sbyte	De -128 a 127	Entero de 8 bits con signo
byte	De 0 a 255	Entero de 8 bits sin signo
char	U+0000 a U+ffff	Carácter Unicode de 16 bits
short	De -32.768 a 32.767	Entero de 16 bits con signo
ushort	De 0 a 65.535	Entero de 16 bits sin signo
int	De -2.147.483.648 a 2.147.483.647	Entero de 32 bits con signo
uint	De 0 a 4.294.967.295	Entero de 32 bits sin signo
Long	De -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	Entero de 64 bits con signo
ulong	De 0 a 18,446,744,073,709,551,615	Entero de 64 bits sin signo

83

Operadores Aritméticos

84

Ahora que hemos definido variables de tipo entero, podemos utilizarlas con operadores Aritméticos, los que nos permitirán realizar operaciones.

Los operadores aritméticos son:

Operador	Significado	Orden de evaluación
*	Multiplicación	De existir varios operadores en la misma expresión se evalúan de izquierda a derecha
/	División	
%	Resto de la división	
+	Suma	Idem anterior
-	Resta	

Las expresiones en lenguaje C# deben leerse de derecha a izquierda. Por ejemplo:

```
Valor = 3 * numero;      /* Al ser ejecutada, realiza primero la
                           multiplicación y luego la asignación.*/
```

84

Los tipos reales

85

Todas las aplicaciones que hemos presentado y desarrollado han sido convenientemente seleccionadas para utilizar variables de tipo entero. Esto está muy lejos de constituir una realidad general.

Si debiéramos incorporar un atributo a nuestras clases o variable a nuestros programas que utilice valores de tipo real requeriremos utilizar alguno de los siguientes tipos:

float: permite almacenar valores de tipo real con precisión simple.
32 bits - Rango -3.4×10^{38} to $+3.4 \times 10^{38}$ - Precisión : 7 dígitos

double: permite almacenar valores de tipo real con el doble de precisión que una variable float.

64 bits - Rango: 5.0×10^{-324} a $\pm 1.7 \times 10^{308}$ - Precisión : 15-16 dígitos

decimal: proporcionan mayor precisión que las variables de tipo double e intervalo mas reducido.

128 bits (de -7.9×10^{28} a 7.9×10^{28}) / (10^0 a 28) - Precisión : 28-29 dígitos

Basadas en
aproximación

85

Los tipos reales

86

Por defecto, en C# los valores reales son tratados como variables de tipo double y esto hace que si en una expresión utilizamos un valor constante, el mismo sea tratado como double por defecto.

Es por esto que cuando deseamos que los valores reales sean entendidos de otra manera debemos indicarlo. Por ejemplo:

```
static void Main(string[] args)
{
    float v;
    v = 15.3F; //F indica Float
    Console.WriteLine("El valor de v es{0}", v);
    Console.ReadKey();
}
```

En este caso, la letra F a continuación de la constante 15.3 indica que el valor debe ser tratado como de tipo **float**. Algo similar ocurre con el tipo **decimal** si deseamos indicar que un valor debe ser tratado de esta forma, debemos indicarlo a través de agregar la letra **M** al final de la constante.

86

Los tipos reales

87

Otra característica que surge a partir del uso de valores reales, es la forma en la que los mismos serán mostrados en la pantalla. En general suele ser necesario indicar la cantidad de dígitos a mostrar o el símbolo peso si se trata de un valor monetario.

Esto se puede resolver sencillamente indicando el formato deseado al mostrar el valor en la pantalla. Para esto se indica con una letra el formato deseado.
Por ejemplo:

```
static void Main(string[] args)
{
    float v;
    v = 15.3234324F;
    Console.WriteLine("El valor de v es{0:C}", v);
    Console.ReadKey();
}
```

La letra C indica que el valor debe ser tratado como un valor monetario.

87

Los tipos reales

88

Además del caso anterior, existen otros modificadores de formato que pueden ser utilizados:

Especificador	Descripción
C o c	Indica que se trata de un valor monetario y agrega el símbolo monetario apropiado con dos decimales.
N o n	Indica dos decimales luego de la coma.
E o e	Notación científica
FN o fn	Indica una cantidad de decimales establecida por el valor de n. Por ejemplo {0:F4}
,z:Fn	Indica la cantidad de posiciones en que se alinea la coma y decimales después. p.ej. {0,30:F5} indica 30 espacios antes de la coma y 5 decimales después de la coma.

88

Operadores Aritméticos

89

• Las operaciones aritméticas en las aplicaciones se realizan utilizando los “Operadores Aritméticos”. Estos indican las posibles operaciones a realizar entre variables de tipo numérico.

• No todos los mismos operadores tienen el mismo orden de precedencia, es por esto que se debe prestar atención a la forma en la que se escriben las operaciones aritméticas para asegurarnos que sean ejecutadas según nuestras intenciones.

Operador	Significado
*	Multiplicación
/	División
%	Resto de la división
+	Suma
-	Resta

• Es posible alterar el orden de evaluación de los operadores en una expresión aritmética utilizando paréntesis.

89

La clase convert

90

• Las operaciones de lectura realizadas con `Console.ReadLine()` devuelven una cadena de caracteres (string) que no puede ser asignada a una variable numérica.

• Para esto utilizamos la clase `Convert` con el método adecuado a la variable que se pretenda asignar (recibe un string y devuelve un valor numérico) según la tabla:

Métodos	Tipo de retorno
<code>Convert.ToByte ()</code>	sbyte o byte
<code>Convert.ToInt16 ()</code>	short y ushort
<code>Convert.ToInt32 ()</code>	int y uint
<code>Convert.ToInt64 ()</code>	long y ulong
<code>Convert.ToSingle()</code>	float
<code>Convert.ToDouble()</code>	double
<code>Convert.ToDecimal()</code>	decimal

Por ejemplo: `Numero = Convert.ToInt32(Console.ReadLine());`

90

Operadores Lógicos

91

Una condición es construida a partir de valores constantes, variables y operadores. Obteniéndose de esta forma una expresión condicional. Los operadores que se pueden utilizar en una expresión condicional son:

Estos operadores lógicos permiten definir expresiones que se utilizan como condiciones lógicas en varias estructuras.

	Operador	Ejemplo	Significado
Operadores de igualdad	==	$x == y$	x es igual a y
	!=	$x != y$	x es diferente de y
Operadores relacionales	>	$x > y$	x es mayor que y
	<	$x < y$	x es menor que y
	>=	$x >= y$	x es mayor o igual que y
	<=	$x <= y$	x es menor o igual que y

91

Una aplicación de ejemplo

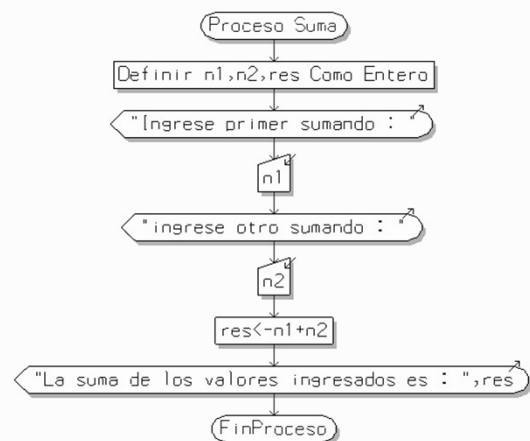
92

Consideremos la implementación de una aplicación a modo de ejemplo. "Implemente un programa que permita sumar dos valores de tipo entero ingresados por el usuario".

En PSeInt

```

Proceso Suma
Definir n1,n2,res Como Entero;
Escribir 'Ingrese primer sumando : ';
Leer n1;
Escribir 'ingrese otro sumando : ';
Leer n2;
res<-n1+n2;
Escribir 'La suma de los valores ingresados es : ',res;
FinProceso
  
```

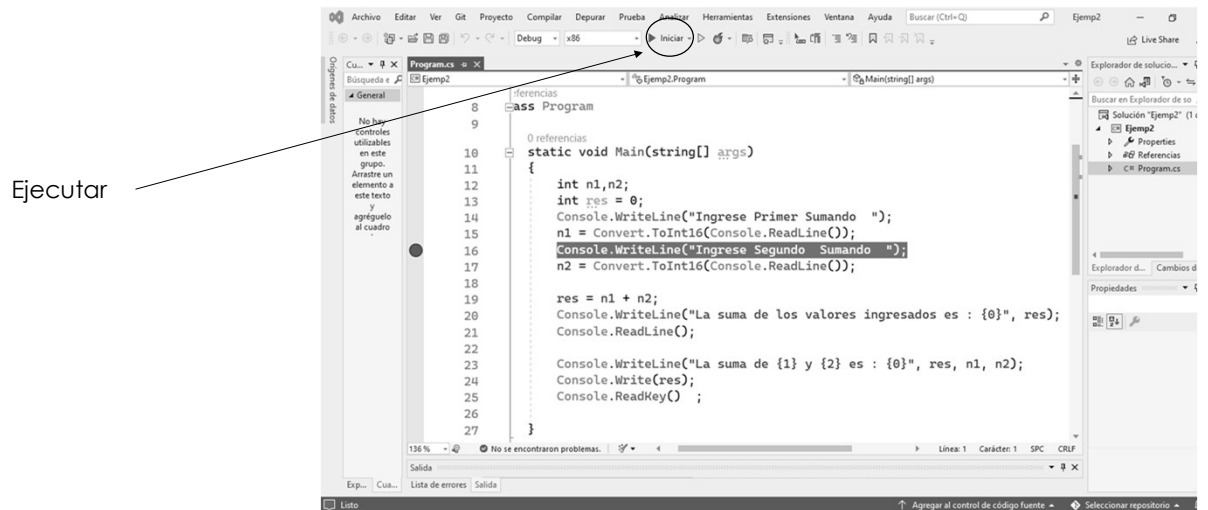


92

Una aplicación de ejemplo en C#

93

Consideremos la implementación de una aplicación a modo de ejemplo. "Implemente un programa que permita sumar dos valores de tipo entero ingresados por el usuario".



93

Estructura SI

94

Las expresiones condicionales son evaluadas a través de estructuras.

La estructura condicional SI tiene la siguiente sintaxis:

Palabra reservada que denota el uso de la estructura.

if

(condición)

Expresión lógica a evaluar.

```

acción 1;
acción 2;
acción 3;
...
acción n;

```

Grupo de órdenes a ejecutar en caso de que la condición sea evaluada como verdadera.

}

- La condición a evaluar se debe encerrar entre paréntesis.
- if (a == 0)
- Las acciones a ejecutar se delimitan con las llaves.

94

Estructura SI

95

También es posible indicar acciones cuando la condición o expresión lógica, no se cumple. Para ello se usa la instrucción **else**

```
if ( condición )
{
    acción 1;
    acción 2;
    acción 3;
    ...
    acción n;
}
else
{
    acción f1;
    acción f2;
}
```

Expresión lógica a evaluar.

Grupo de órdenes a ejecutar en caso de que la condición sea evaluada como verdadera.

Grupo de órdenes a ejecutar en caso de que la condición sea evaluada como FALSA.

95

Estructura SI

96

Ejemplo

```
namespace NombreAPP
{
    0 referencias
    class Program
    {
        0 referencias
        static void Main(string[] args)
        {
            float dividendo, divisor, cociente;

            Console.Write("Ingrese dividendo: ");
            dividendo = Convert.ToSingle(Console.ReadLine());
            Console.Write("Ingrese divisor: ");
            divisor = Convert.ToSingle(Console.ReadLine());

            if (divisor != 0)
            {
                cociente = dividendo / divisor;
                Console.WriteLine("El resultado de la division es : {0,8:f3} ", cociente);
            }
            else
            {
                Console.WriteLine("Division entre cero no permitida ");
            }
        }
    }
}
```

96

Estructuras iterativas en C#

Así como en diagramas de flujo estudiamos las estructuras Mientras , Para y Repetir es el momento de analizar su implementación en C#. Veremos como se aplican estas estructuras en el lenguaje y las características de uso de las mismas.

• Analizaremos en la clase de hoy las instrucciones

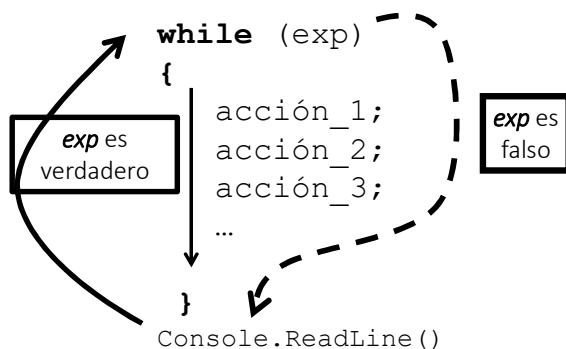
- While
- For
- Do {} While

97

97

MIENTRAS – While en C#

Como ya hemos dicho esta estructura repite un fragmento de código mientras se cumple una expresión lógica (sea verdadera). Cuando la condición es falsa la ejecución del programa continúa con la primer expresión que se encuentre después de las llaves que delimitan el bloque.



• Si la condición es falsa la primera vez que se analiza, el fragmento de código asociado a la estructura no se ejecuta nunca.

• Si las componentes de la expresión lógica no se alteran dentro del bloque, el fragmento de acciones asociado a la misma se ejecutará infinitamente, sin poder salir del bloque.

• Recomendada para cuando NO SE CONOCE la cantidad de iteraciones requeridas.

98

98

Repaso Mientras - Aplicaciones

Iteración controlada por un **centinela**:

```
static void Main(string[] args)
{
    int dato;
    int c = 0;
    Console.WriteLine("Ingrese -1 para terminar");
    Console.Write("Nuevo valor?");
    dato = Convert.ToInt32(Console.ReadLine());
    while (dato != -1)
    {
        c = c + 1;
        Console.Write("Nuevo valor?");
        dato = Convert.ToInt32(Console.ReadLine());
    }
    Console.WriteLine("Se ingresador {0} valores", c);
    Console.ReadKey();
}
```



99

Ejemplos - A

Se requiere una aplicación que permita determinar la nota promedio obtenida por un grupo de alumnos en un examen. La cantidad de alumnos es desconocida y es por esto que la aplicación deberá considerar que se termina la carga de los valores a ingresar el valor -1.

Considere que los valores de las calificaciones se encuentran en el rango de 0 a 10 y el promedio debe ser mostrado como un valor real con dos decimales.

Además, se quiere saber al finalizar la aplicación, si alguno de los alumnos ha obtenido la calificación 10.

No interesa saber cuantos alumnos obtuvieron un 10, solo si alguno.

100

Repaso Estructura PARA

Así como encontramos situaciones en las que no es conocida la cantidad de iteraciones, también se dan otras en las que SI SE CONOCE LA CANTIDAD de iteraciones de antemano:

“Se desea calcular el promedio de 4 valores ingresados por el usuario”

“Se necesita conocer cuantos valores se encuentran en el intervalo A y B siendo divisibles por 3”

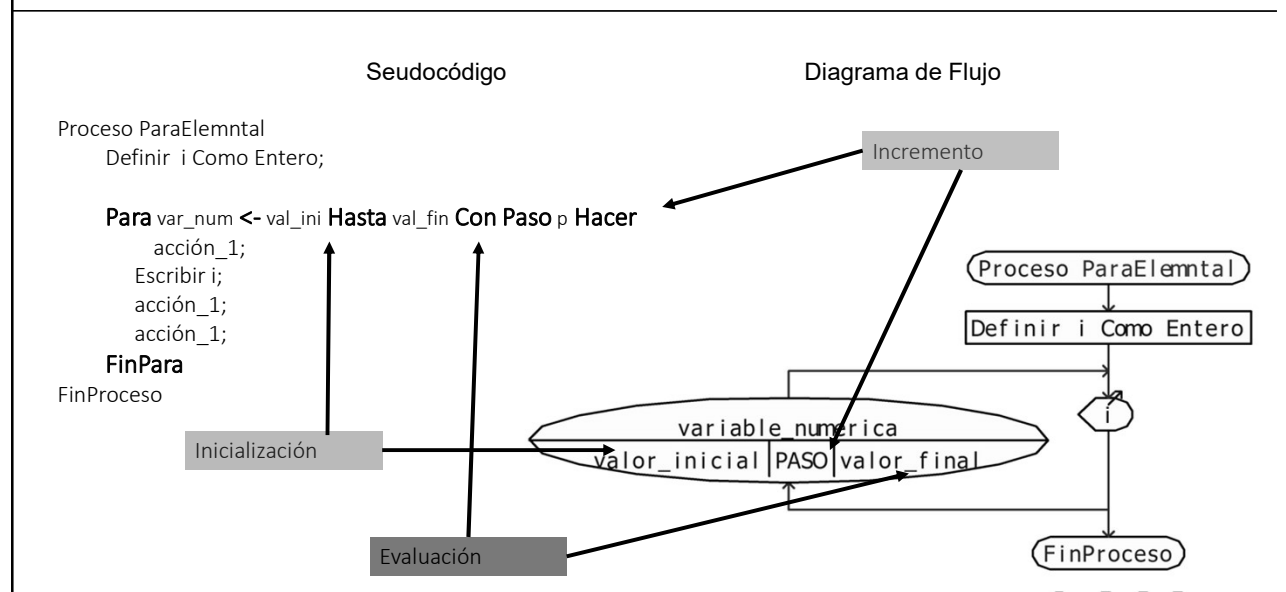
Para casos como estos la estructura **PARA** nos da la posibilidad de controlar exactamente la cantidad de iteraciones usando una variable de control de modo de producir:

- Inicialización
- Evaluación de una condición
- Un incremento.

101

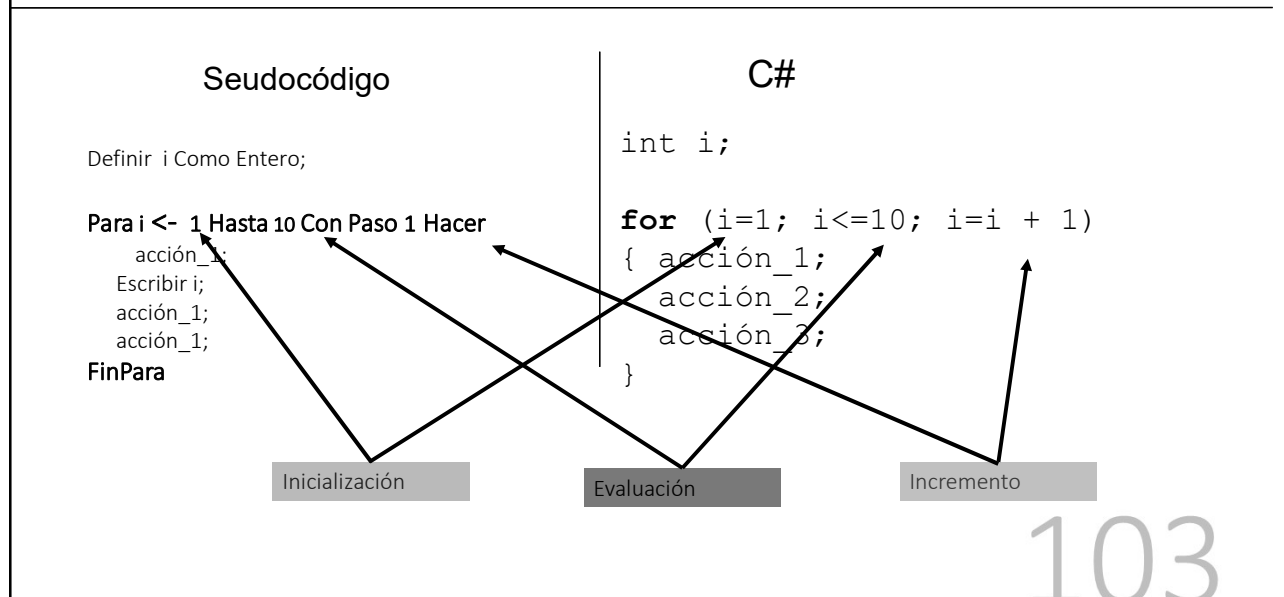
101

Repaso PARA



102

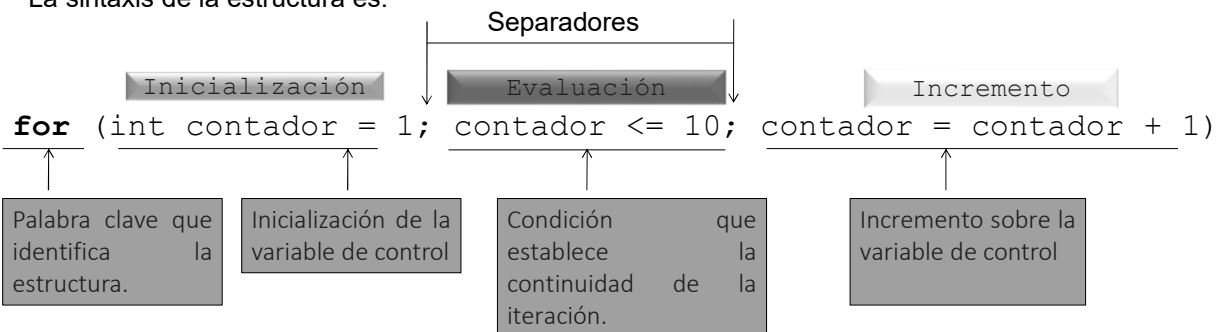
Representación PARA



103

Estructura PARA Sintaxis C# - for

La sintaxis de la estructura es:



- Plantea la idea de iteración a partir de indicar un valor inicial, una condición y un incremento.
- La variable de control puede estar definida previamente, junto con el resto de las variables (recomendable).
- Como en todas las estructuras, el bloque de órdenes asociado a la estructura debe estar encerrado entre llaves (a excepción que se trate de una sola orden).

104

Estructura PARA Ejemplo de uso

Ejemplo 1: Se tienen 5 alumnos que han rendido un examen parcial. Se requiere una aplicación que determine el valor de la calificación promedio obtenida por los alumnos.

```
static void Main(string[] args)
{
    int cont;
    double calificacion;
    double acum = 0;
    double prom;
    for (cont = 0; cont < 5; cont = cont + 1)
    {
        Console.Write("Indique la calificación {0}:", cont + 1);
        calificacion = Convert.ToDouble(Console.ReadLine());
        acum = acum + calificacion;
    }
    prom = acum / 5.0;
    Console.WriteLine("El promedio es : {0:F2}", prom);
    Console.ReadKey();
}
```

105
ej6

105

Ejemplo de uso - PARA

Ejemplo 2: Se necesita conocer cuantos valores son divisibles por 3 en un intervalo dado por un usuario contando los valores con pasos de 2. (de dos en dos)

```
static void Main(string[] args)
{
    int desde, hasta; /* Variables utilizadas para almacenar los extremos del
                       intervalo de interés */
    int i; /* Variable de control para la estructura PARA */
    Console.WriteLine("Ingrese los extremos del intervalo:");
    desde = Convert.ToInt32(Console.ReadLine());
    hasta = Convert.ToInt32(Console.ReadLine());
    for (i = desde; i <= hasta; i = i + 2)
    {
        if ((i % 3) == 0)
        {
            Console.WriteLine("{0}", i);
        }
    }
    Console.ReadKey();
}
```

ej7

ej7b

106

106

Operadores compuestos

Con el objetivo de expresar ciertas expresiones de manera más compacta, en el lenguaje C# es posible utilizar operadores de expresión compuestos. Estos simplemente permiten expresar una expresión de la forma:

<Variable> = <Variable> Operador <Expresión>;

Cómo: <Variable> Operador= <Expresión>;

Por ejemplo:

La expresión ...	es equivalente a
a = a + 1;	a += 1;
b = b - 5;	b -= 5;
c = c * 3;	c *= 3;
d = d % 2;	d %= 2;
e = e / 5;	e /= 5;

107

Incremento (++) y decremento (--)

Además de los operadores vistos el lenguaje C# incorpora dos operadores para realizar incrementos y decrementos en variables. Estos operadores son **++** y **--**.

Siempre incrementan y/o decrementan en un unidad a la variable a la cual se los aplica.

Tienen como particularidad que pueden ser utilizados como prefijo o postfijo.

Por ejemplo:

```
int a = 4;
int c;
a ++;      /* Incrementa el valor de a en uno equivale a a=a+1 */
++ a;      /* Incrementa el valor de a en uno */
c = a ++; /* Asigna el valor 6 a c, después se incrementa a y queda en 7 */
c = ++ a; /* primero incrementa a (vale 8) y luego asigna el valor 8 a c */
```

Es por esto que en general la estructura PARA suele utilizarse como:

for(int i = 0; i < 10 ; i++)

108

Estructura Repetir – do while

Hasta ahora hemos visto dos estructuras iterativas:

- La estructura MIENTRAS (while) la cuál utilizamos cuando la cantidad de iteraciones no es conocida.
- La estructura PARA (for) la cuál utilizamos cuando la cantidad de iteraciones requeridas en un algoritmo es conocida.

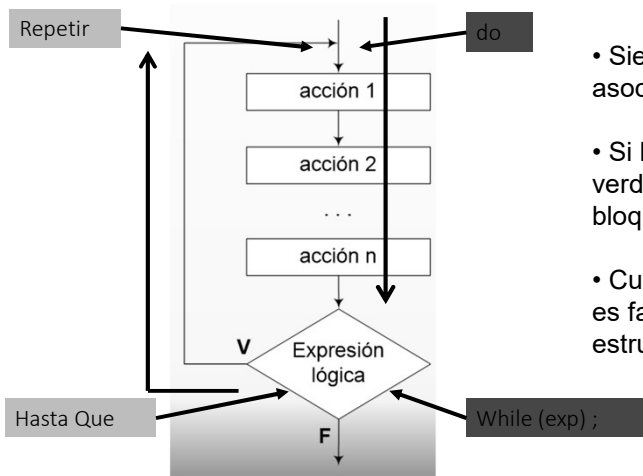
La estructura REPETIR (do - while) es una nueva estructura que se utiliza cuando la cantidad de iteraciones requerida en un algoritmo es desconocida, pero sin embargo, se requiere que se ejecute por lo menos una vez un conjunto de órdenes o acciones. Esta es precisamente la diferencia que guarda con la estructura MIENTRAS.

109

109

Estructura Repetir – do while

La representación de la estructura utilizando diagrama de flujo es:



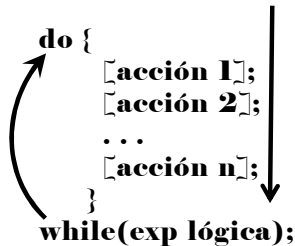
- Siempre se ejecuta el conjunto de acciones asociado a la estructura, al menos la primera vez.
- Si la expresión lógica evaluada al final del bloque es verdadera, se inicia nuevamente la ejecución del bloque de acciones asociadas a la estructura.
- Cuando el resultado de evaluar la expresión lógica es falso, se finalizan las iteraciones, saliendo de la estructura.

110

110

Sintaxis C# - do while

La sintaxis de esta estructura es:



- El fragmento de código sobre el que la estructura se aplica es el delimitado por las llaves asociadas a las palabras claves `do` y `while`.

- En una primera iteración todo el código es ejecutado.

- Finalizada la ejecución de la acción `n`, es evaluada la expresión lógica asociada a la palabra clave `while`.

- Mientras la expresión lógica sea evaluada como verdadera, continuarán las iteraciones comenzando otra vez desde la primer acción después del **do**.

- Cuando la evaluación de la expresión lógica de como resultado **falso**, se finalizarán las iteraciones. Continuando con la siguiente instrucción después del **while(exp);**

- De este comportamiento se desprende que de la misma manera que con la estructura `while`, el conjunto de acciones iterado debe modificar de alguna manera la expresión lógica evaluada.

111

111

Ejemplo Do – While

```

static void Main(string[] args)
{
    int valor = 0;
    do
    {
        Console.WriteLine("Ingrese un valor entre 0 y 999 (negativo finaliza:");
        valor = Convert.ToInt32(Console.ReadLine() );
        if (valor >= 100)
        { Console.WriteLine("Tiene 3 dígitos.");
        }
        else
        { if (valor >= 10)
          { Console.WriteLine("Tiene 2 dígitos.");
          }
          else
          { Console.WriteLine("Tiene 1 dígito.");
          }
        }
    } while (valor > 0);    //Repite si a condición se cumple,
} // Con valor menor que cero sale
  
```

112

112

Ejemplo Do – While

Se requiere una aplicación que calcule el promedio de 3 valores enteros positivos. La aplicación debe validar que los valores ingresados sean positivos y mostrar el valor del promedio calculado en pantalla con dos decimales.

113

113

Ejemplo Do – While

Uso de do
while para
validar un valor
ingresado

```
static void Main(string[] args)
{
    int num;    /* Valor ingresado por el usuario */
    int i;      /* contador de iteraciones */
    int acum;   /* Acumulador */
    double p;   /* promedio calculado */
    acum = 0;
    for (i = 0; i < 3; i++) {
        do
        {
            Console.Write("Ingrese el valor {0}:", i + 1);
            num = Convert.ToInt32(Console.ReadLine());
            if (num < 0)
                Console.WriteLine("Error!\nEl valor debe ser positivo");
        } while (num < 0);
        acum = acum + num;
    }
    p = acum / 3.0;
    Console.WriteLine("El promedio es {0:F2}", p);
    Console.ReadKey();
}
```

114

114

Estructuras Iterativas

Usos recomendados

Mientras (while)

Se recomienda su uso cuando no se conoce o no se puede determinar dentro del algoritmo la cantidad de iteraciones necesarias. Usos de centinelas y banderas.

Para (for)

Se recomienda en los casos que se conoce, o el se puede determinar en el algoritmo, la cantidad de iteraciones necesarias.

Repetir (do while)

Se recomienda su uso cuando no se conoce la cantidad de iteraciones pero se tiene certeza de que al menos una es necesaria.

115

115

Según -

Ejemplo

```
class Program
{
    static void Main(string[] args)
    {
        string opcion;
        opcion = Console.ReadLine();
        switch (opcion)
        {
            case "hola":
            case "HOLA":
                Console.WriteLine("Que tal");
                break;

            case "adios":
            case "ADIOS":
                Console.WriteLine("Chau");
                break;
            default:
                Console.WriteLine("No
reconocido");
                break;
        }
        Console.ReadKey();
    }
}
```

116

116

Estructura SEGÚN

- Esta estructura solo puede realizar comparaciones de igualdad sobre una variable.
- Se pueden utilizar tantos valores de comparación utilizando la palabra clave *case* como se requiera.
- La clausula *default*, utilizada para indicar el conjunto de acciones a ejecutar en caso de no encontrar coincidencia puede o no utilizarse.
- Necesariamente cada uno de los *casos* debe finalizar con la palabra reservada *break*.
- Si se quiere ejecutar el mismo conjunto de órdenes para dos casos de comparación diferente, es posible hacerlo colocando todas las palabras *case* asociadas al mismo bloque ejecutable juntas.

117

117

Estructura SEGÚN - Ejemplo

Desarrolle una aplicación que se comporte como una calculadora permitiendo realizar las operaciones aritméticas básicas. Para esto la aplicación debe permitir al usuario seleccionar la operación a realizar a través de un menú.

La aplicación debe finalizar a través de una de las opciones.

[Ver solución](#)

118

118

Ejemplo Según

```

class Program
{
    static void Main(string[] args)
    {
        double op1,      /* Operadores a utilizar */
        op2;             /* en las diferentes operaciones */
        double res;       /* Resultado de la operación realizada */
        int op;           /* Opción seleccionada por el usuario */
        bool continuar;   /* Bandera utilizada para indicar
                           * cuando la aplicación debe finalizar */

        continuar = true;
        while (continuar)
        {
            Console.WriteLine("Indique la opción deseada:");
            Console.WriteLine("\t1 - Sumar");
            Console.WriteLine("\t2 - Restar");
            Console.WriteLine("\t3 - Multiplicar");
            Console.WriteLine("\t4 - Dividir");
            Console.WriteLine("\t5 - Salir");
            Console.Write(">");
            op = Convert.ToInt32(Console.ReadLine());

```

119

119

Ejemplo Según

```

        switch (op)
        {
            case 1:
                Console.Write("Indique el valor del primer operador:");
                op1 = Convert.ToDouble(Console.ReadLine());
                Console.Write("Indique el valor del segundo operador:");
                op2 = Convert.ToDouble(Console.ReadLine());
                res = op1 + op2;
                Console.WriteLine("El resultado es: {0}", res);
                break;
            case 2:
                Console.Write("Indique el valor del primer operador:");
                op1 = Convert.ToDouble(Console.ReadLine());
                Console.Write("Indique el valor del segundo operador:");
                op2 = Convert.ToDouble(Console.ReadLine());
                res = op1 - op2;
                Console.WriteLine("El resultado es: {0}", res);
                break;

```

120

120

Ejemplo Según

```

case 3:
    Console.Write("Indique el valor del primer operador:");
    op1 = Convert.ToDouble(Console.ReadLine());
    Console.Write("Indique el valor del segundo operador:");
    op2 = Convert.ToDouble(Console.ReadLine());
    res = op1 * op2;
    Console.WriteLine("El resultado es: {0}", res);
    break;
case 4:
    Console.Write("Indique el valor del primer operador:");
    op1 = Convert.ToDouble(Console.ReadLine());
    Console.Write("Indique el valor del segundo operador:");
    op2 = Convert.ToDouble(Console.ReadLine());
    if (op2 != 0)
    { res = op1 / op2;
      Console.WriteLine("El resultado es: {0}", res);
    }
    else
    { Console.WriteLine("Error!\nNo es posible dividir por
cero!!!!");    }
    break;
case 5:

```

121

Ejemplo Según

```

case 5:
    continuar = false;
    break;
default:
    Console.WriteLine("Opción no reconocida");
    break;
    }
    }
    }
    }
    }

```

122

Programación I

Bibliografía

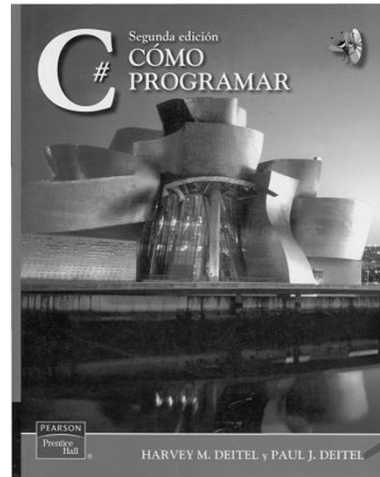
Cómo programar en C#

Segunda Edición

Harvey M. Deitel y Paul J. Deitel

ISBN 970-26-1056-7

Capítulo 6



123

Muchas
Gracias !!!!

124